Efficient GPU Memory Management for Nonlinear DNNs

Donglin Yang
University of North Carolina at Charlotte
Charlotte, NC, USA
dyang33@uncc.edu

Dazhao Cheng
University of North Carolina at Charlotte
Charlotte, NC, USA
Dazhao.Cheng@uncc.edu

ABSTRACT

Deep neural networks (DNNs) have been widely applied in the field of artificial intelligence, e.g., natural language processing, computer vision, etc. Researchers and industry practitioners typically use GPU to train complex hundred-layers deep networks. However, as the networks going wider and deeper, the limited GPU memory becomes a significant bottleneck, restricting the size of networks to be trained. In the training of DNNs, the intermediate layer outputs are the major contributors to the memory footprint. Offloading and prefetching feature maps is one of the crucial techniques to overcome the GPU memory shortage by utilizing the CPU DRAM as an external buffer for the GPU. However, we find that the layer-by-layer asynchronous approach cannot be effectively applied to the overlap between communication and computation, particularly for nonlinear networks. Furthermore, the default memory management policy could cause high GPU memory fragmentation for the networks with complex nonlinearities. Based on these observations, we adopt an efficient graph analysis and exploit the layered dependency structures to improve the overlap ratio. To achieve minimal memory fragmentation, we design a Group Tensors By Mobility (GTBM) placement policy to allocate tensors on the proposed unified memory pool for data structures with varied data sizes and dynamic dependencies. We implement and evaluate our system, Dymem, on several linear and nonlinear networks. Compared with vDNN and SuperNeurons, our proposed approach can achieve memory cost reduction by up to 31%. The dependency-aware strategy can improve the end-to-end throughput for nonlinear networks by up to 42%.

CCS CONCEPTS

• **Software and its engineering** → **Memory management**; • **Computer systems organization** → *Multiple instruction, single data.*

KEYWORDS

Neural networks; GPU Memory Management; Resource Management

HPDC '20, June 23–26, 2020, Stockholm, Sweden

ACM ISBN 978-1-4503-7052-3/20/06…$15.00
https://doi.org/10.1145/3369583.3392684

ACM Reference Format:
Donglin Yang and Dazhao Cheng. 2020. Efficient GPU Memory Management for Nonlinear DNNs. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '20), June 23–26, 2020, Stockholm, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3369583.3392684

1 INTRODUCTION

Deep learning (DL) [24] has achieved great success in various domains such as image classification [35], natural language processing [8], object detection [36], speech recognition [17], etc. Obtaining accurate deep learning models is a computation-intensive process, which requires large amounts of data and substantial computing capacity. Previous studies have shown that wider and deeper DNNs can significantly increase the model performance [23][3]. Recently, nonlinear architectures have been proposed to further improve the quality of image recognition tasks [34][13]. However, the limited size of GPU DRAM has been a major bottleneck for researchers to explore deeper and wider DNNs for better generalization performance. For example, it is reported that VGG-16 [32], which is composed of 16 computation-intensive convolution layers, requests a total of 28GB of memory usage for batch size 256 [4]. A representative nonlinear network, Inception-V4 requests up to 45GB memory to keep the entire network on the GPU in training [5]. However, the largest GPU memory capacity offered by the commercial NVIDIA Volta architecture so far is 32GB [2]. The memory shortage of GPU limits deep learning practitioners to deploy wider and deeper DNNs. There are many other system challenges for training deep neural networks. In this paper, we focus on memory optimization for nonlinear networks.

Many approaches have been proposed to reduce the GPU memory footprint of DNN training. However, these solutions have their limitations. For example, most prior works propose reducing the model size to reduce the memory footprint. However, this strategy either provides low memory footprint reduction or results in a loss in training accuracy [14][15]. Firstly, in DNN training, parameter weights only account for a small fraction of the total memory footprint. In training, intermediate feature maps are the primary contributor to the significant increase in memory footprint in DNN training. These intermediate values should be stored/stashed in the forward pass so that they can be reused later in the backward pass. Additionally, approaches that apply lower precision computations for DNN training, mostly in the context of ASICs and FPGAs, either do not target feature maps (and thus achieve low memory footprint reduction) or result in reduced training accuracy [20]. Memory compression [10] and data encoding [19] is another approach to reduce the GPU memory requirement for training, which, however, introduces high-performance overhead. State-of-the-art memory footprint reduction approaches for training swap data structures

back and forth between CPU and GPU memory [28][38]. However, existing swapping approaches are inefficient in reducing the memory footprint for training.

Inspired by the fact that DNN training follows a series of layer-wise computations, vDNN [28] and SuperNeurons [38] propose to virtualize the memory usage of deep neural networks across both GPU and CPU memories. Considering that GPU can only process one layer at any given time, it is not necessary to overprovision the memory allocation to accommodate the entire neural network on the GPU. vDNN and SuperNeurons release or move data structures, particularly the intermediate feature maps, between CPU and GPU, by exploiting the inter-layer dependencies and reuse patterns of DNNs. However, those techniques are not well-tuned to address the dependency and memory variations in nonlinear networks. Firstly, the core idea of vDNN and SuperNeurons is to offload data of one network layer when it is not required in the near future and can be released from GPU DRAM, saving space for other layers. The offloaded data is brought back to the GPU when needed in the backward pass. It can achieve optimal performance if the communication between CPU and GPU can be well hidden by computation to utilize the bandwidth. However, we observe that offloading data structures from the GPU to CPU or prefetching data back to GPU from CPU layer by layer brings significant inefficiency. For example, transfer time can be longer or shorter than the forward computation time across layers so that the communication can only be partially overlapped with the computation. Usually, the communication time is much longer than the computation time in Pooling layers. In contrast, the computation time is usually much longer than the communication time in convolution layers. Specifically, for nonlinear blocks, where there are join or fork connections, more benefit can be earned by aggressively advancing the computation in the forward pass or the prefetching operations in the backward pass. Secondly, none of the existing work presents an efficient solution to handle the memory fragmentation problem for nonlinear networks. Different from linear networks, which follow a simple and fixed execution pattern, causing negligible memory wastage, nonlinear networks exhibit varied dependencies and dynamic references. As a result, those complex nonlinear blocks, which has different data size, varied resident duration, and dynamic reference counts, interleave with layers which have simple dependencies.

We demonstrate that the default memory management can lead to higher fragmentation because the released memory regions cannot be coalesced into a larger one, resulting in free but usable space. Overall, we propose and design Dymem, a novel approach for training nonlinear networks. Instead of using a layer-by-layer strategy, Dymem adopts a more greedy asynchronous solution to maximize the DRAM bandwidth, balancing memory usage, and performance improvement. Furthermore, we first analyze the root cause of GPU memory fragmentation in DNN training. Then, we design a Group Tensors By Mobility (GTBM) placement policy to allocate tensors on the proposed unified memory pool based on mobility, exploiting the dependencies, and reuse distances. In a nutshell, we make the following technical contributions:

- We empirically study and demonstrate the inefficiency of memory swapping and memory allocation solutions in existing works. Based on the observations, we motivate the need for dependency-aware memory management for nonlinear networks.
- We present a memory-efficient graph analysis to construct an execution order for nonlinear networks and propose a dynamic offload/prefetch strategy to maximize the performance and usage of bandwidth.
- We design the first unified memory pool for nonlinear networks and propose a Group Tensors By Mobility (GTBM) placement policy to conserve contiguity for different tensors based on the newly defined tensor mobility.
- We implement Dymem to evaluate various nonlinear DNNs and perform comprehensive evaluations with various depths. It achieves memory cost reduction by up to 31% and improves the end-to-end throughput for nonlinear networks by up to 42%.

