moDNN: Memory Optimal Deep Neural Network Training on Graphics Processing Units

Xiaoming Chen, *Member, IEEE*, Danny Ziyi Chen, *Fellow, IEEE*, Yinhe Han, *Member, IEEE*, and Xiaobo Sharon Hu, *Fellow, IEEE*

Abstract—Graphics processing units (GPUs) have been widely adopted to accelerate the training of deep neural networks (DNNs). Although the computational performance of GPUs has been improving steadily, the memory size of modern GPUs is still quite limited, which restricts the sizes of DNNs that can be trained on GPUs, and hence raises serious challenges. This paper introduces a framework, referred to as moDNN (memory optimal DNN training on GPUs), 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 develop new heuristics to judiciously schedule data offloading and prefetching transfers, together with convolution algorithm selection, to optimize memory usage. We further devise a new sub-batch size selection method which also greatly reduces memory usage. moDNN can save memory usage up to 59×, compared with an ideal case which assumes that the GPU memory is sufficient to hold all data. When executing moDNN on a GPU with 12 GB memory, the training time is increased by only 3 percent, which is much shorter than that incurred by the best known approach, vDNN. Furthermore, we propose an optimization strategy for moDNN on multiple GPUs again by utilizing the idea of overlapping data transfers and GPU computations. The results show that 3.7× speedup is attained on four GPUs.

Index Terms—Deep neural networks, graphics processing units, memory usage

1 Introduction

In the past decade, deep learning has become a very powerful paradigm and made tremendous advances. Deep neural networks (DNNs) have shown great promise in numerous machine learning applications, such as image processing [1], object detection [2], speech recognition [3], natural language processing [4]. A variety of DNNs have been developed (e.g., VGG-Net [5], GoogLeNet [6], U-Net [7], residual networks (ResNet) [8], fully convolutional networks (FCN) [9], etc.), that target different applications. A number of software frameworks have also been developed to implement deep learning, e.g., Caffe [10], TensorFlow [11], Theano [12], Torch [13], etc., which have greatly facilitated the development and applications of DNNs.

Many studies have demonstrated that increasing the scale of neural networks (NNs) can significantly improve the accuracy of NN results (e.g., [14], [15]). These results have largely promoted the investigation of scaling up NNs. Some state-of-the-art DNNs have hundreds of or even over 1000 layers. Such large-scale DNNs have raised significant challenges on conducting DNN training efficiency. Thanks to the tremendous computational efficiency offered by graphics processing units (GPUs), training large-scale DNNs has become feasible in recent years. Nowadays, almost all the existing deep learning frameworks support DNN training on GPUs, by invoking GPU libraries such as cuBLAS [16] and cuDNN [17] based on the compute unified device architecture (CUDA) [18].

Although the computational performance of current GPUs has been increasing steadily, the GPU memory size is still a major obstacle which restricts the maximum scale of DNNs that can be trained on GPUs. During DNN training, we need to store the inputs, weights, activations, temporary data, and any workspace in the GPU memory, posing a high memory usage requirement. The memory usage of state-of-the-art DNNs can easily reach tens of or even over 100 gigabytes, which greatly exceed the memory size of current high-end GPUs. If a DNN cannot fit into the given GPUs' memory, usually one has to reduce the scale of the network (e.g., using a network with fewer layers and/or parameters), which can cause undesirable accuracy loss.

A number of approaches have been investigated to reduce memory usage for DNN training: network pruning [19], [20], precision reduction [21], [22], [23], [24], output re-computation [25], [26], static memory allocation [27], batch partitioning [10], [28], and out-of-core training [29]. Among them, network pruning and precision reduction can lead to accuracy loss, and output re-computation can lead

- *X. Chen is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China. E-mail: chenxiaoming@ict.ac.cn.*
- *D.Z. Chen and X.S. Hu are with the Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556. E-mail: {dchen, shu}@nd.edu.*
- *Y. Han is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China. E-mail: yinhes@ict.ac.cn.*

Manuscript received 30 Jan. 2018; revised 14 June 2018; accepted 14 Aug. 2018. Date of publication 23 Aug. 2018; date of current version 13 Feb. 2019. (Corresponding author: Xiaoming Chen.)
Recommended for acceptance by M. Kandemir.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2866582

to high performance (i.e., training time) degradation. The last three approaches do not incur any accuracy loss, where out-of-core training offers higher memory usage reduction than static memory allocation. Following the idea of out-of-core training, NVIDIA recently proposed vDNN [29]. Data that are not being used are offloaded to the host memory, and 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 convolutional layers). Batch partitioning is very efficient for memory usage reduction, but existing DNN frameworks which support batch partitioning just leave the setting of the partitioning to users, which cannot guarantee the best partitioning.

This paper aims to tackle the memory challenge of DNN training on GPUs by proposing a *memory optimal DNN training* framework for GPUs, referred to as moDNN. moDNN not only enables the training of much larger-scale DNNs on a single GPU, but also helps reduce the memory quota on multiple GPUs. Like vDNN [29], moDNN is also based on the general concept of out-of-core training. However, in moDNN, we have designed new heuristics to judiciously schedule data transfers and select convolution algorithms such that both memory usage and performance are optimized. We also adopt the idea of batch partitioning to cooperate with data transfer scheduling to further reduce memory usage without affecting the accuracy. Different from the existing batch partitioning implementations [10], [28], moDNN automatically selects the sub-batch size such that both memory usage and performance are optimized. With batch partitioning, moDNN is able to handle any user-specified batch size, as long as the memory budget is not smaller than the theoretical lower bound of the memory requirement. By integrating these techniques, 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. All the above techniques are applicable to both single-GPU and multiple-GPU systems.[1]

moDNN has been implemented in an in-house DNN framework. A number of experiments based on VGG-Nets [5], ResNets [8] and an FCN [9] have been conducted. Experimental results show that moDNN can save memory usage up to 59×, compared with an ideal case which assumes that the GPU memory is sufficient to hold all data. When executing moDNN on a GPU with 12 GB memory, the performance is degraded by only 3 percent, which is much shorter than that incurred by vDNN. moDNN achieves 3.7× speedup on four GPUs, compared with the ideal case on a single GPU.

2 Related Work

There is a wide range of approaches aiming to reduce memory usage for DNN training. They can be generally classified into six categories: network pruning, precision reduction, output re-computation, static memory allocation, batch partitioning, and out-of-core training.

Network Pruning. Network pruning tries to prune insignificant weights without degrading accuracy much [19], [20]. By choosing a proper pruning strategy, most of the small weights can be pruned without significant loss of accuracy. As a result, both the performance and energy efficiency can be improved. However, we have observed that weights only account for a small fraction of the total memory usage for large-scale DNNs, so pruning weights is not very effective for reducing memory usage, especially for large-scale DNNs.

Precision Reduction. Many recent studies have exploited the use of fixed-point or binary representations instead of floating-points to boost the performance of DNNs [21], [22], [23], [24], which also lead to significant memory usage savings. However, the accuracy is only verified for the studied DNNs and there is no theoretical guarantee that the accuracy will not be affected much with lower-precision numbers for all DNNs. In other words, lower precisions may lead to severe accuracy loss for other DNNs.

Output Re-Computation. While the above methods are primarily targeted at performance improvement, which also help reduce memory usage of DNNs, some studies have considered how to directly optimize memory usage. The output re-computation approach discards some layers' outputs when the memory is insufficient, and re-computes them when required [25], [26]. In other words, this approach sacrifices training time to improve memory usage, hence can incur high performance degradation.

Static Memory Allocation. MXNet [31] adopts a static memory allocation method to reduce memory usage [27]. It has a number of fixed-size buffers and uses a graph-coloring algorithm [32] to assign data to buffers based on the live intervals of data. It can result in redundant memory requirement, because the memory requirement is the total size of all the buffers, and the buffers cannot be fully filled at the same time. In addition, data with long live intervals must reside in the GPU memory during their live intervals, resulting in wasted memory consumption. Thus, this method is not very effective to reduce memory usage.

Batch Partitioning. Some DNN frameworks (e.g., [10], [28]) support partitioning a training batch into multiple sub-batches, such that the memory requirement is reduced by approximately a factor of the number of sub-batches. However, these DNN frameworks leave the partitioning to users, which cannot guarantee optimal partitioning. Furthermore, since current DNN frameworks typically put all required data on GPU, the sub-batch size tends to be small for large-scale DNNs, leading to performance degradation due to lowered GPU resource utilization.

Out-of-Core Training. Rather than always keeping the required data in the GPU memory, NVIDIA proposed vDNN [29], which adopted the idea of data offloading and prefetching. This method utilizes the feature of modern GPUs that computations and data transfers can be overlapped. In vDNN, data that are not being used are offloaded to the host memory, and are prefetched into GPU memory before they are used. vDNN does not incur any accuracy loss, and the performance loss is somewhat small, but simply offloading the outputs of all layers or all convolutional layers is certainly not the best solution.

1. A preliminary version of this work was published in Design, Automation and Test in Europe 2018 [30]. This work in addition proposes an improved sub-batch size selection method and an optimized extension for multiple GPUs.

3 Preliminaries

In this section, we introduce some preliminaries of this work, including some basics of DNN training and the GPU memory management approach that we use.

3.1 DNN Training

DNNs are commonly trained by a backward propagation (BP) algorithm together with an optimization method [33] (e.g., gradient descent). The purpose of DNN training is to minimize the error as a function of the weights of the 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., subset) of training samples. Note that besides gradient descent, batch-based training is also used in other DNN training algorithms (e.g., conjugate gradient [34]). moDNN is applicable to all batch-based training methods and we just use gradient descent, the most popular DNN training method, as an example to describe the methodologies of moDNN.

