Breaking the limitation of GPU Memory for Deep Learning Workloads

Haoyu Zhang[1,a)] Mohamed Wahib[2,b)] Lingqi Zhang[1,c)] Yohei Tsuji[1,d)] Satoshi Mtsuoka[3,1,e)]

Abstract: GPU memory can be insufficient for Deep Learning workloads with respect to the model and dataset sizes. Although model parallelism could help, significant modification of the code is needed for every case. An alternative general solution to this problem is to use out-of-core methods. Recent work proposed data-swapping and CUDA Unified Memory (UM) methods to break the limitation of GPU memory capacity. However, there is a lack of detailed analysis, via performance modeling, of the behavior and limitations of those methods. In this paper we analyze the behavior in terms of both single layer and the whole model. as well as propose a performance model based on the analysis to study how out-of-core training behaves and hence empower the co-design process for Deep Learning workloads.

Keywords: GPU, Out-of-Core, Deep Learning, Memory, Performance Model

1. Introduction

As the model size and the scale of datasets for Deep Learning (DL) become increasingly large, the memory consumption of training Neural Networks (NNs) increases dramatically. Even though the latest generation of Nvidia GPUs have up to 16GB (V100), that Capacity still seems insufficient in a lot of cases. For example, if we want to use Kronecker-Factored Approximate Curvature (K-FAC)[1][2] optimizer in with large network, such as ResNet-200[3], the local batch-size of training can not be larger than 6 samples, and in ResNet-1001, the local batch-size drops to 2 samples. This problem also happens in Faster-RCNN[4], the local batch-size can not go over 2 samples when training on ResNet-200. Distributed data parallelism can be used if enough GPUs are available for training. However, the accuracy of training can be affected since the batch normalization layer doesn't work well with small local batch-sizes[5].Since normal implementations of Batch Normalization (BN) in famous frameworks like Caffe, PyTorch, Tensorflow are all unsynchronized. This implementation will leads to data only normalized within each GPU separately. In normal case, the local batch-size usually already large enough for BN layers to work. But in some case, the local batch-size will be only 2 or 4 in one GPU, which will suffer a lot from the sample bias, and further degrade the accuracy. In addition, one may not be able to train with even one sample when the sample is extremely large, such as the case with high resolution satellite Imagery. Images can be up to 400MB/sample while the widely used ImageNet[6] dataset has images that are smaller than 100KB/sample (re-sized to 224×224).

Although model parallelism could be a solution, significant modification of the code is needed for every case. Another general solution to this memory capacity problem is to use out-of-core methods. Recent work already proposed data-swapping and CUDA Unified Memory (UM) methods[7] to break the GPU memory limitation. However, there is a lack of detailed analysis for the behavior and bottlenecks of those out-of-core methods. In this paper we analyze the behavior in terms of both single layer and the whole model as well as proposed a performance model based on these analysis to estimate the training time of these methods, hence empower the co-design process for Deep Learning workloads.

2. Background and Related Work

2.1 vDNN and vDNN++

Virtualized DNN (vDNN)[8] is a runtime memory manager that virtualizes the memory usage of DNNs in order to enable training DNNs with both GPU and CPU memory simultaneously when training huge DNNs that can not fit into GPU memory. It utilized the cudaMemcpyAsync() API to swap-in/out data between CPU and GPU memory and used two streams:$stream_{memory}$ and $stream_{compute}$ to enable data swapping and layer computing process simultaneously. However, the performance of vDNN is not quite good due to its poor data swapping schedule. It synchronizes computation and swap-in/out of data at the end of each layer, which can cause inefficiencies.

In the improved version vDNN++[9], the schedule was improved, the authors proposed a better schedule that enable the computation to begin as early as possible.

[1] Tokyo Institute of Technology, Dept. of Mathematical and Computing Science, Tokyo, Japan
[2] AIST-Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory
[3] RIKEN Center for Computational Science,Hyogo,Japan
[a)] zhang.h.am@m.titech.ac.jp
[b)] mohamed.attia@aist.go.jp
[c)] zhang.l.ai@m.titech.ac.jp
[d)] tsuji@smg.is.titech.ac.jp
[e)] matsu@is.titech.ac.jp

2.2 PALEO

PALEO[10] is a performance model for the consumption of resources during Deep Learning model training. It divided execution time into two parts: compute time and communication time. Each portions is estimated using information of the user's environment (hardware, framework, algorithms, and communications strategies). The compute time is calculated by several factors that include: the input, the type of layers, and the hardware performance. On the other hand, the communication time is estimated by calculating the computational dependencies, the bandwidth of the hardware and parallel strategies. Given a specified network model and hardware information, PALEO can estimate the execution time without actually running the training code. This paper builds on the method of PALEO to out-of-core methods in order to estimate the execution time of training model out-of-core.

