Reducing the Memory Cost of Training Convolutional Neural Networks by CPU Offloading

Tristan Hascoet*†
Kobe Univeristy
tristan@people.kobe-u.ac.jp

Weihao Zhuang*
Kobe Univeristy
zhuangweihao1996@gmail.com

Quenti Febvre
Sicara
quentin.febvre@gmail.com

Yasuo Ariki
Kobe Univeristy
ariki@kobe-u.ac.jp

Tetsuya Takiguchi
Kobe Univeristy
takigu@kobe-u.ac.jp

Abstract

In recent years Convolutional Neural Networks (CNN) have enabled unprecedented progress on a wide range of computer vision tasks. However, training large CNNs is a resource intensive task that requires specialized Graphical Processing Units (GPU) and highly optimized hardware-specific implementations for efficient computation. In training CNNs on modern systems, GPU memory is a major bottleneck limiting both the size of the input and the model architectures being considered: The backpropagation algorithm requires to accumulate the activation values of hidden layers in live GPU memory during the forward pass as these activations are needed for the computation of the weights gradient during the backward pass of the backpropagation algorithm. In this paper, we propose to alleviate this memory bottleneck by by offloading the input activation of hidden layers to the CPU: during the forward pass, we transfer input activations to the CPU upon computation to free up GPU memory, and transfer these activations back to the GPU when needed by the backward pass. On a typical work station with a NVIDIA GTX 2080, we are able reduce the memory consumption of traning VGG by xxx% with a minimal additional wall time overhead of xxx%. Our method is orthogonal to other techniques for memory reduction such as quantization and sparsification so that it can be easily combined.

*Equal contribution
†Corresponding author

1 Introduction

Over the last few years, Convolutional Neural Networks (CNN) have enabled unprecedented progress on a wide array of computer vision tasks. One disadvantage of these approaches is their resource consumption: Training deep models within a reasonable amount of time requires special Graphical Processing Units (GPU) with numerous cores and large memory capacity. Given the practical importance of these models, a lot of research effort has been directed towards algorithmic and hardware innovations to improve their resource efficiency such as low-precision arithmetic [**?**], network pruning for inference [**?**], or efficient stochastic optimization algorithms [**?**].

In this paper, we focus on a particular aspect of resource efficiency: optimizing the memory cost of training CNNs. Given the ubiquity of CNN for practical computer vision applications, optimizing the memory consumption of CNN training has the potential to impact a wide range of applications. Here, we only present a few of the most interesting potential impacts of such optimization:

Low-memory GPUs: Training large CNN requires special GPUs with large memory capacity. Typical desktop GPUs memory capacity is too small for training large CNNs. As a result, getting into deep learning research comes with the barrier cost of either buying specialized hardware or renting live instances from cloud service providers, while standard laptop GPUs remain idle untapped resources. Reducing the memory cost of deep model training allows training deep nets on standard graphic cards without the need for specialized hardware, effectively removing this barrier cost.

Research in optimization: Recent works on stochastic optimization algorithms have highlighted the benefits of large batch training [**?**, **?**, **?**, **?**].For example, in Imagenet, linear speed-ups in training have been observed with increasing batch sizes up to tens of thousands of samples [**?**]. Optimizing the memory cost of CNN training may allow further research on the optimization trade-offs of large batch training. Very large batch training on small datasets like MNIST and CIFAR10 is computationally inefficient with current stochastic optimization algorithms [**?**]. However, for such small datasets, memory optimization would allow to process the full dataset in one pass through the networks. The ability to process the full dataset in one pass allows to easily train CNNs on the true gradient of the error. Hence, memory optimization techniques opens the door for research on gradient descent optimization of neural networks outside the realm of Stochastic Gradient Descent.

There is an inherent trade-off between the memory consumption and

computation wall time of the CNN training procedure: Previous approaches in optimizing the memory consumption of training CNN, including gradient checkpointing and reversible network architectures, trade off memory consumption for additional computations by recomputing all or a subset of the hidden layers activations during the backward pass.

Instead, our approach reduce the memory consumption without introducing any additional computation by leveraging an under-utilized resource: the CPU memory. We propose to temporarily offloading GPU memory buffers to the CPU during the forward pass of the computation, and transfering these memory buffers back into GPU memory as needed by the backward pass of the backpropagation algorithm to compute the gradients.

