Efficient Multi-GPU Memory Management for Deep Learning Acceleration

Youngrang Kim, Jaehwan Lee
Korea Aerospace University
Goyang-city, Republic of Korea
kimyr207@gmail.com, jlee@kau.ac.kr

Jik-Soo Kim
Myongji University
Yongin-city, Republic of Korea
jiksoo@mju.ac.kr

Hyunseung Jei, Hongchan Roh
SK Telecom ML infra Lab.
Seongnam-city, Republic of Korea
{hsjei, hongchan.roh}@sk.com

***Abstract*—In this paper, we propose a new optimized memory management scheme that can improve the overall GPU memory utilization in multi-GPU systems for deep learning application acceleration. We extend the Nvidia's vDNN concept (a hybrid utilization of GPU and CPU memories) in a multi-GPU environment by effectively addressing PCIe-bus contention problems. In addition, we designed and implemented an intelligent prefetching algorithm (from CPU memory to GPU) that can achieve the highest processing throughput while sustaining a large min-batch size. For evaluation, we have implemented our memory usage optimization scheme on Tensorflow, the well-known machine learning library from Google, and performed extensive experiments in a multi-GPU testbed. Our evaluation results show that the proposed scheme can increase the mini-batch size by up to 60%, and improve the training throughput by up to 46.6% in a multi-GPU system.**

***Keywords*-Convolutional neural network, GPGPU memory, Multi-GPU**

I. Introduction

Convolutional neural network (CNN) uses the Convolution layer to extract features of input data and perform training using those features [1]. General Purpose GPUs (GPGPUs) can be used to speed up the parallel operations in CNN, however due to the physical limitation in the amount of available GPU memory, it is not always possible to compute large-batch input data or large CNN models. In the case of a typical CNN, the feature map data, which are the outputs of convolution layers, occupy the largest portion in GPGPU memory. Feature map data are generated in the process of feed-forwarding, but it is not used for the actual operation until it is reused during the backward-propagation process. Therefore, the feature map data can stay in GPU memory for a long time without actual usage until the backward-propagation process starts.

vDNN [2] is a runtime memory management system provided by NVIDIA that virtualizes the GPU and CPU memory usage. . In order to overcome the physical limitation of GPGPU memory, vDNN swaps out feature map data, which normally remain in GPU memory for reuse but are not immediately required for processing, to CPU memory. In the back-propagation stage, the feature map data (that was swapped out to CPU memory) are swapped in back into GPU memory when the data is needed. However, during this swap-in/out process using the memcpy operation, time delay can occur due to inappropriate swap timing decision. Moreover, such time delays caused by vDNN can get worse in "multi-GPU" environments since vDNN accounts for only a single GPU.

In this paper, we propose a new memory management scheme to efficiently utilize GPU memory by swapping-in/out feature map data between multi-GPUs and CPUs. Specifically, our contributions in this paper can be as followings:

- First, we propose a new solution that resolves the PCIe bus bottleneck problem that can occur when multiple GPUs try to transfer data from/to CPU simultaneously. Typically, a GPU transfers data to a CPU via PCIe bus, and when multiple GPUs are connected to the same PCIe Switch, the bandwidth of the PCIe bus should be shared by multiple GPUs. If the vDNN's original approach is applied on multi-GPU system, the whole computation may be delayed due to PCIe bus bottleneck as multiple GPUs transfer data to and from the CPU at same time. In order to address this problem, we restrict the number of GPUs that can concurrently execute swap-in/out for each layer in a learning graph constructed with multi-layers.
- Second, we design and implement an effective algorithm to find the optimal swap-in/out timing in an adaptive way. To avoid time-delay due to swap-in/out, in the backward-propagation process, vDNN transfers a specified layer's data from the CPU memory to GPU during the former layer's computation process. However, when the size of data to be transmitted is large or the computation time is too short, time-delay can occur. Therefore, we propose an optimal timing trigger for swap-in that can prevent both unnecessary time-delays and out of memory errors.
- Third, we verify the effectiveness of our solution in multi-GPU nodes by performing real extensive experiments. We have implemented our optimization scheme in Google machine learning library, called Tensorflow. We describe how to implement swap-in/out operations and triggers to swap in Tensorflow, and present Tensorflow's internal architecture in detail related to optimizing memory usage and kernel executions.

