Layer-Centric Memory Reuse and Data Migration for Extreme-Scale Deep Learning on Many-Core Architectures

HAI JIN, BO LIU, WENBIN JIANG, YANG MA, and XUANHUA SHI, Huazhong University of Science and Technology
BINGSHENG HE, National University of Singapore
SHAOFENG ZHAO, Huazhong University of Science and Technology

Due to the popularity of Deep Neural Network (DNN) models, we have witnessed extreme-scale DNN models with the continued increase of the scale in terms of depth and width. However, the extremely high memory requirements for them make it difficult to run the training processes on single many-core architectures such as a Graphic Processing Unit (GPU), which compels researchers to use model parallelism over multiple GPUs to make it work. However, model parallelism always brings very heavy additional overhead. Therefore, running an extreme-scale model in a single GPU is urgently required. There still exist several challenges to reduce the memory footprint for extreme-scale deep learning. To address this tough problem, we first identify the memory usage characteristics for deep and wide convolutional networks, and demonstrate the opportunities for memory reuse at both the intra-layer and inter-layer levels. We then present Layrub, a runtime data placement strategy that orchestrates the execution of the training process. It achieves layer-centric reuse to reduce memory consumption for extreme-scale deep learning that could not previously be run on a single GPU. Experiments show that, compared to the original Caffe, Layrub can cut down the memory usage rate by an average of 58.2% and by up to 98.9%, at the moderate cost of 24.1% higher training execution time on average. Results also show that Layrub outperforms some popular deep learning systems such as GeePS, vDNN, MXNet, and Tensorflow. More importantly, Layrub can tackle extreme-scale deep learning tasks. For example, it makes an extra-deep ResNet with 1,517 layers that can be trained successfully in one GPU with 12GB memory, while other existing deep learning systems cannot.

CCS Concepts: • **Computer systems organization → Architectures**; **Neural networks**; **Heterogeneous (hybrid) systems**;

Additional Key Words and Phrases: Data placement, DNN, GPU, memory efficiency

This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1003500, and the National Natural Science Foundation of China under Grants No. 61672250 and No. 61732010. The idea was first presented as a poster at the 23rd ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'18). We sincerely express thanks to the anonymous referees for their careful corrections and suggestions. We gratefully acknowledge the support of NVIDIA Corporation with the excellent GPUs used for this research.

This is a new article, not an extension of a conference paper.

Authors' addresses: H. Jin, B. Liu, W. Jiang (corresponding author), Y. Ma, and X. Shi, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China; emails: {hjin, borenaliu, wenbinjiang, yangm, xhshi}@hust.edu.cn; B. He, Department of Computer Science, School of Computing, National University of Singapore, 119077, Singapore; email: hebs@comp.nus.edu.sg; S. Zhao, Wuhan National Laboratory for Optoelectronics, Division of Data Storage System, Huazhong University of Science and Technology, Wuhan 430074, China; email: zsf@hust.edu.cn.

1544-3566/2018/09-ART37 $15.00
https://doi.org/10.1145/3243904

ACM Reference format:
Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. 2018. Layer-Centric Memory Reuse and Data Migration for Extreme-Scale Deep Learning on Many-Core Architectures. *ACM Trans. Archit. Code Optim.* 15, 3, Article 37 (September 2018), 26 pages.
https://doi.org/10.1145/3243904

1 INTRODUCTION

The tremendous resurgence of *Deep Neural Networks* (DNNs) has made them one of the most promising machine learning techniques; they can analyze complex information and achieve high accuracy in various practical domains. Based on an analysis of the recent development of DNNs [18, 26, 36, 40], it is clear that the aforementioned breakthrough is mainly due to two reasons: large-scale training data and large-scale neural network structures.

Since DNN architecture is primarily responsible for improvements in model quality [20], the design principles of DNNs are worth considering. *Depth* and *width* are acknowledged by almost all researchers [18, 36, 40, 44, 45] as the most two important and effective factors for constructing DNNs. Proposed by VGGNet [36], and then inherited by ResNet [18, 19], more depth means stacking more functional layers; it can reduce the complexity of choosing hyper-parameters and improve the robustness of models. Compared to depth, width is another orthogonal factor for building networks, which accumulates more feature maps produced by convolutions of different sizes; this was raised by GoogLeNet [39–41], developed by WRN and ResNeXt [44, 45]. We illustrate the DNN models with these two types, deep and wide (as shown in Figure 1(a) and (b)).

There is evidence showing that the accuracy of DNN models can be significantly increased with steadily growing depth and width. Based on our experimental studies in Figure 2, for the training dataset ImageNet [10], the classification accuracy is improved by VGGNet and ResNet-family (depth varies from 50 to 1,001; width varies from 2 to 10) from 91.2% to above 95%; and the memory usage on the model-training process rises to almost 60GB. In other words, the great strength of deep learning has triggered a race to build models that go deeper or wider, leading to extreme-scale DNN models. However, the sizes of extreme-scale models determine that the training process on them requires huge computation and memory resources, which are studied in Table 1. For example, 152-layer ResNet [18] requires almost 11.61GB memory footprint and 11,300 million floating-point arithmetic operations per 32 images to accomplish image recognition tasks.

As a result, many state-of-the-art hardware accelerators like the *Graphic Processing Unit* (GPU) and *Application Specific Integrated Circuit* (ASIC) are deployed to speed up the training of extreme-scale DNNs. However, all of them have a relatively limited DRAM memory. Considering the quantities of data shown in Table 1, there is a huge gap between the current hardware and memory-intensive DNN applications. Moreover, the quantities of memory footprint and computation vary linearly along with the batch size of training data. Memory cost is a key challenge in the development of extreme-scale deep learning using accelerators. First, current systems (such as Caffe and TensorFlow) cannot handle the extreme-scale models on a single GPU, which causes the failure of training. They can only handle the models that fit into the GPU. Second, to avoid out-of-memory failure, current systems rely on the users to decide whether the model can fit into the GPU, which is a difficult task for normal users. We need an approach that can handle the extreme-scale models in a more elastic manner, even with moderate training overhead.

Some systems including DistBelief [9] and Adam [6] attempt to train models by partitioning and distributing the models over multiple GPUs. This type of parallelism is generally referred to as model parallelism, and can apparently relieve the memory pressure. However, it is not cost-effective due to its excessive communication. First, heavy data communication causes worse

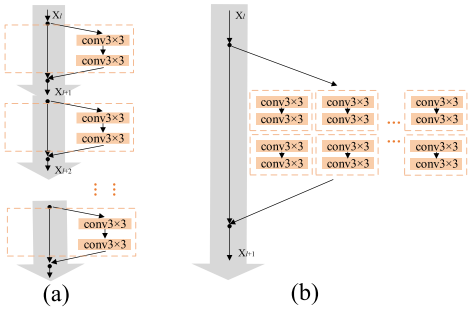


Fig. 1. DNN models (e.g., ResNet) with two types: (a) deep network and (b) wide network.

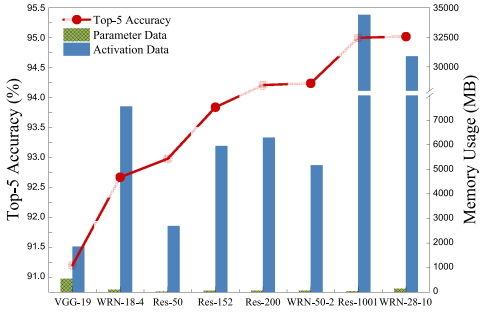


Fig. 2. Accuracy benefits and memory usage for deep and wide network models (top-5 accuracy is single-crop rate on the ImageNet validation set, batchsize = 32).

Table 1. Quantitative Analysis of DNN Model Configurations

Application area	DNN structure	Quantity of DNN		
		Training data	Model size (param/interm)	Training computation
Speech Recognition	4-layer LSTM-RNN [34] (sequence length=800)	2975-hour Google 5M LID Corpus [28])	148MB/49.2GB	0.09TFLOPS
Image Recognition	19-layer VGGNet [36] (batchsize=64)	60,000 images (Cifar-10 [25])	548MB/7.24GB	3.75TFLOPS
	152-layer ResNet [18] (batchsize=32)	1 million images (ImageNet [10])	60.2MB/11.61GB	11.3GFLOPS
	1,001-layer ResNet [18] (batchsize=32)	1 million images (ImageNet [10])	40.8MB/67.2GB	0.15TFLOPS
Activity Recognition	199-layer P3D ResNet [32] (batchsize=128)	1 million videos (Sports-1M [23])	232.5MB/23.69GB	0.07TFLOPS

We do a statistical analysis based on the model configuration (including training batch size) described in the corresponding cited papers. The model size is represented by parameter data and intermediate data (param/interm for short).

traffic. Second, complex synchronizations on model partitions make it hard to hide communication. Moreover, considerations on selecting effective partitioning strategies, balancing workloads across GPUs, and guaranteeing convergence are also open problems for model-parallelism. Consequently, as far as we know, very few training systems (e.g., TensorFlow [1]) explicitly support model-parallelism, and most other systems (e.g., Caffe [22], Caffe2 [12], MXNet [4], Theano [2], CNTK [35], DyNet [29], et al.) cannot directly implement model-parallelism.

It is therefore of great necessity to design a memory management strategy for existing deep learning systems to implement extreme-scale DNNs on a single GPU machine. In Table 2, we list the memory allocation plans of existing systems and summarize the explicit memory optimization strategies utilized by them. Except for the original version of Caffe, which allocates memory space for the whole network, almost all other systems just allocate memory space for the related layers being involved in the current layer-centric feedforward computation. Derived from Caffe, vDNN [33] and GeePS [8] both introduce the GPU memory pool to temporarily store intermediate data and migrate data between GPU and CPU memory in the training process. However, GPU memory pool causes double memory consumption on GeePS and significant performance loss on vDNN, which will be discussed in the latter sections. As another alternative, Memonger on MXNet [4] (MXNet for short in Table 2 and latter sections) is proposed to achieve memory efficiency by

Table 2. Characteristic Summary of Memory Optimization Policies Used in Existing Systems

	Memory Allocation Plan	Memory Management Policy	Performance Overhead	Data Reuse	Data Migration	Wide Networks	Extreme-Scale Training
Caffe [22]	a whole network (d+g)	in-place	–	×	×	×	×
TensorFlow [1]	related layers (d/g)	best-fit with coalescing	–	×	×	×	×
GeePS [8]	related layers (d+g)	memory pool	high	×	√	×	×
vDNN [33]	related layers (d+g)	memory pool	high	×	√	×	×
MXNet [3, 4]	related layers (d/g)	extra computation	medium	×	×	×	√
Layrub (proposed)	related layers (d)	data reuse, data migration	low	√	√	√	√

d: Activation data; g: counterpart gradient; d+g: allocate memory for both activation data and the counterpart gradient; d/g: allocate memory for either of them but not for all; – : none, as Caffe and TensorFlow have no explicit memory optimization policy, there is no comparison of performance overhead for them; √ : available, × : not available.

executing extra forward computation, which leads to a waste of computing resources. Moreover, it can be observed that for wide networks, the memory optimization policies mentioned above do not work well. This is because the current works all tend to make conservative memory allocation that accommodates the current layer, including at least two pieces of memory space for the activation and counterpart gradient, as shown in Table 2. The issue of memory relaxation still exists, since the layers of wide networks are always dense and have excessive footprints. In the existing schemes, almost 50% of the allocated memory is wasted and never reused or recycled. This will be discussed further in Section 3.