They key challenge in our approach is to efficiently overlap the computation and data transfers between CPU and GPU to minimize the overhead in wall time introduced by these data transfer. We describe an efficient implementation of this approach that allows us to reduce by up to XXX % the memory cost of training a VGG network with a minimal wall time overhead of XXX. We compare the memory vs. wall time trade off of our approach to gradient checkpointing to illustrate the efficiency of our approach.

The remainer of this paper is organized as follows: In Section 2, we briefly review related work. Section 3 introduces the preliminary notions necessary to understand the root of the GPU memory bottleneck. Our approach is described in detail in Section 4, and Section 5 presents the results of our evaluation.

2 Related Work

Research into resource optimization of CNNs covers a wide array of techniques, most of which are orthogonal to our work. We briefly present some of these works:

On the architectural side, Squeezenet [**?**] was first proposed as an efficient neural architecture reducing the number of model parameters while maintaining high classification accuracy. MobileNet [**?**] uses depth-wise separable convolutions to further reduce the computational cost of inference for embedded device applications.

Network pruning [**?**] is a set of techniques developed to decrease the model weight size and computational complexity. Network pruning works by removing the network weights that contribute the least to the model output. Pruning deep models has been shown to drastically reduce the memory cost and computational cost of inference without significantly hurting model

accuracy. Although pruning has been concerned with optimization of the resource inference, the recently proposed lottery ticket hypothesis [**?**] has shown that specifically pruned networks could be trained from scratch to high accuracy. This may be an interesting and complementary line of work to investigate in the future to reduce training memory costs.

Low precision arithmetic has been proposed as a mean to reduce both memory consumption and computation time of deep learning models. Mixed precision training [**?**] combines float16 with float32 operations to avoid numerical instabilities due to either overflow or underflow. For inference, integer quantization [**?**, **?**] has been shown to drastically improve the computation and memory efficiency and has been successfully deployed on both edge devices and data centers. Integrating mixed-precision training to our proposed architecture would allow us to further reduce training memory costs.

Most related to our work, gradient checkpointing was introduced as a mean to reduce the memory cost of deep neural network training. Gradient checkpointing, first introduced in [**?**], trades off memory for computational complexity by storing only a subset of the activations during the forward pass. During the backward pass, missing activations are recomputed from the stored activations as needed by the backpropagation algorithm. Follow-up work [**?**] has since built on the original gradient checkpointing algorithm to improve this memory/computation trade-off.

In contrast, our approach does not induce any additional computation: Instead of computing a set of missing hidden activations during the backward pass, we propose to offload the hidden activations to the CPU during the forward pass, and to transfer these activations back to GPU memory during the backward pass.

Reversible models [**?**] constrain the CNN architecture to feature invertible transformations. This allows the activation values of lower layers to be reconstructed from those of higher layers during the backward pass. Reversible networks have been shown to offer a better memory/computation trade-off than gradient checkpointing at the cost of constraining the CNN architecture.

Our best performing model combines revertible operations with CPU offloading: we use the invertible BN-Leaky ReLu block design proposed in [**?**] to efficiently deal with normalization and non-linearity layers, and offload to CPU the activations of the pooling and convolution layers.

3 Preliminaries

Let us consider a model F of N sequential layers trained to minimize an error e defined by a loss function $\mathcal{L}$ for an input x and ground-truth label $\bar{y}$:

$$F : x \to y \tag{1a}$$
$$y = f_N \circ ... \circ f_2 \circ f_1(x) \tag{1b}$$
$$e = \mathcal{L}(y, \bar{y}) \tag{1c}$$

During the forward pass, each layer f_i takes as input the activations z_{i-1} from the previous layer and outputs activation features $z_i = f_i(z_{i-1})$, with $z_0 = x$ and $z_N = y$ being the input and output of the network respectively. During the backward pass, the gradient of the loss with respect to the hidden activations are propagated backward through the layers of the networks using the chain rule as:

$$\frac{\delta \mathcal{L}}{\delta z_{i-1}} = \frac{\delta \mathcal{L}}{\delta z_i} \times \frac{\delta z_i}{\delta z_{i-1}} \tag{2}$$

