Layup: Layer-adaptive and Multi-type Intermediate-oriented Memory Optimization for GPU-based CNNs

WENBIN JIANG, YANG MA, BO LIU, and HAIKUN LIU, Huazhong University of Science and Technology
BING BING ZHOU, The University of Sydney
JIAN ZHU, SONG WU, and HAI JIN, Huazhong University of Science and Technology

Although GPUs have emerged as the mainstream for the acceleration of *convolutional neural network* (CNN) training processes, they usually have limited physical memory, meaning that it is hard to train large-scale CNN models. Many methods for memory optimization have been proposed to decrease the memory consumption of CNNs and to mitigate the increasing scale of these networks; however, this optimization comes at the cost of an obvious drop in time performance. We propose a new memory optimization strategy named Layup that realizes both better memory efficiency and better time performance. First, a fast layer-type-specific method for memory optimization is presented, based on the new finding that a single memory optimization often shows dramatic differences in time performance for different types of layers. Second, a new memory reuse method is presented in which greater attention is paid to multi-type intermediate data such as convolutional workspaces and cuDNN handle data. Experiments show that Layup can significantly increase the scale of extra-deep network models on a single GPU with lower performance loss. It even can train ResNet with 2,504 layers using 12GB memory, outperforming the state-of-the-art work of SuperNeurons with 1,920 layers (batch size = 16).

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

Additional Key Words and Phrases: Convolutional neural network, GPU, memory management

ACM Reference format:

Wenbin Jiang, Yang Ma, Bo Liu, Haikun Liu, Bing Bing Zhou, Jian Zhu, Song Wu, and Hai Jin. 2019. Layup: Layer-adaptive and Multi-type Intermediate-oriented Memory Optimization for GPU-based CNNs. *ACM Trans. Archit. Code Optim.* 16, 4, Article 39 (October 2019), 23 pages.
<https://doi.org/10.1145/3357238>

This work is supported by the National Natural Science Foundation of China under Grants No. 61672250 and No. 61672251. 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: W. Jiang, Y. Ma, B. Liu, H. Liu (corresponding author), J. Zhu, S. Wu, and H. Jin, National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China; emails: {wenbinjiang, yangm, borenaliu, hkliu, janezhu, wusong, hjin}@hust.edu.cn; B. B. Zhou, Centre for Distributed and High Performance Computing, School of Information Technologies, The University of Sydney, NSW 2006, Australia; email: bing.zhou@sydney.edu.au.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/10-ART39

<https://doi.org/10.1145/3357238>

1 INTRODUCTION

Deep learning (DL) has become increasingly important in many applications [26, 29, 37, 38] and has benefited from the popularity of *deep neural networks* (DNNs). As the dominant DNNs, *Convolutional neural networks* (CNNs) are used both for computer vision [16] and certain other applications [44]. *Graphics processing units* (GPUs) [15] have emerged as the main computational workhorse for these applications, due to their high performance, low cost, and wide availability [7]. Of course, higher accuracy is always the primary goal of CNN models, and this is often accompanied by larger amounts of training data and more complex models. Unfortunately, there is a critical limitation in GPUs: the entire data used for GPU computation must be put into GPU memory for execution. Since the physical memory of GPUs is usually very limited, the data scales of CNNs often exceed their memory. In this case, memory optimization solutions must be sought.

A straightforward solution to overcome this restriction is to apply a model-parallel approach [6, 12]. This means that a large network model is divided into several components and deployed on a corresponding number of GPUs. However, in practice, model parallelism is rarely used in popular DL systems such as Caffe [24], TensorFlow [1], MXNet [3], and others [2, 8, 14, 30, 33, 35, 45], since it would introduce much more complexity into the construction of distributed models and much greater communication overhead, which could significantly reduce the process performance.

The mainstream solution is to reduce the memory consumption of DL training processes as far as possible. Reusing feature maps (a type of intermediate data) to save memory usage is a classic method that has been studied in many existing works. vDNN [32], GeePS [11], and Layrub [25] utilize CPU memory as a limitless assistant memory for the temporary storage of intermediate data. By using a CPU-GPU transfer, the memory usage in the GPU can be significantly decreased. In contrast, Chen et al. [4] execute extra forward computation to trade off the memory space of feature maps. These works usually pay main attention to the efficiency of memory consumption; however, the execution time efficiency of these methods has been significantly overlooked. In addition, since there are many types of intricate intermediate data, the potential of memory reuse methods has not yet been fully exploited. Our investigative study shows that there are still two issues that have significant impacts on the training performance and memory efficiency of CNNs and which urgently need to be addressed.

The first is that a single memory optimization (CPU-GPU transfer or extra forward computation) cannot handle the diversity of the different types of layers, which causes obvious performance degradation. Since a CNN model usually consists of different types of layers (such as convolutional, pooling, and *rectified linear unit* (ReLU) layers), the type and scale of a layer determine the memory allocation pattern and have critical impacts on performance. This issue has long been overlooked by existing memory optimization works. Our investigation indicates that the layer type significantly affects the execution time of memory optimizations. Figure 1 illustrates a comparison of execution time between the two most popular memory optimizations on two popular networks (AlexNet and VGGNet) at layer-wise level. It is clear that these optimizations give rise to very different execution times for these types of layers; in other words, optimizations have very different effects on different types of layers. Hence, some layers are more suitable for extra forward computation (such as ReLU and norm layers), while others are more suitable for CPU-GPU transfer (such as conv layers). In fact, based on our tests, a layer-aware performance improvement of up to 9.8× can be obtained by choosing appropriate optimization methods for each of the appropriate layers in AlexNet. However, most existing strategies only employ a single optimization technique for all CNN layers, and this kind of single, uniform optimization approach mismatches the inherent heterogeneity of CNNs with different types of layers.

Fig. 1. Performance comparison between extra forward computation and CPU-GPU transfer for CNNs.

The second issue is that there are diverse types of important intermediate data (such as feature maps, gradient maps, workspace data, and cuDNN handles) that need to be taken into account in the memory reuse of CNNs. Traditionally, feature maps and gradient maps consume the majority of GPU memory. However, our investigation shows that after certain memory optimizations are applied, significantly reducing the memory required, the memory occupied by other types of intermediate data (such as workspace data and cuDNN handles) can no longer be ignored. In fact, the latter becomes very large during DL training processes after optimization. Here, workspaces are responsible for convolutional computation and cuDNN handles are responsible for holding cuDNN contexts. In general, the memory requirements of workspaces and cuDNN handles are lower than those of feature maps and gradient maps. However, as mentioned above, it is unreasonable to ignore them after feature maps and gradient maps have been optimized. Since existing works overlook this issue, this offers a valuable opportunity for memory optimization that can significantly mitigate the memory pressure of DL training processes on GPUs.