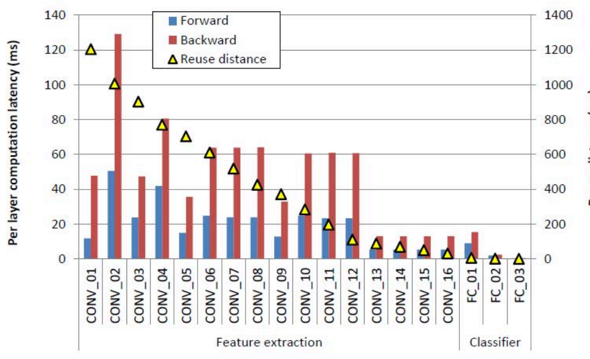


Figure 1. VGG-16s per layer computation latency for forward and backward propagation (left axis). Right axis shows the reuse distance of each layers input feature maps. (This figure is exactly the same as Figure 6 in the vDNN paper [2].

To evaluate our new solution, we experimented the actual CNN with one, two, and four GPUs while changing mini-batch sizes. From the results, we confirmed that mini-batch size increases and image processing throughput per second is improved compared to non-swap case and the original vDNN. In case of a single-GPU case, mini-batch size increases by up to 60% for VGG-16[3] training. In addition, we show that our effective trigger decision in a multi-GPU system can reduce PCIe bus bottlenecks, resulting in image processing throughput improvement by 46.6% compared to the existing swap execution method.

The rest of this paper is structured as follows. In Section 2, we explain the characteristics of CNN and the practical performance limitation by GPU memory size, and describe the swap-in/out method to overcome this issue. In Section 3, we describe our new solution to efficiently expand GPU memory for a multi-GPU system and implementation details of our proposed design on Tensorflow. In addition, we present an effective swap-in trigger decision algorithm for avoiding time-delay. In Section 4, we present comparative experimental results and verify the effectiveness of our proposed solution. Related work is presented in Section 5, and in Section 6 we conclude this paper.

II. Background

A. *Convolutional Neural Network(CNN) on GPUs*

CNN specializes in extracting features among Deep Learning algorithms, and shows high accuracy on vision applications [1]. Unlike the learning model using only Perceptron, a CNN consists of multiple convolution layers and fully-connected layers. Convolution layers extract the features of input data, and fully-connected layers classify the features. There are VGG-16 [3], Resnet [4], and GoogleNet [5] in typical CNNs. Since executing these CNN models on CPUs take a long time, usually researchers use GPGPU to reduce execution time by parallelized computing with

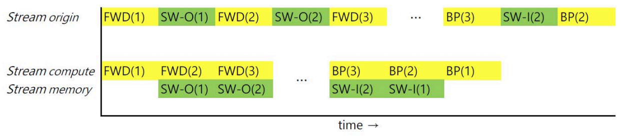


Figure 2. Performance effect of Swap-in/out. FWD(n) and BP(n) are the forward and backward computations for layer(n). Swap-out(n) is the transfer of feature map data in layer(n) to CPU memory and Swap-in(n) is the transfer of data to GPU memory for gradient descent operation in layer(n).

thousands of cores. In the convolution layer, the size of each feature map data, which is the output value of the convolution layer, is much larger than parameters used in each layer. Mini-batch size, the number of input data being simultaneously executed in a training batch in a single GPU, does not affect weight parameters. However, feature map data will be increased by the mini-batch size. Thus, GPUs will require a large amount of memory space when training a large mini-batch.

All machine learning algorithms using Deep Learning, including CNN, can be broadly divided into two steps. First, it is a feed-forwarding process which calculates loss value by comparing it with a label after executing an input's inference. The second is the backward-propagation process of executing gradient descent using loss value calculated in feed-forwarding and updating parameters [6].

