Scientific
Research
Publishing

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

Tristan Hascoet[1][*][#], Weihao Zhuang[1][#], Quenti Febvre[2], Yasuo Ariki[1], Tetsuya Takiguchi[1]

[1]Kobe University, Kobe, Japan
[2]Sicara, Paris, France
Email: *tristan@people.kobe-u.ac.jp, zhuangweihao1996@gmail.com, quentin.febvre@gmail.com,
ariki@kobe-u.ac.jp, takigu@kobe-u.ac.jp

Abstract

In recent years, Convolutional Neural Networks (CNNs) 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 implementations to get optimal performance from the hardware. GPU memory is a major bottleneck of the CNN training procedure, limiting the size of both inputs and model architectures. In this paper, we propose to alleviate this memory bottleneck by leveraging an under-utilized resource of modern systems: the device to host bandwidth. Our method, termed CPU offloading, works by transferring hidden activations to the CPU upon computation, in order to free GPU memory for upstream layer computations during the forward pass. These activations are then transferred back to the GPU as needed by the gradient computations of the backward pass. The key challenge to our method is to efficiently overlap data transfers and computations in order to minimize wall time overheads induced by the additional data transfers. On a typical work station with a Nvidia Titan X GPU, we show that our method compares favorably to gradient checkpointing as we are able to reduce the memory consumption of training a VGG19 model by 35% with a minimal additional wall time overhead of 21%. Further experiments detail the impact of the different optimization tricks we propose. Our method is orthogonal to other techniques for memory reduction such as quantization and sparsification so that they can easily be combined for further optimizations.

Keywords

Deep Learning, CNN, Optimization

[#]Equal contribution.

---

1. Introduction

Over the last few years, Convolutional Neural Networks (CNNs) [1] [2] 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 [3], network pruning [4], or efficient stochastic optimization algorithms [5].

In this paper, we focus on a particular aspect of resource efficiency: optimizing the GPU 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 [6] [7] [8]. For example, in Imagenet, linear speed-ups in training have been observed with increasing batch sizes up to tens of thousands of samples [7]. 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 [7]. However, for such small datasets, memory optimization would allow processing 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 exact gradient of the loss function. Hence, memory optimization techniques open 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: Existing approaches to optimize the memory consumption of CNN training, (gradient checkpointing, 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 reduces the GPU memory consumption without intro-

ducing any additional computation by leveraging an under-utilized resource: host-device communication. We propose to temporarily offload GPU memory buffers to the CPU during the forward pass of the computation, and transferring these memory buffers back into GPU memory as needed by the gradient computations during the backward of the backpropagation algorithm.

The key challenge in our approach is to efficiently overlap the GPU computations with the data transfers between CPU and GPU in order to minimize the overhead in wall time introduced by these data transfer. In this paper, we describe an efficient implementation of this approach that allows us to reduce by up to 35% the memory cost of training a VGG network with a minimal wall time overhead of 21%. We compare the memory vs. wall time trade-off of our approach to gradient checkpointing to illustrate the efficiency of our approach.

The remainder of this paper is organized as follows: In Section 2, we briefly review the literature for related work. Section 3 introduces the preliminary notions necessary to understand the root of the GPU memory bottleneck. Section 4 presents our approach and details the different tricks needed for efficient implementation. Finally, 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 [9] was first proposed as an efficient neural architecture reducing the number of model parameters while maintaining high classification accuracy. MobileNet [10] uses depth-wise separable convolutions to further reduce the computational cost of inference for embedded device applications.

Network pruning [4] 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 efficiently reduce the memory and computational cost of inference without significantly hurting model accuracy. Although pruning methods focus on the optimization of inference, the recently proposed lottery ticket hypothesis [11] 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 [12] combines float16 with float32 operations to avoid numerical instabilities due to either overflow or underflow. For inference, integer quantization [3] [13] 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 is first introduced in [14], 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 [15] has since built on the original gradient checkpointing algorithm to improve this memory/computation trade-off.

In contrast, our approach does not introduce 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 [16] [17] 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 approach combines revertible operations with CPU offloading: we use the invertible BN-Leaky ReLu block design proposed in [18] to efficiently deal with normalization and non-linearity layers, and only 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 $\overline{y}$:

$$F : x \rightarrow y \tag{1a}$$

$$y = f_N \circ \cdots \circ f_2 \circ f_1(x) \tag{1b}$$

$$e = \mathcal{L}(y, \overline{y}) \tag{1c}$$

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:

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

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

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

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 of the backward pass. Once the gradients computed during 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 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 GPU memory space for the computation of higher layers activations.