Based on the two aforementioned findings, we present a strategy called Layup that can further optimize the GPU memory usage for CNNs. The major contributions of our work are as follows:

- We characterize the optimization costs of different CNN layers and identify the impacts of different memory-optimized methods on the execution time, from which we derive a lightweight heuristic to direct the choice of optimization methods with less time overhead.
- We propose a fast layer-adaptive memory management method that selects different memory optimizations for different layers, based on the types of layers.

- We present a multi-type data reuse strategy that improves the efficiency of the memory reuse of CNNs by exploiting as many opportunities as possible for the memory reuse of multi-type data, and particularly for gradient maps, workspaces, and cuDNN handles.

We integrate Layup into Caffe, a popular DL framework, and perform rigorous evaluations and an analysis of the results on various representative networks. Experiments show that Layup can significantly extend the scale of extra-deep network models on a single GPU, with relatively small loss of performance. For ResNet, we can even extend this to 2,504 layers on a single GPU with 12GB memory (batch size = 16). We have opened the source codes of Layup on GitHub.¹

2 RELATED WORK

Memory shortage is always a key problem when training large DL models on GPUs. Various methods have been presented by different researchers to address this issue.

Model compression technology reduces memory costs by reducing the redundancy of parameters, such as via pruning [18, 19] and quantification [9, 10]. The former removes unimportant weights in the network, while the latter uses low-precision weights in a forward or backward pass. However, these methods are not suitable as system-level optimizations. The advantage of model compression technology is mainly reflected in the deployment of neural networks in embedded devices. The benefits of compression technology are very limited in model training, due to the small proportion of the memory usage of the parameters.

Model parallelism [7, 12] is an important approach involving a divide-and-conquer strategy. In this method, a CNN model is explicitly divided into several smaller chunks and allocated to different working nodes that can run in parallel. Consequently, explicit data exchange will create very heavy communication traffic, and complex synchronizations between different working nodes make it harder to hide communication. Moreover, issues related to efficient division strategies design, workload balance across GPUs, and convergence guarantee are also critical problems for model-parallelism. This kind of method can be regarded as a heavy-weight strategy that incurs a great deal of extra overhead.

Compared with model parallelism, lightweight, memory-efficient methods become more attractive. Some of these achieve memory savings for feature maps by executing extra forward computation, beyond the regular forward and backward propagation computation, and generally use heuristics to decide where to execute extra computation, leveraging the abundant computational resources of GPUs. Typical examples are sublinear memory optimization (e.g., memonger in MXNet) [4] and extensions of this approach [17]. However, as described in the previous section, in CNNs, this extra computation may cause the execution time to deteriorate markedly for some layers, and especially for the convolutional layers. Another drawback of this strategy is the over-subscription of GPU computing resources due to the extra computation, which also degrades the execution performance of the training process.

Several recent works have introduced memory reuse to the training process to create savings in the memory usage of feature maps, and these are highly relevant to the reuse strategy proposed in this study. vDNN [32] and GeePS [11] mainly focus on CPU-GPU transfer and normally treat CPU memory as a virtual and limitless memory pool for the temporary storage of feature maps. These approaches use prefetching and offloading techniques to hide the communication cost. Layrub [25] shows that feature maps and gradient maps in the training process can be reused from algorithmic and architectural perspectives. The primary limitation of this work is that its inter-layer memory

¹<https://github.com/CGCL-codes/Layup>.

Table 1. Numbers of Different Layers in the CNN Models Used in Experiments

Layer	AlexNet	VGGNet-16	GoogLeNet	ResNet-34	ResNet-50	ResNet-101
conv	5	13	59	37	53	104
relu	7	15	61	33	49	100
norm	2	/	2	37	53	104
pool	3	5	16	2	2	2
fc	3	3	5	1	1	1
dropout	2	2	3	/	/	/
concat	/	/	9	/	/	/
scale	/	/	/	37	53	104
eltwise	/	/	/	16	16	33
Total	22	38	155	163	227	448

reuse strongly depends on the transfer bandwidth between CPU and GPU, which may introduce significant costs and long delays due to bandwidth contention.

SuperNeurons [42] is a dynamic GPU memory scheduling approach that builds a tensor cache to store certain tensors in GPU memory to maximize their reuse. The layer type is used as a criterion to decide to whether to recompute a certain tensor or to restore it from the tensor cache. However, this approach does not consider the difference in execution time between the CPU-GPU transfer and the extra forward computation for different layer types. Moreover, the performance of SuperNeurons decreases as the batch size increases.

In general, the existing works mentioned above concentrate on space optimizations but do not find effective methods to maintain ideal time performance. They are not yet able to tap into the full potential of memory optimizations (e.g., CPU-GPU transfer and extra-forward computation). Furthermore, most existing approaches primarily focus on feature maps and gradient maps. Unlike these, the strategy proposed here tries to achieve both better execution time and higher memory efficiency by presenting a more adaptive layer-wise memory optimization and considering more intermediate data.

3 CHARACTERIZATION AND IMPLICATION

In this section, we make a comprehensive characterization of existing methods for memory reuse, and some implication is discussed to make our coming proposed strategy more understandable. In Section 3.1, we show our experimental setup. Then, we delve into the cost of computation and communication in CNN layers to make exploration on the limitations of existing memory-optimized methods in Section 3.2. In Section 3.3, we clarify the types and scale of the intermediate data that have the potential to be optimized. Our in-depth comprehensive characterization sheds light on some neglected realities and summarizes several insights into solving these inefficiencies.

3.1 Experimental Setting

Our experiments for the characterization use the famous image dataset ImageNet [13]. Benchmark CNN models include AlexNet [27], GoogLeNet [40, 41], VGGNet [36], and the ResNet family [20, 22, 43], which cover a wide spectrum of problems and networks. Table 1 gives the statistical information about the layers chosen from these networks. The convolutional layers make up the largest proportion of the total layers. The activation layers show a similar trend to the convolutional layers in all of the networks, and these are always accompanied by normalization and scaling layers. In AlexNet, *Local Response Normalization* (LRN) is used for normalization layer, while in ResNet, *Batch Normalization* (BN) is used [23].

