Out-Of-Order BackProp: An Effective Scheduling Technique for Deep Learning

Hyungjun Oh
Hanyang University
tcpip15@hanyang.ac.kr

Junyeol Lee
Hanyang University
shie007@hanyang.ac.kr

Hyeongju Kim
Hanyang University
gudwn0520@hanyang.ac.kr

Jiwon Seo*
Hanyang University
seojiwon@hanyang.ac.kr

Abstract

Neural network training requires a large amount of computation and thus GPUs are often used for the acceleration. While they improve the performance, GPUs are underutilized during the training. This paper proposes *out-of-order (ooo) backprop*, an effective scheduling technique for neural network training. By exploiting the dependencies of gradient computations, ooo backprop enables to reorder their executions to make the most of the GPU resources. We show that the GPU utilization in single- and multi-GPU training can be commonly improve by applying ooo backprop and prioritizing critical operations. We propose three scheduling algorithms based on ooo backprop. For single-GPU training, we schedule with multi-stream ooo computation to mask the kernel launch overhead. In data-parallel training, we reorder the gradient computations to maximize the overlapping of computation and parameter communication; in pipeline-parallel training, we prioritize critical gradient computations to reduce the pipeline stalls. We evaluate our optimizations with twelve neural networks and five public datasets. Compared to the respective state of the art training systems, our algorithms improve the training throughput by 1.03–1.58× for single-GPU training, by 1.10–1.27× for data-parallel training, and by 1.41–1.99× for pipeline-parallel training.

CCS Concepts: • **Computing methodologies** → **Machine learning**.

Keywords: Deep learning systems

ACM Reference Format:
Hyungjun Oh, Junyeol Lee, Hyeongju Kim, and Jiwon Seo. 2022. Out-Of-Order BackProp: An Effective Scheduling Technique for Deep Learning. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3492321.3519563

*Corresponding author and principal investigator

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04…$15.00
https://doi.org/10.1145/3492321.3519563

1 Introduction

Deep neural networks (DNNs) are now widely used in many domains. Because training and running neural networks are computationally expensive, GPUs are commonly used for the acceleration. While they substantially speedup the performance, GPUs are often underutilized when running neural network tasks. At a single GPU level, many of neural network kernels have low GPU resource utilization [46, 60, 76]; at a cluster level, half of the GPUs running neural network tasks are idle, wasting their computation cycles [37].

Hardware resource utilization is not a new problem. In the 80s and 90s, the increasing transistor density and clock frequency of CPUs made it challenging to efficiently utilize CPU resources. To improve the utilization efficiency, instruction-level parallelism (ILP) had been extensively studied in both hardware and software aspects [42, 66]. Techniques such as instruction pipelining and out-of-order execution are proposed to increase the degree of ILP and CPU utilization.

Inspired by the past studies on ILP and carefully investigating DNN tasks, we propose scheduling optimizations for DNN training. Although GPU underutilization for single and multi-GPU DNN training is caused by different reasons, we learned that their performance can be largely improved by scheduling their operations efficiently. We proposed three scheduling algorithms for single- and multi-GPU (data- and pipeline-parallel) training. The algorithms, while they differ largely in details, apply the same principle of prioritizing critical operations and increasing execution concurrency to improve GPU utilization and training performance.

All the scheduling algorithms are based on our novel scheduling technique, which we call *out-of-order backprop*. Although existing deep learning systems perform backpropagation strictly in the reverse order of the network layouts, we observed that a subset of gradient computations may be executed in an out-of-order manner. By exploiting this property, we schedule the gradient computations such that the critical ones are executed with higher priorities. For single-GPU training, out-of-order backprop helps to mask the kernel launch overhead; for data-parallel and pipeline-parallel training, the technique helps to maximize the overlapping of inter-GPU communication with gradient computations.

This paper contributes out-of-order backprop and the scheduling algorithms based on this technique. We summarize our specific contributions in the following.

Concept of out-of-order backprop. We propose *out-of-order backprop* as a general principle for scheduling the computations in DNN training. Exploiting the computation dependencies in the training, it enables the execution of the gradient computations out of their layout order so that more critical computations are executed with higher priorities.

Scheduling algorithms for single and multi-GPU training. We designed three scheduling algorithms based on ooo backprop and the list scheduling technique. For single-GPU training, we schedule weight- and output-gradient computations in multiple GPU streams and in an out-of-order manner. For data-parallel training, our scheduling algorithm reorders the gradient computations to maximize the overlapping of the communication and computation. For pipeline-parallel training, we prioritize critical gradient computations to reduce the pipeline stalls. All our algorithms make use of gradient computation reordering (i.e., ooo backprop), which no prior work had exploited for single or multi-GPU training.

Implementation in real-world deep learning systems. We implement out-of-order backprop and our scheduling algorithms in TensorFlow, a widely-used deep learning system. We modified TensorFlow's execution engine and its compiler XLA to implement our scheduling techniques; we also added an efficient support for an auxiliary GPU stream to concurrently execute a subset of gradient computations. Moreover, we implemented our scheduling algorithms in BytePS, the state of the art parameter-communication system for distributed neural network training [38].

Extensive evaluation and availability. We evaluate ooo backprop and the scheduling algorithms with twelve neural networks in computer vision and natural language processing. The evaluation is performed on three different GPU models, with four types of network interconnect, and on a cluster of up to forty eight GPUs. For all (single- and multi-GPU) training methods, our scheduling algorithms largely improve the performance, compared to the respective state of the art systems. For single-GPU training, our scheduling algorithm using multi-stream out-of-order computation improves the training performance by 1.03–1.58× over TensorFlow XLA; compared to Nimble, a state of the art deep learning execution engine, we exceed its performance by 1.28× on average. For data-parallel training, our technique that prioritizes critical computations outperforms BytePS by 1.10–1.27× on a cluster with sixteen to forty eight GPUs. For pipeline-parallel training, we outperform GPipe by 1.41–1.99× on a cluster with four to thirty six GPUs; compared to PipeDream that applies *weight stashing* and thus changes the semantics of the training, our execution runs 1.31× faster on average. We report the analysis of our performance improvements for the three training methods. We open-sourced our implementations and scheduling algorithms in TensorFlow and BytePS as well as the execution schedules for the evaluated neural network models [7].

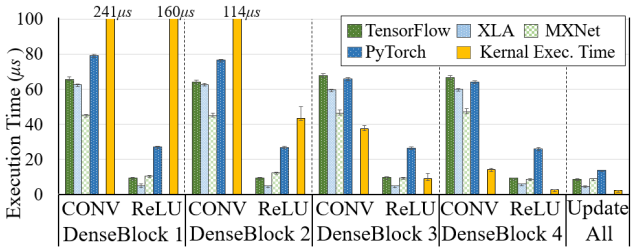

Figure 1. Kernel issue overhead of deep learning systems for convolution and ReLU operations in DenseNet-121.

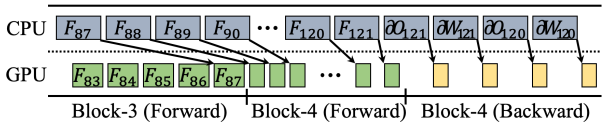

Figure 2. Timeline of training DenseNet-121; the kernel issue overhead (top) and their executions on GPU (bottom).

The rest of the paper is organized as follows. Section 2 reviews GPU underutilization problems. Section 3 presents ooo backprop, our core scheduling technique. Section 4 and 5 describe our scheduling algorithms for single- and multi-GPU training. Section 7 explains our implementation. Section 8 evaluates our proposed scheduling algorithms. Section 9 discusses related work and Section 10 concludes.

2 GPU Underutilization Problems

DNN training requires a large amount of computation and GPUs are widely used for the acceleration. Although they largely speed up the training, GPUs are often underutilized during the training [37, 46, 60, 76]. Here we review the GPU underutilization problems in single- and multi-GPU training.

Analysis for single-GPU training. In single-GPU training, GPU underutilization is caused by kernel issue/execution overhead and idling SMs (Stream Multiprocessors) during kernel executions. We first consider the kernel issue overhead. In deep learning systems, DNN training is represented by a computation graph with DNN operations and dependencies. The systems have an *executor* that traverses the graph and asynchronously issues the GPU kernels. If the latency of issuing the kernels is longer than their executions on GPU, this overhead may become the performance bottleneck.