To address the memory issue derived from the extreme-scale DNNs, we design an efficient memory arrangement interface, called Layrub, for data-parallel deep learning in a GPU-based environment. We first analyze the DNN training process, and attempt to understand the memory access characteristics of DNNs. For different types of structures, including deep and wide structures, we identity two different reuse opportunities for memory space in Section 2.2, and propose three corresponding strategies to effectively reduce memory consumption. Specifically, this article makes the following contributions:

—It analyzes the sequential layer-centric training processes, exploits the independent computation and memory access operations between different layers, and reduces memory consumption by rearranging these operations.
—For wide networks, an intra-layer memory space reuse strategy is proposed for the feedforward process, using the memory space of activation data as the space for gradient data.
—For deep networks, an inter-layer space reuse strategy based on CPU-GPU data migration is proposed to maintain a constant amount of GPU memory in all iterations of training, regardless of the depth of the DNN structure.
—A hybrid reuse strategy combining both intra-layer and inter-layer strategy is proposed to work perfectly with most DNNs, and obtain a maximum memory reduction for extreme-scale training on a single GPU.

2 BACKGROUND AND MOTIVATION

To clearly illustrate our contributions, we analyze the DNN structure and give an overview of the training process. Several mathematical notations will be used to characterize Layrub.

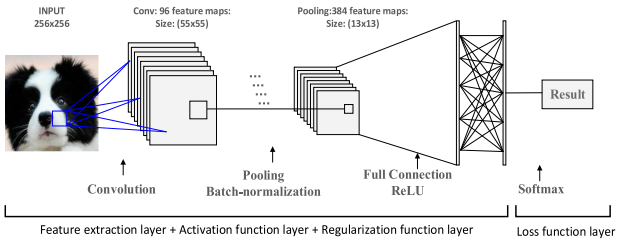


Fig. 3. An example of a CNN model with multiple types of layers for the image classification.

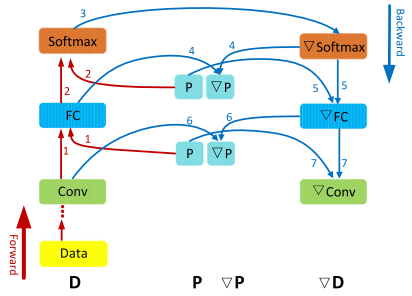


Fig. 4. Training process on a CNN (P, ∇P are the weight parameters and the counterpart gradients).

2.1 Layer-Centric Training Process

DNNs, such as *Convolutional Neural Networks* (CNNs), powerful for computer vision tasks [13], *Recurrent Neural Networks* (RNNs), for natural language processing, are all generally *layer-centric*, hierarchically organized models containing multiple types of layers. These layers can be categorized [20] as below:

(1) *Feature extraction layer.* It detects edges and any other feature representations of an image, a piece of video, or even a paragraph of text, broadly implemented as convolution layer, downsampling and pooling layer, and fully connected layer.
(2) *Activation function layer.* It is introduced to switch the linear feature representations into nonlinearity, practically implemented as tangent, sigmoid, *rectified linear unit* (ReLU) activation functions, and so on.
(3) *Loss function layer.* Sometimes called as an objective function layer, it selects one or more loss functions such as softmax and logistic regression to be applied at the last stage of DNNs.
(4) *Regularization function layer.* It is always referred to a family of techniques to resolve overfitting, specifically including dropout and batch normalization [20].

Figure 3 illustrates a CNN model that contains all four types of layers mentioned above, which is specific to the image classification task. The goal of this task is to train a CNN to make the targets in images (such as dogs and cats) be classified correctly. To train a DNN model is to adjust its weight parameters to minimize the gap between the correct classes of the targets and the classes obtained by the trained DNN model [38]. The training process can be divided into the following steps: the forward phase, which aims to obtain the predicted classes; the backward phase, which aims to minimize the aforementioned gap (generally referred to as the *loss*); and the weight parameter update phase, which aims to adjust and update the weight parameters.

Forward Phase. In this phase, CNN transforms raw input data into a series of intermediate activations in a layer-by-layer fashion, and lastly presents the final output used to compute the *loss*. As shown on the left of Figure 4 with red arrows, the first layer (*Data*) is an input layer that reads a batch of raw data chosen from the shuffled training set in domain-specific modalities, and connects to a sequence of intermediate layers (*Conv*, *FC*, and *Softmax*). The intermediate layers (denoted as l) consist of a certain number of neurons, some of which apply an activation function $f_{(l)}$ to the input item $z^{(l)}$ (i.e., neuron status), and produces output item $a^{(l)}$ (i.e., neuron activation), which is described in Equation (1). Some layers utilize the weight parameter and bias, which are denoted as $w^{(l)}$ and $b^{(l)}$, to participate in the computation for the neuron status $z^{(l+1)}$ of the next layer, which

is described in Equation (2):

$$a^{(l)} = f_{(l)}(z^{(l)}), \tag{1}$$

$$z^{(l+1)} = w^{(l)} * a^{(l)} + b^{(l)}. \tag{2}$$

Backward Phase. In this phase, a loss function layer is defined to evaluate prediction loss at the end of forward phase. The gradient of the *loss* is obtained from the output of the last layer (n), then back-propagated through the network, exactly shown as the flow of right-hand blue arrows in Figure 4. There are two main computations of partial derivatives in backward phase for layer l: (1) compute $\frac{\partial loss}{\partial w^{(l)}}$, the gradient of the *loss* relative to the weight $w^{(l)}$ by Equation (3); (2) compute $\delta^{(l)}$, the gradient of the *loss* relative to the status $z^{(l)}$ by Equation (4). These computations are generally derived from the chain rule of calculus [27]:

$$\frac{\partial loss}{\partial w^{(l)}} = \delta^{(l+1)} * a^{(l)}, \tag{3}$$

$$\delta^{(l)} = \delta^{(l+1)} * w^{(l)} * f'_{(l)}(z^{(l)}). \tag{4}$$

Based on the layer categorization [20] mentioned above, the computation of $\delta^{(l)}$ should be divided into two variants: for the feature extraction layers which are parametric, the input item $z^{(l)}$ and output item $a^{(l)}$ can be treated as equivalent, so $f'_{(l)}(z^{(l)})$ should equal to 1; for the activation function layers that are parameter-free, the weight parameters $w^{(l)}$ can be treated as an identity matrix. Consequently, the computation of $\delta^{(l)}$ in Equation (5) can be adapted as below:

$$\begin{cases} \delta^{(l)} = \delta^{(l+1)} * w^{(l)} & \text{if layer}(l) \text{ is parametric,} \\ \delta^{(l)} = \delta^{(l+1)} * f'_{(l)}(z^{(l)}) & \text{if layer}(l) \text{ is parameter-free.} \end{cases} \tag{5}$$

Update of Weight Parameters with SGD. To minimize the aforementioned *loss*, update of the weight parameters of DNNs are typically performed by the iterative *Stochastic Gradient-Descent* (SGD). Weight parameters are generally updated by the gradients of the *loss* relative to each weight $\frac{\partial loss}{\partial w^{(l)}}$, which are obtained in the backward phase. The SGD update rule can be described as Equation (6):

$$w_{t+1}{}^{(l)} = w_t{}^{(l)} - \eta \frac{\partial loss}{\partial w_t{}^{(l)}}, \tag{6}$$

where $w_t{}^{(l)}$ denotes weights of layer l at the SGD iteration step t, η is the learning rate. To achieve the minimization of *loss*, the training process is generally comprised of millions of iterations.

2.2 Motivation: Memory Efficiency on Extreme-Scale Training

We first look into the memory usage characteristics from an algorithmic perspectives (Equations (1)–(6)), and offer the following key observations.

Observation 1: The memory footprint of the activation data and its gradient is equally important. First, we note that the backward phase requires about twice as much memory space as the forward phase, since the backward phase uses not only activations and parameters of each intermediate layer (except the input data layer), but also their counterpart gradients. Second, we note that $|P| = |\nabla P|$ and $|D| = |\nabla D|$, meaning that both the parameters P and their gradients ∇P (corresponding to $w^{(l)}$ and $\frac{\partial loss}{\partial w^{(l)}}$ in Equations (3)–(6)) have the same size, so as the activation data (corresponding to $a^{(l)}$ and $\delta^{(l)}$ in Equations (1)–(5)). Our motivation for intra-layer memory reuse between activation data and their gradients emerges from here. The identical size of them, such as *Conv* and ∇*Conv*, *FC* and ∇*FC*, as demonstrated in Figure 4, offers an opportunity to reuse space between them. The coordinated execution sequence discussed in Section 3.1 is used to implement this.

Observation 2: The memory footprint between different layers is independent. Following the chain rule for layer-centric feedforward computation, we also show further viewpoints from an architectural perspective. First, from the execution flow in the feedforward phase shown in Figure 4, we learn that the training process is achieved in a layer-by-layer fashion, regardless of the depth of the network and whether the structure has a linear [36] or non-linear shape [18, 40]. Second, instead of presenting the data for all layers in GPU memory over the entire training process, only the smallest amount of memory required for layer-centric computation is enough, and this will be exploited in the next section. These properties are suitable for our inter-layer memory reuse strategy between layers with independent memory footprint, and for maintaining the lowest memory usage for layer-centric computation.