Table 2. Experimental Environment

GPU	CPU	RAM	Software Configuration
NVIDIA Tesla K40m-12GB	Intel Xeon E5-2620 @ 2.40GHz, 24 cores	256GB	cuDNN v5.1, CUDA toolkit 7.5, Ubuntu 14.04
NVIDIA Tesla P100-PCIE-16GB	Intel Xeon E5-2680 v4 @ 2.40GHz, 28 cores	256GB	cuDNN v6.0, CUDA toolkit 8.0, Ubuntu 14.04

Fig. 2. Performance comparison between two memory optimizations (CPU-GPU transfer and extra forward computation) on AlexNet.

We characterize these networks and their layers using an NVIDIA Tesla K40m GPU or an NVIDIA Tesla P100-PCIE GPU, as shown in Table 2. GPU runtime information is gathered by NVIDIA Visual Profiler and NVIDIA SMI. Caffe [24] is used for the benchmark models training. We also make heavy modifications to the framework of Caffe to implement our strategies. It should be noted that although Caffe is used in the current implementation, it is easy to integrate our proposed design and optimization into other popular DL systems frameworks such as MXNet [3].

3.2 Issue 1: Performance Costs of Memory-optimized Methods

We begin with a simple performance comparison between two representative memory optimizations, CPU-GPU transfer and extra forward computation, to identify where the speedup bottlenecks are located. AlexNet is used for this comparison. Both of these memory optimizations are evaluated based on their best performance implementations [4, 25] on Caffe. As shown in Figures 2(a) and 2(b), we make the following findings: the CPU-GPU transfer outperforms the extra forward computation in the CONV1-CONV5 and FC6-FC8 layers (by an average speedup of 7.4 $\times$), but underperforms for the rest of the layers.

The differences of the two sets of layers arise in the scales of computation and memory usage. For the eight layers with better performance using the CPU-GPU transfer method, the convolutional and fully connected operations are slowed down by matrix multiplications, which are always computationally intensive. The rest of the layers show better performance using extra forward computation, since their computations are relatively sparse, and the numerous memory footprints of these layers cannot be ignored, which are always memory-intensive. To further identify the sensitivities of the optimized methods for each dimension, an analytic performance model is designed to estimate the training time.

The cost of the extra forward computation method can be obtained from Equation (1):

$$Cost_{extra-forward} = \frac{FLOPs_{Layer(i)}}{maxFLOPs \times UtilRate}, \quad (1)$$

where $FLOPs_{Layer(i)}$ is the computational intensity in an arbitrary layer i ($i = 1, 2, \dots, n$), i.e., the number of floating point operations (FLOPs) in the layer; $maxFLOPs$ denotes the GPU computing capability; $UtilRate$ is the percentage of the GPU computing capability that can be demanded.

The number of FLOPs in each layer is easily counted and can be calculated by the number of matrix multiplication operations. Here, we take the convolution layer as an example. The number of floating point operations is:

$$FLOPs_{Conv} = 2H_F W_F C_F \times H_O W_O \times N_F \times N, \quad (2)$$

where N , H_O , and W_O are the batch size, height, and width of the output data, respectively; N_F , H_F , W_F , and C_F are the number, height, width, and channel of the filters, respectively. For other types of layers, their FLOPs can also be obtained by similar ways.

The cost of the CPU-GPU transfer method is given by Equation (3):

$$Cost_{Transfer} = \frac{InputSize_{Layer(i)}}{Bandwidth}, \quad (3)$$

where $InputSize_{Layer(i)}$ is the size of the input data for layer i ($i = 1, 2, \dots, n$); $Bandwidth$ is the PCIe interconnect bandwidth or NVLink bandwidth, which depends on the hardware configuration. Since an input image or feature map is a four-dimensional matrix, $InputSize_{Layer(i)}$ in Equation (3) is $N_I \times C_I \times H_I \times W_I$, where N_I , C_I , H_I , and W_I are the batch size, the height, the width, and the channel of the input data, respectively.

A further insight is that our experimental results indicate that it is difficult for existing memory optimization solutions to guarantee execution time. A transfer strategy moves feature maps between GPU memory and CPU memory, and the communication cost depends on the scale of the data transmitted from the current layer and the bandwidth between the GPU and CPU. An extra forward computation strategy requires extra forward calculation to recover the feature maps dropped in the forward pass. This additional computation cost depends on the FLOPs required by the current layer and the computing capability of the GPU. The layer type is a dominant factor in the extra forward computation method, and the computation-sensitive layer should use CPU-GPU transfer for a better overlap between computation and transformation. The transfer-sensitive layer should use extra computation. Considering the non-uniform amounts of computation and memory requirements of the different layers, a layer-adaptive scheme is desired to achieve memory savings with lower latency.

3.3 Issue 2: Multi-type Intermediate Data in CNN Training Process

Here, we aim to clarify the types and scale of the data generated at the intermediate stage of the training process. In a typical CNN training process, there are the following types of intermediate data:

Feature map: In CNNs, the feature map is the output of a single layer in the forward pass. Specifically, the i th layer takes the $(i-1)$ -th feature map as its input to produce the i th feature map as its output.

Gradient map: The gradient map is the gradient differential of each feature map in the backward pass. For the last layer, the gradient of the cost function is derived with respect to the output of the last layer, and the gradient maps of the rest layers are derived by the chain derivation rule. The i th layer takes the i th gradient map as its input, and produces the $(i-1)$ -th gradient map as its output. In general, gradient maps correspond to feature maps and have the same size of memory footprints.

Workspace: The cuDNN library [5] provides several algorithms for accelerating convolutional computation, and the workspace is the additional memory space used by these algorithms. In a naive memory allocation strategy, the memory cost of the workspace is multiplied by the number of convolutional layers.

cuDNN handle: The cuDNN handle is a pointer to the structure that holds the cuDNN library context. The handle must be initialized by `cudaCreate()` before calling other cuDNN library functions. This initialization operation allocates hardware resources. Each cuDNN layer requires a cuDNN handle in the setup phase, and the memory cost is multiplied by the number of cuDNN layers, meaning that the memory cost for this part of the data cannot be ignored. For a more detailed discussion, see Section 4.2.

Our investigation suggests that it is necessary to consider how to optimize all of the multi-type intermediate data to realize better memory efficiency. This could be achieved by finding opportunities to reuse the memory space of these data while ensuring the performance of training process.