In our preliminary experiments, we measured the kernel issue overhead in TensorFlow, PyTorch, and MXNet. For many DNN models, the overhead is the bottleneck of the training. Particularly, recent convolutional neural networks (CNNs), such as DenseNet or MobileNet, are largely affected by this overhead as they have many light-weight convolutions. Figure 1 shows the kernel issue overhead for DenseNet-121 (on Intel Xeon E5-2698 and NVIDIA V100). For the convolutions in DenseBlock-3 and 4, their issue overhead is up to 4× larger than their execution times; since the two DenseBlocks take up two thirds of the total execution, this overhead is critical.

Figure 2 is the (simplified) actual timeline of training DenseNet in TensorFlow; part of forward and backward computation is shown. For DenseBlock-3's forward computation, the kernel issue overhead is masked by the previously issued kernels. However, the masking effect disappears by the end of Block-4, when the idle time between the kernels increases. This is also reported in other recent studies [41, 53].

Moreover, we observed GPU's kernel execution overhead. Even if the kernel issue overhead is completely masked, there is 1–2μs idle time between the kernel executions (e.g. forward computation of DenseBlock-3 in Figure 2). This overhead is caused by the GPU's execution engine for setting up the SMs for the kernel execution [63, 70]. For short-running kernels the overhead is non-trivial; e.g., for the convolution kernels in DenseBlock-3 and 4, their executions are 15–40μs and thus the kernel execution overhead is 3–13% of their execution.

The other cause of the GPU underutilization is idling SMs during and end of kernel executions. During the execution of a kernel, its thread blocks are scheduled and executed in the SMs. If the kernel has a smaller number of thread blocks than the number that the SMs are capable of running at once, the SMs may not be fully utilized. For example, for the weight gradient kernels in DenseBlock-4 (on V100 GPU), half of the kernels run with the same configuration of 448 thread blocks. However, the SMs are capable of running 1,520 of the thread blocks and thus they are underutilized in terms of the thread block capacity. Also, at the end of the kernel execution when its last thread blocks are scheduled, the SMs are likely to be underutilized because they are running fewer number of thread blocks than their capacity. This problem is also referred to as tail underutilization [17, 50].

Analysis for GPU clusters. It is reported that the utilization of GPU clusters is less than 52% for neural network tasks [37]. The low cluster utilization is caused by many factors, such as inefficiency of task scheduling, but the primary reason is the communication overhead in data-parallel training and the pipeline stalls in model-parallel training.

In data-parallel training, workers communicate to synchronize their weight parameters in each training iteration. Because the forward computation of the next iteration blocks until the synchronization is completed, GPU cycles are wasted during the synchronization. It is reported that the wasted GPU cycles from the synchronization is 15–30% of the total execution time if wait-free backpropagation is applied to overlap the synchronization and gradient computation [74]. Prioritization of parameter communication [28, 36, 38, 44, 56] may further reduce the cost but still the overhead is 10–25% as our evaluation shows. Recently, asynchronous parameter communication schemes are proposed to improve the training throughput [31, 47, 75]. However, because they may incur accuracy loss, those asynchronous schemes are less widely applied in practice [27, 39, 71].

Now let us consider model-parallel training. For large models such as GPT, cross-layer model-parallelism is commonly used, where each layer is assigned to one of working GPUs for training, which performs the layer's forward and backward computation. Due to the computation dependency, only a subset of the GPUs perform computation at once. That is, in both forward and backward propagation, many of the GPUs are stalled waiting for the computation result from the GPUs for the earlier (or later, in backpropagation) layers.

To mitigate the cost of the execution stalls, pipelining of the GPU computations is further proposed [35, 51]. In pipeline-parallel training, input data (i.e. mini-batch) is split into *micro-batches*, which are sequentially fed to the GPUs for the concurrent executions of the GPUs. Although using micro-batches increases the number of concurrently in-use GPUs, still a substantial portion of the GPUs are idle during the training as we show in Section 8.4. Recently PipeDream proposed to train with multiple versions of weight parameters to increase the overlapping the computations of different micro-batches [35, 51]. However, this technique causes parameter staleness in a similar manner to the asynchronous communication schemes in data-parallel training, and thus it may negatively affect the learning efficiency [14**?**].

Formulation of a common optimization problem. The GPU underutilization in single-GPU, data-parallel, and pipeline-parallel training is caused by different reasons; thus different optimization techniques are previously proposed for them [35, 36, 41, 51]. While the problems in the three training methods seem to be a separate issue, they can be formulated as a same optimization problem. In all three training methods, we need to optimize for the GPU utilization and training throughput. This requires carefully scheduling the training operations to maximize, e.g., the overlapping of the computation and communication. Commonly for the three training methods, we formulate the problem of optimizing the execution of a single training iteration as following.

$$\begin{aligned}
&\underset{T:\mathbb{C}\rightarrow\mathbb{R}^{|\mathbb{C}|}}{\text{minimize}} \quad T(F_L)+F_L\\
&\text{s.t. } T(\delta O_{L+1})=0,\ \{T(\delta W_i),\ T(\delta O_i)\}\geq T(S[\delta O_{i+1}])+S[\delta O_{i+1}],\\
&\quad T(S[\delta O_i])\geq T(\delta O_i)+\delta O_i,\ T(S[\delta W_i])\geq T(\delta W_i)+\delta W_i,\\
&\quad T(F_i)\geq T(S[\delta W_i])+S[\delta W_i],\ T(F_{i+1})\geq T(F_i)+F_i,\ \text{ where}
\end{aligned}$$

- the constraints are the dependencies of the gradient computations, synchronizations, and forward computations.
- $F_i, \delta O_i, \delta W_i$ are i'th layer's forward, output and weight gradient computation; they also denote their execution times.
- $S[\delta O_i]$ and $S[\delta W_i]$ are the synchronization of δO_i and δW_i. They also denote the synchronization costs (time).
- L is the number of layers, $\mathbb{C}$ is the set of all operations, i.e., $\{F_1, \delta W_1, S[\delta W_1], ...\}$, and $\mathbb{R}$ is the set of real numbers.
- T is the function mapping the start time for the operations.

The goal is to find T that minimizes the makespan. Note that we start with the backpropagation and end with the next iteration's forward propagation; the gradient computation

of loss (δO_{L+1}) is scheduled at time zero and the completion time of the last layer's forward computation is minimized.

This is a variant of job shop scheduling problem [25], which is known to be NP [22, 32]. Heuristic algorithms such as list scheduling [16] or HEFT [67] are generally used to find a good solution that satisfies the constraints. We show in Section 4 and 5 that our algorithm based on the list scheduling technique [16] can find optimized kernel schedules for all three training methods (with different prioritization schemes). Before we present the scheduling algorithms, we first describe our core scheduling technique that allows flexible scheduling of neural network operations.

3 Out-Of-Order BackProp

In forward propagation of DNN training, each layer computes the output with its input, i.e., the prior layer's output. The computed outputs are stored for backpropagation. The final output is compared with the target value to compute the loss. In backpropagation, the computation is performed in the backward direction to calculate the output gradient (δO) and weight gradient (δW) for each layer. The computed output gradient is used to calculate the gradients in the prior layer. However, the weight gradient is not used to compute any other gradients; it is only used to update the layer's weight parameters. This dependency is shown in Figure 3 (a).

Exploiting the dependencies of gradient computations, we propose *out-of-order backprop* (*ooo backprop* in short), which schedules the weight gradient computations out of their layout order. In conventional backpropagation, the gradient computation and weight update are performed in the reverse order of the layers in a network. That is, the two gradient computations for a layer are completed before starting the previous layer's gradient computations. Figure 3 (a) shows the execution of conventional backpropagation.

Out-of-order backprop decouples the gradient computations from the structure of a neural network, thus scheduling the computations in a flexible manner. As a result, more critical computations can be scheduled with higher priorities. For example, if applied to pipeline-parallel training, we can schedule all the output gradient computations of a GPU before its weight gradient computations so that the next GPU may promptly start its computation. This is shown in Figure 3 (b). The current GPU computes δO_i and δO_{i-1}, sending the output to the next GPU assigned with the prior two layers; then it computes δW_i and δW_{i-1} concurrently with the next GPU computing δO_{i-2} and δO_{i-3} (not shown).

