Dynamic GPU Memory Management for Training Deep Neural Networks

Shriram S B Guide: Anshuj Garg & Prof. Purushottam Kulkarni

Abstract

GPUs are being used to train neural networks. Most widely used machine learning frameworks require the intermediate outputs of all layers of a neural network to fit in the DRAM memory of GPU. GPU memory is a limited resource and hence, this requirement limits the size of the neural networks that can be trained in a given GPU. We study a previously proposed memory manager - vDNN, which virtualizes CPU and GPU memory, hence allowing networks to be trained on the same GPU which couldn't be trained earlier due to lack of GPU memory. However, the training time of a neural network with vDNN increases considerably when availability of GPU memory is less. We propose an improvement in its design, which improves its performance by about 42% to 65% for the evaluated networks(AlexNet, VGG) relative to performance of baseline memory manager used by the current machine learning frameworks. Finally we discuss the issues of memory fragmentation and CPU memory consumption while designing such memory manager.

1 Introduction

Deep neural networks (DNNs) have recently been successfully deployed in various application domains such as computer vision[6], speech recognition[7], and natural language processing[8] thanks to their superior performance compared to traditional state-of-the-art approaches. To facilitate the design and study of DNNs, a large number of machine learning frameworks [1, 3, 2, 4] have been developed in the recent years. GPUs are used for training these neural networks as the operations used in training are data parallel and GPUs have much higher throughput compared to CPUs in this situation[9].

ImageNet LSRVC[5], a popular visual recognition competition which evaluates algorithms for object detection and image classification at large scale has been won by neural networks since 2012. The memory consumption of these networks have increased drastically over time. For example, AlexNet(2012)[6], for a batch size of 128, consumes 1.1 GB which is well below the 12 GB DRAM capacity of current-state-of-the-art NVIDIA Titan X. The more recent VGG-16(2014)[10] consumes 15 GB. With the most recent ImageNet winning network adopting more than hundred convolutional layers[11], the trend in deep learning is moving towards larger and deeper network designs[12, 13, 14, 15]. Current machine learning frameworks follow a network-wide memory allocation policy in which they allocate space required for the whole network on the GPU. Typically, a GPU performs only one layer's computation at any point of time which in turn requires space only for that layer's input and output. So, intermediate outputs(feature maps) remains unused for a large part of training time. This becomes more severe for deep networks in which 53% to 79%[16] of allocated memory remains not being used at all at any given time. With this policy, large networks like VGG require multiple GPUs for training.

To work around this, a runtime memory management solution - vDNN[16] was proposed previously which virtualizes the memory usage of deep neural networks across both GPU and CPU memories. 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. This reduced the GPU memory requirement of vDNN considerably with VGG-16(128) consuming only 6 GB of GPU memory with almost no performance loss[16]. However, when total GPU memory availability decreases, in addition to using slower memory optimal convolution algorithms, time spent in transfer of data between CPU and GPU memory exceeds the time taken for computation. This leads to significant performance loss, with VGG-16(128) facing 61% performance(training time) loss when only 4.8 GB of GPU memory is available[16]. This performance loss occurs due to the synchronization of computation and memory transfer(Section 3.2) at the end of each layer's computation.

We propose an improvement in the design of vDNN which removes the need for this synchronization and improves its performance significantly(Section 5) while consuming the same amount of memory as vDNN. Our work involved the following:

- Study the approach followed by vDNN - Section 3.2
- Re-implement vDNN as it is not available publicly
- Propose and implement a design which improves its performance, and compare it with vDNN on AlexNet and VGG with two different batch sizes - Section 5
- Discuss the issues of CPU memory consumption and memory fragmentation which we expect to work on in the future 6

2 Background

2.1 Neural Network Architecture and Training

Convolutional neural networks are one of the most popular machine learning (ML) algorithms for high accuracy computer vision tasks. While other types of networks are also gaining tractions (e.g., recurrent neural networks for natural language processing), all of these DNNs are trained using a backward propagation algorithm[17] via stochastic gradient-descent (SGD). For clarity of exposition and owing to their state-of-the-art performance in the ImageNet competition, this paper mainly focuses on the feedforward style convolutional neural networks commonly seen in AlexNet[6], GoogLeNet[14], and VGG[10]. However, the key intuitions of our work are equally applicable to any neural network that exhibits layer-wise computational characteristics and is trained via SGD, detailed later in this section.

DNNs are designed using a combination of multiple types of layers, which are broadly categorized as convolutional layers (CONV), activation layers (ACTV), pooling layers (POOL), and fully-connected layers (FC). A neural network is structured as a sequence of multiple instances of these layers. DNNs for computer vision tasks in particular are broadly structured into the following two modules: 1) the feature extraction layers that detect distinguishable features across input images, and 2) the classification layers that analyze the extracted features and classify the image into a given image category. Feature extraction layers are generally designed using CONV/ACTV/POOL layers and are positioned as the initial part of the DNN. The classification layers are built up using the FC layers and are found at the end of the DNN computation sequence. The general trend in deep learning is to design the network with a large number of feature extraction layers so that a deep hierarchy of features are trained for robust image classification.

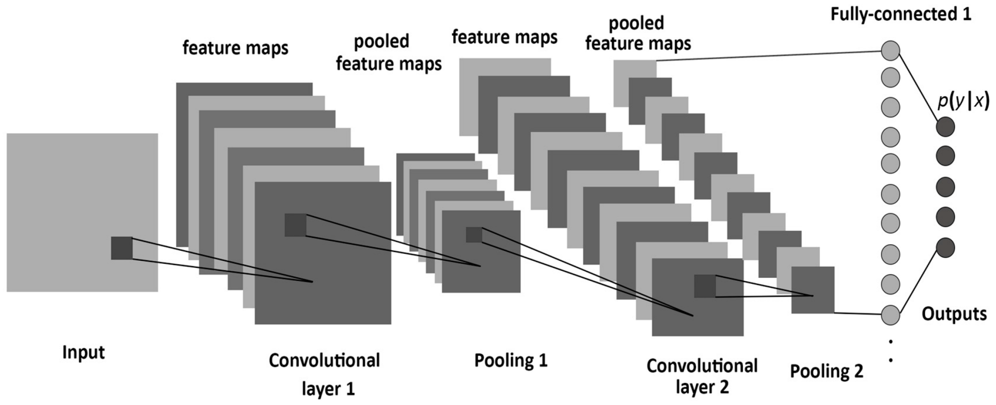


Image Courtesy: [27]

Figure 1: Typical Convolutional Neural Network

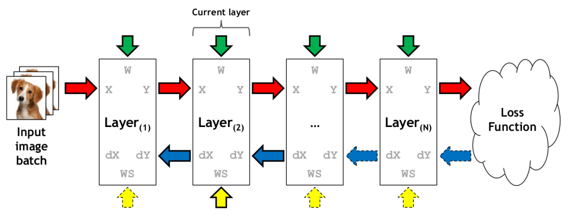


Image Courtesy: [16]

Figure 2: Memory allocations required for linear networks using the baseline memory manager (bold arrows). For inference, the sum of all green (W) and red (X) arrows are allocated. For training, two additional data structures for dX and dY are required: both are sized to the maximum of all blue (dY) arrows and are reused while traversing back the layers during backward propagation. An optional temporary buffer, called workspace in cuDNN [20] (yellow arrow, WS), is needed in certain convolutional algorithms. The workspace buffer is sized with the maximum workspace requirement among all layers and is reused during backward propagation.

