Training Deeper Models by GPU Memory Optimization on TensorFlow

Chen Meng [1], Minmin Sun [2], Jun Yang [1], Minghui Qiu [2], Yang Gu [1]
[1] Alibaba Group, Beijing, China
[2] Alibaba Group, Hangzhou, China
{mc119496, minmin.smm, muzhuo.yj, minghui.qmh, gy104353}@alibaba-inc.com

Abstract

With the advent of big data, easy-to-get GPGPU and progresses in neural network modeling techniques, training deep learning model on GPU becomes a popular choice. However, due to the inherent complexity of deep learning models and the limited memory resources on modern GPUs, training deep models is still a non-trivial task, especially when the model size is too big for a single GPU. In this paper, we propose a general dataflow-graph based GPU memory optimization strategy, i.e.,"swap-out/in", to utilize host memory as a bigger memory pool to overcome the limitation of GPU memory. Meanwhile, to optimize the memory-consuming sequence-to-sequence (Seq2Seq) models, dedicated optimization strategies are also proposed. These strategies are integrated into TensorFlow seamlessly without accuracy loss. In the extensive experiments, significant memory usage reductions are observed. The max training batch size can be increased by 2 to 30 times given a fixed model and system configuration.

1 Introduction

Recently deep learning plays an increasingly important role in various applications [1][2][3][4][5]. The essential logic of training deep learning models involves parallel linear algebra calculation which is suitable for GPU. However, due to physical constraints, GPU usually has lesser device memory than host memory. The latest high-end NVIDIA GPU P100 is equipped with 12–16 GB device memory, while a CPU server has 128GB host memory. On the contrary, the trend for deep learning models is to have a "deeper and wider" architecture. For example, ResNet [6] consists of up to 1001 neuron layers and a Neural Machine Translation(NMT) model consists of 8 layers using attention mechanism [7][8], and most of layers in NMT model are sequential ones unrolling horizontally which brings non-neglectable memory consumption.

In short, the gap between limited GPU device memory capacity and increasing model complexity makes memory optimization a necessary requirement. In the following, the major constituents of memory usage for deep learning training process are presented.

Feature maps. For deep learning models, feature map is the intermediate output of one layer generated in the forward pass and required for gradients calculation during the backward phase. Figure 1 shows the curve of the ResNet-50's memory footprint for one mini-batch training iteration on ImageNet dataset. The max value of the curve gradually emerges with the accumulation of feature maps. The size of feature map is typically determined by batch size and model architecture(for CNN the stride size and output channel number, for RNN the gate number, time-step length and hidden size). The feature map no longer needed will be de-allocated, which results in the declining of the curve. For complex model training, users have to adjust batch size or even redesign their model architectures to work around "Out of Memory" issue. Although with model parallelism [9], one training task could be split onto multiple devices to alleviate this problem, this brings additional communication overhead. And the bandwidth limitation across devices[1] may slow down the training process significantly.

Weights. Compared with feature maps, weights occupied a relatively small proportion of memory usage [11]. In this paper, weights are treated as persistent memory resident in GPU memory that can not be released until the whole training task is finished.

Temporary memory. A number of operations need additional memory for some algorithms such as Fast-Fourier-Transform (FFT) based convolution. It is temporary and will be released inner the operation. The size of temporary memory can be auto-tuned by enumerating each algorithm in the GPU software libraries such as cuDNN [12], so it can be ignored.

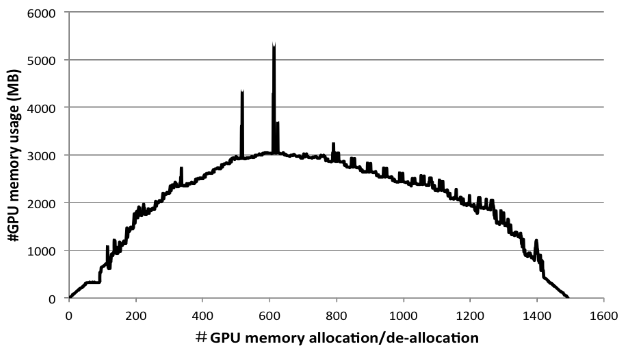

Figure 1: Varying curve of ResNet-50's memory footprint through one training step. The horizontal axis is the number of allocation/de-allocation times and the vertical axis corresponds to current total bytes of memory footprint.