Observation 3: The activation data is the major contributor to the memory footprint. In addition, we do not optimize parameters and their gradients, even though the parameter gradients are potential choices for reusing the memory space of parameter data. For data-parallelism training, parameter data need to be kept in memory since the parameter updating process could exchange and aggregate them between different nodes. There is another reason that the sizes of the parameters are smaller than those of the activation data, which has been illustrated in Figure 2.

3 DESIGN OF LAYRUB

The main objective of Layrub is to manage data placement and reduce the memory usage of DNNs. In this section, first we exploit the coordinated execution sequence enforced to facilitate the intra-layer reuse strategy, and then introduce three strategies, intra-layer, inter-layer, and hybrid from algorithmic and architectural perspectives to reuse GPU memory space.

3.1 Coordinated Execution Sequence

As the core of deep learning, the layer-centric feedforward computation generally performs the forward and backward phases iteratively in a number of passes. It is composed of a sequence of layer-based operations. A forward operation in one layer cannot happen earlier than its previous layers, because it requires the output data of the previous layers as its input data. Similarly, a backward operation also depends on its following layers. These dependencies and relationships between operations directly determine the memory placement and memory consumption during each iteration. It is noted that the core idea of our strategies is *reuse*, which is motivated by the observation that the memory space of neuron data of the current layer can be reused by other layers. This means that the memory space allocated for some data is reused for others if these are generally independent of each other. Therefore, to reorganize the input or output data placement, we excavate the remaining independence among these sequential operations, and exploit an appropriate coordinated execution sequence.

Taking the training process of a simple CNN model in Figure 4 as an example, we observe operations 4 and 5 in the original backward phase. The former is designed to obtain the parameter gradient ∇P of the layer (FC), and the latter aims to obtain the neuron data gradient ∇FC. These types of operations are interchangeable without affecting the correctness of computation. If we therefore focus on the parameter gradient calculation in a layer-centric manner, and calculate the neuron data gradient of the corresponding layer at a later point, the memory footprints of the two operations will be independent of each other. Following our adaptation, the *coordinated execution sequence* of the backward phase would be fixed: *the parameter gradient calculation of one layer will take place ahead of neuron gradient calculation of the same layer.* As shown in Figure 4, operation 4 is arranged before operation 5, and operation 6 is arranged before operation 7. The rest of the operations could be carried out in the same manner.

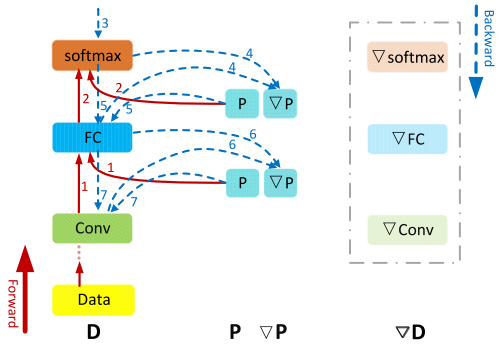


Fig. 5. Intra-layer space reuse.

3.2 Intra-Layer Memory Reuse

The intra-layer memory reuse strategy is designed to tackle the severe memory ballooning issue in DNNs, particularly for wide networks, as the memory consumption is extremely high within one layer (a large number of neuron activations concentrating at certain layers). As described in Section 3.1, we reestablish an alternative coordinated operation sequence. Based on this, an intra-layer memory reuse strategy is proposed which relies on the operation independence between the forward and backward phases of one layer, and reuses the memory space between them. Therefore, the data placement of the training process could be reorganized: the neuron gradient is able to reuse the memory space of the neuron data.

For instance, as illustrated in Figure 5, operation 1 represents the forward phase of layer (*FC*), and operations 4 and 5 both represent the backward phase of layer (*FC*). The neuron data *FC* obtained by operation 1 will be utilized by operation 4 at first, and after that the memory footprint of neuron data *FC* will be reused by operation 5 to calculate the gradient data ∇FC. The rest of the operations could be carried out in the same manner: the memory footprint of *Conv* will be reused by operation 7 to calculate the gradient data $\nabla Conv$.

The proposed intra-layer strategy reuses the memory space of the forward phase for that of the backward phase within one layer, which means that the proportion of memory savings reaches up to 50% in the corresponding layers. In particular, the intra-layer method is most beneficial for training wide network models. Compared with existing conservative methods, which only reduce memory usage by about 10% for the extra-wide models, the proposed intra-layer strategy can achieve a moderate effect on memory savings. We describe our implementation of this intra-layer strategy in Section 4, and demonstrate its effectiveness empirically in Section 5.

3.3 Inter-Layer Memory Reuse and Data Migration

While the intra-layer strategy reuses and saves memory space at the intra-layer level, it cannot reuse memory space across different layers. For the deep networks whose depth is extremely unbounded (e.g., VGGNet [36], ResNet [18]), the proportion of memory consumption is still unacceptable after utilizing the intra-layer method. To tackle this issue, we introduce an inter-layer memory reuse and data migration strategy that can reuse a constant amount of memory space for the sequential operations of different layers, and can migrate related data between CPU and GPU memory. It considers both the mathematical laws of training and the heterogeneous multi-core architectures of deep learning systems. The proposed strategy is derived from two general observations.

First, we observe from Figure 4 that if the computation of parameter gradients of one layer is far from being executed, as when operation 7 is distant from operation 1, the memory footprint of layer

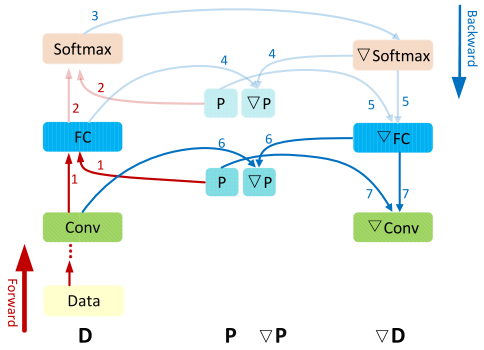


Fig. 6. Inter-layer space reuse and data migration.

(*Conv*) could be reused by another layer, after the activation data of layer (*Conv*) are temporarily migrated from GPU memory to CPU memory. Until the time comes to execute operations 6 and 7, the data for layer (*Conv*) will be migrated back to GPU memory. This is consistent with the conclusion obtained in Section 2.2: operations and memory footprint all follow the layer-by-layer fashion. Therefore, we can repeatedly reuse the same constant amount of memory space for the independent and sequential operations from different layers.

Second, considering the mathematical restrictions of feedforward computation, the computation of parameter gradients for one layer is executed as shown in Equation (3). For example, in Figure 6, when aiming to obtain the parameter gradient ∇P of layer (*Conv*), both the activation data *Conv* of the current layer and the neuron gradient data ∇FC of the previous layer must be used simultaneously. This means that we need only keep those data that will be utilized intermediately in the backward computation. Here, CPU memory is treated as an unloading zone for temporary storage of the remaining data, since this memory is much larger than that of the GPU.

The core idea behind this inter-layer strategy is a variant of the classical producer-consumer problem. Compared to the original memory allocation plan, where consumers are assigned all the resources they require at one time, it is more appropriate to distribute resources to them when they truly need them. Then, the consumer should release these resources as soon as they finish. With respect to memory consumption in the DNN training process, a finite volume of GPU memory can be interpreted as the resources, and the layer-centric operations in a DNN model can be regarded as the consumers, which need enough GPU memory to store their neuron data.

A data migration strategy is proposed to enable consumers to reuse GPU memory resources in a timely manner. However, it is a key issue that how to choose a proper time to start to migrate data between GPU and CPU. Moreover, the different shapes of networks (i.e., linear or non-linear shapes) will exacerbate the problem. Taking this into account, we record the sequence of memory access on GPU for the data of each layer and trace the number of reference count for the data that is referred and used by other layers. In this way, a *fine-grained memory access sequence* of the training process is established to determine when to migrate data. Furthermore, considering the overhead of data migration, we enable two CUDA streams: one is the default stream, which interfaces to the cuDNN library [5] and takes over all computations of forward and backward; the other is a separated stream specific for the CPU-GPU data migration via PCI-e. These streams can be executed concurrently on GPU to overlap the computation with migration and hide the latency of migration. The implementation detail will be described in Section 4.1.

Note that the lowest memory usage for every layer to carry out layer-centric feedforward computation is different in an arbitrary network. For simplicity, therefore, we select two pairs or more

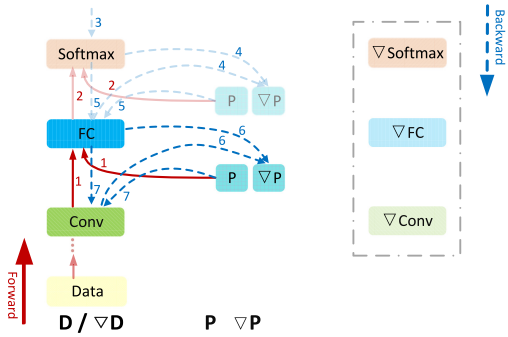


Fig. 7. Hybrid of two memory reuse strategies.

of the largest memory spaces (such as *Conv* and ∇*Conv*, *FC*, and ∇*FC*) as the constant resource region, which will be repeatedly requested and reused by continuous layers.

3.4 Hybrid Layer-Centric Memory Reuse

The intra-layer strategy reuses memory space at the intra-layer level, whereas the inter-layer strategy reuses memory space at the inter-layer level. They exert their advantages in either wide networks or deep networks, respectively. However, only one single strategy cannot address the memory issue of the networks both wide and deep. Two strategies have their own limitations on the extreme-scale DNNs, which will be illustrated in Sections 5.2.1 and 5.2.2.

As a consequence, we hybridize these two strategies, and propose a hybrid layer-centric memory reuse strategy to perfectly accommodate almost all DNNs. Moreover, these two strategies are not necessarily in conflict with each other so long as a coordinated operation sequence (as described in Section 3.1) is predefined. Since the parameter gradient calculation for each layer would take place ahead of neuron gradient calculation, the operation independence between continuous layers cannot be disturbed by the intra-layer strategy. Therefore, the memory footprint independence between layers can also be maintained, which motivates the inter-layer strategy. As described in Figure 7, the layer-centric gradient calculation for ∇*Conv* still can be executed after utilizing the intra-layer strategy, and only two pieces of spaces (one for ∇*FC*, the other for ∇*Conv*) are kept in GPU memory. In this way, the inter-layer strategy can be implemented naturally.

By building the inter-layer strategy on the basis of intra-layer strategy, the amount of GPU memory space for layer-centric computation shrinks largely. This strategy obtains a maximum memory reduction for extreme-scale training on a single GPU. The results of comparative experiments demonstrate its effectiveness in Section 5.2.