3. Performance Model

3.1 Assumptions

Our performance model is based on the following assumption:

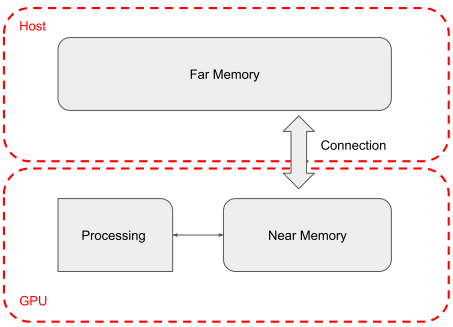

Fig. 1: Structure of Hardware

- We consider the GPU memory as Near Memory and other memory/storage as Far Memory.
- The connector between the two memories (e.g. PCIe) is bi-directional.
- Re-computation is not used.
- swap-in/out in is applied using a given strategy and schedule (will be discussed in following sections)
- Figure 1 shows the abstract structure of hardware used for the performance model.

3.2 Parameters

Table 1 shows the parameters in our performance model:

3.3 Single Layer Analysis

The goal of our model it to achieve the highest performance possible, which can be represented as the product of utilization and the peak performance.

$$MAX\Big[Proc_{util} \times Proc_{peak}\Big](At\ time\ T_j)$$

According to the little's law:

Table 1: Summary of parameters

Part	Var	value
Connection		
Conn	$Conn_{Th}$	Connection Throughput(buffers/second)
	$Conn_L$	Connection Latency
Far Memory (We defined memory out of GPU as Far Memory)		
FM	FM_{Th}	Far Memory Throughput(buffers/second)
	FM_L	Far Memory Latency
	FM_{Ca}	Far Memory Capacity
Near Memory (We define memory of GPU as Near Memory)		
NM	NM_{Th}	Near Memory Throughput(buffers/second)
	NM_L	Near Memory Latency
	NM_{Ca}	Near Memory Capacity
Processing		
Proc	$Proc_{Th}$	Processing Throughput(buffers/second)
	$Proc_L$	Processing Latency
	$Proc_{Peak}$	Processing Peak Performance
	$Util$	Utilization of Processing Unit
Buffers		
Buffer	$Buffer_i$	i=1...N, the variables are devided into buffers
	$Buffer\_size_i$	Size of each buffer
Time Step		
T	T_j	1...T, time steps during training
swap-in/out		
Swap_in	$Swap\_in_{Th}$	Throughput for swap-in data
	$Swap\_in_{size}$	number of buffers swapped in
Swap_out	$Swap\_out_{Th}$	Throughput for swap-out data
	$Swap\_out_{size}$	number of buffers swapped out

$$Concurrency = Latency \times Throughput$$

The utilization in time step T_j can be represented as follows:

$$\begin{aligned} Proc_{util} &= Concurrency(T_j)/Concurrency_{full} \\ &= Concurrency(T_j)/Proc_{Th} \times Proc_L \end{aligned} \quad (1)$$

The concurrency at time step T_j means the number of buffers that are available in GPU memory at the current time step, which means the utilization can be formulated as:

$$Proc_{util} = Num\_avil(T_j)/Proc_{Th} \times Proc_L \quad (2)$$

To get the number of available buffers, we have to know which buffer are left after processing at previous time step, and what layers have been swapped in during the same time period. So $Num\_avil(T_j)$ can be represented as:

$$\begin{aligned} Num\_avil(T_j) &= Num\_avil(T_{j-1}) - Proc_{Th} \times T + Swap\_in_{Th} \times T \\ &= Num\_avil(T_{j-1}) + (Swap\_in_{Th} - Proc_{Th}) \times T \end{aligned} \quad (3)$$

From this equation, given the assumption that we are doing same processing on each layer,they will take the same time, we can know that if $Swap\_in$ is faster than Processing. The value of Num_{avil} will keep growing until the memory consumption hit the upper limit of GPU memory.