4 OVERCOMING GAPS: PURSUING BOTH MEMORY EFFICIENCY AND PERFORMANCE

Motivated by the characterization issues mentioned above, we present Layup, a layer-adaptive and multi-type intermediate-oriented memory optimization strategy for GPU-accelerated CNN models. Layup features three novel designs:

- A fast layer-adaptive method switch that dynamically selects different memory optimization methods for different CNN layers.
- An analytical model that enables performance cost estimation for a GPU processing node.
- A multi-type intermediate data reuse strategy that considers more types of intermediate data, such as workspaces and cuDNN handles.

4.1 Selection of Layer-adaptive Dynamic Memory Optimization Methods

In this section, we explain and discuss the mechanism used for layer-adaptive dynamic selection of memory optimization methods. We observe that the ratios of the transfer cost and the extra forward cost are different for different layers. It is reasonable to set a threshold to determine whether to select the transfer or the extra forward strategy for a specific layer. Here, we determine the threshold by comparing the cost of CPU-GPU transfer with the cost of extra forward computation:

$$Threshold_{Layer(i)} = \frac{Cost_{Transfer}}{Cost_{extra-forward}}. \quad (4)$$

By substituting the corresponding variables in Equation (4) by Equations (1) and (3), we revise the threshold equation to

$$Threshold_{Layer(i)} = \frac{InputSize_{Layer(i)}}{FLOPs_{Layer(i)}} \times \frac{maxFLOPs \times UtilRate}{Bandwidth}, \quad (5)$$

Fig. 3. Threshold analysis between data transfer and extra forward, considering the time of the computation task as the unit time.

where the first part of the right-hand side of Equation (5), $\frac{InputSize_{Layer(i)}}{FLOPs_{Layer(i)}}$, is only related to the size of the input data and the type of the layer; the second part depends on the hardware configuration of the node, which is independent of the network model. For a specific node, the second part can be regarded as a fixed coefficient.

Based on the following analysis for three kinds of situations, we divide the layers in the neural networks into two types: compute-sensitive layers and transfer-sensitive layers.

Thanks to the asynchronous concurrent execution mechanism [31] provided by CUDA, memory copy operations between the device and the host can be performed concurrently with the computing tasks of the device. This offers us the opportunity to parallelize data transfers and normal training tasks, so some or all of the time required for data transfer can be hidden. However, the extra computation must be dealt with after the normal computation, meaning that the time for extra computation cannot be hidden under the normal computation. The details of the threshold are illustrated in Figure 3.

For the situation where $Threshold_{Layer(i)} \leq 1$, that is, $Cost_{Transfer}$ is less than or equal to $Cost_{extra-forward}$ (see Figure 3(a)), the data transfer and the calculation process can completely overlap, and the overhead for data transfer can be regarded as zero. It is therefore clear that data transfer is preferable in this situation.

More attention should be paid to the situation where $1 \leq Threshold_{Layer(i)} \leq 2$ (see Figure 3(b)). In this situation, the overhead for data transfer is still less than that of one extra forward, meaning that it is still better to use data transfer to obtain the feature map than to use the extra forward process.

For the situation where $Threshold_{Layer(i)} \geq 2$ (see Figure 3(c)), the overhead for data transfer is greater than that for the extra forward approach, meaning that the extra forward approach becomes the better choice.

In summary, $Threshold_{Layer(i)} = 2$ is a watershed. When $Threshold_{Layer(i)} \geq 2$, the layer is transfer-sensitive; otherwise, it is compute-sensitive. Based on this, we present a layer-adaptive data placement strategy described as follows:

Fig. 4. Breakdown of performance after applying threshold analysis.

- (a) For the compute-sensitive layer, in the forward pass, the input feature map of the current layer is asynchronously transferred to CPU memory. In the backward pass, the strategy fetches the feature map that is temporarily stored in CPU memory and will be used later. The CPU-GPU data transfer overlaps with the computation.
- (b) For the transfer-sensitive layer, the strategy drops the input feature map of the current layer in the forward pass. However, we keep the memory block unreleased and reuse it for the next feature map. In the corresponding backward pass, the strategy recovers the dropped feature maps using extra forward computation.

Figure 4 shows the layer-by-layer performance of CNN model after applying our proposed method on the cases of Figure 1. For any layer in Figure 4, a black column indicates that the corresponding layer selects the corresponding optimization method, and a gray column indicates that the corresponding layer does not select it. Taking AlexNet as an example, the time overhead of the CPU-GPU transfer method is about 210ms, and the time overhead of the extra-forward method is about 250ms. After applying the threshold selection optimization, the time overhead of AlexNet decreases to 126ms. Namely, about 40%~50% time overhead can be reduced, which is very considerable.

In practice, to capture the transfer cost and forward cost of each layer, we also propose a profiling-based method to perform pure forward propagation on the target CNN model before the actual training (see Algorithm 1 for details). Specifically, we use a CUDA event API [31, 34] to measure the forward time and transfer time for each layer.

Algorithm 2 describes how to use this strategy for iterative training. Separate CUDA streams are used to implement asynchronous data transfer and extra forward in pipeline, respectively,

ALGORITHM 1: Profiling costs of transfer and forward**Function** Profile()

```

for  $i = 0 \rightarrow L$  do
    forward_timer  $\rightarrow$  start() // start timer
    layer[i]  $\rightarrow$  Forward()
    forward_time[i] = forward_timer  $\rightarrow$  stop() // record forward time of each layer
    transfer_timer  $\rightarrow$  start() // start timer
    layer[i]  $\rightarrow$  TransferToHost()
    transfer_time[i] = transfer_timer  $\rightarrow$  stop() // record transfer time of each layer
    if transfer_time[i]/forward_time[i] > threshold then
        | layer[i]  $\rightarrow$  SetType(TRANSFER_SENSITIVE)
    else
        | layer[i]  $\rightarrow$  SetType(COMPUTE_SENSITIVE)
    end
end

```

ALGORITHM 2: Training with a layer-adaptive data placement strategy**Function** Train(transfer_stream, compute_stream)

```

for  $i = 0 \rightarrow L$  do
    layer[i]  $\rightarrow$  Forward()
    if layer[i]  $\rightarrow$  Type() == COMPUTE_SENSITIVE then
        | // transfer feature maps from GPU to CPU memory
        | layer[i]  $\rightarrow$  TransferToHost(transfer_stream)
    else
        | // drop the input feature map
        | layer[i]  $\rightarrow$  FreeInput()
    end
    // synchronize different streams
    Synchronize();
end

for  $i = L \rightarrow 0$  do
    layer[i]  $\rightarrow$  Backward()
    if layer[i]  $\rightarrow$  Type() == COMPUTE_SENSITIVE then
        | // fetch the feature map that will be used soon
        | layer[i - 1]  $\rightarrow$  TransferToDevice(transfer_stream)
    else
        | // recover the feature map dropped in forward pass
        | layer[i - 1]  $\rightarrow$  Forward(compute_stream)
    end
    // synchronize different streams
    Synchronize();
end

```

to ensure that the training performance is not seriously affected. One CUDA stream is used for memory copy operations between the host and the device, and the other is used to manage extra forward tasks on the device.