Before propagating the loss gradient with respect to its input to the previous layer, each parameterized layer computes the gradient of the loss with respect to its parameters. In vanilla SGD, for a given learning rate η, the weight gradients are subsequently used to update the weight values as:

$$\frac{\delta \mathcal{L}}{\delta \theta_i} = \frac{\delta \mathcal{L}}{\delta z_i} \times \frac{\delta z_i}{\delta \theta_i} \tag{3a}$$
$$\theta_i \leftarrow \theta_i - \eta \times \frac{\delta \mathcal{L}}{\delta \theta_i} \tag{3b}$$

For most layers, the computation of either gradients are functions of the layer's input activations z_{i-1}: For example, convolution layers need the values of input activations to compute the weight gradients:

$$\frac{\delta \mathcal{L}}{\delta \theta_i} = z_{i-1} \star \frac{\delta \mathcal{L}}{\delta z_i} \tag{4}$$

while Rectified Linear Unit layers need the input activations values to compute the gradients of the loss with respect to its imputs:

$$\frac{\delta \mathcal{L}}{\delta z_{i-1}^j} = \begin{cases} \frac{\delta \mathcal{L}}{\delta z_{i-1}^j}, & \text{if } z_{i-1}^j \geq 0 \\ 0 & \text{if } z_{i-1}^j < 0 \end{cases} \tag{5}$$

Hence, backpropagation implementations in deep learning frameworks store hidden layers activations in GPU memory upon computation during the forward pass. Activations accumulate in live memory buffers throughout the full forward pass until used for gradients computations in the backward pass. Once the gradients computed in the backward pass, their associated hidden activation buffers can be freed from live memory. However, the accumulation of activation values stored within each layer along the forward pass creates a major bottleneck in GPU memory. In the next Section, we detail our approach to alleviate this memory bottleneck.

4 Propose Method

4.1 Framework

The input activations of each layer are kept in GPU memory only to be used for the computation of the layer weight gradients during the backward pass. Hence, the activations of lower layers are kept idle in GPU memory during the full time span of the forward and backward computations through higher layers. We propose to offload these activations to the CPU during this idle time in order to free up some memory space for the computation of higher layers activations.

Figure XXX illustrates our approach. During the forward pass (top), activations are computed forward through the network layers. Instead of keeping these activations idle in GPU memory, activation values are transfered to the CPU memory immediatly after their computation. In the backward pass (bottom), gradients are backpropagated backward through the networ layers following equations XXX. Our implementation synchronizes the transfer of the layers input activations back to GPU right before they are needed for their layer's gradient computation. Hence the key challenge in our implementation consists in synchronizing the data transfers with the computations so that only the minimal amount of activation values is loaded in GPU memory at any given time, while the least amount of time is spent waiting for the data transfer.

To achieve this goal, we propoze optimization along two axes: The first consists in optimizing the data transfer speed between CPU and GPU memory, using efficient memory access and data compression schemes. The second consists in efficient parallelization to maximally overlap the computations with the data transfer. The following subsections details optimizations along these two axes.

4.2 Parallelization

Figure XXX illustrates the execution through time of a forward and backward pass through a toy network with and without parallelization of the data transfer. Without parallelization, computation and data transfers are performed sequentially so that the total wall time is given by the sum of the computation and data transfer time. $\mathcal{T}_{total} = \mathcal{T}_{comp} + \mathcal{T}_{data}$ Parallelization aims to overlap the computation and data transfer so that the total wall time is given by $\mathcal{T}_{total} = \mathcal{T}_{comp} + \mathcal{T}_{idle}$, where $\mathcal{T}_{idle}$ represents synchronization delays in cases where the computation is stopped to await for the required data transfer to complete.

The key challenge in this parallelization scheme consists in efficiently managaing memory allocation and data transfer so as to minimize the time $\mathcal{T}_{idle}$ spent awaiting for data transfer. In this paper, we adopt a simple parallelization strategy: During the forward pass, activations z_i are transfered to CPU upon computation by layer i as $z_i = f_i(z_{i-1})$, and the GPU memory buffered are freed as the next layer computation completes as $z_{i+1} = f_{i+1}(z_i)$.