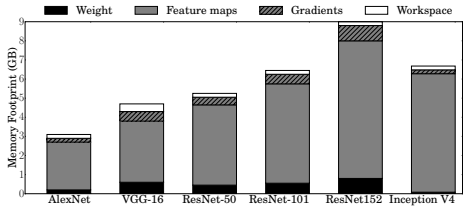
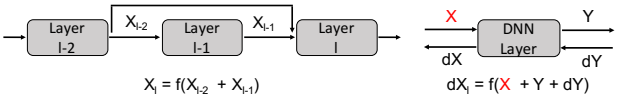


Figure 1: The breakdown of memory footprint in DNN training for different networks (GB).



(a) Forward pass. (b) Backward pass.

Figure 2: Forward and backward computation. X, Y, dX and dY are input feature maps, output feature maps, output gradient maps and input gradient maps respectively.

The rest of this paper is organized as follows. Section 2 gives background and motivations on memory management for nonlinear DNNs. Section 3 describes the detailed system design and solution to schedule and manage tensor allocation on GPU. Section 4 presents the experimental methodology, and Section 5 reports the evaluation results. Section 6 reviews related work. Section 7 concludes the paper.

2 BACKGROUND AND MOTIVATION

DNNs typically composes of an input and output layer with multiple hidden layers in between. Due to the limited GPU memory capacity, efficient memory management is important to run deeper and wider DNNs on GPU. This section summarizes the DNN models, current memory management optimization technique and motivates our work to overcome existing limitations.

2.1 DNNs Training on GPUs

DNN training is based on a set of inputs and obtained outputs. The training process consists of two phases: forward and backward

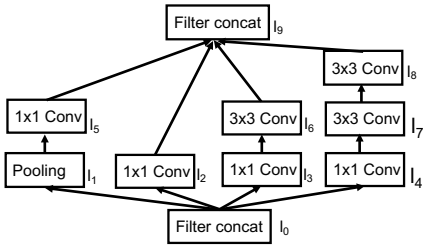
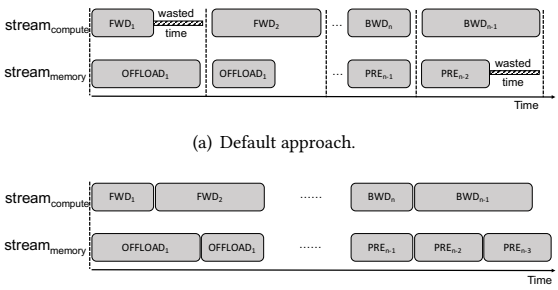


Figure 3: Schema of Inception-v4 network.



(a) Default approach.

(b) Aggressive approach.

Figure 4: Synchronization w/i and w/o barrier.

passes [25][29]. Specifically, the backward propagation algorithm is to propagate the error and search the gradient of the loss function that can be applied to adjust the parameters towards improving accuracy [26]. It consists of four types of data structure in the DNN training: feature maps, weights, gradients, and workspace. Feature maps are the intermediate results that are consumed in the following forward or backward layers. Gradient maps are the intermediate results that are generated in the backward pass and consumed by the dependent layers. Weight value decides how much influence the input will have on the output. The workspace is the intra-layer storage to speed up the layer computation. In particular, the workspace requires additional but temporary GPU DRAM to achieve better performance of the Convolution algorithm. In the forward pass, the input feature maps X are fed to the current layer in the forward direction through the network. Each hidden layer accepts the input data, processes it as per the activation function, and passes its output to the successive layer. Nonlinear networks contain one-to-many (fork) and many-to-one (join) inter-layer dependencies. For example, feature maps X from layer $l-2$ and layer $l-1$ l are joined as the input for layer l, as shown in Figure 2(a). Backward propagation computes the gradient in the weight space of a feedforward neural network, with respect to a loss function. Typically, in a backward calculation, a layer requires it stashed input feature maps X, output feature maps Y, and input gradient maps dY to obtain the output gradient maps dX, which can be shown in Figure 2(b). The backward propagation can only be performed when all of these required data structures are available on GPU.

Traditional Convolution Neural Networks (CNN) typically consists of several basic building layers, including Convolution (CONV), Pooling (POOL), Activation (ACT), Softmax, Fully Connected (FC), Batch Normalization (BN), and Dropout layers. A linear neural network is structured as a sequence of independent and interconnected instances. Recently, several nonlinear networks, such as Inception V4 [34] and ResNet [16], have been proposed to improve the state-of-the-art performance of image classification further. Training the elaborate neural network exhibits significant challenges considering the limited GPU memory. To understand the memory consumption behavior of DNN training, we conduct a study on GeForce GTX TITAN X GPU with six representative CNNs. Figure 1 shows the breakdown of the GPU memory footprint. It can be learned that the GPU memory tends to be mostly occupied by feature maps. For example, more than 90% of GPU DRAM is required by Inception V4. Furthermore, it is a waste to reside those feature maps on GPU memory even though they have future but distant dependencies, especially for the very deep neural networks. Thus, the intermediate feature maps are the key factors for optimizing memory usage in DNN training.

2.2 Nonlinearities in DNNs

For linear networks, data is sequentially propagated in both the forward and backward passes, following a fixed sequential execution pattern. Compared with linear networks, nonlinear networks have a high degree of dependency variations. To illustrate these, we use a representative nonlinear block from Inception-V4 [34] as an example, which is shown in Figure 3. There are two simple nonlinear connections: fan and join. In this example, the fan connection creates four branches after layer l_0. Each of them has a different number of layers. Each branch has to finish its computation before it reaches the join connection, i.e., layer l_9. Nowadays, a deep nonlinear network could have hundreds of fan and join connections inside the network, resulting in a complex network architecture [18]. Note that the GPU can only process a single layer's computation at any given time due to such inter-layer data dependencies. In terms of memory allocation, care should be taken because, in nonlinear networks, multiple layers consume the output feature maps from a previously processed layer in a fan connection. For example, the output feature maps from layer l_0 can only be released from GPU memory until layers l_1, l_2, l_3, and l_4 have been propagated. Similarly, in the join connection, all the output feature maps from preceding layers should reside on the GPU until their final consumer has completed the propagation. These nonlinear variations complicate runtime resource management, which requires a more efficient solution.

2.3 Motivation for Efficient GPU Memory Management

2.3.1 Memory Offload/Prefetch for DNNs. Several memory-reduction techniques have been proposed to address the problem of limited GPU resident memory. For example, vDNN [28] and Superneurons [38] choose to offload selected layers to the preallocated pinned CPU memory and prefetch these data back to GPU when required. Typically, in the forward pass, the input feature maps X from the preceding layer can be offloaded to CPU memory if there is no more dependency, after which these data can be released from the GPU memory. The runtime uses two independent processes to complete

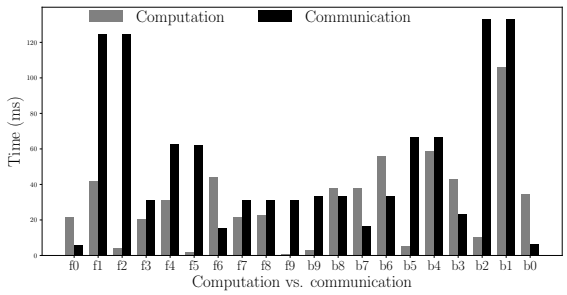


Figure 5: Computation and communication time for different layers in forward and backward pass.