In the process of feed-forwarding, feature map data that are generated in all convolution layers are used for input of the next layer, and are reused again to execute a gradient descent algorithm in back-propagation. Figure 1 shows computation times of feed-forwarding and back-propagation, and reuse distances of feature map data between feed-forwarding and back-propagation. As seen from Figure 1, we can confirm that the feature map data that is created earlier has the longest reuse distance because the back-propagation process uses the data in reverse order of feed-forwarding. Another feature of CNN is that in case of multiple convolution layers, the size of feature map data decreases as it goes to the next layer. That means feature map data of the first convolution layer has the largest size, thus it occupies the largest amount of GPU memory for the longest time. As a result, GPU memory should contain feature map data for reuse, which makes it difficult to use memory efficiently for computation.

B. *vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design*

NVIDIA developed vDNN (Virtualized Deep Neural Networks) [2], which is a runtime memory management scheme that virtualized GPU and CPU memory. Because of physical GPGPU memory capacity limitations, the size of computable mini-batches and learning networks for deep learning are also limited. The purpose of vDNN is to reduce the maximum and average GPU memory usage, allowing developers

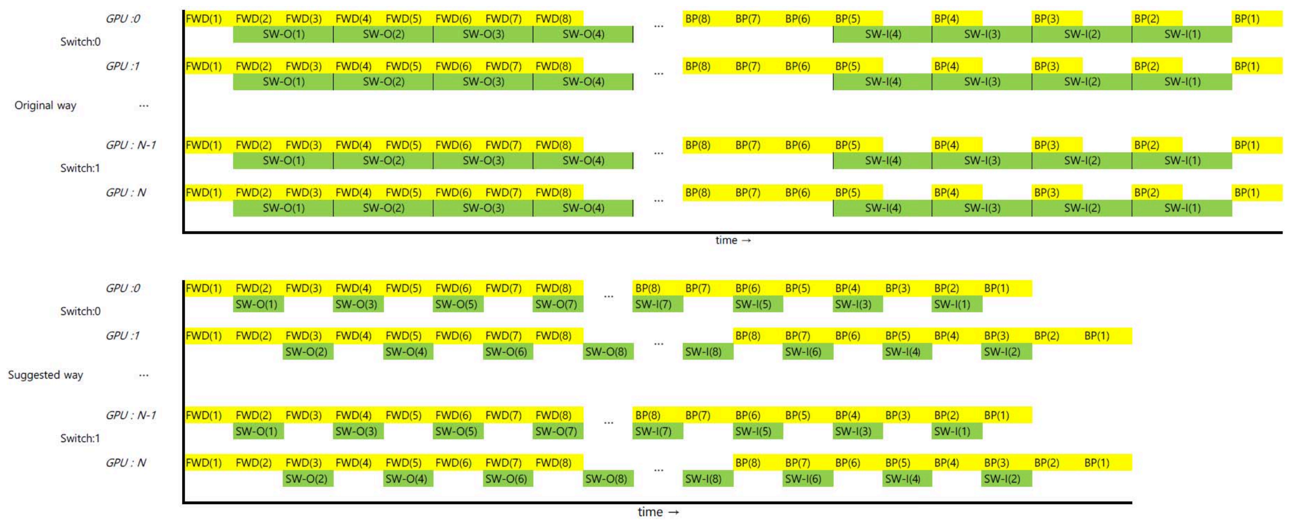

GPU :0
Switch:0
GPU :1
Original way
GPU : N-1
Switch:1
GPU : N
time →
FWD(1) FWD(2) FWD(3) FWD(4) FWD(5) FWD(6) FWD(7) FWD(8)
SW-O(1) SW-O(2) SW-O(3) SW-O(4)
BP(8) BP(7) BP(6) BP(5) BP(4) BP(3) BP(2) BP(1)
SW-I(4) SW-I(3) SW-I(2) SW-I(1)
Suggested way
SW-O(1) SW-O(3) SW-O(5) SW-O(7)
SW-O(2) SW-O(4) SW-O(6) SW-O(8)
SW-I(7) SW-I(5) SW-I(3) SW-I(1)
SW-I(8) SW-I(6) SW-I(4) SW-I(2)

Figure 3. Timeline of swap-in/out on a multi-GPU system using the original vDNN (above) and restricting the number of GPU to swap-in/out of each layer(below)

to train a large network, by conservatively allocating GPU memory for computation of the given layer. Therefore, vDNN swaps-out the data in GPUto CPU memory (for future reuse but not immediately needed). In the back-propagation process, vDNN swaps-in the data from CPU to GPU memory for reuse.