Figure 1 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 transferred to the CPU memory immediately after their computation. In the backward pass (bottom), gradients are backpropagated backward through the network layers following Equations (3). 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 propose a set of optimization tricks organized along two axes: The first consists in optimizing the data transfer speed between CPU and GPU memory, using efficient memory accesses 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 2 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
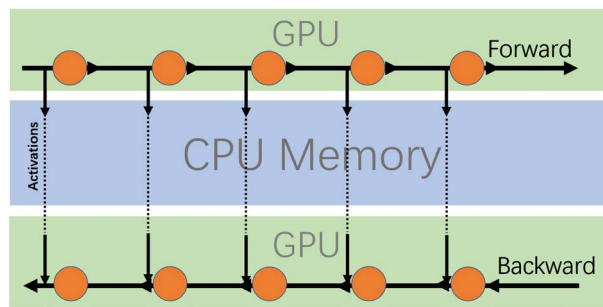

Figure 1. Illustration of CPU offloading. During the forward pass (top), hidden activation buffers are transferred to CPU memory upon computation. During the backward pass (bottom), hidden activation buffers are transferred back to GPU memory just in time to compute the weight gradients.
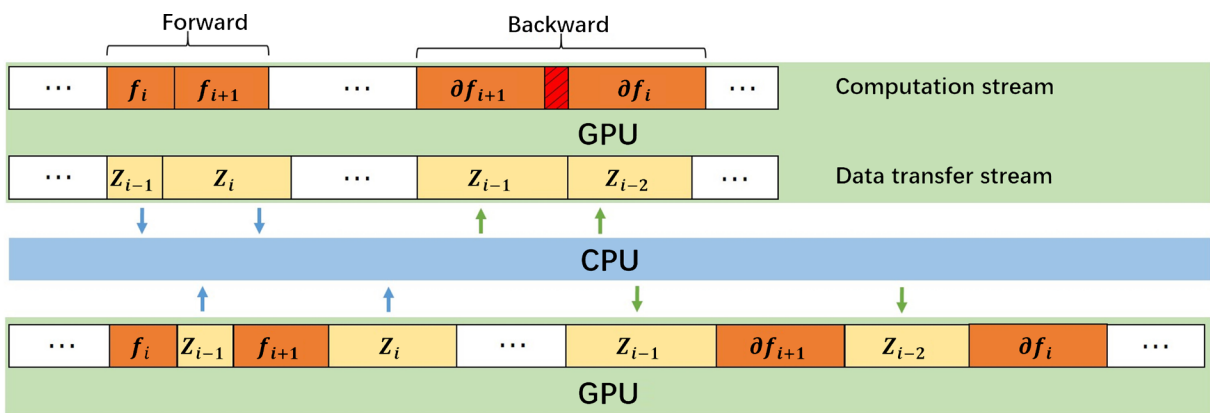

Figure 2. Illustration of the parallel CPU offloading execution through time. (Bottom): without parallelism, computations and data transfers are sequentially executed within the same stream so that the overhead in wall time corresponds to the total time of data transfers $\mathcal{T}_{data}$. (Top): Computations and data transfer are executed in parallel in their dedicated stream. The overhead in wall time $\mathcal{T}_{idle}$ is spent as the computation stream awaits for data transfers to complete (illustrated in red).

is stopped to await for the required data transfer to complete.

The key challenge in this parallelization scheme consists in efficiently managing 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 transferred to CPU upon computation by layer i as $z_i = f_i(z_{i-1})$, The GPU memory buffer of z_i must then await for both the data transfer to CPU and the next layer computation $z_{i+1} = f_{i+1}(z_i)$ to complete before de-allocation in order to avoid unfortunate overwriting by concurrent operations.

One important exception to this rule concerns skip connections, as illustrated in **Figure 3**. Skip connections induce a delay in the GPU buffer de-allocation as the input of residual blocks must be kept in memory until the end of the residual block computation to be added to the output. In ResNet architectures, this delay is short enough to have little impact on the memory/wall time trade-off. However, this means that our method is poorly suited to densely connected architectures such as DenseNet [19] or UNet [20] as their long-range skip connections introduce large delays in the GPU buffer de-allocation.