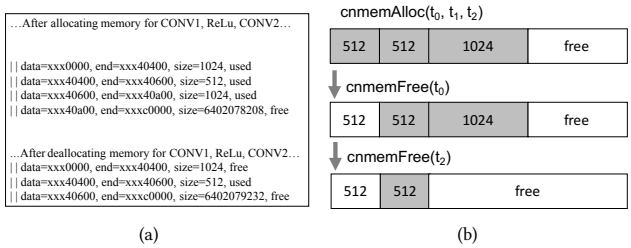
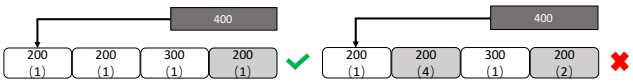


Figure 6: GPU memory allocation process.



(a) Linear network. (b) Nonlinear network.

Figure 7: 400 represents the memory request. (4) represents the reference counts. White blocks represent free regions, which can be coalesced if they are contiguous while grey blocks represent occupied regions.

the computation and communication, which enables the CPU-to-GPU data transfers to overlap with the computation asynchronously. In the backward pass, for those offloaded layers, the runtime should bring the feature tensors X back to the GPU DRAM before the backward dependent layer starts its propagation. The prefetch operation for layer m can be overlapped with the computation of layer l in the backward pass, where $l > m$. Ideally, this design can maximize the performance by hiding the communication by the computation time. To ensure the safety of parallel streams, they enforce a synchronization at the end of each layer, which means the communication and computation stream cannot advance each other in both the backward and forward passes, as shown in Figure 4(a). However, this design largely depends on the communication/computation ratio. It works well for the linear network, e.g., VGG-16 [32], which consists of twelve computation-intensive convolution layers. The offload/prefetch operation can be well hidden because convolution layers require longer computation time than transfer. This is inefficient, particularly for nonlinear networks. To demonstrate this, Figure 5 presents the communication and computation time for ten layers in both the forward and backward pass of GoogleNet [34]. From the Figure, f_5's computation time is much lower compared to offloading time, while the next layer's forward computation time is higher than the communication time. If f_5's input is decided to be offloaded, then it is not necessary to wait for the offloading of f_5 before starting the next layer's computation. A more efficient synchronization without a barrier is shown in Figure 4. Similarly, in the backward pass, when the layer b_7 is being propagated, the prefetching operation for layer b_5 can be initiated after the transfer of layer l_6 is finished. Secondly, the backward pass of each layer requires memory space for gradients input and output maps besides input and output feature maps compared with forward. Hence, in the forward pass, its peak memory requirement is not higher than the backward pass if this aggressive strategy is adopted to advance computation. However, care should be taken in the backward pass because aggressively prefetching data does not always bring the benefit. These observations motivate us to propose a more efficient memory scheduling strategy to balance memory saving and performance.

2.3.2 GPU memory fragmentation. Training a deep DNN on GPU with limited memory results in frequently caching and freeing tensors in a training iteration. To avoid the nontrivial allocation/deallocation overhead from using the native CUDA API, *cudaMalloc*, and *cudaFree*, GPU memory pool is always adopted as an effective memory optimization technique [38][28]. It preallocates a continuous chunk of memory as a shared memory pool and takes over memory management from the operating system. The preallocated memory pool returns a list of allocated but empty addresses. For an allocation request, the memory pool finds the first node with enough free memory from the empty list. After that, it updates the available list and the occupied list to track memory usage. For a deallocation request, the memory pool locates the node in the allocated list with the hash table, and then the pool puts the node back to the empty list. The sequences of allocation/free do not affect training but can impact the amount of fragmentation. For example, memory for the input of the layer to be prefetched could be allocated before/after allocating space for gradients of the input of the layer to be propagated in the backward pass. Figure 6 demonstrates the allocation process of CNMeM [1], which is a GPU memory pool developed by Nvidia. We allocate three tensors on the GPU, which require 512, 512, and 1024MB GPU memory, respectively. Then tensor t_0 and t_2 are freed from GPU. From the log information 6(a), we can see that coalescing operations can combine available contiguous regions into a lager page in the same virtual space. However, the released region for t_0 cannot coalesce with the other available regions because they are not in the continuous address, which is illustrated in Figure 6(b). This buffering and paging strategies work at the coarse memory granularity, which results in inefficiencies for nonlinear networks whose data sizes and dependencies vary significantly.

Case study: Figures 7 represent two simple examples of linear and nonlinear networks. In a linear network, all layers have only one reference. If a coming allocation request for tensor is 400MB, for the linear network, it can serve the request because the coalesced regions from the released tensors have enough space. However, for the nonlinear network, though there is 200MB and 300MB empty list, it

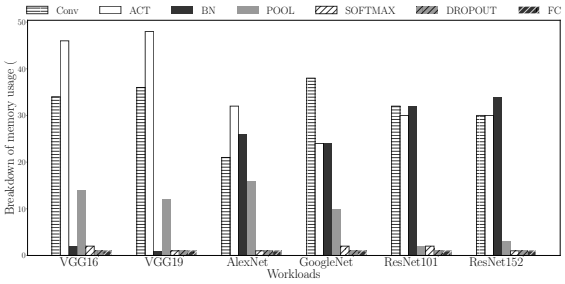


Figure 8: The fraction of memory usage by various layers for different networks.

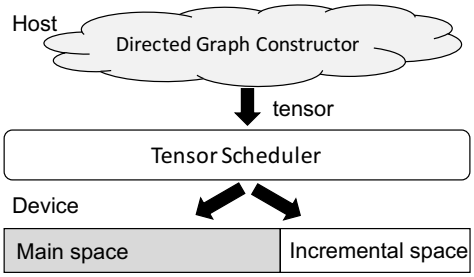


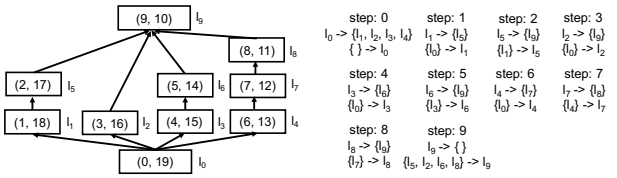
Figure 9: The system architecture of Dymem. Constructor performs graph analysis. Scheduler manages prefetch/offload operations. Allocator handles tensor allocation/deallocation.

can not serve the incoming request because they are distributed at two separate lists. Between these two fragmented spaces, the allocated tensor resides on the GPU longer than other tensors because multiple layers have dependencies on it. Fragmentation happens at the virtual address level. It is not allowed to modify the page tables on the GPU. And it is also impossible to move data around the GPU with negligible overhead. The best-fit algorithm requests an additional 400MB memory to satisfy the demand. When peak memory consumption is close to the GPU memory capacity, this fragmentation might impact the trainability of networks.

Furthermore, from Figure 8, we can learn that the memory usage of CONV, ACT, BN, and POOL layers can account for more than 90% of total usage. However, it is not fruitful to offload Dropout, Softmax, and FC layers because they only require less than 1% of the total memory. These layers require less memory but stay longer until no more dependency. It requires us to manage the allocation of these tensors carefully. Otherwise, it can cause further fragmentation. Based on the above observations, an efficient tensor placement policy should be proposed while considering both the varied data sizes and graph dependencies.

3 SYSTEM DESIGN AND IMPLEMENTATION

The design objective of our dynamic memory manager (Dymem) is to automatically manage the memory usage of DNNs while minimizing the overhead and maximizing the reduction of memory load. Dymem is a host-side runtime that interfaces with GPU to dynamically move, allocate, and release tensors. Figure 9 shows the



(a) Schema of Inception-v4 network. (b) Forward pass of Inception-v4.

