OUT-OF-CORE TRAINING FOR EXTREMELY LARGE-SCALE NEURAL NETWORKS WITH ADAPTIVE WINDOW-BASED SCHEDULING

Anonymous authors
Paper under double-blind review

ABSTRACT

While large neural networks demonstrate higher performance in various tasks, training large networks is difficult due to limitations on GPU memory size. We propose a novel out-of-core algorithm that enables faster training of extremely large-scale neural networks with sizes larger than allotted GPU memory. Under a given memory budget constraint, our scheduling algorithm locally adapts the timing of memory transfers according to memory usage of each function, which improves overlap between computation and memory transfers. Additionally, we apply virtual addressing technique, commonly performed in OS, to training of neural networks with out-of-core execution, which drastically reduces the amount of memory fragmentation caused by frequent memory transfers. With our proposed algorithm, we successfully train ResNet-50 with 1440 batch-size with keeping training speed at 55%, which is 7.5x larger than the upper bound of physical memory. It also outperforms a previous state-of-the-art substantially, i.e. it trains a 1.55x larger network than state-of-the-art with faster execution. Moreover, we experimentally show that our approach is also scalable for various types of networks.

1 INTRODUCTION

Deep Neural Networks (DNNs) achieve outstanding results in various tasks. In particular, it has been demonstrated that larger neural networks outperform smaller ones. For example, on image classification tasks, He et al. (2016) shows ResNet with 1k-layers achieves accuracy improvement by 2% over ResNet-110 without any changes except the number of layers. Likewise, larger models achieve better performances on natural language processing (Devlin et al., 2018; Brown et al., 2020) and image generation (Wang et al., 2018b; Brock et al., 2019). Supported by these evidences, making model larger is one promising way to improve the model performance and realize brand-new systems in the deep learning research and development.

Despite the high demand for large models, the GPU memory size is limited. For instance, NVIDIA A100, one of the latest GPU devices, has only 40GB as its memory. Such limitation on memory inevitably places an upper bound on the scope with which deep learning researchers and developers design architectures of neural networks. It also limits the capacity of neural networks to perform better on existing tasks or deploy richer amount of data that are not tractable on current GPU limitations, such as 4K videos, 3D contents, and so on.

One possible way to address the limitations on GPU memory size is "out-of-core execution". This method utilizes CPU memory as a temporary cache for the GPU computation. Since neural networks, especially feed-forward networks, can be executed layer by layer sequentially, we can transfer data from GPU to CPU memory when the variables are not necessary at the current computation. In fact, the CPU memory size is much larger than GPU memory, e.g., larger than 1TB. Thus, using CPU memory as a cache for GPU memory, we can virtually extend the size of GPU memory, as if it has memory larger than 1TB.

As a naive strategy to realize out-of-core execution, we can transfer memory between GPU and CPU before and after every layer execution. While this approach can execute the maximum size of model on limited memory budget, this approach puts GPU computation on hold at every layer until the end