During the backward pass, the input activation z_{i-1} of layer i must be transferred back into GPU memory for the backward gradient computations δf_i to proceed. Hence, we overlap the transfer of z_{i-1} to GPU with the backward computations δf_{i+1} of the upper layer $i+1$ to avoid GPU idle time. As formalized, we thus synchronize the data transfer of z_{i-1} with the beginning of the backpropagation through f_{i+1}.

We use threading and locks to handle the parallelism and synchronization on the CPU side, and CUDA streams to handle the parallelization on the GPU device. CUDA events provide synchronization primitives to precisely track the completion of kernel executions. **Algorithm 1** and **Algorithm 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
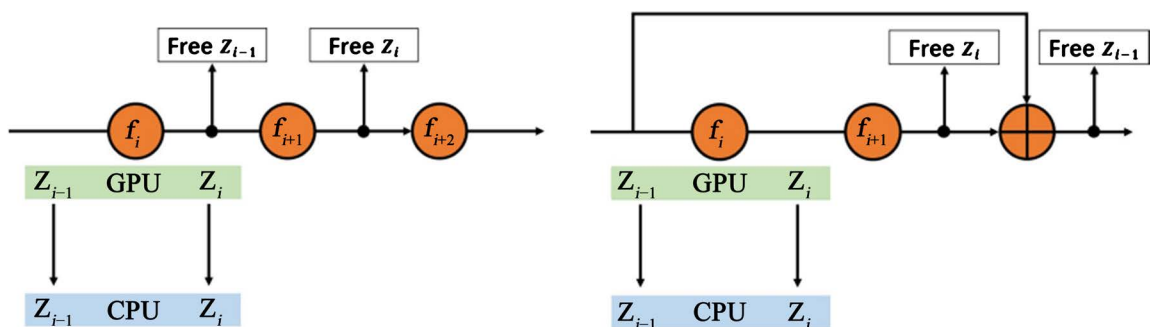

Figure 3. Illustration of the delay in memory de-allocation induced by residual connections. (Left) without residual connection, the input z_{i-1} of layer f_i can be freed from GPU memory as f_i computation terminates. (Right) Residual connections induce a delay in memory de-allocation as z_{i-1} must be kept in GPU memory to be added to the output of the residual block.

Algorithm 1. Forward procedure through layer i with parallel CPU offloading.

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);

$S_{comp} \Leftarrow z_i \leftarrow f_i\left(z_{i-1}\right)$;

$S_{comp} \Leftarrow E^i_{comp}$;

In Thread T_{data}:

$\qquad S_{data} \Leftarrow P_{i-1} \leftarrow z_{i-1}$;

$\qquad S_{data} \Leftarrow E^i_{data}$;

$\qquad$ Wait (E^i_{data}, E^i_{comp});

$\qquad$ Free (z_{i-1}).

Algorithm 1. 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 de-allocation without blocking the execution of upward layers.

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

Data: Layer f_i, output gradients $\dfrac{\delta\mathcal{L}}{\delta z_i}$

CPU pinned memory buffer P_{i-1}

CPU thread T_{comp}

CUDA events E^i_{data}, E^{i+1}_{data}, E^i_{comp}

CUDA Streams S_{data}, S_{comp}

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

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 ($\dfrac{\delta\mathcal{L}}{\delta z_{i-1}}$, $\dfrac{\delta\mathcal{L}}{\delta\theta_i}$);

$S_{comp} \Leftarrow \dfrac{\delta\mathcal{L}}{\delta z_{i-1}} \leftarrow \dfrac{\delta\mathcal{L}}{\delta z_i} \times \dfrac{\delta z_i}{\delta z_{i-1}}$;

$S_{comp} \Leftarrow \dfrac{\delta\mathcal{L}}{\delta\theta_i} \leftarrow \dfrac{\delta\mathcal{L}}{\delta z_i} \times \dfrac{\delta z_i}{\delta\theta_i}$.

---

will reduce memory consumption without any overhead in wall time. Hence, CPU offloading is best suited for computation-intensive operations like convolutions. In contrast, batch normalization 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 layers, we reconstruct their input activations from their output using their inverse operation, as proposed in [18] 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 bytes per second, by the volume $\mathcal{V}$ of the data buffer in bytes:

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

Hence, minimizing data transfer time can be achieved by either maximizing the transfer speed $\mathcal{B}$ or minimizing the activation volume $\mathcal{V}$ through data compression. In this paper, we focus on maximizing data transfer speed using pinned CPU memory buffers.