An FP pass computes the DNN's output from the first layer to the last layer. The FP computation of one layer (say, layer l) can be described by

$$\mathbf{Z}^l = g^l(\mathbf{X}^l, \mathbf{W}^l), \mathbf{Y}^l = f^l(\mathbf{Z}^l), \tag{1}$$

where $\mathbf{X}$ and $\mathbf{Y}$ are the input and output, respectively, $\mathbf{Z}$ is an intermediate variable, and $\mathbf{W}$ is the weight. g performs some operation on the input and weight (e.g., for convolutional and fully-connected layers, g computes inner products, and for pooling layers, g downsamples the input). f is an activation function. Once the FP pass is finished, one can calculate the error E for each sample in the training batch by comparing the DNN's output with the ground truth (i.e., label).

A BP pass propagates the error in the opposite direction (from the last layer to the first layer) to update the weights. The BP computation of one layer is divided into three steps. The first step (BP_1, error back propagation) calculates the derivative of the error with respect to each layer's $\mathbf{Z}$, which is denoted by $\boldsymbol{\delta}$. $\boldsymbol{\delta}$ is an "error sensitivity" term that measures how much that layer is responsible for any errors in the DNN's output. $\boldsymbol{\delta}^l$ is calculated from the higher layers' $\boldsymbol{\delta}$'s according to the chain rule

$$\boldsymbol{\delta}^l = \frac{\partial E}{\partial \mathbf{Z}^l} = \sum_{s \in \mathbb{L}} \frac{\partial E}{\partial \mathbf{Z}^s} \cdot \frac{\partial \mathbf{Z}^s}{\partial \mathbf{Y}^l} \cdot \frac{\partial \mathbf{Y}^l}{\partial \mathbf{Z}^l} = \frac{df^l}{d\mathbf{Z}^l} \cdot \sum_{s \in \mathbb{L}} \boldsymbol{\delta}^s \cdot \frac{\partial g^s}{\partial \mathbf{Y}^l}, \tag{2}$$

where $\mathbb{L}$ is a set that contains those immediate successor layers taking layer l's output as an input. The second step (BP_2) calculates the weight increment (i.e., the gradient) for each layer, which, according to the gradient descent method [35], is expressed as

$$\Delta \mathbf{W}^l = -\eta \cdot \frac{\partial E}{\partial \mathbf{W}^l} = -\eta \cdot \boldsymbol{\delta}^l \cdot \frac{\partial g^l}{\partial \mathbf{W}^l} \tag{3}$$

where η is the learning rate. Equation (3) is for one sample. Since we use a batch of training samples in each iteration, the gradient should be averaged across the batch, i.e.,

$$\Delta \mathbf{W}^l = -\frac{\eta}{N} \sum_{n=1}^{N} \boldsymbol{\delta}_n^l \cdot \frac{\partial g^l(\mathbf{X}_n^l, \mathbf{W}^l)}{\partial \mathbf{W}^l}, \tag{4}$$

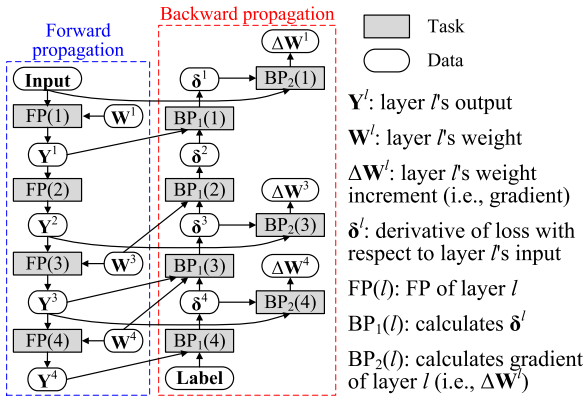


Fig. 1. A TDFG representation of an example 4-layer CNN (convolutional layer, pooling layer, convolutional layer, and fully-connected layer).

where N is the batch size. After the first and second steps of all layers are finished, the last step (BP_3) is performed for all layers to accumulate the weight increments

$$\mathbf{W}^l += \Delta \mathbf{W}^l. \tag{5}$$

3.2 Task and Data Flow Graph

The training process of one iteration is like a U-shape curve. In other words, during FP, all layers are computed in the forward order, and during BP, all layers are computed in the reversed order. We can build a task and data flow graph (TDFG), which is a directed acyclic graph (DAG), to depict all the data dependencies during the training of one iteration. Fig. 1 illustrates the TDFG for an example 4-layer convolutional neural network (CNN). The TDFG depicts all the detailed data dependencies at the task and data block level. The BP_3 tasks (weight update, corresponding to (5)) are not drawn in this TDFG to avoid forming cycles. As mentioned in Section 3.1, weight update is performed after all the tasks in the TDFG are finished, hence omitting the BP_3 tasks does not affect the dependencies. For convolution related tasks, a temporary workspace may be needed (not drawn in Fig. 1), since fast convolution algorithms (e.g., the Winograd algorithm [36]) typically need some workspace. By utilizing TDFGs, moDNN is able to handle any complex dependencies in DNN training.

To understand why offloading and prefetching can reduce memory usage of DNN training, note that actually we do not need to always keep any involved data in the GPU memory during training. All the tasks in a TDFG are topologically sorted and will be executed sequentially on GPU following this order in training. This means that, if a task is to be executed, we only need to store its input data, output data and temporary workspace in the GPU memory. Other data are not required. This gives us an opportunity that when a task is executing, if we have some free memory, we can load the input data of future tasks to the GPU memory, overlapping with the execution of the task. On the other hand, if a task's output will not be used for a long time, it can be offloaded to the host memory to vacate some space. The offloading operation can also be overlapped with executions of tasks.

In the TDFG, the sizes of all data blocks are known from the given DNN and batch size. To produce an optimal schedule, we also need the execution times of the tasks and the transfer times of the data blocks. They are measured by

conducting a profiling step 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. For large-scale DNNs, we have observed that weights (i.e., $\Delta\mathbf{W}$'s and $\mathbf{W}$'s) typically consume a very small fraction of the total memory usage. Thus in moDNN, weights always reside in the GPU memory and are not offloaded.

3.3 GPU Memory Management

Conventionally, CUDA memory management functions `cudaMalloc` and `cudaFree` are called to allocate and free GPU memories. By calling native CUDA functions, however, we cannot control the distribution of data in the memory space. This is not desirable since we seek to optimize memory usage. Instead, we have implemented a simple tool to manage GPU memory. This memory manager performs virtual allocation and free operations during the scheduling process. Our moDNN framework records all the allocated memory addresses (i.e., offsets) for generating the training code. Before training, a single GPU memory space (i.e., a memory pool) is allocated. The recorded offsets are added to the head address of the pool to generate the actual addresses during training.

Our memory manager utilizes the conventional linked list based implementation [37] to allocate and free memories. A doubly-linked list is used to store the spatial distribution of the memory space. The nodes in the linked list store the starting addresses, sizes, and states (occupied or free) of the segments in the memory space. Allocation/free operations insert/delete nodes into/from the linked list. There are different memory allocation strategies [38]. In this paper, considering the fact that some data blocks are of the same size (e.g., $\mathbf{Y}$ and $\boldsymbol{\delta}$ in the same layer), we adopt a two-step allocation method. For a memory allocation request, we first search for a free block whose size is exactly equal to the requested size, which helps reduce "holes" (i.e., fragmentations) in the memory space. If this operation fails, we then allocate the first found free block that is big enough to hold the requested size. Note that allocation and free operations are only invoked during scheduling. After a schedule is produced, all the offsets are determined.

4 The moDNN Framework

In this section, we give a high-level overview of our moDNN framework. We first define the problem we aim to solve, and then present an overview of the proposed moDNN solution. We will also analyze the challenges of the problem and compare moDNN with vDNN. The detailed algorithms will be described in the next section.

4.1 Problem Definition

The problem we seek to solve is defined as follows.

Problem Definition. *Given a DNN with training parameters (e.g., batch size, number of iterations, learning rate, etc.) and one or more GPUs, assume that the GPU memory size or a user-specified memory budget (if given) is* ***insufficient*** *to hold all data associated with the training using one batch of samples. moDNN must make the DNN trainable on the given GPU platform, keeping any given training parameter unchanged, such that the performance (i.e., the total training time) is optimized without affecting accuracy.*

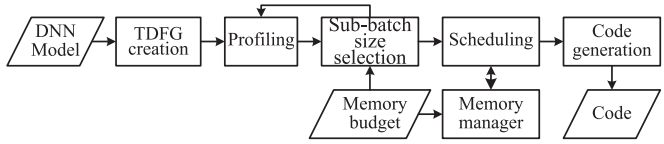


Fig. 2. The framework of moDNN.

At the highest level, moDNN adopts the idea of out-of-core algorithms [39]. Leveraging the fact that modern GPUs support overlapping computations and data transfers, we can offload data which are not being used to the host memory, and when they will be used, they are prefetched to the GPU memory in advance. A relevant theoretical study is the red-blue pebble game [40], which determines the lower bound on data transfers for out-of-core algorithms. The red-blue pebble game, however, was studied only for some special problems. To our best knowledge, finding an optimal solution for a general DAG has not been solved, and some variants are NP-complete [41]. Since finding a globally optimal solution for our problem seems computationally prohibitive, we aim to develop effective heuristics.

4.2 Framework Overview

Our moDNN framework is shown in Fig. 2. It is built on the following three key techniques.

- *Data offloading and prefetching*. Modern GPUs support overlapping computations and data transfers, which enables offloading unused data to the host memory with negligible cost. moDNN judiciously selects data to offload or prefetch.
- *Sub-batch size selection*. Reducing the batch size is a natural approach for reducing memory usage. However, different batch sizes often result in different accuracies [42]. In moDNN, we partition a batch into multiple sub-batches and accumulate the gradients from all the sub-batches at the completion of each batch, resulting in unchanged accuracy. This is a general approach and is not restricted to gradient descent. The sub-batch size is automatically selected by moDNN.
- *Convolution algorithm selection*. Convolution can be implemented by different methods on GPUs, such as general matrix multiplication (GEMM), implicit GEMM [17], the Winograd algorithm [36], fast Fourier transform [43], etc. Fast convolution methods (e.g., the Winograd algorithm [36]) typically need some workspace so the convolution algorithm must be carefully selected when optimizing memory usage.

