vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design

Minsoo Rhu Natalia Gimelshein Jason Clemons Arslan Zulfiqar Stephen W. Keckler
NVIDIA
Santa Clara, CA 95050
{mrhu, ngimelshein, jclemons, azulfiqar, skeckler}@nvidia.com

***Abstract*—The most widely used machine learning frameworks require users to carefully tune their memory usage so that the deep neural network (DNN) fits into the DRAM capacity of a GPU. This restriction hampers a researcher's flexibility to study different machine learning algorithms, forcing them to either use a less desirable network architecture or parallelize the processing across multiple GPUs. We propose a runtime memory manager that virtualizes the memory usage of DNNs such that both GPU and CPU memory can simultaneously be utilized for training larger DNNs. Our virtualized DNN (vDNN) reduces the average GPU memory usage of AlexNet by up to 89%, OverFeat by 91%, and GoogLeNet by 95%, a significant reduction in memory requirements of DNNs. Similar experiments on VGG-16, one of the deepest and memory hungry DNNs to date, demonstrate the memory-efficiency of our proposal. vDNN enables VGG-16 with batch size 256 (requiring 28 GB of memory) to be trained on a single NVIDIA Titan X GPU card containing 12 GB of memory, with 18% performance loss compared to a hypothetical, oracular GPU with enough memory to hold the entire DNN.**

I. Introduction

Deep neural networks (DNNs) have recently been successfully deployed in various application domains such as computer vision [1], speech recognition [2], and natural language processing [3] thanks to their superior performance compared to traditional state-of-the-art approaches. Such proliferation of deep learning techniques has led several software frameworks to be developed in recent years to analyze and facilitate the design of neural networks [4, 5, 6, 7]. The list of available frameworks continue to expand with developers constantly adding more features and improving computational efficiency to foster research in the area of deep learning. Due to the tremendous compute horsepower offered by graphics processing units (GPUs), these frameworks provide strong backend support for GPU software libraries such as cuDNN [8]. In fact, almost every group today involved in training neural networks is deploying GPUs for accelerated deep learning [9].

While these popular machine learning (ML) frameworks facilitate the study of DNNs, a major limitation of the use of these frameworks is that the DRAM capacity limits of the GPU(s) in the system eventually limit the size the of the DNN that can be trained (Section II-C). To work around the memory capacity bottleneck [10, 11], ML practitioners must either use less desirable DNN architectures (e.g., smaller number of layers, smaller batch sizes, less performant but more memory-efficient convolutional algorithms) or parallelize the DNN across multiple GPUs [12]. Figure 1 highlights how the memory consumption trends of the ImageNet [13] winning DNNs have evolved over time. AlexNet [1], for instance, only contained 5 convolutional layers with 2 fully-connected layers and required a "mere" 1.1 GB of memory allocation for training, which is well below the 12 GB memory capacity of the state-of-the-art NVIDIA Titan X. The more recent VGG-16 [14], on the other hand, contains 16 convolutional layers and 3 fully-connected layers, incurring a total of 28 GB of memory usage for batch size 256. Because a single GPU can only accommodate a batch size of 64 for VGG-16, training with batch 256 requires parallelization across multiple GPUs or the network must be sequentially executed multiple times with smaller batches. With the most recent ImageNet winning network adopting more than a hundred convolutional layers [15], the trend in deep learning is to move towards larger and deeper network designs [14, 16, 17, 18]. As a result, alleviating the rigid physical memory limitations of GPUs is becoming increasingly important.

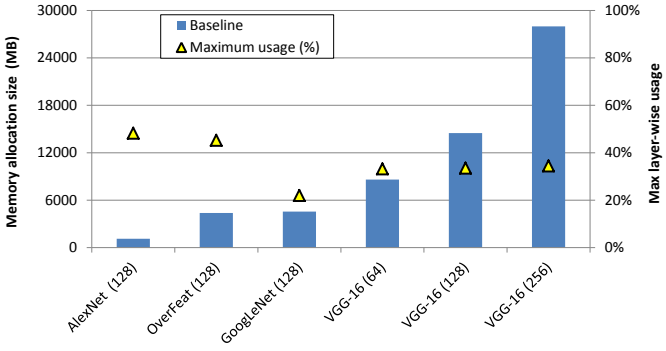

Fig. 1: GPU memory usage when using the baseline, network-wide allocation policy (left axis). The right axis shows the maximum fraction of this baseline allocation actually utilized when traversing through the network layer-wise. The numbers next to the names of each network refer to the batch size throughout this paper. Studied DNNs are detailed in Section IV-C.

In this paper, we propose *virtualized Deep Neural Network* (vDNN), a runtime memory management solution that virtualizes the memory usage of deep neural networks across both GPU and CPU memories. Our vDNN allows ML practitioners to deploy larger and deeper networks beyond the physical capacity of available GPUs, enabling them to focus more on their algorithms while the system architecture and run-

978-1-5090-3508-3/16/$31.00

time system transparently manage the allocation, placement, movement, and release of their data. The motivation behind vDNN is based on the following three key observations: 1) DNNs trained via stochastic gradient-descent (SGD) are designed and structured with multiple layers [19]; 2) the training of these neural networks involves a series of *layer-wise* computations, the order of which is statically fixed and repeated for millions to billions of iterations throughout the entire training process; and 3) even though the GPU can, at any given time, only process a single layer's computation (due to the layer-wise computational characteristics of SGD-based DNN training), popular ML frameworks adopt a *network-wide* memory allocation policy because DNN training requires the intermediate feature maps of all the layers in the network to be backed up in GPU memory for gradient updates (Section II-C). In other words, existing memory management schemes *overprovision* the memory allocations to accommodate the usage of the *entire* network layers, even though the GPU is only using a subset of this allocation for the layer-wise requirements. We observe that such memory underutilization issue becomes more severe for deeper networks, leading to 53% to 79% of allocated memory not being used at all at any given time (Figure 1). The goal of vDNN is to *conservatively* allocate GPU memory for the *immediate* usage of a given layer's computation so that the maximum and average memory usage is drastically reduced, allowing researchers to train larger networks. To achieve this goal, vDNN exploits the data dependencies of allocated data structures, particularly the intermediate feature maps that account for the majority of memory usage (Section II-C), and either releases or moves these intermediate data between GPU and CPU memory. Specifically, vDNN either 1) aggressively *releases* these feature maps from the GPU memory if no further reuse exists, or 2) *offloads* (and later *prefetches*) to (from) CPU memory if further reuse does exist but is not immediately required. By exploiting the inter-layer memory access and reuse patterns of DNNs, our vDNN memory manager intelligently overlaps the normal DNN computations with the offload/prefetch/release operations, effectively virtualizing the memory usage of DNNs with little to no performance loss. The operations of vDNN are completely transparent to programmers and enable them to train larger and deeper neural networks that consume memory well beyond the limits of physical memory of GPUs today. The key contributions of our work are:

