moDNN: Memory Optimal DNN Training on GPUs

Xiaoming Chen[1,2], Danny Z. Chen[1], Xiaobo Sharon Hu[1]
[1]Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, USA
[2]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
Email: chenxiaoming@ict.ac.cn, {dchen, shu}@nd.edu

***Abstract*—Graphics processing units (GPUs) are widely adopted to accelerate the training of deep neural networks (DNNs). However, the limited GPU memory size restricts the maximum scale of DNNs that can be trained on GPUs, which presents serious challenges. This paper proposes an moDNN framework to optimize the memory usage in DNN training. moDNN supports automatic tuning of DNN training code to match any given memory budget (not smaller than the theoretical lower bound). By taking full advantage of overlapping computations and data transfers, we have developed heuristics to judiciously schedule data offloading and prefetching, together with training algorithm selection, to optimize the memory usage. We further introduce a new sub-batch size selection method which also greatly reduces the memory usage. moDNN can save the memory usage up to 50×, compared with the ideal case which assumes that the GPU memory is sufficient to hold all data. When executing moDNN on a GPU with 12GB memory, the performance loss is only 8%, which is much lower than that caused by the best known existing approach, vDNN. moDNN is also applicable to multiple GPUs and attains 1.84× average speedup on two GPUs.**

I. Introduction

In the past decade, deep neural networks (DNNs) have shown great promise in numerous practical applications, such as image classification [1], object detection [2], natural language processing [3], etc. Many studies have demonstrated that increasing the scale of DNNs can greatly improve the accuracy of DNN results [4], [5]. A variety of DNNs (e.g., VGG-Net [6], GoogLeNet [7], residual network (ResNet) [8], etc.) have been developed recently. Some DNNs have hundreds of layers. Such large-scale DNNs raise significant computational challenge for training. Thanks to the tremendous computational efficiency offered by graphics processing units (GPUs), training DNNs has become feasible. However, the GPU memory size is still an obstacle which restricts the scale of DNNs that can be trained on GPUs. State-of-the-art DNNs can use tens of gigabytes which greatly exceed the memory size of current high-end GPUs. If a DNN cannot be fit into GPUs' memory, one has to reduce the DNN's size, which can cause undesirable accuracy loss, or use more GPUs, which significantly increase the hardware cost.

There have been several approaches aiming to reduce the memory usage of DNNs: network pruning, precision reduction, output re-computation, and memory management. Network pruning tries to prune small weights without losing accuracy much [9]. As a result, both performance and memory usage are improved. However, since weights consume little memory for large-scale DNNs, pruning weights is not effective for reducing memory usage. Recent studies exploited the use of fixed-points or binary numbers to boost the performance of DNNs [10]–[13], which also bring significant memory usage savings. However, lower precisions can lead to accuracy loss. The output re-computation approach discards some outputs when the memory is insufficient, and re-computes them when required [14]. This approach incurs high performance degradation. Rather than always keeping data in GPU memory, NVIDIA proposed vDNN, which adopted the idea of data offloading and prefetching [15]. Data that are not being used are offloaded to the host memory, and they are prefetched into GPU memory when required. vDNN does not incur any accuracy loss, but is rather brute force (simply offloading the outputs of all layers or all convolution (CONV) layers).

This paper aims to tackle the memory challenge of DNN training on GPUs by proposing an moDNN (*memory optimal DNN training on GPUs*) framework. Like vDNN [15], moDNN adopts the idea of data offloading and prefetching. We introduce heuristics to judiciously schedule data transfers and select CONV algorithms such that both memory usage and performance are optimized. We also propose a novel sub-batch size selection method which cooperates with data transfer scheduling to further reduce memory usage without impacting the accuracy. moDNN can automatically produce training code for *any* given DNN and memory budget without losing accuracy, while achieving superior performance by ensuring that the memory usage tightly fits the memory budget.

II. Preliminaries

1) DNN Training: DNNs are commonly trained by backward propagation (BP) together with an optimization method (e.g., gradient descent) [16]. The purpose of training is to minimize the error as a function of the weights of a DNN. Typically, a complete training process includes many iterations; an iteration includes a forward propagation (FP) pass and a BP pass using a batch (i.e., a subset) of training samples. An FP pass computes the DNN's output from the first layer to the last layer. A BP pass propagates the error in the opposite direction to update the weights. Details of DNN training can be found in related books/tutorials, e.g., [16].