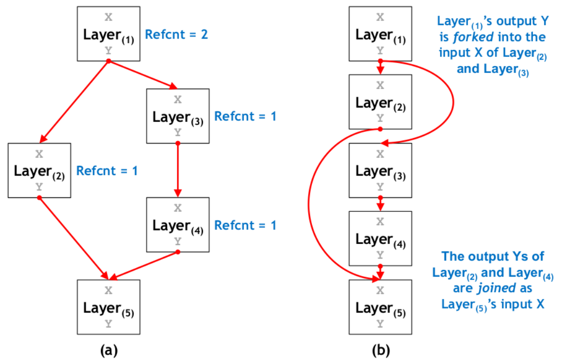

X
Layer(1)
Y
Refcnt = 2
Layer(3)
Refcnt = 1
Layer(2)
Refcnt = 1
Layer(4)
Refcnt = 1
Layer(5)
(a)
Layer(1)'s output Y is forked into the input X of Layer(2) and Layer(3)
The output Ys of Layer(2) and Layer(4) are joined as Layer(5)'s input X
(b)

Image Courtesy: [16]

Figure 3: (a) The computation graph and its inter-layer dependencies of a GoogLeNet-style, non-linear feedforward network during forward propagation. Refcnt refers to the number of consumer layers that depends on the current, producer layer's Y. The order in which the GPU processes each layer's forward computation is shown in (b), from layer (1) to layer (5) , highlighting the layer-wise computation of DNN training. The producer-consumer relationship is reversed during backward propagation.

A neural network needs to be trained before it can be deployed for an inference or classification task. Training entails learning and updating the weights of the layers of a neural network by performing the operations of forward and backward propagation algorithms[17]. The direction of traversal, as well as the mathematical operations that must be performed, differ for forward and backward propagation.

Forward Propagation. Forward propagation is performed from the first (input) layer to the last (output) layer, whereas backward propagation is performed in the opposite direction (last to first layer), from right to left in Figure 2. Intuitively, forward propagation traverses the network layer-wise and performs the aforementioned feature extraction and classification tasks on a given input, leading to an image classification. During forward propagation, each layer applies a mathematical operation to its input feature maps X and stores the results as output feature maps Y. For linear feedforward DNNs, the resulting Y of layer$_{(n-1)}$ is directly used as the input X by layer$_{(n)}$ (Figure 2). The computation flow of forward propagation is therefore a serialized process, as layer(n) can initiate its operation only when the preceding layer$_{(n-1)}$ is finished with its computation and forwarded its output Y to layer$_{(n)}$'s input X. Non-linear network topologies can contain one-to-many (fork) and many-to-one (join) inter-layer dependencies, but forward propagation still involves a series of layer-wise computations as detailed in Figure 3. Note that the GPU can only process a single layer's computation at any given time due to such inter-layer data dependencies. As a result, the minimum, per layer memory allocations required are determined by the layer's input-output relationships and its mathematical function. For instance, a CONV layer using the most memory-efficient convolutional algorithm (e.g., implicit GEMM in cuDNN (ref GEMM in cuDNN)) requires three data structures, the input/output feature maps (X and Y) and the weights of the layer (W) for forward propagation. Employing a fast-fourier-transform (FFT) based convolution algorithm however requires an additional, temporary workspace (WS) buffer to manage transformed maps.

Backward Propagation. For DNNs that are not fully trained, the inferred image category might be incorrect. As a result, a loss function is used to derive the magnitude of the inference error at the end of forward propagation. Specifically, the gradient of the loss function is derived with respect to the last layer$_{(N)}$'s output: $\frac{\partial Loss}{\partial Y_{(N)}}$. This value is forwarded to last layer$_{(N)}$ as its input gradient maps (dY), and the output gradient maps (dX) are derived based on the chain rule [17]: $\frac{\partial Loss}{\partial X_{(N)}} = \frac{\partial Loss}{\partial Y_{(N)}} \frac{\partial Y_{(N)}}{\partial X_{(N)}}$. Deriving the value of dX for layer$_{(N)}$ generally requires memory for both its input/output gradient maps (dY and dX) and also the input/output feature maps (X and Y) for this layer. For linear networks, the calculated dX of layer$_{(N)}$ is directly passed on to the preceding layer$_{(N-1)}$ to be used as dY for layer$_{(N-1)}$'s dX derivation (Figure 2).

This chain rule is similarly used to derive the gradients of the weights to update the network model. Similar to forward propagation, backward propagation is also performed layer-wise to the respective incoming gradient maps, dYs. Once backward propagation reaches the first layer, the weights are adjusted using the weight gradients so that the prediction error is reduced for the next classification task. Hence, training a network involves both forward and backward propagation, which are repeated for millions to billions of iterations. In SGD-based training, the network input is generally batched with hundreds of images (e.g., 128 and 256 images for best performing AlexNet and VGG-16), which increases memory allocation size but helps the network model better converge to an optimal solution.

2.2 GPU Programming Interface

GPU is divided into Streaming Multiprocessors(SM) and each SM contains a specific number of Streaming Processors(SP). The main component of an SP is ALU. GPU follows the SIMT(Single Instruction Multi Thread) model in which there is one control unit per SM which is common to all the SPs inside it.

This work uses CUDA programming interface[18]. A CUDA program is a unified source code encompassing both host and device(GPU) code. A function defined with the keyword "__global__" is compiled to be executed on the device and is called a kernel function. A call to a kernel function takes as arguments Nb - number of blocks and Nt - threads per block to be launched. The call will launch Nb blocks(collectively known as a grid) with each containing Nt threads. Each thread in the grid has a unique id which is the pair (block id, thread id in block). Each block will be scheduled by the device to be executed by a single SM.