Although a profile-based method is helpful in determining the time cost of memory optimization methods, it is impractical in some distributed multiple GPUs-based environments, because of the largeness of the number of GPU nodes with different resources, the variety of the CNN workloads, and the limitation of the interconnect bandwidths. In fact, in a distributed cluster environment,

Fig. 5. Examples of gradient map optimization: (a) a linear network with a width of one needs only two memory blocks for gradient maps; (b) a nonlinear network with one branch, for which the width can be regarded as two, needs only three memory blocks for gradient maps. The dotted rectangles represent sliding windows.

the profile-based method cannot perceive factors outside the node itself, such as different GPU resources and bandwidth between nodes. A model based on Equation (5) would greatly improve the efficiency of selection of the memory optimization methods. In addition, the analysis model is also helpful in analyzing the influence of different factors on threshold selection.

4.2 Multi-type Intermediate Data Reuse Strategy

Gradient map: There are already several existing methods that try to optimize the memory usage for this type of intermediate data, and the intra-layer strategy of Layrub [25] is typical of these. The gradient map reuses the memory space of the corresponding feature map. Although it can achieve up to 50% memory savings, this intra-layer memory reuse method introduces some data dependency hazards, i.e., *write-after-read* (WAR) hazards, between the gradient of parameter and the gradient of activation data.

The synchronization control introduced to resolve this hazard reduces the degree of the computational parallelism, which impairs the training performance; it also cannot be used directly on the activation layer, which is unacceptable for CNNs, since they usually have a large number of activation layers.

To optimize the memory usage of the gradient map, we present a new memory optimization method based on a sliding window, which involves no WAR hazard and can be applied to any type of layer. A backward propagation usually consists of a series of layer-wise computations, and only a single-layer backward computation can be performed at any one time. Based on this situation, a sliding window is typically used for backward computation. It contains a single layer at any given moment and slides from the last layer to the first during the backward computation. This motivates us to reuse the GPU memory for gradient maps layer-by-layer.

Figure 5(a) shows a five-layer linear network, assuming that the backward computation proceeds for the current layer l_3 and the gradient maps required for this calculation are G_3 and G_2 . When the calculation for l_3 is complete, the window slides to l_2 , resulting in G_3 leaving the window. This

means that we can reuse the memory for G_1 . For all of the layers in the network, we only need two memory blocks to hold these gradient maps, and we can therefore reduce the memory cost of the gradient maps to $O(1)$.

More generally, as shown in Figure 5(b), for a non-linear network having one branch, when the window slides to l_4 , the backward computation of l_4 needs G_2 , G_3 , and G_4 taking up three memory blocks. When the backward computation of l_4 is finished, the window slides to l_3 ; the gradient map G_4 is no longer needed, and G_{13} (which has newly entered the window) can reuse the memory block for G_4 .

From the above examples, it is obvious that $w + 1$ memory blocks are required for the gradient maps (w is the width of the network). That is to say, the memory cost of the optimized gradient maps is $O(w)$, which is linearly correlated to the width of the network. Fortunately, for both simple networks such as LeNet [28] and complex networks such as Inception-V4 [39], although the depths of the networks vary greatly, the change in the widths of the networks is far less than that of the depths. Therefore, our proposed sliding window-based approach can save a great deal of memory in the gradient maps of diversified networks.

Workspace and cuDNN handle: Although feature maps and gradient maps consume the majority of GPU memory for all networks, there are other types of intermediate data in the training process, such as workspaces and cuDNN handles. Several optimization methods have been proposed for feature maps and gradient maps, and these can greatly reduce the memory required. As a result, workspaces and cuDNN handles have become increasingly important sources of intermediate data that consume GPU memory; however, few researchers have noted this.

To illustrate the importance of the workspaces and cuDNN handles, we show the proportions of the memory footprints of these data in Figure 6. For comparison, we show the memory usage of a naive memory allocation method that allocates all of the memory space required by the training process, and the memory usage of a state-of-the-art optimization method with the CPU-GPU transfer that achieves significant saving rates for feature maps and gradient maps. After applying optimizations to feature maps and gradient maps, which causes the memory footprints of these to drop dramatically, the proportion of the memory consumed by the remaining intermediate data (such as workspaces and cuDNN handles) obviously increases and becomes hard to ignore, as shown in Figure 6(b).

In the original Caffe, each request for a workspace or cuDNN handle requires a new memory allocation, resulting in a linear relationship between the memory cost of the workspace/cuDNN handle and the number of network layers, which is a wasteful approach given the limited amount of GPU memory.

Fortunately, we observe that the processes of most CNNs are layer-wise; that is, they are processed layer-by-layer, and different workspaces of different layers are logically independent, as are the different cuDNN handles. The optimization strategies for the workspace and cuDNN handle are therefore relatively straightforward.

After the calculation of the i th layer is completed, the memory block corresponding to the workspace becomes idle and can then be directly reused by the workspace of the next layer. The optimization strategy for the cuDNN handle is carried out in the same way, as shown in Figure 7. Based on these optimizations, a constant memory space can be reused for the workspace and cuDNN handles.

Thus far, we have described our proposed optimizations for multi-type intermediate data in neural network training. In the optimization strategy for the feature map, a slight performance overhead is incurred due to the synchronization of pipelines. The optimization strategies for the other intermediate data do not incur any performance penalty.

Fig. 6. Breakdown of GPU memory usage for several CNN models. The right-hand axis shows the fraction of the workspace and the cuDNN handle. The number close to the name of each network is the batch size. After optimization of feature maps and gradient maps, the proportion of memory required for the workspace and handle increases dramatically (from 7% to 49% in ResNet-152).

Fig. 7. Reuse of workspace and cuDNN handle, where ws: workspace of convolution; h: cuDNN handle. The same color indicates that different data use the same memory block.