2) Task and Data Flow Graph: The training process of one iteration is like a U-shape curve. We can build a task and data flow graph (TDFG), which is a directed acyclic graph (DAG), to depict all the data dependencies during one iteration's training. Fig. 1 illustrates the TDFG for an example 4-layer CONV neural network (CNN). The TDFG shows all involved tasks in one iteration except the last step, weight update (i.e., adding $\Delta \mathbf{W}^l$ to $\mathbf{W}^l$ for all layers). For all CONV related tasks, a workspace may be needed, since fast CONV

This work was supported by the National Science Foundation (NSF) under grants CCF-1217906, CNS-1629914, CCF-1617735 and CCF-1640081, and the Nanoelectronics Research Corporation (NERC), a wholly-owned subsidiary of the Semiconductor Research Corporation (SRC), through Extremely Energy Efficient Collective Electronics (EXCEL), an SRC-NRI Nanoelectronics Research Initiative under Research Task IDs 2698.004 and 2698.005.

978-3-9819263-0-9/DATE18/©2018 EDAA

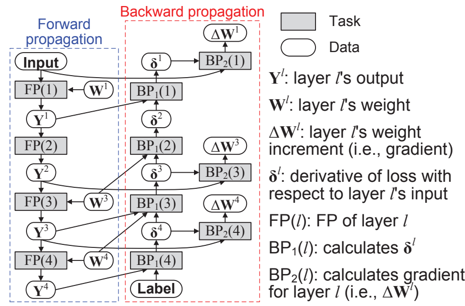

Fig. 1: A TDFG representation of an example 4-layer CNN (CONV-pooling-CONV-fully connected).

algorithms (e.g., the Winograd algorithm [17]) typically need some temporary workspace.

In the TDFG, the sizes of all data blocks are known from the given DNN and batch size. The execution times of tasks and the transfer times of data blocks can be measured via profiling before training. Since a training process usually includes thousands of or even more iterations, adding a profiling step has negligible effect on the overall performance. All the tasks are topologically sorted and will be executed following this order in training. For state-of-the-art DNNs, we have observed that weights (i.e., $\Delta\mathbf{W}$s and $\mathbf{W}$s) typically consume a small fraction of the total memory usage. So in moDNN, weights always reside in the GPU memory and are not offloaded.

III. moDNN Framework

A. Problem Definition

Suppose we are given a DNN with training parameters and one or more GPUs, and the GPU memory size or a user-specified memory budget (if given) is **insufficient** to hold all data involved during training. The purpose of moDNN is to *make the DNN trainable on the given GPU platform, keeping the given training parameters (e.g., the batch size) unchanged, such that the performance (i.e., the total training time) is optimized without affecting accuracy*.

At the highest level, moDNN adopts the idea of out-of-core [18] algorithms. A relevant problem is the red-blue pebble game [19], which determines the lower bound on data transfers for such algorithms. The red-blue pebble game was studied only for some special problems. Finding the optimal solution for a general DAG has not been studied, and some variants are NP-complete [20]. Since finding the globally optimal solution for our problem seems computationally prohibitive, we aim to develop effective heuristics.

B. Framework Overview

Our moDNN framework is shown in Fig. 2. It builds on three key techniques. 1) Data offloading and prefetching. Modern GPUs support overlapping computation and data transfer, which enables offloading unused data to the host memory with negligible cost. 2) Sub-batch size selection. Reducing the batch size is a natural idea to reduce memory usage. However, a smaller batch size may impact accuracy. In moDNN, we partition a batch into multiple sub-batches and accumulate the gradients from all sub-batches, resulting in unchanged accuracy. 3) CONV algorithm selection. CONV

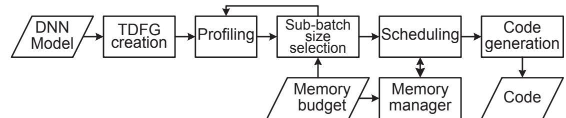

Fig. 2: Framework of moDNN.

can be implemented by different methods on GPUs, such as general matrix multiplication (GEMM), implicit GEMM [21], the Winograd algorithm [17], fast Fourier transform [22], etc. Faster CONV algorithms (e.g., the Winograd algorithm [17]) typically need some workspace so the CONV algorithm must be carefully selected. Among these techniques, the first is out-of-core related, and the other two are DNN related. So moDNN explores both architecture- and application-level features to optimize the memory usage for DNN training.