CUDA streams allow different kernels to be executed in parallel. Kernels launched in different streams are executed parallely(if sufficient SMs and SPs are available) and ones launched in same stream are executed sequentially. CUDA provides the function cudaMemcpy() to transfer data between host and device. This is synchronous and no kernel can execute while transfer is occurring. It also provides the function cudaMemcpyAsync() which takes the CUDA stream also as argument, thereby allowing transfer to be overlapped with computation. CUDA provides functions cudaMalloc()/cudaFree() for memory allocation/free in device. These are synchronous and allocation/free is managed by the device. We use a software memory manager - CNMEM[19] which first requests a pool of sufficient memory enough for training the network. Then, the functions CNMEMMalloc()/CNMEMFree() are used for allocation/free which are asynchronous w.r.t. to kernel execution and are comparatively much faster than cudaMalloc()/cudaFree(). NVIDIA provides a deep learning library - cuDNN[20] which provides kernels for forward and backward propagation of different kinds of layers like convolution, pooling, fully connected, batchnorm, dropout, etc..

3 Prior work

3.1 Unified CPU/GPU Memory Access

Page migration based virtualization solutions that expose both CPU and GPU memory for page allocations have been proposed previously. This (regardless of whether the virtualization feature is provided by future CUDA runtime extensions or programming models such as OpenMP [21]) must transfer pages via PCIe, which involves several latency-intensive processes such as CPU interrupts for system calls, page-table updates, TLB updates/shootdowns, and the actual page transfer. Pichai et al. [22] and Power et al. [23] proposed TLB designs that leverage the unique memory access patterns of GPUs for optimizing the throughput of memory address translations for unified CPU/GPU memory. Later, Zheng et al. [24] reported that the latency to page-in a single 4 KB page to the GPU is 20 to 50 μs, meaning the PCIe bandwidth utilization using page-migration is 80 to 200 MB/sec, as opposed to the DMA initiated cudaMemcpy that achieves an average 12.8 GB/sec out of the 16 GB/sec maximum PCIe bandwidth. As the amount of data to be paged in/out via PCIe can be of the order of GBs (VGG-16(128) consumes 15 GB) for very deep networks , ML frameworks will suffer from huge performance penalties when relying on page-migration for training DNNs. Rhu et al. [16] therefore proposed an application-level virtual memory management solution(vDNN) specifically tailored for DNNs. The next subsection describes vDNN in detail.

3.2 Virtualized Deep Neural Network - vDNN

3.2.1 Motivation

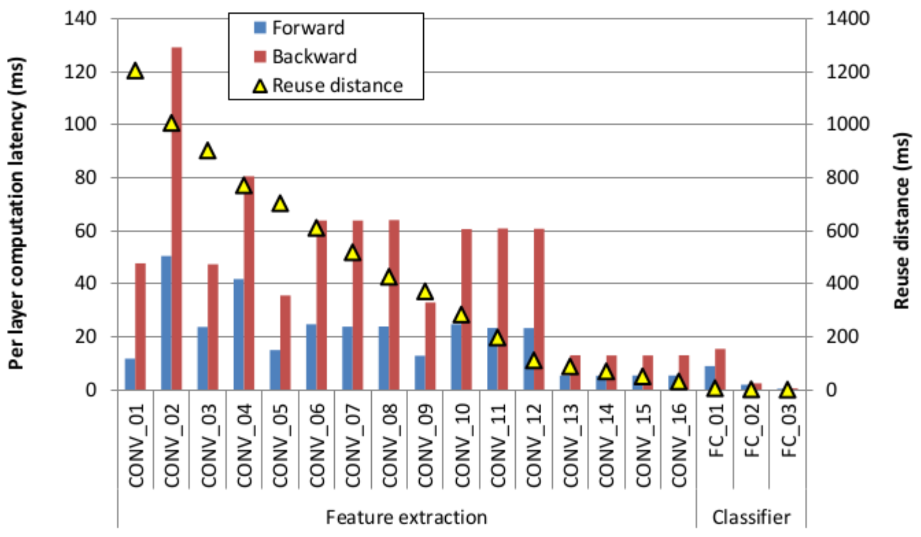


Image Courtesy: [16]

Figure 4: VGG-16's per layer computation latency for forward and backward propagation (left axis). Right axis shows the reuse distance of each layer's input feature maps, X. We define the reuse distance of a layer$_{(n)}$'s X as the latency between the completion of layer$_{(n)}$'s forward propagation and the start of the same layer$_{(n)}$'s backward propagation.

Figure 4 shows the per layer memory usage of VGG-16(64) during forward propagation, which provides the following key observations. First, the intermediate feature maps and workspace (left axis) incur an order of magnitude higher memory usage compared to the weights (right axis) of each layer. Second, most of these intermediate data structures are concentrated on the feature extraction layers and are less significant in the later classifier layers. Third, the weights, while smaller in size compared to these intermediate data, are mostly concentrated on the classifier layers due to their full connectivity. Lastly, the per layer memory usage is much smaller than the 28 GB of memory required by the baseline policy, showing significant opportunities for memory savings with a fine-grained, layer-wise memory management policy.

3.2.2 Core Idea

Memory requirement per individual layer is substantially smaller than what is actually provisioned with the baseline, network-wide memory allocation policy. vDNN adopts a sliding-window based, layer-wise memory management strategy in which the runtime memory manager conservatively allocates memory from its memory pool for the immediate usage of the layer that is currently being processed by the GPU. Intermediate data structures that are not needed by the current layer are targeted for memory release to reduce memory usage. During forward propagation, once a given layer$_{(n)}$'s forward computation is complete, layer$_{(n)}$'s X is not reused until the GPU comes back to the same layer$_{(n)}$'s corresponding backward computation. Because the reuse distance of layer$_{(n)}$'s X is of the order of milliseconds to seconds (e.g., more than 60 ms and 1200 ms for the first layer of AlexNet and VGG-16 (64), respectively), deep networks end up allocating a significant number of Xs that effectively camp inside the GPU memory without immediate usage. vDNN therefore conditionally offloads these intermediate Xs to CPU memory via the system interconnect (e.g., PCIe, NVLINK [25]), overlapping the transfer of a layer's input with its output computation on GPU, if they are targeted for memory release(Section 3.4 details about which layers are chosen for offloading its X). While applying this idea to non-linear networks, X should be released only after forward propagation of all layers which depend on X has been completed.

During backward propagation, after layer$_{(n)}$'s backward propagation, its Y and dY are no longer required because the GPU has already completed the gradient updates for this layer (Figure 2). So, vDNN immediately frees up a layer's Y and dY once this layer's backward computation is complete. If a layer's X has been offloaded to host memory, vDNN should guarantee that the offloaded data is copied back to GPU memory before the gradient update is initiated. Naively copying back the data on-demand will serialize the backward computation behind the memory copying operation of X. vDNN therefore launches a prefetch operation for layer$_{(n)}$'s offloaded feature maps, which is overlapped with layer$_{(m)}$'s backward computation, with $n < m$, so that prefetching is launched before its actual usage, hiding prefetching latency.