- This work is the first to present a detailed, quantitative analysis on GPU-based DNN *training*, as opposed to recent literature targeting energy-efficient accelerators for DNN *inference* [20, 21, 22, 23, 24, 25, 26, 27, 28, 29].
- To the best of our knowledge, our work is the first that provides an in-depth characterization study on the memory access characteristics of DNNs and their effect on the GPU memory system from an architectural perspective.
- This work identifies the key limitations of current ML frameworks' memory management policies as they require the network-wide memory usage of the target DNN to monolithically fit within the physical capacity of the GPU. We demonstrate this by showing that existing frameworks fail in training 6 out of the 10 studied DNNs when their memory allocation size (14 GB to 67 GB) exceeds the GPU memory budget (12 GB in NVIDIA's Titan X).
- We propose, implement, and evaluate a runtime memory manager called vDNN that virtualizes the memory usage of neural networks across CPU and GPU memories. Our vDNN solution reduces the average GPU memory usage of these 6 memory hungry networks by 73% to 98%, allowing them to be trained on a single Titan X card. Compared to a hypothetical, oracular GPU containing enough memory to hold the entire DNN, vDNN incurs 1% to 18% performance overhead.

II. Background and Motivation

This section provides an overview of modern DNNs, the memory management policies of current ML frameworks, and their key limitations that motivate this work.

A. DNN Architecture

Convolutional neural networks are one of the most popular 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 [19] 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 [1], OverFeat [30], GoogLeNet [17], and VGG [14]. 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 [14, 15, 17].

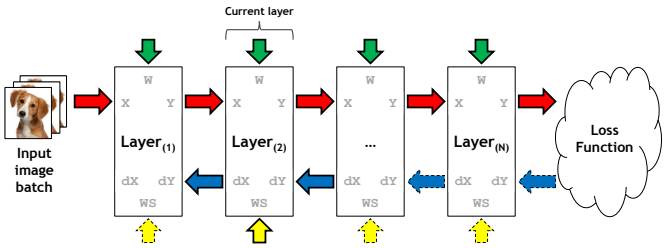

Fig. 2: Memory allocations required for linear networks using the baseline memory manager (bold arrows). For inference, the sum of all green (W) and red (X) arrows are allocated. For training, two additional data structures for dX and dY are required: both are sized to the maximum of all blue (dY) arrows and are reused while traversing back the layers during backward propagation. An optional temporary buffer, called *workspace* in cuDNN [8] (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.

B. DNN Training vs. Inference

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 [19]. 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 layer's 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[1]. For instance, a CONV layer using the most memory-efficient convolutional algorithm (e.g., implicit GEMM in cuDNN [8][2]) 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.

[1]Popular activation functions (sigmoid/tanh/ReLU [1]) can be refactored into an *in-place* algorithm using element-wise computation. Both Caffe and Torch leverage this in-place memory optimization and only allocate memory space for Y and dY for forward (Y) and backward (both Y and dY) propagation [31]. This paper adopts this in-place optimization for both baseline and vDNN for a conservative evaluation.

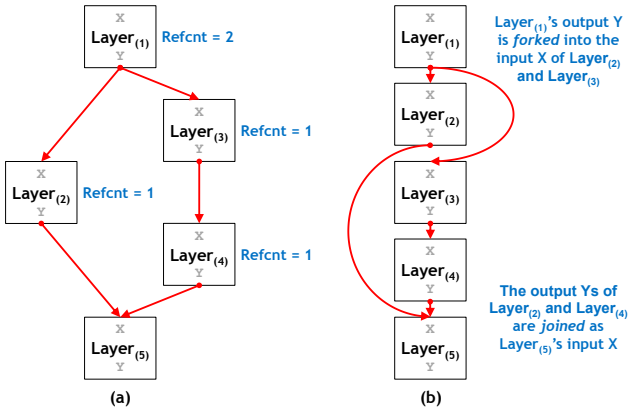

Fig. 3: (a) The computation graph and its inter-layer dependencies of a GoogLeNet-style, non-linear feedforward network during forward propagation. Refcnt refers to the number of consumer layers that depends on the current, producer layer's Y. The order in which the GPU processes each layer's forward 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.

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:

$$\frac{\partial Loss}{\partial Y_{(N)}} \tag{1}$$

The value in Equation 1 is forwarded to the last layer$_{(N)}$ as its input gradient maps (dY), and the output gradient maps (dX) are derived based on the *chain rule* [19]:

$$\frac{\partial Loss}{\partial X_{(N)}} = \frac{\partial Loss}{\partial Y_{(N)}} \cdot \frac{\partial Y_{(N)}}{\partial X_{(N)}} \tag{2}$$

Because the output dX ($\frac{\partial Loss}{\partial X_{(N)}}$) is the product of the input dY ($\frac{\partial Loss}{\partial Y_{(N)}}$) with $\frac{\partial Y_{(N)}}{\partial X_{(N)}}$, 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).

[2]cuDNN (version 4.0) provides six different convolutional algorithms. Implicit GEMM requires the least memory allocation as no additional workspace is needed. FFT-based convolutional algorithms on the other hand incur larger memory allocations because of the additional data structures required to store the feature maps transformed into frequency domain. More details are available in [8, 32].

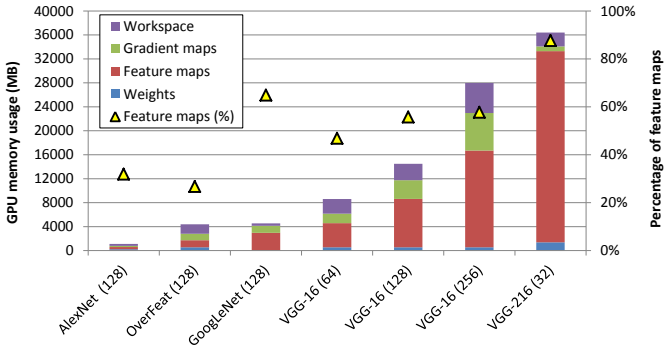

Fig. 4: Breakdown of GPU memory usage based on its functionality (left axis). The right axis shows the fraction of allocated memory consumed by feature maps.

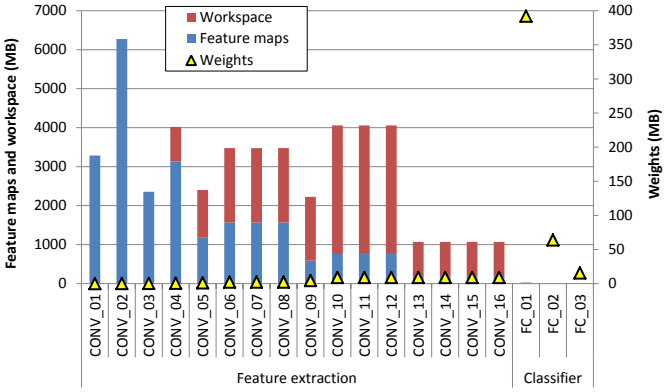

Fig. 5: Per layer memory usage of VGG-16 (256). For brevity, we only show the memory usage during forward propagation and for layers that contain weights (CONV and FC). Left axis corresponds to the sum of workspace and per layer input/output feature maps. The right axis corresponds to the memory consumption for storing weights. The memory usage during backward propagation follows similar trends to this figure.