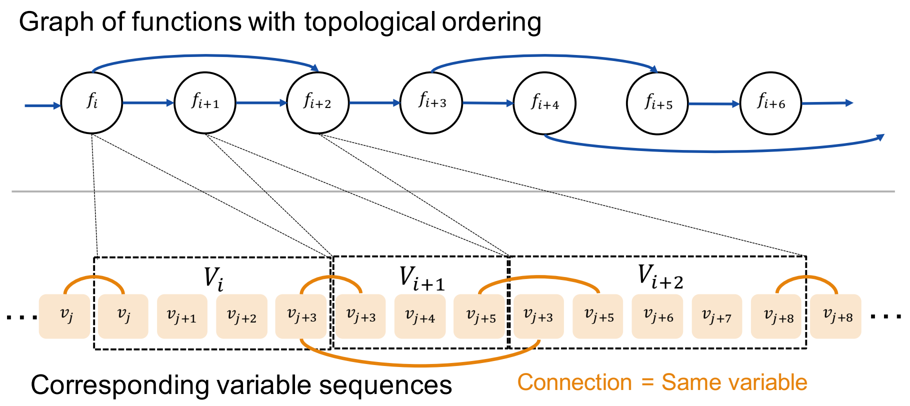


Figure 1: (Top) Directed graph of functions in topological order. Any feed-forward neural networks can be considered as a sequence of functions and can be executed from f_1 to f_n in order. (Bottom) A sequence of variables corresponding to the sequence of functions. Each function uses variable internally and different functions can have the same variable (connection between variables). We estimate distances between functions by sum of bytes of variables within two functions (e.g. we estimate the distance between f_i and f_{i+3} as the sum of bytes of $\{V_{i+1}, V_{i+2}\}$ in this picture).

of corresponding memory transfers. On the other hand, if we place too many variables on GPU to accelerate computation, we can execute only models with limited size. Therefore, it is necessary to find a better memory transfer algorithm that enables execution of larger models without sacrificing computational time.

Many existing works attempted to achieve a faster execution with models larger than allotted GPU memory in the setting with out-of-core execution (Rhu et al., 2016; Jin et al., 2018; Wang et al., 2018a). Recently, Le et al. (2019) formulated out-of-core execution as inserting memory transfer operations in the graph on TensorFlow (Abadi et al., 2015). This method can be applied to arbitrary architecture of neural network, and they successfully enable a faster execution with larger model than previous works. However, they only focus on the fixed distance represented by edges on the graph and do not consider memory usage of each function and variable. It thus results in significant overhead, especially as the model size grows. Moreover, to the best of our knowledge, none of the previous methods tackled a memory fragmentation problem caused by frequent memory transfers. The memory fragmentation wastes the limited memory budget on GPU and limits the scalability of model size.

In this paper, we introduce a novel memory transfer scheduling algorithm for the training of extremely large-scale neural networks. Our key contributions are summarized as follows:

- We propose a novel cost-aware memory swapping scheduling algorithm that is simple yet can find a faster schedule. Our model is the first to schedule memory transfers for training neural networks with locally adaptive distance by modeling the memory usage, which can find a faster schedule than existing methods.

- We introduce a novel memory allocation strategy to reduce memory fragmentation caused by memory swapping. Based on a virtual addressing technique that is mainly used in OS, our memory allocation strategy suppresses memory fragmentation and improve the trainable model size.

- Through experiments, we validate our proposed method performs well in terms of both training time and model size. On ResNet50, we show our algorithm clearly outperforms a previous work. Additionally, we evaluate our method on the various networks for image recognition, semantic segmentation, and image generation to show our approach can be applied to various tasks.

2 PRELIMINARIES

Feed-forward neural networks can in general be represented as a directed acyclic graph (DAG) G(V, E) of functions (Fig.1 (Top)), where V is a set of function $\{f_i\}_{i=1}^N$ and there is an edge from f_i to f_j if f_j uses an output of f_i as an input. Through topological ordering, we can index all functions in ascending order from inputs to outputs of entire network such that if there is an edge from i to j, then we always have $i < j$. Thus, we can simply consider a given network as a sequence of functions $[f_1, \dots, f_n]$ and can train the network by executing functions from f_1 to f_n in order. Out-of-core execution on DNNs can utilize this fact of function order as a strong assumption to find a better schedule for memory swapping. Namely, we can select which memory we should transfer from GPU to CPU (*Swap-out*) and decide when we should start transferring a memory from CPU to GPU (*Swap-in*) on the given topological order of functions.