As clearly feature maps are the main constitute of GPU memory usage, we focus on the feature maps to propose two approaches to resolve GPU memory limitation issues, i.e.,"swap-out/in" and memory-efficient Attention layer for Seq2Seq models. All these optimizations are based on TensorFlow [13]. TensorFlow has its built-in memory allocator that implements a "best-fit with coalescing" algorithm. The design goal of this allocator is to support de-fragmentation via coalescing. However, its built-in memory management strategy hasn't taken any special consideration for memory optimization for training big models.

In a nutshell, we summarize our contributions as follows. Focusing on the feature maps, two approaches to reduce memory consumption of training deep neural networks are proposed in this paper. The dataflow-graph based "swap-out/in" utilizes host memory as a bigger memory pool to relax the limitation of GPU memory, and memory-efficient Attention layer for optimizing the memory-consuming Seq2Seq models. The implementation for these approaches are integrated into TensorFlow in a seamless way and can be applied transparently to all kinds of models without requiring any changes to existing model descriptions.

The rest of this paper starts with related work. Then our approaches are described, finally followed by experiments and conclusion.

2 Related Work

To reduce memory consumption in single-GPU training, there are some existing ideas and work:

Leveraging host RAM as a backup to extend GPU device memory. CUDA 8.0 enables Unified Memory with Page Migration Engine so that unified Memory is not limited by the size of GPU memory. However, our tests show that it can bring a severe performance loss(maximum ten times degradation). And another way is using a run-time memory manager to virtualize the memory usage

[1]maximum of 32GB/s for PCIe 3.0*16 [10], while a maximum of 1.25GB/s for 10-gigabit Ethernet

by offloading the output of each layer and prefetching it when necessary [11], which can only be applied to the layer-wise CNN models, not to sequence models.

Using re-computation to trade computation for memory consumption. It is already used by frameworks such as MXNet [14]. MXNet uses a static memory allocation strategy prior to the actual computation. While TensorFlow uses a dynamic memory allocation strategy in which each allocation and de-allocation is triggered in the runtime, so this strategy can not be migrated directly to TensorFlow. And Memory-efficient RNN is proposed by [15]. However, for those sequence models with attention mechanism, the attention layer actually requires much more memory space than LSTM/GRU.

There are also some other optimization methods, such as Memory-Efficient DenseNets [16] and Memory-Efficient Convolution [17] with significantly reducing memory consumption. However, Memory-Efficient DenseNets is applicable for special cases, and Memory-Efficient Convolution reduces the temporary memory in the convolution computation while the temporary memory can be ignored compared with the feature maps.

In this paper, a general based approach "swap-out/in" is proposed, which is targeted for any kind of neural network. To pursue more memory optimizations for Seq2Seq models, memory-efficient attention algorithm are designed. The implementation of these approaches is integrated into TensorFlow seamlessly by formulating the optimization as a graph rewriting problem and can be applied transparently to all kinds of models on TensorFlow without any changes to model descriptions.

3 Our Approaches

In this section, we begin by introducing our "swap out/in" method and then present our optimized Seq2Seq models.

3.1 Swap out/in: Rewriting Dataflow Graph

TensorFlow uses an unified **dataflow graph** to represent a model training task. As shown in Figure 2, nodes(*Relu_fwd*, etc.) represent computation. Edges (*Relu*, etc.) carry tensors (arrays or dependencies) between nodes. Every node is scheduled to be executed by the executor. This graph can be viewed as an intermediate representation of a training task, so optimization over the graph is general and transparent for models.

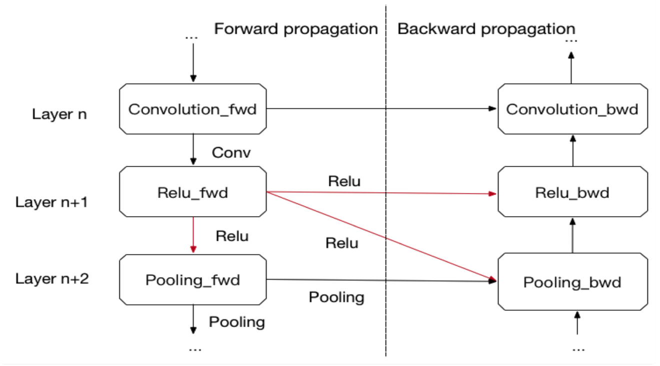

Figure 2: Reference count.