Figure 10: Execution order for Inception-v4 network in the forward pass. l_i represents i_{th} layer. $l_0 \rightarrow \{l_1, l_2, l_3, l_4\}$ represents that layer l_1, l_2, l_3, l_4 have dependency on layer l_0.

overall system architecture. In this section, we first introduce how to perform graph analysis to construct a memory-efficient execution flow, particularly for nonlinear networks. Then, based on the results from the graph constructor, we propose a tensor scheduler to utilize dependency features to asynchronously offload/prefetch candidates with different variable sizes and resident duration. Lastly, we implement a unified GPU memory pool and propose a contiguity-conserving placement policy to allocate/deallocate the scheduled tensors.

3.1 Execution Graph Construction

Algorithm 1 Execution flow for nonlinear blocks.

```
1:  function flowConstruct(int layerId)
2:      if layerID == Null then
3:          return;
4:      end if
5:      refcnt++;
6:      if layerId.refcnt < prevLayer.refcnt then
7:          return;
8:      end if
9:      execFlow.push(layerId)
10:     L = layerId -> get-next();
11:     for l ∈ L do
12:         flowConstruct(l)
13:     end for
14: end function
```

Given a nonlinear network, we need a memory-efficient approach to set up the execution order. Since cuDNN [7] implements deep learning primitives at layer granularity, we use tensors as the basic scheduling unit. For basic networks, during the forward propagation, the results from $layer_{n-1}$ can be applied as the input for $layer_n$. The computation flow can be regarded as a sequential process. Only when the preceding layer is finished, then can it initiate the next layer's computation. This chain rule is similarly applied in the backward pass but in a reversed order. For networks with nonlinear blocks, there are nonlinearities such as one-to-many (fork) and many-to-one (join) connections. Depth-First-Search (DFS) algorithm is used to decide the execution sequences for these nonlinear dependencies, which is shown in Algorithm 1. Whenever there is a fork connection, DFS is applied to explore all the executable layers until it reaches the join connection in the nonlinear blocks, as

Table 1: The computation and communication time in residual block (ms).

Layer	1×1 CONV	3×3 CONV	Join
Computation	25	76	3
Communication	66	68	×

shown in lines 7 to 8. Figure 10(a) shows the schema for Inception-A blocks in the Inception-v4. The detailed execution order obtained by DFS is demonstrated in Figure 10(b). In this example, the Inception block should be propagated in four branches in both the forward and backward passes. In the forward pass, $l_0 \rightarrow \{l_1, l_2, l_3, l_4\}$ represents that output feature maps from layer l_0 should reside in the GPU memory until layers l_1, l_2, l_3, l_4 are executed because of dependencies. Similarly, during the backward pass, in the branch $l_8 \rightarrow l_7 \rightarrow l_4$, when l_8 is being executed, layer l_4 should be prefetched from CPU memory asynchronously based on DFS. The reason why DFS should be applied to construct the execution graph lies in two properties: First, DFS requires less memory space to reach the join connection node in the nonlinear blocks when exploring the traversal path. For example, the branch $l_4 \rightarrow l_7 \rightarrow l_8$ illustrates simple dependency, in which the corresponding data for those memory-intensive convolution layers can be released from GPU memory sequentially. Second, inside those nonlinear blocks, e.g., residual block and Inception grid, most layers are computation-intensive Convolution layers. The DFS can serialize the sequences of convolution layers in each branch, mostly.

3.2 Dependency Aware Memory Offloading and Prefetching

After obtaining the execution graph, Dymem automatically manages the offload and release operations for tensors so as to effectively improve the overlap ratio between communication and computation. We employ two separate *cudaStreams* to transfer tensors in/out of external memory asynchronously. $stream_{compute}$ interfaces to the cuDNN handle and sequences all the computations in the forward and backward pass. $stream_{memory}$ is responsible for the tensor placement, movement, allocation, and deallocation.

3.2.1 Memory Offload. During forward propagation, if $layer_n$ is available for offloading, Dymem first allocates a pinned memory region in the host via *cudaMallocHost*(), then $stream_{memory}$ can asynchronously swap feature maps from this layer via non-blocking memory transfer. When the asynchronous offload is completed, the *cudaEvent* is register to record this event. Because the input features for CONV, POOL and ACTV layers are read-only data structures, we can start the offload operation for these when they are being performed forward propagation. As for $stream_{compute}$, $layer_n$'s computation can be started as soon as $layer_{n-1}$'s computation is completed without waiting the completion of the offload operation of $layer_{n-1}$. Nonlinear blocks, e.g., Residual blocks, can benefit from this strategy because of the join operations do not necessarily wait for the completion of tensors transfer from 1×1 CONV and 3×3 CONV, which is illustrated from the Table 1. $stream_{compute}$ guarantees the completion of computation for $layer_n$ by using the *cudaStreamSynchronize*() API. When both of these two events for $layer_n$ are finished, a shared queue is used to record this tensor. The release of the tensors chosen for offloading from GPU is done when there is no dependency for these layers in the shared queue. An individual thread is launched to release the $layer_n$ from the GPU memory. At the end of the forward propagation, we synchronize $stream_{compute}$ and $stream_{memory}$ to make sure that $stream_{memory}$ has offloaded its feature maps. This safely ensures that all layers chosen to be offloaded are offloaded from GPU memory before the start of backward propagation, maximizing the memory saving and improving the performance greedily. However, there is an exception that the execution for the next layer has to be blocked if the available memory is not enough, waiting for the release for completed layers. In general, memory space is traded for performance in the forward pass.

3.2.2 Memory Prefetch. In the backward pass, prefetching the offloaded input feature maps back to GPU can be overlapped with the computation of backward propagation using *cudaMemcpyAsync*() as well. After an asynchronous transfer for $layer_n$ is completed, a *cudaEvent* is registered in the $stream_{memory}$, after which the computation can be started for this layer. The $stream_{compute}$ is synchronized with the offload event to guarantee that the computation can be safely launched with available input feature maps. Similar to the forward pass, we only synchronize $stream_{compute}$ and $stream_{memory}$ at the end of backward propagation before the next iteration. Instead of launching the prefetch operations in the reverse order simply, we have to consider the execution order and prefetch latency when searching for the optimal candidate layer. Another problem is that if the prefetched $layer_m$ is too far away from the overlapped $layer_n$, the memory saving benefit will be reduced because the prefetched data be reused immediately, wasting the GPU memory. Jointly considering the memory saving and prefetch latency, we propose an efficient searching algorithm to decide the layer to be prefetched, which is shown in Algorithm 2. Whenever there is a nonlinear block, we decide the preceding layers based on DFS, which is similar to the forward pass. After obtaining the layer, we restrict that no more than two Convolution layers residing in the GPU, as is illustrated in the line 11. This is because Convolution layers are computation intensive. Prefetching these layers too early will under-utilize the GPU resources. As long as it is not convolution layer and not available yet in the GPU memory, it can be chosen as the candidate, as is shown in line 14. This is because other layers require shorter computation time compared with Convolution layers. This feature can gain performance improvement because the prefetch latency can be well hidden by the computation time.

3.3 Contiguity-conserving Memory Management

In this section, we first define tensor mobility based on the varied data size and dynamic dependencies. Then we propose Group Tensors By Mobility (GTBM) as the placement policy to classify tensors before allocation. We further implement a unified memory pool, consisting of main space and incremental space, to host different tensors so as to achieve lower memory fragmentation.