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, `dY`s. 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. Because of the stochastic nature of SGD-based backward propagation, 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.

C. Motivation: Scalable and Memory-Efficient DNN Design

To aid the design and deployment of neural networks, a variety of ML frameworks have been developed in recent years, including Caffe, Torch, Neon, TensorFlow, and Theano [9]. The rich set of features offered by these frameworks coupled with their ability to accelerate DNN training and inference using GPUs greatly simplifies the process of implementing neural networks. Despite their flexibility, popular ML frameworks suffer from severe limitations in the way they allocate and manage memory.

To illustrate the shortcomings of ML frameworks in managing memory, consider the example shown in Figure 2. When training a DNN using existing ML frameworks, the memory required across *all* of the layers of the network must fit within the physical GPU memory capacity. The key reason for this GPU-side, network-wide memory allocation strategy is to reap performance benefits. More specifically, page-migration based virtualization solutions that expose both CPU and GPU memory for page allocations (regardless of whether the virtualization feature is provided by future CUDA runtime extensions or programming models such as OpenMP (4.0) [33]) 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. Prior work [34] 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 10s of GBs for very deep networks (Figure 15), ML frameworks will suffer from huge performance penalties when relying on page-migration for training DNNs.

Note that because of the layer-wise gradient update rule of the backward propagation algorithm (property of the chain rule, Section II-B), each layer's feature maps (`X`) are later *reused* during its own backward propagation pass. This means that *all* `X`s must still be available in GPU memory until backward computation is completed. Figure 4 shows the amount of memory usage based on its functionality and the growing significance of feature maps as networks become deeper. Because deeper networks need to keep track of a larger number of `X`s, the fraction of memory allocated for feature maps grows monotonically as the number of layers increases. Training the network itself is still done layer-wise, however, regardless of the depth of the neural network. The baseline network-wide memory allocation policy is therefore both extremely wasteful and not scalable because it does not take into account the layer-wise DNN training. Figure 5 shows the per layer memory usage of VGG-16 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

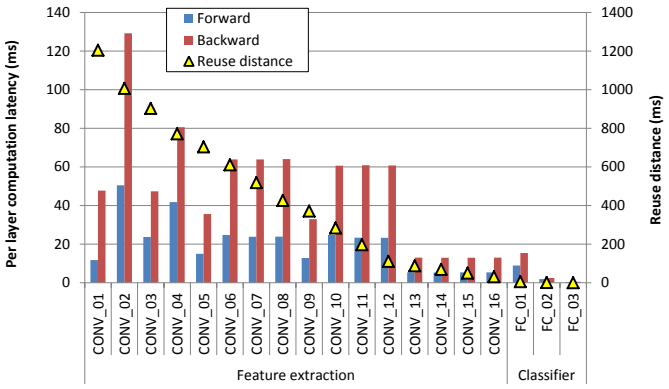

Fig. 6: VGG-16's per layer computation latency for forward and backward propagation (left axis). Right axis shows the reuse distance of each layer's input feature maps, X. We define the reuse distance of a layer$_{(n)}$'s X as the latency between the completion of layer$_{(n)}$'s forward propagation and the start of the same layer$_{(n)}$'s backward propagation.

GB of memory required by the baseline policy (Figure 1), showing significant opportunities for memory savings with a fine-grained, layer-wise memory management policy.

III. Virtualized DNN

The design objective of our virtualized DNN (vDNN) memory manager is to *virtualize* the memory usage of DNNs, using both GPU and CPU memory, while minimizing its impact on performance. vDNN is completely transparent to the programmer as the allocation, placement, movement, and release of data is seamlessly orchestrated by the system architecture and the runtime system. Such abstraction enables ML practitioners to focus more on their ML algorithm and not have to worry about the low level details of GPU memory management. vDNN primarily optimizes the memory usage of the feature extraction layers as the majority of memory usage is concentrated on these layers, accounting for 81% of memory usage on AlexNet and 96% on VGG-16 (256). More specifically, we target the feature maps of these feature extraction layers as these intermediate data structures account for the majority of GPU memory usage (Figure 4 and Figure 5). The intuitions of vDNN can also be applied to weights and to the classification layers, but with less of a memory saving benefit.

A. Design Principle

Previous sections highlighted the fact that the 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.

Forward Propagation. As discussed in Section II-C, deep networks have to keep track of a large number of the intermediate feature maps (Xs) that are extracted during forward propagation. Once a given layer$_{(n)}$'s forward computation is complete, however, 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 on 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 (Figure 6). As a result, tackling these Xs for memory optimization is crucial for efficient utilization of GPU memory as these intermediate data account for a significant fraction of memory allocations (Figure 4). vDNN therefore conditionally *offloads* these intermediate Xs to CPU memory via the system interconnect (e.g., PCIe, NVLINK [35]) if they are targeted for memory release. Section III-C details the vDNN memory transfer policy that decides which layers are chosen for offloading its X. Once the offload operation is complete, vDNN *releases* the offloaded X from the memory pool to reduce GPU memory usage.

Care must be taken however when evaluating the feasibility of offloading a layer's input X. This is because, for non-linear network topologies, multiple layers can be the consumers of a previously computed layer's output feature maps (Y). For instance, layer$_{(2)}$ and layer$_{(3)}$ in Figure 3 are both using the output Y of layer$_{(1)}$ as its input X. Offloading and consequently releasing the input X of layer$_{(2)}$, before reaching

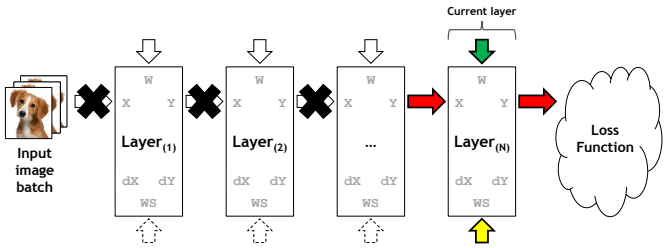

Fig. 7: Execution flow of a linear network during forward propagation. The figure assumes that layer$_{(N)}$ is currently being processed by the GPU. During this layer's forward computation, the data associated with the arrows marked with black Xs (all preceding layer's input feature maps) are not used and can safely be released from the memory pool.

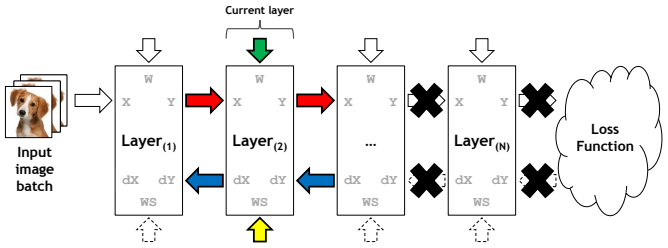

Fig. 8: Execution flow of a linear network during backward propagation. The figure assumes that layer$_{(2)}$ is currently being processed by the GPU. Data associated with the arrows marked with black Xs can safely be released because they will not be reused during the training of this input image batch.