To swap the data between GPU and CPU memory, data can be transferred using the "memcpy" function on the PCIe bus. Executing memcpy after computing each layer can cause a time delay when sending and receiving data, so the total execution time can increase and the performance can be degraded. In order to solve this, the memory manager of the vDNN makes another separate stream for swap-in/out by using data reuse pattern of deep learning like Figure 2, so it can run both computation and swap-in/out streams in parallel. In the case of backward-propagation, vDNN's memory manager sets swap-in timing for a particular layer's data during the computation in the previous layer (i.e. prefetch as shown Figure 2). This prefetch can reduce time-delay caused by data transfer. By using vDNN, computation and data transfer can be overlapped so that it can virtualize GPU memory efficiently without losing performance, and it also enables training with a large mini-batch or learning model in a single GPU. However, when using the vDNN method in the multi-GPU system, data exchange between multiple GPUs and the CPU at the same time can cause a PCIe bus bottleneck, which results in significant performance degradation.

III. Architecture and Development

In this section, we describe how to extend swap-in/out scheme of vDNN in a multi-GPU system. To avoid PCIe bus bottleneck, our system design suggests that data transfer from/to multiple GPUs should not be overlapped. This design enables memory of each GPU to be expanded even when learning through multi-GPUs without performance degradation so that we can train with larger mini-batches and deep networks. We have modified Tensorflow source code to reflect our new memory management scheme so that it can be easily applied to existing applications.

A. *Multi-GPU Based Data Parallel Training*

The most popular way to use multi-GPUs on a single machine for accelerating deep learning is data parallel learning [7]. In data parallel training, input training data is divided into a mini-batch of dataset and fed to each GPU, and after back-propagation, the gradients of the learning result from multi-GPUs are aggregated on the CPU or one GPU. All GPUs are given different input values, and training latency in each GPU is almost same when using the same size mini-batch to train the learning model. In the case of vDNN, during the process of feed-forwarding, when computation of a specific layer is completed, result data of the computation is swapped out to CPU memory, and swapped back to GPU for reuse when the backward-propagation starts. But when using multiple GPUs with the same performance, the latency to finish a specific layer's computation is highly likely to be the same, so most GPUs try to swap-in/out the data simultaneously. At this point, if N GPUs are connected to the same PCIe Switch, transferring data to the CPU should share the bandwidth of PCIe, thus total transfer time is N times longer than the case of a single GPU. Therefore, the total execution time can be substantially delayed by these slow and concurrent data transfers.

To address this bottleneck problem, we propose a method to reduce the I/O contention from multiple GPUs by adjusting the number of GPUs that can perform swapping

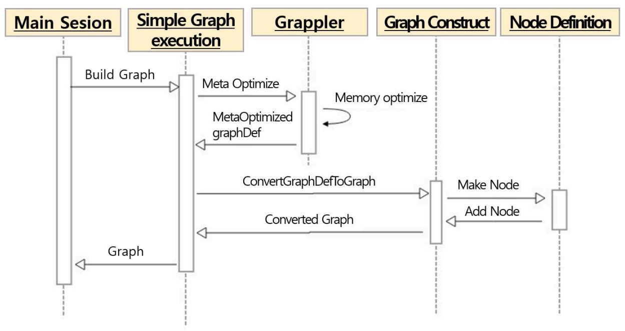


Figure 4. Process of converting graphDef written in Python in Tensorflow to graph after performing memory optimization at Grappler

operations on a specific layer. The algorithm to set these numbers is as follows.

- Step 1. Set the group by GPUs connected on a single PCIe switch.
- Step 2. Set the time interval x between the consequent two layers to execute swap as the total number of layers a in the deep learning model divided by the number of layers b to need to swap (i.e. $x = a/b$).
- Step 3. Within each PCIe switch group, the ith GPU can swap "$i\%x + j \cdot x$"th layer in the deep learning model ($0 \leq i <$ # of GPUs, $0 \leq j < b$).