Among these techniques, the first is architecture related, and the other two are application related. So moDNN explores both architecture- and application-level features to optimize memory usage for DNN training.

moDNN first builds the TDFG based on the given DNN. It then determines the sub-batch size based on the profiling results and the memory budget. Profiling aims to measure the tasks' execution times and the data blocks' transfer times. Since profiling needs the sub-batch size to obtain

accurate measurements, profiling and sub-batch size selection are performed iteratively until the sub-batch size converges. Next, moDNN computes a schedule for data offloading and prefetching together with convolution algorithm selection. The scheduling goal is to minimize the finish time of the TDFG, maintaining that the memory usage never exceeds the memory budget. Instead of dynamically scheduling the training process, moDNN produces a static schedule for the given DNN and GPU platform. Since the TDFG structure and dependency relations do not change, a static schedule is sufficient to ensure efficient memory usage. The last step generates the training code that can directly run on the given GPU platform based on the schedule.

4.3 Challenges

Finding a globally optimal solution for the three key techniques discussed above is not trivial. Actually, the complexity of the problem is exponential. This can be concluded from the following intuitive explanation. For each task in the TDFG, we have a number of different choices (e.g., which data to be offloaded, when to offload and prefetch, which algorithm is selected, etc.). Finding a globally optimal solution is to determine the choices for every task such that the finish time of the TDFG is minimized. It is easy to understand that finding the shortest execution time for each task individually does not necessarily lead to the shortest finish time of the entire TDFG, because the choice of a task can impact the choices of all the future tasks. Hence, finding a globally optimal solution needs to traverse all possible choices. Suppose task t has N_t choices, the last task will 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 of the problem, we resort to developing heuristic algorithms to find a good solution. However, the interaction among the three techniques, which reflects the conflicts between performance and memory usage, is still rather challenging. 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 fast convolution algorithms. Using fast convolution algorithms need some temporary workspace, which may lead to more offloading operations and also reduces the opportunity of prefetching data for future tasks. Considering these conflicting directions, a good starting point of our heuristic algorithms is to comprehensively consider both the benefit and penalty of possible choices such that desirable tradeoffs are achieved.

4.4 Qualitative Comparison with vDNN

Although our idea of offloading and prefetching seems similar to vDNN [29], moDNN has three distinct advantages over vDNN. (1) We introduce an automatic sub-batch size selection method, which cooperates with data transfer scheduling and convolution algorithm selection to optimize memory usage. vDNN does not have this feature. (2) We judiciously select data to offload by comprehensively considering both the benefit and penalty, while vDNN simply offloads the outputs of *all* layers or *all* convolutional layers. (3) Convolution algorithms are also carefully 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, which will be seen from the experimental data.

5 moDNN Algorithms

This section describes the moDNN algorithms in detail. It focuses on presenting the fundamental methodology of moDNN without considering the implementation on multiple GPUs. In other words, the methodology described in this section is based on a single GPU. In the next section, we will describe the special optimizations for multiple GPUs.

5.1 Sub-Batch Size Selection

A task in the TDFG can be executed on a GPU only if the GPU memory is sufficient to hold its input data, output data and temporary workspace during execution (for convolution related tasks). In the TDFG, the weights consume a fixed amount of memory, but the memory usages of the activations, input, label, and $\boldsymbol{\delta}$'s are proportional to the batch size. Hence, decreasing the batch size is a natural idea to reduce memory usage. However, changing the batch size can impact the accuracy [42], which violates our requirements stated in Section 4.1. 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 must be accumulated from all the sub-batches at the completion of each batch. Accumulating gradients from sub-batches on one GPU is natively supported by cuBLAS and cuDNN functions, so no extra overhead is introduced. The idea of partitioning a batch has been used in some DNN frameworks, such as Caffe [10] and CNTK [28]. However, these software packages just leave the setting of the sub-batch size to the users. Instead, in moDNN, this parameter is automatically selected such that the tradeoff between performance and memory usage is well balanced.

5.1.1 Theoretical Lower Bound of Memory Requirement

We first determine the theoretical lower bound on memory requirement 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)

$$M_{\min}(t, a(t), b) = \sum_{d(b)\in\mathbb{I}(t,b)} \text{Size}(d(b)) + \text{Size}(O(t,b)) + WS(a(t), b), \tag{6}$$

where $\mathbb{I}(t,b)$, $O(t,b)$ and $WS(a(t),b)$ are the input data set, output data and workspace of task t, respectively. $a(t)$ represents the algorithm candidate adopted by task t. The memory usages of $O(t,b)$ and $\mathbb{I}(t,b)$ are proportional to b but $WS(a(t),b)$ may not. Implicit GEMM [17] requires zero workspace but other convolution algorithms need some temporary workspace. For a single task, the theoretical lower bound of the memory requirement corresponds to

when the sub-batch size is 1 and the implicit GEMM convolution algorithm is used.

For the entire TDFG, the theoretical lower bound of the memory requirement is the maximum task-wise memory requirement instead of the sum of all the tasks' requirements, because all the tasks are executed sequentially in a topological-sort order and the memory can be reused for different tasks. According to this observation, the theoretical lower bound of the memory requirement is

$$M_{\min} = S_W + \max_{t \in \mathbb{T}} \left\{ \sum_{d(1) \in \mathbb{I}(t,1)} \text{Size}(d(1)) + \text{Size}(O(t,1)) \right\}, \quad (7)$$

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

Actually, the minimum memory requirement can be further reduced, noting that weights always reside in the GPU memory in our current implementation. If we also treat weights as regular data blocks which can be offloaded, the minimum memory requirement can be even lower. For large-scale DNNs, when the sub-batch size is 1, the memory usage is mainly consumed by the weights. In this case, if we keep only the required weights in the GPU memory, then the weight memory usage can be significantly reduced, as well as the total memory usage.

5.1.2 Automatic Sub-Batch Size Selection

We now discuss how to select the sub-batch size based on the user-specified memory budget. Of course, we can set the sub-batch size to 1, so that the memory requirement is minimized. However, using a small sub-batch size cannot fully utilize the massive parallelism of GPUs, leading to performance degradation. On the other hand, if the sub-batch size is too big and the memory budget can only hold few tasks, then offloading operations will happen frequently, which may incur severe performance degradation. Therefore, parallelism and memory margin should be carefully balanced when selecting the sub-batch size.

We use the following criterion to select the sub-batch size. The sub-batch size is selected such that any $\alpha|\mathbb{T}|$ consecutive tasks in the topological order can be executed on GPUs without any out-of-memory problems. $|\mathbb{T}|$ is the number of tasks in the set $\mathbb{T}$. α is an empirical parameter, and we use 15% in this work based on our extensive experimental study. The workspace size of the fastest possible algorithm is considered when selecting the sub-batch size. To put it formally, we select the sub-batch size b such that the following value is not larger than the user-specified memory budget

$$\max_{1 \le t \le |\mathbb{T}| - \alpha|\mathbb{T}| + 1} \left\{ \begin{array}{l} \sum_{k=t}^{t+\alpha|\mathbb{T}|-1} \left[\begin{array}{l} \sum_{d(b) \in \mathbb{I}(k,b)} \text{Size}(d(b)) \\ + \text{Size}(O(k,b)) \end{array} \right] \\ - \text{Size}(\text{reused data}, b) \\ + \max_{t \le k \le t+\alpha|\mathbb{T}|-1} \{WS(a_{\text{fastest}}(k), b)\} \end{array} \right\} + S_W. \quad (8)$$

The first term in the outer max operator is the sum of all the input data size and output data size of $\alpha|\mathbb{T}|$ consecutive tasks. The second term is the total size of the reused data of these tasks. For example, the output of a task is usually an input of the next task. In this case, the output is reused so we only need to consider its size once. The third term is the workspace size. Since the workspace of different tasks can be reused, we only need the maximum workspace size.

In order to collect the workspace size and performance of all valid convolution algorithms for each task, we conduct a profiling step on the given GPU platform. Profiling in turn requires the sub-batch size in order to get accurate measurements. To deal with this dependency, we iteratively do profiling and sub-batch size selection (see Fig. 2). During the iterations, the sub-batch size is selected in the range from 1 to the user-specified batch size by a binary search. This process continues until the sub-batch size converges.

The reason why our sub-batch size selection method works well can be explained intuitively as follows. On one hand, the sub-batch size is selected such that the memory budget can only hold 15 percent of the tasks, so the sub-batch size tends to be large and the performance may be guaranteed. On the other hand, keeping the memory of 15 percent of the tasks on GPUs has sufficient memory margin, even if the memory budget is used up by some tasks. We have observed that the offloading and prefetching latencies can be almost hidden by computations in this situation.

In practice, we find that adding a regularization step which tunes b to be a power of 2 can usually improve the GPU performance. For example, b is tuned to a multiple of 64 if b is larger than 64, to a multiple of 32 if b is between 64 and 32, and so on. The purpose of this operation is to fully utilize the performance of cuDNN, since the algorithm implementation of cuDNN is typically sub-matrix based and the sub-matrix size is naturally a power of 2.

If the sub-batch size thus determined is equal to the user-specified batch size, then it means that we need not partition batches. Otherwise, each batch is partitioned into sub-batches of size b and the training of a batch is done by multiple rounds. The gradients are accumulated from all the sub-batches at the completion of each batch.