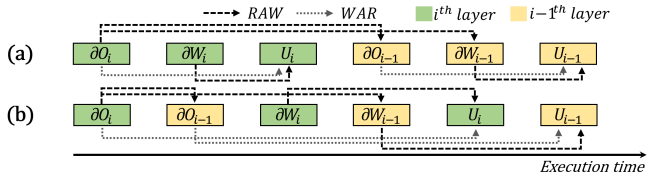

Figure 3. Backpropagation of two layers. The dependencies of $\delta O/\delta W$ computations and weight updates (U) are shown; (a) is conventional backpropagation and (b) is ooo backprop.

When scheduling weight gradient computations, we need to consider both GPU utilization and memory overhead. Because the weight gradient computation of a layer requires the layer's input and output gradient, they must be retained in memory until the computation is done. Our proposed scheduling algorithms consider this memory overhead and find efficient schedules with minimal memory overhead.

4 Scheduling for Single-GPU Training

In single-GPU training the kernel issue/execution overhead and idling SMs in kernel executions cause the GPU underutilization. This section presents our technique that applies concurrent GPU streams and ooo backprop to improve the GPU utilization for single-GPU training.

4.1 Multi-Stream Out-Of-Order Computation

To mask GPU's idle cycles, we propose to use two GPU streams, namely *main-stream* and *sub-stream*. In main-stream we allocate the operations in the critical path, i.e., output gradient computations and forward computations of all layers; we set this stream's priority high in the GPU execution engine. In sub-stream we run weight gradient computations and weight updates. Using two streams in this manner needs an additional constraint in the optimization problem we defined in Section 2: $T(\delta W_i) \geq T(\delta W_j) + \delta W_j$ if $T(\delta W_i) \geq T(\delta W_j)$.

To solve this optimization problem, we use the list scheduling technique and prioritize the co-scheduling of the kernels with highest speedup when run together. Because the GPU execution engine dynamically determines the SM allocations for the kernels, it is not feasible to apply fine-grained scheduling to exactly overlap the executions of two kernels. Hence we exercise a coarse-grained control and apply region-based scheduling. That is, we divide the forward and backward propagation into multiple regions with similar compute characteristics; e.g. a ResNet block can be a single region (forward and backward separately) as it consists of the same repeated convolutions. Then for each region we co-schedule the kernels that give highest speedups.

More specifically our scheduling proceeds as following:

1. For all the region pairs we profile their concurrent kernel runs and record the speedups over their sequential runs.
2. We sequentially schedule the main-stream kernels.
3. For the sub-stream kernels, we compute their schedulable time intervals and regions; then we assign the kernels to those regions as the schedulable candidates.
4. For each region, among its candidates we find the sub-stream kernel with the highest speedup for the main-stream kernels in the region. Then we select and schedule the region-kernel pair with the highest speedup.
5. We repeat step 4 until all the sub-kernels are scheduled.

Algorithm 1: Multi-Region Joint Scheduling

Input: $R[1..N]$: main-stream kernel schedule split into N regions,
$T_{main}(R[i])$: total exec time of $R[i]$'s main-stream kernels,
$T_{sub}(k, R[i])$: exec time of the sub-stream kernel k in $R[i]$.
Output: Sub-stream's schedule $S[1..N]$ for N regions

1 $S[i] = [],\ now[i] = 0$ for $i = 1, ..., N$;
2 $\mathcal{U} = \{\delta W_2, ..., \delta W_L\}$; $C = \{R[1], ..., R[N]\}$;
3 **while** $\mathcal{U} \neq \emptyset$ **do**
4 **foreach** region $R[i] \in C$ **do**
5 find δW_k runnable at $now[i]$ w/ max speedup p in $R[i]$
6 $tmp[i] = (\delta W_k, p)$;
7 $j = \arg\max_i f(i) = tmp[i][2]$;
8 $(\delta W_K, P) = tmp[j]$;
9 $S[j]$.append(δW_K); $\mathcal{U}.remove(\delta W_K)$;
10 $now[j] \leftarrow now[j] + T_{sub}(\delta W_K, R[j])$;
11 **if** $now[j] \geq T_{main}(R[j])$ **then** C.remove$(R[j])$;
12 **return** *S[1..N]*;

The overhead of the profiling is minimal because it can be performed as part of the training and also the number of regions that we use is fairly small; in our evaluation we used eight regions for DenseNet-121 (one for each DenseBlock). This algorithm is a variant of list scheduling, which divides the timeline into multiple regions and jointly schedules for those regions altogether. This region-based approach works well in practice because recent neural networks often have sub-structures with similar operations such as DenseBlocks or ResNet blocks. Algorithm 1 shows the pseudocode for step 4 and 5 above. In the algorithm, we simulate the timeline of the regions (denoted by $now[i]$) using the expected kernel execution time and the profiling results. In lines 4–6 we find the sub-stream kernel that gives the highest speedup within each region; then we select the kernel and region with the highest speedup (lines 7–8) and schedule the kernel in the region (line 9). We update the timeline for the scheduled region (line 10–11) and repeat the scheduling process.

With the schedule given by Algorithm 1, we compute its memory usage. If the peak memory usage is too high, we pre-schedule for k regions at the beginning of the backpropagation; in the k regions, we schedule the weight gradient computations as soon as they are ready, and thus the peak memory usage is decreased. Then we re-run Algorithm 1 for the remaining regions, increasing k for each re-run.

Because the caching effect of consecutively running the two gradient computations of a layer is not large, reordering them does not degrade the cache performance much. For convolutions, the two gradient computations first transform the inputs into much larger matrices and then compute with them. As those matrices are not shared by the two computations, the caching effect is limited. For GEMM, the inputs shared by the two gradient computations are smaller than the non-shared ones and thus the caching effect is not large.

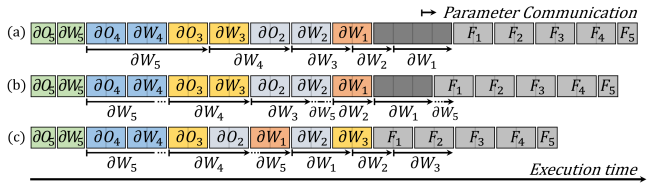

Figure 4. Data-parallel training with (a) conventional execution, (b) prioritizing the communications, (c) further prioritizing the computations. The black box is GPU idle time.

4.2 Pre-compiled Kernel Issue

Multi-stream ooo computation reduces the idling SMs by overlapping the kernel executions. However, the optimization may not be effective when the kernel issue overhead from the CPU is large. To mitigate the kernel issue overhead, we apply *pre-compiled kernel issue*.

In neural network training, the same computation graph is repeatedly executed and thus the same sequence of kernels are invoked over and again. NVIDIA released CUDA Graph API [26] that supports *capturing* a sequence of kernel issues and compiling them into an *executable graph*, which can be launched with a very small overhead using CUDA Graph Launch API. We apply this technique to reduce the kernel issue overhead so that multi-stream ooo computation (with multi-region joint scheduling) can effectively reduce the idling SMs and improve GPU utilization. A similar technique of collectively launching neural network kernels has been recently used by Nimble [41] (but without multi-stream ooo computation); MXNet [11] applies a similar principle but at the framework level without using CUDA Graph API.

5 Scheduling for Multi-GPU Training

In multi-GPU training, the main performance overhead is 1) parameter communication in data-parallel training and 2) pipeline stalls in pipeline-parallel training. This section describes our scheduling algorithms for multi-GPU training.

5.1 Data-Parallel Training

The optimization of data-parallel training is equivalent to the problem we defined in Section 2, if the synchronization of output gradient ($S[\delta O_*]$) is set to no-op with empty execution time. Like the single-GPU training case, this problem is NP hard and we propose a heuristic scheduling algorithm.