TensorFlow uses a dynamic strategy for its memory management. The essential idea of this strategy is the timing of the allocation and de-allocation of tensors. During the runtime, a tensor is not allocated until when the executor starts to execute the corresponding node, and its de-allocation is triggered automatically when its **reference count** decreases to 0. In Figure 2, reference count of *Relu* is 3 since it is referenced by three nodes. After *Relu_bwd* is completed, *Relu*'s reference count becomes 0 and then it is released. In short, the life cycle of *Relu* lasts from when *Relu_fwd* starts to run till *Relu_bwd* finishes. As a result, all feature maps generated in the forward phase gradually accumulate until the loss-function layer finishes, thus resulting in the max memory footprint (Figure 1) which limits the trained model complexity under the available memory budget.

Focus on the point stated above, **Swap out/in** is proposed in this paper, which rewrites the dataflow graph by cutting off the reference edges between feature maps and backward nodes, and meanwhile, inserting swap-out/in related nodes to ensure the logical equivalence. Note that those inserted nodes are placed on "CPU" device. So before the actual execution, TensorFlow transparently inserts the communication node for those tensors across devices. As shown in Figure 3, *tensor b* can be released immediately after it is transferred to CPU memory. And when it is reused by *node e*, it can be transferred back to GPU memory. So in this graph-based optimization Swap out/in, there are two essential problems:

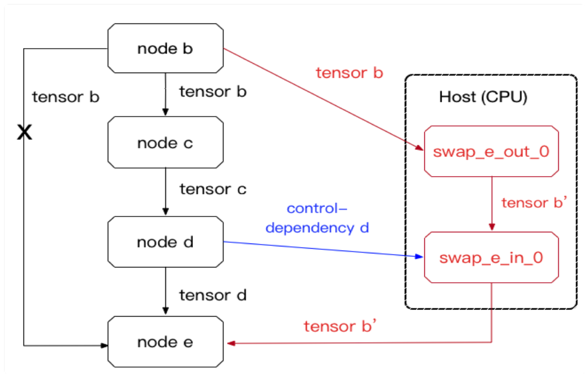

Figure 3: Atomic operation of the swap out/in optimization. Remove the reference edge from *node e* to *node b*, and insert nodes and edges in red or blue.

3.1.1 Which feature maps to be swapped out?

A straightforward strategy is swapping all the feature maps, which achieves the best memory optimization. However, this brings significant performance degradation due to frequent data transfer. So some of the feature maps can still be resident in GPU memory for less overhead. With taking a predefined memory threshold as input, a candidate list of feature maps can be obtained by heuristically searching over forward sub-graph. The searching is presented in a reverse topological-order because the *chain rule* [18] derives that the longer *the critical path* between a node and *loss-function node* is, the longer the life-cycle of the feature map is. Combining this with the fact that the swap out/in brings overhead of data transfer between CPU and GPU, it is appropriate to allocate GPU memory budget to those with short life-cycle and swap those with long life-cycle, which is helpful to overlap the communication time with the computations.

3.1.2 When to be swapped back in?

Swap-in too early makes memory space occupied by non-active feature maps while too late block the downstream computation. So feature maps should be swapped in as late as possible until data transfer cannot be overlapped with the computation. The **control dependency** from **trigger node** in Figure 3 is used to control the timing swap-in node begin running. Estimated completion-times of all nodes are used for choosing the proper trigger node. Completion time means when each node is completed. It can be estimated by static scheduling the graph using a critical path of weighted Directed Acyclic Graph (DAG) algorithm. The weights in DAG is actually the execution time of each node that can be obtained by pre-run or static complexity analysis.

In summary, swap out/in can optimize the graph with a given memory threshold. The optimized graph is logically equivalent to the original graph and can be run on TensorFlow directly.