3.3 Core Operations and its Design

vDNN is prototyped as a layer on top of cuDNN[20]. vDNN employs two separate CUDA streams[18] to overlap normal DNN computations with the memory allocation, movement, and release operations of vDNN. stream$_{compute}$ is the CUDA stream that interfaces to the cuDNN handle and sequences all the layer's forward and backward computations. stream$_{memory}$ manages the three key components of vDNN; the memory allocation/release, offload, and prefetch.

Memory Allocation/Release. Since allocation/release is frequently done, an open-source software memory manager is used[19]. When the program launches, the vDNN memory manager is allocated with a memory pool that is sized to the physical GPU memory capacity. Whenever vDNN allocates (and releases) data structures, the underlying memory manager will reserve (and free) memory regions from this memory pool. This avoids making calls to cudaMalloc() or cudaFree(), which makes request directly to the device and takes a significant amount of time for allocation/release.

Memory Offload. When a layer is chosen for offloading, vDNN first allocates a pinned host-side memory region using cudaMallocHost(). stream$_{memory}$ then launches a non-blocking memory transfer of this layer's X to the pinned memory via PCIe using cudaMemcpyAsync(), overlapping it with the same layer's forward computation of cuDNN. vDNN synchronizes stream$_{compute}$ and stream$_{memory}$ at the end of each layer's forward computation. This approach guarantees that the offloaded data is safely released from the memory pool before the next layer begins forward computation. vDNN[16] stated that this maximizes the memory saving benefits of offloading.

Memory Prefetch. Prefetching the offloaded Xs back to GPU memory is also implemented using cudaMemcpyAsync() to overlap data transfers with the computations of backward propagation, with layer$_{(n)}$'s prefetch overlapped with layer$_{(m)}$'s computation($n < m$). Prefetching data too early in time will suboptimally utilize GPU memory as the prefetched data will once again camp inside the GPU memory without immediate usage. vDNN prefetch algorithm is carefully designed to balance the memory saving benefits of offloading with the timeliness of prefetching. Computation of CONV layer is much higher compared to other layer types. So, for finding layer to prefetch, vDNN looks only till previous CONV layer. Prefetch is done only if some layer till that CONV layer requires prefetch. Any layer's prefetch beyond that CONV layer could be overlapped with that CONV layer's computation and prefetched in much less time compared to computation of CONV layer(Figure 6). vDNN synchronizes stream$_{compute}$ and stream$_{memory}$ so that the next layer's backward computation is stalled until the prefetch operation is finalized. Consequently, any prefetch operation launched during layer$_{(n)}$'s backward computation is guaranteed to be ready before layer$_{(n-1)}$'s computation.

3.4 Memory transfer policy

Determining the best layers to offload their feature maps is a multi-dimensional optimization problem that must consider: 1) GPU memory capacity, 2) the convolutional algorithms used and the overall layer-wise memory usage, and 3) the network-wide performance. The first two factors determine whether we are able to train the network at all (which we refer to as trainability of a network), while the last factor decides overall training productivity. Performance loss primarily comes from: 1) the additional latency possibly incurred due

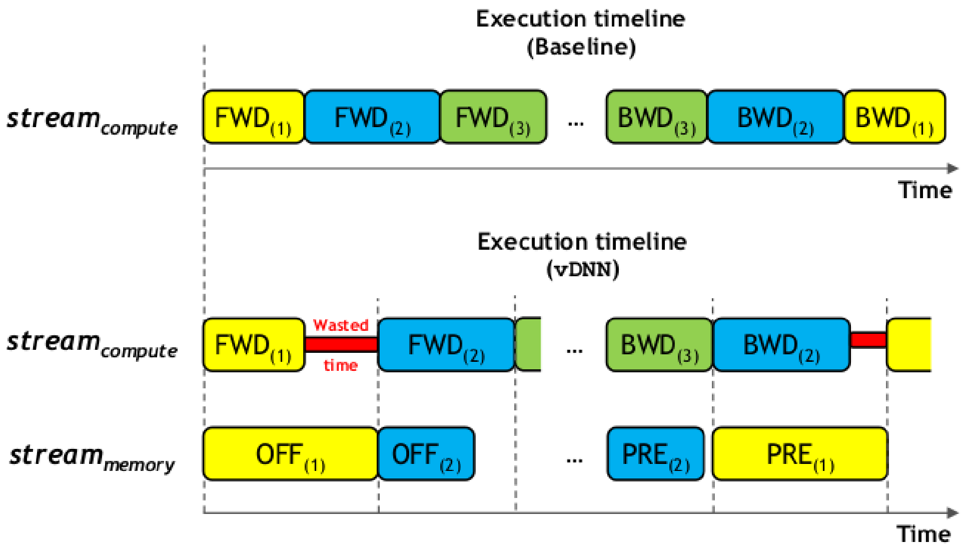
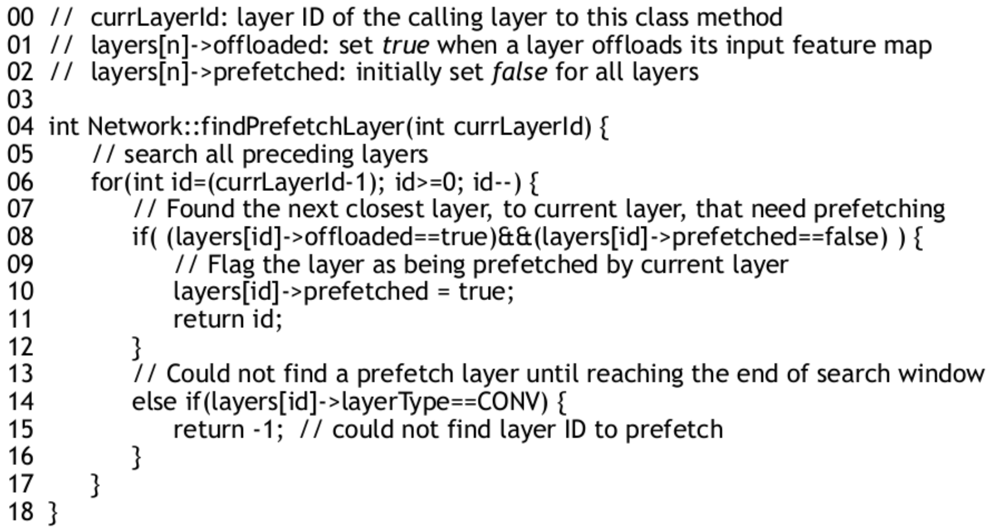


Image Courtesy: [16]