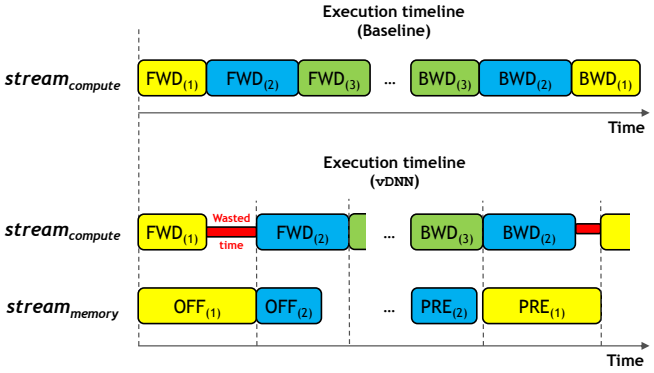

Fig. 9: Performance effect of offload and prefetch. $\text{FWD}_{(n)}$ and $\text{BWD}_{(n)}$ are the forward and backward computations for $\text{layer}_{(n)}$, respectively. $\text{OFF}_{(n)}$ is the offloading of $\text{layer}_{(n)}$'s X and $\text{PRE}_{(n)}$ is the corresponding prefetch operation for $\text{layer}_{(n)}$.

```
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 }
```

Fig. 10: 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.

$\text{layer}_{(3)}$'s forward computation, is problematic as these two layers share the same data structure for the input X. vDNN therefore keeps track of the inter-layer dependencies in the form of a dataflow graph (e.g., Refcnt in Figure 3) and allows the offload/release operation to be initiated only when the currently processing layer is the last consumer of its input feature maps. Figure 7 is an example execution flow of a linear DNN during forward propagation, highlighting when it becomes safe to release a layer's X.

Backward Propagation. Similar to forward propagation, vDNN aggressively releases data structures that are not needed for training the remaining layers' backward computation. During $\text{layer}_{(n)}$'s backward propagation, $\text{layer}_{(n+1)}$'s Y and dY are no longer required because the GPU has already completed the gradient updates for this layer (Figure 8). Again, by leveraging the layer-wise DNN backward propagation, vDNN immediately frees up a layer's Y and dY once this layer's backward computation is complete. X and dX are not released as the preceding layer's backward propagation will be needing these values for gradient derivation. Note that if a layer has offloaded its X 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 $\text{layer}_{(n)}$'s offloaded feature maps, which is overlapped with $\text{layer}_{(m)}$'s backward computation, with $n < m$, so that prefetching is launched before its actual usage, hiding prefetching latency.

B. Core Operations And Its Design

vDNN is prototyped as a layer on top of cuDNN [8]. Each layer keeps track of the cross-layer data dependencies of input/output feature maps so that the vDNN offload and release operations are properly scheduled. vDNN employs two separate CUDA streams [36] to overlap normal DNN computations with the memory allocation, movement, and release operations of vDNN. $\text{stream}_{compute}$ 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. The CUDA library only supports *synchronous* memory (de)allocations, meaning that any calls to cudaMalloc() or cudaFree() will enforce an additional synchronization across all the GPUs within a node. To safely enable vDNN memory operations while not fall into the pitfalls of synchronous CUDA APIs, we employ the open-source *asynchronous* memory allocation/release API library distributed by NVIDIA [37]. 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 without having to call cudaMalloc() or cudaFree().

Memory Offload. Offloading input feature maps is one of the key enablers of vDNN's memory savings. 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. The current implementation of vDNN synchronizes $\text{stream}_{compute}$ and stream_{memory} at the end of each layer's forward computation if stream_{memory} has offloaded its feature maps. This approach guarantees that the offloaded data is safely released from the memory pool before the next layer begins forward computation, maximizing the memory saving benefits of offloading. Because the Xs of CONV and POOL layers are read-only data structures, overlapping $\text{layer}_{(n)}$'s offload operation with the same layer's forward propagation does not create any correctness issues. ACTV layers are already refactored into an in-place algorithm and only use Y and dY for gradient updates, obviating the need for memory offloading (Section II-B). Figure 9 provides an overview of vDNN's offload operation. Here, the baseline system is able

to immediately launch layer$_{(2)}$'s forward computation once layer$_{(1)}$ is complete. The execution of layer$_{(2)}$ is stalled for vDNN, because stream$_{compute}$ must wait until the offloading operation of stream$_{memory}$ is complete, blocking layer$_{(2)}$'s computation. The computation of layer$_{(3)}$ is not delayed however because the offload latency for layer$_{(2)}$ is completely hidden inside the latency to compute the same layer's forward propagation.

Memory Prefetch. Similar to offloading, prefetching the offloaded Xs back to GPU memory is implemented using cudaMemcpyAsync() to overlap data transfers with the computations of backward propagation. However stream$_{memory}$ launches prefetch operations in the reverse order relative to the offload operations from forward propagation (Figure 9). As mentioned in Section III-A, the general rule of prefetching is to overlap the memory copy operation of layer$_{(n)}$'s offloaded data with layer$_{(m)}$'s backward computation, with layer ID m always being higher than n to maximize the benefit of both prefetching and latency hiding. In other words, when the GPU starts the backward propagation of layer$_{(m)}$, vDNN determines the best layer to prefetch among the preceding layers (as $n < m$).

If the distance between the prefetched layer$_{(n)}$ and overlapping layer$_{(m)}$ is too far away, the memory saving benefit of vDNN offloading will be reduced because the reuse time of this prefetched data will be distant in the future. In other words, prefetching data too early in time will again suboptimally utilize GPU memory as the prefetched data will once again camp inside the GPU memory without immediate usage. We carefully designed the vDNN prefetch algorithm to avoid this pitfall and balance the memory saving benefits of offloading with the timeliness of prefetching. Figure 10 is a pseudo-code of the vDNN prefetch algorithm that determines the best candidate layer for prefetching. Before stream$_{compute}$ starts a layer's backward computation, vDNN first searches for a potential layer that requires prefetching of its X. If the search operation is successful (line 11), the layer ID to be prefetched is returned by findPrefetchLayer routine and is used to launch its prefetch operation via stream$_{memory}$. Similar to offloading, 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. This benefit of course comes at the cost of a potential performance loss when the prefetch latency is longer than the overlapped computation, which we detail in Section V-C.