By adjusting the number of GPUs that can perform swapping operations with this mechanism, the number of GPUs executing swap for a specific layer decreases as shown in Figure 3, which can relieve the PCIe bottleneck caused by memcpy operations.

B. *Development on Tensorflow*

To apply our scheme to actual training, we implemented it on Tensorflow, which is an open source deep learning package provided by Google. Tensorflow operates in a "Define and run" way in which a developer predefines the graph in the python level, and Tensorflow session analyzes the graph and passes an execution command to compute. The created graph consists of variables storing values and Nodes for executing arithmetic operations. Since it is the "Define and run" way, each Node is initially defined as NodeDef that consists of metadata such as name, attribute, shape, and input, but does not have actual values and computation operation. To execute actual computation on GPU, NodeDef has to be converted to Node and be implemented as kernel to perform computations on a target device (such as GPU).

To enable memory swap functions in Tensorflow efficiently, we developed a new memory optimizer that inserts swap functions to execute memcpy into proper Nodes in execution graph (shown in Figure 4. In Tensorflow, an optimizer in Grappler has a role to optimize the graph that a user creates before converting GraphDef into the execution graph. The detailed converting process is as follows. First, in the main Session, GraphDef is transmitted to the Simple

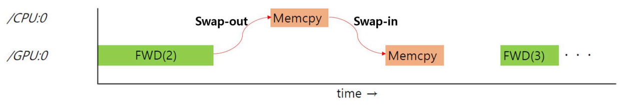


Figure 5. If trigger is not specified, swap-out and swap-in are performed consecutively

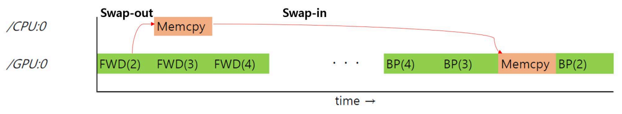
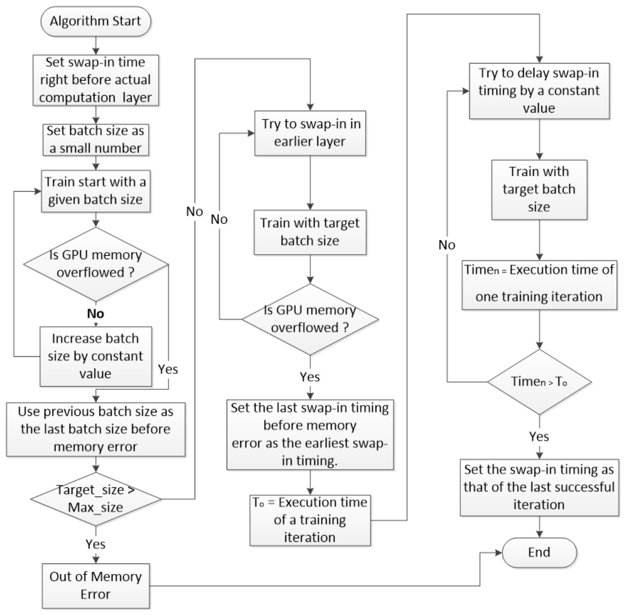


Figure 6. Adjusting swap-in executing time by setting trigger

Graph execution object to be converted into Graph for executing computation. Before converting, Simple Graph Execution transfers graphDef to Grappler to execute optimization with various optimizing options. Grappler is a layer for executing various optimizations inside Tensorflow. We added a memory optimizer performing swap-in/out functions in Grappler. The GraphDef optimized inside Grappler is returned to the Simple Graph execution object, and after that, converting NodeDef composing GraphDef is converted to Node one by one.

To implement swap-in/out function in Tensorflow, we use Identity Operation in the operation of Node. The Identity operation is for storing the result value of a computation and it can designate the location where this is stored by setting the location of the device. By setting the value computed by GPU to input of Identity operation and setting the device option of that Identity operation to CPU, this data can be transferred to the CPU memory, and not the GPU. The feature map data generated in feed-forwarding can be swapped out to the CPU memory from the GPU in this way, and it can be swapped back into GPU memory by setting the input of the operation which executes the gradient descent to swapped out data at feed-forwarding.