4 IMPLEMENTATION

This section describes the implementation details of Layrub. Additionally, it identifies and resolves some problems when implementing the proposed strategies in this section.

4.1 System Implementation

Layrub is built as a C++ library that manages data placement for deep learning systems running on GPUs. Layrub can be implemented for different systems including MXNet and TensorFlow. For clarity of presentation, we use Caffe as an example to illustrate the implementation details of Layrub. A DNN model is often deployed in a single CPU process that launches NVIDIA accelerated library calls or CUDA kernels to execute computations on GPUs. This process calls Layrub functions to manage data placement and take over the ordinary computation iterations. The Layrub

Table 3. Layrub Function Calls Used for Managing Data Placement

Methods	Owner	Arguments	Description
addRef()	Tracer	A layer index	Traverse the network, set up a Ref for data of each layer
subRef()	Tracer	A layer index	Trace the Refs in the formal forward phase
reuse()	IntraLayer	Layer type, pointers to the activation data and its gradient	Share the pointer of data to its gradient
Offload()	InterLayer	Layer index and Ref	Offload data from GPU memory to CPU
Loadin()	InterLayer	Layer index and Ref	Load data from CPU memory to GPU
reuse()	InterLayer	Layer type, pointers to the activation data and its gradient	Share the pointers across layers

function calls are summarized in Table 3. Algorithm 1 gives an example of a DNN training process using Layrub. Three main components and corresponding methods are illustrated as below:

— **Tracer.** Before the formal training process, a network model must be initialized by being traversed layer-by-layer. `Tracer` calls the method `addRef()` to accumulate the reference count for the activation data (`Ref` for short) of each layer. Here `Ref` is defined to record the number of times the activation data of the current layer has been referred by the subsequent layers ("referred" means that the data has participated in the computation of the subsequent layers). `addRef()` will add 1 to `Ref` as long as one of the latter layers refers to the data. After the traverse, `Tracer` can hold the `Refs` of every layer, and then a *fine-grained memory access record* of the training process can be established. In the formal training process, when the forward computation of a layer is executed, `Tracer` will call the method `subRef()` to reduce `Ref` of the data that is referred by this layer. If the `Ref` equals to 0, it means that the data will never be referred by the latter layers again, and there is no dependency between the data and the computation of other layers. Therefore, the activation data can be safely offloaded from GPU memory to CPU without disturbing the correct computation.

— **IntraLayer**. To reuse memory space at the intra-layer level, `IntraLayer` calls the method `reuse()` to share the pointers of the activation data and its gradient in a layer. Specifically, in the backward phase, `reuse()` directly writes the corresponding gradient values into the space address of the activation data in the same layer.

— **InterLayer.** To reuse memory space at the inter-layer level, as well as overlap the computation and migration, two asynchronous CUDA streams are enabled to achieve both computation and migration concurrently. The migration criteria is that: (1) In the forward phase, the time when the activation data is offloaded to CPU memory can be determined by the `Ref` of current layer: if the `Ref` equals to 0, `InterLayer` will call `Offload()` to migrate the activation data through `migrate_stream`. (2) In the backward phase, the time when the activation data will be loaded in GPU memory depends on the index of layers, meaning that when the backward computation of a layer l is executed by `default_stream`, at the same time, `InterLayer` will call the method `Loadin()` to migrate the activation data of layer (l-1) through `migrate_stream`. After finishing the task of computation or migration for one layer, the streams will be synchronized and used to execute task for the next layer.

4.2 Memory-Efficient ReLU Activation Function

In the proposed intra-layer space reuse strategy, the memory space of the neuron data is reused to obtain the gradients of the same layer. Therefore, the neuron data and corresponding gradient data should remain independent of each other. However, unlike other types of functional layers that always comply with this independence, it is difficult to apply the intra-layer strategy in the

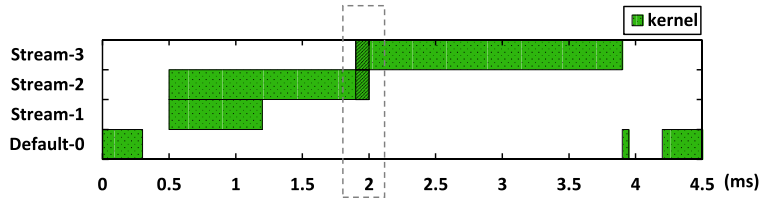
ALGORITHM 1: Iterations of training process on Layrub

```
1:  function TRAIN(net)
2:      //L: total number of layers in a network (net); T: total number of iterations, iteration is denoted
        as i
3:      for l = 1 → L do    //When initiate a net, set up reference count (Ref) for data of each layer l
4:          Tracer[l].addRef()
5:      end for
6:      for i = 1 → T do    //In the iteration i, Tracer traces the active Ref for each layer
7:          for l = 1 → L do    //The forward phase
8:              //The computation of forward and backward can be executed by the default_stream
9:              layer[l].Forward(default_stream)
10:             //When the forward is executed, Tracer subtracts 1 from the Ref
11:             Tracer[l].subRef()
12:             //When the data of l has no dependency on other layers, it can be offloaded by the
                migrate_stream
13:             if Tracer[l].Ref == 0 then
14:                 InterLayer[l].Offload(migrate_stream)
15:                 Synchronize()    //Synchronize default_stream and migrate_stream for each layer
16:                 //After the offload, the space of current layer will be reused by the next layer
17:                 InterLayer[l].Reuse()
18:             end if
19:         end for
20:         for l = L → 1 do    //The backward phase
21:             //Memory space can be reused from forward to backward at the intra-layer level
22:             layer[l].Backward(IntraLayer[l].Reuse(), default_stream)
23:             //The required data will be loaded in GPU in advance by the migrate_stream
24:             InterLayer[l-1].Loadin(migrate_stream)
25:             Synchronize()    //Synchronize default_stream and migrate_stream for each layer
26:         end for
27:     end for
28: end function
```

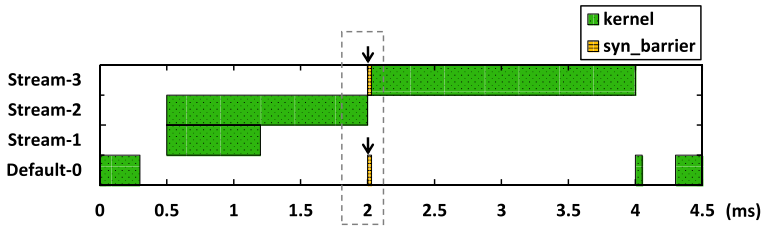
activation function layer ReLU, since the neuron status data $z^{(ReLU)}$ is needed to obtain the derivation of $\delta^{(ReLU)}$. In other words, when we implement space reuse in the ReLU layer, the mathematical rules of ReLU make it hard to reuse data between the forward and backward phases, as shown in Equation (7). Based on the parameter-free derivative computation in Equation (6), the derivative of ReLU is denoted as below:

$$\delta^{(ReLU)} = \delta^{(l+1)} * f'_{(ReLU)}(z^{(ReLU)}) = \begin{cases} 0 & z^{(ReLU)} \leq 0 \\ \delta^{(l+1)} & z^{(ReLU)} > 0 \end{cases} \tag{7}$$

To tackle the aforementioned problem, we further analyze the mathematical prerequisites. It is observed that the $z^{(ReLU)}$ is only used to show a positive or negative value in the backward phase, as demonstrated in Equation (7). Since the $z^{(ReLU)}$ is not normally involved in a 32-bit floating point backward computation, we propose a memory-efficient ReLU activation function that compresses the original neuron data of ReLU and stores the compressed data in an extra CPU memory space. After transferring this compressed data to an extra region, the original memory space of the neuron data can be reused by the corresponding gradient data, complying with the



(a) Kernel execution without synchronization barriers



(b) Kernel execution with synchronization barriers

Fig. 8. Kernel execution timelines with and without synchronization barriers.

intra-layer strategy. In our implementation, the *char* type is chosen as the dominant data type for compressing the neuron data $z^{(ReLU)}$; this has the shortest length (8-bit) supported by the CUDA function. Therefore, the extra memory consumption is reduced to 1/4 of the original data size, with almost no additional time consumption. The proportion of memory savings for DNN models utilizing the memory-efficient ReLU function is shown in the next section.

4.3 Synchronization Control on Tasks

Popular deep learning systems such as Caffe [22] and TensorFlow [1] all execute training tasks on GPU by calling accelerated CUDA-aware libraries, including general mathematical libraries such as cuBLAS [31] and domain-specific libraries accelerating deep learning primitives such as cuDNN [5]. Since cuDNN specifies that the training tasks can only be executed by applying streams technology, deep learning systems which call cuDNN will execute training task in streams.

Figure 8 shows four kernel execution timelines captured by the NVIDIA profiler [30] when conducting the backward process on a convolution layer. For the convolution layer, cuDNN enables three streams (Streams 1, 2, and 3 in Figure 8) to obtain the gradient of bias, gradient of weight, and gradient of neuron activation data, respectively. In addition, Caffe enables a default stream (Default 0 in Figure 8) to dispatch the rest of the CUDA operations, which are irrelevant to the gradient computation in the backward.

As we know, data used by concurrent kernels should be independent, since the kernels in different streams are fully asynchronous. Taking the intra-layer space reuse strategy as an example, the memory space of the activation data is reused by the gradient of the activation data. When obtaining the gradient of bias and the gradient of weight (which are referred to as Stream 1 and Stream 2), it needs to read the activation data; and when obtaining the gradient of activation data (which are referred to as Stream 3), it needs to write back to the space of the activation data. Thus, the activation data used by Streams 2 and 3 are interdependent.

If the data of concurrent kernels is interdependent, the data dependence hazard will induce errors into the feedforward computation. This is the problem faced when implementing the alternative fine-tuning computation order on frameworks such as Caffe, using cuDNN accelerated library calls. For example, the overlap of the green bars in Figure 8(a) is caused by the concurrent

Table 4. System Configuration (Only One GPU in TITAN Z Card Is Used in Our Experiment)

GPU	Machine	Environment
NVIDIA GeForce GTX TITAN Z GPU, 6082MB	Intel Core i7-920 (2.67GHz 8-core), 64GB RAM	64-bit Ubuntu 14.04, CUDA toolkit 7.5, and cuDNN v4
NVIDIA Tesla K40M GPU, 11580MB	Intel Xeon E5-2620 (2.40GHz 24-core), 256GB RAM	64-bit Ubuntu 14.04, CUDA toolkit 7.5, and cuDNN v4

kernels execution between Stream 2 and Stream 3. In the intra-layer space reuse strategy, without intervening in this overlap, the accuracy of model training for the dataset Cifar-10 severely decreases, to 56% from the normal 75%, after we finish training iterations. This degradation in model accuracy is unacceptable. Therefore, we propose a synchronization control method to completely avoid the accuracy degradation and guarantee the correctness of computation.