C. vDNN 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*. If vDNN were to use the most memory-efficient algorithm for all layers (e.g., implicit GEMM in cuDNN [8] which does not require any WS allocations) while also having all layers offload/prefetch, the GPU memory usage will be the lowest. Performance will likely suffer, however, compared to a baseline with the fastest convolutional algorithms adopted for each layers; the performance loss primarily comes from 1) the additional latency possibly incurred due to offload/prefetch, and 2) the performance difference between memory-optimal implicit GEMM and the performance-optimal convolutional algorithm. Going with the fastest algorithm, without any offload/prefetch, will result in the highest possible performance, but the potential memory overheads for the faster algorithm's workspace and the cumulative Xs that camp inside the GPU memory will likely overflow GPU memory. Given that optimizing the layer-wise memory usage and its performance is in itself a multi-dimensional optimization problem, selecting the most optimal hyperparameters across the entire network is non-trivial. We therefore adopt 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. Feature extraction layers are mostly composed of CONV and ACTV layers with intermittent POOL layers that downsize the dimensionality of the feature maps. More than 70% to 80% of the (forward/backward) computation time however is spent on the CONV layers for deep neural networks. We therefore evaluate two *static* vDNN memory transfer options that exploit this computational characteristic. The first option we explore is to have the vDNN memory manager offload all of the Xs of all of the layers. This policy, vDNN$_{all}$, is our most memory-efficient solution as all Xs are offloaded and released from the GPU, drastically reducing device memory usage. The second vDNN policy is to only offload Xs for the CONV layers and leave the remaining layers' Xs resident inside GPU memory (vDNN$_{conv}$). The vDNN$_{conv}$ policy is based on the observation that CONV layers have a much longer computation latency than ACTV/POOL layers, being more likely to effectively hide the latency of offload/prefetch. Not surprisingly the performance of vDNN$_{conv}$ is generally higher than vDNN$_{all}$. But vDNN$_{all}$ has the advantage of consuming the least GPU memory, significantly enhancing the trainability of a DNN. We later evaluate the memory usage and performance of these two static policies with both the memory-optimal and performance-optimal convolutional algorithms employed.

Dynamic vDNN. While static vDNN is simple and easy to implement, it does not account for the system architectural components that determine the trainability and performance of a DNN (e.g., maximum compute FLOPs and memory bandwidth, memory size, effective PCIe bandwidth, etc). For DNNs that comfortably fit within GPU memory, neither vDNN$_{all}$ nor vDNN$_{conv}$ is optimal as the best approach is to have all the memory allocations resident in GPU without

any offloading and employ the fastest possible convolutional algorithm. Large and deep networks, on the other hand, might not have the luxury of using faster convolutional algorithm. So being able to fit such network on the GPU is the best optimization vDNN could make. We therefore develop a *dynamic* vDNN policy that automatically determines the offloading layers and the convolutional algorithms employed, at runtime, to balance the trainability and performance of a DNN. Dynamic vDNN leverages several properties of DNN training. First, we exploit the millions to billions of iterations of the same forward/backward propagation pass that are required for training. NVIDIA's cuDNN provides a runtime API that experiments with all available convolution algorithms for a given layer, evaluating each algorithm's performance and its memory usage. Current ML frameworks leverage this API to undergo an initial profiling stage to determine the best algorithms to deploy for each CONV layer for best performance. The overhead of such profiling is on the order of a few tens of seconds, which is negligible relative to the days to weeks required for DNN training. Our dynamic vDNN augments this profiling stage with a number of additional profiling passes to select the best layers to offload and the best per layer algorithm. Once the baseline profile stage is completed and the fastest possible convolutional algorithms are derived for all CONV layers, dynamic vDNN employs the following additional profiling passes:

1) First, the static $\texttt{vDNN}_{all}$ is tested for a single training pass with all CONV layers using the memory-optimal, no-WS incurred algorithm. This initial pass determines if the target DNN can be trained at all as it requires the least GPU memory.
2) If $\texttt{vDNN}_{all}$ 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 $\texttt{vDNN}_{conv}$ and $\texttt{vDNN}_{all}$ respectively. If successful, vDNN employs the succeeded configuration for the rest of training. If both $\texttt{vDNN}_{conv}$ and $\texttt{vDNN}_{all}$ 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 budget. If so, then the given layer's convolutional 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 $\texttt{vDNN}_{conv}$ with each CONV layer initially using its own performance-optimal algorithm. If $\texttt{vDNN}_{conv}$ fails, then another training pass is launched with the more memory-efficient $\texttt{vDNN}_{all}$. If $\texttt{vDNN}_{all}$ also fails with this greedy algorithm, then vDNN resorts back to the very first $\texttt{vDNN}_{all}$ solution, with the memory-optimal, no-WS algorithms applied across the entire network.

While other possible settings might better balance performance and trainability, we find that our dynamic vDNN performs competitively without having to exhaustively search for globally optimal parameter selections.

IV. Methodology

A. vDNN Memory Manager

We implemented a host-side memory manager that interacts with the latest and fastest version of cuDNN 4.0 [8], serving as the GPU back-end. 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, $\texttt{stream}_{compute}$ and $\texttt{stream}_{memory}$ as discussed in Section III-B. The classification layers remain unchanged and use the same cuBLAS routines used in Torch. The vDNN API closely resembles that of Torch and Caffe, providing the high level abstractions of the target DNN and each of its layer compositions.

While there are subtle differences between Torch, Caffe, and Theano's memory allocation scheme, prior work [9] quantitatively demonstrated that all three frameworks exhibit comparable performance and memory requirements[3]. We therefore choose Torch's memory management policy as baseline to compare against vDNN given its widespread deployment across both academia and industry (e.g., Facebook and Google DeepMind). This baseline policy adopts the network-wide allocation policy discussed in Section II-C. However, we further improve this baseline policy using the following strategy to reduce memory consumption during the backward propagation phase [38, 39]: rather than allocating separate dY and dX for all individual layers, we only allocate the minimally required number of each of these data structures and reuse them after each layer's backward computation is complete (Figure 2).

B. GPU Node Topology

We conducted experiments on NVIDIA's Titan X [40], which provides the highest math throughput (single precision throughput of 7 TFLOPS), memory bandwidth (max 336 GB/sec), and memory capacity (12 GB) in the family of Maxwell GPUs. The GPU communicates with an Intel i7-5930K (containing 64 GB of DDR4 memory) via a PCIe switch (gen3), which provides a maximum 16 GB/sec data transfer bandwidth.

[3]Because TensorFlow is the least performant in terms of GPU memory usage and training speed [9], we do not discuss its memory management policy further in this paper.

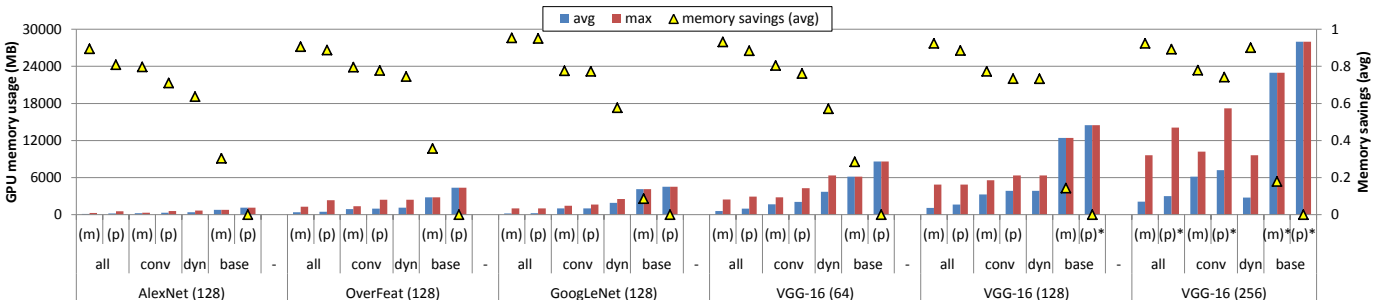

Fig. 11: Average and maximum memory usage (left axis). Right axis corresponds to the savings in average memory usage.