moDNN first builds the TDFG based on the given DNN. It then determines the sub-batch size based on profiling results and the memory budget. Profiling is to measure the tasks' execution times and data blocks' transfer times. Since profiling needs the sub-batch size to get accurate measurements, the two steps are done iteratively until the sub-batch size converges. Then, moDNN computes a schedule for data offloading and prefetching together with CONV algorithm selection. The scheduling goal is to minimize the finish time of the TDFG, while the memory usage never exceeds the memory budget. Since the TDFG structure does not change, moDNN produces a static schedule for the given DNN. The last step generates the training code based on the schedule. The memory manager shown in Fig. 2 is used to perform virtual allocation and free operations during scheduling. We use a linked list based approach [23] to achieve this goal.

Finding the globally optimal solution for the three key techniques discussed earlier is not trivial. Considering one GPU, as all tasks are executed sequentially, any choice (i.e., which data to be offloaded, when to offload and prefetch, which algorithm is selected, etc.) made for one task affects the choices of all the future tasks. Suppose task t has N_t choices, we have $\prod_{t\in\mathbb{T}} N_t$ choices in total, where $\mathbb{T}$ is the task set. Obviously, the search space is exponentially large.

Due to the exponential nature, we resort to develop heuristic algorithms to find the solution. However, the interaction among the three techniques, which reflects **the conflicts between performance and memory usage**, is still a challenge. Using a larger sub-batch size increases the parallelism, and hence improves the performance. However, it leads to more memory usage which can invoke more offloading and prefetching operations and reduce the opportunity of using faster CONV algorithms. Using faster CONV algorithms may require more memory, which reduces the opportunity of prefetching data for future tasks. Considering these conflicting directions, the starting point of our heuristic algorithms is to comprehensively consider both the benefit and penalty of possible choices such that good tradeoffs are achieved.

Although our idea of offloading and prefetching seems similar to vDNN [15], moDNN has three distinct advantages over vDNN. 1) We introduce an automatic sub-batch

size selection method, which cooperates with data transfer scheduling and CONV algorithm selection to optimize the memory usage. 2) We judiciously select data to offload by comprehensively considering the benefit and penalty, while vDNN simply offloads the outputs of *all* layers or *all* CONV layers. 3) CONV algorithms are also selected by considering both the benefit and penalty, while vDNN simply selects the fastest possible algorithm for each task. The three new techniques result in both reduced memory usage and increased performance compared with vDNN.

IV. moDNN Algorithms

This section describes the moDNN algorithms. For illustration purposes, we use one GPU as an example. But moDNN can be easily applied to multiple GPUs by partitioning a batch into equal-sized parts. Due to the page limit, we cannot present all details and choose to focus on high-level ideas to make the paper self-contained. We also omit the discussion on code generation since it is relatively straightforward.

A. Sub-batch Size Selection

A task in the TDFG can be executed on GPU only if the GPU memory is sufficient to hold its input data, output data and temporary workspace during execution (for CONV related tasks). Since the memory usages of a task's inputs and outputs are proportional to the batch size, decreasing the batch size is a natural idea to reduce memory usage. In order to attain an equivalent training corresponding to the user-specified batch size, after a batch is partitioned into multiple sub-batches, the training of one batch needs to be done by multiple rounds, and the gradients are accumulated from all the sub-batches.

We first determine the theoretical lower bound on memory usage as a function of the sub-batch size. For an individual task t, the minimum memory requirement for sub-batch size b is (excluding the weights which always reside in GPU memory)

$$M_{\min}(t,b)=\sum_{d\in\mathbb{I}(t)}\text{Size}(d,b)+\text{Size}(O(t),b)+\text{Size}(WS(t),b) \quad (1)$$

where $\mathbb{I}(t)$, $O(t)$ and $WS(t)$ are the input data set, output data and workspace of task t, respectively. The memory usages of $O(t)$ and $\mathbb{I}(t)$ are proportional to b but $WS(t)$ may not. Implicit GEMM [21] requires zero workspace so it is treated as the baseline algorithm. From the above analysis, the theoretical lower bound of the memory requirement for executing the entire TDFG corresponds to the case that the sub-batch size is 1 and we always use the baseline algorithm, i.e.,

$$M_{\min} = S_W + \max_{t\in\mathbb{T}}\{\sum_{d\in\mathbb{I}(t)}\text{Size}(d,1) + \text{Size}(O(t),1)\} \quad (2)$$

where S_W is the total weight memory size. moDNN can generate a proper schedule for the given DNN as long as the user-specified memory budget is not smaller than $M_{\min}$.

