applied sciences

MDPI

Article

Efficient Use of GPU Memory for Large-Scale Deep Learning Model Training

Hyeonseong Choi and Jaehwan Lee *

School of Electronics and Information Engineering, Korea Aerospace University, Goyang-si 10540, Korea; chyon794@gmail.com
* Correspondence: jlee@kau.ac.kr; Tel.: +82-2-300-0122

Abstract: To achieve high accuracy when performing deep learning, it is necessary to use a large-scale training model. However, due to the limitations of GPU memory, it is difficult to train large-scale training models within a single GPU. NVIDIA introduced a technology called CUDA Unified Memory with CUDA 6 to overcome the limitations of GPU memory by virtually combining GPU memory and CPU memory. In addition, in CUDA 8, memory advise options are introduced to efficiently utilize CUDA Unified Memory. In this work, we propose a newly optimized scheme based on CUDA Unified Memory to efficiently use GPU memory by applying different memory advise to each data type according to access patterns in deep learning training. We apply CUDA Unified Memory technology to PyTorch to see the performance of large-scale learning models through the expanded GPU memory. We conduct comprehensive experiments on how to efficiently utilize Unified Memory by applying memory advises when performing deep learning. As a result, when the data used for deep learning are divided into three types and a memory advise is applied to the data according to the access pattern, the deep learning execution time is reduced by 9.4% compared to the default Unified Memory.

Keywords: deep learning; large-scale model; CUDA Unified Memory; PyTorch

Citation: Choi, H.; Lee, J. Efficient Use of GPU Memory for Large-Scale Deep Learning Model Training. *Appl. Sci.* **2021**, *11*, 10377. https://doi.org/10.3390/app112110377

Academic Editor: Giancarlo Mauri

Received: 7 October 2021
Accepted: 31 October 2021
Published: 4 November 2021

1. Introduction

To achieve high accuracy when performing deep learning, the size of deep learning models is increasing [1–6]. However, there is a difficulty in using large-scale deep learning models that can achieve high accuracy due to the limitation of single-GPU memory [7–12]. NVIDIA introduced CUDA Unified Memory technology from CUDA 6, allowing the GPU memory to be expanded by integrating the system's host (CPU) memory and device (GPU) memory [13–15]. CUDA Unified Memory enables automatic data migration between host memory and device memory via virtual memory. For example, if the GPU tries to access a virtual page that does not exist in GPU memory, a page fault occurs. After that, the page fault is resolved by mapping the accessed page to the physical page of the GPU and migrating the data existing in the host memory to the GPU memory. Through this, device memory and host memory are logically integrated to achieve the same effect as the memory of the GPU is expanded. Therefore, it is possible to train large-scale deep learning models with expanded GPU memory using CUDA Unified Memory technology.

NVIDIA provides CUDA Unified Memory technology from CUDA 6.0 to expand GPU memory. CUDA Unified Memory technology integrates CPU memory and GPU memory so that it can be used as a single address space; therefore, CUDA Unified Memory enables the training of large-scale deep learning models by expanding the small GPU memory. In addition, starting with CUDA 8.0, memory advise can be used to efficiently manage data allocated to unified memory according to the access pattern [13,16,17]. NVIDIA divides memory advise into three types according to access patterns: *Read mostly*, *Preferred location*, and *Access by*. *Read mostly* makes efficient use of read-intensive data. When accessing data

set to *Read mostly*, a replication of the data is created in the device that accessed the data. *Preferred location* sets the device where the data will be located; therefore, the occurrence of page faults due to data access is minimized by locating the data in the device that will mainly use the data. *Access by* sets the device to access the data. Devices set to *Access by* are directly mapped to the data; therefore, when device accesses the data, a page fault does not occur.

The data required for deep learning can be roughly classified into three types: model parameters, input data, and intermediate results. Model parameters must be updated with new values every iteration and must be on the GPU until training is complete. Input data are changed to new data every iteration and are not changed due to deep learning operations. Intermediate results are data that are newly generated for each iteration through deep learning operations, and are no longer needed after one iteration. The access pattern of data used in deep learning is similar to the memory advise supported by NVIDIA. Therefore, when deep learning is performed by applying CUDA Unified Memory technology, it is possible to efficiently perform deep learning using memory advise.

In this paper, we extend GPU memory using CUDA Unified Memory technology for large-scale deep learning model training. To the best of our knowledge, our proposed scheme is the first approach to efficiently apply CUDA Unified Memory technology and hints for memory management provided by NVIDIA to deep learning. In addition, we propose an efficient GPU memory utilization method for large-scale training model training using memory advise. The main contributions of our work are as follows:

- First, we propose the most efficient memory advise for the three types of data required for deep learning. Deep learning data can be divided into three types according to access patterns: model parameters, input data, and intermediate results. Further, memory advise can manage data by dividing it into three types: *Read mostly, Preferred location,* and *Access by*. In addition, the access patterns of the three types of deep learning data are similar to those that can be managed with memory advise. Therefore, it is possible to efficiently utilize GPU memory by setting three types of deep learning data as appropriate memory advise. We conducted an experiment to train a large-scale model, BERT-Large [18], by applying our optimized memory advise scheme, and analyze the results of the experiment to suggest efficient memory advise.
- Second, we apply CUDA Unified Memory to PyTorch and set different memory advises for three types of data required for deep learning. It is difficult to train a large-scale model due to the limitation of single GPU memory. Existing PyTorch does not support CUDA Unified Memory technology, so it is impossible to perform deep learning if a memory size larger than the available GPU memory is required to perform deep learning. Most of the deep learning models that achieve high accuracy are very large, such as BERT. Therefore, we apply CUDA Unified Memory technology by modifying PyTorch, an open source deep learning framework, to train large-scale learning models that can achieve high accuracy. In addition, we use the modified PyTorch to verify efficient memory advise through an experiment in which we actually perform large-scale model training.

We trained BERT-Large, a large-scale deep learning model, using PyTorch to which CUDA Unified Memory technology is applied. When the mini-batch size is 32 and the max sequence size is 256, we confirmed that the training time is reduced by up to 9.4% by using appropriate memory advise. Using vanilla PyTorch, it is possible to train BERT-Large in mini-batches of size up to 16 with a GPU with 24 GB of memory. However, if our proposed scheme is applied to PyTorch, training can be performed with a mini-batch of size 32, which was impossible to train with vanilla PyTorch.

The rest of the paper is organized as follows. In Section 2, we describe the CUDA Unified Memory and how to use CUDA Unified Memory efficiently. The data management scheme we propose for efficient deep learning is described in detail in Section 3. In addition, Section 4 describes how we modified PyTorch to apply CUDA Unified Memory.

In Section 5, we analyze the performance of the proposed GPU memory utilization method. We present the related work in Section 6. Finally, we conclude our paper in Section 7.

2. Background

2.1. CUDA Unified Memory

From CUDA 6.0, NVIDIA provides CUDA Unified Memory, a technology to overcome the limit of single GPU memory size [13,14,16]. CUDA Unified Memory is a technology that allows physically separated host memory and device memory to be integrated into one logical address space using virtual memory scheme. Unified Memory works in the following way. First, when the GPU attempts to access data in the CPU memory, a page fault occurs. After that, the host unmaps the page mapped to the CPU memory and migrates the data to the GPU memory. Finally, the page fault is resolved by mapping the page to the GPU memory. By using CUDA Unified Memory, GPU can use data stored in physically different locations without explicitly calling memory copy functions such as *cudaMemcpy()* and *cudaMemcpyAsync()*. However, since a page fault occurs when accessing data in a physically different location, the ongoing process should be stopped to resolve the page fault and migrate data, so performance may be degraded.