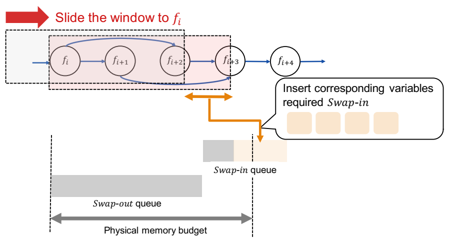
Some early works focus on forward and backward process of DNNs (Rhu et al., 2016; Wang et al., 2018a; Jin et al., 2018). They employ a simple memory swap scheduling where they perform *Swap-out* for the outputs of forward functions and *Swap-in* at corresponding backward functions. While these methods enable the training of larger models than the setting without out-of-core execution, there are two drawbacks on these methods. First, when a variable is used multiple times in a forward pass, they perform *Swap-out* only at the last usage, which means the data of the variable is kept on GPU memory until the last usage during the forward computation (even for backward). This is problematic when the network is very large and it has, for example, skip connections which are seen in currently popular models such as DenseNet (Huang et al., 2017) and U-Net like architectures (Ronneberger et al., 2015). Second, they do not carefully handle the overhead caused by *Swap-in*. In order to make data for a variable ready on GPU memory before starting computation, they use a naive heuristic where they trigger a *Swap-in* operation for a variable used in a function right before the previous function of that (e.g. trigger *Swap-in* for f_i at f_{i-1}). It causes large overhead if f_{i-1}'s computation is too small compared to the memory transfer latency for *Swap-in*, i.e. the function has to wait for the completion of the memory transfer.

Le et al. (2019) recently proposed *Large Model Support* (LMS) as a module on TensorFlow, in which they consider a distance on a graph (DAG) of a neural network to determine the memory swap scheduling which handles the two issues described above. They introduce two hyperparameters: *swapout_threshold* and *swapin_ahead*. They selectively trigger *Swap-out* on a variable used at a function f_i when $d(f_i, f_j) \leq$ *swapout_threshold* where $d(\cdot, \cdot)$ is a distance on the graph between two vertices and f_j is a closest function which uses the variable used in f_i ($i < j$). This allows triggering *Swap-out* at any point of the entire graph computation including forward and backward, and enables training larger models with skip connections. Then, they trigger *Swap-in* for the swapped out variable right before executing a function f_k where $d(f_k, f_j) =$ *swapin_ahead* and $k < j$. Consequently, they successfully train ResNet50 with 4.9 times larger batch-size and with less overhead, which has not been achieved by the previous methods.

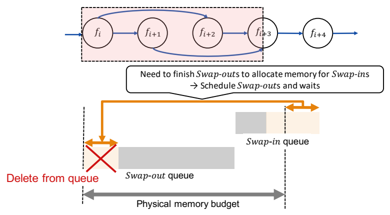
While Le et al. (2019) achieves promising results, there still remain the following two issues: 1) They rely on a fixed distance across the entire graph to determine the memory swap scheduling, while an actual cost (latency) for computation and memory transfer between two vertices varies depending on operations and data sizes appeared between them, which may lead undesirable overhead due to the gap between the cost of computation and memory transfers. 2) Both of two hyperparameters mentioned above balance a trade-off between reducing memory transfer overhead and GPU memory usage. Increasing either of the hyperparameters may result in out-of-memory error. Finding good hyperparameters with less overhead and without raising memory error is difficult for humans. Therefore they introduce an automatic tuning mechanism which relies on a fairly complicated simulation based on memory profiling and computational cost estimation for each operator manually defined by humans.

3 SCHEDULING MEMORY SWAPPING BASED ON LOCALLY-ADAPTIVE WINDOW

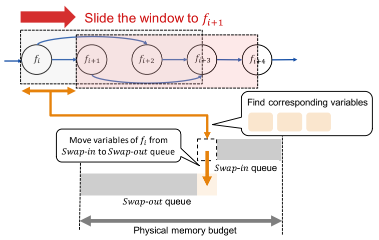
To overcome the aforementioned issues, we propose a novel memory transfer scheduling algorithm which adaptively determines distance thresholds for the *Swap-in* and *Swap-out* operations while considering memory budget limitation. Aside from a sequence of functions $[f_1, \dots, f_N]$ which we



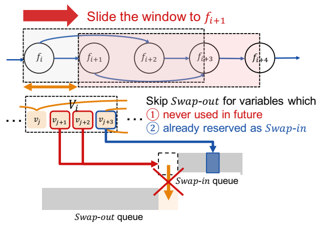
(a) Schedule *Swap-in*



(b) Determine *Swap-out*


Slide the window to f_{i+1}
f_i
f_{i+1}
f_{i+2}
f_{i+3}
f_{i+4}
Find corresponding variables
Move variables of f_i from *Swap-in* to *Swap-out* queue
Swap-in queue
Swap-out queue
Physical memory budget