We now discuss how to select the sub-batch size based on the user-specified memory budget. Intuitively, the sub-batch size should be large enough to fully utilize the GPU resources. However, if the sub-batch size is selected such that the memory budget is used up for the baseline algorithm, we have no memory space to use faster CONV algorithms which need additional workspace. Thus the sub-batch size should not be too large. To strike a good balance, the workspace size should be considered when selecting the sub-batch size.

In order to collect the workspace size and performance of all CONV algorithms for each task, we conduct a profiling step on the given GPU platform. Profiling in turn requires the sub-batch size to get accurate measurements. To deal with this dependency, we iteratively do profiling and sub-batch size selection (see Fig. 2). The sub-batch size is iteratively updated by the following method until it converges:

$$b = (M - S_W)/\max_{t\in\mathbb{T}}\{M_{\min}\ (t,b)\} \quad (3)$$

where M is the user-specified memory budget. $M_{\min}(t,b)$ is calculated by Eq. (1) where the workspace size of the fastest algorithm of task t is used for Size$(WS(t),b)$.

B. Scheduling Approach and Algorithm Selection

Scheduling determines the optimal data transfers and CONV algorithms, while satisfying the given memory budget. The objective is to minimize the finish time of the TDFG, which is achieved through (1) maximally overlapping data transfers with computations, (2) minimizing offloading, (3) judiciously prefetching future data, and (4) selecting the optimal CONV algorithms. Algorithm 1 summarizes the scheduling approach. It consists of three major steps for each task: (1) preparing the input and output data, which may require offloading certain data, (2) selecting the optimal CONV algorithm, if the task is CONV related, and (3) determining the data to be prefetched for future tasks. Algorithm 1 simulates the actual execution process. The notion of "current time" refers to the time in the simulated execution process.

Algorithm 1 simulates the executions of all the tasks in a topological order. For task t that is to be executed, we first prepare its input and output data (lines 3-8). If some input data are not in the GPU memory, we need to first allocate memory spaces and then load the data to the GPU memory. We also allocate space for the output at the same time (line 3). If the allocation fails, we try to offload some data that are not being used (lines 4-5). If an available offloading scheme (which specifies which data to be offloaded) cannot be found, it must be caused by fragmentations in the memory space, because the sub-batch size selection method guarantees that the memory requirement of any task does not exceed the memory budget. Defragmentation (offloading all data and then reloading required data) can solve this problem. Once we have allocated sufficient memory spaces, input data are loaded into the GPU memory (line 6). The delay caused by offloading and data loading is added to the current time (line 8).

If task t is a CONV related task, the best CONV algorithm is selected by considering both the benefit and penalty (lines 9-14). The benefit is the time saved by a faster algorithm. Due to the workspace required by the faster algorithm, more offloading operations may be required, and some prefetching operations for future tasks have to be delayed. They are both included in the penalty. The best algorithm is the one with the maximum gain (benefit minus penalty) (line 11). The delay caused by offloading is added to the current time to get the start time of task t (line 14).

Algorithm 1: Data transfer scheduling and CONV algorithm selection.

1 $T = 0$; // Current time
2 **for** task $t = 1, 2, \cdots, |\mathbb{T}|$ in topological order **do**
  // Prepare for task t's inputs and output
3 Allocate memories for any $d \in \mathbb{I}(t)$ that is not in GPU memory and for $O(t)$;
4 **if** allocation fails **then**
5   Find an offloading scheme (do defragmentation when necessary) and then re-allocate;
6 Load any $d \in \mathbb{I}(t)$ that is not in GPU memory;
7 **if** offloading and data loading cause delay T_{cost} **then**
8   $T += T_{\text{cost}}$;
  // Select CONV algorithm for task t
9 **if** task t is CONV related **then**
10   **for** all possible algorithms for task t **do**
11   Select the algorithm with the maximum gain (defined in Eq. (5));
12   Allocate workspace for the selected algorithm alg;
13   **if** offloading for workspace allocation causes delay $T_{\text{off,alg}}$ **then**
14    $T += T_{\text{off,alg}}$; // T is now the start time of task t
  // Prefetch data for future tasks
15 **for** $s = t+1, t+2, \cdots, |\mathbb{T}|$ **do**
16   **if** $\mathbb{I}(s)$ or $O(s)$ cannot be allocated when executing task s **then**
17   Break;
18   **else if** at least one $d \in \mathbb{I}(s)$ that is not in GPU memory **then**
19   **if** prefetching for task s should start now **then**
20   Allocate memory for prefetching;
21   **if** allocation fails **then**
22   **if** an offloading scheme can be found **then**
23   Do offloading, re-allocation and prefetching for task s;
24   **else**
25   Break;
26   **else**
27   Break;
28 $T = FT(t) = T + T_{\text{alg}}(t)$; // T is now the finish time of task t. $T_{\text{alg}}(t)$ is the execution time of algorithm alg of task t.
29 **for** $d \in \mathbb{I}(t)$ **do**
30   **if** d will no longer be used **then**
31   Free d; // Free data that will not be used