5 EVALUATION

We do the evaluation in the following three aspects: First, the execution time potential of Layup is evaluated, comparing with some existing state-of-the-art works, including vDNN [32], Layrub [25], Caffe [24], and SuperNeurons [42]. Since SuperNeurons is the most effective approach among them and was put forward recently, we carry out a separate comparison with this technique in more detail. Second, in addition to execution time, the memory efficiency is another important issue that is focused on here; in fact, to some extent, memory efficiency is more important than execution

Fig. 8. Normalized execution time for Layup and two single methods (the lower, the better; the implementations of Transfer and Extra-Forward are based on the ideas in Reference [32] and Reference [4], respectively).

time. We carry out some concrete comparisons with state-of-the-art systems to demonstrate the memory efficiency of our presented approaches. Finally, we exploit the extremity of network scales that can be held in a single GPU physical memory. The results strongly confirm the advantages of our proposed memory optimization method.

We implement our optimizations in the Caffe framework, inserting a layer-adaptive method selection approach and transformation code, and implement a memory reuse strategy for multi-type intermediate data.

5.1 Execution Time Comparison

Comparison with static optimizations: Our layer-adaptive dynamic selection method chooses an optimal memory optimization method for each layer to minimize the execution time of the overall network model. To verify its effectiveness, we select a series of representative CNNs and compare our proposed method, Layup, with two existing static optimizations, CPU-GPU transfer (Transfer in Figure 8) and extra forward computation (Extra-Forward in Figure 8), without layer-adaptive dynamic selection. The results are shown in Figure 8. For a straightforward comparison between Layup and the two single optimizations, we implement the CPU-GPU transfer and extra-forward strategies in Caffe, based on the ideas used in Reference [32] and Reference [4], respectively. The execution time of the original Caffe is chosen as a baseline for comparison. The extra overheads of different methods are represented by different textures in Figure 8. For each method, the total overhead is the sum of the extra overhead and the baseline overhead of the original Caffe. The results show that Layup has the smallest extra overhead for each network model, compared with the two static optimizations. In detail, the lowest extra overheads of the transfer and the extra-forward methods are 12.5% and 17.9% of the baseline training time, respectively. The average extra overheads over all seven network models are 19.5% and 21%, respectively. In contrast, we can see that the lowest extra overhead of Layup is 5.7%, and the average extra overhead of Layup is 11.2% for all models. It is therefore clear that compared with the two existing static ideas, our proposed dynamic layer-adaptive selection method has the best time efficiency.

Comparison with Caffe-based memory optimizations: We also evaluate Layup approach by straightforwardly comparing with other state-of-the-art Caffe-based memory optimizations (e.g., vDNN [32] and Layrub [25]). As in the aforementioned experiments, several representative CNNs [20, 27, 36, 39, 40] are used for this evaluation. Since some methods can not execute the training processes for all models (as no open-source software was available, or there was no support for some models), we therefore only list those results that could be obtained. To carry out a straightforward comparison between Layup, vDNN (not open-source yet), and Layrub, we also

Fig. 9. Normalized execution time for Layup and other Caffe-based methods (the lower, the better; vDNN_all(p) is a memory-efficient policy that uses performance-optimal convolutional algorithms).

use a normalized execution time as before. The original Caffe is used as the baseline, for which the execution time is equal to one, as shown in Figure 9 (the lower the results, the better).

The results show that Layup achieves a remarkable improvement compared to the other optimizations. Using GoogLeNet (batch size = 128) as an example, the extra overhead in vDNN is about 38%, while that for Layup is only 14%. Overall, Layup, Layrub, and vDNN exhibit average extra overheads of 12%, 22%, and 26% compared to the baseline, and Layup outperforms Layrub and vDNN with average improvements of 10% and 14%, respectively, thus justifying our method of dynamic selection of memory optimization.

Comparison with SuperNeurons: As described above, SuperNeurons [42] is one of the best and most prominent works that has appeared over the past year, and we therefore carry out a separate comparison with this to approve the efficiency of Layup. SuperNeurons also uses dynamic memory management for training deep DNNs, but ignores the layer-based difference between CPU-GPU transfer and extra forward computation, which is the point addressed by the proposed scheme.

For a representative comparison between Layup and SuperNeurons, we use two types of GPUs, a Tesla K40m and a Tesla P100. Several typical network models such as AlexNet, VGGNet, and the ResNet family are used in this evaluation.

Figure 10 shows some performance results for Layup and SuperNeurons with varying batch size. Taking AlexNet in Figure 10 as an example, the training speed of Layup remains relatively stable as the batch size increases. However, the performance of SuperNeurons on P100 is significantly reduced with increasing batch size. When the batch size > 960, the training speed of SuperNeurons drops dramatically by about 19%. For SuperNeurons on K40m, the batch size that can be handled is much lower than that of Layup. In general, on both GPUs, the performance of Layup is better than that of SuperNeurons. The training speed of Layup is on average 10% higher than that of SuperNeurons on K40m, and on average 30% higher than SuperNeurons on P100.

For the ResNet family, a noteworthy phenomenon arises when the network models have a small number of layers. Our experimental results show that for ResNet-101 (Figure 10(c)) and some models with fewer layers, our proposed Layup is inferior to SuperNeurons in terms of the scalability of the batch size. Unlike the layer-centric optimization approach of Layup, SuperNeurons uses a fine-grained tensor as the basic memory optimization unit, resulting in Layup performing less well than SuperNeurons in terms of the scalability of shallow nonlinear networks. However, for ResNet-152 (Figure 10(d)), ResNet-200 (Figure 10(e)), and ResNet-269 (Figure 10(f)), Layup outperforms SuperNeurons in scalability. It is clear that the deeper the network, the more significant advantages of Layup. The major reason for this trend is that as the number of network layers

Fig. 10. Performance comparison between Layup and SuperNeurons.

increases, the memory costs of the workspaces and cuDNN handles become increasingly significant and cannot be ignored. Unlike most existing approaches, which leave these unoptimized, Layup effectively controls and refines the memory cost of both. In fact, no matter how deep the network is, Layup can reuse a constant memory space for the workspaces and the cuDNN handles.

Comparison with model parallelism: Since model parallelism is the most straightforward method for solving the memory issue, we compare the data parallelism used in Layup with the model parallelism used in TensorFlow [1]. Due to the complexity of model parallelism, most existing DL systems (such as Caffe) do not support model parallelism. Fortunately, TensorFlow does support this kind of parallelism, and we therefore choose TensorFlow to implement a case for comparison.