5.2 Scheduling and Algorithm Selection

Scheduling determines the optimal data transfers and which convolution algorithms to use for all tasks, 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 and computations, (2) minimizing offloading operations, (3) judiciously prefetching future data, and (4) selecting the optimal convolution algorithms. Algorithm 1 summarizes the scheduling flow. 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 convolution algorithm, if the task is convolution related, and (3) determining the data to be prefetched for future tasks. Algorithm 1 does not actually perform training but simulates the training process to generate the schedule. The notion of "current time" used in the algorithm flow 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 which 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 memory space for the output at the same time (line 3). If the allocation fails, then 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, then it must be caused by fragmentations in the memory space, because the sub-batch size selection method guarantees that the memory requirement of any single task does not exceed the memory budget. Defragmentation (offloading all data and then reloading the 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).

Algorithm 1. Data Transfer Scheduling and Convolution Algorithm Selection

1: $T = 0$; // T is the current time
2: **for** task $t = 1, 2, \ldots, |\mathbb{T}|$ 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 convolution algorithm for task t
9: **if** task t is convolution related **then**
10: **for** all possible algorithms for task t **do**
11: Select the algorithm with the maximum gain (defined in Eq. (10));
12: Allocate workspace (do offloading when necessary) 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** defragmentation will be conducted for 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.
// Free data that will not be used
29: **for** $d \in \mathbb{I}(t)$ **do**
30: **if** d will no longer be used **then**
31: Free d;

If task t is a convolution related task, the best convolution algorithm is selected by considering both the benefit and penalty (lines 9-14). The benefit is the time saved by a faster algorithm compared with implicit GEMM which is treated as the baseline. 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. The incurred delays 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).

Next, prefetching data is considered for future tasks (lines 15-27). For each future task s ($s > t$), we first predict whether a defragmentation will be conducted when the time is just before executing task s. If the prediction result is yes, then prefetching is stopped (lines 16-17) for the following reason. When executing task s, if a defragmentation is conducted, all the data in the GPU memory will be offloaded, leading to useless prefetching operations. If prefetching is predicted to be useful, then 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).

Prefetching for future tasks has no impact on the start time of task t since it is overlapped and has no dependency. After prefetching for future tasks is scheduled, task t is executed by updating its finish time (line 28). Finally, we 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 convolution algorithms. We elaborate these aspects below. Our discussion of data offloading and prefetching is based 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.

5.2.1 Offloading Scheme

Offloading is invoked when a memory allocation fails. In this situation, we try to offload some data to vacate their spaces to make available a contiguous space that is not smaller than the requested size. Since we assume that 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 after the first offloading operation. For the input data and ground truth label, no offloading is required since they are originally copied from the host.

Although offloading operations are overlapped with computations, offloading may still cause delay to the next task to be executed (and to all future tasks as well). Fig. 3 illustrates two situations. Suppose the memory allocation for task s fails and we have to offload task t's output. The situation of Fig. 3a has delay overhead and the situation of Fig. 3b does not. During the scheduling process, we record the estimated start and

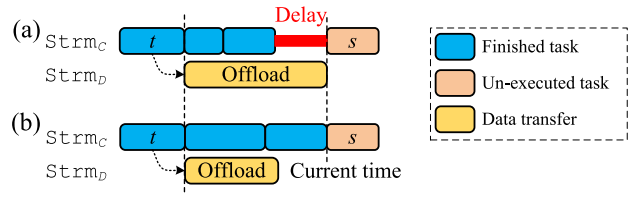


Fig. 3. Offloading example (the offloaded data block is generated by task t). (a) Delay is caused. (b) No delay is caused.

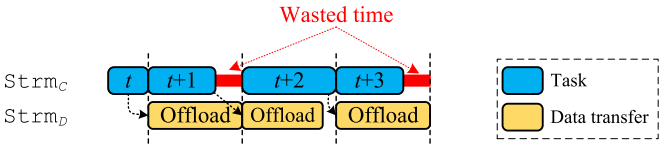


Fig. 4. Illustration of vDNN's data transfers.

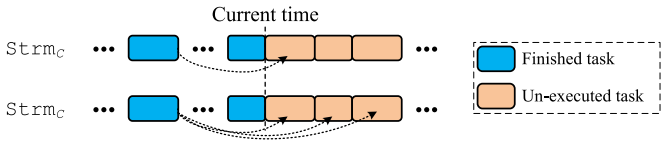


Fig. 5. If a data block will be used by the next task, or by a set of consecutive tasks immediately from the next task, then it should not be offloaded or freed. The data block can be generated by a task or from prefetching (only the former case is illustrated).

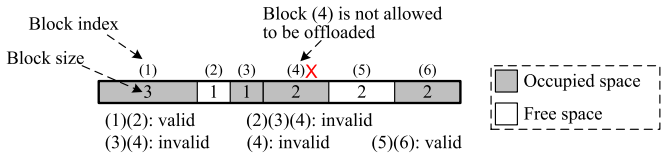


Fig. 6. An example of offloading scheme selection (for an allocation of size 4).

finish times for all the tasks in $Strm_C$ and for all the data transfers $Strm_D$. So the delay to the next task caused by an offloading operation can be easily estimated.

In vDNN [29], one data transfer can be overlapped with only one computation task, as illustrated in Fig. 4. Actually, this is an unnecessary requirement. It is easy to see that only if a task has a dependency on a data transfer (e.g., a task needs a prefetching operation for its input, or a task is waiting for an offloading operation to re-allocate some memory spaces), the task needs to wait for the data transfer; otherwise, such an imposed requirement is unnecessary, and on the contrary, causes wasted time and increases the synchronization cost.

To address this shortcoming of vDNN, in moDNN, we only handle necessary synchronizations. A necessary synchronization means that a task and a data transfer really have a dependency. As shown in Fig. 3 (also see Fig. 7), we explore more flexible data transfers such that one data transfer can be overlapped with multiple computation tasks, and vice versa. In Fig. 3, there are only two necessary synchronizations here. First, task t's output is offloaded immediately after task t is finished. Second, task s needs to wait for the offloading operation to finish and then task s can re-allocate memory. It is easy to see that our approach can reduce the delay overhead and synchronization cost compared with the vDNN approach.

Till now, we have introduced the basic concept of offloading and how it works. There are still some practical issues which are explained below.

Finding an Offloading Scheme. When performing offloading, the memory spatial distribution must be taken into account because we need a big enough contiguous space instead of multiple discontiguous segments. As described in Section 3.3, we design a tool to manage the GPU memory using a linked list data structure. The linked list allows us to search easily for an available offloading scheme by traversing the linked list. Searching for an offloading scheme is to find a set of contiguous segments (corresponding to a set of contiguous nodes in the linked list) whose total size is not smaller than the allocation size.

Before describing how to find an offloading scheme by traversing the linked list, we first describe two criteria for determining whether a data block can be offloaded or freed. (1) If a data block has not been used since its generation (by a task) or its latest prefetching, then it should not be offloaded or freed. This is easy to understand since we want to minimize unnecessary data transfers. (2) If a data block will be used immediately, then it should not be offloaded or freed either. A data block to be used immediately means that it will be used either by the next task that is to be executed, or by a set of consecutive tasks immediately from the next task, as depicted in Fig. 5. A data block that does not meet these two criteria can be offloaded or freed when searching for an offloading scheme.

To find an offloading scheme (lines 5, 12, and 22 in Algorithm 1), we traverse the linked list. For each node in the linked list, we take its corresponding data block as the first block that is to be offloaded. Contiguous data blocks at higher addresses are considered, to see if they can form a contiguous space that is not smaller than the requested size. If a data block that cannot be offloaded or freed is reached but the total size is still insufficient, then this scheme is invalid. We traverse all valid offloading schemes and select the scheme with the lowest delay overhead. It is possible that no valid offloading scheme can be 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. 6 shows an example for offloading scheme selection, in which data block (4) is not allowed to be offloaded. For an allocation of size 4, we find two valid offloading schemes (1)(2) and (5)(6). The final scheme is determined based on the delay overhead.

5.2.2 Prefetching Scheme

Once a data block has been offloaded, it needs to be prefetched when it is required. The prefetching operation should finish before the task (say, task s) that needs the data starts (otherwise, task s would be delayed). However, prefetching should not start too early since the prefetched data consume memory and are not needed by tasks executed before s. Therefore, the start time of a prefetching operation should be carefully determined. Another issue is the usefulness of the prefetching operation. When task s is to be executed, all the input data should be in the GPU memory and the output memory should be allocated. If the GPU memory cannot hold all the inputs and the output, we need to do a defragmentation. In this case, the prefetched data are freed without being used, leading to a useless prefetching

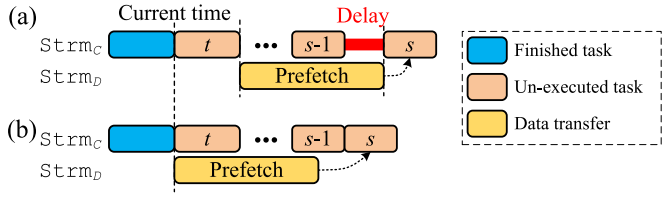


Fig. 7. Determining the start time of prefetching. (a) Delay is caused by late start. (b) No delay is caused if prefetching starts at a proper time.

operation. Considering these factors, we should carefully decide whether a prefetching operation is really useful and when it should start.

Determining the Start Time of Prefetching (Line 19 of Algorithm 1). Prefetching is scheduled only after a task $(t-1)$ is finished and before the next task (t) starts. This can be easily synchronized by CUDA events in the implementation. There is no easy way to start a prefetching operation at an arbitrary time. Fig. 7 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 check whether it will cause delay to task s if the prefetching operation is scheduled after task t is finished, i.e., if the following inequality holds,

$$T + T_{\text{alg}}(t) + \sum_{\substack{d \in \mathbb{I}(s) \text{ and} \\ d \text{ is not in GPU memory}}} T_{\text{trans}}(d) > EST(s), \quad (9)$$

where $T_{\text{alg}}(t)$ is the execution time of algorithm alg for task t, $T_{\text{trans}}(d)$ is the transfer time (the same for offloading and prefetching) of data d, and $EST(s)$ is the expected start time of task s. If (9) holds, then prefetching for task s should start now, i.e., before task t (line 19 in Algorithm 1). Otherwise, the prefetching for task s will be examined again after task t finishes and before task $t+1$ starts.