Next, prefetching data is done for future tasks (lines 15-27). For each future task s ($s > t$), we first predict if its input and output data can be allocated when the time is just before executing task s. If the prediction fails, prefetching is stopped (lines 16-17) for the following reason. When executing task s, if its input and output data cannot be allocated, we may need defragmentation, leading to useless prefetching operations. If prefetching is predicted to be useful, we determine if the prefetching for task s should start now (before executing task t) by considering whether it will cause delay if the prefetching starts later (line 19). If we decide to prefetch for task s, we then allocate memory spaces (including offloading attempt when the allocation fails) and perform the prefetching (lines 20-23).

After prefetching for future tasks is scheduled, task t is executed by updating its finish time (line 28). We finally free any data that will no longer be needed (lines 29-31).

As can be readily seen, the success of the moDNN scheduling algorithm hinges on finding good offloading schemes, determining when to prefetch what, and selecting the optimal CONV algorithms. We elaborate these aspects below. Our discussion builds on the fact that in practice two CUDA streams $\texttt{Strm}_C$ and $\texttt{Strm}_D$ are used to execute computations and data transfers, respectively, so that they are overlapped.

1) Offloading: Offloading is invoked when a memory allocation fails. In this situation, we try to offload some data to

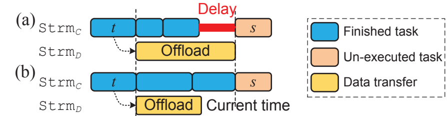

Fig. 3: Offloading example (the offloaded data block is generated by task t). (a) Delay is caused. (b) No delay is caused.

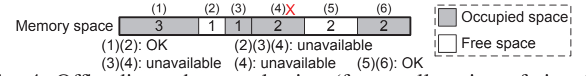

Fig. 4: Offloading scheme selection (for an allocation of size 4).

obtain a contiguous space that is not smaller than the requested size. Since weights cannot be offloaded in moDNN, we can select $\mathbf{Y}$s and $\boldsymbol{\delta}$s (see Fig. 1) to offload to vacate their spaces. Once a data block has been offloaded, it does not need to be offloaded again and we just free it in future offloading operations, as the host memory already has its copy.

Offloading may cause delay to the next task to be executed (and to all future tasks). Fig. 3 illustrates two cases, in which Fig. 3a has delay overhead and Fig. 3b does not. During the scheduling process, we record the estimated start and finish times for all the tasks and data transfers in $\texttt{Strm}_C$ and $\texttt{Strm}_D$ respectively. So the delay to the next task caused by an offloading operation can be easily estimated.

In vDNN [15], one data transfer is overlapped with one computation task. Actually this is an unnecessary requirement which increases the delay overhead and synchronization cost. In moDNN, we explore more flexible data transfers such that one data transfer can be overlapped with multiple computation tasks, and vice versa (see Figs. 3 and 5), to reduce the delay overhead and synchronization cost.

Finding an offloading scheme: We use a linked list to represent the memory spatial distribution, where each node corresponds to an occupied or free segment in the GPU memory, and an occupied segment represents a data block in the TDFG. Searching for an offloading scheme is to find a set of contiguous segments whose total size is not smaller than the allocation size. This is done by traversing the linked list. Each node in the linked list can be considered as the first block to be offloaded. Contiguous data blocks at subsequent addresses are added to the offloading list until the total size is sufficient. In order to minimize unnecessary data transfers, we set two criteria to determine whether a data block can be offloaded. If a data block (a) will be used immediately by the next task, or (b) has not been used since its generation (by a task) or its latest prefetching, it should not be offloaded or freed. We traverse all possible offloading schemes and select the scheme with the lowest delay overhead. It is possible that no available offloading scheme is found. This is caused by fragmentations in the memory space. In this situation, we offload all data and then reload the required data for the next task (i.e., defragmentation). Fig. 4 shows an example for offloading scheme selection, where data block (4) is not allowed to be offloaded. For an allocation of size 4, we find two available offloading schemes (1)(2) and (5)(6). The final scheme is determined based on the delay overhead.