Algorithm 2 Searching the candidate layer.

```
1: function searchPrefetchLayer(int layerId)
2:     n = 0;
3:     if layerId-> type == CONV then
4:         n++;
5:     end if
6:     next = flowConstruct(layerId).pop();
7:     while id do
8:         if next-> type == CONV && n < 2 then
9:             pf.push(id); n++;
10:            next-> pf = True;
11:        else if next-> of && !(next-> pf) && next-> type !=
       CONV then
12:            next-> pf = True;
13:            pf.push(next);
14:        end if
15:        next = flowConstruct(next).pop();
16:    end while
17: end function
```

3.3.1 Design Principles. In the operating system, the internal fragmentation is defined as the inability to satisfy an allocation request because a suitably large contiguous block of memory is not free even though enough memory may be free overall [11]. The scope of internal fragmentation not only depends on the layout of free memory nodes but the size of the request. Here we define the unusable free space term, U_f. It measures how much of the available free memory cannot be used to satisfy an allocation:

$$U_f(j) = \frac{TotalFree - \sum_{i=j}^{i=n} 2^i k_i}{TotalFree}, \tag{1}$$

in which j is the desired allocation (i.e., the size of the request is 2^j), $TotalFree$ is the number of free memory nodes, 2^n is the largest request allocation that can be satisfied, and k_i is the number of free memory nodes of size 2^i. A term of 0 implies there is no memory fragmentation. The term tending towards 1 implies high fragmentation, in which the request cannot be satisfied. Based on the analysis of the best-fit algorithm in Section 2.3.2, the memory fragmentation could increase if the contiguous and noncontiguous tensors are both allocated to a contiguous portion of the virtual address. If fragmentation happens, the memory pool has to grow its size so as to satisfy the demand. In order to decrease U_f during the training, we propose a contiguity-conserving allocation strategy. The core idea is to provide a soft guarantee that all of the tensors having the same dependencies or similar lifetime should be allocated in the same region.

3.3.2 Group Tensors By Mobility. Group Tensors By Mobility (GTBM) considers the address space as being split into three arenas. Tensors are placed such that each arena contains tensors of the same page mobility type. For our purposes, three mobility types are defined as below:

- **Movable tensors** are from those layers who have simple dependencies, i.e., one reference count. They can be released after the being propagated and the released space could coalesce with each other soon.
- **Temporary tensors** are those tensors that are known to exist for a very short period of time, such as fork connections or tensors waiting to be joined. These tensors exist longer than movable tensors and are not supposed to be mixed with them.
- **Reclaimed tensors** are tensors from layers that are not selected for offloading. These tensors require less memory capacity and could be reclaimed after being propagated.

The placement policy is to group tensors of the same mobility type within an arena of the matching type. The pseudo-code for the grouping process is summarized in Algorithm 3. The grouping procedure for Dropout, Softmax, and FC layers are shown in the line 5, which should be classified as reclaimed tensors. For the nonlinear block shown in Figure 10, in the forward pass, layer l_0 has multiple dependencies, which should be put into the temporary tensors group. But for layers l_5, l_2, and l_6, though they have only one reference, they should be grouped as the temporary tensors because their results will not immediately be concatenated, which is illustrated in the line 5. However, in the backward pass, they should be regarded as movable tensors because the results from layers l_5, l_2, and l_6 can be consumed immediately. The release operations will be initiated when the reference number of the currently processing layers has been decreased to zero. For example, after forward computation of l_9 is finished, the feature maps of layers l_5, l_2, and l_6 can be released from the GPU.

3.3.3 Unified Memory Pool. The memory manager is a host-side interface, serving as the GPU back-end. The memory space is divided into main and incremental areas. Inside the main space, the allocation can be started from both the low and high-end available addresses. For the memory operations, we employ the open-source asynchronous memory allocation/release API library provided by Nvidia CNMeM [1]. The allocation starting from the low end will use the default *cnmemMalloc*() API, following the default best-fit heuristic. For the allocation starting from high-end address, we implement a new *cnmemHighMalloc*() API, pointing the starting address to the high end. For the incremental space, we use the default *cudaMalloc*() to request a new GPU memory outside of the existing pool. Though this procedure will cause initialization overhead, it is negligible because of the minimal proportion of these layers in the neural networks.

For movable tensors, Dymem allocates tensor in the memory pool via *cnmemMalloc*() from the low-end available address. Since they have a simple dependency, i.e., one dependent layer, it only resides on the GPU for one step and then will be offloaded to the CPU memory. In the forward/backward pass, the released space in the low-end address can always be utilized by the coming layers if the communication is well hided or the subsequent layers. As for temporary tensors, Dymem places the coming tensors via *cnmemHighMalloc*() starting from the high-end available address. In this scenario, contiguity is conserved in the high-end address since all of the tensors from this group have noncontiguous usage. Inside the nonlinear blocks, the tensors that have higher reference counts are always allocated before the ones with lower references. So the used space can be deallocated in an order opposite to the allocation, resulting in minimal memory fragmentation. Regarding the incremental space, which hosts reclaimed tensors, because the

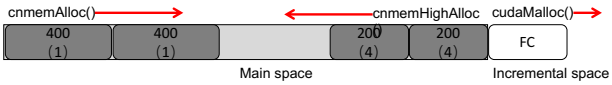


Figure 11: The workflow of the unified memory pool. Dark blocks represent main space and white blocks represent incremental space.

backward propagation follows the reversed order of forward pass, the allocation is from low to high-end while the deallocation is from the high to low-end, leading to no fragmentation.

The configuration of memory pool size is a trial-and-error process. To ensure the trainability of networks, Dymem initiates a large enough main space for the first iteration. Based on the aggregated consumption of memory, Dymem adjusts the memory pool size by removing the smallest squeeze gap between the low- and high-end regions in training, i.e., $mem\_size = (init - squeeze\_gap) + \beta$, in which $squeeze\_gap = \min\{highest\_avail - lowest\_avail\}$. The β is reserved space in case of fluctuation. If the training fails, an additional β is provisioned.

Algorithm 3 Group Tensors By Mobility.

```
1: function tensorGroup(int layerId)
2:     if layerId -> type == FC || layerId -> type == Softmax ||
   layerId -> type == Dropout then
3:         reclaim ← (layerId);
4:     else if layerId -> refcnt > 1 || !(layerId -> get-next()) then
5:         temporary ← (layerId);
6:     else
7:         movable ← (layerId);
8:     end if
9: end function
```

4 METHODOLOGY

4.1 Baselines

We choose vDNN [28] and SuperNeurons [38] as the baselines for performance comparisons. Regarding memory management, vDNN uses the default Nvidia CNMeM [1] library to allocate/deallocate tensors. SuperNeurons adopts a fast heap-based GPU memory pool utility. The core concept is to divide the preallocated pool into an allocated list and an empty list. For these two techniques, we implement the best-fit algorithm as the memory management policy. The execution order for the nonlinear network is not detailed in vDNN. So we adopt the same construction, DFS, for vDNN for comparison. We can only release tensors from GPU memory when there is no further reference in the forward or backward passes. As for the tensor scheduling policy, we implement the dynamic policy mentioned in the vDNN paper, which automatically decides the offloading layers employed to balance the trainability and performance of a DNN at runtime. As for SuperNeurons, we only implement the liveness analysis and unified tensor management components because recomputation for specific layers is not considered in our work.