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 speed up data transfer between CPU and GPU devices through PCIe lanes. However, allocating pinned memory buffers is a time-consuming operation, as illustrated in Table 1. Hence, naively allocating pinned memory buffers within each forward pass would slow down training due to the overhead of pinned memory allocation. Instead, we propose to allocate the pinned memory buffers beforehand, during the initialization of the CNN. Our implementation scans the network architecture upon instantiation and allocates dedicated pinned memory buffers for each target layer activation (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 1 quantifies the gains in transfer speed brought by the use of pinned memory buffers

Table 1. Measurement of data transfer and pinned memory allocation speed on our workstation. Pinned memory buffers bring a substantial speedup in both directions of the data transfer.

Operation	Speed (GB/s)
Pinned memory allocation	1.95
GPU → CPU (pageable)	1.75
GPU → CPU (pinned)	6.5
CPU → GPU (pageable)	4.1
CPU → GPU (pinned)	6.1

and, in Section 5, we evaluate the impact of using pinned memory buffers on the overall efficiency of our method.

Another way to reduce the time of data transfer between CPU and GPU is to minimize $\mathcal{V}$ by compressing 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 would slow down the computation. Either sparsifying or quantizing the layers activations are interesting solutions. However, compression would incur minor losses of precision in the activation values, which might affect the training dynamics. Although previous works suggest that such minor losses in precision do not negatively impact training, investigation of compression techniques would require a detailed analysis of the impact of compression on accuracy, which is out of the scope of this paper. Hence we leave this question open for future work as an interesting direction for further optimization.

5. Experiments

We conduct our experiments on a workstation with an NVIDIA Titan X GPU connected to the host through PCIe1 16× lanes, which represents a peak bandwidth of 6.5 GB/s. We believe this set-up to be representative of a typical workstation for individual practitioners, if not slightly unfavorable, as our device to host bandwidth is rather limited. Unless specified otherwise, we conduct our experiments on a batch of 32 RGB images with a resolution of 128 by 128 pixels. This allows us to fit the training of the largest baseline architectures in memory, while keeping a reasonable training speed for fast iteration. For each experiment, we are interested in the total memory usage and Wall time of the computation for a full training iteration including the forward and backward pass through the network. It is important to note that our method does not modify the actual computations performed by the network so that the training dynamics and accuracy are unaffected by CPU Offloading. Hence, we focus on reporting the memory vs. wall time trade-off, as the accuracy reached by the model is strictly unaffected.

5.1. Memory vs. Wall Time Trade-Off

Our implementation allows controlling for the memory foot-print vs. computation wall time trade-off by either using more aggressive parallelization schemes or by offloading different subsets of hidden layers input activations. In this section, we use the parallelization scheme described in Section 4.2 and explore the memory vs. time trade-off by off-loading different subsets of the model's layers.

Figure 4 compares the memory/Wall time trade-off of CPU off-loading to the trade-off provided by gradient checkpointing, as implemented in the Pytorch [21] library on the VGG19 architecture. In gradient checkpointing, this trade-off is defined by the number of checkpointed layers through the full architecture.

Figure 4 shows the pareto frontier for an extensive search in both check-pointing and CPU offloading layer subset. CPU offloading leads to a better trade-off between memory consumption and wall time. This observed performance is due to the fact that the data transfer between host and device can be efficiently parallelized with the computation, hence incurring a minor overhead, while gradient checkpointing cannot parallelize the gradient computations and activation reconstructions. Hence, the additional cost of sequentially reconstructing the hidden activations during the backward pass induces a higher overhead.

5.2. Ablation Study

Table 2 illustrates the speed up brought by both parallelization and pinned memory buffers. We record the timings of a training iteration needed to achieve maximal memory reduction on the VGG19 architecture for an input batch of 32 images with 218 by 128 pixel resolution. We start by recording the time for a baseline CPU offloading approach without parallelization or data transfer optimization. We then successively integrate the pinned memory buffers and parallelization to this baseline and observe the improvement.