Synchronization Control of Tasks. To address the problem of accuracy degradation while achieving a memory-efficient training process, we isolate the interdependent tasks from all derivation tasks of the parametric layers, such as the convolution layer. The synchronization barrier operations (two tiny yellow bars in Figure 8(b)) are inserted into the corresponding streams to segregate these interdependent tasks from each other, as shown in Figure 8(b). Synchronization barriers manage the execution sequence of CUDA calls, so that a given CUDA operation in a stream will not start until all previous CUDA operations in all other streams have finished. In our case, the gradient computation of the activation data in Stream 3 will not start until the gradient computation of the bias and weight in Streams 1 and 2 are finished. Therefore, even though kernel launches are asynchronous, we do not observe the overlapping execution of two kernels that have been launched to different streams, and the data dependence hazard caused by concurrent kernels execution does never happen. As a result, the accuracy degradation is completely avoided.

It should be noted that this approach will suffer from rather slow execution in the convolutional layers, due to solving the data dependence hazard. In Figure 8, we observe that the synchronization control overhead can be approximately measured by the degree of overlap between asynchronous kernels launched to different streams. The overlap (also called concurrency) between kernels always depends on the remaining *Streaming Multiprocessor* (SM) resources available for a given kernel, after the SMs have been scheduled for the preceding kernels. Fortunately, as shown in the tests, for a NVIDIA K40 (a typical GPU that possesses 2,880 CUDA cores) the degree of overlap reaches only 3.05%. It means that the overhead for managing synchronization barriers is about 3.05%, which is acceptable compared to the prolonged training time.

5 EVALUATION

In this section, we evaluate the performance of Layrub in extreme-scale deep learning over GPU. We choose representative DNN models widely applied in tasks including image classification, language model, and video description.

5.1 Experiment Setup

Experimental Environment Setup. We conduct experiments on machines with two types of GPUs, as described in Table 4. The GPU communicates with the CPU via a PCI-e bus (gen3), which provides a maximum 16GB/sec data transfer bandwidth. Except where otherwise indicated, all experiments are performed in K40M. In the distributed experiment, the machines are inter-connected via a 1Gbps Ethernet interface, and each has one K40M. We deploy Layrub on the basis of Caffe [22], which is forked from the official BVLC version, and modify it with our three strategies.

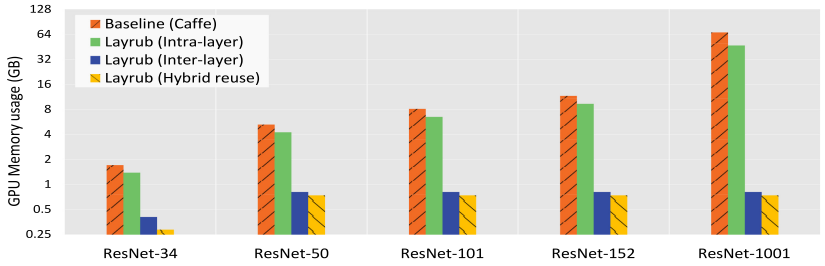


Fig. 9. GPU memory allocation for deep network models (with 32 batch size of ImageNet dataset).

Models and Datasets. Our experiments use some well-known CNN models including LeNet [27], Cifar10_quick [25], AlexNet [26] and its variant CaffeNet, VGGNet [36], GoogLeNet [39], ResNet [18], and WRN [45]; and two LSTM-based models, including LSTM-RNN [24] for language model and LSTM-CNN [11] for video description. All configurations of these models follow the default hyper-parameters (learning rate, momentum, etc.) provided by Caffe and cited articles. In addition, we use two famous image classification datasets: Cifar-10 [25] and ImageNet [10]; and a typical video dataset: UCF101 [37].

Comparisons. The comparison contains two parts. First, to demonstrate the memory efficiency of the three proposed strategies, we use the original version of Caffe without dedicated memory optimization as the baseline. These experiments aim to measure the memory cost of intermediate data, which is statically allocated before starting the computation [42] in some systems such as Caffe and MXNet. We can therefore estimate the exact memory consumption of the activation data and their corresponding gradients. Second, to present the advantages of Layrub in various aspects, we also compare it with other well-known systems, such as GeePS with Caffe [8], vDNN [33], original MXNet [3], MXNet with memonger [4] (except where otherwise indicated, all experiments on MXNet discussed in this article are performed with its memory-efficient strategy memonger), TensorFlow [1]. Their latest versions are obtained from Github. In the experiments, we report the total runtime memory usage with NVIDIA System Management Interface (nvidia-smi). Note that the runtime memory cost of the parameters and extra workspace required by cuDNN are not the main concern of our evaluation, as explained in Section 2.2.

5.2 Memory Efficiency on Layrub

5.2.1 For Deep Network Models. To demonstrate the advantage of Layrub in training deep network models, we take training processes on the models of ResNet-family, including Resnet-34/50/101/152/1001, continuously increasing the number of network layers. These are all widely utilized and developed by researchers. Layrub enhances the possibility of training all deep models, without needing to consider the limitations of GPU memory. Regardless of the depth of the network, we keep the memory usage to a mostly constant level. The results presented in Figure 9 show that the GPU memory allocation remains at 759.5MB while increasing the number of network layers from 50 to 1,001.

Compared to the baseline (Caffe), Figure 9 shows that our inter-layer strategy performs better for most deep models, obtaining a spectacular memory saving proportion of up to 98.9% in ResNet-1001. However, the inter-layer strategy has its own coverage and limitation. It cannot universally achieve remarkable effects for all types of DNN structures, such as LeNet and Cifar10_quick, of which the deep feature is not so obvious. Therefore, we propose the hybrid reuse strategy to address the shortcomings of the two strategies mentioned above. The results show the

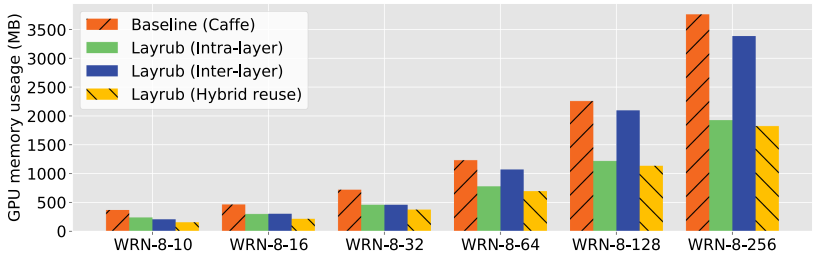


Fig. 10. GPU memory allocation for wide network models (with 128 batch size of Cifar-10 dataset).

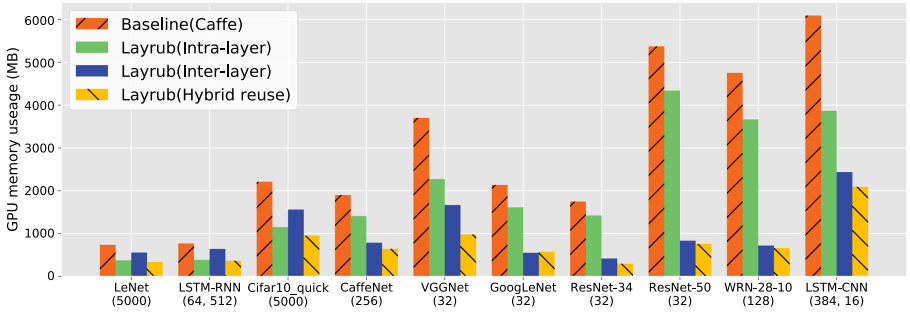


Fig. 11. GPU memory allocation for well-known DNN models. Bracket below the model name denotes the default batch size; for LSTM-RNN, it denotes batch size=64, sequence length=512; for LSTM-CNN it denotes batch size=384, train frame=16.

hybrid reuse strategy outperforms the inter-layer strategy in every model with a little but steady superiority.

5.2.2 For Wide Network Models. To highlight the potential of Layrub in training wide networks, we collect another group of DNN models by extending the width of models from 10 to 256. More specifically, we train a WRN-family with a small batch size of 128 in Cifar-10 dataset. Note that WRN-16-1 with a width of one is actually an original ResNet structure with 16-depth. Zagoruyko and Komodakis [45] study the effect of the width of layers on classification accuracy by incrementally adding more filters to these convolutional layers, such as for WRN-16-10, from 64 filters to 640 filters. We follow a similar approach to increase the width of WRN by fixing the depth of the model and gradually adding almost 20 times as many filters, obtaining the WRN configurations in Figure 10. To exclude the underlying factors arising from the depth, we decrease the depth from 16 to 8, and observe that the intra-layer strategy starts to demonstrate its potential for wide network models. In WRN-8-128 and WRN-8-256, the inter-layer strategy performs poorly and only achieves average memory reduction of 13.6%. Conversely, the intra-layer strategy can reduce the memory usage to a stable level close to 50%. This is consistent with the results shown in Figure 11, in which the effect of the inter-layer reuse strategy falls behind that of the intra-layer strategy on LeNet, Cifar10_quick, and LSTM-RNN, neither of which are deep models.

Note that the in-place optimization that is utilized by the original Caffe also has been involved in Layrub. It differs substantially from our intra-layer strategy, because in-place optimization only works for element-level operations, but not for the complex matrix operations such as convolution. Beyond that, our intra-layer strategy also outperforms the memory sharing optimization utilized in MXNet, since memory sharing happens among arbitrary two data without considering whether their memory footprint equals to each other. In contrast, the intra-layer strategy happens when

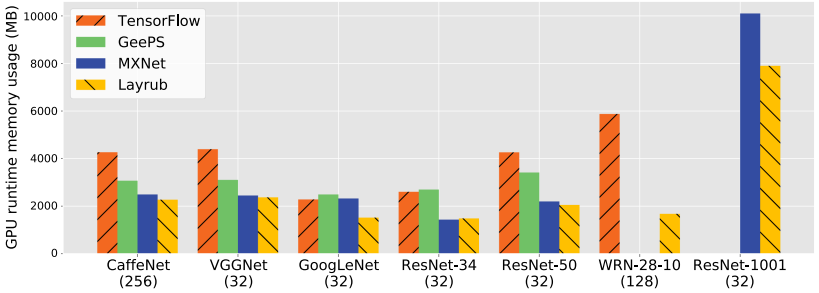


Fig. 12. Total runtime memory consumption for different systems.

these two data have same size, which can achieve a far more precise and accurate memory savings; more details can be found in Section 5.3.1.