In (9), the expected start time EST is estimated by the following method. Before scheduling, the ESTs of all the tasks are estimated by assuming that all tasks use the fastest algorithm. During scheduling, some delay can be incurred by offloading, prefetching, defragmentation, or slower algorithms. Once a delay is introduced to a task, the same amount of delay is added to the ESTs of all the future tasks.

Determining Whether Prefetching is Useful (Line 16 of Algorithm 1). The necessity of this operation is explained as follows. Suppose we are considering the prefetching for task s and the current time is before task t starts. When executing task s, we need to allocate memory spaces for its output and any input data which are not in the GPU memory. If the allocation fails, then we need to do offloading or defragmentation. If the prefetched data are freed by defragmentation, then it is a useless prefetching operation. To avoid such waste, we need to predict if a defragmentation will be conducted when task s is to be executed. This can be achieved by predicting the memory distribution at a future time. However, it is impossible to predict the exact future memory distribution because the choices made for the future tasks before task s (i.e., tasks $t, t+1, \ldots, s-1$) impact the scheduling of task s, and these choices, in turn, depend on the current decision (i.e., whether the prefetching operation for task s is useful). Hence, we resort to conducting an approximate prediction by guessing the scheduling for tasks $t, t+1, \ldots, s-1$. The prediction method is described below.

Since we do not know which data will be offloaded and which data will be prefetched during the scheduling for tasks $t, t+1, \ldots, s-1$, we assume two aggressive strategies to predict the future memory distribution. First, we assume that any data block in the GPU memory can be offloaded except for the case if it meets one of the two criteria which prevent offloading unused data (described in Section 5.2.1). This assumption leads to an upper bound on the free memory space at a future time. Second, we assume that any possible prefetching operations which may be scheduled after task t and before task s (for any required data that are not in the GPU memory during this time interval) are scheduled, regardless of whether these prefetching operations are useful or not in practice. This assumption gives the maximum opportunity of prefetching data to future tasks. Furthermore, we can predict exactly which data will be freed according to lines 29-31 of Algorithm 1. With these strategies, we get an estimation of the total free memory space at a future time. Based on this, we can predict whether a defragmentation is required when preparing for task s's input and output data, by simply judging if the predicted free space is sufficient to hold task s's input and output.

5.2.3 Convolution Algorithm Selection

Different convolution algorithms (e.g., GEMM, implicit GEMM [17], the Winograd algorithm [36], fast Fourier transform [43], etc) have different performance and memory requirements, which are collected in the profiling step. Implicit GEMM requires zero workspace so it is treated as 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 allocation of the workspace may increase data offloading operations for the current task, and also reduce the opportunity of prefetching data for future tasks. In theory, it can impact all the future tasks, yielding an exponential search space. For heuristics, we look ahead only one future task in moDNN.

For task t, we check all possible algorithms one by one by considering both the benefit and penalty. There are only less than 10 convolution algorithms implemented by cuDNN, so checking all possible algorithms for each task is an inexpensive operation. We consider the following "gain" for algorithm alg

$$\Delta T_{\text{alg}}(t) = T_{\text{base}}(t) - T_{\text{alg}}(t) - T_{\text{off,alg}}(t) - T_{\text{pre,alg}}(u), \quad (10)$$

where u is the nearest future task that 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 $T_{\text{pre,alg}}(u)$ is the delayed prefetching time for task s. $T_{\text{off,alg}}(t)$ is estimated using the method shown in Fig. 3. $T_{\text{pre,alg}}(u)$ is estimated by considering the memory requirement of the prefetching for task u and the workspace size of algorithm alg for task t. In other words, we can easily find out how many prefetching operations for task u have to be delayed due to the workspace allocation for algorithm alg of task t. Note that this operation is different from predicting whether a prefetching operation for a future task is useful, because here we just need to

check the current memory distribution instead of predicting a future memory distribution. To select the optimal algorithm for task t, we just select the algorithm with the maximum $\Delta T_{\text{alg}}(t)$ defined in (10).

6 Extension to Multiple GPUs

In this section, we discuss how to extend the proposed moDNN framework to multiple GPUs. We only consider multiple GPUs on a single computer in this paper. Distributed training is out of the scope of this paper. We first analyze the bottleneck of a naive extension of moDNN, and then propose our optimization strategy.

6.1 Bottlenecks on Multiple GPUs

Generally, there are two approaches to parallelize DNN training on multiple GPUs: data parallelism and model parallelism [44]. This paper adopts a direct extension of moDNN to multiple GPUs via data parallelism, which is much simpler to implement than model parallelism. Each GPU has two streams, $\texttt{Strm}_C$ and $\texttt{Strm}_D$, for computations and data transfers, respectively. One point to mention is that since all the $\texttt{Strm}_D$ streams share the same peripheral component interconnect express (PCI-e) bus, the transfer speed for each GPU is lowered compared to the single-GPU case. A direct extension of moDNN is described as follows. First, a batch is partitioned into multiple equal-sized portions. Then, each GPU processes one portion using the schedule produced by moDNN. Finally, the gradients are accumulated from all the GPUs (each GPU may accumulate the gradients from all sub-batches first).

The above straightforward extension, however, is quite inefficient on multiple GPUs. The major bottleneck comes from the gradient accumulation step [45]. This is mainly due to that the PCI-e bus is not fast enough and the GPU-to-GPU communications must be through the PCI-e bus. Regardless whether we use GPUs or central processing units (CPUs) to accumulate the gradients, we always need to first transfer the gradients to the host memory and then transfer them to the GPUs' memories, either explicitly (by two `cudaMemcpy` calls) or implicitly (by one `cudaMemcpyPeer` call). Such gradient transfers consume a large portion of the total time. Although NVIDIA has developed NVLink recently to provide higher data transfer speed and support direct GPU-to-GPU communications, the PCI-e bus is still very widely used now. Thus it is important to develop optimization approaches for PCI-e bus based GPU platforms, which is the focus of this work.

6.2 Proposed Approach

Instead of putting gradient transfers and accumulations in the last step of each training iteration, we propose to transfer and accumulate gradients during each iteration. To see why this idea works, note that it is unnecessary to do gradient accumulation in the last step of each iteration. Take $\mathbf{W}^4$ in Fig. 1 as an example. $\mathbf{W}^4$ can be updated by $\mathbf{W}^4 += \Delta\mathbf{W}^4$ after $\Delta\mathbf{W}^4$ is produced by $\text{BP}_2(4)$ and the last use of $\mathbf{W}^4$ (i.e., task $\text{BP}_1(3)$) is finished (otherwise, a read/write contention may occur). Furthermore, we notice that during training, the CPUs are only in charge of invoking GPU tasks, and thus they are idle most of the time.

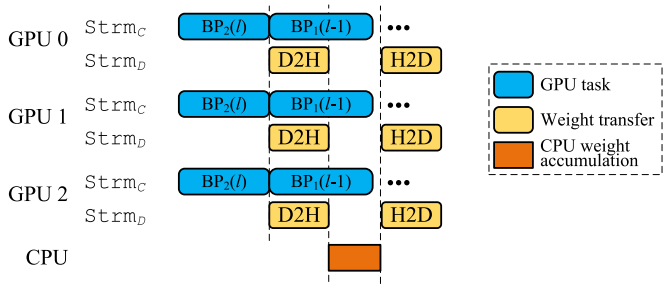


Fig. 8. The proposed gradient transfer and accumulation approach for multiple GPUs (D2H: device to host; H2D: host to device).

In order to reduce the cost of gradient accumulations, we propose to perform gradient accumulations with the CPUs. This approach also helps utilize the host resources. Based on this idea, gradient transfers can be interleaved with other data transfers (i.e., offloading and prefetching operations) and can also be overlapped with computations. Furthermore, gradient accumulations by the CPUs can be overlapped with GPU computations, so that the overhead caused by gradient transfers and accumulations can be significantly reduced. However, this new approach can still introduce some overhead to the training time. First, since gradient transfers are inserted in the offloading and prefetching lists, it may cause delay to future data transfers. Second, gradient accumulations are executed by CPUs which are much slower than GPUs, so they may cause delay.

Fig. 8 illustrates our proposed approach. Once any GPU produces a $\Delta\mathbf{W}^l$ by task $\text{BP}_2(l)$, we transfer it to the host memory by adding an offloading operation in $\texttt{Strm}_D$. (Fig. 8 does not show offloading or prefetching operations of other data.) Then the host CPUs perform an accumulation operation. After the gradient accumulations from all $\Delta\mathbf{W}^l$'s are finished on the host, we can transfer the resulting $\mathbf{W}^l$ to all the GPUs by inserting a prefetching operation in each GPU's $\texttt{Strm}_D$. The new weights will be used in the next training iteration. Before invoking such host-to-device transfers, we also need to wait for the last use (i.e., task $\text{BP}_1(l-1)$) of the corresponding $\mathbf{W}^l$ to finish. If the training samples assigned to each GPU are partitioned into multiple sub-batches, the gradient transfers and accumulations by the CPUs are only invoked during the training of the last sub-batch. For all the other sub-batches in each iteration, the gradients are accumulated on each GPU. It is likely that the gradient transfer operations of the same layer invoked by all the GPUs are executed simultaneously, since all the GPUs tend to run synchronously, as shown in Fig. 8. But, note that except for the ending of each iteration, we do not perform any other synchronization for all the GPUs to make them run synchronously. The synchronous execution is just an expected phenomenon since all GPUs run the same tasks.