However, when we used an Identity operation for swap, we had a timing issue due to Tensorflow's aggressive parallelization execution. Tensorflow tries to optimize performance by parallel execution of operations that do not have data dependency, thereby increasing resource utilization. Since memcpy operation that executes swap-in does not have any data dependency until executing back-propagation of layer, Tensorflow results in performing memcpy (swap-in back) as fast as possible to improve computation efficiency. But in this case, feature map data will be swapped-out to CPU memory as illustrated in Figure5 and immediately swapped back in to the GPU, so GPU memory cannot be extended. In order to set the memcpy operation to perform swap-in just before executing the gradient descent operation, the memcpy operation should be set with a trigger as the operation before performing gradient descent as presented in Figure 6.



Algorithm Start
Set swap-in time right before actual computation layer
Set batch size as a small number
Train start with a given batch size
Is GPU memory overflowed ?
No
Increase batch size by constant value
Yes
Use previous batch size as the last batch size before memory error
Target_size > Max_size
Yes
Out of Memory Error
No
Try to swap-in in earlier layer
Train with target batch size
No
Is GPU memory overflowed ?
Yes
Set the last swap-in timing before memory error as the earliest swap-in timing.
T₀ = Execution time of a training iteration
Try to delay swap-in timing by a constant value
Train with target batch size
No
Timeₙ = Execution time of one training iteration
Timeₙ > T₀
Yes
Set the swap-in timing as that of the last successful iteration
End

Figure 7. Flow chart for our new trigger setting algorithm

C. *Efficient Trigger Settings for Swap-in*

In the original vDNN, starting time to swap-in for layer (N)'s feature map data is set as layer (N+1)'s gradient descent operation. However, if swap-in data of layer (N) takes a longer time than gradient descent operation of layer (N + 1), layer (N)'s gradient descent computation can be delayed. Moreover, when data to be swapped-in is bigger, time delay takes longer. To avoid this time delay, we design an advanced trigger setting algorithm by comparing actual computing latency and data transfer latency, instead of a fixed interval for a layer. This algorithm is performed while executing the operation of the early iterations of the whole training, and it can be explained by flow chart in Figure 7 and the following three steps.

- Step 1. In the given learning model and system, check the maximum mini-batch size that can be computable with an original vDNN method, and compare it with user's desirable mini-batch size. If the maximum mini-batch size is less than the user's desirable size, then stop the algorithm and report out-of-memory error.
- Step 2. Find out the earliest time range of each trigger (=prefetch) where learning process works well without out-of-memory error.
- Step 3. Searching for the latest time range for each trigger when swap-in is performed without any additional time-delay.

In the first step, swap-in time is set to be performed immediately before the gradient descent operation that requires that data, and we can increase size of mini-batch by a constant C for every iteration until out of memory occurs. When out of memory occurs, the size of mini-batch in the last successful iteration is set to the maximum batch size that can be computed in that model and system.

When swap-in is executed early, before the back-propagation computation stage, there is an advantage that GPU can fetch the data for computation immediately so time delay does not occur. However, out-of-memory may occur if swap-in to GPU memory is executed too earlier. Therefore, in the second step, we explore a way to find the earliest swap-in timing in an incremental way while not incurring a memory overflow.

In the last step, within the range in which an out-of-memory error does not occur, the algorithm tries to delay the trigger as late as possible where time-delay does not occur. For every iteration, it checks latency of total computation and compares it with the actual latency when the trigger is set in the second step, and searches for the position where time-delay starts to occur. When time-delay starts to occur with a specific trigger, the algorithm sets the trigger which is used in the last successful iteration as the optimized trigger.