5.2.3 For Well-Known DNN Models. The analysis above leads to the conclusion that both the intra-layer and the inter-layer strategy have their own advantages, while accompanied with some drawbacks in the meantime. We therefore propose the hybrid memory reuse strategy, which makes two strategies complementary to each other. Figure 11 confirms that the hybrid strategy is suitable for almost types of networks. It reduces memory usage from 55% to 98.9% during the training processes of these well-known models. For GoogLeNet, the memory allocation on hybrid strategy is slightly higher than that on inter-layer strategy because of a special regularization layer LRN, which consumes the largest memory footprint on GoogLeNet. Therefore, we cannot deploy intra-layer strategy easily. In this condition, the hybrid strategy turns into the inter-layer strategy, and the extra memory footprint is caused by the intra-layer that needs space for memory-efficient ReLU. Fortunately, since the extra memory footprint is only 24MB, the side-effect is almost negligible. Due to the memory efficiency of the hybrid strategy, we regard it as the core contribution of our article.

Insights: To demonstrate memory efficiency on Layrub, we use three strategies of Layrub for the training process on a K40M GPU, and compare all results to the baseline Caffe. The first two memory reuse strategies (the intra-layer reuse and inter-layer reuse) effectively alleviate the memory pressure on the two types of networks, deep models and wide models. The third strategy, which hybridizes the two strategies, achieves a near-constant proportion of memory allocation across most of the DNNs.

5.3 Comparison with Well-Known Systems

5.3.1 Memory Efficiency Comparison. We compare Layrub with other well-known systems that have some explicit memory optimization policies, including GeePS [8], MXNet with memonger [4] (MXNet for short), and TensorFlow [1]. To obtain the maximum memory savings, unless otherwise stated, we employ the hybrid reuse strategy as Layrub by default in the next sections. Since the memory optimization policies of these systems are different, we evaluate the total runtime memory consumption of all the well-known models on four systems. In particular, to realize data migration, GeePS does double buffering with the memory pool that is twice as large as the memory space required by the activation data. It means that, compared with Layrub, GeePS consumes almost double the memory for activation data. It can be observed in Figure 12 that Layrub uses 1482MB GPU runtime memory for ResNet-34, while GeePS uses 2705MB. For this reason, GoogLeNet(128) and VGGNet-16(64) in Figure 13 cannot be trained in GeePS due to being out of memory. TensorFlow has its memory allocator, which implements a greedy and best-fit with coalescing policy. Some

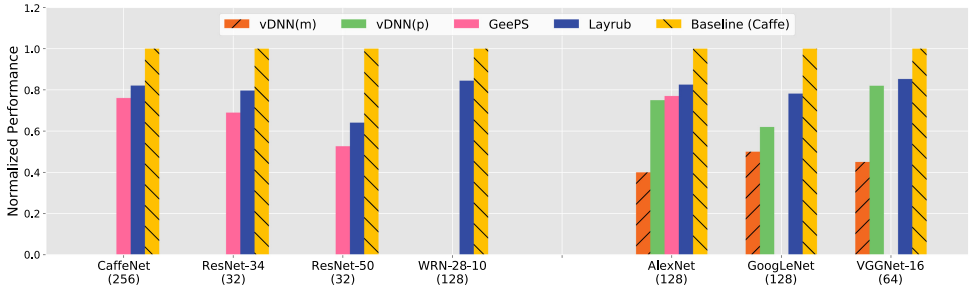


Fig. 13. Normalized performance comparison on well-known models (normalized to the baseline (Caffe), and the closer to 1 the better).

systems are unable to perform the training processes for all models, we therefore show the results for seven models to prove the memory-efficiency of Layrub.

Our strategy achieves runtime memory savings of 40.6% on average and up to 69.2% over GeePS, and 50.1% on average and up to 71.5% over TensorFlow. Moreover, compared with MXNet, Layrub still performs well by achieving runtime memory savings of 35.2% on GoogLeNet. It is noted that among so many memory-efficient systems, only MXNet and Layrub can train ResNet−1001 on one GPU with 12GB memory. We give more evaluations on the extreme-scale training in Section 5.4.

5.3.2 Time Performance Comparison. Our intra-layer strategy has a near-zero effect on time overhead. However, the best memory efficiency achieved is no higher than 50%. In comparison, our inter-layer strategy can obtain a spectacular memory saving proportion by reusing the data space across adjacent layers. However, accompanying by the extreme memory efficiency, this strategy also introduces an additional performance degradation in terms of training time due to the data migration.

Comparison of Layrub, GeePS, and vDNN. To illustrate the time performance on Layrub, we compare it with the state-of-the-art works GeePS [8] and vDNN [33], which also utilize CPU-GPU transfer policy to achieve the memory efficiency. They use the technology of GPU memory pool, which is not adopted by us considering its extra overhead. GeePS does double buffering with the memory pool to realize time-efficient data migration; vDNN allocates and releases memory asynchronously in the memory pool; they all result in obvious performance loss.

To make a direct comparison between Layrub, GeePS, and vDNN (which is not yet open-source), we adopt the performance-time[1] criterion $P_{opt} = \frac{T_{base} \times P_{base}}{T_{opt}}$ for calculating the normalized time performance, which is the same as that used in vDNN. Since Layrub, GeePS, and vDNN are all developed on the original Caffe, we take the original Caffe as the baseline of the normalized performance comparison. The normalized performance of baseline (Caffe) equals to 1 — the closer to baseline the better. Taking the VGGNet-16(64) in Figure 13 as an example, the normalized performance in memory-optimal policy of vDNN (expressed as vDNN(m) in Figure 13) is lower than 50%, while ours is 83%. Even though the memory efficiency on Layrub and vDNN(m) is all about the same, the performance overhead of vDNN(m) is much more serious. In addition, compared with the performance-optimal policy of vDNN (expressed as vDNN(p) in Figure 13), which sacrifices the memory efficiency for the time performance, Layrub still has advantages on the time efficiency over the vDNN. Therefore, instead of the memory pool, we repeatedly reuse the fixed memory

[1]We denote P as the performance of a system, T as the execution time of a system. In the article, P_{base} is the performance of baseline ($P_{base(Caffe)} = 1$), P_{opt} is the performance of memory-efficient systems including Layrub, GeePS, vDNN, and MXNet; T_{base} and T_{opt} are the execution time of them, respectively.

Table 5. Training Time (Sec) Comparison on Well-Known Models (on the K40M with 20 Iterations)

Model	MXNet			Layrub			TensorFlow
	$T_{base(M)}$	$T_{opt(M)}$	Ratio	$T_{base(L)}$	$T_{opt(L)}$	Ratio	$T_{base(T)}$
LeNet	2.373	2.783	+17.28%	2.148	2.342	+9.05%	5.143
Cifar10_quick	5.478	7.565	+31.61%	7.223	8.178	+13.23%	–
AlexNet	27.343	34.393	+25.78%	20.808	25.355	+21.85%	19.161
VGGNet	44.209	57.224	+29.44%	91.815	110.759	+20.63%	64.060
GoogLeNet	8.926	11.233	+25.85%	9.726	12.392	+27.41%	–
ResNet-34	13.897	17.849	+28.44%	37.387	46.928	+25.52%	10.101
ResNet-50	17.947	22.492	+25.32%	55.131	86.425	+56.76%	15.951
WRN-28-10	–	–	–	79.397	94.021	+18.42%	42.394
ResNet-1001	–	324	–	–	712.200	–	–
ResNet-1517	–	–	–	–	1148.860	–	–

$T_{base(M)}$ denotes the training time of original MXNet; $T_{opt(M)}$ denotes the training time of MXNet with Memonger; $T_{base(L)}$ denotes the time of original Caffe; $T_{opt(L)}$ denotes the time of Layrub; $T_{base(T)}$ denotes the time of TensorFlow, which has no explicit memory optimization; – means that the system has not supported the designated model yet, or the system lacks enough memory space to train this model.

spaces (at least three pieces of spaces), and utilize the `Tracer` and streams to achieve both better memory utilization and time efficiency.

Comparison of Layrub, MXNet, and TensorFlow. We also compare Layrub with MXNet and TensorFlow in Table 5, which are state-of-the-art systems. Since they are different frameworks of different running performance, we execute the training process under one epoch (an epoch is a single pass through the full training set), and record the average execution time of 20 iterations. The original versions of Caffe, MXNet, and TensorFlow are regarded as the baseline without any explicit memory optimization strategy. According to the results, in Layrub the proportion of extra overhead required is at least 9.05% of the original training time of the baseline Caffe, and the average proportion of extra time is 24.1% for eight well-known models. By contrast, in the optimized MXNet, the extra overhead is added at least 17.28% compared with MXNet without optimization, and the average proportion of extra overhead is 26.3%. Compared to them, although TensorFlow outperforms on time efficiency for some models, its memory efficiency performs the worst due to lack of effective memory optimization strategy, and the extreme-scale models such as ResNet-1001 cannot be trained in TensorFlow.

Time Overhead Analysis on Layrub and MXNet. Data migration inevitably leads to performance overhead due to the restricted bandwidth of PCI-e, which interconnects CPU and GPU. Therefore, Layrub launches a separated CUDA stream to complete the data migration, jointly executes multiple streams to overlap the computation and data migration, and properly hides the latency as much as possible. By taking re-computation to trade memory space, MXNet also has substantial latency that is hard to be hidden. For a clearer comparison between Layrub and MXNet, a common T_{base} is introduced in Figure 14, which is treated as a common baseline with no data migration or no extra re-computation. The unhidden overhead in Layrub is shown as the blue bars in Figure 14. Since the main bottleneck is the PCI-e with 16GB/s bandwidth, the time overhead in general will decrease significantly with the use of a faster communication link between the CPU and GPU (i.e., NVLink and next-Gen NVLink, which provide at least 150GB/s bandwidth). As a promising new direction to solve the problem of overhead, we will verify the practical effect of the NVLink techniques as long as the necessary experimental environment is ready.