Algorithm 1: Forward procedure through layer i with parallel CPU offloading. Double arrows indicate the asynchronous execution of a CUDA directive within a stream. Data transfers are executed within dedicated CUDA stream and CPU thread to synchronize the memory deallocation without blocking the execution of upward layers

Data: Layer f_i, input activation z_{i-1}
CPU pinned memory buffer P_{i-1}
CPU thread T_{data}
CUDA events E^i_{data}, E^i_{comp}
CUDA Streams S_{data}, S_{comp}

Result: z_i

Allocate(z_i);
$S_{comp} \Leftarrow z_i \leftarrow f_i(z_{i-1})$;
$S_{comp} \Leftarrow E^i_{comp}$;

In Thread T_{data}:
$S_{data} \Leftarrow P_{i-1} \leftarrow z_{i-1}$;
$S_{data} \Leftarrow E^i_{data}$;
Wait(E^i_{data}, E^i_{comp});
Free(z_{i-1});

One important exception to this rule concerns skip connections, as illustrated in Figure XXX. Skip connections induce a delay in the GPU buffer dealocation as the input to residual blocks must be kept in memory until the end of the residual block computation. In the experiment section, we will show that this delay slightly degrades the memory consumption of CPU offloading for residual networks compared to sequential architectures like VGG.

During the backward pass, the input activation z_{i-1} to layer i must be transferred back into GPU memory before the backward gradient computations to avoid idle time in the GPU. Hence, we synchronize the data transfer of z_{i-1} with the beginning of the backpropagation through the upward layer f_{i+1}.

Algorithm 2: Backward procedure through layer i with parallel CPU offloading.

Data: Layer f_i, input activation z_{i-1}
CPU pinned memory buffer P_{i-1}
CUDA events E^i_{data}, E^{i+1}_{data}
CUDA Streams S_{data}, S_{comp}

Result: $\frac{\delta \mathcal{L}}{\delta z_{i-1}}$, $\frac{\delta \mathcal{L}}{\delta \theta_i}$

$Free(z_i)$;
$Allocate(z_{i-1})$;
$S_{data} \Leftarrow z_{i-1} \leftarrow P_{i-1}$;
$S_{data} \Leftarrow E^i_{data}$;
$Wait(E^{i+1}_{data})$;
$Allocate(\frac{\delta \mathcal{L}}{\delta z_{i-1}}, \frac{\delta \mathcal{L}}{\delta \theta_i})$;
$S_{comp} \Leftarrow \frac{\delta \mathcal{L}}{\delta z_{i-1}} \leftarrow \frac{\delta \mathcal{L}}{\delta z_i} \times \frac{\delta z_i}{\delta z_{i-1}}$;
$S_{comp} \Leftarrow \frac{\delta \mathcal{L}}{\delta \theta_i} \leftarrow \frac{\delta \mathcal{L}}{\delta z_i} \times \frac{\delta z_i}{\delta \theta_i}$;

We use threading and locks to handle the synchronization on the GPU side and CUDA streams to handle the parallelization on the GPU side. Algorithm 1 and 2 provide pseudo-code for the forward and backward pass respectively.

With the above parallelization scheme, the overhead $\mathcal{T}_{idle}$ in wall time is given by the sum of the difference in the computation and data transfer time at each layer. If data transfer is faster than the computation in each layer, CPU offloading will reduce memory consumption without any overhead. On

a typical workstation with a single NVIDIA GTX 1080 Ti and PCIe lanes, we found that to be the roughly the case for convolution layers.

However, batch normalization layers and activation layers come with negligible computational cost but significant data transfer time so that offloading the input activations of such layer comes with a significant overhead. Instead of offloading these activations, we reconstruct their input activations from their output using their inverse operation as proposed in [**?**], and only offload the input activations of the convolution and pooling layers to CPU.

4.3 Data Transfer Optimization

The time needed to transfer data between devices is given by the product of the transfer speed bandwidth $\mathcal{B}$ in bits per second by the volume $\mathcal{V}$ of the data buffer in bits:

$$\mathcal{T}_{data} = \frac{\mathcal{V}}{\mathcal{B}} \tag{6}$$

In this section, we propose to minimize data transfer time by simultaneously maximizing the transfer speed $\mathcal{B}$ using pinned CPU memory buffers and minimizing the activation volume $\mathcal{V}$ by compressing data before transfer.