Both parallelization and pinned memory buffers bring substantial improvements in speed and have proven critical for performance. Compression of the
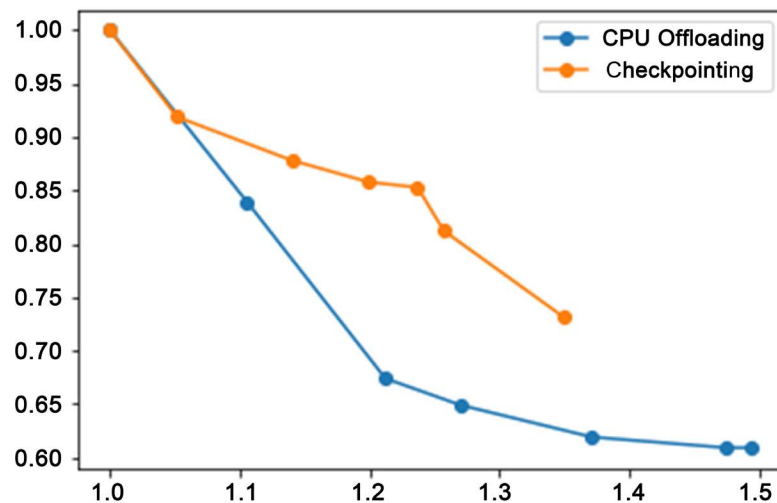

Figure 4. Comparison of the memory vs. wall time trade-off achieved by CPU offloading and sequential gradient checkpointing. The x axis represents the relative memory reduction obtained from the baseline model. The y axis represents the relative increase incurred in Wall time to achieve this memory reduction.

Table 2. Speed-ups achieved by the pinned memory buffers and parallelization techniques.

Operation	Time (ms)	Rel. Improvement
Baseline	560	0%
Pinned memory	355	−36.6%
Parallelization	236	−21.2%

hidden states to either sparse representations or lower precision encodings may bring similar speed-up and would be very interesting to investigate in future work. We expect that simply converting the hidden activations to FP16 before sending them to the host device should further reduce the data transfer time by half.

5.3. Architecture Comparison

In this section, we compare the memory gains and time overhead achieved through CPU offloading on different popular architectures (Table 3).

Both ResNet architectures and MobileNet [10] feature batch normalization layers, for which we used the invertible operation of [18]. Invertible batch normalization layers further reduce the memory consumption with little time overhead, which explains the slightly better trade-off of ResNet architectures. On the other hand, the MobileNet architecture uses depth-wise convolutions instead of the regular convolution layers used in both VGG and ResNet architectures. Depth-wise convolutions have lower arithmetic intensity [22] than convolution layers, *i.e.*; for input activations of similar size, they perform less, hence faster, operations. As computations are performed faster while the data transfer time remains constant, CPU offloading of depth-wise convolutions incurs a more important overhead. This result highlights the fact that CPU offloading works best for computationally intensive layers.

5.4. Discussion

The memory/time trade-off achievable by CPU offloading is a function of the ratio between computation and data transfer speed. This ratio varies with the specific hardware being run on, the algorithm used for the convolutions, as well as the efficiency of their implementation.

On the software side, the experiments presented in this paper were run with state-of-the art cudnn implementations of the winograd convolution algorithm through the PyTorch framework [21]. On the hardware side, we ran experiments on a single Nvidia Titan X GPU connected to the host via a 6 GB/s bandwidth connection, which we believe to be representative of a typical deep learning student/researcher's workstation.

Table 3. Memory vs. time trade-off achieved by different architectures. The Wall Time Overhead column shows the additional time needed to achieve the given memory reduction (the lower the better). Memory Reduction represents the reduction of peak memory usage achieved with CPU offloading (the higher the better).

Architecture	Wall Time Overhead	Memory Reduction
VGG16	19.5%	32.5%
VGG19	21.0%	36.9%
Resnet18	17.6%	54.4%
Resnet50	15.3%	56.8%
MobileNet	63.2%	67.8%

The exact trade-offs achieved will differ depending on the specific hardware and software used in different systems, as well as the parallelization strategy used for CPU offloading. However, the fact that data transfers between CPU and GPU can be efficiently parallelized with the GPU computations means that memory gains can be achieved with minimal time overhead on any system given an appropriate parallelization strategy.

Searching for an optimal parallelization strategy given a fixed computational graph and hardware system is a very interesting problem left open to investigate. In this paper, we provide a proof of concept by showing that even a rather naive parallelization strategy with minimum optimization leads to a competitive memory/time trade-off relatively to the well-established method of gradient checkpointing.