C. DNN Benchmarks

Conventional DNNs. First, we evaluate existing, state-of-the-art ImageNet winning DNNs: AlexNet [1], OverFeat [30], GoogLeNet [17], and three different batch sizes for VGG-16 (the deepest network with 16 CONV and 3 FC layers) [14]. The network configurations of these DNNs (e.g., layer type, batch size, etc.) are identical to the reference models maintained by the researchers at Facebook [41]. While the memory usage of AlexNet, OverFeat, and GoogLeNet is already below the 12 GB memory capacity of Titan X (Figure 4), we use it to evaluate the performance regression on these networks with vDNN. VGG-16 is one of the largest and deepest DNN architecture to date, requiring substantial memory capacity for trainability (using up to 28 GB of memory when trained with the best performing batch size of 256). Accordingly, Simonyan and Zisserman [14] parallelized VGG-16 (256) across four GPUs, with each GPU training VGG-16 (64) that fits within a single GPU memory budget. We therefore study VGG-16 with three batch sizes (64/128/256) and use it as a representative, future-looking DNN architecture that stresses the memory capacity limits of today's GPUs.

Very Deep Networks. To highlight vDNN's scalability in training very deep networks, we collected a second set of benchmarks by extending the number of CONV layers of VGG, from 16 CONV layers to 416 CONV layers. The original VGG network features a homogeneous architecture that only uses 3×3 convolutions (with stride 1 and pad 1) and 2×2 pooling operations (with stride 2), from the first to the last feature extraction layer. The feature extraction layers are conceptually divided into five groups of CONV layers, separated by the intermediate POOL layers. The only difference among these CONV layer groups is that the number of output feature maps grows from 64 to 512, from the first to the last layer group. Simonyan and Zisserman [14] studied the effect of layer depth on classification accuracy by incrementally adding more CONV layers to each of these layer groups, going from 8 CONV layers to 16 CONV layers. We follow similar measures to deepen the layer depth of VGG by gradually adding 100 more CONV layers to VGG-16, resulting in VGG-116/216/316/416 configurations. Each addition of 100 CONV layers is done by adding 20 more CONV layers to each of the five CONV layer groups. The added CONV layers have the same number of output feature maps that are employed for that layer group. We use these four VGG-style networks to perform a case study on vDNN's scalability on training very deep networks that require much more memory. Compared to conventional DNNs whose input batch size is in the order of hundreds of images, we study these very deep networks with a relatively small batch size of 32 in order to highlight the memory scaling effect of layer *depth* on DNNs.

V. Results

This section evaluates the effect of vDNN on GPU memory usage, off-chip memory bandwidth utilization, GPU power consumption, and overall performance. The static vDNN_{all} and vDNN_{conv} policies are denoted as `all` and `conv` in all the figures discussed in this section and 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 III-C. Memory management policies that fail in training the network, due to memory oversubscription, are marked with $(*)$.

A. GPU Memory Usage

Because vDNN adopts a layer-wise memory allocation policy, the GPU memory usage during forward/backward propagation will fluctuate depending on the memory offloading policy chosen and the convolutional algorithm employed for a given layer (Figure 5). We therefore discuss both the maximum and average memory usage as shown in Figure 11. The maximum memory usage corresponds to the largest memory allocated across the entire run, which decides whether the target DNN application can be trained at all. The average memory on the other hand reflects how much memory has been used on average, and conversely, freed up during forward/backward propagation. The smaller the average memory usage becomes, the more likely vDNN will have headroom to improve performance by: 1) employing performance-efficient convolutional algorithms that require larger workspace, and 2) reducing the total number of offload layers and prevent potential performance drops due to offloading (Figure 9).

Because the baseline policy provisions the memory allocations to accommodate the entire network usage, the maximum and average memory usage are identical. The baseline policy

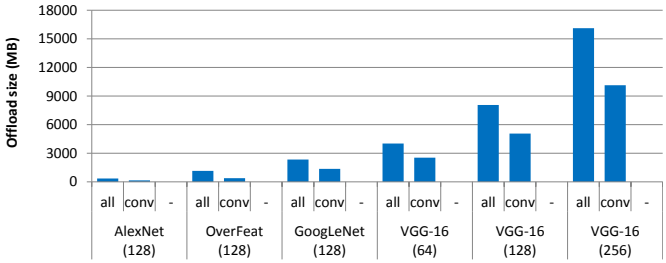

Fig. 12: The size of GPU memory allocations offloaded to host-side pinned memory using `cudaMallocHost()`.

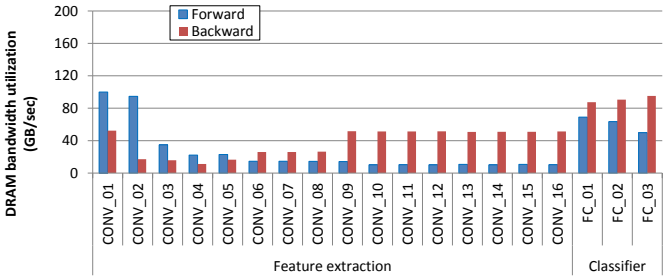

Fig. 13: Maximum DRAM bandwidth utilization for each CONV layer's forward and backward propagation.

therefore is not able to train networks like VGG-16 with batch 128 and 256 , which require more than the physically available 12 GB of memory. Our vDNN enhances the trainability of a network by significantly reducing its memory requirements. Overall, the memory-optimal $\text{vDNN}_{all}(m)$ shows both the smallest average and maximum memory usage as it always offloads a layer's input feature maps while using the most memory-efficient algorithms. As a result, vDNN_{all} exhibits the highest offload traffic being sent to host memory, reaching up to 16 GB of GPU memory savings for VGG-16 (256) (Figure 12). Such aggressive offloading significantly improves memory efficiency and achieves an average 73% and 93% reduction on the maximum and average memory usage of the six networks shown in Figure 11. When employed with the performance-optimal algorithm, the average memory savings of vDNN_{all} are slightly reduced to 64% and 90% for the maximum and average memory usage. Because vDNN_{conv} only offloads the feature maps for the CONV layers, its memory savings is not as high as vDNN_{all}. However, vDNN_{conv} still reduces the maximum and average memory usage by 52% and 76% on average, even with the performance-optimal algorithms employed across the network.

vDNN_{dyn} allocates the largest memory among the three vDNN policies, reducing the maximum and average memory consumption by 49% and 69% on average compared to baseline. This is because vDNN_{dyn} tries to balance its memory usage and performance, seeking to fit the network within GPU memory while still optimizing performance by minimizing the number of offload layers and employing the fastest possible convolutional algorithms. The static vDNN_{all} and vDNN_{conv}, on the other hand, do not consider the overall performance when the offloaded layers are chosen. For instance, VGG-16 (128) trained with memory-optimal vDNN_{all} only uses up to 4.8 GB out of the 12 GB of available memory. This configuration leads to a 61% performance loss (Section V-C) as vDNN_{all} fails to exploit the remaining 7.2 GB of the memory for performance optimizations. vDNN_{dyn} tries to bridge this gap by dynamically deriving the offload layers as well as the best convolutional algorithms to employ for each layer. We further discusses vDNN's impact on performance in detail at Section V-C.