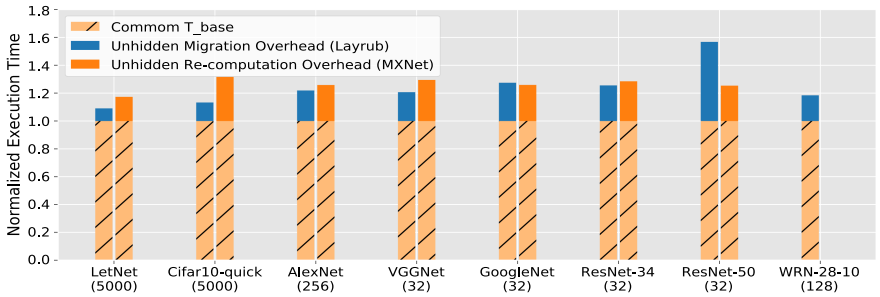


Fig. 14. Normalized execution time on Layrub and MXNet (common T_{base} here is a common baseline for both MXNet and Layrub, which is normalized to 1).

Table 6. Runtime Memory Usage for Extreme-Scale Models (with 32 Batch Size of ImageNet Dataset)

	Titan Z			K40M		
Model	**Caffe**	**MXNet**	**Layrub**	**Caffe**	**MXNet**	**Layrub**
ResNet-34	4426MB	1554MB	1251MB	4430MB	1431MB	1257MB
ResNet-50	OOM	2298MB	1802MB	11425MB	2195MB	1805MB
ResNet-302	OOM	4873MB	3444MB	OOM	4911MB	3448MB
ResNet-695	OOM	OOM	5964MB	OOM	7885MB	5968MB
ResNet-1001	OOM	OOM	OOM	OOM	10104MB	7901MB
ResNet-1172	OOM	OOM	OOM	OOM	11455MB	9022MB
ResNet-1304	OOM	OOM	OOM	OOM	OOM	9872MB
ResNet-1517	OOM	OOM	OOM	OOM	OOM	11239MB

Insights: To achieve time efficiency, CUDA stream technique is utilized to complete the data migration and hide the latency. Furthermore, the NVLink interconnect will be another significant technique to fundamentally reduce the time overhead. Layrub can obtain the best memory efficiency for most networks at the moderate cost of 24.1% on average higher training time. Compared with other prior work such as vDNN, GeePS, and MXNet, Layrub has advantages on both the time efficiency and the memory efficiency over it.

5.4 Extreme-Scale Models

The ResNet-family is taken as a case study for the extra-deep models, each one of which has at least dozens of GB of memory footprint. With regard to the selection rule for the depth of ResNet models, such as 1001 or 1517, we follow the instruction [18] given by the authors of ResNet. We take MXNet as a contrast, since the experimental results show that for the rest of the comparison systems, it is impossible to train the deep models with more than 1,001 layers with a single GPU. In addition, we use two types of GPUs that have different memory capacity, as described in Table 4.

Table 6 demonstrates the superiority of Layrub for training extreme-scale models on different hardware environments (in Table 6, OOM means out of memory). When the depth of the model increases to 1,517, Layrub is still able to make the training process work well on the K40M. The corresponding runtime memory consumption of ResNet-1517 is 11,235MB on the K40M. In contrast, MXNet can train the extra-deep ResNets when the depth of these is no greater than 1,172. Our results show that for MXNet, the runtime memory consumption on ResNet-1172 is 11,455MB, which is very close to the upper boundary of K40M memory of 11580MB. Overall, as shown in

Table 7. Extreme-Scale Batch Size and Training Time in Layrub (on K40M with 1-Epoch, Models All Use ImageNet Dataset, Except That ResNet-56 and WRN-28-10 Use Cifar-10); BS_{norm}: Normal Batch Size Following the Default Configuration of Models; $BS_{extreme}$: Extreme-Scale Batch Size

Model	Extreme-Scale Batch Size			Training Time in Layrub		
	BS_{norm}	$BS_{extreme}$	Ratio	BS_{norm}	$BS_{extreme}$	Ratio
AlexNet	128	2100	+1641%	1h 51m	1h 43m	−8.19%
VGGNet	64	240	+333%	30h 34m	29h 18m	−4.16%
ResNet-34	32	416	+1300%	28h54m	25h 21m	−13.25%
ResNet-50	32	272	+850%	53h54m	50h 31m	−6.26%
ResNet-56	256	5000	+1900%	4m 3s	3m 43s	−9.16%
WRN-28-10	128	1376	+882%	35m	32m	−8.64%

the Table 6, Layrub is very robust and provides a significant improvement in runtime memory consumption on multiple many-core architectures. For instance, the largest number of depth in ResNets has been extended to 695 on Titan Z, 1,517 on K40M.

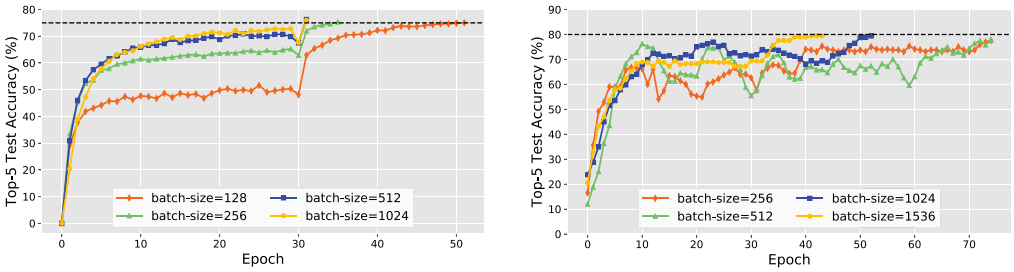
Insights: To break down the GPU memory barrier when training extreme-scale models, and allow designers to explore deeper or wider network architectures, the hybrid reuse strategy of Layrub is chosen to obtain the most memory efficiency for training processes of some extreme-scale models.

5.5 Extreme-Scale Data Batch Size

Our primary goal is to enable DL designers to explore possible neural architectures within the limited GPU memory. The exploitations consist of not only extremely scaling up the depth or width of networks, but also extremely scaling up the batch size of datasets for networks. Scaling up the batch size inside a GPU can significantly improve the GPU resource utilization and accelerate the convergence of a model. It is therefore essential to expand the batch size under the constraint of GPU memory resources. However, the quantities of memory footprint vary linearly along with the batch size of training data. For example, when training the ResNet-family, the batch size of ImageNet dataset for a 12GB GPU is generally set as 32 or 16 due to the memory limitation.

Fortunately, with the help of our memory reuse strategy, the batch size of ImageNet dataset for the well-known models can expand 16.4 times larger than the normal batch size in the original Caffe, as shown in Table 7. In addition, we also compare the training time of one epoch for six medium-scale models in Layrub when scaling the batch size from a normal size (BS_{norm}) to an extreme-scale size ($BS_{extreme}$). The experimental results demonstrate that the proposed memory reuse strategy does not incur noticeable performance loss after scaling the batch size. Instead, the training time for six models including ResNet-34, ResNet-50, and WRN-28-10 all have an obvious decrease. Table 7 shows that the training time per epoch for ResNet-34 decreases 13.25% while scaling the batch size from 32 to 416. This is mainly because the computing resources of GPU are fully utilized by the data of a larger batch size, and the latency of data migration has been mostly hidden inside the more time-consuming feedforward computation.

End-to-End Training Results. Another potential advantage of scaling up a larger batch size is likely to speed up the convergence of a model, and reduce the entire training time consumption by using less epochs. Figure 15 demonstrates an end-to-end training comparison of Layrub on AlexNet and ResNet-56 with scaling batch sizes. To achieve the similar test accuracy in Layrub, AlexNet with batch size = 1024 only uses 31 epochs, while AlexNet with batch size = 128 has to use 53 epochs. Although the training time per epoch of Layrub is about 23% (refer to Tables 5 and 7) higher than that of Caffe with no memory optimization, Layrub still has about 1.4 times speedup



(a) AlexNet, ImageNet dataset (b) ResNet-56, Cifar-10 dataset

Fig. 15. Training curves of different batch sizes on two models without data augmentation.

Table 8. Execution Time and Speedup of Distributed Layrub on Three Models (on K40M with 1-Epoch, Models All Use ImageNet Dataset)

Model	Distributed Support		
	Layrub (one node)	Layrub (two nodes)	Speedup
AlexNet(256)	1h 47m	54m	1.98×
ResNet-50(32)	53h 16m	26h 54m	1.98×
ResNet-1001(32)	395h 25m	201h 3m	1.97×

compared with Caffe, which means that the overall training time in Layrub significantly decreases by scaling to a proper batch size. Similar results can be gotten by the training curve of ResNet-56.

Insights: The purpose of the proposed memory reuse strategies is not only to enable extreme-scale models to be trained on a single GPU, but also to extend the batch size of training data for some medium-scale models during the training processes. After scaling up the batch size to a proper size, the computing sources of GPU can be fully utilized, and the model convergence can be accelerated.

5.6 Scale with Multiple Nodes

In this subsection, we evaluate the memory efficiency and accelerating performance of Layrub with multiple nodes. Since Caffe-MPI [21] is a parallel version of Caffe for multi-nodes GPU cluster, we implement our strategies on Caffe-MPI to provide the distributed support. Table 8 presents the execution time comparison of Layrub when using one node or two nodes (each node with one K40M GPU). In addition, we also list the speedup ratios of distributed Layrub on three representative models. As shown in Table 8, Layrub using two nodes achieves a speedup of 1.98 times on AlexNet compared to that using single node. Implemented on the basis of Caffe-MPI, Layrub is exactly the one that achieves both steady and near-linear speedup of 1.97 times for all three models. Although we have not evaluated with more nodes yet due to the limitation of experimental environment, we still can predict that the large-scale models would be accelerated in a high ratio by using more nodes with the optimization of our strategy.

Insights: the proposed memory optimization can be implemented in the multi-nodes environment, and tackle the memory issue of extreme-scale models. Furthermore, the proposed strategies have no side-effect on the accelerating performance when scaling with multiple nodes.

6 DISCUSSION

The core of our inter-layer strategy is data migration, which trades extra communication with memory space. Another alternative is the re-computation strategy, which trades extra

computation with memory space. Although both of them achieve extraordinary memory efficiency, they are not optimal due to the overlook of layer characteristics hidden in the network architectures, which means that they only work for certain types of layers. Taking the AlexNet as an example, based on our investigation, data migration outperforms re-computation in all feature extraction layers (including convolutional and full-connected layers) by averaged 6.5x speedup, while performs worse in the rest of the layers (including activation and regularization layers, i.e., relu and normalization layers). Therefore, it requires a method to bridge the gap between the layer characteristics and the current memory optimization strategies, which will be one of our future works.