(c) Schedule *Swap-out*



(d) Skip scheduling

Figure 2: An overview of our scheduling algorithm at function f_i. Red rectangle shows *schedule-window*. Sliding this window from f_1 to f_n, our algorithm performs (a) → (b) → (c) at each function to schedule *Swap-in* and *Swap-out*. (a) Schedule *Swap-in* for all new variables coming into the window. (b) If scheduling exceeds physical memory budget by scheduling *Swap-in*, we allocate enough memory for *Swap-in* by determining *Swap-out* for previous variables. Specifically, *Swap-out* is reserved right after the previous function using the variable, and waits for the *Swap-out* right before f_i. (c) Schedule *Swap-out* for variables used by f_i and move the window to the first variable of next function. (d) shows how we reduce unnecessary memory transfers. We skip *Swap-out* for the variables used by f_i if variables are never used in future or already reserved as *Swap-in*.

call *function-sequence* later, we introduce a sequence of all variables used in order in the *function-sequence*, which we call *variable-sequence*. Let $V = \{v_j\}_{j=1}^{N_v}$ be a set of all variables used on a given network and $\hat{V}_i \subseteq V$ be a subset of variables used by f_i. Given a topological order of functions $[f_1, \dots, f_n]$, we can also define a topological sequence of variables (i.e., *variable-sequence*) $\boldsymbol{v} = flatten([\hat{V}_1, \dots, \hat{V}_n])$ where $flatten()$ represents flatten function (representing given matrix as 1-dimensional array). Since multiple functions can use the same variable either as input or output, the same variable can appear multiple times in this sequence (Fig.1 (Bottom)). Each variable has its size in bytes, and a state representing whether it is on GPU or CPU now. We denote the size in bytes of variable v_j as b_{v_j}, and state of variable $\sigma(v_j) \in \{0, 1\}$, which returns 1 if v_j is placed on CPU, otherwise 0.

Given a *variable-sequence*, our goal is to find a better schedule for which variables we should apply *Swap-in* and *Swap-out* at each function. Unfortunately, finding the optimal schedule for *Swap-in* and *Swap-out* on $\boldsymbol{v}$ is difficult, due to the huge search space. Let $V_i^{'}$ be a set of variables which never appears from function f_i, and n be the number of functions. At each function, we must consider which variables are on GPU. Therefore, the entire search space can be represented as $O(\prod_{i=1}^{n} 2^{|V \setminus V_i^{'}|})$. This is not tractable, since modern neural networks have more than a hundred functions and variables.

In this paper, we employ a simple local greedy search in *variable-sequence* which requires only a single hyperparameter *schedule-window*. At first, let us focus on when to trigger *Swap-in* for variables swapped out previously. It is most ideal that we trigger *Swap-in* such that the computation starts immediately after the completion of *Swap-in* memory transfer. If we trigger *Swap-in* too early, it unnecessarily consumes more memory, while more overhead if too late. In order to determine this ideal timing, we have to precisely estimate computation time and memory transfer time, which is non-trivial. Instead, we use a very simple assumption where both computation and memory transfer time are roughly proportional to the data size involved with them. We trigger *Swap-in* operations at a function f_i by looking-ahead *variable-sequence* until accumulated variable size in bytes exceeds the threshold *schedule-window*. More formally, we trigger *Swap-in* for variables in a sub-sequence $\boldsymbol{v}[l : r]$ where l is the index of the first variable of $\hat{V}_i$ on $\boldsymbol{v}$, and r is the maximum index which satisfies $\sum_{i=l}^{r} b_{\boldsymbol{v}_i} \leq$ *schedule-window*.