CPU memory can be accessed through either pageable memory or pinned memory address spaces. Pinned memory allows for Direct Memory Access (DMA), which can significantly speedup data transfer between CPU and GPU devices through PCIe lanes. However, allocating pinned memory buffers is a time consuming operation, as illustrated in Table XXX. Hence naively allocating pinned memory buffers within each forward pass would result in slow data transfer rate and slow down training, which is undesirable. Instead we propose to allocate the pinned memory buffers once at the initialization of the CNN. Our implementation scans the network architecture upon instantiation and allocates dedicated pinned memory buffers for each target layer activations (convolutions and max-pooling). During the forward pass, we transfer the GPU data directly into the pre-allocated pinned memory buffer, and, during the backward pass, we transfer the activations back to GPU from these pinned buffers. Table XXX quantifies the gains in transfer speed brought by the use of pinned memory buffers.

Another way to reduce the time of data transfer between CPU and GPU is to compress the hidden layer activations before transferring them between devices. Compression can be achieved by diverse means including sparsification, quantization or low rank factorization techniques. Low-rank factorization would induce non-negligible additional computational costs, which

Operation	Speed (MB/s)
Pinned Memory Allocation	xxx
GPU $\rightarrow$ CPU (Pageable)	xxx
GPU $\rightarrow$ CPU (Pinned)	xxx
CPU $\rightarrow$ GPU (Pageable)	xxx
CPU $\rightarrow$ GPU (Pinned)	xxx

Table 1: xxx

would undesirably slow down the computation. Sparsifying the layers activations is an interesting direction, but the affect of sparsification on training dynamics is poorly understood so we . A number of work have shown that neural networks can train to equally high accuracy with low precision arithmetic. In the experiment section, we present results obtained by reducing input activations to half precision, which allows us to halve the data transfer time without any loss in accuracy, although more agressive quantization strategies would be interesting to investigate in future work.

5 Experiments

XXX

5.1 Memory vs. Time Trade-Off

Our implementation allows to control for the memory foot-print vs. computation Wall time trade-off by either using more or less agressive parallelization schemes or by only offloading different subsets of hidden layers input activations. In this section, we explore the trade-off by controlling the latter using the parallelization scheme described in Section XXX.

Figure XXX compares the memory/Wall time trade-off of our algorithm implementation to the trade-off provided by gradient checkpointing, as implemented in the Pytorch [] library on a XXX architecture. In gradient checkpointing, this trade-off is defined by the number of checkpointed layers through the full architecture. This figure presents the computation time and memory footpring of one iteration of training the XXX architecture on a batch of XXX color images with XXX by XXX spatial resolution.

Observations XXX

5.2 Ablation study

Table XXX illustrates the speed up brought by each improvement proposed in Section XXX. We record the timings of a training iteration needed to achieve maximal memory reduction on the XXX architecture for an input batch of XXX XXX by XXX images. We start by recording the time for a baseline CPU offloading approach without parallelization nor data transfer optimization. We then gradually add the different improvement to this baseline starting from pre-allocated pinned memory buffers, half precision compression and parallelization.

Operation	Time (ms)
Baseline	xxx
Pinned Memory	xxx
Compression	xxx
Parallelization	xxx

Table 2: xxx

5.3 Architecture comparison

In this section, we compare the memory gains and time overhead achieved through CPU offloading on different popular architectures.

Architecture	Wall Time Overhead	Memory Reduction
AlexNet	xxx	xxx
VGG16	xxx	xxx
Resnet18	xxx	xxx
Resnet50	xxx	xxx
MobileNet	xxx	xxx

Table 3: xxx

Observations

6 Conclusion

Training CNNs is a resource intensive task that requires intense optimization to complete within reasonable times. To achieve this, one must harness all

resources available. In this paper, we proposed to leverage an under-utilized resource in typical CNN training pipelines: the CPU memory. We propose a simple solution to alleviate the GPU bottleneck of training CNNs