2.2. How to Use CUDA Unified Memory Efficiently

For efficient use of CUDA Unified Memory technology, NVIDIA supports memory advise starting with CUDA 8.0. Memory advise can be divided into three types according to data access patterns [13,16]. Figure 1 shows how memory advise works.

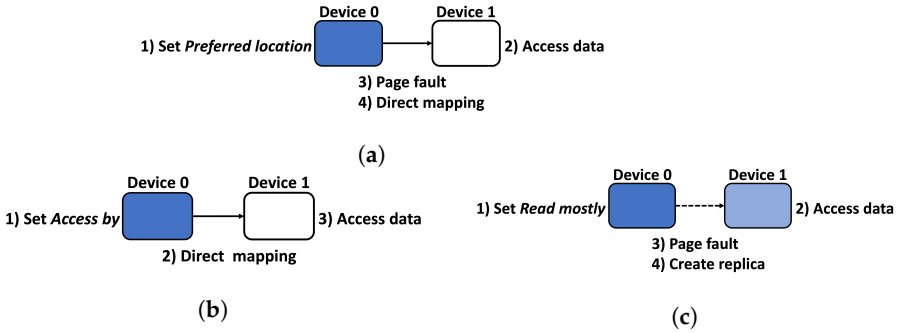


Figure 1. Operation method according to the type of memory advise: (**a**) Data set as *Preferred location* are directly mapped when accessed from other devices; (**b**) If data are set to *Access by*, a direct mapping is created with a specific device; (**c**) When another device accesses data set to Read mostly, a replica is created on the accessed device.

- ***Preferred location*** can set the physical location where data are actually stored. Data set as *Preferred location* are fixed in the designated physical location. Therefore, if the data are accessed from another device, data are not migrated to the device but data are directly mapped. For example, as shown in Figure 1a, if Device 1 attempts to access data fixed to Device 0, a page fault occurs in the first access. After the page fault, the data that are set as the *Preferred location* and fixed to Device 0 are mapped directly to the Device 1 instead of unmapping them from the memory of Device 0 and migrating to Device 1 memory. Therefore, the next time the Device 1 accesses the data, a page fault will not occur. *Preferred location* can operate efficiently when specific data needs to be physically fixed in specific device.
- ***Access by*** specifies direct mapping to a specific device, as depicted in Figure 1b; therefore, a device that is directly mapped by designating *Access by* does not generate a page fault when accessing the corresponding data. The difference between data set as *Access by* and data set as *Preferred location* is that data set as *Access by* are not fixed to a specific device but can be moved freely; therefore, the direct mapped device has the advantage that the data can be accessed without a page fault no matter where they are physically located; however, in the case of frequently accessed data, they may be

an overhead because it takes a long time to access them because they are physically located on a different device.

- ***Read mostly*** can be used efficiently when applied to read-intensive data. As you can see from Figure 1c, if data set to *Read mostly* are accessed by a device that is not physically stored, a replica is created on the accessed device instead of migrating the accessed data to the device that accessed the data. Therefore, the data set to *Read mostly* have the advantage that it can be accessed quickly from the device because there are copies in the device's memory after a page fault occurs in the first access; however, if the data set to *Read mostly* are modified, there is a disadvantage that the data must be modified for all devices where replication exists.

NVIDIA supports the data prefetch scheme in addition to memory advise to efficiently utilize CUDA Unified Memory technology. Before accessing the data allocated to unified memory, the user can call *cudaMemPrefetchAsync()* to prefetch the data to the device where the data will be used. Data prefetched through *cudaMemPrefetchAsync()* can be accessed without a page fault when accessed from the device because the data are already in the device. Data prefetch operates differently depending on the settings of memory advise. In the case of data set to Read mostly, if data prefetch is performed, a replica is created in the device. When the data set as the Preferred location are prefetched to a device other than the device where the data are fixed, the data are moved to another device instead of being fixed to the device.

2.3. PyTorch

PyTorch is one of the most representative open-source deep learning frameworks. PyTorch operates in a Define-by-Run method, unlike Caffe and Tensorflow, which are the representative deep learning frameworks that operate in a Define-and-Run method. By operating in the Define-by-Run method, PyTorch solves the difficulty of debugging, which was a disadvantage of the existing Define-and-Run deep learning framework. Further, PyTorch is a Python framework designed to be easy to use for existing Python users.

Due to the Global Interpreter Lock (GIL), Python can only run one thread at a time; therefore, Define-and-Run deep learning frameworks based on Python show better performance than Define-by-Run frameworks. PyTorch solves the performance degradation problem of Define-by-Run frameworks in the following way. PyTorch is implemented using the C++ language to achieve high performance. This allows PyTorch to run multiple threads at the same time, solving the problem that the GIL causes Python to only run one thread at a time. In addition, PyTorch can run Python code on the CPU while computing on the GPU via the CUDA Stream mechanism. Thus, PyTorch can increase the utilization of the GPU even when using Python, which has a large execution overhead. PyTorch is implemented to dynamically allocate and reuse memory according to the characteristics of deep learning, so users can use memory efficiently. In addition to the above methods, by extending Python's multiprocessing module, problems caused by GIL can be avoided and users can easily implement parallel programs. Further, PyTorch avoids wasting memory through reference counting.

3. Efficient Data Management Scheme for Deep Learning

In this study, we enable large-scale model training by expanding GPU memory by integrating CPU memory and GPU memory through CUDA Unified Memory technology in a single GPU environment. In this section, we propose a method to efficiently manage data used in deep learning using memory advise and data prefetch supported by CUDA 8.0. For example, an efficient convolution algorithm requires feature map data and layer weights required for feed forwarding, and an additional temporary buffer is required when using a fast-Fourier-transform-based convolution algorithm [19]. In this paper, we classify these data into two types: model parameters and intermediate results. Since each layer weight is part of the model parameters, the layer weights can be classified as model parameters. Feature map data and temporary buffer are data generated and used during

feed forwarding or backpropagation, so we classified these data as intermediate results. However, the data required to perform deep learning are not only intermediate results, such as feature map data and temporary buffers, and model parameters, but also input data to be trained. Therefore, we can classify the data needed in deep learning into three types: model parameters, intermediate results, and input data. The three types of data needed to perform deep learning are created, modified, and used at different times. This means that the three types of data needed to perform deep learning each have different access patterns.

The characteristics of the three types of data used in deep learning are as follows:

- First, there are **model parameters**. Model parameters are generated and initialized before starting deep learning. Model parameters are weights for the features of the input data. When performing feed forwarding, the weight of each hidden layer(part of model parameters) generates feature map data to be passed to the next hidden layer through mathematical operation with the feature map data output from the previous hidden layer. After performing backpropagation, model parameters are updated with new values. The updated model parameters will be used by the GPU for training in the next iteration. In addition, in most deep learning models, model parameters account for the smallest proportion of the total data of deep learning [19]; therefore, we recommend that the model parameters remain on the GPU until training is complete.
- Second, there are **intermediate results**. The intermediate result is the feature map data of each hidden layer that is generated as a result of mathematical operation of model parameters and input data when performing feed forwarding, and a temporary buffer used when backpropagation is performed. Therefore, intermediate results are newly generated and used by feed forwarding and backpropagation at every iteration, and are not used again after the iteration ends. Further, since the intermediate results are the largest among the three types of data required to perform deep learning, it is difficult to store all the intermediate results in the GPU memory [19]. Every intermediate result generated in each hidden layer during feed forwarding does not always have to be in the GPU memory. However, when performing backpropagation, the intermediate results generated during feed forwarding in each hidden layer must be in GPU memory before backpropagation of that hidden layer is performed.
- Finally, there are **input data**. The input data are the data to be trained by passing them to the first layer (input layer) of the deep learning model. Input data are delivered to the GPU, which performs deep learning, with new data every iteration. In addition, the input data are not changed due to the deep learning operation and are no longer needed after one iteration. Therefore, the input data do not always have to be on the GPU until training is finished.