In data-parallel training, the parameter synchronization overhead ($S[\delta W_*]$ in the problem definition) is the major performance bottleneck. Figure 4 shows an example execution timeline of data-parallel training. In conventional backpropagation (a), the parameter communication, denoted by the arrows, postpones the forward computations and results in GPU idle cycles (the dark gray boxes in the timeline). If we apply the prioritized parameter communication technique that is proposed in recent studies [28, 36, 38, 44, 56], the performance is slightly improved as shown in Figure 4 (b). The communication of δW_1–δW_4 is prioritized over δW_5 (denoted by the dotted arrow), which reduces the execution

Algorithm 2: Reverse First-K Scheduling

Input: k: number of layers to reschedule, MXM: max memory.
Output: Optimized schedule D

1 $max_k = \arg\max_j f(j) = M_{fwd} - \sum_{i=j+1}^{L} M(\delta O_i) + \sum_{i=1}^{j} M(\delta W_i)$ s.t. $f(j) < MXM$;
2 $k \leftarrow \min(max_k, k)$;
3 **for** $i \leftarrow L$ **down to** 1 **do**
4 **if** $i > k$ **then** $D.append(\delta W_i)$;
5 $D.append(\delta O_i)$;
6 **for** $i \leftarrow 1$ **to** k **do** $D.append(\delta W_i)$;
7 **return** D;

time by one unit time. In addition to the communication prioritization, we can further improve the performance by prioritizing the *computations* in the critical path. Specifically, let us schedule the computation of δW_2 and δW_3 (in this order) after the computation of δW_1 as shown in Figure 4 (c). Then the communication of δW_1 is masked by the computation of δW_2 and δW_3. The training time is reduced by three unit times, improving the performance by 16% compared to conventional backpropagation in (a) and by 12% compared to the prioritized communication in (b). Note that we can obtain this optimal schedule by reversing the computation of $\delta W_1 - \delta W_3$; for this example, we can obtain another optimal schedule if we reverse the computation of $\delta W_1 - \delta W_4$.

To find the optimal execution schedules such as Figure 4 (c), we need to design a list scheduling algorithm and prioritize the computations in the critical path. However, since parameter synchronization is the dominant factor of the performance, we achieve (mostly) the same effect by simply advancing the gradient computations upon which the critical synchronizations depend. Thus we design a heuristic algorithm, namely reverse first-k scheduling, that reversely orders the weight gradient computations of first k layers. Because those k layers are computed earlier in the forward propagation than the other layers, the synchronization of their weight gradients are the critical operations; hence prioritizing their gradient computations shortens the critical path and minimizes the total execution time. For example, the executions in Figure 4 (c) is equivalent to the result of applying reverse first-k scheduling with k=3.

Algorithm 2 describes reverse first-k scheduling. It reorders the backpropagation portion of the training. In lines 1–2, it adjusts the value of k to satisfy the given memory constraint; function $M()$ returns the amount of temporary memory that is used by the computation. In lines 3–5, it schedules the gradient computations from layer L down to layer 1 in the same way as conventional backpropagation except for the weight gradient computations of layer 1 to layer k (line 4). Lastly it schedules the weight gradient computations of layer 1 to layer k in the reverse order (line 6).

We determine the optimal value of k by applying the following heuristic search, assuming that the throughput is roughly a concave function of k. First we set k to be 0 and the increment variable Δk to be one tenth of the number of layers (L). We keep increasing k by Δk and measuring the throughput for all k<L. For the k value that gives the maximum throughput, we repeat the search within the range of (k-Δk, k+Δk), with setting Δk to be half of its previous value. We repeat this process until we find optimal k that gives the maximum throughput.

If the optimal value of k is given, Algorithm 2 effectively prioritizes the critical synchronizations. If k is too small, the synchronizations are not fully masked; if it is too large, the network bandwidth may be underutilized. The above heuristic search can efficiently find the optimal k that gives the fastest throughput. List scheduling, on the other hand, does not need to find such optimal values but it requires the execution times of the parameter synchronizations. Because it may not be easy to estimate the synchronization time, reverse first-k scheduling is more effective and suitable in practice.

5.2 Pipeline-Parallel Training

In pipeline-parallel training, each GPU is assigned with a subset of layers and computes for the assigned layers. The optimization of pipeline-parallel training is equivalent to the previous optimization problem, if we set $S[\delta W_*]$ to be no-op. For pipeline-parallel training, we first describe our optimization techniques without micro-batches, that is, for cross-layer model parallelism. Then we describe how our techniques improve pipeline parallelism with micro-batches.

5.2.1 Cross-Layer Model Parallelism. We can improve the performance of cross-layer model-parallel training by prioritizing the critical computations. As such, we propose *gradient fast-forwarding*, which prioritizes the executions of the output gradient computations over those of the weight gradient computations in the GPU's allocated layers. This

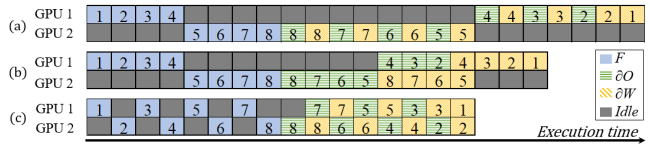

Figure 5. Cross-layer model parallelism; (a) conventional execution, (b) gradient fast-forwarding, (c) modulo allocation.

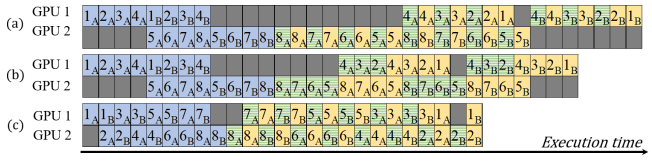

Figure 6. Pipeline Parallelism; (a) conventional execution, (b) gradient fast-forwarding, (c) modulo allocation.

way, the weight gradient computations in one GPU may overlap with the output gradient computations in the next GPU. Figure 5 (a) and (b) compare the execution timeline of training with two GPUs without and with the fast-forwarding. In (a), cross-layer model parallelism is applied and only a single GPU is used at a time; in (b) with gradient fast-forwarding, the computations of GPU_1 run concurrently with the weight gradient computations of GPU_2. The total execution time is reduced from 23 to 19 unit times resulting in 21% speedup.

We further speed up the training by efficiently partitioning a neural network and allocating the partitions to GPUs. In conventional cross-layer model-parallel (and pipeline-parallel) training, consecutive layers of a neural network are assigned to a same GPU to minimize inter-GPU communication overhead [35, 51]. Contrary to this conventional practice, we propose *modulo layer allocation* that may increase the inter-GPU communication. That is, when training with n number of GPUs, we allocate l'th layer of the neural network to $GPU_{l \bmod n}$; GPU_i computes for layer l_i, l_{i+n}, l_{i+2n}, etc. While it may increase the communication overhead, this modulo allocation gives much higher GPU utilization than the conventional allocation scheme, as we elaborate below.

Consider the training of the eight-layer neural network in Figure 5 (b). With gradient fast-forwarding, half of the backpropagation runs concurrently, but the two GPUs are idle for the other half of the backpropagation. If modulo allocation is further applied, both GPU_1 and GPU_2 are utilized for more than 90% of the backpropagation as shown in Figure 5 (c). Compared to the conventional execution in (a), the execution in (c) takes only 16 unit times, which is 1.44× speedup.

Moreover, modulo allocation reduces the (additional) memory overhead. Gradient fast-forwarding, if used with the conventional layer allocation scheme, requires retaining output gradients in memory for the delayed computations. Modulo allocation, however, hands over each layer's output gradient to a next GPU and immediately computes its weight gradient, thus lowering the memory pressure.

One drawback of modulo allocation is the increased inter-GPU communication. We can reduce the communication overhead by grouping a few consecutive layers and applying modulo allocation to the groups rather than the layers. In our evaluation, we applied modulo allocation at at a transformer level for NLP models. The details are discussed later but applying modulo allocation in this way gives a large performance gain over leading edge pipeline-parallel systems.

5.2.2 Pipeline Parallelism with Micro-Batches. The two techniques, that is, gradient fast-forwarding and modulo allocation, work well with the micro-batch technique. Figure 6 (a) shows the timeline of the pipeline-parallel training with micro-batches. The subscript denotes the micro-batch ID. The same eight-layer neural network in Figure 5 is trained with two GPUs, but a single mini-batch is now split into two micro-batches; the numbers denote the layer IDs and the subscripts denote the micro batch IDs. With micro-batches, Figure 6 (a) shows that the execution of the backpropagation of micro-batch A overlaps with that of micro-batch B, improving the GPU utilization over the cross-layer model-parallel training in Figure 5 (a). The fast-forwarding technique further increases the GPU utilization by overlapping the weight and output gradient computation of micro-batch B as shown in Figure 6 (b). In addition, if modulo allocation is applied as shown in Figure 6 (c), the pipeline stall in the forward computation is largely reduced. Also, modulo allocation increases the overlapping of the weight and output gradient computation in backpropagation, thereby improving the performance.