Fig.2 shows an overview of our scheduling procedure at f_i. Let the sub-sequences determined by *schedule-window* at f_{i-1}, f_i, and f_{i+1} be $\boldsymbol{v}[l^- : r^-]$, $\boldsymbol{v}[l : r]$, and $\boldsymbol{v}[l^+ : r^+]$, respectively. Our scheduling procedure at f_i comprises following three steps: First, we schedule *Swap-in* for variables $\{v_{si} \in \boldsymbol{v}[r^- + 1 : r] \mid \sigma(v_{si}) = 1\}$ (Fig.2a). Second, we determine the completion of previously scheduled *Swap-out* (oldest) such that the total variable size of *Swap-in* and *Swap-out* doesn't exceed the physical GPU memory budget (Fig.2b). These memory transfers are scheduled to be performed before executing f_i. Finally, we reserve *Swap-out* for $\hat{V}_i$, which is the same as $\{v_{so} \in \boldsymbol{v}[l : l^+ - 1]\}$, after executing f_i and move *schedule-window* to the next function f_{i+1} (Fig.2c). To reduce redundant memory transfers, we skip schedules for variables if the same variables are already scheduled (Fig.2d).

Compared to Le et al. (2019), our algorithm determines both *Swap-in* and *Swap-out* scheduling according to only a single hyperparameter with a physical memory budget constraint. Also, note that our algorithm adaptively controls the *Swap-in* distance threshold at each function in *function-sequence* in order to consider computation and memory transfer time for efficient scheduling, while we simply use a fixed-size window in bytes in *variable-sequence*.

4 Reducing memory fragmentation with virtual addressing

Most deep learning frameworks (e.g. Tensorflow (Abadi et al., 2015), PyTorch (Paszke et al., 2019), and Neural Network Libraries) commonly utilize the "best-fit" algorithm with caching GPU memory as a memory allocation strategy. This approach utilizes memory space effectively by reusing previously allocated memory for multiple variables. It performs well when the number of reusing the same memory space is limited. However, in the setting with out-of-core execution, frequent memory transfers between CPU and GPU result in reusing the same memory beyond the number of times manageable by the best-fit algorithm. Thus, the memory allocation system will divide its cached memory repeatedly, which results in severe memory fragmentation. This fragmentation is generally known as External Fragmentation in the field of OS memory allocation. It is also important to tackle this memory fragmentation problem to maximize trainable model size under fixed memory budget.

Since estimating exact amount of external fragmentation is difficult, we estimate it as the worst case. Let a requested size of bytes be m_r. In the worst case, when this m_r cannot be used for any successive variables and also cannot be merged with consecutive memories, this m_r memory is kept in cache without being used during the training. Thus, the memory request of m_r bythe would waste m_r bytes in the worst case. When we grow model size with out-of-core execution, m_r could be in order of GBs.

To tackle this external memory fragmentation, we apply virtual addressing (VA), which is commonly used in memory management on OS, to the training of DNNs. Namely, we manage both physical and virtual memory addresses on neural network libraries. When the memory allocation is requested, we map small physical memory chunks with constant size to a consecutive virtual address. Once a variable is cleared during the training of neural network, we release a virtual address and cache physical memories for the future requests. Since physical memories could be combined in arbitrary order, virtual addressing never suffers from external fragmentation on physical memory address. In consequence, we can simply estimate the amount of fragmentation in virtual addressing by the difference between requested memory size and the size of allocated virtual address for this request (this memory fragmentation is known as internal fragmentation in OS).

Let m_c be a size of physical memory chunk. To minimize the amount of wasted memory by internal fragmentation, we allocate virtual address for the request as $m_a = km_c$ where $k = \lceil \frac{m_r}{m_c} \rceil$. In this case, we can estimate the amount of internal fragmentation (IF) for a single allocation request as:

$$IF = m_a - m_r < m_c. \tag{1}$$

Batch size	64	128	190	256	512	928	1120	1248	1440
Baseline	235	255	258	x	x	x	x	x	x
LMS (Le et al., 2019)	235	255	241	236	209	137	x	x	x
Baseline	336	492	581	x	x	x	x	x	x
Schedule	320	439	533	559	457	400	x	x	x
Schedule + VA	304	389	415	424	446	409	398	347	321

Table 1: Training time and model size comparison between Le's method and ours. We use images / sec (ips) for all values on the table, where x indicates Out-of-Memory and thus cannot be trained. The top 2 rows show Le's results and bottom 3 rows show ours. Since computational environments are different (P100 vs V100), we show the results of baseline for both methods separately. All results for Le et al. (2019) are from the best values reported on the paper.