The Table 1 summarizes the access patterns according to the types of deep learning data. According to the access characteristics of the deep learning data described above, it can be seen that the access patterns supported by memory advise and the access patterns of the deep learning data are similar. Model parameters remain in GPU memory until training is finished, and are accessed and updated by the GPU at every iteration. Therefore, model parameters can be managed efficiently when set as the *Preferred location*. Input data are accessed and used by GPU, but they are not changed due to deep learning operation, so they can be used efficiently by setting it to *Read mostly*. However, the input data are changed to new data read from the storage device every iteration. Due to this characteristic, it can be seen that the input data are modified with new data at the start of each iteration. It is also necessary to consider not using memory advise, since input data are accessed only once and are changed every iteration. Intermediate results are used by the GPU at every iteration; however, since the intermediate results are very large, it is difficult for all intermediate results to always reside on the GPU; therefore, intermediate results can be used efficiently by moving them to GPU memory through data prefetch before being used.

Table 1. Characteristics of deep learning data.

Data Type	Does Data Change Due to Deep Learning Operation?	Is Data Accessed by GPU on the Next Iteration?	Techniques for Managing Data
Model Parameters	Yes	Yes	*Preferred location*
Input Data	No	No	*Read mostly* or no memory advise
Intermediate results	Yes	No	Prefetch

4. Implementation in PyTorch

PyTorch, an open source deep learning framework, currently does not support CUDA Unified Memory technology; therefore, it is impossible to train a large-scale deep learning model that requires more than the GPU memory size through PyTorch in a single GPU environment. In this study, to overcome the limitations of existing PyTorch, CUDA Unified Memory technology is applied to PyTorch to enable large-scale deep learning model training.

PyTorch performs computation in the Define-by-Run method [20]. PyTorch does not allocate the amount of memory required for operation in advance in the way that Tensorflow does, which operates in a Define-and-Run method, but rather allocates GPU memory when a user requests or requires additional memory space due to GPU operation. The user can copy a specific tensor to another device's memory by calling the *to()* function. The user can call *to('cuda')* to copy data to GPU memory. The tensor on which *to('cuda')* is called is transferred to GPU memory, and the tensor can be used for computation using the GPU. Further, PyTorch automatically allocates additional GPU memory space for the computed results using the tensor in the GPU and stores the computed results in the GPU memory. Table 2 show the allocated GPU memory size according to the training progress when ResNet-50 is trained using PyTorch. The ResNet-50 model is a representative convolutional neural network (CNN) for image classification consisting of 50 convolution layers.

Table 2. GPU memory usage according to the training progress.

Initialize Training Model	After Feed Forwarding	After Backpropagation
519 MiB	6027 MiB	7121 MiB

Looking at Table 2, one can see that the GPU memory usage is 519 MiB when the ResNet-50 model is initialized in the GPU. Further, it can be seen that GPU memory usage increased to 6027 MiB after feed forwarding because PyTorch automatically allocated a memory space to additionally store intermediate results generated by performing feed forwarding. Finally, by performing backpropagation, additional intermediate results were generated, increasing the GPU memory usage to 7121 MiB. This shows that PyTorch allocates GPU memory when additional GPU memory is needed due to the user's *to('cuda')* call and the result of GPU operation.

As a result of analyzing the source code of PyTorch, we confirmed that all GPU memory allocation of PyTorch is performed through CUDACachingAllocator. CUDA Unified Memory technology can be applied to PyTorch simply by calling *cudaMallocManaged()* instead of *cudaMalloc()* in the *malloc()* function of CUDACachingAllocator. When implemented in a simple way, the user cannot use the general GPU memory allocation method, but also has a problem where only fixed memory advise can be set; therefore, a problem arises where users have no choice but to use unified memory even when the memory required for deep learning is smaller than the GPU memory. Further, even when unified memory is used due to insufficient GPU memory, CUDA Unified Memory technology cannot be used efficiently because the same memory advise must be set for data with different access patterns. Therefore, we need to implement so that users can choose whether to apply CUDA Unified Memory and choose memory advise. In order to implement the

above functions and allow users to use them, it is difficult to implement because it is necessary to implement a new user interface function and add a new option. To solve the above problems and efficiently apply CUDA Unified Memory technology, we analyzed PyTorch's GPU memory allocation method. As a result of the analysis, we confirmed that the functions were called in the order shown in Figure 2 when allocating memory by calling PyTorch *to("device:index")*.

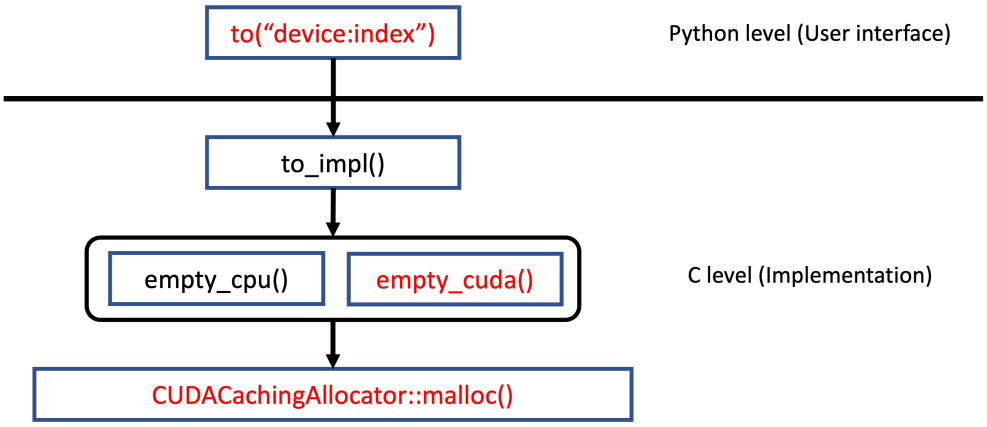


Figure 2. Order of memory allocation function calls via *to("device:index")*. (The functions we modified are colored in red.)

When a PyTorch user calls the Python level function *to("device:index")*, the C level implementation *to_impl()* is called. When *to("device:index")* is called, the *"device:index"* argument is passed to the C level and is used to create a device object. The *to_impl()* function checks the device information and calls *empty_cuda()* or *empty_cpu()* depending on the device type. Through the above analysis, we were able to confirm that the device information is transferred to the C level when *to("device:index")* is called. The device in the *"device:index"* argument is either 'cuda' or 'cpu', which tells the type of device on which the tensor operation should be performed. In addition, the index informs the number of the device to perform the operation from among the devices of the corresponding type. Further, the index is information on which GPU to store the tensor on. Thus, we implemented this to deliver information regarding memory advise to the C level through the index rather than the device.

In the Figure 2, the functions we modified to set the memory advise are marked in red. We modified a part of PyTorch's C level implementation so that memory advise information can be passed through index information when calling *to("device:index")*. The memory advise delivered through the index information is passed to the *malloc()* function of CUDACachingAllocator, so that the tensor created at the Python level can be set to the desired memory advise. Users can set tensor's memory advise simply by changing *tensor.to("cuda:index")* in the vanilla PyTorch code to *tensor.to("cuda:memoryadvise")*. When *tensor.to("cuda:memoryadvise")* is called, the tensor is copied to unified memory and memory advise is applied. In the case of prefetch, calling *tensor.to("cuda:memoryadvise")* not only copies the tensor to unified memory, but also calls *cudaMemPrefetchAsync()* to prefetch the tensor to GPU memory. Users can set the memory advise in five different ways: noadvise, preferredlocation, accessby, readmostly, and prefetch.