Memory allocation for DNNs can be treated as a less general case of the graph optimization problems where the graph of forward computation is a chain and gradients have to be chained in a reverse order [15]. This simplification leads to relatively simple heuristics and memory saving strategies, which have been implemented by this article. To explore all the opportunities of memory saving on the training process, we investigate the overall memory consumption of almost all types of data required for training, including training data, parameter data, activation data and their counterpart gradients, cuDNN workspace data, and cuDNN handle data. It shows that for most models, the activations and their gradients consume the majority of memory (e.g., 85% in GoogLeNet, 96% in ResNet-50), which supports our motivation to reuse their memory footprint. After the activations are optimized by our strategies, the memory consumption of remaining data becomes hard to ignore (e.g., both cuDNN workspace data and handle data consume highly to 1GB memory in ResNet-152), which will be considered in our future work.

Given the popularity of TensorFlow over other frameworks, it is necessary to give some indications of how easily Layrub can be integrated into TensorFlow. The Tracer (described in Section 4.1) can be introduced to the memory allocator of TensorFlow to record the memory access sequence of tensors. In the forward phase, when the Ref is equal to 0, the corresponding tensor can be offloaded from GPU memory to CPU. Then the allocator can allocate and reuse its memory space to the latter tensor. In the backward phase, when the tensor is required by a node, it can be loaded into GPU memory in advance. By migrating almost all tensor data to CPU memory, it can achieve the best memory efficiency for TensorFlow.

7 RELATED WORK

There have been various efforts aiming to reduce the memory usage of DNN models. Model quantization and compressed learning techniques [14, 17] have been used to obtain low-precision weights and activations to reduce memory footprint. Network pruning [16, 17] is another approach removing little-used or low-weight connections, and also achieves a reduction in model size. However, since researchers strive to achieve the highest accuracy, methods that lead to accuracy degradation are not advisable from the perspective of systematic optimization. Our discussion will stay within a certain range of accuracy-preserving methods. Layrub is a system-level optimization that re-organizes the data placement on the training process, with no side-effect on model accuracy.

Several prior works [8, 33] discuss mechanisms to support memory reduction in the training process. GeePS [8] and vDNN [33] both employ a CPU-GPU transfer to deal the data used in training to mitigate the memory pressure on GPUs. A similar work, SuperNeurons [43], dynamically allocates the memory for convolution workspaces to achieve the high performance. Compared with them, Layrub selects a fixed number of the largest pieces of memory space and reuses them repeatedly over the entire training process, which can achieve better memory utilization.

Several frameworks [1, 3, 7] have been developed to analyze and accelerate DNN models. For example, TensorFlow and MXNet both represent the data dependencies of deep learning using a dataflow graph, thus implicitly enabling auto-parallelization. However, even though they use

a computation graph as a descriptive approach, they still tend to use coarse-grained forward and backward operations to make the graph simpler [4]; this actually cannot be separated from a layer-centric manner, and thus is able to utilize Layrub to optimize the memory usage. With respect to memory optimization, frameworks such as Torch, MXNet, and Caffe2 [12] all implement similar memory sharing optimizations based on a computation graph analysis. However, it should be noted that our approach involves an inter-layer space reuse strategy that achieves a very high proportion of memory savings by offloading/fetching data to/from CPU memory. It can be orthogonally utilized with the above optimization planning to obtain further gains.

8 CONCLUSION

We preset Layrub, a data placement strategy for extreme-scale deep learning on GPUs. Layrub focuses on extreme memory optimization for the training processes of different shapes of DNN models. We show empirically that Layrub can reuse a constant amount of GPU memory space regardless of the depth of the network. It can even train a 1,517-layer ResNet by reducing the memory allocation from roughly 100GB to 759.5MB. We further demonstrate the advantages of Layrub on a variety of DNN models and datasets, by comparing it favorably to GeePS, vDNN, MXNet, and TensorFlow. Unlike the current systems, which rely on the users to decide whether the model can fit into the GPU, Layrub can handle the extreme-scale models in a more elastic manner, even with moderate training overhead. In addition, Layrub can be compatible with TensorFlow, MXNet, and other frameworks, since the layer-centric strategies presented in Layrub could be used to maintain memory efficiency in them. In the future, a further study could be developed to ensure both the memory and time efficiency for distributed extreme-scale training.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Savannah, GA, 265–283.

[2] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math compiler in Python. In *Proceedings of the 9th Python in Science Conference (SciPy'10)*. Austin, Texas, 1–7.

[3] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Proceedings of Workshop on Machine Learning Systems at the 28th Annual Conference on Neural Information Processing Systems (LearningSys'15)*. Montreal, Canada, 1–6.

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv:1604.06174* (2016).

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. *arXiv:1410.0759* (2014).

[6] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Broomfield, CO, USA, 571–582.

[7] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. 2002. *Torch: A Modular Machine Learning Software Library*. Technical Report EPFL-REPORT-82802. Idiap, Martigny, Valais, Switzerland.

[8] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. 2016. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. ACM, London, UK, 1–16.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *Proceedings of the 25th Annual Conference on Neural Information Processing Systems (NIPS'12)*. Curran Associates, Inc., Lake Tahoe, Nevada, 1223–1231.

[10] Jia Deng, Wei Dong, Richard Socher, Lijia Li, Kai Li, and Feifei Li. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of the 22nd IEEE Conference on Computer Vision and Pattern Recognition (CVPR'09)*. IEEE, Miami, F, 248–255.

[11] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. 2015. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR'15)*. IEEE, Boston, MA, 2625–2634.

[12] Facebook. 2017. Caffe2. Retrieved December 13, 2017 from https://caffe2.ai.

[13] Qichuan Geng, Zhong Zhou, and Xiaochun Cao. 2017. Survey of recent progress in semantic image segmentation with CNNs. *Science China Information Sciences* 61, 5 (2017), 1–18.

[14] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. 2014. Compressing deep convolutional networks using vector quantization. *arXiv:1412.6115* (2014).

[15] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-efficient backpropagation through time. In *Proceedings of the 29th Annual Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., Barcelona, Spain, 4125–4133.

[16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. ACM, Seoul, Korea, 243–254.

[17] Song Han, Huizi Mao, and William J. Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *Proceedings of the 4th International Conference on Learning Representations (ICLR'16)*. San Juan, Puerto Rico, 1–14.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. IEEE, Las Vegas, NV, 770–778.

[19] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. 2016. Deep networks with stochastic depth. In *Proceedings of the 14th European Conference on Computer Vision (ECCV'16)*. Springer, Amsterdam, Netherlands, 646–661.

[20] Forrest Iandola. 2016. *Exploring the Design Space of Deep Convolutional Neural Networks at Large Scale*. Ph.D. Dissertation. University of California, Berkeley.

[21] Inspur. 2017. Caffe-MPI. Retrieved August 24, 2018 from https://github.com/Caffe-MPI/Caffe-MPI.github.io.

[22] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia (MM'14)*. ACM, Orlando, Florida, USA, 675–678.

[23] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Feifei Li. 2014. Large-scale video classification with convolutional neural networks. In *Proceedings of the 27th IEEE Conference on Computer Vision and Pattern Recognition (CVPR'14)*. IEEE, Washington, DC, 1725–1732.

[24] Jan Koutník, Klaus Greff, Faustino Gomez, and Jürgen Schmidhuber. 2014. A clockwork RNN. In *Proceedings of the 31st International Conference on International Conference on Machine Learning (ICML'14)*. JMLR, Beijing, China, 1863–1871.

[25] Alex Krizhevsky and Geoffrey E. Hinton. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th Annual Conference on Neural Information Processing Systems (NIPS'12)*. Curran Associates, Inc., Lake Tahone, Nevad, 1097–1105.

[27] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[28] Ignacio Lopez-Moreno, Javier Gonzalez-Dominguez, Oldrich Plchot, David Martinez, Joaquin Gonzalez-Rodriguez, and Pedro Moreno. 2014. Automatic language identification using deep neural networks. In *Proceedings of the 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'14)*. IEEE, Florence, Italy, 5337–5341.

[29] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, and Trevor Cohn. 2017. DyNet: The dynamic neural network toolkit. *arXiv:1701.03980* (2017).

[30] NVIDIA. 2017. CUDA toolkit 8.0 documentation: profiler. Retrieved December 13, 2017 from http://docs.nvidia.com/cuda/profiler-users-guide/index.html#axzz4p7v1jezC.

[31] NVIDIA. 2017. Introduction to cuBLAS. Retrieved December 13, 2017 from http://docs.nvidia.com/cuda/cublas/index.html#axzz4oxZMgelu.

[32] Zhaofan Qiu, Ting Yao, and Tao Mei. 2017. Learning spatio-temporal representation with pseudo-3D residual networks. In *Proceedings of the 2017 IEEE International Conference on Computer Vision (ICCV'17)*. IEEE, Venice, Italy, 5534–5542.

[33] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, Taipei, China, 1–13.

[34] Haşim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Proceedings of the 15th Annual Conference of the International Speech Communication Association (InterSpeech'14)*. Singapore, 338–342.

[35] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*. ACM, San Francisco, California, USA, 2135–2135.

[36] Karen Simonyan and Andrew Zisserman. 2016. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*. San Diego, USA, 1–14.

[37] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. 2012. *UCF101: A Dataset of 101 Human Actions Classes from Videos in the Wild*. Technical Report CRCV-TR-12-01. University of Central Florida, Orlando, FL.

[38] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[39] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. 2017. Inception-v4, inception-ResNet and the impact of residual connections on learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'17)*. AAAI Press, San Francisco, California, 4278–4284.

[40] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR'15)*. IEEE, Boston, MA, 1–9.

[41] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. IEEE, Las Vegas, NV, 2818–2826.

[42] Marc Gonzalez Tallada. 2016. Coarse grain parallelization of deep neural net works. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. ACM, Barcelona, Spain, 1–12.

[43] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*. ACM, Vienna, Austria, 41–53.

[44] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the 30th IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*. IEEE, Honolulu, Hawaii, 5987–5995.

[45] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide residual networks. In *Proceedings of the 27th British Machine Vision Conference (BMVC'16)*. BMVA Press, York, UK, 87.1–87.12.

Received January 2018; revised May 2018; accepted July 2018