Figure 5: Performance effect of offload and prefetch. FWD (n) and BWD (n) are the forward and backward computations for layer (n) , respectively. OFF (n) is the offloading of layer (n) 's X and PRE (n) is the corresponding prefetch operation for layer (n) .



```
00 //  currLayerId: layer ID of the calling layer to this class method
01 //  layers[n]->offloaded: set true when a layer offloads its input feature map
02 //  layers[n]->prefetched: initially set false for all layers
03
04 int Network::findPrefetchLayer(int currLayerId) {
05       // search all preceding layers
06       for(int id=(currLayerId-1); id>=0; id--) {
07             // Found the next closest layer, to current layer, that need prefetching
08             if( (layers[id]->offloaded==true)&&(layers[id]->prefetched==false) ) {
09                   // Flag the layer as being prefetched by current layer
10                   layers[id]->prefetched = true;
11                   return id;
12             }
13             // Could not find a prefetch layer until reaching the end of search window
14             else if(layers[id]->layerType==CONV) {
15                   return -1;  // could not find layer ID to prefetch
16             }
17       }
18 }
```

Image Courtesy: [16]

Figure 6: Pseudo code explaining how vDNN finds layers to prefetch. Notice how the search operation is only up to the next closest CONV layer (line 14), guaranteeing that the prefetched X will not end up being used too far away in the future as it restricts the prefetch layer to be within the search window of layers - From [16]

to offload/prefetch, and 2) the performance difference between different convolution algorithms. Selecting the optimal hyperparameters which fit the network inside GPU while maximizing performance is non-trivial. vDNN adopts the following heuristic-based memory transfer policies that narrow the parameter choices and simplify the optimization problem, while still performing robustly in practice.

Static vDNN. Two memory transfer options - vDNN$_{all}$ and vDNN$_{conv}$ are used. vDNN$_{all}$ chooses to offload all Xs from GPU which drastically reduces device memory usage. vDNN$_{conv}$ chooses to offload only

CONV layers. vDNN$_{conv}$ is based on the observation that CONV layers have much longer computation latency than other layers, being more likely to effectively hide the latency of offload/prefetch. vDNN$_{conv}$ performs much better than vDNN$_{all}$ as the latter incurs latency overheads of offload/prefetch for non-CONV layers most of the time.

Dynamic vDNN. Static vDNN doesn't take into account the system specifications like maximum compute FLOPs and memory bandwidth, memory size, etc. For DNNs that completely fit in GPU memory, the best approach is to have all the memory allocations resident in GPU without any offloading and employ fastest possible CONV algorithm. Large and deep networks might not completely fit in memory and may require offload of some layers. Dynamic vDNN automatically determines the layers to be offloaded and the convolutional algorithms to be employed, at runtime, to balance the trainability and performance of a DNN. It uses the runtime API provided by NVIDIA's cuDNN that experiments with all available convolution algorithms for a given layer and evaluates each algorithm's performance and its memory usage. This is used to run a base profiling stage. It takes around tens of seconds which is negligible compared to time taken for millions of iterations of forward-backward propagation. Once the base profiling is completed, it employs the following additional profiling passes:

1. First, the static vDNN$_{all}$ is tested for a single training pass with all CONV layers using the memory-optimal, no-WS incurred algorithm(Figure 2). This initial pass determines if the target DNN can be trained at all as this requires the least GPU memory.

2. If vDNN$_{all}$ passed, another training phase is launched with all CONV layers employing the fastest algorithms but without any offloading. Such a configuration, if it passes successfully, will be adopted for the rest of the full training procedure as it provides the highest performance while guaranteeing trainability. If this profiling phase fails due to memory oversubscription, two additional training passes are tested with the same fastest algorithms, but with vDNN offloading enabled for both vDNN$_{conv}$ and vDNN$_{all}$ respectively. If successful, vDNN employs the succeeded configuration for the rest of training. If both vDNN$_{conv}$ and vDNN$_{all}$ fails, we move on to the next profiling pass below to further reduce memory usage.

3. The last phase is based on a greedy algorithm that tries to locally reduce a layer's memory usage, seeking a global optimum state in terms of trainability and performance. When traversing through each layer, vDNN first calculates whether using the fastest algorithm will overflow the GPU memory. If so, then the current layer's CONV algorithm will be locally downgraded into a less performant but more memory-efficient one, until it reaches the memory-optimal implicit GEMM. This greedy-based approach first tries vDNN$_{conv}$ with each CONV layer initially using its own performance-optimal algorithm. If vDNN$_{conv}$ fails, then another training pass is launched with the more memory-efficient vDNN$_{all}$. If vDNN$_{all}$ fails with this greedy algorithm, it tries vDNN$_{conv}$ with memory-optimal CONV algorithm for all layers. If this also fails, then vDNN resorts back to the very first vDNN$_{all}$ solution, with the memory-optimal, no-WS algorithms applied across the entire network.

vDNN$_{all}$ reduces the memory consumption by up to 73%[16], drastically improving the trainability of a network.

4 Improved Design for Asynchronous Memory Transfer

4.1 Issues with vDNN's Design

In the case where vDNN$_{all}$ with performance optimal algorithms is chosen by vDNN due to limited GPU memory, significant performance reduction is observed. Performance loss of about 21% for AlexNet(128) and about 19% for VGG-16(128) is observed(Figure 7). The primary cause of this performance loss is the synchronization of stream$_{compute}$ and stream$_{memory}$ at the end of each layer(Figure 5). CONV with 1x1 filter and POOL layer's computation are much faster compared to the transfer of its input. Hence, vDNN gets stalled for a long time before the start of next layer's computation. CONV algorithms with larger filter size are also becoming faster with every new release of cuDNN and faster GPUs. Currently, for high end GPUs, some of the CONV layer's computation is faster than transfer. In coming years, with faster GPUs

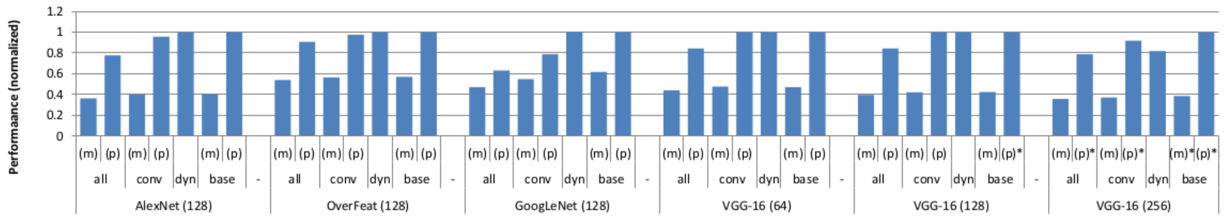


Figure 7: vDNN: Overall performance (normalized to baseline) From [16]. Experiments done with NVidia Titan X

and improvement in algorithms, transfer time will be the bottleneck. Avoiding synchronization improves performance while consuming available memory to the maximum possible extent. We propose a solution which removes the synchronization while having peak memory consumption same as that of vDNN.