However, if $Num\_avil(T_{j-1}) < Proc_{Th}$, which means the previous time step, there would not be enough buffers for the GPU to process. The concurrency will not be $Concurrency_{full}$, it will be the same as $Num\_avil(T_{j-1})$. In other words, what remains after processing should be $Max(Num\_avil(T_{j-1}) - Proc_{Th} \times T, 0)$, then we can get:

$$Num\_avil(T_j) = Max(Num\_avil(T_{j-1}) - Proc_{Th} \times T, 0) + Swap\_in_{Th} \times T \quad (4)$$

Which means if processing is faster, the value will always be $Swap\_in_{Th}$.

Assume that we can keep the swap-in data, The $Swap\_in_{Th}$ can be calculated by hardware parameters.

$$Swap\_in_{Th} = Min(FM_{Th}, NM_{Th}, Conn_{Th}) \quad (5)$$

As discussed before, if swap-in faster than processing, the memory consumption will finally hit the limitation (GPU memory capacity). The speed of swap-in will be affected by the memory space left. At time step T_{j-1} if there is not enough memory space for swap-in, the number of buffers swapped in at time step T_j will be $Capacity - Num\_avil(T_{j-1})$, while the concurrency remains the same:

$$Num\_avil(T_j) = Max(Num\_avil(T_{j-1}) - Proc_{Th} \times T, 0) + Min(Swap\_in_{Th} \times T, Capacity - Num\_avil(T_{j-1})) \quad (6)$$

3.4 vDNN-Like Strategy

On the condition that GPU is always doing same processing, all layers in the model are totally the same, which means processing faster or swapping faster will be similar for each layer in the model. However, in real models, the layers will never be totally the same. We take a simple example [8] (vDNN) as swap-in/out strategy for our model (shown in figure 2), during backward propagation buffer of $layer_{l-1}$ starts swap-in at the same time as $layer_l$ starts processing. And $layer_l$ will run right after both swap-in and $layer_{l+1}$ processing finished.

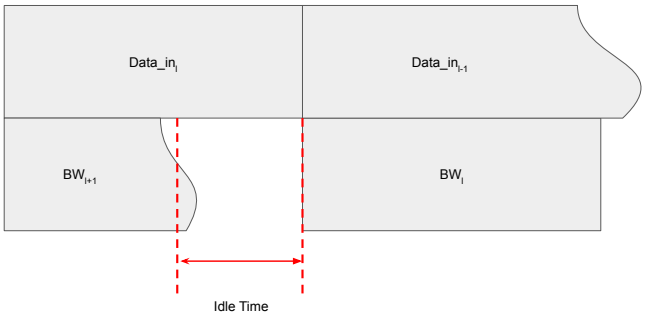

Fig. 2: Time line of swap-in strategy

In this case, different layer have different $Proc_{Th}$ and $Proc_L$ while we can assume the attributes related to swap-in/out stay the same. If swap-in of $layer_l$ takes more time than process $layer_{l+1}$, there will be idle time in during the backward, which makes the training efficiency lower.

we divide backward to several sections, the start time of each section is the start time of processing (or swap-in, they are same time point), while the end time is the time that both swap-in and processing finishes at.

$$\begin{aligned} T &= Max(T_{proc}(l), T_{Swap}(l-1)) \\ &= Max(Num\_buffer(l)/Proc_{Th}(l), Num\_buffer(l-1)/Swap\_in_{Th}) \end{aligned} \quad (7)$$

Then we can get the average concurrency of this section:

$$Concurrency = Proc_{Th} \times Proc_L \times T_{Proc}/T \quad (8)$$

Further we can get the utilization:

$$\begin{aligned} Util &= T_{Proc}/T \\ &= \frac{Num\_buffer(l)/Proc_{Th}(l)}{max(Num\_buffer(l)/Proc_{Th}(l), Num\_buffer(l-1)/Swap\_in_{Th})} \end{aligned} \quad (9)$$

3.5 Capacity-based Strategy

Now that swap-in is only affected by the memory capacity of the GPU, there is no dependency on which layer is being processed, we should keep on swapping in as far as we have enough Memory space for the buffer. This enables us to keep as much as possible available data for processing at any time step (obviously any wait will cause a drop in the average swap-in throughput).

On the other hand, in framework that call cuDNN, GPUs can not do any processing until all buffers for the layer have been prepared (i.e. swapped in) during backward propagation.