4.2 Optimizing Convolution Algorithm

The speed of CONV layers significantly impacts the training performance. We implement a dynamic strategy to utilize the availability of the GPU memory pool. The dynamism can achieve a tradeoff between memory saving and performance gain. Since the allocation of convolution workspace does not affect the functionality of the training, so we prioritize the allocation for those required feature maps, gradients maps, and weights, etc. The runtime will profile the available memory space when it enters a new layer and increment the amount of GPU memory for workspace in a fine-grained granularity, i.e., 1MB. The runtime will stop requesting more GPU memory for workspace if it causes failure in training. Then, the corresponding decision is regarded as the optimal configuration for the current state. The baselines, vDNN and SuperNeurons, adopt the same dynamic strategy to achieve the balance between memory saving and performance gain. We also implement the memory-optimal algorithm as the baseline, in which no extra workspace memory is required for Convolution layers.

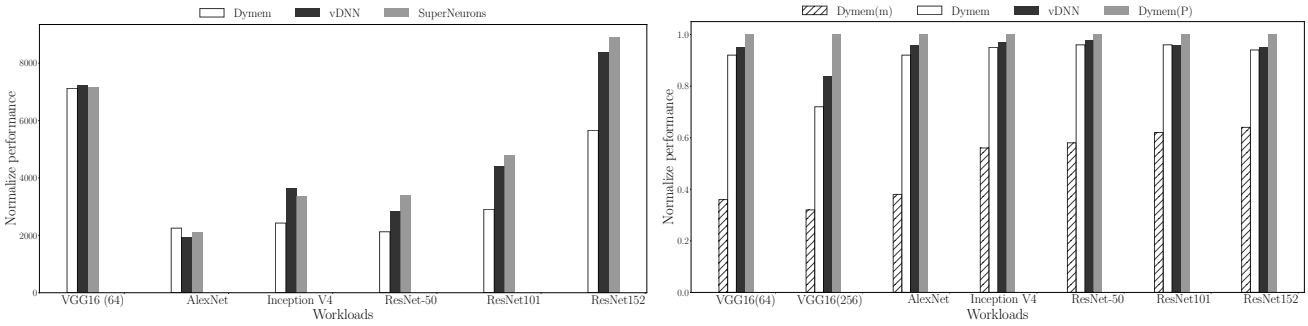
4.3 DNN Benchmarks

4.3.1 Linear Networks. First, we perform the evaluation compared with vDNN and SuperNeurons on linear networks, VGG-16 and AlexNet. We use the same training configurations as the published paper [32][22]. For AlexNet, we configure the batch size as 256. there are 23 forward steps and 23 backward steps. VGG-16 is one of the largest and deepest DNN architecture, which has 16 CONV and 3 FC layers. It requires substantial memory capacity for trainability. To ensure the trainability, we configure the batch size as 64 and 128. We evaluate the performance regression of the end-to-end training and the peak memory consumption for one iteration. Since SuperNeurons requests a fixed and large enough memory pool, we measure memory usage in terms of aggregated memory usage during the runtime.

4.3.2 Nonlinear Networks. We further perform the evaluation against vDNN and SuperNeurons on two representative nonlinear networks, ResNet [16] and Inception V4 [34]. Specifically, we implement the basic Residual block, which has two 3×3 convolutional layers with the same number of output channels. Each convolution layer is followed by a batch normalization layer and a ReLU activation function. Then, the skip connection joins the output from two convolution layers with the original input before the final activation function. We also evaluate the performance using various depths for ResNet, e.g. ResNet-32, ResNet-50, ResNet-101 and ResNet-152. The difference among these networks is the number of residual blocks. Since vDNN does not report the evaluation results for ResNet, we follow the implementation from Torch [27] to implement the memory management policy because it is adopted as the baseline for vDNN. For all of the above benchmarks, we use the image dataset CIFAR-10 [21].

5 EVALUATION

Our experimental evaluation is performed on GeForce GTX TITAN X with 12 GB GPU memory. The machine has 3.4 GHz Intel i7-3770 CPU (20 cores) and 32 GB CPU memory. The GPU communicates with CPU via a PCIe switch, which has 16GB/sec data transfer



(a) Maximum memory usage.

(b) Normalized convolutional performance.

Figure 12: Overall GPU memory usage and normalized performance of convolution layers. The batch size of Inception V4 is 128. The batch size for ResNet with different depths are 100. Dymem(m) and Dymem(p) represent running Dymem with memory-optimal and performance-optimal algorithms.

bandwidth. The machine is installed with Ubuntu-16.04, CUDA 9.0, CuDNN 7.0, and g++ 5.4.0.

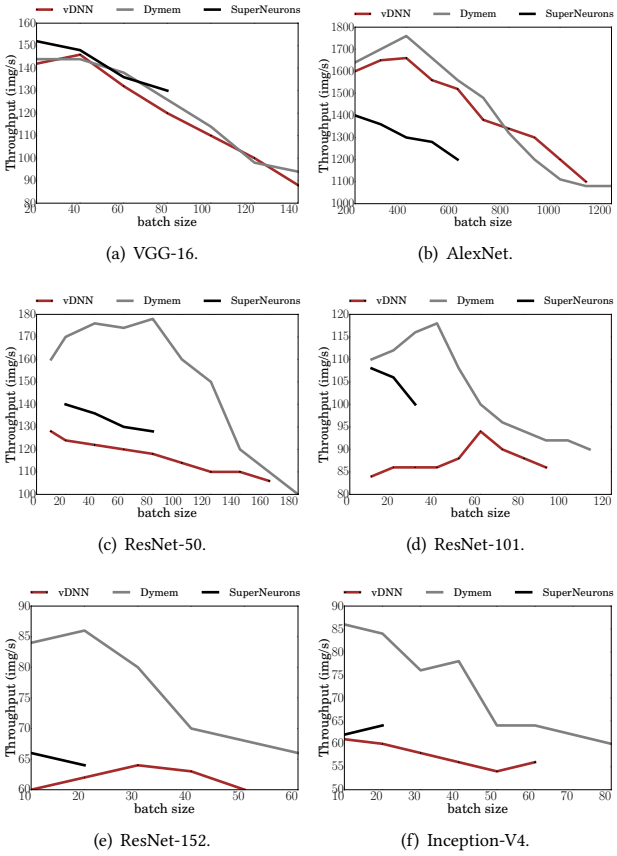
5.1 Reduction on GPU memory usage

Figure 12(a) summaries the aggregated memory usage among Dymem, vDNN, and SuperNeurons for different DNNs. Because all of these three approaches apply a layer-wise memory allocation policy, the GPU memory usage during forward/backward pass will fluctuate depending on the tensors chosen for offloading/prefetching. So we use the aggregated memory to represent the maximum allocated GPU memory for one entire iteration, which is the minimum requirement to enable the trainability of the networks. From the results of VGG-16 and AlexNet, we can see that there is no difference between these three strategies. For such linear networks, layers are propagated sequentially. Dymem falls back to the same best-fit algorithm, which is adopted by SuperNeurons and vDNN as well. For VGG-16, because convolution layers dominate the training process, leading to no difference between vDNN and SuperNeurons. Dymem even requires nearly 250MB more memory capacity than vDNN for AlexNet. Because Dymem aggressively prefetches more layer's data structure than vDNN, trading the memory space for performance improvement, which is detailed in Section 5.2. For non-linear networks, ResNet-50, ResNet-101 and ResNet-152 vary their network depths by changing the combinations of for-loop residual blocks. We can see that when the depth of ResNet is increased, memory consumption is not linearly increased. The performance of Dymem shows considerable scalability for different depths of networks compared with vDNN and SuperNeurons. Specifically, for Inception V4, the maximum memory footprint is reduced from 3650MB to 2527MB, resulting in 31% memory saving compared with vDNN. SuperNeurons shows better performance than vDNN to handle the tensor scheduling for Inception V4, whose inception branches are much more complex than ResNet. However, Dymem can still achieve 28% memory saving compared with SuperNeurons by minimizing the GPU memory fragmentation caused by the simple best-fit policy.