6 Combining Scheduling Techniques

Although we proposed three scheduling algorithms separately for single-GPU, data-parallel, and pipeline-parallel training, it is possible to combine some of the algorithms to further improve the performance. Consider, for example, the training of a large NLP model using both data-parallelism and pipeline-parallelism. It is possible to apply reverse first-k scheduling for the weight gradient computations in the first k layers and also apply gradient fast-forwarding for the last L-k layers (where L is the number of layers). In this way, the synchronization overhead in data-parallel training and the execution stall in pipeline-parallel training can be reduced altogether; we leave the problem of finding optimal k as a future work. In another example of training a computer vision model, we can apply both multi-stream ooo computation and reverse first-k scheduling; that is, the latter can be applied to the first k layers to reduce the synchronization overhead and the former to the L-k layers to reduce the kernel issue/execution overhead.

Moreover, our scheduling techniques can be applied with *check-point and re-computation*,[13], a technique that is often used for training neural networks whose intermediate tensors are larger than the size of GPU memory. With the re-computation technique, only a subset of the layer outputs in the forward computation is stored (i.e., check-pointed) and most of the outputs are discarded to reduce the memory overhead; those discarded outputs are re-computed during the backpropagation when they are required by the relevant gradient computations. The re-computation can be combined, to some extent, with our scheduling algorithms. Consider reverse first-k scheduling for data-parallel training that is applied with the re-computation. Because the algorithm only re-orders the weight gradient computations of first k layers, by the time we start the gradient computations for those k layers, most of the check-pointed outputs are already deallocated. Thus we have some amount of available memory to re-order those k (or maybe fewer) weight gradient computations to reduce the parameter synchronization overhead.

7 Implementation in TensorFlow

We implemented our optimizations in TensorFlow (v2.4) and XLA. To implement out-of-order backprop, we eliminated the use of `tf.group` that puts the weight and output gradient computations of a layer into a single node in the computation graph. By putting them in separate nodes, we remove the unnecessary dependencies for those gradient computations.

Moreover, we implemented an efficient support for an auxiliary GPU stream in TensorFlow executor. Although TensorFlow executor has preliminary implementation for multiple GPU streams (which is disabled by default [64]), it uses much more memory compared to the single-stream executions. That is, TensorFlow allocates the memory used by a kernel (such as input tensors or workspace memory for convolution) when the kernel is issued, which is before when the kernel is executed on GPU. For single-stream executions, because the kernels are executed in the order that they are issued, the memory block used by a kernel is reclaimed immediately after the kernel is issued if the memory block is not referenced by later-issued kernels. For multi-stream executions, this is not possible because the order that the kernels are issued may be different from the order that they are executed. Thus TensorFlow reclaims the memory when the kernel execution completes. This may increase the memory usage. Rather than using the generic but expensive multi-stream support in TensorFlow, we implemented a light-weight version that supports only one additional stream (i.e., sub-stream), for running the weight gradient computations. We also assigned a separate memory allocator for the temporary memory used by the sub-stream kernels. TensorFlow XLA, however, has an efficient memory allocator for multiple GPU streams based on its heap simulator; thus we exploit this memory allocator for our implementation on XLA. We used NVIDIA's event/stream APIs to enforce the dependency between the main-stream and sub-stream kernels. Our prototype is based on TensorFlow, but all our techniques can be implemented in MXNet or PyTorch. For example, in PyTorch ooo backprop can be implemented by modifying its autograd engine or by using its forward/backward hook mechanism (with operator customization); we used the latter to implement gradient fast-forwarding in Megatron 2 [54], as we described in Section 8.4.2.

8 Evaluation

This section presents a comprehensive evaluation of our scheduling algorithms. We evaluate our algorithms with twelve neural networks and five public datasets on a single and multiple GPUs. Because our optimizations do not change the semantics of neural network training, we only evaluate the training throughput and the memory overhead. The details of the evaluation are described later, but the short summary is that out scheduling algorithms effectively improve the performance of single-GPU training as well as data- and pipeline-parallel training. Compared to the respective state of the art systems, we speed up the throughput by 1.03–1.58× for single-GPU training, by 1.10–1.27× for data-parallel training, and by 1.41–1.99× for pipeline-parallel training.

Table 1. Models, datasets, and evaluation setup.

Training Method	Model	Dataset	GPU
Single GPU Training	DenseNet-{121,169} MobileNet V3 Large ResNet-{50,101}	CIFAR100 ImageNet	Titan XP V100
Data-Parallel Training	DenseNet{121,169} MobileNet V3 Large ResNet-{50,101,152}	ImageNet	Titan XP × 8 P100 × 20 V100 × 48
Pipeline-Parallel Training	RNN (16 Cell), FFNN BERT-{12,24,48} GPT-3 (Medium)	IWSLT MNLI OpenWebText	V100 × 36

8.1 Experimental Setup

Models and datasets. Table 1 describes the twelve neural networks and five datasets that are used for the evaluation. These models are state of the art neural network models that are commonly used in computer vision and natural language processing (NLP). DenseNet-{121, 169}, MobileNet, and ResNet-{50,101,152} are established CNN models in computer vision. DenseNet has *growth rate* as its hyper parameter (denoted by k); we set k=12, 24, and 32, which are the same as those used by the authors [34]. MobileNet also has a hyper parameter, namely *multiplier* (denoted by α); we use α=0.25, 0.5, 0.75, 1, which are also the same as those used by its authors [33]. For the language processing, we used Recurrent Neural Networks (RNNs), BERT-{12,24,48}, and GPT-3 (Medium) as they are the representative NLP models. We also experimented with a simple feed forward neural network (FFNN). We used public datasets that are widely used to evaluate CNNs (CIFAR100 [40] and ImageNet [57]) and language models (IWSLT [10], MNLI [69], and OpenWebText [24]).

GPUs and interconnects. We used NVIDIA Titan XP, P100, and V100 GPUs for the training as shown in Table 1. Titan XP GPUs are installed on a small cluster of eight machines. The machines have Intel Xeon E5-2620 v4 running at 2.1GHz and 64GB of DRAM. Each machine has a single Titan XP GPU connected via PCIe 3.0 x16; the machines are connected via 10Gb Ethernet. P100 GPUs are deployed on a cluster of twenty machines, each containing one P100 GPU connected via PCIe 3.0 x16; the machines have Intel Xeon E5-2640 v3 running at 2.6GHz and 32GB of DRAM. The machines are connected via 20Gb Ethernet. V100 GPUs are installed on a public cloud (Amazon AWS). The instance types for the evaluation are shown in Table 2, which summarizes the cluster settings. We used the two private clusters described earlier and two public clusters on AWS. The AWS instances have Intel Xeon E5-2686 v4 running at 2.3GHz. The instances in Pub-A cluster have 244GB of DRAM and four V100 GPUs and the instances in Pub-B cluster have 488GB DRAM and eight V100 GPUs. We used up to forty eight V100 GPUs in Pub-A cluster and up to thirty six V100 GPUs in Pub-B cluster.

Table 2. GPU cluster settings.

Cluster Name	Instance Type	GPUs (# per node×node #)	Interconnect inter-GPU, inter-node
Priv-A	Private Cluster	Titan XP (1×8)	PCIe, 10Gb
Priv-B	Private Cluster	P100 (1×20)	PCIe, 20Gb
Pub-A	AWS p3.8xlarge	V100 (4×12)	NVLink, 10Gb
Pub-B	AWS p3.16xlarge	V100 (8×5)	NVLink, 25Gb

Other training settings. We train with the batch sizes that are used by the authors of the models; we also tested with the maximum batch sizes on the GPUs. For DenseNet, MobileNet, and ResNet, we set the batch size per GPU to be 32, 64, 96, and 128 for CIFAR100 and ImageNet [29, 33, 34]. The maximum global batch size that we used is 6,144 for ResNet-50 with 48×V100 GPUs. To evaluate BERT and GPT-3, we set the batch size to be 96 for the fine-tuning [15, 43, 58, 72]; for the pre-training, we set the batch size to be 512–1872 for BERT and 96–216 for GPT-3, which is similar to commonly used batch size for pre-training the models [23, 30, 48, 58, 59, 61].