During Forward propagation, only swap-out would happen. One simple strategy is to swap-out whenever one layer's forward step is concluded (including the last layer). This can cause performance inefficiency before the backward phase starts. We have to wait until the last layer is swapped out and then swap-in it again before we can start the backward phase. (shown in Figure 3)

Figure 4 shows the capacity based swap strategy. We can get a good estimation of the memory consumption from the parameters of the layers in the model. This means even before the forward starting, we can know when to stop the swap-out (from layer 3 in this example). When the backward phase starts, the data needed still remains in the GPU memory, so the process can start as soon as the forward stage ends. When one layer is processed, the data will be swapped out immediately, in order to make new space for data to swap-in. In this example, layer 2 can be swapped-in in a very early stage to avoid some CPU stalling caused by data dependency.

Now that we have clear strategies for swapping data (i.e. capacity based swap schedule), we can make a more detailed performance model based on this.

Let the number of layers that can be kept in GPU memory be Ca_{layers}. At the very beginning of the backward stage, the processing of the layers will not be impacted by the swap-in. If swap-in is fast enough, all the layers will be swapped in before the processing of the first layer. But if swap-in is relatively slow, the processing may finally catch up with swap-in at a certain time step T_{catch}, which satisfies the equation:

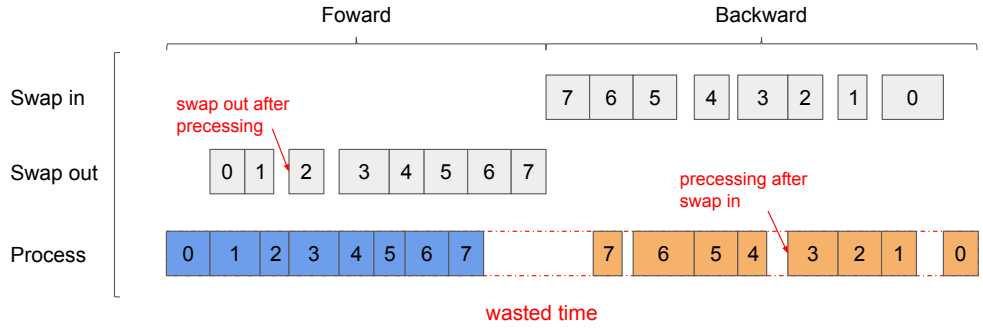

Fig. 3: Simple vDNN schedule may leads to waste of time when switching from forward to backward, backward of 7 have to wait until 7 swapped in

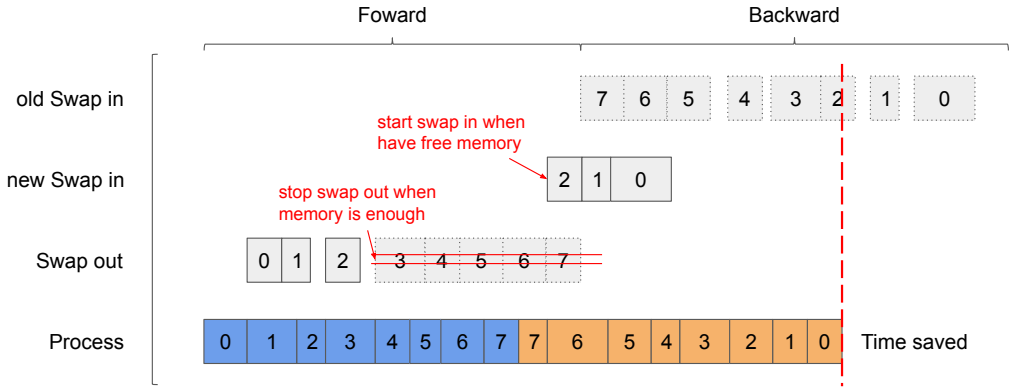

Fig. 4: Capacity Based schedule can save the time wasted by waiting for swap-out and make swap-in much earlier

$$\sum_{i=1}^{Ca_{layers}} (Proc_{th}(i) \times T_{proc}(i)) = Swap\_in_{Th} \times T_{catch} + NM_{Ca} \tag{10}$$

The value of T_{Catch} can be calculated by:

$$T_{catch} = \frac{\sum_{i=1}^{Ca_{layers}} (Proc_{th}(i) \times T_{proc}(i)) - NM_{Ca}}{Swap\_in_{Th}} \tag{11}$$

After T_{Catch}, the situation becomes the same as discussed in previous section.