By modifying only the functions marked in red in the Figure 2, the intermediate results generated by the operation on the GPU cannot use CUDA Unified Memory. In addition, the information of memory advise of intermediate results cannot pass to the *malloc()* function of CUDACachingAllocator via *to("device:index")*. Therefore, we created a new Python level user interface function to set the memory advise of intermediate results, and using the newly created C level function, the information of the memory advise is passed to the *malloc()* function of CUDACachingAllocator. We newly added three functions colored

in blue in the Figure 3. Figure 3 shows the calling sequence of the newly implemented functions to pass the memory advise information of the intermediate results.

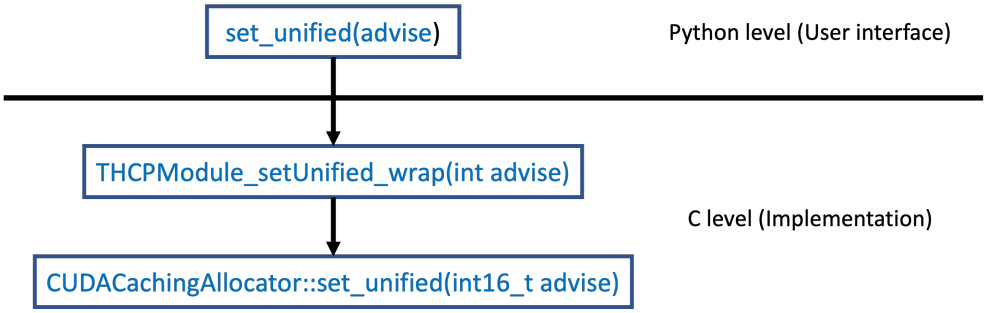


Figure 3. Sequence of function calls to pass memory advise of intermediate results. (The functions we added are colored in blue.)

We added the *set_unified(advise)* function at the Python level to set the memory advise of intermediate results. First, the user Calls the Python level function *set_unified(advise)*. After that, the memory advise is transferred from Python level to the C level via the C Python binding function *THCPModule_setUnified_wrap(int advise)*. The memory advise passed to the C level is passed to the *set_unified(int16_t advise)* function of CUDACachingAllocator implemented to set the memory advise of intermediate results. Finally, the *set_unified(int16_t advise)* function checks the received argument and sets memory advise; therefore, users can set the memory advise of intermediate results by simply adding *set_unified(advise)* to the vanilla PyTorch code. Further, users can make PyTorch use CUDA Unified Memory by calling the *set_unified(advise)* function. Users can input one of five types of advise: noadvise, preferredlocation, accessby, readmostly, and prefetch. Memory advise of intermediate results is applied when data are created. Further, in the case of prefetch, when data are created, *cudaMemPrefetchAsync()* is called and prefetched to the GPU.

Figures 4 and 5 show the vanilla PyTorch code and the modified PyTorch code to apply CUDA Unified Memory. *torch.cuda.set_device(args.gpu)* is an API that sets the GPU to be used for operation, so users do not need to modify it to apply CUDA Unified Memory. Users add *torch.cuda.set_unified(args.advise)* to make PyTorch use CUDA Unified Memory. when *torch.cuda.set_unified(args.advise)* is called, the memory advise of intermediate results is set. So, users input the desired memory advise to *torch.cuda.set_unified(args.advise)*. *model.cuda(args.gpu)* copies the training model to the GPU and works the same as *model.to("cuda:args.gpu")*. Users can copy the model to unified memory by changing *model.to("cuda:args.gpu")* to *model.to("cuda:args.parameter")*. Users can use CUDA Unified Memory with memory advise by modifying the vanilla PyTorch code with just two lines. We conducted experiments by changing the vanilla PyTorch code in Figure 4 as modified PyTorch code in Figure 5. The results of the experiment are described in the next section.

```
torch.cuda.set_device(args.gpu)
model = model.cuda(args.gpu)
```

Figure 4. PyTorch code to copy model to the GPU.

```
torch.cuda.set_unified(args.advise) # add for set cuda memory advise
torch.cuda.set_device(args.gpu)
model = model.to('cuda:{}'.format(args.parameter)) # add for set cuda memory advise
```

Figure 5. Modified PyTorch code to apply CUDA Unified Memory .

5. Experiment and Performance Analysis

To analyze the performance of PyTorch to which CUDA Unified Memory is applied, we configured the following experimental environment. We used Python 3.6.12, CUDA 10.1, NVIDIA driver 440.54, and PyTorch 1.6.0 as software. As GPU, NVIDIA's Quadro RTX

6000 was used, and the GPU memory size of Quadro RTX 6000 is 24 GB. As a deep learning model, BERT-Large model was used. BERT-Large consists of 24 layers and 16 self-attention heads, and the hidden size of BERT-Large is 1024. BERT-Large is a large-scale deep learning model with a total of 340 M model parameters. We used BERT in our experiments because it is a widely used representative large-scale deep learning model. When training a BERT-large using a general GPU, it is difficult to train with a mini-batch and max sequence of sufficient size. We were able to train the BERT-Large with a mini-batch size of 16 using a GPU with 24 GB of memory; however, it was not possible to train a BERT-Large with a mini-batch size of 32. We conducted the experiment by dividing the case where the data size is smaller than the GPU memory size and the case where the data size is larger than the GPU memory size.

5.1. Execution Time According to Memory Advise

Figures 6–9 show results of experiment. We performed 100 iterations of all experiments. The execution time shown in the graphs are the average time it takes to perform one iteration. The legends of the graphs are in the form of memory advise of model parameters-memory advise of input data. The legend of Original in Figure 6 is when unified memory is not used. In the graphs, PL stands for *Preferred location*. Further, RM means *Read mostly* and NA means no memory advise. The x-axis of the graphs shows the memory advise setup of the intermediate results. For example, in the case of PL-RM and noadvise in the graph, model parameters are set to *Preferred location*, input data are set to *Read mostly*, and memory advise of intermediate results is not set.

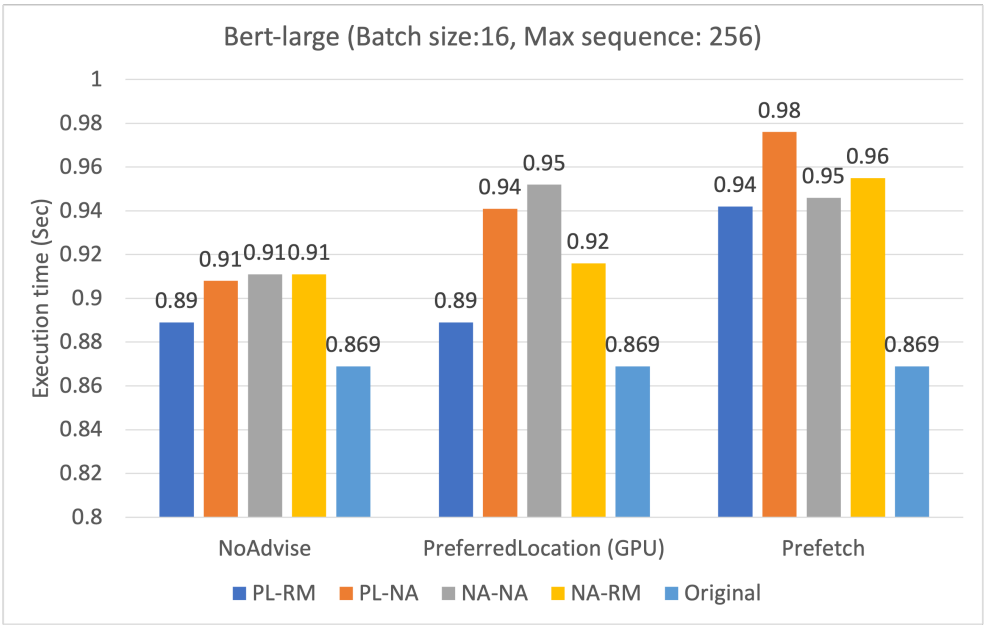


Figure 6. Execution time when the data size is smaller than GPU memory size. (Note that the y-axis of this graph starts with 0.8.)