It is notable that the upper-bound of IF doesn't depend on m_r. Hence, the maximum size of internal fragmentation during training (IF_{max}) can be bounded by

$$IF_{max} \quad < \quad N_{max}m_c, \tag{2}$$

where N_{max} is the maximum number of variables that can be used on GPU memory simultaneously. In general, this is suitable for the training of extremely large-scale neural networks under out-of-core execution. In the out-of-core execution, we perform *swap-out* and only variables within *Schedule-window* are placed on GPU. Therefore, N_{max} becomes smaller as model size becomes larger, and IF_{max} also approaches a small value.

Obviously, it is better to use smaller physical memory chunk to reduce internal fragmentation. However, there is a restriction on device. On CUDA Driver API, the minimum size of physical memory chunk is defined depending on the type of GPU. In our environment, the minimum size of physical memory chunk is 2MB. Besides, using smaller physical memory chunks causes additional overhead for virtual address mapping. We experimentally decide the size of physical memory as 40MB, which balances well between the cost of allocation and the amount of memory fragmentation.

5 EXPERIMENTS

All experiments were conducted on an IBM POWER9 machine with 594GB of CPU RAM and NVIDIA Tesla V100 GPUs (each GPU has 16GB of memory). Note that we only used single GPU for all experiments. CPU and GPU are connected by two NVLinks (each can transfer memory at 50GB/s). We employ Neural Network Libraries[1] (NNL) as our deep learning framework under CUDA Toolkit 10.2 and cuDNN v8.0.2.

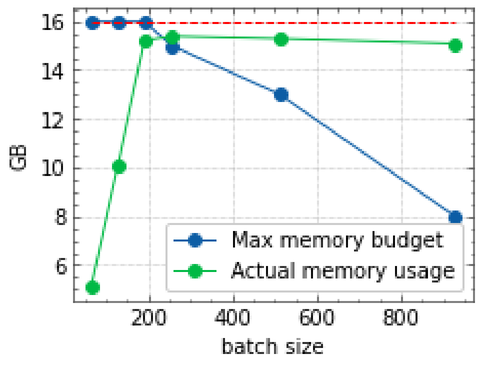
In all experiments, we evaluate both scheduling alone and scheduling with virtual addressing described in Section 4. To evaluate scheduling alone, we use the default memory allocator on NNL, which applies "best-fit" algorithm with caching memory in nearly identical manner as other deep learning frameworks, such as Torch or TensorFlow.

5.1 IMAGENET CLASSIFICATION

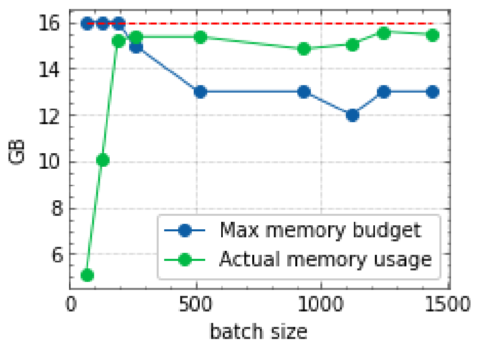
We first compare our method with Le's algorithm on ResNet-50 training on ImageNet (Deng et al., 2009). We keep image size as 224×224 and increase only batch size, which is the same condition reported by Le et al. (2019). Both methods are examined on the same GPU memory budget (16GB). We examined our method on NVIDIA Tesla V100, while Le et al. (2019) uses P100. Thus, all computations are theoretically faster on our environment. This however implies a more challenging criterion for our model, as will be described below.