In addition, if T_{catch} is larger than $\sum_{i=1}^{n} T_{proc}(i)$ (n indicates the number of layers), it means processing can not catch up with data transfer, the whole training can keep 100% utilization.

Now we can get the new utilization:

$$Util = \begin{cases} \dfrac{Num\_buffer(l)/Proc_{Th}(l)}{max(Num\_buffer(l)/Proc_{Th}(l), Num\_buffer(l-1)/Swap\_in_{Th})}, \\ 1, \qquad T < \dfrac{\sum_{i=1}^{Ca_{layers}} (Proc_{th}(i) \times T_{proc}(i)) - NM_{Ca}}{Swap\_in_{Th}} \end{cases} \tag{12}$$

3.6 OOC-Paleo

Base on the two strategy above, we can involve data transfer into original Paleo model to calculate the training time when using out-of-core methods for large deep learning workloads. The computing time will be calculated by original model and the swapping time will be calculated by $Swap\_in/out_{Th}$ and the data size for each layer to be transferred. Finally we could calculate the total training time by the specific chosen strategy.

4. Experiments and Case Study

4.1 Hardware Platform

The following experiments are all performed on DGX-1. The details of the environments are shown in the Table 2.

Table 2: Environment Information

GPU	Tesla V100 (Volta)
GPU memory capacity	16 GB
CPU	Intel(R) Xeon(R) CPU E5-2698 v4
CPU memory capacity	512 GB
CPU-GPU interconnect	PCI-Express gen3 x16
CPU-GPU bandwidth	16 GB/sec
OS	Ubuntu 16.04.5 LTS
CUDA	CUDA 10.1
cuDNN	cuDNN 7.5

4.2 Case Study: IBM optimized Chainer

IBM have an optimized Chainer version that support out-of-

core method. The strategy they used is vDNN-like one. One can simply switch on/off the out-of-core by set a flag named OOC, however, because of the strategy, if you set OOC while the workload actually can run within GPU memory, there will be performance loss.

But we can make use of this feature to find out the relationship between swapping time and computing time. We instrument the code to get the time elapsed from beginning of each layer to the beginning of next layer as $T_j - T_{j-1}$ in our model (results shown in Appendix A). And then tested same workload in both switch on and off OOC flag. Figure 5 shows the results of total running time under two situation.

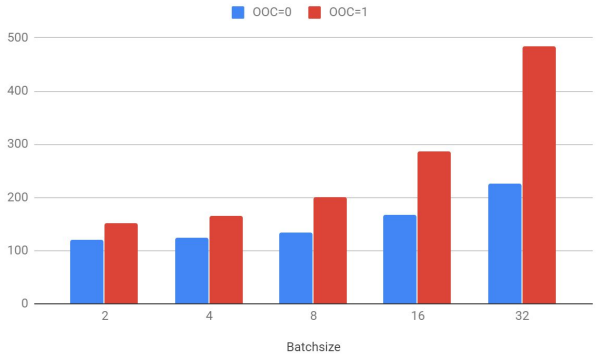

Fig. 5: The total running time for 1 iteration under OOC on/off.

According to the results, we find that even with small batch size, the elapsed time with out-of-core method will be larger than normal method. As the input size growing, the swapping time will be much more longer than computing time. Thus we can conclude that the $Swap\_in_{Th}$ is the bottleneck of this out-of-core method.

4.3 Case Study: UM-Chainer

We implemented a out-of-core version of Chainer[11] framework using UM. In this case, we used a swap/prefetch strategy similar to OC-Caffe[7]. In our implementation, we used $cudaMemPrefetchAsync()$ as data transfer method. because we don't swap data out explicitly, the strategy is very similar to Capacity based strategy.

One of the advantages that using UM rather than data swapping is this method can almost keep the same performance when workloads can be run within GPU memory. In addition, data swap method will still have Out of Memory (OOM) problem when input size is too big.

However, according to our micro benchmark, we find that the throughput of UM prefetch is not so good as swapping pinned memory that used in IBM version.