6. Conclusion

Convolutional Neural Networks have become the backbone of computer vision systems. Despite their great success, one major drawback of these models is their intense resource consumption: Training CNNs needs highly optimized implementations leveraging all possible hardware resources. In this paper, we proposed to leverage GPU-CPU communication, an under-utilized resource in typical CNN training pipelines, to alleviate the GPU bottleneck in training CNNs. Our experiment on a standard single-GPU work station shows a more effective memory/wall time trade-off than gradient checkpointing. Our approach is orthogonal to other resource optimization approaches such as pruning and quantization so these approaches can easily be combined in future work.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., *et al.* (1998) Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, **86**, 2278-2324. https://doi.org/10.1109/5.726791

[2] Krizhevsky, A., Sutskever, I. and Hinton, G.E. (2012) ImageNet Classification with Deep Convolutional Neural Networks. *Proceedings of the* 25*th International Conference on Neural Information Processing Systems*, Volume 1, Lake Tahoe, 3-6 December 2012, 1097-1105.

[3] Kingma, D.P. and Ba, J. (2014) Adam: A Method for Stochastic Optimization.

[4] Shallue, C.J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R. and Dahl, G.E. (2018) Measuring the Effects of Data Parallelism on Neural Network Training.

[5] McCandlish, S., Kaplan, J., Amodei, D. and OpenAI Dota Team (2018) An Empirical Model of Large-Batch Training.

[6] You, Y., Gitman, I. and Ginsburg, B. (2017) Large Batch Training of Convolutional Networks.

[7] Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J. and Keutzer, K. (2016) SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and < 0.5 MB Model Size.

[8] Molchanov, P., Tyree, S., Karras, T., Aila, T. and Kautz, J. (2016) Pruning Convolutional Neural Networks for Resource Efficient Inference.

[9] Frankle, J. and Carbin, M. (2018) The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks.

[10] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., *et al.* (2017) Mixed Precision Training.

[11] Jacob, B., Kligys, S., Chen, B., Zhu, M.L., Tang, M., Howard, A., Adam, H. and Kalenichenko, D. (2018) Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Salt Lake City, 18-22 June 2018, 2704-2713. https://doi.org/10.1109/CVPR.2018.00286

[12] Wu, S., Li, G.Q., Chen, F. and Shi, L.P. (2018) Training and Inference with Integers in Deep Neural Networks.

[13] Martens, J. and Sutskever, I. (2012) Training Deep and Recurrent Networks with Hessian-Free Optimization. In: *Neural Networks: Tricks of the Trade*, Springer, Berlin, 479-535. https://doi.org/10.1007/978-3-642-35289-8_27

[14] Chen, T.Q., Xu, B., Zhang, C.Y. and Guestrin, C. (2016) Training deep nets with sublinear memory cost.

[15] Gomez, A.N., Ren, M.Y., Urtasun, R. and Grosse, R.B. (2017) The Reversible Residual Network: Back-Propagation without Storing Activations. 31*st Conference on Neural Information Processing Systems*, Long Beach, 2214-2224.

[16] Jacobsen, J.-H., Smeulders, A. and Oyallon, E. (2018) i-RevNet: Deep Invertible Networks.

[17] Huang, G., Liu, Z., Van Der Maaten, L. and Weinberger, K.Q. (2017) Densely Connected Convolutional Networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Honolulu, 21-26 July 2017, 4700-4708. https://doi.org/10.1109/CVPR.2017.243

[18] Ronneberger, O., Fischer, P. and Brox, T. (2015) U-Net: Convolutional Networks for Biomedical Image Segmentation. In: *International Conference on Medical Image Computing and Computer-Assisted Intervention*, Springer, Berlin, 234-241. https://doi.org/10.1007/978-3-319-24574-4_28

[19] Rota Bulò, S., Porzi, L. and Kontschieder, P. (2018) In-Place Activated BatchNorm for Memory-Optimized Training of DNNs. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Salt Lake City, 18-22 June 2018, 5639-5647. https://doi.org/10.1109/CVPR.2018.00591

[20] Howard, A.G., Zhu, M.L., Chen, B., Kalenichenko, D., Wang, W.J., Weyand, T., Andreetto, M. and Adam, H. (2017) MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.

[21] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z.M., Desmaison, A., Antiga, L. and Lerer, A. (2017) Automatic Differentiation in PyTorch. 31*st Conference on Neural Information Processing Systems*, Long Beach, 1-4.

[22] Wu, B.C., Wan, A., Yue, X.Y., Jin, P., Zhao, S.C., Golmant, N., Gholaminejad, A., Gonzalez, J. and Keutzer, K. (2018) Shift: A Zero Flop, Zero Parameter Alternative to Spatial Convolutions. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Salt Lake City, 18-22 June 2018, 9127-9135. https://doi.org/10.1109/CVPR.2018.00951