If we have M GPUs, then this gradient accumulation approach produces $2M$ `cudaMemcpy` calls for one layer's weights. This means that the overhead of gradient transfers increases linearly as the number of GPUs increases. We have observed that our approach is generally good on four GPUs (see the results presented in Section 7.5). It can be expected that the proposed approach would perform poorer on systems with more GPUs, due to the following reason. With more GPUs, the workload of each GPU becomes smaller, but the overhead of gradient transfers becomes

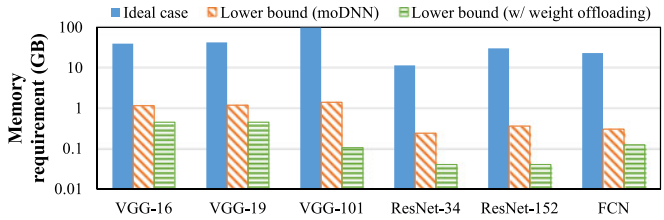


Fig. 9. Reduction in the memory requirement.

even higher. Since typically there are at most four GPUs on a single computer, our approach is still of significance in practice. For more GPUs on multiple computers, other approaches have been proposed to reduce the machine-to-machine communication overhead (e.g., [46]), which are out of the scope of this paper. However, our approach is complementary to such approaches, because moDNN can always be applied to the GPUs on each single machine, in a distributed environment.

As a final note of this section, the purpose of moDNN is to reduce the memory usage of DNN training but not to optimize the gradient accumulation step. When implementing moDNN on multiple GPUs, the overhead incurred by gradient transfers and accumulations can also be partially eliminated by leveraging the fundamental idea of moDNN (i.e., overlapping data transfers and GPU computations). The resulting approach, in turn, benefits moDNN on multiple GPUs.

7 Experimental Evaluation

moDNN is implemented using C++ and CUDA [18]. Experiments are conducted on a Linux server equipped with four NVIDIA K40 GPUs. Each K40 GPU has 12 GB memory. The host memory size is 256 GB which is sufficient to handle our test cases. We use cuDNN [17] to compute convolution and pooling operations, and use cuBLAS [16] to compute matrix-matrix multiplications for fully-connected layers.

We use the following DNNs to evaluate our moDNN framework: VGG-16 [5], VGG-19 [5], VGG-101 (created by increasing the convolutional layers of VGG-19), ResNet-34 [8], ResNet-152 [8], and a fully convolutional network [9]. These very recent DNN models are of different scales and applicabilities. The batch sizes of the six DNNs are 256, 256, 128, 256, 256, and 128, respectively, which are widely used in practice.

In this section, the "ideal case" refers to the assumption that the GPU memory is sufficient to hold all data and workspaces needed during training. The fastest convolution algorithm is always assumed in the ideal case. The ideal case together with vDNN [29] is treated as the baseline for memory usage and performance comparisons. If the memory requirement of the ideal case exceeds 12 GB, then we run each task individually and accumulate the computation time to estimate the overall training time. This method was also used in [29]. For the training time, we refer to the training time of one batch (i.e., one iteration).

7.1 Memory Requirement Reduction

Fig. 9 shows the memory requirements by the ideal case and the lower bound of moDNN. The lower bound which is calculated by (7) is the minimum memory requirement achievable by moDNN and is independent of how many GPUs we are using. Except ResNet-34, the other five DNNs all need more than 12 GB memory for the ideal case, so they cannot be trained directly on one K40 GPU. Even for the latest GPUs with 24 GB memory, the three VGG networks and ResNet-152 still cannot be trained. However, as shown in Fig. 9, moDNN greatly reduces the memory requirement. The lower bound corresponds to the sub-batch size of 1. Compared with the ideal case, the memory requirement lower bound is reduced by 59× on average. Even for VGG-101 which requires nearly 100 GB memory by the ideal case, the memory requirement lower bound of moDNN is only 1.4 GB, which can easily fit into almost all low-end GPUs or even embedded GPUs.

As mentioned in Section 5.1.1, the minimum memory requirement can be further reduced if we also offload weights. Fig. 9 also shows the memory requirements of this case. If weights can also be offloaded, the memory requirement lower bound is reduced by 396× on average compared with the ideal case. The reduction rate generally increases with the increase of the DNN scale. However, since the current implementation of moDNN can already reduce the memory requirement to a very low level, we do not offload weights in the current implementation of moDNN.

If we look at the trend of the memory requirement reduction for the DNNs of the same type (e.g., the three VGG networks or the two ResNets), we can see that the memory requirement reduction rate increases with the increase of the DNN scale. This observation can be simply explained as follows. When the scale of a DNN increases, the total memory usage tends to increase proportionally, as the memory is mainly consumed by all the layers' $\mathbf{Y}$'s and $\boldsymbol{\delta}$'s. However, the memory consumption of the weights does not increase so rapidly, which also means that the theoretical lower bound of the memory requirement does not increase significantly (since most of the weight memory usage is typically consumed by fully-connected layers). This observation implies that moDNN can attain higher memory requirement reduction for larger-scale DNNs.

We do not compare the memory requirement between vDNN and moDNN here, because the original vDNN does not have the feature of sub-batch size selection. However, we can easily apply our sub-batch size selection to vDNN. From this point of view, the theoretical lower bound of the memory requirement of (modified) vDNN should be identical to that of moDNN. If sub-batch size selection is not applied to vDNN, then moDNN can save more than 10× memory usage than vDNN (see the next section).

7.2 Comparison with vDNN

Here, we compare the performance between moDNN and vDNN. vDNN is implemented based on our best understanding of its methodology [29].

We first test moDNN and vDNN on one K40 GPU. The memory size of a K40 GPU is 12 GB (the actual available memory size is a little smaller than 12 GB). Fig. 10 shows the comparison on the training time of one batch. The average performance degradation (i.e., training time increase) of moDNN is only 3 percent, while vDNN incurs 31 percent performance degradation on average, compared with the

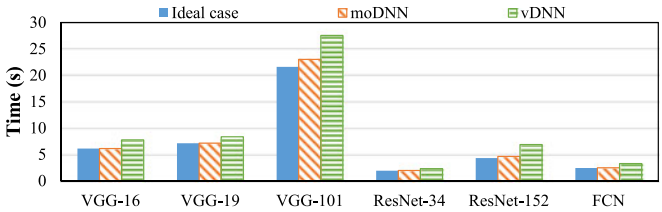


Fig. 10. Performance comparison on one K40 GPU (with a 12 GB memory budget).

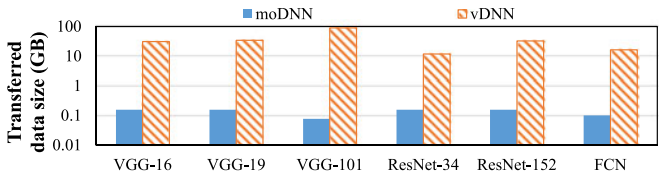


Fig. 11. Transferred data size comparison on one K40 GPU (with a 12 GB memory budget).

ideal case. The performance degradation of moDNN is mainly caused by the additional synchronizations and the lowered performance of using the sub-batch sizes that are smaller than the user-specified batch sizes. For vDNN, however, the performance degradation is mainly due to the wasted time caused by the unnecessary waiting in the inflexible data transfer scheduling (see Fig. 4).

Fig. 11 compares the transferred data sizes (including both offloading and prefetching). By judiciously selecting the sub-batch sizes and also the data to be transferred, moDNN reduces the amount of data transfer by 378× on average, compared with vDNN, under a 12 GB memory budget. The high reductions in the transferred data size mainly come from our sub-batch size selection and the judicious selection of data to offload and prefetch. Since vDNN does not support sub-batch size selection, once the memory budget cannot hold all the involved data, it offloads the outputs of all layers or all convolutional layers, leading to many offloading and prefetching operations. moDNN supports sub-batch size selection, so that using smaller sub-batch sizes gives rise to larger free memory space so as to reduce offloading and prefetching operations. In addition, moDNN judiciously selects data to offload, which further reduces the size of offloaded data.

An important feature of moDNN is its ability to fit any user-specified memory budget as long as the memory budget is not smaller than the theoretical lower bound. Fig. 12 shows the performance results under different memory budgets (on one K40 GPU). Since vDNN does not support sub-batch size selection, it fails when the memory budget is smaller than some threshold. To make a fair comparison, we also apply our sub-batch size selection method to vDNN, and the resulted approach is called vDNN+. The training times of moDNN and vDNN+ are both normalized to the corresponding DNN's ideal-case training time. The performance becomes poorer as the memory budget decreases. moDNN always attains better performance than vDNN+. This is mainly due to our new data transfer scheduling and convolution algorithm selection methods. For all the data points we have tested in Fig. 12, moDNN achieves an average speedup of 1.26× against vDNN+.

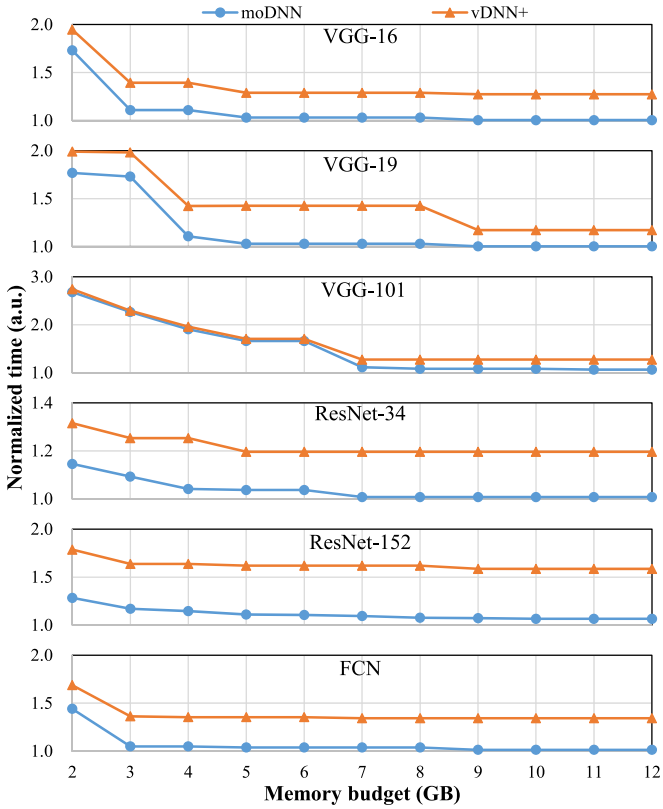


Fig. 12. Training time comparison under different memory budgets on one K40 GPU.