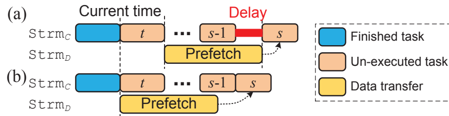

Fig. 5: Determining the start time of prefetching. (a) Delay is caused by late start. (b) No delay is caused.

2) Prefetching: Once a data block has been offloaded, it needs to be prefetched when it is required. The prefetching operation should start before the task (say, task s) which needs the data, otherwise task s would be delayed. However, prefetching should not start too early, since the prefetched data block consumes memory that is unnecessary for tasks executed before s. Considering these factors, the start time of a prefetching operation should be carefully determined. Fig. 5 explains how to determine the start time of a prefetching operation. The current time is just before the execution of task t. Whether the prefetching operation for task s should start now is determined by the following criterion. We predict whether it will cause delay to task s if the prefetching is scheduled after task t is finished, i.e.,

$$T + T_{\text{alg}}(t) + \sum_{\substack{d \in \mathbb{I}(s) \text{ and} \\ d \text{ is not in GPU memory}}} T_{\text{trans}}(d) > EST(s) \quad (4)$$

where $T_{\text{alg}}(t)$ is the execution time of algorithm alg of task t, $T_{\text{trans}}(d)$ is the transfer time of data d (same for offloading and prefetching), and $EST(s)$ is the expected start time of task s. If (4) holds, prefetching for task s should start now, i.e., before task t (line 19 in Algorithm 1).

Another issue is that we need to predict if the prefetching for task s is useful (line 16 in Algorithm 1). Before executing task s, we need to allocate memory spaces for its output and input data which are not in the GPU memory. If the allocation fails, we need to do offloading or defragmentation. If the prefetched data block is freed by defragmentation, it is a useless prefetching. To avoid such waste, we need to predict if defragmentation will be conducted when task s is to be executed. This is achieved by predicting the memory distribution at a future time. It is difficult to get an exact prediction because the current choice will impact the scheduling for future tasks. But we can conduct an approximate prediction by guessing the scheduling for the tasks before task s. Details are omitted due to space limit.

3) Algorithm Selection: Different CONV algorithms (e.g., GEMM, implicit GEMM [21], the Winograd algorithm [17], fast Fourier transform [22]) have different performance and memory requirements, which are collected in the profiling step. Implicit GEMM requires zero workspace so it is the baseline. Other algorithms may be faster but require some workspace. Always using the fastest algorithm for every task is not the best choice, because the fastest algorithm may need more workspace, which increases data offloading operations for the current task, and reduces the opportunity of prefetching data for future tasks. In theory, it can impact all the future tasks, yielding an exponential searching space. For heuristics, we only look ahead one future task in moDNN.

For task t, we check all possible algorithms one by one by considering both the benefit and penalty. We consider the following "gain" for algorithm alg:

$$\Delta T_{\text{alg}}(t) = T_{\text{base}}(t) - T_{\text{alg}}(t) - T_{\text{off,alg}}(t) - T_{\text{pre,alg}}(s) \quad (5)$$

where s is the nearest future task which needs to prefetch data, $T_{\text{base}}(t)$ is the execution time of the baseline algorithm, $T_{\text{off,alg}}(t)$ is the offloading time needed by the workspace allocation for algorithm alg, and $T_{\text{pre,alg}}(s)$ is the delayed prefetching time for task s. $T_{\text{off,alg}}(t)$ is estimated using the method shown in Fig. 3. $T_{\text{pre,alg}}(s)$ is estimated by considering the memory requirement of the prefetching for task s and the workspace size of algorithm alg of task t. Namely, we can predict how many prefetching operations for task s have to be delayed due to the workspace allocation for algorithm alg of task t. To select the optimal algorithm for task t, we just select the algorithm with the maximum $\Delta T_{\text{alg}}(t)$ defined in Eq. (5).