Table.1 shows comparison between Le's algorithm and ours in terms of training time and trainable model size. We first validate our scheduling without VA. Our scheduling not only successfully

[1]https://github.com/sony/nnabla



(a) Maximum defined memory budget and actual memory usage in the setting with scheduling alone.



(b) Maximum defined memory budget and actual memory usage in the setting with scheduling with VA.

Figure 3: Maximum defined memory budget and actual memory usage. In the setting with scheduling alone, there are almost 2× differences at 928 batch size. This indicates memory fragmentation. On the other hand, scheduling with VA prevents such fragmentation.

trains the maximum size of model which Le's approach can train (batch size = 928), but also finds a faster schedule than Le's approach. Our schedule can keep training speed at 68% at 928 batch size, while Le's method performs at 53%. It is noteworthy that comparing results with absolute value of ips is also reasonable, because in out-of-core execution, our primary interest is how much memory transfers we can overlap with computation. Since the model size and memory budget are the same on both methods, faster computational environment means that we have to transfer the same amount of memory within shorter time. Even with such disadvantage, our schedule clearly outperforms Le's method in terms of absolute ips.

While our method successfully finds faster schedule, scheduling alone cannot train larger model than 928 batch size as well as Le's method. Fig.3 shows maximum memory budget we can define for scheduling (blue line) and actual memory usage (green line). Red line shows physical memory budget. Fig.3a shows the results in the setting with scheduling alone. At 928 batch size, network consumes almost 16GB while we set 8GB as a memory budget for scheduling. In fact, if we set larger than 8GB for 928 batch size, training process causes Out-of-Memory even though a schedule is found. This difference between the memory budget we set and actual usage indicates the memory fragmentation and this is the reason why we cannot train a model with batch size lager than 928. On the other hand, using our novel VA allocator with scheduling, we successfully train the same model with 1,444 batch size, which is 1.6× larger than previous limitation with small overhead. Moreover, our scheduling with VA enables faster training at batch size of 928 against scheduling alone. This is because we can set much more physical memory budget for scheduling. As we can see on Fig.3b, the difference between the memory budget we can set for scheduling and actual usage is smaller, even when we increase batch size. This indicates that VA drastically reduces fragmentation and we can conclude that VA is beneficial for out-of-core execution, especially for training extremely large-scale DNNs. Note that if we employ smaller m_c, this difference becomes smaller and we can train much larger models, but the overhead in terms of training time becomes larger due to the cost of virtual addressing.

5.2 Training time and model size on various neural network architectures

We examined our proposed method over various network architectures in terms of training time and memory usage as the model size grows. Followings are the list of network architectures we examined: DenseNet (Huang et al., 2017), Pix2PixHD (Wang et al., 2018b), and DeepLabv3+ (Chen et al., 2018). For all networks, we increased batch size with fixed image size and all computations are executed with float32 precision. As training dataset for each model, we use ImageNet with 224×224 as image size for DenseNet, Cityscapes dataset (Cordts et al., 2016) with 512×1024 for Pix2PixHD, and PASCAL VOC dataset (Everingham et al., 2010) with 513×513 for DeepLabv3+.

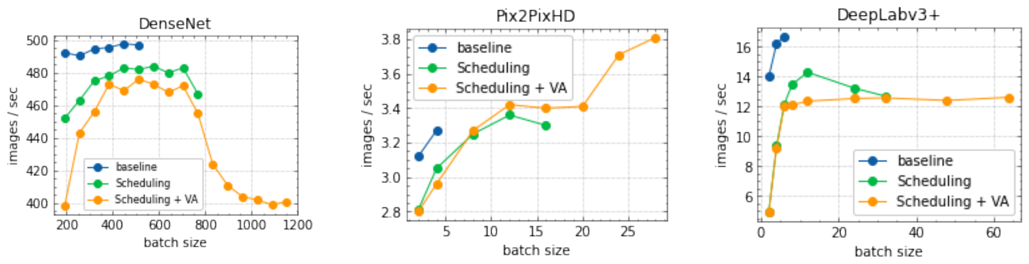


Figure 4: Training time as model size grows for various architectures. We evaluate average iteration time including forward, backward, and update over 100 iterations.