B. Impact on Memory System

While vDNN helps virtualize DNN's memory usage, it does come at the cost of adding more read (offload) and write (prefetch) traffic to the GPU memory subsystem, potentially interfering with the normal cuDNN operations. Because the additional vDNN memory traffic can be up to the bandwidth of the PCIe (maximum of 16 GB/sec for gen3), its effect on performance will be determined by the normal cuDNN operation's memory bandwidth intensity. Figure 13 shows the baseline's maximum DRAM bandwidth utilization for VGG-16, which is measured separately for each CONV layer's forward and backward propagation. The feature extraction layers rarely saturate the 336 GB/sec of peak memory bandwidth, providing more than enough headroom for vDNN's offload/prefetch traffic. Even if a hypothetical, future convolutional algorithm were to completely saturate the off-chip DRAM bandwidth, vDNN's additional traffic will approximately incur up to a worst-case (16/336) = 4.7% performance overheads, which we believe is reasonable given the benefit of virtualized memory.

C. Performance

Figure 14 summarizes the performance of vDNN compared to baseline. For a conservative evaluation, we only compare the latencies incurred in the feature extraction layers because the classifier layers are executed identically for baseline and vDNN. Because the baseline policy requires more than 12 GB of memory for VGG-16 (128) and VGG-16 (256) with performance-optimal algorithms (15 GB and 28 GB respectively), it is impossible to train these two networks on a Titan X. We therefore establish an *oracular* baseline that removes the memory capacity bottlenecks of these two networks for a conservative evaluation. The performance of this oracular baseline is estimated by configuring all CONV layers with the fastest algorithms and evaluating the latencies of each layers individually. The latencies are later accumulated altogether to estimate overall performance. Overall, vDNN_{all} and vDNN_{conv} with memory-optimal algorithms exhibit an average 58% and 55% performance loss (maximum 65% and 63% degradation) compared to baseline, an expected result as the memory manager puts no effort into balancing memory usage and overall performance. The dynamic vDNN_{dyn} does much better in terms of balancing memory efficiency and overall throughput, closing the performance gap between the static vDNN and baseline and reaching an average 97% of baseline's throughput (worst case 82% of the oracular baseline, for VGG-16 (256)).

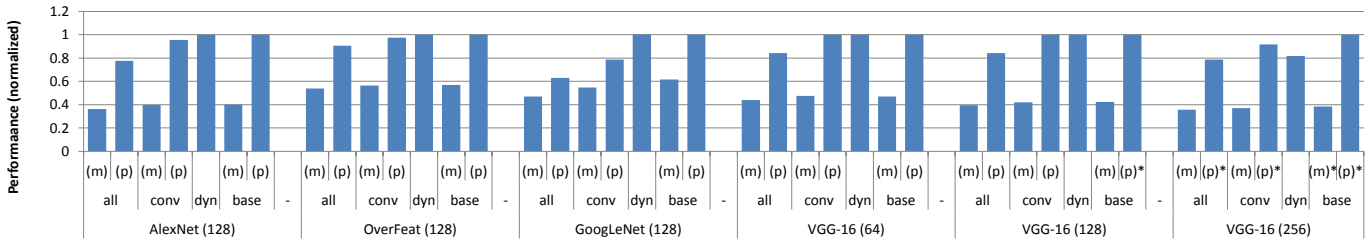

Fig. 14: Overall performance (normalized to baseline).

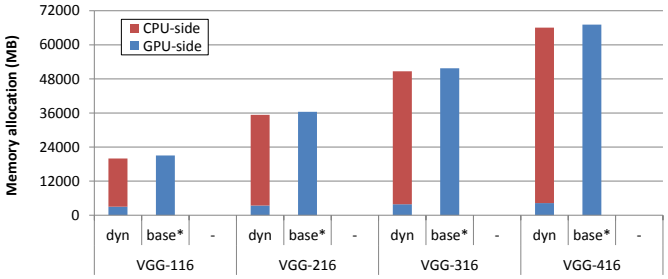

Fig. 15: CPU/GPU memory allocations for VGG-style, very deep networks (batch 32) with $\texttt{vDNN}_{dyn}$ and baseline.

D. Power

This section discusses the effect of $\texttt{vDNN}_{dyn}$ on overall GPU power consumption. We use the system profiling utility of `nvprof` [42] to measure the average and maximum GPU power consumption. Each network is executed for 50 iterations of forward and backward propagation and the reported average and maximum power consumption is averaged altogether. All but VGG-16 (128) have been executed with the performance-optimal convolutional algorithms because VGG-16 (128) can only be trained with the memory-optimal algorithms under baseline (Figure 11). Note that the results for VGG-16 (256) are not discussed as this configuration can only be trained with `vDNN`, making it impossible to compare against baseline. Overall, $\texttt{vDNN}_{dyn}$ incurs 1% to 7% maximum power overheads. As discussed in Section V-B, the offload/prefetch memory traffic of `vDNN` is one of the biggest contributors to the instantaneous rise in peak power consumption. Nonetheless, the average power consumption (energy/time) is rarely affected because of the following two factors: 1) $\texttt{vDNN}_{dyn}$ does not incur any noticeable performance overhead for these five networks, and 2) the studied DNNs rarely saturate the peak DRAM bandwidth (Figure 13), so the additional energy overheads of `vDNN` memory traffic is expected to be negligible on average (Section V-B).

E. Case Study: Training Very Deep Networks

To highlight `vDNN`'s scalability in training very deep networks, we perform a case study on four VGG-style networks that contain hundreds of CONV layers and scale up the network memory requirements. As mentioned in Section IV-C, the batch size is set to be much smaller than those studied in previous subsections (which ranges from batch 128 to 256) as means to highlight the memory scaling effect of layer depth despite its small batch size. Figure 15 shows the memory allocation requirements of baseline and $\texttt{vDNN}_{dyn}$ for these very deep neural networks. As the number of CONV layers increases from 16 to 416, the baseline memory requirements monotonically increase by 14 times (from 4.9 GB to 67.1 GB), even with a small batch size of 32. Thanks to its layer-wise memory allocation policy, $\texttt{vDNN}_{dyn}$ significantly reduces the memory usage of all four networks, only using up to 4.2 GB of GPU memory and having all remaining 81% to 92% of overall memory allocations to reside in CPU memory. Compared to the oracular baseline, $\texttt{vDNN}_{dyn}$ also did not incur any noticeable performance degradations because the offload and prefetch latency is completely hidden inside the layer's DNN computations while still being able to employ the performance-optimal algorithms across the network.