4.2 Motivation and Core Idea

For most of the networks(AlexNet[6], VGG[10], GoogleNet[14]), layers whose computation could be done faster(CONV with 1x1 filter, POOL) are surrounded by layers whose computation requires a significant amount of time. Forward propagation requires about only half the memory consumed by backward propagation as in the latter case, space for both X, Y and dX, dY for a layer has to be allocated.

Forward propagation. During forward propagation, layer$_{(n+1)}$'s computation is started as soon as layer$_{(n)}$'s computation is completed irrespective of whether offload of layer$_{(n)}$ layer has been completed or not. If offload of layer$_{(n)}$ hasn't been completed when computation is done, then most likely, one of the subsequent layers, say m(most likely close to n) will take significant time for its computation. Hence, by the time layer$_{(m)}$'s computation is done, all previous offload requests would have completed. Starting the computation of layer$_{(n+1)}$ before offload of layer$_{(n)}$ is complete requires memory for Y of layer$_{(n+1)}$ to be allocated before X of layer$_{(n)}$ is freed. This has the potential of increasing the peak memory consumption. But in most of the cases, this doesn't happen, as $m - n$ is small and backward pass requires significant memory compared to forward pass. In the rare event that memory request is made which exceeds the desired peak memory(equal to that of vDNN), computation is stalled till offload of some layers is complete. Release of offloaded layer$_{(n)}$'s X should be 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, space for layer$_{(n)}$'s 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. This doesn't require any extra memory as in this case, we free Y and dY of layer$_{(m)}$ and allocate dX for layer$_{(m-1)}$ which is same sequence of allocation/free even if computation of layer$_{(m-1)}$ is stalled till prefetch of layer$_{(n)}$. However, computation has to be stalled if prefetch of layer to be computed hasn't been completed yet. This doesn't improve performance if vDNN$_{all}$ is chosen as computation of layer$_{(m-1)}$ has to be stalled till prefetch of layer$_{(m-1)}$'s X, which is overlapped with layer$_{(m)}$'s computation. But, this slightly improves performance of vDNN$_{conv}$. This is because CONV layers are interspersed with POOL layers. If layer$_{(n+1)}$, layer$_{(n-1)}$ are CONV and layer$_{(n)}$ is POOL, layer$_{(n+1)}$ and layer$_{(n)}$'s computation happen overlapped with layer$_{(n-1)}$'s prefetch in our design. On the other hand, in vDNN, layer$_{(n+1)}$'s computation is only overlapped with layer$_{(n-1)}$'s prefetch and layer$_{(n)}$'s computation happening after completion of prefetch.

4.3 Modifications to vDNN's Design

We call our design vDNNExt. Our design remains almost same as vDNN except for a few minor changes to avoid synchronizing the two streams at the end of each layer.

Memory Offload. If a layer(say n) has been chosen for offloading, then, just after issuing start of computation of this layer in stream$_{compute}$(non-blocking), a pinned host-side memory region is allocated

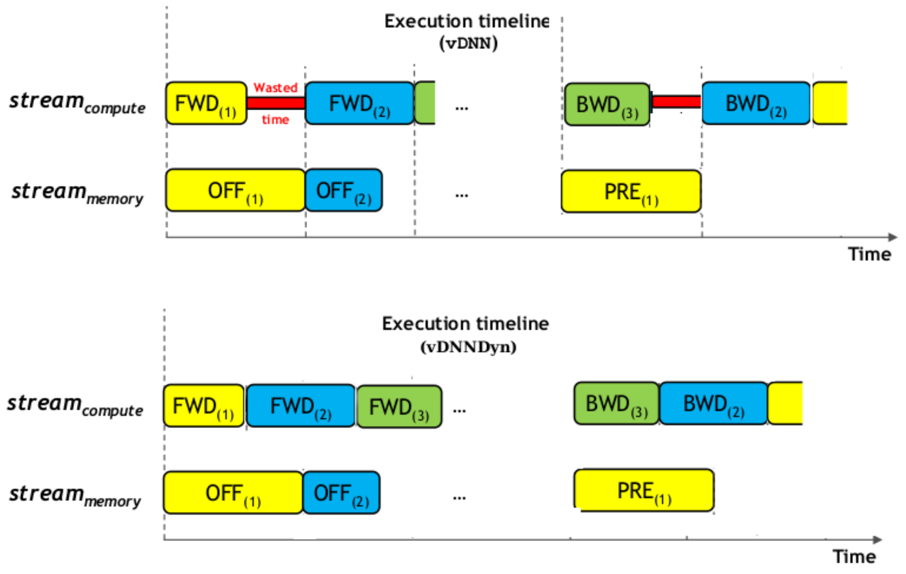


Figure 8: Comparison between synchronous and asynchronous memory transfer. When sufficient memory is available, vDNNExt starts the computation of next layer without waiting for offload of previous layers to complete.

using cudaMallocHost(). In stream$_{memory}$, non-blocking memory transfer of this layer's X to the pinned memory is issued via cudaMemcpyAsync(). This is followed by launch of a cudaEvent(event_offload_done[n]) in stream$_{memory}$ which helps to find out if the offload has been completed. After completion of computation of layer$_{(n)}$(ensured by calling cudaStreamSynchronize(stream$_{compute}$)), a separate thread is launched which waits on the cudaEvent launched previously (event_offload_done[n]) and releases layer$_{(n)}$'s X. 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 repeat this for subsequent layers. 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 are offloaded and released before start of backward propagation. Also, call to cnmemMalloc() has to wait if serving the request exceeds the desired memory consumption. Conditional statement is used with cnmemMalloc and cnmemFree. During a call to cnmemMalloc(), if memory is not available, the call is blocked using this conditional statement. Every time cnmemFree is called, it broadcasts on this conditional statement.

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

5 Experimental Evaluation

5.1 Experimental Setup

5.1.1 Memory Manager

We implemented vDNN as described in Section 3.2 and further added the proposed improvement in its design(Section 4). Our implementation uses the latest version of cuDNN 7.1[20] which serves as the GPU backend. All the layers that constitute a DNN's feature extraction layer have been implemented using cuDNN, and the execution of each layer is orchestrated using two CUDA streams, stream$_{compute}$ and stream$_{memory}$ as discussed previously. The classification layer remains unchanged and uses cuBLAS library provided by CUDA toolkit. Our implementation currently supports only linear neural networks.