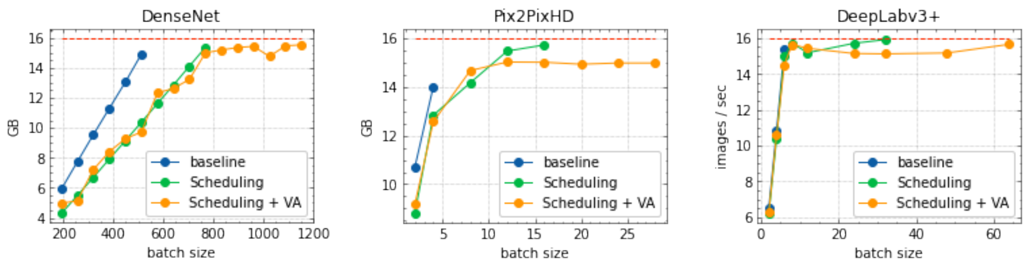


Figure 5: Memory usage as model size grows for various architectures. We evaluate the peak memory usage during forward, backward, and update for each network. Red dashed line represents physical memory budget (16GB).

Fig.4 shows training time against batch-size on 3 models. The horizontal axis represents batch size, and the vertical axis represents ips for a single training step, including forward, backward and update. In the baseline setting (blue line), the maximum batch sizes trained on 16GB memory for DenseNet, Pix2PixHD, and DeepLabv3+ are 512, 4, and 6, respectively. With our scheduling (green line), we can successfully train the models that are 1.5x, 4x, and 5.3x larger than baseline setting. However, as we can see in Fig.5, training with scheduling alone reaches the physical memory limit because of memory fragmentation. Applying proposed VA with our scheduling (orange line), trainable batch size clearly improves with small overhead for all networks. Compared to the setting with scheduling alone, our scheduling with VA can train 1.5x to 2x larger models. Fig.5 shows that we can keep the memory usage almost constant with VA regardless of batch size. It is notable that in pix2pixHD we achieve performance gain compared to baseline setting. We consider that larger batch size is computationally beneficial, especially for slower networks (Pix2PixHD achieves only around 3 images / sec).

6 Conclusion

In this paper, We propose a novel out-of-core algorithm that enables faster training of extremely large-scale neural networks with sizes larger than allotted GPU memory. Under a given memory budget constraint, our scheduling algorithm locally adapts the timing of memory transfers according to memory usage of each function, which improves overlap between computation and memory transfers. Additionally, we apply virtual addressing technique, commonly performed in OS, to training of neural networks with out-of-core execution, which drastically reduces the amount of memory fragmentation caused by frequent memory transfers. Beyond GPU memory limitation, we empirically show that our proposed method enables training of much larger networks than existing methods, without sacrificing the training time. While our proposed algorithm clearly demonstrates improvements over previous models, it is still not optimal, e.g., we ignore the order of variables for *Swap-out*. We leave further optimization as future work.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale GAN training for high fidelity natural image synthesis. In *International Conference on Learning Representations*, 2019.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 801–818, 2018.

Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *Int. J. Comput. Vision*, 88(2):303–338, June 2010. ISSN 0920-5691. doi: 10.1007/s11263-009-0275-4.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pp. 630–645. Springer, 2016.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. *ACM Trans. Archit. Code Optim.*, 15(3), September 2018. ISSN 1544-3566. doi: 10.1145/3243904.

Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. Automatic gpu memory management for large neural models in tensorflow. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2019, pp. 1–13, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367226. doi: 10.1145/3315573.3329984.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8026–8037. Curran Associates, Inc., 2019.

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*, MICRO-49. IEEE Press, 2016.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi (eds.), *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pp. 234–241, Cham, 2015. Springer International Publishing. ISBN 978-3-319-24574-4.

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. *SIGPLAN Not.*, 53(1):41–53, February 2018a. ISSN 0362-1340. doi: 10.1145/3200691.3178491.

Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018b.