TensorFlow supports model parallelism when training CNNs by simply dividing different layers into different GPUs. For this experiment, we choose three deep models, ResNet-302, ResNet-404,

Fig. 11. Performance comparison between Layup (data parallelism) and TensorFlow (model parallelism).

Fig. 12. Comparison of runtime memory usage (Layup vs. two single optimizations; the implementations of Transfer and Extra-Forward are based on the ideas in Reference [32] and Reference [4], respectively).

and ResNet-500, whose memory usage is significantly larger than 12GB, and use two GPUs in a single machine to train these models. Generally, Layup trains the two models using data parallelism and TensorFlow using model parallelism. We find that ResNet-500 (batch size = 16) is the deepest model that TensorFlow can train successfully with two K40m GPUs (each with 12GB memory).

Figure 11 shows that the data parallelism of Layup outperforms the model parallelism of TensorFlow in terms of runtime performance (with average speedups of 1.4× and up to 1.8×). The degree of advantage of the data parallelism of Layup increases with the number of the layers of the model increasing, compared with the model parallelism of TensorFlow. Additionally, the slowdown of TensorFlow on ResNet-500 to some extent stems from the full load of GPU memory (it consumes 11,391 MB, nearly the entire memory of K40m). Compared to TensorFlow, Layup only consumes 7,069 MB. The excessive memory load of TensorFlow reduced its training speed.

5.2 Runtime Memory Comparison

Comparison with static optimizations: Since we aim to improve both performance and runtime memory efficiency, we also evaluate the efficiency of our proposed memory reuse strategy for CNNs. First, we compare the memory efficiency of Layup with the two aforementioned static optimizations, the CPU-GPU transfer (Transfer in Figure 12) and the extra forward computation (Extra-Forward in Figure 12), using several mainstream models. As shown in Figure 12, it is obvious that our approach achieves optimal memory savings across all models. It should be noted that the memory consumption of Layup on the AlexNet model is only 1% lower than that of the Transfer method, and the difference between them is almost negligible. However, on ResNet-152, the memory consumption of Layup is about 50% lower than that of the Transfer method. The reason for this is that the proportions of workspaces and cuDNN handles are different in the two

Fig. 13. GPU memory usage (Batch size = 32).

Fig. 14. Comparison of runtime memory usage on K40m (Layup vs. different systems).

models. For ResNet-152, workspaces and handles consume a considerable proportion of memory, as described in Section 4.2. The optimization of workspaces and handles by Layup means that the model training has less memory consumption.

Comparison using deep CNN models: Since ResNet is a representative and popular deep CNN model, we use it as an example to demonstrate the advantages of our presented Layup on deep CNN models. The depth of ResNet is set to 101/152/200/269/1,001, successively. Since most of these models require more than 12GB of memory, we use the theoretical value as the baseline. The theoretical value is calculated as follows: The amount of memory required by the model is calculated layer-by-layer and accumulated to obtain the overall memory required. As shown in Figure 13, compared with the baseline, Layup can significantly reduce the memory cost of deep network training. For ResNet-1001, the GPU memory required of the baseline is as high as 73GB, while Layup only consumes about 6GB GPU memory during training, saving 92% of GPU memory. Note that for reading convenience, a logarithmic base-2 coordinate is taken as the vertical axis.

Comparison with well-known systems: Layup is also compared with some well-known systems on GPU K40m and P100, including TensorFlow, MXNet with memonger (MXNet as abbreviation) [4], Caffe, and Layrub. Some systems cannot train all models, such as DenseNet-121 with Layrub, and we present only those results that could be obtained.

Comparing Figure 14 with Figure 15, it is clear that the memory usage for the same model on different GPUs is almost identical for the same system.

As shown in Figures 14 and 15, Caffe has almost the worst memory consumption during the training phase, since it has few memory optimizations other than the inplace optimization (in-place

Fig. 15. Comparison of runtime memory usage on P100 (Layup vs. different systems).

[4] is an optimization for the activation layer that stores the output value of the activation layer directly into the memory of input value). The results from TensorFlow are moderately good. Moreover, ResNet-1001 with batch size of 32 on a 12GB K40m GPU can only be trained by MXNet, Layrub, and Layup successfully. Compared with MXNet and Layrub, Layup consumes the least memory.

In general, our proposed Layup scheme consumes less GPU memory than most other approaches, achieving an average reduction in runtime memory usage of 40.98% over TensorFlow, 43.71% over MXNet, and 66.43% over Caffe.

For Layrub, the comparison of results becomes rather more complicated. For AlexNet and VGG, Layup falls slightly behind Layrub. The reason for this is that Layrub involves an intra-layer level memory reuse between the feature map and the corresponding gradient, which is effective for some models in which the width is greater (for example, AlexNet and VGG have a number of convolutional filters that make the width larger). However, our main target is to optimize extra-deep networks such as DenseNet-121 [21] and ResNet-1001 in Figure 14, which shows that our strategy outperforms all other baseline systems, including Layrub; for example, on ResNet-1001, Layup outperforms Layrub by achieving runtime memory savings of 32.35%.

SuperNeurons usually tries to exchange memory for computation when the memory of the GPU is abundant; in other words, it always attempts to use up the memory of the GPU to obtain better computation performance. It is therefore disadvantaged in a comparison of memory consumption with other approaches when the scales of the networks are relatively small; it makes little sense to draw a comparison under these conditions, and such comparison is not given in detail here.

However, the primary advantage of SuperNeurons is that it can support extreme-scale networks, which is also a point of advantage of Layup. This issue will be discussed in the coming section.

5.3 Exploitation of Extreme-scale Networks

Extreme depth: As discussed above, one of our major objectives is to enable machine learning practitioners to exploit deeper network architectures using GPUs with limited physical memory. Here, we take the ResNet family as an example. Experimental data illustrate that our proposed Layup scheme can extend ResNet up to 2,504 layers (batch size = 16) on a GPU with 12GB memory, thus outperforming SuperNeurons [42] with 1,920 layers by 30%. In a more extreme case, Layup can train ResNet-2801 with a batch size of one, as shown in Table 3 (of course, a batch size of 16 is more practical in many applications). In general, the experimental data show that our proposed method allows a single GPU to process larger CNN models.