5.1.2 System Specifications

We conducted experiments on NVIDIA's Geforce GTX 970, which provides single precision throughput of 3.5 TFLOPS, memory bandwidth of max 224 GB/sec, and memory capacity of 3.5 GB. The GPU communicates with an Intel i7-3770 (containing 32 GB of DDR3 memory) via a PCIe switch (gen3) which provides a maximum of 16 GB/s data transfer bandwidth. Its performance is much less compared to current-state-of-the-art NVIDIA's Titan X which has single precision throughput of 7 TFLOPS, maximum memory bandwidth of 336 GB/s. If these experiments were performed in Titan X, a large difference in performance of vDNN and base might be observable as memory transfer rate is much less compared to computation throughput. In the case of GTX 970, no performance difference is observed between vDNN and vDNNExt with vDNN$_{conv}$ as all CONV algorithms use filter size > 1 and are much slower compared to transfer time of its input. However, significant performance difference is observable in the case of vDNN$_{all}$ for both memory and performance optimal CONV algorithms. POOL layer's computation time is much less compared to transfer time of its input which leads to this observation.

5.1.3 DNNs used for Evaluation

We evaluate the following state-of-the-art linear ImageNet winning DNNs.

- AlexNet with batch sizes 128 and 512
- VGG-16 with batch sizes 16 and 42

In case of AlexNet, AlexNet(512) doesn't fit in memory for baseline memory manager using performance optimal algorithm. In case of VGG-16, VGG-16(42) doesn't fit in memory for baseline memory manager. We show that vDNN and vDNNExt enable the training of AlexNet(512) and VGG-16(42) with the same GPU space and vDNNExt is performs much better than vDNN in some cases.

The static vDNN$_{all}$ and vDNN$_{conv}$ policies are denoted as *all* and *conv* in all the figures discussed in Section 5.2 and 5.3. These are each evaluated with both memory-optimal and performance-optimal (denoted as (m) and (p)) convolutional algorithms across the network. The baseline memory manager (base) is similarly evaluated with both memory-optimal and performance-optimal algorithms. The algorithms are dynamically chosen for vDNN$_{dyn}$ (denoted as *dyn*) as discussed in Section 3.4.

5.2 GPU Memory Usage

This section evaluates the effect of vDNN on GPU memory usage.

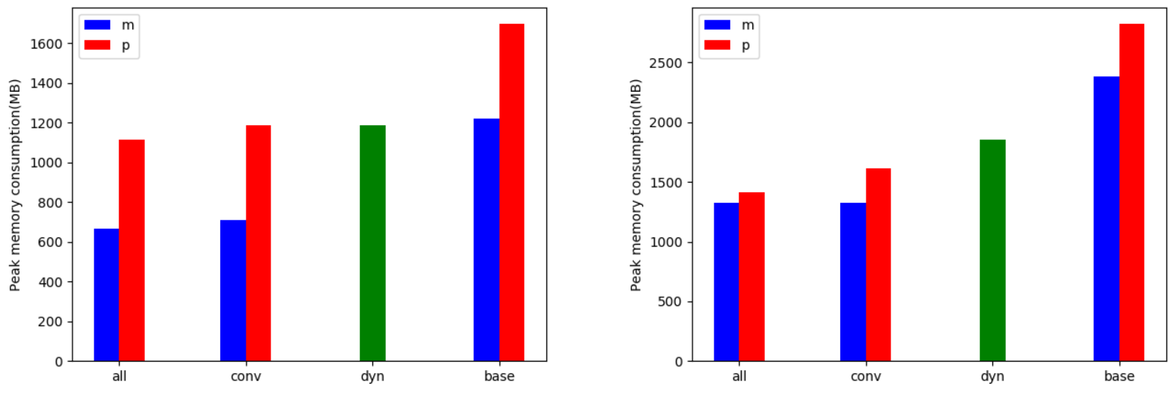


Figure 9: GPU Memory consumption of AlexNet(128)(left) and VGG-16(16)(right)

Table 1: AlexNet(512)

	Total GPU space(MB)
base(m)	3443
dyn	2113
conv(m)	2113
all(m)	1947

Table 2: VGG-16(42)

	Total GPU space(MB)
dyn	2617
conv(m)	2612
all(m)	2612

vDNN enhances the trainability of a network by significantly reducing its memory requirements. vDNN$_{all}$ with memory optimal CONV algorithm consumes the least amount of memory saving about 53-60% of GPU memory. When performance optimal algorithms are employed, memory saving reduces to being in the range 34-50%(Figure 9). Memory consumed by vDNNExt is same as that of vDNN in all the cases(conv/all offload and memory/performance optimal algorithms). vDNN also enables training of networks when it cannot be trained with base memory manager(Table 1 and 2).

5.3 Performance of vDNN and vDNNExt

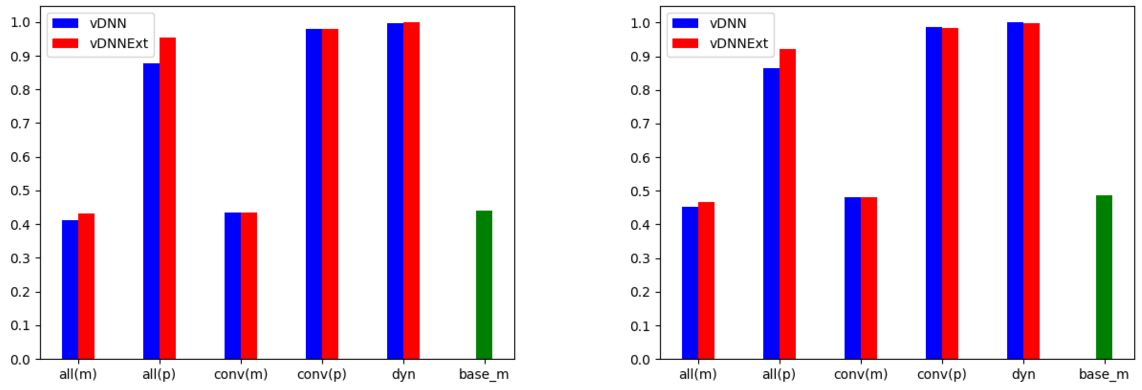


Figure 10: Performance of AlexNet(128)(left) and VGG-16(16)(right) normalized to baseline memory manager which uses performance optimal CONV algorithm

Table 3: Average time for one iteration(ms) of AlexNet(512)(left) and VGG-16(42)(right)

	vDNNExt	**vDNN**
dyn	1475	1482
conv(m)	1475	1482
all(m)	1497	1553

	vDNNExt	**vDNN**
dyn	864	864
conv(m)	1722	1722
all(m)	1777	1835

vDNNExt improves the performance of vDNN$_{all}$ considerably(Figure 10). When performance optimal algorithm is used, it gives a relative performance improvement of 46%-65% over vDNN. Here, relative is in the sense that performance of baseline memory manager which uses performance optimal CONV algorithm is maximum achievable and relative performance improvement is the decrease in difference in training time compared to initial difference. When memory optimal algorithm is used, the relative improvement is about 42%-64%, here performance of baseline memory manager using memory optimal CONV algorithm being the maximum achievable. In terms of absolute performance, for AlexNet, vDNNExt's performance is 95% of baseline(CONV performance optimal) compared to 87% performance of vDNN when using vDNN$_{all}$ with performance optimal CONV algorithm. When using memory optimal algorithm, vDNNExt's performance is 98% of baseline(CONV memory optimal) compared to 94% performance of vDNN. Similar results are observed for VGG-16(16). Overheads due to threads and maintaining locks leads to a slight reduction in performance of vDNNExt compared to vDNN when none of the layer's transfer takes more time than computation, as observed in VGG-16 with vDNN$_{dyn}$(Figure 10).