Table I
MAXIMUM COMPUTABLE MINI-BATCH SIZE AND THROUGHPUT FOR VGG-16 AND INCEPTION-V3 MODEL ON A 12GB MEMORY GPU

$Model$	$Non-Swap$	$Swap$
VGG-16	$80_{batch}(62.95_{image/sec})$	$128_{batch}(73.93_{image/sec})$
Inception-v3	$114_{batch}(62.67_{image/sec})$	$150_{batch}(63.34_{image/sec})$

IV. EVALUATION

In this section, to evaluate our proposed scheme, we checked the size of a computable mini-batch compared with the original vDNN on a single GPU system to find out how much GPU memory has been extended. We also checked training throughput, which is defined as the number of images processed per second on each GPU, to confirm the improvement in computing performance. In addition, to verify the performance improvement on a multi-GPU system, we experimented by scaling up the number of GPUs and analyzed the results for each case.

A. *Single-GPU Experiment*

The main objective of our research is how to effectively extend GPU memory utilization to accommodate more training feature data in a given GPU. Thus, we check out the maximum mini-batch size in a single GPU without generating memory overflow error. In this experiment, we used NVIDIA GTX Titan Xp (12GB of memory), and our system trained Imagenet dataset using *VGG-16* and *Inception-v3*, which are typical CNN models. For both cases, not using swap functionality (denoted as "$Non-Swap$"), and using swap operation ("$Swap$"), we increased the size

Table II
IMAGE PROCESSING THROUGHPUT PER GPU WITH TWO GPUS

$Batchsize$	$Non - Swap$	$Swap_{original}$	$Swap_{optimized}$
120	$83.56_{image/sec}$	$56.33_{image/sec}$	$81.14_{image/sec}$
130	$Out\ of\ memory$	$56.66_{image/sec}$	$84.14_{image/sec}$
140	$Out\ of\ memory$	$58.86_{image/sec}$	$86.65_{image/sec}$
150	$Out\ of\ memory$	$60.12_{image/sec}$	$88.15_{image/sec}$

Table III
IMAGE PROCESSING THROUGHPUT PER GPU WITH FOUR GPUS

$Batchsize$	$Non - Swap$	$Swap_{original}$	$Swap_{optimized}$
110	$66.53_{image/sec}$	$51.06_{image/sec}$	$68.31_{image/sec}$
120	$Out\ of\ memory$	$52.49_{image/sec}$	$70.11_{image/sec}$
130	$Out\ of\ memory$	$53.09_{image/sec}$	$70.24_{image/sec}$
140	$Out\ of\ memory$	$54.86_{image/sec}$	$70.36_{image/sec}$

of the mini-batch to the maximum size that does not cause an out of memory error.

For each CNN model, about 60% and 30% of the computable mini-batch size in swap case is increased compared to the non-swap case. By increasing the mini-batch size, computation performance also improved as shown in Table I. In the case of VGG-16, there is about 17% improvement of performance, however the Inception-v3 model has only a little improvement, mainly due to different characteristics of each learning model. In the case of VGG-16, feature map data consume GPU memory much more than Inception-v3, so GPUs cannot utilize available cores for computation since they do not have enough input data. However, by effectively extending the available amount of memory, more input data can be computed in parallel so the computation performance has been improved. For the Inception-v3 model, the proportion of allocation for feature map data in GPU memory is not that large and the GPU core utilization is already high which results in similar performance with the Non-Swap mechanism.

Therefore, the effect of the swapping scheme can be different according to the learning model of CNN. If the ratio of feature map data is relatively high like the VGG-16 model, performance improvement can be obtained by exploiting our proposed design.

B. Multi-GPU Experiment

We suggested how to avoid the I/O contention in the PCIe bus by restricting the number of GPUs that can be swapped in/out for each layer in order to reduce time delay. To see the implications of our approach in terms of performance, we conducted experiments by using two and four GPUs (NVIDIA Tesla P100 with 16 GB memory) respectively, connected by PCIe on a single server. In addition, when gradient aggregation is executed in a single GPU in the case of data parallelization, the overall memory of GPUs cannot be utilized enough, so that gradient aggregation is done by the CPU. For these experiments, our system uses a single server equipped with four Tesla P100 (16 GB) connected to two PCIe switches, and each switch is connected to two GPUs. In order to see the effect of our proposed scheme, we used VGG-16 model to train Imagenet data.