We trained the models with multiple optimizers (SGD, momentum, RMSProp, and Adam optimizers) and report the throughput with momentum optimizer as training with other optimizers show similar trend. For BERT and GPT, we use Adam optimizer which is used by its authors [9, 18]. To measure the training throughput, we start the training and wait a few epochs for the training to warm up. Then we measure the throughput by taking the average over ten iterations; we repeat this ten times and report the average and the standard error of the ten runs. For the memory evaluation, we set TensorFlow's `allow_growth` flag as `true` to compactly allocate the required memory. We used `nvidia-smi` to measure the memory usage; we also investigate and report the memory allocation of TensorFlow's `bfc_allocator`.

8.2 Evaluation of Single-GPU Training

We first measured the speedup in single-GPU training brought by multi-stream out-of-order computation and pre-compiled kernel issue. As the optimizations may incur memory overhead, we also report the additional memory usage.

Training throughput. We measure the throughput of training DenseNet, MobileNet, and ResNet in Table 1 with CIFAR100 and ImageNet. We evaluate TensorFlow 2.4 XLA (the baseline) and XLA with our two optimizations. For comparison we also evaluate Nimble [41], a state of the art deep learning execution engine based on PyTorch's JIT compiler; Nimble is reported to largely outperform PyTorch [55], TorchScript [8], TVM [12], and TensorRT [6]. We also tested a simple optimization of running two gradient computations of a layer concurrently (and using barrier synchronization after the two computations of each layer), but it does not improve the performance much (up to 3% speedup) and may even degrade the performance; we do not show the results as they are similar to previous results [4] (also due to space).

Figure 7 shows the throughput of the models on NVIDIA V100 normalized by those of the baseline (XLA). The numbers above the x-axis are the actual throughput (images per seconds). For the models and batch sizes in the figure, our optimized training (denoted by OOO-XLA) improves the throughput of XLA by 1.09–1.21× for DenseNet-121, by 1.07–1.19× for MobileNet, and by 1.03–1.06× for ResNet. The maximum performance gain by OOO-XLA over XLA is (not shown in the figure) 1.54× for DenseNet-121 (k=12 and batch=32) and 1.58× for MobileNet (α=0.25 and batch=32). OOO-XLA outperforms Nimble by up to 1.35× (DenseNet: k=12 and batch=32, not shown) and minimum 1.07× (DenseNet: k=24 and batch=32); Nimble ran out of memory for most models with 64 batches (denoted by N/A). When we examined the speedup by multi-stream ooo computation separately, it gives minimum 1–2% (ResNet-{50,101}, batch=64) and maximum 15% (MobileNet: α=0.25 and batch=32, not shown) speedup over XLA with our pre-compiled kernel issue.

With 128 batch size, Nimble ran out of memory for all the models; XLA and OOO-XLA ran out of memory for most of the DenseNet and ResNet models. For MobileNet, OOO-XLA runs 1.04–1.09× faster than XLA with 128 batches. In Titan XP, the three systems ran out of memory for all the models with 128 batch sizes; with 32 and 64 batch sizes, the performance gain of OOO-XLA is similar to that of V100.

We observed that applying multi-stream ooo computation without weight gradient computation re-ordering (and without barrier synchronization after each layer) often gives a decent speedup. For DenseNet-121, for example, it gives 1.39× speedup over XLA, while fully applying multi-stream ooo computation (with re-ordering) gives 1.54× speedup. This makes the technique very pragmatic, as it can be simply applied without multi-region joint scheduling to achieve a decent speedup; to further improve the performance we can apply the scheduling algorithm.

In summary, OOO-XLA runs 1.03–1.58× faster than XLA over all GPUs and models (average 1.18×). Compared to Nimble, OOO-XLA runs 1.0–1.55× faster (average 1.28×). Note that XLA is already well optimized running 1.1–3.1× faster than TensorFlow in our evaluation. OOO-XLA further improves the performance by up to 58%, running 1.2–4.9× faster than TensorFlow (results not shown due to space).

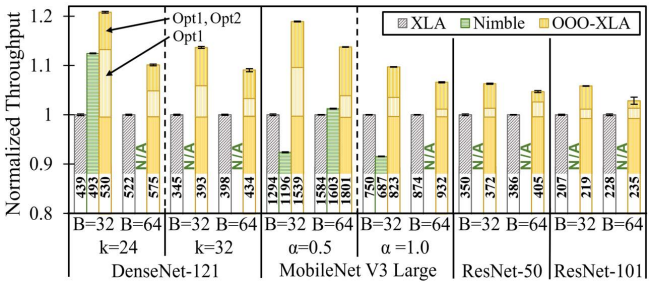

Figure 7. Training throughput normalized by that of XLA for batch size 32, 64. Opt1 is pre-compiled kernel issue; Opt2 is multi-stream ooo computation. The numbers above x-axis are the actual throughput. N/A means out of memory error.

Memory overhead. In all the experiments for single-GPU training, we set the memory constraint to be 1.1× of the conventional execution; i.e., OOO-XLA may use 10% more memory compared to the baseline. For all the models, however, the peak memory usage is increased by less than 0.1%. The maximum increment is for DenseNet-121, for which the weight gradient computations in DenseBlock-4 are delayed to run with the forward computations in DenseBlock-1 as shown in Figure 8. For the delayed computations, the memory pressure is increased, but not by much because the intermediate tensors in DenseBlock-4 are relatively small.

This memory overhead is illustrated in Figure 9, which compares the memory usage of conventional backpropagation (green line) and our multi-stream ooo computation (yellow line). The figure aligns the memory usage by the output gradient computations; that is, the memory usage of m at layer L_i means that the execution uses temporary memory of m when computing the output gradient of L_i. Due to the delayed computations of DenseBlock-4, our execution uses maximum 200MB more memory than the conventional execution. However, the peak memory is only increased by 10MB (or 0.1%) in the beginning of the backpropagation as shown in the figure. For the other models, the weight gradient computations run concurrently with the corresponding output gradient computations in the same region, hence no additional memory is required.

Discussion. The kernel issue overhead is eliminated by the pre-compiled kernel issue technique; then multi-stream ooo computation further reduces the kernel execution overhead and the idling SMs during kernel executions. We studied the effect of multi-stream ooo computation in more detail. Specifically we look into the training of DenseNet-121 and MobileNet where the speedup by multi-stream computation is the highest. As shown in Figure 8, the weight gradient computations in DenseBlock-1–4 are reordered and executed in sub-stream. We examine the execution of the region R2 and R5 in the figure because multi-stream computation gives the minimum (6%) and maximum (10%) speedup for the two regions respectively.

For R2, the output gradient (δO) kernels in DenseBlock-3 are running with the weight gradient (δW) kernels in DenseBlock-3. More than thirty percent of the δO kernels in R2 have the same number of thread blocks as the SM capacity, hence the SMs are running at their maximum thread block capacity. Hence, the performance gain by having the sub-stream kernels is limited to reducing the kernel execution overhead. When we sum up this overhead in R2, it totaled to about 5% of R2's execution time; this is similar to the 6% speedup achieved by multi-stream computation.

In contrast, the main-stream kernels in R5 have much larger number of thread blocks than the SM's capacity; for the sub-stream kernels (δW in DenseBlock-4), half the kernels run with the same 448 thread blocks even though the SMs are capable of running 1,520 of them. By running those

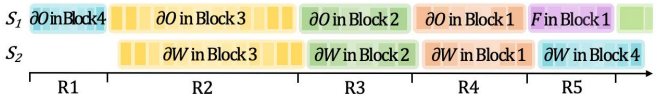

Figure 8. Training DenseNet-121 with main-stream (S_1) and sub-stream (S_2). The kernels of the same DenseBlock are encoded with the same color. R1–R5 are the scheduling regions, which are mapped to the DenseBlocks.