Figure 6 shows the training time when the data size is smaller than the GPU memory size. The y-axis of Figure 6 starts with 0.8. BERT-Large can be trained with the input mini-batch size of 16 and the max sequence size of 256 without using unified memory. When deep learning is performed without applying the Unified Memory, the time to perform one iteration is about 0.87 s, which shows the best performance. On the other hand, when unified memory is applied, performance is lower than when unified memory is not applied. In particular, when the model parameters are set as the *Preferred location* and no advise is set in the input data, the training time when prefetching the intermediate results is about 0.98 s, showing the lowest performance. Further, when the memory advise of the model parameters and the memory advise of the input data are the same, prefetching

the intermediate result shows the lowest performance. The reason for showing the lowest performance when prefetching intermediate results is that if the data size is smaller than the GPU memory size, the communication overhead due to prefetch is larger than the deep learning operation time on the GPU; therefore, if deep learning can be performed in GPU memory, it is better not to prefetch intermediate results. When model parameters are set as *Preferred location* and input data are set to *Read mostly*, if *Preferred location* or memory advise is not set for intermediate results, the execution time is 0.89 s, which is only 0.019 s longer than when the unified memory is not applied. This is the best performance when unified memory is applied. The reason for such good performance seems to be that, when memory advise is set as described above, no other overhead occurs except for the overhead caused by the initial page fault when deep learning is performed.

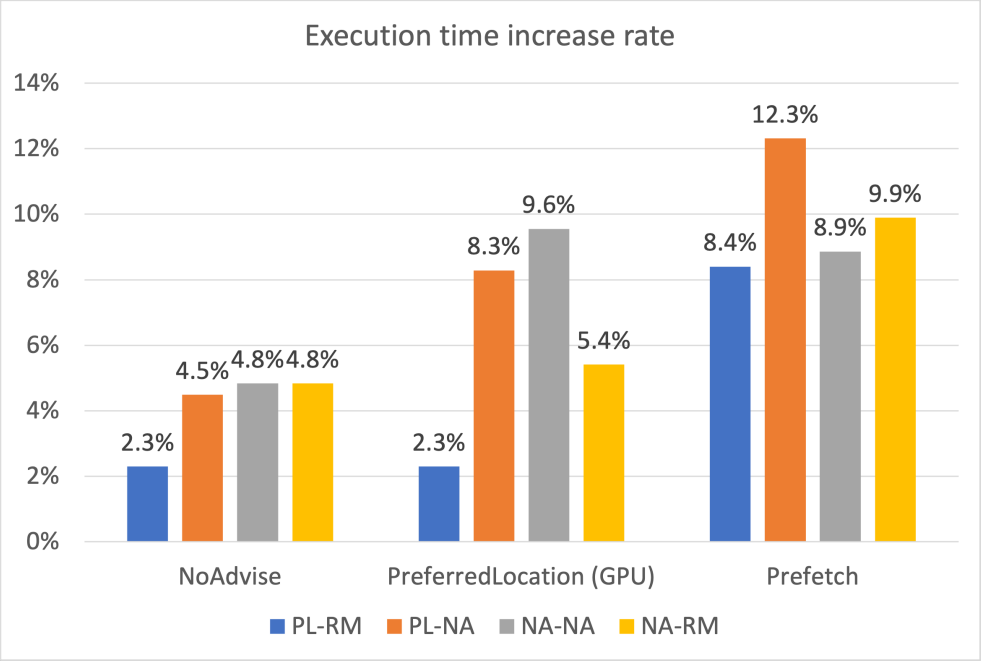


Figure 7. Training time increase rate due to the use of unified memory when the data size is smaller than the GPU memory size.

Figure 7 shows the increase rate of training time due to the use of unified memory when deep learning can be performed in GPU memory. The learning time increases by up to 12.3% and at least by 2.3% when the unified memory is applied. Therefore, if deep learning is possible within the GPU memory, applying unified memory cannot efficiently perform deep learning. Further, it is better not to prefetch intermediate results when unified memory is applied.

Figure 8 is a graph showing the execution time when deep learning cannot be performed without using unified memory because the data size is larger than the GPU memory size. Please note that y-axis of Figure 8 starts with 14. Figure 8 shows that the performance is the lowest when memory advise is not set for model parameters and input data; however, setting the model parameters to the *Preferred location* gives the best performance. If deep learning can be performed in GPU memory, prefetching intermediate results shows low performance. On the other hands, when training within the GPU memory is not possible, prefetching intermediate results showed the best performance. The reason for showing these contradictory results seems to be that the execution time of the deep learning operation is large enough to have little effect of communication overhead on performance. When the intermediate results are prefetched and the model parameters are set as the Preferred location, the execution time when the input data are set to Read mostly is 14.84 s. Further, the execution time is 14.83 s when no memory advise is set for input data.

Figure 9 shows the reduction rate of training time when memory advise is set on the basis of when memory advise is not set. Setting the model parameters to the *Preferred location* and prefetching intermediate results reduced training time by 9.4% and shortened the training time the most. On the other hand, if we do not set memory advise for model parameters and input data and set intermediate results as *Preferred locations*, the training time increases by 0.3%. Experimental results show that training can be performed most efficiently by setting the model parameter to the *Preferred locations* and prefetching intermediate results when learning within the GPU memory is not possible. Further, if the memory advise is not set properly, the performance may be worse than if the memory advise is not set.

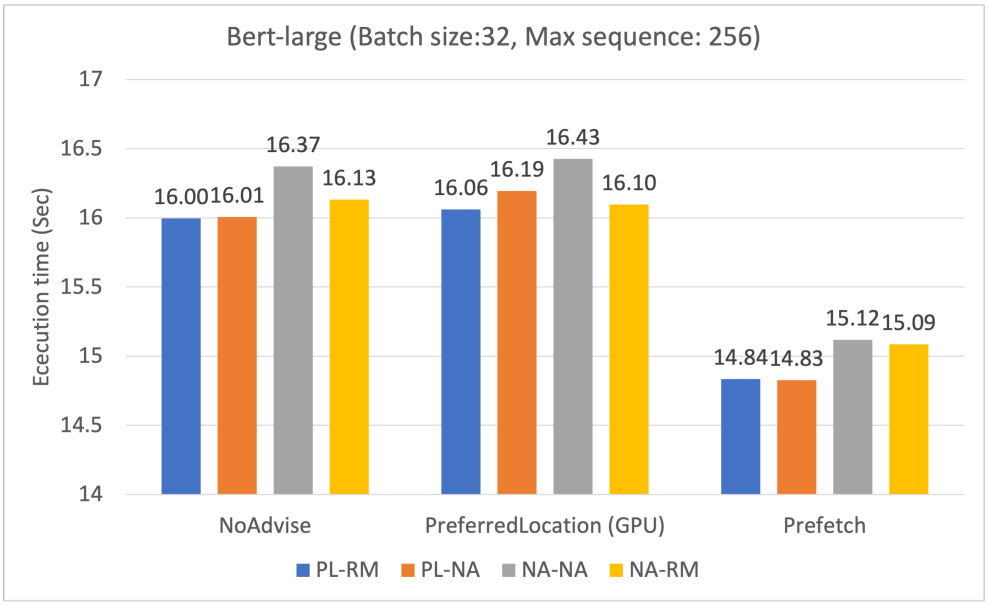


Figure 8. Execution time when data size is larger than GPU memory size. (Note that the y-axis of this graph starts with 14.)