3.2 Memory Efficient Seq2Seq Model

To further exploit the potential of memory optimization for Seq2Seq models[19] [20], a dedicated effort is still necessary. Usually, **Seq2Seq** models consist of encoder, decoder, and attention layers.

Encoder and decoder are RNN(LSTM/GRU) in most cases. Actually **attention layer** requires much more memory space than LSTM/GRU. In this section, we propose a memory-efficient additive attention layer [7] since the additive attention out-performs the multiplicative attention [21], and it is adopted in our in-house Seq2Seq models.

3.2.1 Memory Efficient Attention Layer.

Firstly, a quantitative analysis is taken to figure out why Attention layer requires the huge amount of memory space. Assuming the time steps of RNN are the same, they are denoted as T. Also, denote hidden/output size as H, and batch size as B. Then the memory required by RNNs is O($T * B * H$). All the encoder outputs are denoted as $Keys$, and the decoder output of previous time steps is denoted as $Query$. During computing the Attention $Scores$, there is an intermediate result with the same size of $Keys$, that is required by the gradient back-propagation, at each decoder time step. As a result, the memory required for each decoder time step in Attention layer is O($T * B * H$), resulting into the total memory complexity O($T^2 * B * H$).

Although the Attention intermediate result consumes the most amount of memory, the computation cost is extremely cheap - just an addition between $Keys$ and $Query$. If the intermediate results are dropped, their memory is saved with only a little extra re-compute time during backward phase. This optimization will reduce the total memory cost from O($T^2 * B * H$) to O($T * B * H$).

As shown in Fig 4, in the operator with memory optimization, the intermediate result is removed, and an extra addition is added in the gradient operator to re-compute the intermediate result from Keys and Query. Note that this optimization avoids writing the intermediate result to the memory, which saves memory access and makes the Attention computation even faster.

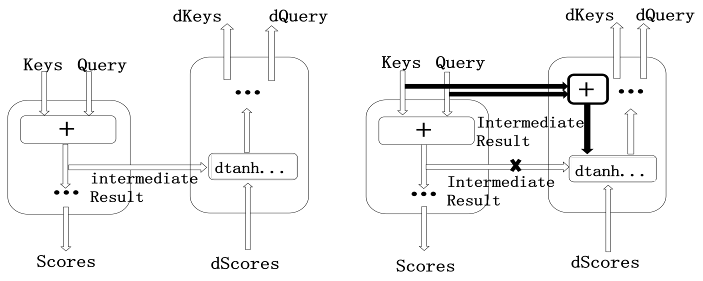

Figure 4: Optimization on Attention operation. $d*$ means the gradients. The left side is without memory optimization and the right side is with memory optimization.

4 Experiment Results

In the following experiments, our proposed methods are applied on general CNN models and several sequence models. Under the 12GB GPU memory limit of Tesla M40, we compare the optimized versions against baselines. The baselines directly run on the official version TensorFlow r1.2.

4.1 Experiment Setup

Our proposed optimization approaches are evaluated on general models such as ResNet, Inception-V3, GAN [22] [23], Language Model(TF-LM[2]), and NMT (TF-NMT[3]). Datasets used here include ImageNet and WMT[4] and PTB[5] for sequence models.

We evaluate the memory optimization effect by comparing the max batch size and the max layers that can be trained on TensorFlow. B_{base} indicates the max batch size without optimization and B_{opt} indicates the max batch size after applying memory optimization. M_{base} indicates the max memory usage without optimization and M_{opt} indicates the max memory usage after applying optimization.

[2]https://github.com/tensorflow/models/tree/master/tutorials/rnn/ptb

[3]https://github.com/tensorflow/models/tree/master/tutorials/rnn/translate

[4]http://www.statmt.org/wmt15/translation-task.html

[5]https://catalog.ldc.upenn.edu/LDC99T42

4.2 Swap out/in Results

Table 1(a) presents that, running the optimized graph, the max batch size can be increased by up to 4 times. And the benefit on NMT model is inferior to CNN models, which also demonstrate sequence models need to be further optimized. Specifically, when using 32 batch size and set ImageNet as dataset, Table 1(b) presents that, it is only possible to train a ResNet-200 model ("OOM" means "Out of Memory"). And after swap out/in is applied, ResNet-1001 and even ResNet-2000 can be trained successfully.