The result of training performance (images/s) of different batch-size is shown in Figure7. As the batch-size grows, eventually the out-of-core effects starts to appear between [32,64]. The performance begin to decreasing after the memory footprint exceeds the GPU memory. As discussed in last section, the data moving time should already be larger than computing time when out-of-core happened, but the performance doesn't drop suddenly, it is because in Capacity based strategy, there would be

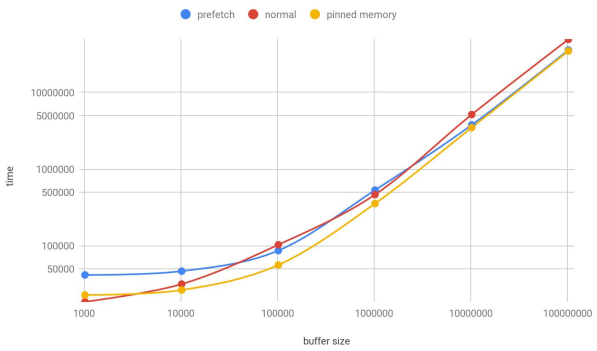

Fig. 6: Transfer time under different buffer size, although prefetch will be faster than normal memory copy, the latency is relatively big. Pinned memory copy is faster than prefetch, yet for very large buffer size, their speed is almost the same.

a stage that *Util* can still keep 100% at the beginning of backward. However, as the input size growing, the duration will become shorter and shorter. And finally the performance converges to a constant value, the performance becomes limited by the data swapping throughput.

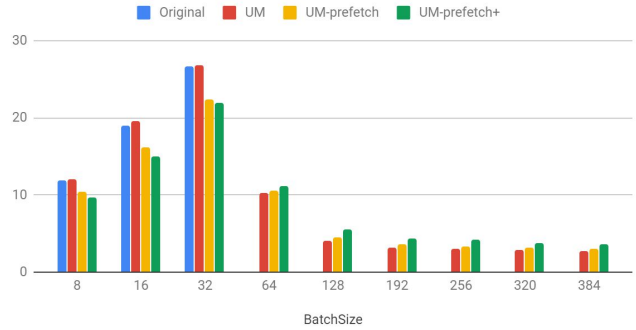

Fig. 7: Test results of UM-Chainer, UM version is implemented by directly using Unified Memory, UM-prefetch/prefetch+ is version that applied prefetch strategies.

On the other hand, although we have applied Memory Prefetch in this case, the speed up is only by about 35%, the main bottleneck for out-of-core training may not be the swap-in/out strategy but the swapping throughput.

4.4 OOC-Paleo simulation

Further more, we can use our OOC-Paleo to simulation the performance as we enlarger the throughput, when the total time stopped decrease, I will find what exactly throughput we need for the out-of-core methods.

Take VGG16[12] as an example, we gradually add the throughput of data transfer and Figure 8 shows the result of experiment.

This result also shows that the throughput now is the bottleneck for out-of-core methods, since the total time will decrease apparently once the throughput become larger.

5. Discussion

From these two case studies, we can find that:1.The data transfer is not faster enough even when the buffer is not very large,

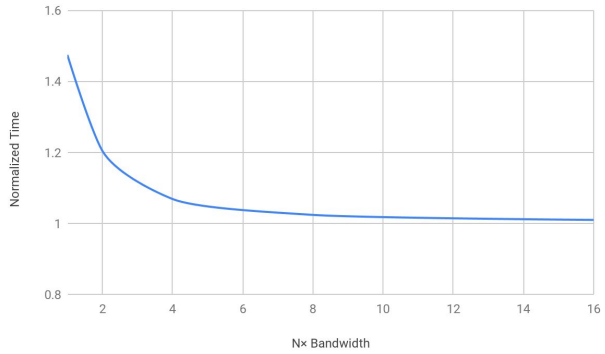

Fig. 8: The Result of Simulation, as throughput growing, the normalized time gradually converged to 1.

2.Even though you have a good enough strategy, the speed will still slow down a lot due to the data transfer is slow. Then the capacity will not be so important as latency and throughput of the data transfer. We can not change the latency since it is decided by the rule of physics, yet we could buy more throughput. It is true that we can simply buy larger memory GPU to avoid out-of-core situation, but it also takes much more money than adding connection throughput.

On the other hand, other than buying more throughput, we can also choose to use memory that have lower throughout. According to our analysis, data transfer will be affected by several factors,in equation 5, usually the $Conn_{Th}$ will be the lowest (usually the connection would be PCI), which means it is not necessary to use high throughput memory, which can save money for the whole system.

6. Conclusion

In this paper we did a detailed analysis about the behavior of out-of-core performance in terms of both single layer and the whole model. In addition, we proposed OOC-Paleo, a performance model for estimate the running time of out-of-core method. In our case study, we also implemented UM-Chainer framework to enable Chainer training with Unified Memory for workload that exceeded GPU memory limitation. According to out case study, the main bottle neck of out-of-core method is not the data transfer strategy but the throughput of data transfer. In addition, our study also shows that when considering system for out-of-core workloads, we can move budget from memory to throughput in order to get a better performance.