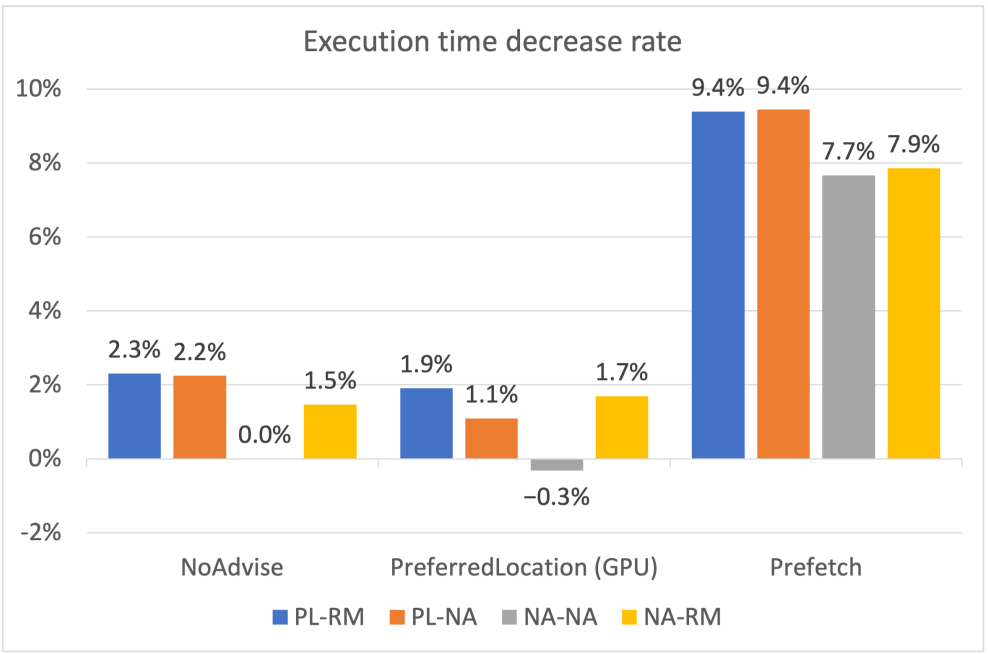


Figure 9. Reduction of training time according to memory advise when data size is larger than GPU memory.

5.2. Results of GPU Page Fault Profiling Using Nvprof

In this section, we profile the experiment (mini-batch size: 32, max sequence: 256) performed in the previous section when the data size is larger than the GPU memory size using nvprof, a profiling tool of NVIDIA, and analyze the profiling results. When the data size required for deep learning is larger than the GPU memory, the memory advise setting that shows the best performance is when the intermediate result data are prefetched, the model parameters are set as *Preferred location*, and input data are set to *Read mostly*. Further, the memory advise setting that showed the lowest performance is the case where the intermediate results are set as the *Preferred location* without giving memory advise to the model parameters and input data. Table 3 summarizes the profiling results of GPU page fault profiling. Table 3 is the result of profiling learning for 10 iterations by applying each memory advise. Table 3 shows that the number of page faults due to unified memory. When the memory advise is not applied, the page faults occur the most. The difference in the elapsed time for data migration of unified memory according to memory advise is so small that it is meaningless. Further, data migration takes more time for Host to Device than Device to Host. Therefore, looking at Table 3, as can be expected from the results shown through the graphs of the Figures 8 and 9 in the previous section, the memory advise with the best performance generated the least number of page faults.

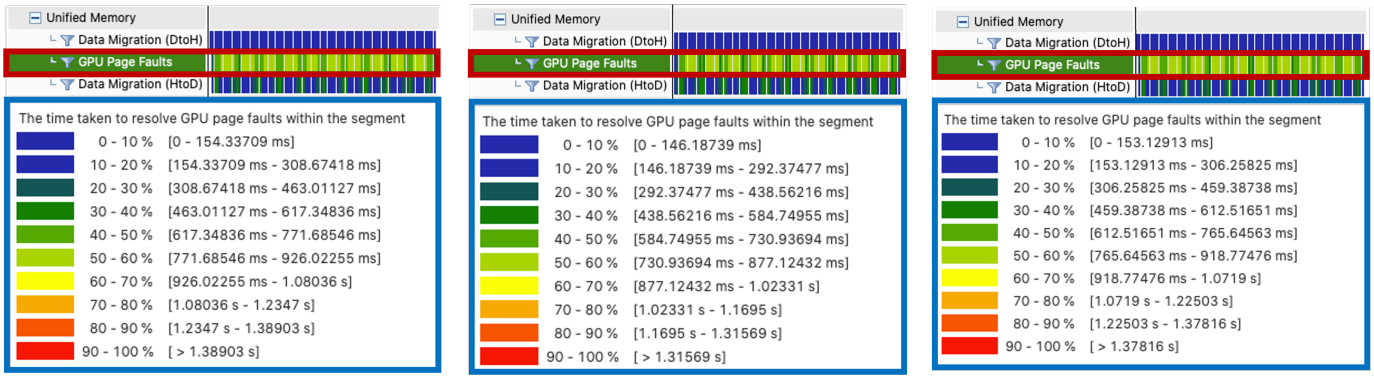
Table 3. Summary of profiling results.

	No Advise	Best Advise	Worst Advise
Page fault	5,782,973	5,645,687	5,695,792
Data migration Device to Host (s)	23.327	24.068	23.334
Data migration Host to Device (s)	40.846	39.636	40.867

We used NVIDIA Visual Profiler, a graphic profiling tool, to check the pattern of page faults according to each memory advise. Figure 10 are visualizing the profiling results using NVIDIA Visual Profiler. Figure 10a is the case where no memory advise is applied, and Figure 10b,c are the results of applying the memory advise showing the highest performance and the memory advise showing the lowest performance, respectively. In the upper red area of Figure 10a–c, you can check the pattern of page faults due to unified memory when deep learning is performed. NVIDIA Visual Profiler divides the execution of the process into equal-width sections, and expresses the percentage of time it takes to resolve GPU page faults in each section as a color. As can be seen in the upper red area of each figure, colors appear in a similar pattern regardless of memory advise when deep learning is performed. This shows that page faults occur in a similar pattern regardless of memory advise when deep learning is performed. In the blue area at the bottom of Figure 10a–c, you can check the time it takes to resolve the page fault according to the color of each section. The width of each section of unified memory in NVIDIA Visual Profiler is the total execution time divided by the same number; therefore, even with the same color in the three pictures, the time taken to resolve the page fault is the shortest when the memory advise that shows the best performance is applied. From the profiling results, we can confirm that GPU page faults can be reduced by applying the optimal memory advise. In addition, by reducing GPU page faults, the overhead caused by page faults is reduced, and thus, operator resources of the GPU can be used more efficiently.