Table 1: Evaluation of Swap out/in. GPU memory limit is 12GB

(a) General Models.

Model	B_{base}	B_{opt}
ResNet-50	144	664(+361%)
Inception-V3	208	548(+163%)
GAN	24	48(+100%)
NMT	496	824(+66%)

(b) ResNet.

Model	M_{base}	M_{opt}
ResNet-101	5815MB	2660MB
ResNet-200	10662MB	3052MB
ResNet-1001	OOM	5979MB
ResNet-2000	OOM	10650MB

4.3 Sequence Models Results

We evaluate the memory-efficient Attention layer proposed in this paper on TF-NMT model(Table 2(b)) and memory-efficient LSTM/GRU on TF-LM(Table 2(a)), it can be observed that the memory saving aligns with the quantitative analysis that memory saving of the Attention operator is from O($T^2 * B * H$) to O($T * B * H$).

Table 2: Evaluation of memory-efficient sequence models.

(a) TF-LM model.

LSTM Layers	B_{base}	B_{opt}
1	1800	3000(+67%)
4	750	1500(+100%)
8	350	900(+157%)
16	75	280(+273%)

(b) TF-NMT model.

Time Steps	B_{base}	B_{opt}
50	350	1100(+214%)
100	90	550(+511%)
200	20	230(+1050%)
400	2	60(+2900%)

4.4 Impact on Training Speed

We also measure the runtime cost of each optimization method seperatively. Because each feature map has to be transferred between GPU device and CPU twice in the runtime, the swap out/in optimized training lead to less than 10% performance loss compared to the baselines. The memory-efficient Attention layer makes the NMT slightly faster by up to 9% performance improvement because it saves memory accesses.

5 Conclusion

In this paper, we propose two approaches to reduce the memory consumption of training deep neural networks. The dataflow-graph based Swap out/in method is employed to utilize host memory as a bigger memory pool to relax the limitation of GPU memory, which is targeted for any kind of neural network. We also design and implement dedicated optimization to save huge amount of memory for Seq2Seq models with attention mechanism without affecting the training speed. The implementation for these approaches are integrated into TensorFlow seamlessly without requiring any changes to existing model descriptions. The Results demonstrate these methods outperform the baseline with significant GPU memory consumption savings.

References

[1] A Graves and J Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks the Official Journal of the International Neural Network Society*, 18(5-6):602, 2005.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *International Conference on Neural Information Processing Systems*, pages 1097–1105, 2012.

[3] Claudiu Ciresan Dan, Ueli Meier, Luca Maria Gambardella, and Juergen Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *Corr*, 22(12):3207 – 3220, 2010.

[4] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[5] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*, 2015.

[8] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. In *Conferenceon Empirical Methods in Natural Language Processing (EMNLP)*, 2015.

[9] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, and Paul Tucker. Large scale distributed deep networks. In *International Conference on Neural Information Processing Systems*, pages 1223–1231, 2012.

[10] Jasmin Ajanovic. Pci express 3.0 overview. In *Proceedings of Hot Chip: A Symposium on High Performance Chips*, 2009.

[11] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. Virtualizing deep neural networks for memory-efficient neural network design. *arXiv e-prints abs/1602.08124*, 2016.

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[13] 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. Software available from tensorflow.org.

[14] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[15] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In *Advances in Neural Information Processing Systems*, pages 4125–4133, 2016.

[16] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q Weinberger. Memory-efficient implementation of densenets. *arXiv preprint arXiv:1707.06990*, 2017.

[17] Minsik Cho and Daniel Brand. MEC: Memory-efficient convolution for deep neural network. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 815–824, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[18] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[19] Chen-Yu Lee and Simon Osindero. Recursive recurrent nets with attention modeling for ocr in the wild. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2231–2239, 2016.

[20] Baoguang Shi, Xiang Bai, and Cong Yao. An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition. *IEEE transactions on pattern analysis and machine intelligence*, 2016.

[21] Denny Britz, Anna Goldie, Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures. *arXiv preprint arXiv:1703.03906*, 2017.

[22] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[23] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.