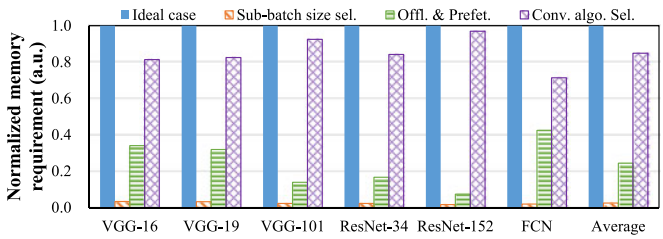


Fig. 13. Normalized memory requirement of each proposed individual technique.

7.3 Memory/Performance Breakdown Analysis

Here we analyze the impacts of the three proposed techniques individually on memory usage and performance. We first evaluate the maximum memory reduction that can be achieved by each technique. Fig. 13 shows the results. From the normalized memory requirements, sub-batch size selection is the most efficient technique on memory usage reduction, and it achieves 42× reduction on average compared with the ideal case. This is easy to understand because we can set the sub-batch size to 1 to achieve the maximum memory reduction. If we only apply data offloading and prefetching, an average a memory requirement reduction of 4.1× can be achieved. This is also approximately the maximum memory reduction that the original vDNN can achieve, since vDNN does not have the feature of sub-batch size selection. Convolution algorithm selection can only reduce the memory requirement by 15 percent on average. This also means that the workspace typically consumes ~15 percent of the total memory usage.

We then evaluate the performance impact of each individual technique on one K40 GPU with a 12 GB memory

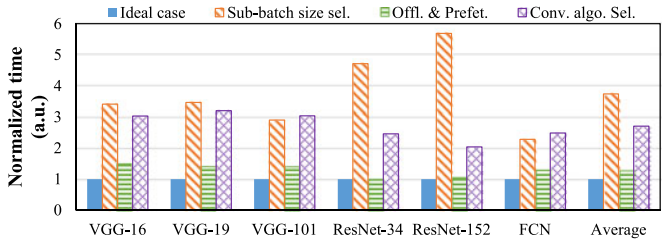


Fig. 14. Normalized training time of each proposed individual technique on one K40 GPU (with a 12 GB memory budget).

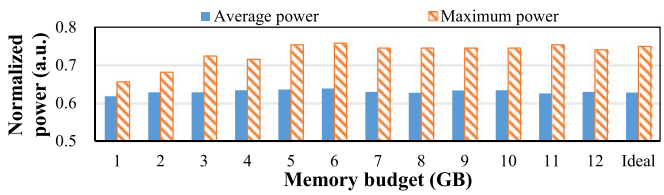


Fig. 15. Power consumption (normalized to the thermal design power of K40, 235W) of ResNet-34 under different memory budgets on one K40 GPU.

budget. The results are shown in Fig. 14. The performance is estimated when each technique achieves the maximum memory reduction. Although sub-batch size selection has the largest effect on memory reduction, it leads to the highest performance degradation, which is 3.74× on average when the memory requirement is minimized. This is due to the fact that sub-batch size of 1 significantly under-utilizes the GPU resources. If we only apply data offloading and prefetching, the average performance degradation is 28 percent. From Fig. 10, we know that the average performance degradation of moDNN is only 3 percent on one K40 GPU. This difference is mainly due to sub-batch size selection. Without sub-batch size selection, the memory budget may be used up so lots of offloading, prefetching, and defragmentation operations need be invoked, leading to serious performance degradation. If we only apply the convolution algorithm selection such that the memory usage is minimized, the performance is degraded by 2.7× on average. Combining with the breakdown analysis for memory requirement, we know that a small memory overhead (∼15 percent) used for workspace can greatly improve the performance (by 2.7×).

The above memory/performance breakdown analysis reveals that applying any single technique cannot achieve the best performance and memory usage reduction simultaneously. The observation demonstrates that the superiority of moDNN stems from the effective combination of the three techniques.

7.4 Power and Energy Analysis

Here we discuss the impact of moDNN on the GPU power and energy consumptions. We measure the average power and maximum power using the profiling tool `nvprof` provided by the CUDA toolkit. Only ResNet-34 is tested to analyze the power and energy consumptions because only ResNet-34 can be trained on one K40 GPU in the ideal case. Fig. 15 shows the measured average power and maximum power. We observe that the average power is rarely affected, compared with the ideal case. The average power is slighted increased (< 3 percent) when the memory

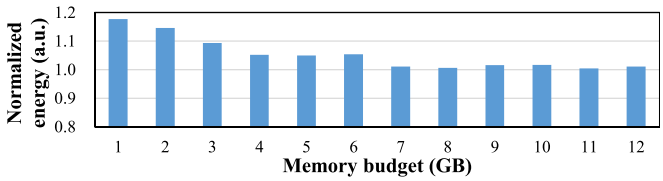


Fig. 16. Energy consumption (normalized to the ideal case) of ResNet-34 under different memory budgets on one K40 GPU.

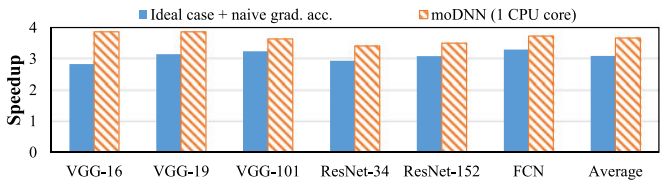


Fig. 17. Speedups on four GPUs against the 1-GPU ideal case.

budget is 5 GB or 6 GB. In this case, the sub-batch size is still 256 but there is not too much free space, so that more offloading and prefetching operations are invoked, consuming some additional power. When the memory budget is smaller than 5 GB, the sub-batch size is smaller than 256 by our sub-batch size selection approach. In this case, the hardware utilization is lowered so the average power is slightly reduced and the maximum power is also reduced with the decrease of the sub-batch size. Since the average power is rarely affected, the energy consumption of training mainly depends on the training time. Fig. 16 shows the normalized energy consumption of ResNet-34. The energy consumption increases with the decrease of the memory budget. The trend is very similar to the performance trend depicted in Fig. 12.

7.5 Results on Multiple GPUs

Here we show the results of moDNN with the proposed gradient accumulation approach on four K40 GPUs. The baseline is the ideal case implemented on four GPUs. In the ideal case, it is assumed that there is sufficient memory on each GPU. Different from moDNN, in the ideal case, the gradients are accumulated by one GPU, which is the naive gradient accumulation approach mentioned in Section 6.1. In other words, a GPU (without loss of generality, say, GPU 0) is in charge of gradient accumulation. At the completion of each training iteration, all the other GPUs send the gradients to GPU 0, and GPU 0 performs weight update and then sends the updated weights to all the other GPUs. In moDNN, gradient accumulations are executed by the CPUs. Our experimental platform is equipped with two Intel Xeon E5-2630 v4 CPUs (20 cores in total) running at 2.2 GHz.

Fig. 17 shows the speedups (against the 1-GPU ideal case) of the ideal case (with naive gradient accumulation) and moDNN on four GPUs. Remember that in moDNN, each GPU deals with a subset of the training samples. The subset on each GPU may be further partitioned into sub-batches by our sub-batch size selection method. For the reported results in Fig. 17, only one CPU core is used. This is the worst case for gradient accumulation. The ideal case achieves 3.1× average speedup on four GPUs. Since the gradient transfers and accumulations in the ideal case are executed at the completion of each training iteration, this mimics a sequential process appended to each parallel training iteration. Such overhead considerably limits the

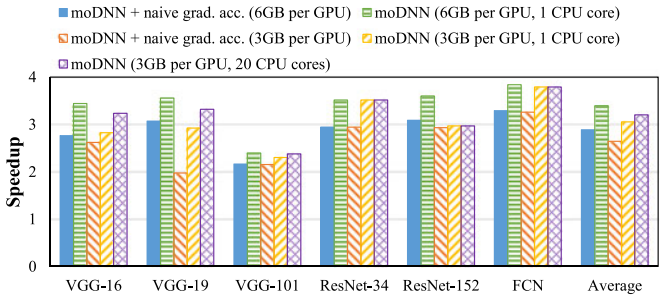


Fig. 18. Speedups under different memory budgets on four GPUs against the 1-GPU ideal case.

scalability of the ideal case on multiple GPUs. moDNN achieves 3.7× average speedup on four GPUs. As can be seen, by utilizing overlapped gradient transfers and GPU computations, the proposed approach significantly improves the scalability of moDNN on multiple GPUs.

We then evaluate moDNN on four GPUs with smaller per-GPU memory budgets, as shown in Fig. 18. Since memory budget is considered, the ideal case which assumes sufficient memory is not applicative. Thus, to show the advantages of the proposed gradient accumulation approach, we consider moDNN with the naive gradient accumulation approach as the baseline. Under the same per-GPU memory budget (6 GB and 3 GB respectively), the proposed gradient accumulation approach is always better than the naive approach. When the per-GPU memory budgets are 6 GB and 3 GB respectively, moDNN with 1-core gradient accumulation achieves 3.4× and 3.1× average speedups, respectively, against the 1-GPU ideal case, while the average speedups achieved by moDNN with the naive gradient accumulation approach are 2.9× and 2.6×, respectively.