Figure 11 shows the pattern of GPU page faults when training a single mini-batch. The area marked in blue in Figure 11 is the area performing feed forwarding and the area marked in red is the area performing backpropagation. Figure 11 shows that most sections are displayed in green when feed forwarding is performed. Further, when performing backpropagation, it can be seen that the ratio of the section marked in yellow is high. The green section means that the percentage of the time it takes to resolve the page fault is 40% to 50%, and the yellow section means 60% to 70%. Through this, we can see that the ratio of the time of the operation is delayed due to a page fault is greater when performing

backpropagation than feed forwarding; therefore, in order to more efficiently utilize the computing resources of the GPU when performing deep learning by applying unified memory, it is thought that we need a technique to identify and prefetch the data required in the backpropagation operation.



(**a**) (**b**) (**c**)

Figure 10. This is a visualization of the profiling results using the NVIDIA Visual Profiler: (**a**) The profiling result when memory advise is not set; (**b**) A visualization of the profiling results when configured with memory advise that shows the best performance; (**c**) A visualization of profiling results when configured with memory advise, which shows the worst performance.

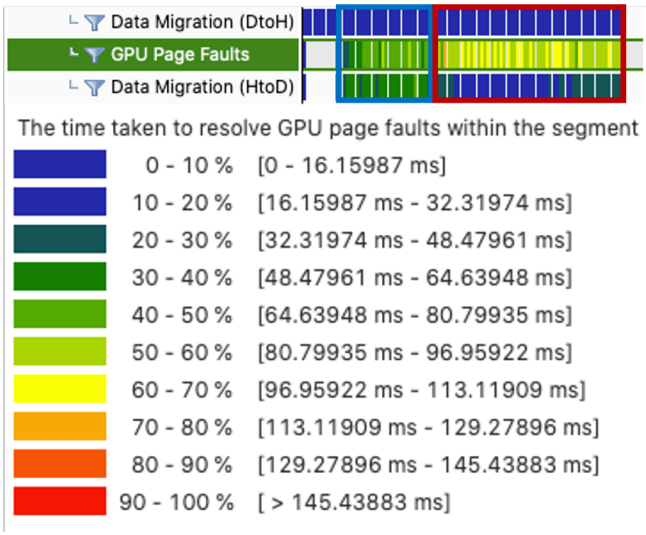


Figure 11. GPU page fault pattern in single mini-batch training.

6. Related Works

Among the methods for optimizing GPU memory when performing deep learning, there is a swap in/out method between GPU memory and CPU memory. vDNN, a memory management system provided by NVIDIA, operates by swapping in/out of feature map data generated by deep learning operations to overcome the limitations of GPU memory [19]. During feed forwarding, vDNN swaps out feature map data not currently used in GPU memory to CPU memory. After that, vDNN swaps in the feature map data required for backpropagation from CPU memory to GPU memory; therefore, it is possible to learn a mini-batch larger than the size of the GPU memory. vDNN is similar to our proposed technique in that it improves the performance of deep learning through efficient memory management. However, unlike vDNN, our work utilizes CUDA Unified Memory to expand GPU memory. In addition, we apply memory advise according to data access patterns to efficiently use CUDA Unified Memory.

Kim, Y., Lee, J., Kim, J.S. et al. proposed a new memory management method for accelerating deep learning applications [11]. They extended NVIDIA's vDNN concept to address the computational performance degradation caused by PCIe bottlenecks in multi-GPU environments. To address the PCIe bottleneck, they limited the number of GPUs that swap feature map data generated in a specific hidden layer when performing deep learning. Further, they designed an algorithm to effectively prefetch the swapped out feature map data into GPU memory. Unlike their research, we propose a memory expansion method in a single GPU environment using CUDA Unified Memory technology and memory advise.

S. Chien, I. Peng and S. Markidis. et al. evaluated the performance of CUDA Unified Memory technology and memory advise on the Intel-Volta/Pascal-PCI platforms and the Power9-Volta-NVLink platform [13]. In their study, they evaluated the performance of CUDA Unified Memory and memory advise using eight applications such as matrix multiplication and convolution operations that use GPUs. Their study evaluated the performance of CUDA Unified Memory and Memory Advise on various hardware platforms. Through experiments, they showed that when the size of data exceeds the size of GPU memory, using memory advise for CUDA Unified Memory can improve performance. However, their study has a limitation in that there is no performance evaluation for the widely used deep learning workloads. Our study shows that the size of trainable mini-batch increases by applying CUDA Unified Memory to deep learning workloads, and that memory can be managed efficiently by using memory advise.

Most of the datasets used in GPU applications such as deep learning are very large, which is not only larger than the GPU memory size but also larger than the host memory size [21]. In most HPC systems, data sets are stored in non-volatile memory (NVM) storage, which has a large capacity; therefore, for complex applications, data movement between GPU memory and NVM storage can be a difficult problem. To solve this problem, Direct Resource Access for GPUs Over NVM (DRAGON) was developed. DRAGON extended GPU memory to NVM storage, not just host memory, by extending NVIDIA's Unified Memory technology; they optimize the data movement by dividing the data access pattern into read-only and write-only, and separately manage the intermediate data generated through GPU operation. DRAGON optimizes only for read-only and write-only, so there is a limitation in that it is difficult to optimize according to data access patterns when performing deep learning. Our proposed scheme can manage memory more efficiently by managing deep learning data by dividing it into three types of model parameters, intermediate results, and input data.

To overcome the limitations of large-scale deep learning model training, Microsoft developed Zero Redundancy Optimizer (ZeRO) by integrating existing data parallelization techniques and model parallelization techniques [22]. Through ZeRO-DP, Microsoft achieved both computational and communication efficiency, which has the advantage of data parallelism, and memory efficiency, which has the advantage of model parallelism, overcoming the limitations of existing parallelization techniques. Microsoft has also developed ZeRO-R to optimize three types of residual state memory: activations, temporary buffers, and unusable memory fragments. Microsoft integrated ZeRO-DP and ZeRO-R to implement ZeRO, a memory-optimized system for deep learning. ZeRO efficiently utilizes multiple GPUs to train large-scale models. Unlike ZeRO, we propose a technique for efficiently training large-scale models on a single GPU environment. In addition, we divide data used for deep learning into three types according to access patterns and apply memory advise to each type of data to efficiently utilize GPU memory.

GPU-Enabled Memory-Aware Model-Parallelism System (GEMS) has been proposed to train large-scale deep learning models using high-resolution images, which are mainly used in digital pathology [23]. In their paper, four types of techniques are proposed: GEMS-Basic, GEMS-MAST, GEMS-MASTER, and GEMS-Hybrid. GEMS-MAST trains a replica of a part of the model by utilizing memory and computing resources that were not used in the existing model parallelization technique. GEMS-MASTER uses more replicas of model

that GEMS-MAST used only two. Through this, GEMS-MASTER can train the model faster by overlapping computations and reducing the number of allreduce executions. They also achieved near-linear scalability with the GEMS-Hybrid. Unlike our work, GEMS requires the use of multiple GPUs to train large-scale models. Our work uses CUDA Unified Memory technology to train large-scale models even in a single GPU environment, and can efficiently manage memory to improve training performance.