V. Experimental Evaluation

moDNN is implemented using C++ and CUDA [24]. Experiments are conducted on one and two NVIDIA K40m GPUs. Each K40m GPU has 12GB memory. We test VGG-16 [6], VGG-19 [6], VGG-101 (created by increasing the CONV layers of VGG-19), ResNet-34 [8] and a fully CONV network (FCN) [25] to evaluate moDNN. The batch sizes of the five DNNs are 256, 256, 128, 256 and 128, respectively. In this section, the "ideal case" refers to the assumption that the GPU memory is sufficient to hold all data involved during training. The ideal case performance is evaluated by accumulating all individual tasks' execution times.

A. Memory Requirement Reduction

Fig. 6 compares the memory requirement. The five DNNs all need more than 12GB in the ideal case, so they cannot be trained directly on one K40m. Even for latest GPUs with 24GB memory, the three VGG networks still cannot be trained. However, moDNN greatly reduces the memory requirement. The theoretical lower bound corresponds to the sub-batch size 1. Compared with the ideal case, the lower bound is reduced by 50× on average. Even for VGG-101 which requires nearly 100GB by the ideal case, the lower bound of moDNN is only 1.4GB, which can easily fit into almost all low-end GPUs.

B. Performance Comparison

Fig. 7 compares the training time (for one batch) and transferred data size for one K40m execution. The average performance degradation (i.e., training time increase) of moDNN

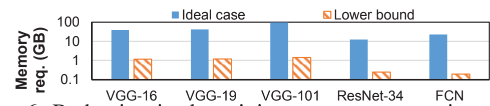

Fig. 6: Reduction in the minimum memory requirement.

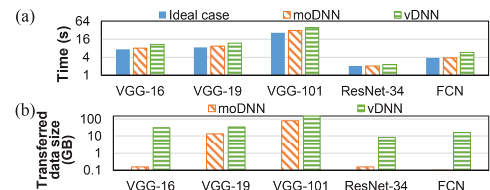

Fig. 7: Comparisons under 12GB memory budget. (a) Performance comparison. (b) Transferred data size comparison.

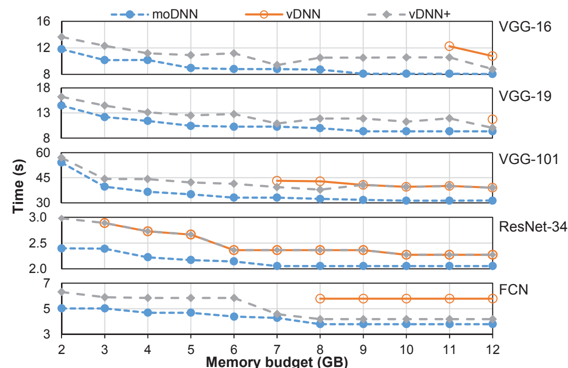

Fig. 8: Performance comparison under different memory budgets.

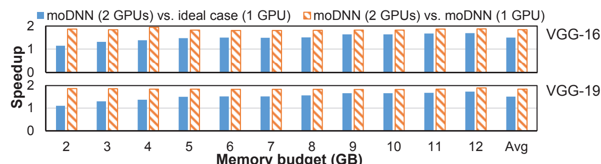

Fig. 9: Results of moDNN on two GPUs.

and vDNN under 12GB budget are 8% and 40%, respectively. Due to the judicious selection of data to be offloaded, moDNN reduces the transferred data size by 5× on average.

An important feature of moDNN is the ability to fit any user-specified memory budget which is not smaller than the theoretical lower bound. Fig. 8 shows the performance results under different memory budgets (for one GPU). Since vDNN does not have the feature of sub-batch size selection, it fails when the memory budget is smaller than some certain value. We extended vDNN to vDNN+ which employs our sub-batch size selection method to reduce memory usage. As shown in Fig. 8, sub-batch size selection also benefits to vDNN, as vDNN+ attains better performance than vDNN under the same budget. moDNN is always better than both vDNN and vDNN+. This is mainly due to our new data transfer scheduling and CONV algorithm selection methods.

C. Results on Two GPUs

Fig. 9 shows the results of VGG-16 and VGG-19 on two GPUs. Compared with the ideal case on one GPU, moDNN on two GPUs achieves 1.5× speedup on average. Compared with moDNN on one GPU under the same memory budget, moDNN on two GPUs achieves 1.84× speedup on average.