In the case of AlexNet(512), base(m) takes about 1473ms for one iteration. In this case also, improvement in vDNN$_{all}$ by vDNNExt is significant. vDNN$_{dyn}$ is sometimes able to perform much better by consuming very little extra memory as observed in VGG-16(42).

If these experiments were conducted in NVIDIA's Titan X, then difference in performance of vDNNExt and vDNN will most likely increase as transfer becomes the bottleneck in that case.

6 Discussions for Future Work

6.1 Memory Fragmentation

Table 4: Expected and Actual GPU memory consumed in AlexNet(128)

	Expected(MB)	**Actual(MB)**
dyn	1188	1625
conv(p)	1188	1666
conv(m)	710	943
all(p)	1113	1591
all(p)	667	809

Table 5: Expected and Actual GPU memory consumed in VGG-16(16)

	Expected(MB)	**Actual(MB)**
dyn	1853	2445
conv(p)	1610	2053
conv(m)	1323	1715
all(p)	1414	1857
conv(m)	1323	1519

One observation made while experimenting was that of external memory fragmentation. The reported memory consumption are the ones which the network should have consumed ideally. Even though the sequence of memory allocation/free is the same for every iteration, this sequence is not known to the memory manager(CNMEM). Hence, it uses some heuristics to best reduce fragmentation. It still consumes about 300-600 MB more than what it should have consumed ideally(Tables 4 and 5) . We show that there are sequences of memory allocation/free such that even if this sequence is known to the memory prior to the start of training, fragmentation is unavoidable. We provide an example of such sequence below. Note that any call to malloc should return a contiguous pool of memory of requested size.

$$a_{11}, a_{12}, a_{13}, a_{14}, d_1, a_{22}, a_{23}, a_{24}, d_2, a_{33}, a_{34}, d_3, a_{44}, d_4 \text{ a - allocation, d - free}$$

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. Also, we ensure that just before d_j, the total memory request sums up to the total pool size of memory manager, for every j. The problem is to provide start addresses for each of the allocation request such that no two requests' memory overlap at any point of time. For any j, just before d_j is served, if a_{ij}, for all $i <= j$ are not occupying contiguous memory locations, then after d_j is served, memory will be fragmented. At the end of block (i + 1), the requested size is exactly equal to the pool size of memory manager. With fragmentation happening in block i, it won't be possible to allocated space for block (i + 1) in this pool. It can be easily seen that this happens for some i = 1, 2, 3 for this sequence.

6.2 CPU memory consumption

For deep networks, significant amount of CPU memory is required as all offloaded layers have to be stored in CPU memory. The feature maps which are offloaded are output of ACTV layers. Sigmoid outputs lie in (0, 1) and ReLU outputs lie in $[0, \infty)$, which has a huge scope for compression. Previous work[26] have proposed architectural change in GPU which compresses the data which it sends, on the fly and decompresses the data which it receives. This reduces both the CPU memory consumed and the amount of data to transfer, hence improving the performance of vDNN as transfer is no longer the bottleneck. Implementing it on hardware is a challenge and it hasn't been done yet. We discuss an approach to this problem which could be implemented without extra hardware support.

Moving the feature maps to hard disk is infeasible as read/write time is of the order of milliseconds which is same as the time taken for one iteration training. But these layers could be compressed by the host in a separate thread as soon as it receives it and kept in memory. However, a feature map has to be decompressed before vDNN makes a prefetch request for that layer. Otherwise, prefetch will stall which in turn will stall the computation. findPrefetchLayer()(Figure 6) could be modified to return two indices in which one is the index of layer to prefetch and another is index of layer for which decompression should be started. Since, compression is much faster than transfer, before the next prefetch request, decompression would have completed, hence preventing stalling of computation.

7 Conclusion and Future Work

Popular machine learning frameworks[1, 2, 3, 4] require users to carefully manage their GPU memory usage so that the network-wide memory requirements fit within the physical GPU memory size. We described a previously proposed solution - vDNN which virtualizes the memory usage of the network across both CPU and GPU memories. However, when availability of GPU memory decreases, vDNN faces considerable performance loss. We looked at the cause of this performance loss and proposed a modification in its design which improves its relative performance by about 42%-65% for AlexNet and VGG. We also looked at the issues of memory fragmentation and CPU memory consumption which arise when designing such memory manager and proposed a host-side compression solution to reduce CPU memory consumption.

As a part of future work, we would like to extend the implementation to support non-linear neural networks. Then, we would like to implement host-side compression proposed in Section 6.2. Another improvement would be to utilize the known sequence of allocation/release to come up with better heuristics to reduce external memory fragmentation.

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

[6] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in Proceedings of the Advances in Neural Information Processing Systems, 2012.

[7] 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.

[8] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural Language Processing (Almost) From Scratch," in arxiv.org, 2011.

[9] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and Scalability of GPU-Based Convolutional Neural Networks," 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 2010.

[10] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in arxiv.org, 2015.

[12] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in Proceedings of the International Conference on Learning Representations, 2015.

[13] Wired (www.wired.com), "Microsoft Neural Net Shows Deep Learning Can Get Way Deeper," 2016.

[14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," in arxiv.org, 2014

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

[16] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., & Keckler, S. W. (2016). VDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). doi:10.1109/micro.2016.7783721

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

[18] NVIDIA, "NVIDIA CUDA Programming Guide," 2016.

[19] NVIDIA, "https://github.com/NVIDIA/cnmem," 2016.

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

[21] OpenMP Architecture Review Board, "OpenMP Application Program Interface (version 4.0)," 2013.

[22] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2014.

[23] J. Power, M. Hill, and D. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in Proceedings of IEEE International Symposium on High-Performance Computer Architecture, 2014.

[24] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, “Toward High-Performance Paged-Memory for GPUs,” in Proceedings of IEEE International Symposium on High-Performance Computer Architecture, 2016.

[25] NVIDIA, ”NVIDIA NVLINK High-Speed Interconnect,” 2016.

[26] M. Rhu, M. Oconnor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, “Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks,” 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018.

[27] “A Framework for Designing the Architectures of Deep Convolutional Neural Networks,” Entropy, vol. 19, no. 6, p. 242, 2017.