VI. Related Work

There have been a variety of proposals aiming to reduce the memory usage of neural networks. Network pruning techniques [43, 44, 45, 46, 47] remove small valued weight connections from the network as means to reduce network redundancy, leading to a reduction in memory consumption. Another redundancy mitigating approach uses quantization [48] or reduced precision [49] to reduce the number of bits required to model the network. A variety of network compression techniques have also been explored by Gong et al. [50] to reduce the memory usage of DNNs.

Although these prior studies reduce DNN memory requirements, they fall short in several respects. First, weights only account for a small fraction of the memory usage in state-of-the-art DNNs, as shown in Figure 4. Thus, proposals that optimize memory usage of weights, while beneficial in terms of memory bandwidth utilization and energy-efficiency, provide only limited opportunity for memory capacity savings. Second, using reduced precision occasionally results in loss of classification accuracy unless carefully tuned for the given network and task. Our proposal optimizes the memory consumptions of the intermediate feature maps which are the most dominant data structures in DNNs.

Several prior works discussed mechanisms to support virtualized memory on GPUs. Pichai et al. [51] and Power et al. [52] proposed TLB implementations that consider the unique memory access patterns of GPUs, improving the throughput of address translations as well as overall system throughput. Zheng et al. [34] discuss features needed in the GPU hardware and software stack to close the performance gap of GPU paged memory versus legacy programmer-

directed memory management techniques. As discussed in Section II-C, page-migration based virtualization solutions are likely to underutilize PCIe bandwidth significantly and incur performance overheads when training networks that oversubscribe GPU memory.

While less directly related to vDNN, a variety of accelerator architectures have also been proposed for DNNs [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 53]. While these custom ASIC designs drastically improve the energy-efficiency of DNNs, none of these address the memory capacity bottlenecks of DNN training, a unique contribution of our work.

VII. Conclusion

Existing machine learning frameworks require users to carefully manage their GPU memory usage so that the network-wide memory requirements fit within the physical GPU memory size. We propose vDNN, a scalable, memory-efficient runtime memory manager that virtualizes the memory usage of a network across CPU and GPU memories. Our vDNN solution reduces the average GPU memory usage of AlexNet by up to 89%, OverFeat by 91%, and GoogLeNet by 95%, substantially improving the memory-efficiency of DNNs. Similar experiments to VGG-16 (256) result in an average 90% reduction in memory usage at a cost of 18% performance penalty compared to an oracular baseline. We also study the scalability of vDNN to extremely deep neural networks, showing that vDNN can train networks with hundreds of layers without any performance loss.

Acknowledgment

We thank our colleagues at NVIDIA for their feedback on this work, and in particular John Tran, Sharan Chetlur, Simon Layton, and Cliff Woolley for their contributions to vDNN concepts and infrastructure.

References

[1] 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.

[2] A. Graves and J. Schmidhuber, "Framewise Phoneme Classification With Bidirectional LSTM and Other Neural Network Architectures," in *Neural Networks*, 2005.

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

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

[5] Torch, "http://torch.ch," 2016.

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

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

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

[9] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative Study of Caffe, Neon, Theano, and Torch for Deep Learning," in *arxiv.org*, 2016.

[10] The Next Platform (www.nextplatform.com), "Baidu Eyes Deep Learning Strategy in Wake of New GPU Options," 2016.

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

[12] A. Krizhevsky, "One Weird Trick For Parallelizing Convolutional Neural Networks," in *arxiv.org*, 2014.

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

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

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

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

[17] 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.

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

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

[20] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

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

[22] Y. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Conference on Solid-State Circuits*, 2016.

[23] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings of ACM/IEEE International Symposium on Computer Architecture*, 2016.

[24] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *Proceedings of ACM/IEEE International Symposium on Computer Architecture*, 2016.

[25] R. LiKamWa, Y. Hou, M. Polansky, Y. Gao, and L. Zhong, "RedEye: Analog ConvNet Image Sensor Architecture for Continuous Mobile Vision," in *Pro-*

ceedings of ACM/IEEE International Symposium on Computer Architecture, 2016.

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

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

[28] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *Proceedings of ACM/IEEE International Symposium on Computer Architecture*, 2016.

[29] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Convolutional Neural Network Computing," in *Proceedings of ACM/IEEE International Symposium on Computer Architecture*, 2016.

[30] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks," in *arxiv.org*, 2013.

[31] S. Chintala, "https://github.com/torch/nn/pull/235," 2015.

[32] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," in *Proceedings of the Advances in Neural Information Processing Systems*, 2014.

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

[34] 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.

[35] NVIDIA, "NVIDIA NVLINK High-Speed Interconnect," 2016.

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

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

[38] S. Gross and M. Wilber, "Training and Investigating Residual Nets," 2016.

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

[40] NVIDIA, "GeForce GTX Titan X (Maxwell)," 2015.

[41] S. Chintala, "https://github.com/soumith/convnet-benchmarks," 2016.

[42] NVIDIA, "CUDA Toolkit 7.5 Documentation: Profiler," 2016.

[43] S. Hanson and L. Pratt, "Comparing Biases for Minimal Network Construction with Back-propagation," in *Proceedings of the Advances in Neural Information Processing Systems*, 1989.

[44] Y. LeCun, S. Denker, and S. Solla, "Optimal Brain Damage," in *Proceedings of the Advances in Neural Information Processing Systems*, 1990.

[45] B. Hassibi and D. Stork, "Second Order Derivatives for Network Pruning: Optimal Brain Surgeon," in *Proceedings of the Advances in Neural Information Processing Systems*, 1993.

[46] S. Han, J. Pool, J. Tran, and W. Dally, "Learning Both Weights and Connections for Efficient Neural Networks," in *Proceedings of the Advances in Neural Information Processing Systems*, 2015.

[47] S. Han, H. Mao, and W. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," in *Proceedings of the International Conference on Learning Representations*, 2016.

[48] V. Vanhoucke, A. Senior, and M. Mao, "Improving the Speed of Neural Networks on CPUs," in *Proceedings of Deep Learning and Unsupervised Feature Learning*, 2011.

[49] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N. E. Jerger, R. Urtasun, and A. Moshovos, "Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets," in *arxiv.org*, 2016.

[50] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing Deep Convolutional Networks Using Vector Quantization," in *arxiv.org*, 2014.

[51] 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.

[52] 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.

[53] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in *Proceedings of ACM/IEEE International Symposium on Computer Architecture*, 2015.