7. Future work

In paper SuperNeurons[13], they discussed a method to reduce memory footprint, which is called recomputing. With recomputing method, we can just discard some of the layers' data(neither keep in NM nor in FM), when it is needed, just do Forward again to get them. This add to the computing complex while reduced the memory footprint, in other word, less data to swap-in. It's a good trade off in the out-of-core case since the overhead of swap-in will usually become very big as the buffer size/batch-size grow. However, the method makes the performance model even more complex and the schedule become hard to predict. In future work we would like to add re-computation method to our performance model.

References

[1] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a kronecker factored eigenbasis. In *Advances in Neural Information Processing Systems*, pages 9550–9560, 2018.

[2] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 12359–12367, 2019.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[4] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[5] Hang Zhang, Kristin Dana, Jianping Shi, Zhongyue Zhang, Xiaogang Wang, Ambrish Tyagi, and Amit Agrawal. Context encoding for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[6] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[7] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, Xiaoyi Lu, and Dhabaleswar K Panda. Oc-dnn: Exploiting advanced unified memory capabilities in cuda 9 and volta gpus for out-of-core dnn training. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 143–152. IEEE, 2018.

[8] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 18. IEEE Press, 2016.

[9] SB Shriram, Anshuj Garg, and Purushottam Kulkarni. Dynamic memory management for gpu-based training of deep neural networks.

[10] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. 2016.

[11] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015.

[12] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[13] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: dynamic gpu memory management for training deep neural networks. In *ACM SIGPLAN Notices*, volume 53, pages 41–53. ACM, 2018.

Appendix
Layer wise results in section 4.2

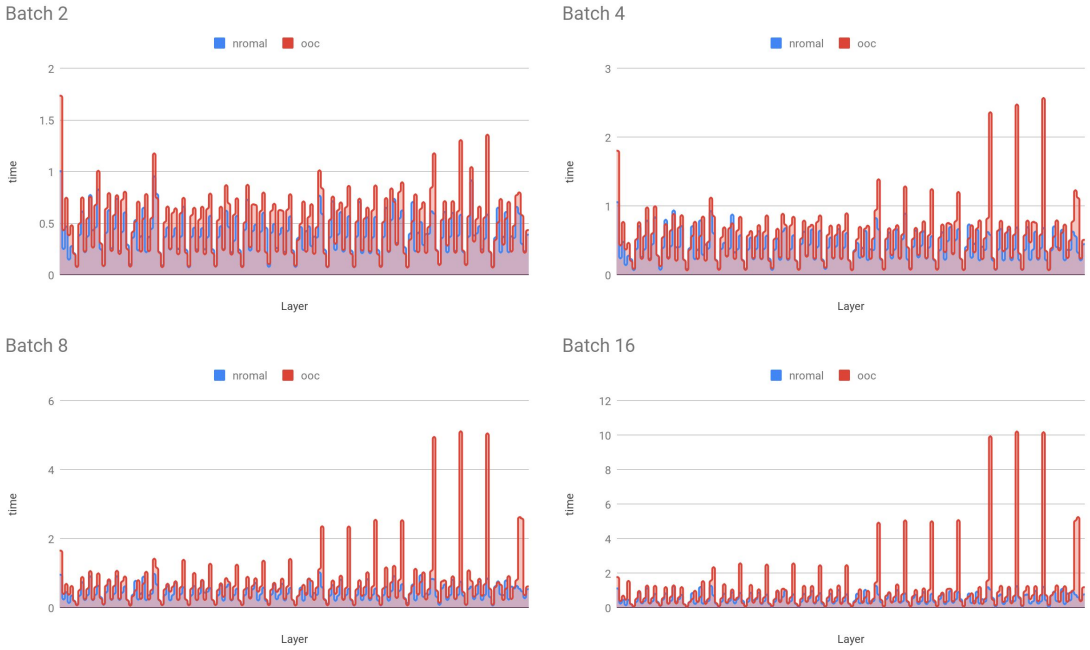

Fig. A·1: The layer wise processing time for ResNet50 with different batchsize during backward. Even though with only batchsize 2, the OOC version still take more time than normal version in most of the layer. In addition, data transfer time of some layer grows dramatically as batchsize growing.