Figure 12(b) reports the performance of the convolution algorithms of Dymem and vDNN. The performance of convolution algorithms can better represent the average utilization of the GPU DRAM during the runtime. For comparison, we implement Dymen with both memory-optimal algorithm and the performance-optimal convolution algorithm as the baseline, in which performance-optimal algorithm is configured to supply with enough memory to run the fastest convolution algorithms. To demonstrate the impact of stressed memory capacity, we especially study VGG-16 running with 256 batch sizes. Since is impossible to train VGG-16 with this configuration on GeForce GTX TITAN X, we employ the layer-by-layer strategy to ensure trainability. The training time that occurred in all convolution layers is accumulated to represent the overall performance because the memory capacity only affects the convolutional performance. As shown in the figure, we can see that the memory-optimal algorithm could result in nearly 60% performance loss on average compared with the performance-optimal algorithms. It is normal because no extra memory space is sacrificed for performance, closing the gap between the memory-optimal and performance-optimal configurations. Both Dymem and vDNN achieve well balancing between memory usage and the overall performance. From the results, we can see that Dymem and vDNN reach an average of 95% and 97% throughput of the performance-optimal. However, when the batch size of VGG-16 is configured to 256, the average throughput is decreased to 84% running in the vDNN. The performance is worse running in Dymem, which is 72% of the performance-optimal setting. Because Dymem prioritizes functionality over performance. For VGG-16, especially in the backward pass, Dymem prefetch more Convolution layers following Pooling layers than vDNN, resulting in less available GPU DRAM for workspace allocation. The minor performance loss from the algorithm could be made up of the benefit from the overlapping between computation and communication, as illustrated in Section 5.2.

5.2 End-to-end throughput evaluation

Figures 13 present the end-to-end training throughput comparison of Dymem to vDNN and SuperNeurons. The training throughput is measured by the number of processed images per second. We vary the batch sizes for different DNNs and compare the corresponding



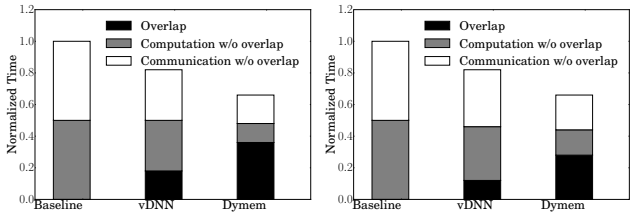
(a) VGG-16. (b) AlexNet.

(c) ResNet-50. (d) ResNet-101.

(e) ResNet-152. (f) Inception-V4.

Figure 13: End-to-end evaluation on throughput of different DNN models.

throughput. For linear networks, VGG-16, and AlexNet, there is not much performance improvement over vDNN and SuperNeurons. Because these networks are composed of simple and sequential layers. For example, VGG-16 consists of 16 convolution layers, which are computation-intensive. The computation time is always longer than the transfer. The propagation computation dominates the total delay. As a result, there is no much performance benefit by removing the layer-by-layer synchronization barriers. In some cases, for linear networks, we can see that SuperNeurons perform better than Dymem and vDNN. Because SuperNeurons only offloads convolution layers, avoiding the communication overhead. However, for both linear and nonlinear networks, when the batch sizes are increased, SuperNeurons cannot train these networks because of the limited memory availability. For nonlinear networks, the results consistently demonstrate the leading throughput on ResNet-50, ResNet-101, ResNet-152, and Inception V4. The largest throughput improvement comes from ResNet-50, running with batch size 100, which achieves up to 42% compared with vDNN. The performance largely results from the improved communication/computation ratio. This is because Dymem could better utilize the overlap of communication and computation among layers. We can also observe that the throughput has slowly deteriorated by increasing



(a) Inception V4. (b) ResNet-32.

Figure 14: Execution time decomposed into the overlapped time, the non-overlapped communication time, and the non-overlapped computation time in two networks.

batch size. This is because GPU memory can only accommodate less network layer with wider networks, resulting in the decreased communication/computation ratio. Less layer overlapping requires the growing communications in more frequent tensor swapping between CPU and GPU. Then, the runtime has to constantly offload the current layer before proceeding to the next one.

5.3 Efficiency of dependency-aware swapping

Figure 14 plots the breakdown of the normalized execution time of two representative nonlinear networks, Inception V4 and Resnet-32. These two networks are training on Dymem and vDNN with the memory-optimal configuration to avoid the impact from the speedup of Convolution. Specifically, the time is decomposed into the overlapped time, the non-overlapped communication time, and the non-overlapped computation time. In this experiment, the baseline only uses one stream, which restricts that the computation and offload/prefetch in both the forward and backward passes are executed sequentially. We also configure the memory-optimal algorithm for these three experiments, so as to avoid the impact of the dynamics in the convolution layers. As shown in the figures, the overlapped time in the baseline is zero since the communication and the computation are performed sequentially. The layer by layer strategy adopted by vDNN can overlap the communication with the computation to some extent by 18% and 12% for Inception V4 and ResNet, respectively. The overlapped time in Dymem is longer than that in the vDNN, showing that a more aggressive batching strategy is more effective in terms of performance. As a result, compared with the baseline, Dymem can achieve nearly up to 46% reduction on the execution time.

5.4 Efficiency of the contiguity-conserving policy

To study the efficiency of the proposed tensors placement policy, we analyze the step-by-step memory usage of ResNet-32 in one iteration running with GTBM and the default best-fit algorithms, which is shown in Figure 15. Here, the "used" memory includes the allocated and the free but unusable memory nodes. For vDNN, the "used" memory counts the memory nodes from the lowest allocated address to the highest available address in the memory pool. For Dymem, it additionally counts the used memory nodes from the high-end in the main space and the incremental space. In this

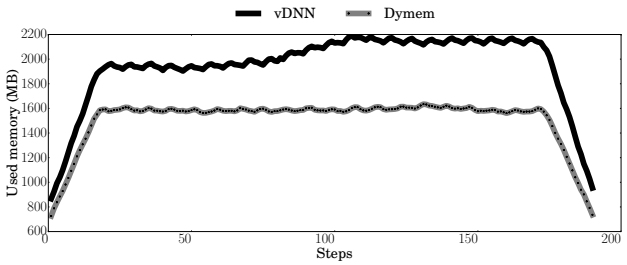


Figure 15: The moving average memory usage of ResNet-32 in one iteration.

Table 2: The failure rate of training with different parameter configurations.

Memory size (MB)	100	300	500
Training failure (%)	5	0.9	0

experiment, we run ResNet-32 without batch normalization under the batch size of 100. Specifically, it consists of 15 residual blocks. In each residual block, there is one join and one fork connections. Between these two connections, there are two branches, including shortcut and residual connections. The residual connection is composed of two Convolution and one Activation layers. After the join connection, the result will be fed into one Activation layer. One iteration requires 190 steps in the forward and backward passes. From the result, we observe vDNN requests more GPU memory at the end of each residual block, i.e., the join connection. Though the occupied memory nodes have been released, the fragmented but unfit memory nodes cause a waste of resources. Compared with vDNN, the unifies memory pool can reduce the fragmented nodes by reorganizing the placement. The peak memory usage occurs in the first residual block in the backward pass, which requires nearly 2500MB GPU memory for vDNN. But for Dymem, the peak memory requirement is 1854MB. The proposed unified memory pool can reduce memory fragmentation.