Out-of-Core DNN training framework (OC-DNN) is a deep learning framework that uses the new Unified Memory feature with Volta and Pascal GPUs [24]. The two main components of OC-DNN are the Interception Library(IL) and OC-Caffe. IL is an independent library that operates between CUDA Runtime/Driver and HPC Platform. IL allows applications to run using data that exceed GPU memory without modifying existing applications. OC-Caffe can save GPU memory and overlap data copy and computation by using Unified Memory buffer. In addition, OC-Caffe changes the existing device-to-device communication to Managed memory-to-Managed memory communication. Further, they use *cudaMemAdvise* and *cudaMemPrefetch* to optimize intra-GPU communication. OC-DNN and our study are similar in that both use CUDA Unified Memory technology. Unlike OC-DNN, We divide deep learning data into three types and use memory advise, a hint for memory management provided by NVIDIA, to efficiently manage memory to improve training performance.

7. Conclusions

In this study, we propose an efficient data management scheme for deep learning using CUDA Unified Memory and memory advise through an experiment to train a large-scale deep learning model. Further, we applied CUDA Unified Memory technology to PyTorch and trained a large-scale model, BERT-Large. According to the results of the experiments conducted in this paper, using unified memory with our optimized memory advise scheme shows similar performance to the vanilla method with negligible overheads. On the other hand, when training on a single GPU is not possible, we show that the most efficient way to use unified memory is to prefetch intermediate results and set the model parameters to *Preferred location* to pin the model parameters to the GPU performing deep learning. If memory advise is properly configured by our proposed method, the deep learning execution time can be reduced by up to 9.4% compared to the vanilla method. Our approach does not modify the training algorithm of deep learning. Most of the deep learning models other than BERT-Large used in this paper can also classify data in the same way as in this study. Therefore, our proposed scheme can improve performance even when training other deep learning models than BERT-Large.

In the future, we plan to perform experiments in various environments and optimize the unified memory by additionally implementing the data prefetch function when performing backpropagation of unified memory.

Author Contributions: Conceptualization, J.L. and H.C.; methodology, J.L. and H.C.; software, H.C.; writing—original draft preparation, H.C. and J.L.; writing—review and editing, H.C. and J.L.; supervision, J.L.; project administration, J.L.; funding acquisition, J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Basic Science Research Program (NRF-2020R1F1A1072696) through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT, MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2021-2018-0-01423) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation) and the GRRC program of Gyeong-gi province (No. GRRC-KAU-2017-B01, "Study on the Video and Space Convergence Platform for 360VR Services").

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Deng, J.; Wei, D.; Socher, R.; Li, L.-J.; Li, K.; Li, F.-F. ImageNet: A large-scale hierarchical image database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009; pp. 248–255.
2. Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; Rabinovich, A. Going deeper with convolutions. In Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA, 7–12 June 2015; pp. 1–9.
3. Real, E.; Aggarwal, A.; Huang, Y.; Le, Q. Regularized Evolution for Image Classifier Architecture Search. *Proc. Aaai Conf. Artif. Intell.* **2018**, *33*, 4780–4789. [CrossRef]
4. Huang, Y.; Cheng, Y.; Bapna, A.; Firat, O.; Chen, D.; Chen, M.; Lee, H.; Ngiam, J.; Le, Q.V.; Wu, Y.; et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, BC, Canada, 8–14 December 2019; pp. 103–112.
5. Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, 7–9 May 2015.
6. Huang, G.; Sun, Y.; Liu, Z.; Sedra, D.; Weinberger, K.Q. Deep networks with stochastic depth. In *European Conference on Computer Vision*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 646–661.
7. The Next Platform. Baidu Eyes Deep Learning Strategy in Wake of New GPU Options. 2016. Available online: www.nextplatform.com (accessed on 7 September 2021).
8. Diamos, G.; Sengupta, S.; Catanzaro, B.; Chrzanowski, M.; Coates, A.; Elsen, E.; Engel, J.; Hannun, A.; Satheesh, S. Persistent RNNs: Stashing Recurrent Weights on-Chip. In Proceedings of the 33rd International Conference on International Conference on Machine Learning, ICML'16, New York City, NY, USA, 19–24 June 2016; pp. 2024–2033.
9. Kim, Y.; Lee, J.; Kim, J.S.; Jei, H.; Roh, H. Efficient Multi-GPU Memory Management for Deep Learning Acceleration. In Proceedings of the 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, Italy, 3–7 September 2018; pp. 37–43. [CrossRef]
10. Kim, Y.; Choi, H.; Lee, J.; Kim, J.; Jei, H.; Roh, H. Efficient Large-Scale Deep Learning Framework for Heterogeneous Multi-GPU Cluster. In Proceedings of the 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W), Umea, Sweden, 16–20 June 2019; pp. 176–181.
11. Kim, Y.; Lee, J.; Kim, J.S.; Jei, H.; Roh, H. Comprehensive techniques of multi-GPU memory optimization for deep learning acceleration. *Clust. Comput.* **2020**, *23*, 2193–2204. [CrossRef]
12. Kim, Y.; Choi, H.; Lee, J.; Kim, J.S.; Jei, H.; Roh, H. Towards an optimized distributed deep learning framework for a heterogeneous multi-GPU cluster. *Clust. Comput.* **2020**, *23*, 2287–2300. [CrossRef]
13. Chien, S.W.; Peng, I.; Markidis, S. Performance Evaluation of Advanced Features in CUDA Unified Memory. In Proceedings of the 2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC), Denver, CO, USA, 18 November 2019; pp. 50–57.
14. Mark Harris. Unified Memory for CUDA Beginners. 2017. Available online: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/ (accessed on 7 September 2021).
15. Nikolay Sakharnykh. Everything You Need to Know about Unified Memory. 2021. Available online: https://on-demand-gtc.gputechconf.com/gtcnew/sessionview.php?sessionName=s8430-everything+you+need+to+know+about+unified+memory (accessed on 7 September 2021).
16. Nikolay Sakharnykh. 2016. Beyond GPU Memory Limits with Unified Memory on Pascal. Available online: https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/ (accessed on 7 September 2021).
17. NVIDIA. 2021. CUDA C++ Programming Guide. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed on 7 September 2021).
18. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, MN, USA, 2–7 June 2019; Volume 1 (Long and Short Papers); Association for Computational Linguistics: Stroudsburg, PA, USA, 2019; pp. 4171–4186. [CrossRef]
19. Rhu, M.; Gimelshein, N.; Clemons, J.; Zulfiqar, A.; Keckler, S.W. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 15–19 October 2016.
20. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*; Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R., Eds.; Curran Associates, Inc.: Barcelona, Spain, 2019; pp. 8024–8035.

21. Markthub, P.; Belviranli, M.E.; Lee, S.; Vetter, J.S.; Matsuoka, S. DRAGON: Breaking GPU Memory Capacity Limits with Direct NVM Access. In Proceedings of the SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, 11–16 November 2018; pp. 414–426. [CrossRef]
22. Rajbhandari, S.; Rasley, J.; Ruwase, O.; He, Y. ZeRO: Memory Optimizations toward Training Trillion Parameter Models. In Proceedings of the SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 9–19 November 2020.
23. Jain, A.; Awan, A.A.; Aljuhani, A.M.; Hashmi, J.M.; Anthony, Q.G.; Subramoni, H.; Panda, D.K.; Machiraju, R.; Parwani, A. GEMS: GPU-Enabled Memory-Aware Model-Parallelism System for Distributed DNN Training. In Proceedings of the SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 9–19 November 2020; pp. 1–15.
24. Awan, A.A.; Chu, C.H.; Subramoni, H.; Lu, X.; Panda, D.K. OC-DNN: Exploiting Advanced Unified Memory Capabilities in CUDA 9 and Volta GPUs for Out-of-Core DNN Training. In Proceedings of the 2018 IEEE 25th International Conference on High Performance Computing (HiPC), Bengaluru, India, 17–20 December 2018; pp. 143–152. [CrossRef]