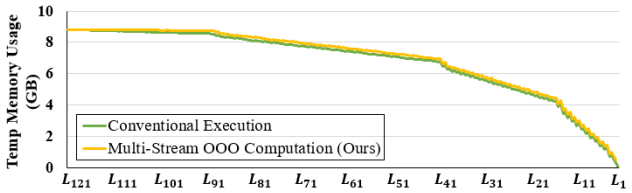

Figure 9. The memory overhead for the backpropagation of training DenseNet-121.

δO and δW kernels concurrently, we provide the opportunity to make most of the SM resources and achieve 10% speedup.

Note that even when the batch size is large, a subset of the kernels in a CNN can have much smaller number of thread blocks than the SM's capacity. This is because the convolutions in early layers of recent CNN models, such as DenseNet or MobileNet, have different characteristics than those in later layers, as shown in Figure 1 and 2.

8.3 Evaluation of Data-Parallel Training

Now we evaluate reverse first-k scheduling algorithm. We use three GPU clusters for the evaluation – a cluster of 8×Titan XP, a larger cluster of 20×P100, and a public cloud of 48×V100 (Pub-A). The settings of the clusters are described in Table 2. For the evaluation of data-parallel training, we trained DenseNet, MobileNet, and ResNet with ImageNet. We measured the performance improvement and memory overhead of reverse first-k scheduling. The baseline for this evaluation is BytePS, the state of the art parameter-communication system for distributed training [38]. BytePS is implemented by the authors of ByteScheduler [56] to further improve its performance; BytePS is reported to be faster than other communication prioritization systems in previous experiments that directly and indirectly compared their performance [1, 28, 36, 38, 44, 56]. We implemented our scheduling optimization on BytePS and measured its training throughput with and without our reverse first-k scheduling. For a subset of the experiments, we also evaluate Horovod, an efficient framework for decentralized distributed training. Our algorithm also improved the performance of Horovod, but the results are similar to the BytePS case and thus we do not show them here (also due to space limit). We do not evaluate the performance of other asynchronous training algorithms [31, 47, 75] as they change the training semantics. Note that ooo backprop can be applied to improve the performance of those asynchronous algorithms.

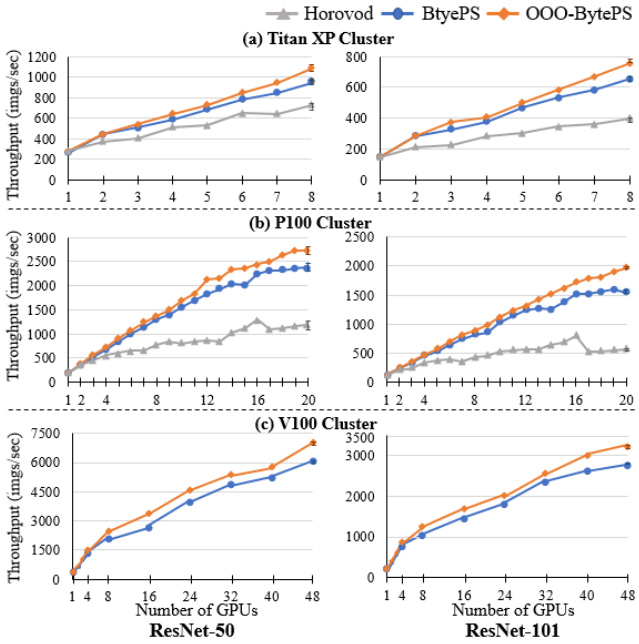

Figure 10. Throughput for the data-parallel training.

Training throughput. The results of the evaluation for the ResNet models are shown in Figure 10; (a) is the results for Titan XP cluster, (b) is for P100 cluster, and (c) is for V100 cluster on AWS. The figure shows the training throughput with 1 to 8, 20, or 48 GPUs for each cluster. For Titan XP and P100 cluster, we used the maximum 64 per-GPU batch size commonly for the two models; the maximum global batch size is 512 for Titan XP cluster and 1280 for P100 cluster. For V100 cluster, we used 128 per-GPU batch for ResNet-50 and 96 per-GPU batch for ResNet-101; the maximum global batch size is 6144 for ResNet-50 and 4608 for ResNet-101. In Titan XP cluster, our reverse first-k scheduling (denoted by OOO-BytePS) is up to 15.3% faster than BytePS and up to 89% faster than Horovod (maximum speedup for ResNet-101 on 8 GPUs). In P100 cluster, OOO-BytePS is up to 27.1% faster than BytePS and 3.5× faster than Horovod both for ResNet-101 on 20 GPUs. In V100 cluster, we achieve up to 26.5% speedup for ResNet-50 on 16 GPUs and up to 19.8% speedup for ResNet-101 on 8 GPUs. On 2–4 GPUs, i.e., when all the GPUs are connected via NVLink, OOO-BytePS runs 1–5% faster than BytePS. In summary, OOO-BytePS achieves 1.1–1.27× speedup over BytePS with 16–48 GPUs on P100 and V100 clusters. For smaller models (DenseNet and MobileNet), we achieved up to 10% and 5.3% speedup respectively.

Memory overhead. Reverse first-k scheduling may increase the memory usage for delaying k-1 weight gradient computations. However, because the memory pressure is already reduced by running the gradient computations of last L-k layers (L is the total number of layers), the reordering does not typically increase the peak memory usage. In all our experiments, the additional memory overhead by reverse first-k scheduling is less than 1% of the total memory usage.

Discussion. To understand the performance gain of reverse first-k scheduling we examine the training of ResNet-50 with 16×V100 GPUs in Pub-A cluster. The performance bottleneck is the first layer's weight synchronization, which is required at the beginning of the forward computation. With sixteen GPUs, BytePS takes 350ms for this synchronization. Thanks to the communication prioritization, the latency is much shorter than those reported by others using similar network settings [39], but it is still substantially large compared to the computation time of 380ms. Our scheduling algorithm reverses first 45 layers' weight gradient computations to overlap δW_1's synchronization with the δW_2–δW_{45}'s computations. The execution time of δW_2–δW_{45} is 85ms, which reduces the 350ms communication time to 265ms. In addition, scheduling δW_2–δW_{20} early in the timeline makes it possible for their synchronization to overlap with part of the backward and forward computation, the effect of which is 65ms of more overlapping. Thus the communication overhead is reduced to 200ms, achieving total 27% speedup.

8.4 Evaluation of Pipeline-Parallel Training

We evaluated pipeline-parallel training with RNN, BERT, and GPT-3. We evaluated their fine-tuning and pre-training. Fine-tuning is evaluated on four V100 GPUs; pre-training is evaluated on up to thirty six V100 GPUs (Pub-B in Table 2).

We measured the speedup of our two optimizations, i.e., gradient fast-forwarding and modulo allocation. The baseline of the experiments is GPipe, which is a state of the art pipeline-parallel training system. We also evaluate PipeDream [51], another leading edge pipeline-parallel training system. PipeDream, however, applies *weight stashing* that brings about parameter staleness and thus changes the semantics of the training. Hence we report its performance as reference points only. For a subset of the experiments, we evaluated DAPPLE [21] and Megatron 2 [54], the state of the art data- and model-parallel training systems. We do not evaluate FT-Pipe [20] (another pipeline-parallel system), as it is designed for the fine-tuning on commodity GPUs; also it failed to run in different settings than their evaluated ones [3].

8.4.1 Fine-Tuning Experiments. We run the fine-tuning of RNN and BERT-24 on four V100 GPUs that are interconnected via NVLink. The following four settings are mainly evaluated: a) cross-layer model parallelism, b) GPipe, c) OOO-Pipe1 (GPipe with gradient fast-forwarding), and d) OOO-Pipe2 (OOO-Pipe1 with modulo allocation). We also report the performance of PipeDream for a subset of the experiments. We use the batch size of 1024 for RNN and 96 for BERT, which is commonly used for their fine-tuning [15, 43, 58, 72]. When applying modulo allocation for RNN and BERT-24, we assign i'th cell and encoder to GPU$_{i \bmod 4}$ respectively.

Figure 11 (a) shows the results. The x-axis is the execution a–d for RNN and BERT, and y-axis is the throughput normalized by that of the single-GPU training. The numbers on the x-axis are the actual throughput (sequences/second). For the RNN model, OOO-Pipe2 runs 1.99× faster than the baseline (GPipe); compared to (cross-layer) model parallelism, OOO-Pipe2 is 1.47× faster. For BERT, OOO-Pipe1 runs 1.15× faster than the baseline and together with modulo allocation OOO-Pipe2 is 1.59× faster. Compared to the single-GPU training of BERT, we achieve 3.2× speedup with four GPUs.