5.5 Sensitivity analysis of the approximation

To quantify the effect of different β, we configure β varying from 100 to 500MB and run ResNet-101. We initiate a large enough memory pool, i.e., 10G (another 2GB for incremental space), for the first iteration. Then we approximate the suitable memory pool size based on the profiled data and β. If the training fails, the current iteration is restarted and assigned with additional β memory space. We obtain the average failure rate, which is shown in Table 2. Since the DNN training remains the same execution sequences, mostly, the approximated pool size is sufficient to serve the memory request. In this experiment, we can see that the optimal configuration should be 300MB, considering the memory-saving and network trainability. The limitation of this profiling-based method is that it requires additional memory and CPU capacities. For different networks with different parameters, the approximation should be repeated to find out the suitable configuration.

6 RELATED WORK

A variety of solutions have been proposed to overcome the GPU DRAM shortage for training deep neural networks. Lossy encodings have been rigorously studied in the domain of DNN inference and training. Network pruning techniques [14] are proposed to reduce the model redundancy so as to reduce the memory consumption. Huffman encoding [15], quantization [37] and reduced precision [20] are also studied to reduce the model size (weights). Network compression [10] is another important approach to reduce the memory usage of DNNs. However, they provide limited opportunities for memory saving because weights are not a major contributor to the total memory requirement. Moreover, some of these techniques, e.g., reduced precision, might result in loss of prediction accuracy if not carefully tuned. Gist [19] investigates approaches to optimize the memory usage for input feature maps, which are the dominated source of memory footprint in DNNs training. This work is orthogonal to our approach.

SuperNeurons [38] and Chen et al. [6] introduce a recomputation strategy to trade computation for memory saving. They consider recomputing the output from selected layers in the forward again in the backward pass instead of prefetching them from CPU memory or keeping them on the GPU DRAM. However, it requires high-level semantics on the computation graph. The overhead from training cannot be negligible because the largest layers require longest recomputation time. This approach works well for linear networks but fail to exploit the memory saving opportunities. Yet, this technique can be applied in conjunction with our work for specific layers, e.g., batch normalization.

Model parallelism is a straightforward strategy to train deep and large networks. DistBelief[9] distributes the network across multiple nodes by partitioning the network so that each node only holds a part of the the network. Tofu [39] enables the training of very large DNN models by partitioning a dataflow graph of tensors across multiple GPU devices. However, these techniques require huge intra-network communications for synchronization. Another approach is to place different layers on different devices via heuristics [30] or machine learning [33]. However,operator placement is not suitable for DNNs with a deep stack of layers. Data parallelism can achieve better performance by adding more GPUs. But the powerful GPU could suffer from sub-linear scaling because of stragglers and costly network transfer across workers. An effective and simple approach to reduce the memory requirement for DNN training is to reduce the minibatch size. However, it slows down the training process because a smaller batch size could result in GPU underutilization [12].

vDNN [28] also proposes a prefetching and offloading technique to transfer the data between CPU and GPU memory so as to fit the large networks in the GPU memory. It tries to overlap communication with computation by asynchronously swapping the data between CPU and GPU via PCIe. However, it requires a synchronization barrier between communication and computation for each layer, which is safe but inefficient. It ignores the benefit from those layers, e.g., POOL and ACT layers, which are cheap to compute. It is a waste to wait for the slow transfer of PCIe bus. SuperNeurons [38] also consider memory swapping but restricts to swap only convolution layers. None of these works take the GPU memory

fragmentation into account when allocating the tensors on GPU. vDNN++ [31] proposes an approximated memory pool to reduce the GPU memory fragmentation, which is limited to linear neural networks only.

7 CONCLUSION AND FUTURE WORK

With the deep neural networks going wider and deeper, there is a need to effectively schedule GPU memory for DNN training to overcome the insufficient capacity. In this paper, we focus on memory management for the training of nonlinear DNNs. We propose the runtime to adopt the layer-wise graph analysis and dependency-aware offloading/prefetching strategy to improve the throughput of DNN training. Furthermore, we design a Group Tensors By Mobility (GTBM) placement policy to allocate tensors on the proposed unified memory pool for data structures with varied data sizes and dynamic dependencies, so as to reduce the GPU memory fragmentation in the training. Compared with the state-of-art vDNN, for linear networks, there is no much performance difference. For nonlinear networks, our proposed solution can achieve memory saving for Inception V4 by up to 31%. The proposed dependency-aware approach can improve the end-to-end training throughput for ResNet-50 by up to 42%. The experiments also show that Dymem can achieve better scalability for nonlinear networks with various network depths. Currently, the proposed solution only supports GPU memory optimization for neural networks with a static dataflow graph and a fixed shape of the input, i.e., DNN. In the future, we are going to extend our work to support dynamic neural networks, whose data samples have variable shapes and the computation graph topology depends on input or parameter values, e.g., RNN and LSTM.

8 ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful suggestions and comments. This work was supported by the NSF grant CCF-1908843.

REFERENCES

[1] [n.d.]. Nvidia CNMeM. https://github.com/NVIDIA/cnmem, 2018.
[2] [n.d.]. NVIDIA V100 TENSOR CORE GPU. https://www.nvidia.com/en-us/data-center/v100/, 2020.
[3] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. 2015. Comparative study of deep learning software frameworks. *arXiv:1511.06435* (2015).
[4] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. 2016. Comparative study of caffe, neon, theano, and torch for deep learning. (2016).
[5] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. 2018. Benchmark analysis of representative deep neural network architectures. *IEEE Access* (2018), 64270–64277.
[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).
[7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv:1410.0759* (2014).
[8] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *Journal of machine learning research* 12 (2011), 2493–2537.
[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.
[10] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. 2014. Compressing deep convolutional networks using vector quantization. *arXiv:1412.6115* (2014).
[11] Mel Gorman and Patrick Healy. 2008. Supporting superpage allocation without additional hardware support. In *Proc. of ACM ISMM*. 41–50.
[12] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv:1706.02677* (2017).
[13] Stephen Grossberg. 1988. Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural networks* 1, 1 (1988), 17–61.
[14] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proc. of IEEE ISCA*, Vol. 44. 243–254.
[15] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv:1510.00149* (2015).
[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proc. of IEEE CVPR*. 770–778.
[17] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. 2012. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine* 29, 6 (2012), 82–97.
[18] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. 2014. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv:1404.1869* (2014).
[19] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient data encoding for deep neural network training. In *Proc. of IEEE ISCA*. 776–789.
[20] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proc. of ACM ICS*. 1–12.
[21] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.
[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
[23] Quoc V Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y Ng. 2011. On optimization methods for deep learning. In *Proc. of IEEE ICML*.
[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.
[25] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. 1990. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*. 396–404.
[26] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. 8024–8035.
[28] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proc. of IEEE Micro*. IEEE, 1–13.
[29] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.
[30] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv:1701.06538* (2017).
[31] SB Shriram, Anshuj Garg, and Purushottam Kulkarni. 2019. Dynamic Memory Management for GPU-Based Training of Deep Neural Networks. In *Proc. of IEEE IPDPS*. IEEE, 200–209.
[32] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556* (2014).
[33] I Sutskever, O Vinyals, and QV Le. 2014. Sequence to sequence learning with neural networks. (2014), 3104–3112.
[34] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.
[35] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proc. of IEEE CVPR*. 1–9.
[36] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. 2013. Deep neural networks for object detection. In *Advances in neural information processing systems*. 2553–2561.
[37] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. 2011. Improving the speed of neural networks on CPUs. (2011).
[38] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: dynamic GPU memory management for training deep neural networks. In *Proc. of PPoPP*. 41–53.
[39] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proc. of ACM Eurosys*.