Table II shows the results of the two GPU experiment with three different memory management schemes. The first case is training without swap-in/out method in order to see the base performance ("$Non - Swap$"). The second case is training with swap-in/out method of the original vDNN, which has performance degradation due to PCIe bottleneck ("$Swap_{original}$"). The last case is our proposed design with the optimized swap-in/out on multi-GPU system ("$Swap_{optimized}$"). In both cases of using swap-in/out mechanism, the size of the computable mini-batch increases up to 25% compared to the $Non - Swap$. When applying swap-in/out in the original way, the size of computable mini-batch increases but computation performance decreases due to the PCIe bottleneck. In the case of applying optimized swap-in/out based on our proposed design, we can confirm that the image throughput increases by about 46.6% per second by reducing the potential bottleneck.

The next experiment uses the total of four GPUs; the result is shown in Table III. As in Table II, although the size of the operable batch size increases, the overall computing performance has decreased for the same mini-batch size. This is because we used twice as many GPUs as the previous experiment, so the bandwidth of PCIe that each GPU can use simultaneously is halved. In the same way as the two GPU experiment, the performance of original swap-in/out method is lower than the case of not using swap-in/out. However, when applying our proposed time-delay reduction solution for a multi-GPU system, computing performance increases by about 30%.

To summarize, our optimized swap solution for multi-GPU systems can increase computable mini-batch size, and improve the throughput of image processing per second compared to the non-swap case and the original swap method, in a single GPU or even in multi-GPU environment.

V. RELATED WORKS

To improve the deep planning computation performance in a GPU, various methods were proposed. In the case of the Unified memory developed by NVIDIA [8], it operates in the CUDA level and uses a method of virtually extending GPU memory by using the host CPU memory like vDNN. Unified Memory can be applied to various applications, not only for Deep Learning, but it is not specialized for a specific operation, so it runs in an on-demand way at runtime. Therefore, although GPU memory can be extended significantly, there is a disadvantage that bottleneck occurring by data transfer can significantly increase the total execution time.

Another method for accelerating GPUs for deep learning is computation graph optimization. This method optimizes the execution graph by rewriting the graph created by the developer via high level optimization. It operates by removing unnecessary operations on a computation graph or fusing multiple operations. The graph rewritten in this manner is converted to the optimal execution code suitable (called kernel) for the hardware device that performs the actual operation, such as GPU and CPU, by a low level optimizer. Tensorflow's XLA [9] and MXnet's NNVM [10] are typical examples of low level optimizers. For example, XLA can reduce the learning time on mobile devices by 20%. In the case of NNVM, learning time can be reduced about 30% for LeNet model and about 20% for ResNet model. However, this low-level optimizer can only optimize the execution for each kernel independently, so it cannot resolve a node-wide resource bottleneck such as I/O contention in the PCIe bus.

VI. Conclusion

In this paper, we proposed a new GPU memory extension solution with resolving I/O bottlenecks that can occur in a multi-GPU system. To accommodate more input feature map data in GPU memory, we exploited swap-in/out method effectively. We have implemented our scheme on Google's deep learning library, Tensorflow, and confirmed that the actual usable memory and computational performance can be improved. Our proposed method can train deeper learning models with larger datasets, and we can expect high performance and efficient utilization of GPU resources for learning data with larger capacity. Moreover, high-speed interconnect hardware (e.g. NVlink) can reduce swap-in/out time-delay, and computing performance can be further improved. In the near future, we plan to apply this method in our heterogeneous cluster equipped with high-bandwidth GPU interconnect to see potential performance improvement.

Acknowledgment

This work was supported by the IT R&D program of MOTIE/KEIT [10076476, Scalable Machine Learning Acceleration Hardware Technology for Big-Data Servers].

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[2] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceedings of the International Conference on Learning Representations*, 2015.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *arxiv.org*, 2015.

[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998.

[7] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," in *arxiv.org*, 2014.

[8] NVIDIA, "Unified memory in cuda 6," 2013.

[9] XLA, "https://www.tensorflow.org/performance/xla," 2017.

[10] NNVM, "https://github.com/dmlc/nnvm-fusion," 2016.