Extreme batch size: Due to the reduction in the GPU memory consumption required by the training model by applying memory optimization, we can not only train deeper networks, but also

Table 3. Maximum Layers of ResNet Supported by GPU K40m

Network (Batch Size)	MXNet	TensorFlow	Layrub	SuperNeurons	Layup
ResNet(16)	1,301	592	1,601	1,920	2,504
ResNet(1)	1,517	1,517	1,700	2,500	2,801

Table 4. Largest Batch Size that Several CNNs Can Reach on GPU P100

Network	MXNet	TensorFlow	SuperNeurons	Layup
AlexNet	1,536	1,600	1,536	1,920
VGG-16	192	128	128	224
ResNet-152	272	64	384	384
ResNet-200	256	48	256	384
ResNet-269	224	32	192	352

increase the batch size of the input data. Table 4 shows the largest batch size that several CNNs can process in different systems on a P100 GPU. In all these models, Layup scales better than the other systems, especially for depth models. Layup can train ResNet-269 with a batch size of 352, which is $1.57\times$ larger than that of the second-place MXNet. Note that SuperNeurons is also able to train ResNet-152 at batch size of 384; however, when training deeper models such as ResNet-200/269, it is significantly inferior to Layup. For ResNet-269, the maximum batch size that Layup can support is $1.837\times$ larger than that of SuperNeurons.

6 CONCLUSION

In this article, we present Layup, a layer-adaptive and multi-type intermediates-oriented memory optimization strategy, to reduce the memory consumption for CNN training processes on GPUs. By carefully analyzing several factors that affect the performance of different types of layers and dynamically selecting different memory optimizations for these layers, we develop a layer-adaptive memory optimization method that realizes better performance, compared with state-of-the-art works, which typically use a single memory optimization for all layers. Moreover, a comprehensive analysis of the sources of memory consumption allows us to reduce memory usage by tapping into the potential of multi-type intermediate data. Experimental results show that Layup can even outperform SuperNeurons with a 30% improvement for ResNet (batch size = 16). All in all, our proposed strategy can save more GPU memory for CNNs while providing better execution time, compared with state-of-the-art memory optimization approaches, especially for deeper networks. Moreover, as a system-level strategy, Layup can be implemented on a variety of mainstream deep learning frameworks. In the future, we will take the performance diversity of different nodes and the complexity of communication in various distributed environments into account and pursue better balance between memory and time efficiencies in distributed deep learning systems.

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, 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)*. 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 the Workshop on Machine Learning Systems at the 28th Conference on Neural Information Processing Systems (LearningSys'15)*. Curran Associates, Inc., 1–6.
- [4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. Retrieved from: *arXiv preprint 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. Retrieved from: *arXiv preprint 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, 571–582.
- [7] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, and Bryan Catanzaro. 2013. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on International Conference on Machine Learning (ICML'13)*, Volume 28. IMLS, 1337–1345.
- [8] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. 2002. *Torch: A Modular Machine Learning Software Library*. Technical Report EPFL-REPORT-82802. Idiap, Martigny, Valais, Switzerland.
- [9] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training deep neural networks with binary weights during propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS'15)*. Curran Associates Inc., 3123–3131.
- [10] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. Retrieved from: *arXiv preprint arXiv:1602.02830* (2016).
- [11] 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, 1–16.
- [12] 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 Conference on Neural Information Processing Systems (NIPS'12)*. Curran Associates, Inc., 1223–1231.
- [13] 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, 248–255.
- [14] Facebook. 2017. Caffe2: A new lightweight, modular, and scalable deep learning framework. Retrieved from: <https://caffe2.ai/>.
- [15] Jeremy Fowers, Greg Brown, John Wernsing, and Greg Stitt. 2013. A performance and energy comparison of convolution on GPUs, FPGAs, and multicore processors. *ACM Trans. Archit. Code Optim.* 9, 4 (2013), 25:1–25:21.
- [16] Qichuan Geng, Zhong Zhou, and Xiaochun Cao. 2018. Survey of recent progress in semantic image segmentation with CNNs. *SCI. CHINA Inform. Sci.* 61, 5 (2018), 1–18.
- [17] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-efficient backpropagation through time. In *Proceedings of the 29th Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., 4125–4133.
- [18] 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)*. IEEE, 1–14.
- [19] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS'15)*. Curran Associates Inc., 1135–1143.
- [20] 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, 770–778.
- [21] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the 30th IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*. IEEE, 2261–2269.
- [22] 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, 646–661.
- [23] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*. IMLS, 448–456.
- [24] 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, 675–678.

- [25] 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 (2018), 37:1–37:26.
- [26] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto.
- [27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th Conference on Neural Information Processing Systems (NIPS'12)*. Curran Associates, Inc., 1097–1105.
- [28] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
- [29] 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 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'14)*. IEEE, 5337–5341.
- [30] 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. Retrieved from: *arXiv preprint arXiv:1701.03980* (2017).
- [31] NVIDIA. 2017. CUDA toolkit 8.0 documentation: Profiler. Retrieved from: <http://docs.nvidia.com/cuda/profiler-users-guide/index.html>.
- [32] 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 IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–13.
- [33] Probir Roy, Shuaiwen Leon Song, Sriram Krishnamoorthy, Abhinav Vishnu, Dipanjan Sengupta, and Xu Liu. 2018. NUMA-Caffe: NUMA-aware deep learning neural networks. *ACM Trans. Archit. Code Optim.* 15, 2 (2018), 24:1–24:26.
- [34] Jason Sanders and Edward Kandrot. 2010. *CUDA by Example: An Introduction to General-purpose GPU Programming*. Addison-Wesley Professional.
- [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, 2135–2135.
- [36] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. Retrieved from: *arXiv preprint arXiv:1409.1556* (2015).
- [37] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. 2012. UCF101: A dataset of 101 human actions classes from videos in the wild. Retrieved from: *arXiv preprint arXiv:1212.0402* (2012).
- [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, 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, 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, 2818–2826.
- [42] 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, 41–53.
- [43] 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, 5987–5995.
- [44] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. 2016. Achieving human parity in conversational speech recognition. Retrieved from: *arXiv preprint arXiv:1610.05256* (2016).
- [45] Wenlai Zhao, Haohuan Fu, Jiarui Fang, Weijie Zheng, Lin Gan, and Guangwen Yang. 2018. Optimizing convolutional neural networks on the Sunway TaihuLight supercomputer. *ACM Trans. Archit. Code Optim.* 15, 1 (2018), 13:1–13:26.

Received April 2019; revised July 2019; accepted August 2019