Our experiments reveal that using more CPU cores to perform gradient accumulations can reduce the gradient accumulation time by up to 2-3×. The low improvement is mainly due to the low workload of gradient accumulation. We have evaluated moDNN with 20-core gradient accumulation under the 3 GB per-GPU memory budget, as shown in Fig. 18. The average speedup against the 1-GPU ideal case is 3.2×.

In Fig. 19, we compare moDNN on four GPUs with moDNN on one GPU. The memory budgets of the two cases are identical, so that in the 4-GPU case, each GPU has a memory budget of 3 GB. The average speedups are 3.1× and 3.3× when we use 1 core and 20 cores to perform gradient accumulations, respectively.

8 Conclusions

In this paper, we presented the moDNN framework to optimize 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 convolution algorithm selection, enabling automatic tuning of DNN training code to match any user-specified memory budget that is not smaller than the theoretical lower bound. Experimental results showed that moDNN reduces the minimum memory requirement by up to 59×. When executing moDNN on a GPU with 12 GB memory, the performance loss is only 3% on average. moDNN achieves better performance than vDNN [29] under the same memory budget. We further proposed an optimization strategy for moDNN on multiple GPUs again by exploring the basic idea of moDNN, and obtained 3.7× speedup on four GPUs. Even with smaller memory budgets per GPU, moDNN can still obtain more than 3× speedup on four GPUs.

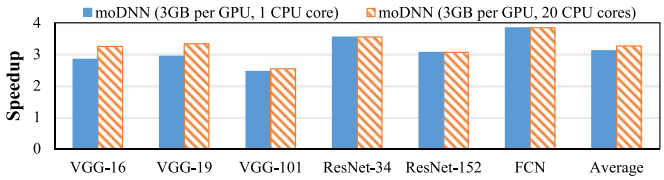


Fig. 19. Speedups of moDNN on four GPUs against moDNN on one GPU.

>The significant reduction on GPU memory usage not only opens up the opportunity for training DNNs on low-cost GPU platforms, but also benefits other memory-hungry deep learning applications/algorithms. For example, moDNN can be directly applied to three-dimensional NNs which require much more memory than conventional two-dimensional NNs. In addition, due to the reduced memory usage, moDNN can also benefit other optimization methods (e.g., conjugate gradient [34]) which require more memory than gradient descent for potential training time and accuracy improvements.

Acknowledgments

This work was supported in part 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.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[2] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2014, pp. 580–587.

[3] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

[4] Y. Goldberg, "A primer on neural network models for natural language processing," *J. Artif. Intell. Res.*, vol. 57, pp. 345–420, 2016.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, pp. 1–14, Apr. 2015. Q1

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.

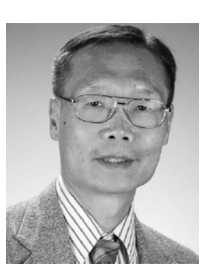
[7] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," in *Proc. Int. Conf. Med. Image Comput. Comput.-Assisted Intervention*, 2015, pp. 234–241.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[9] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 3431–3440.

[10] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.
[11] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv:1603.04467*, pp. 1–19, Mar. 2016.
[12] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A CPU and GPU math compiler in Python," in *Proc. 9th Python Sci. Conf.*, 2010, pp. 1–7.
[13] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *Proc. BigLearn NIPS Workshop*, no. EPFL-CONF-192376, 2011, pp. 1–6.
[14] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of deep learning software frameworks," *ArXiv e-prints*, pp. 1–9, Mar. 2016.
[15] J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, Q. V. Le, and A. Y. Ng, "On optimization methods for deep learning," in *Proc. 28th Int. Conf. Mach. Learn.*, 2011, pp. 265–272.
[16] "cuBLAS." (2016). [Online]. Available: http://docs.nvidia.com/cuda/cublas/ Q3
[17] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, pp. 1–9, 2014.
[18] "CUDA C Programming Guide." (2016). [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[19] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.
[20] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Penksy, "Sparse convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 806–814.
[21] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. 32nd Int. Conf. Int. Conf. Mach. Learn.*, vol. 37, pp. 1737–1746, 2015.
[22] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proc. 33rd Int. Conf. Int. Conf. Mach. Learn.*, vol. 48 , pp. 2849–2858, 2016.
[23] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Proc. 14th Eur. Conf. Comput. Vis.*, 2016, pp. 525–542.
[24] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to + 1 or -1," *arXiv:1602.02830*, pp. 1–11, Mar. 2016.
[25] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, "Memory-efficient backpropagation through time," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 4125–4133.
[26] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv:1604.06174*, pp. 1–12, Apr. 2016.
[27] "Optimizing Memory Consumption in Deep Learning." (2017). [Online]. Available: http://mxnet.io/architecture/note_memory.html
[28] "Microsoft Cognitive Toolkit (CNTK)." (2016). [Online]. Available: https://github.com/Microsoft/CNTK
[29] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2016, pp. 1–13.
[30] X. Chen, D. Z. Chen, and X. S. Hu, "moDNN: Memory optimal DNN training on GPUs," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, Mar. 2018, pp. 13–18.
[31] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv:1512.01274*, pp. 1–6, Dec. 2015.
[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd, ed. Cambridge, MA, USA: MIT Press, 2009.
[33] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org
[34] C. Charalambous, "Conjugate gradient algorithm for efficient training of artificial neural networks," *IEE Proc. G - Circuits Devices Syst.*, vol. 139, no. 3, pp. 301–310, Jun. 1992.
[35] M. Avriel, *Nonlinear Programming: Analysis and Methods*. Courier Corporation, Massachusetts, USA, 2003.
[36] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 4013–4021.
[37] A. Silberschatz, P. B. Galvin, G. Gagne, and A. Silberschatz, *Operating System Concepts*, 4th ed. Reading, MA, USA: Addison-Wesley, 1998.
[38] C. Bays, "A comparison of next-fit, first-fit, and best-fit," *Commun. ACM*, vol. 20, no. 3, pp. 191–192, Mar. 1977.
[39] 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.
[40] J.-W. Hong and H. T. Kung, "I/O Complexity: The red-blue pebble game," in *Proc. 13rd Annu. ACM Symp. Theory Comput.*, 1981, pp. 326–333.
[41] Q. Liu, "Red-blue and standard pebble games: Complexity and applications in the sequential and parallel models," Ph.D. dissertation, Dept. Electr. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2017.
[42] L. Balles, J. Romero, and P. Hennig, "Coupling adaptive batch sizes with learning rates," *arXiv:1612.05086*, pp. 1–10, Jun. 2017.
[43] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," *ArXiv:1312.5851*, pp. 1–9, Mar. 2014.
[44] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al., "Large scale distributed deep networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1223–1231.
[45] L. Wang, W. Wu, G. Bosilca, R. Vuduc, and Z. Xu, "Efficient communications in training large scale neural networks," *arXiv:1611.04255*, Apr. 2017, pp. 1–7.
[46] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "TernGrad: Ternary gradients to reduce communication in distributed deep learning," in *Proc. 31st Conf. Neural Inf. Process. Syst.*, 2017, pp. 1–11.

Xiaoming Chen (S'12-M'15) received the BS and PhD degrees in electronic engineering from Tsinghua University, Beijing, China, in 2009 and 2014, respectively. He was a visiting assistant professor with the Department of Computer Science and Engineering, University of Notre Dame from 2016 to 2017. He is now an associate professor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. His current research interests include GPU-accelerated machine learning, emerging nonvolatile devices and advanced computer architecture design. He served on the Technical Program Committees (TPCs) of Asia and South Pacific Design Automation Conference (ASP-DAC) 2019, International Conference on VLSI Design 2019, Asian Hardware Oriented Security and Trust Symposium (AsianHOST) 2018, and IEEE Computer Society Annual Symposium on VLSI (ISVLSI) 2018. He was a recipient of the 2015 European Design and Automation Association (EDAA) Outstanding Dissertation Award. He is a member of the IEEE.



Danny Ziyi Chen (F'14) is currently a professor with the Department of Computer Science and Engineering, University of Notre Dame. His main research interests include computational biomedicine, biomedical imaging, computational geometry, algorithm design, analysis, and implementation, machine learning, data mining, parallel and distributed computing, and VLSI design. He has authored more than 330 journal and conference papers and holds five U.S. patents in these areas. He is a distinguished scientist of Association for Computing Machinery (ACM). He is a fellow of the IEEE.

Yinhe Han (M'06) received the MS and PhD degrees in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2003 and 2006, respectively. He is currently a professor with the Institute of Computing Technology, Chinese Academy of Sciences. His current research interests include processing-in-memory architectures, machine learning accelerators and chip design for robots. He was a recipient of the Best Paper Award at Asian Test Symposium 2003. He was the program co-chair of Workshop of RTL and High Level Testing in 2009, and served on TPCs of several IEEE and ACM conferences, including Asian Test Symposium and Great Lakes Symposium on VLSI. He is a member of the IEEE, China Computer Federation, and ACM.

Xiaobo Sharon Hu (S'85-M'89-SM'02-F'16) received the BS degree from Tianjin University, China, the MS degree from Polytechnic Institute of New York, and the PhD degree from Purdue University, West Lafayette, Indiana. She is a professor with the Department of Computer Science and Engineering, University of Notre Dame. Her research interests include low-power system design, computing with emerging technologies and real-time embedded systems. She has published more than 300 papers in these areas. She served as an associate editor for the *IEEE Transactions on Very Large Scale Integration Systems*, the *ACM Transactions on Design Automation of Electronic Systems*, and the *ACM Transactions on Embedded Computing*. She is serving as an associate editor for the *ACM Transactions on Cyber-Physical Systems* and on the Editorial Board of IEEE Design and Test. She is the general chair of 2018 Design Automation Conference (DAC) and the TPC co-chair of 2014 and 2015 DAC. She received the NSF CAREER Award in 1997, and the Best Paper Award from DAC 2001 and IEEE Symposium on Nanoscale Architectures 2009. She is a fellow of the IEEE.