There are two bottlenecks for multiple GPUs. 1) Low data transfer speed. Since multiple GPUs share the same PCIe bus, if they invoke data transfers simultaneously, the speed is lowered. This increases the data transfer overhead which impacts the performance. 2) Gradient accumulation. The last step of each iteration is to accumulate the gradients from all GPUs, where transferring the gradients among GPUs has high cost. This is actually a common challenge for multiple GPUs. For two GPUs, we have observed that the overheads of the two bottlenecks are not too high. For more GPUs, however, we need better solutions to address the bottlenecks.

VI. Conclusion

In this paper, we proposed moDNN to optimize the GPU memory usage for DNN training. By taking full advantage of both architecture- and application-level features, we developed three key techniques: data offloading and prefetching, sub-batch size selection, and CONV algorithm selection, enabling automatic tuning of DNN training code to match any user-specified memory budget. Experimental results showed that moDNN reduces the minimum memory requirement by up to 50×. When executing moDNN on a GPU with 12GB memory, the performance loss is only 8% on average. moDNN achieves better performance than vDNN under the same memory budget. The significant reduction in GPU memory usage opens up the opportunity for training DNNs on low-cost GPU platforms.

References

[1] A. Krizhevsky *et al.*, “ImageNet Classification with Deep convolutional Neural Networks,” in *NIPS*, 2012, pp. 1097–1105.
[2] R. Girshick *et al.*, “Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation,” in *CVPR*, 2014, pp. 580–587.
[3] Y. Goldberg, “A primer on neural network models for natural language processing,” *Journal of Artificial Intelligence Research*, vol. 57, pp. 345–420, 2016.
[4] S. Bahrampour *et al.*, “Comparative Study of Deep Learning Software Frameworks,” *arXiv:1511.06435*, Nov. 2015.
[5] J. Ngiam *et al.*, “On optimization methods for deep learning,” in *ICML*, 2011, pp. 265–272.
[6] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv:1409.1556*, 2014.
[7] C. Szegedy *et al.*, “Going deeper with convolutions,” in *CVPR*, 2015, pp. 1–9.
[8] K. He *et al.*, “Deep residual learning for image recognition,” in *CVPR*, 2016, pp. 770–778.
[9] S. Han *et al.*, “Learning Both Weights and Connections for Efficient Neural Networks,” in *NIPS*, 2015, pp. 1135–1143.
[10] S. Gupta *et al.*, “Deep Learning with Limited Numerical Precision,” in *ICML*, 2015, pp. 1737–1746.
[11] D. D. Lin *et al.*, “Fixed Point Quantization of Deep Convolutional Networks,” in *ICML*, 2016, pp. 2849–2858.
[12] M. Rastegari *et al.*, “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks,” in *ECCV*, 2016, pp. 525–542.
[13] M. Courbariaux *et al.*, “Binarized neural networks: Training deep neural networks with weights and activations constrained to + 1 or -1,” *arXiv:1602.02830*, 2016.
[14] A. Gruslys *et al.*, “Memory-efficient backpropagation through time,” in *NIPS*, 2016, pp. 4125–4133.
[15] M. Rhu *et al.*, “vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design,” in *MICRO*, Oct 2016, pp. 1–13.
[16] I. Goodfellow *et al.*, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org
[17] A. Lavin and S. Gray, “Fast Algorithms for Convolutional Neural Networks,” in *CVPR*, 2016, pp. 4013–4021.
[18] J. S. Vitter, “External memory algorithms and data structures: Dealing with massive data,” *ACM Comput. Surv.*, vol. 33, no. 2, pp. 209–271, Jun. 2001.
[19] J.-W. Hong and H. T. Kung, “I/O Complexity: The Red-blue Pebble Game,” in *STOC*, 1981, pp. 326–333.
[20] Q. Liu, “Red-Blue and Standard Pebble Games: Complexity and Applications in the Sequential and Parallel Models,” Ph.D. dissertation, Massachusetts Institute of Technology, 2017.
[21] S. Chetlur *et al.*, “cuDNN: Efficient Primitives for Deep Learning,” *arXiv:1410.0759*, vol. abs/1410.0759, 2014.
[22] M. Mathieu *et al.*, “Fast Training of Convolutional Networks through FFTs,” *ArXiv: 1312.5851*, 2013.
[23] A. Silberschatz *et al.*, *Operating System Concepts*, 4th, Ed. Addison-wesley Reading, 1998.
[24] “CUDA C Programming Guide.” [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[25] J. Long *et al.*, “Fully convolutional networks for semantic segmentation,” in *CVPR*, 2015, pp. 3431–3440.