We observed that GPipe is slower than (cross-layer) model parallelism for the RNN model. The reason for this is as follows. Each cell in RNN has two kinds of computations, one that depends on the output of the previous cell and another that does not. Having this *state independent* computations, training with cross-layer model parallelism achieves fairly high GPU utilization. However, applying micro-batch degrades the performance for two reasons in our analysis. First, it increases the amount of the state independent computation in each cell, which interferes with the state dependent (performance-critical) computations. Second, because of the smaller task sizes, the level of parallelism decreases, which also negatively affects the performance. Our optimizations are applied without micro-batch for the RNN model. Because our techniques prioritize critical computations, they speed up the training without causing the interference problem.

To better understand the performance impact of the two optimizations, we also evaluate the training of a simple feed forward neural network (FFNN) with 16 fully-connected layers. When applying the same set of optimizations, we obtain the experimental results in Figure 11 (a) denoted by FFNN. With the two optimizations OOO-Pipe2 runs 1.5× faster than the baseline. To study the effect of the two optimizations for FFNN, we illustrated the execution timelines with and without the optimizations. For simplicity we assume that all the computations take the same amount of time and the inter-GPU communication latency is negligible. We show the execution timelines for FFNN with 8 layers for the interest of space, but we report our analysis with the 16-layer FFNN. Figure 12 shows the execution timelines (of 8-layer FFNN); (a) is GPipe, (b) is OOO-Pipe1 (gradient fast-forwarding), and (c) is OOO-Pipe2 (OOO-Pipe1 with modulo allocation). The numbers denote the layer index of the computation and the subscript (A,B,C, and D) denotes the index of micro-batches. For the 16-layer FFNN model, gradient fast-forwarding gives 1.22× speedup over GPipe and together with modulo allocation our execution is 1.62× faster than GPipe. Compared to these, our experimental results yield reduced speedup (1.18× and 1.5×) because of the inter-GPU communication overhead and non-uniform kernel execution times.

Communication overhead. We run another set of experiments to study the communication overhead incurred by modulo allocation. We trained BERT-24 on four V100 GPUs in three different interconnect networks: NVLink (50GB),

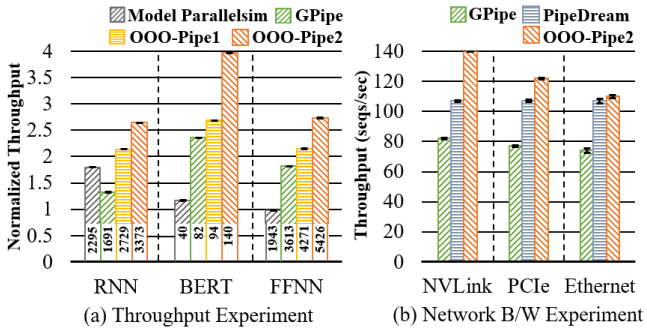

Figure 11. Performance of pipeline-parallel training on 4×V100; (a) is the throughput for RNN, BERT, and FFNN normalized to that of single-GPU training; (b) is BERT-24. The numbers above x-axis are the actual throughput (seqs/sec).

PCIe 3.0 (16GB), and 10Gb Ethernet (1GB). In these networks, we first measured communication to computation ratio of BERT when applying modulo allocation at a transformer level; the ratio is 0.05 (NVLink), 0.16 (PCIe), and 1.8 (Ethernet). For NVLink and PCIe, the communication cost is mostly masked by the computation, except that the computations with very first micro-batches in each GPU (e.g. 2_A, 3_A, and 4_A in Figure 12) are delayed by the communication overhead. For 10Gb Ethernet, however, the communication cost is larger than the computation, and thus the masking effect is much limited; the overall performance is bounded by the communication overhead. Thus in 10Gb Ethernet, we group two consecutive transformers and apply modulo allocation at this group level to reduce the communication overhead at the cost of increased pipeline stalls. With this setting, we evaluated the performance of OOO-Pipe2 in the three interconnect networks and compared to that of GPipe and PipeDream. Figure 11 (b) shows the results. In all three networks, OOO-Pipe2 largely outperforms GPipe by 70% in NVLink, by 58% in PCIe, and by 48% in Ethernet. Note that in 10Gb Ethernet the throughput of OOO-Pipe2 is reduced by half if we apply modulo allocation at a transformer level. Applying modulo allocation at a coarse granularity reduced the communication overhead and improved the performance.

Moreover, when training large NLP models, the typical cluster setting is that a small number of GPUs in a node are interconnected in a high-bandwidth network and the nodes are interconnected in a low-bandwidth network. In such settings, if we apply modulo allocation at a fine granularity (e.g. at a transformer level), the communication to computation ratio is low at the GPU level (for the high-bandwidth network) and also low at the node level (for the combined computations by multiple GPUs). Thus the performance improvements of OOO-Pipe2 is closer to the NVLink case than the Ethernet case, as we show in the pre-training experiments.

Memory overhead. Gradient fast-forwarding incurs memory overhead for storing the input tensors of the delayed computations. When training the NLP models on four V100

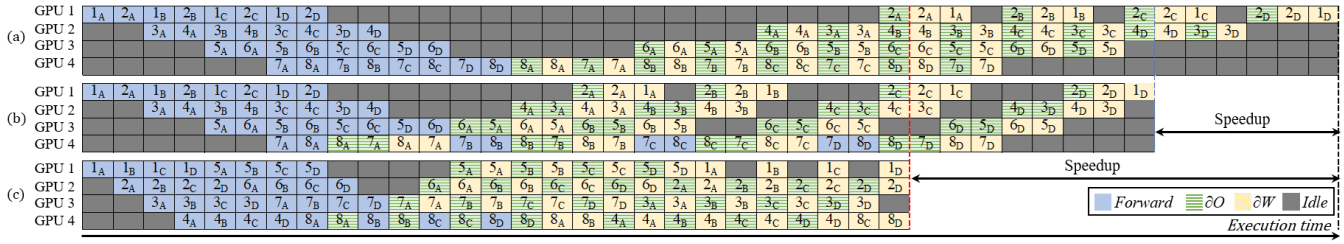

Figure 12. Execution of pipeline-parallel training of 8-layer FFNN; (a) is GPipe, (b) is OOO-Pipe1 (gradient fast-forwarding), and (c) is OOO-Pipe2 ((b)+modulo allocation). The numbers are the layer index and the subscript is the micro-batch index.

GPUs, the maximum memory overhead incurred by the fast-forwarding is 11% over the baseline for BERT. However, modulo allocation eliminates the memory overhead by immediately transferring and discarding the computed outputs, thus using similar amount of memory as the baseline.

8.4.2 Pre-Training Experiments. We evaluate the pre-training of BERT-{12,24,48} and GPT-3 (Medium, 24 decoders) on a larger cluster of 36×V100 (Pub-B in Table 2). We set the max sequence length for BERT be 128 and that for GPT-3 be 512 for the pre-training [18, 52]. We used the vocabulary size of 30,522 for the BERT models and 50,257 for GPT-3, which is commonly used for these models [43, 48, 58]. As we are interested in the scalability of the training with increasing number of GPUs, we perform both weak scaling and strong scaling experiments. In the weak scaling experiments, we train with 8–32 GPUs and with BERT-12–BERT-48. We evaluated three systems: GPipe, PipeDream, and OOO-Pipe2 (ours). For this evaluation, we use the batch sizes of 512–1872 that give the maximum performance for each system.

Figure 13(a) shows the results of the weak scaling experiments. With eight GPUs, the transformers are all assigned to the GPUs on the same machine, which are interconnected via NVLink. In this case, OOO-Pipe2 is 1.73× faster than GPipe and 1.63× faster than PipeDream. With 16 to 32 GPUs, we trained larger models (BERT-24 to BERT-48) and the speedup by OOO-Pipe2 is 41-45% over GPipe and 14-25% over PipeDream. When we increased the number of GPUs from 16 to 32, the performance gain of OOO-Pipe2 does not decrease but it either remains similar (compared to GPipe)

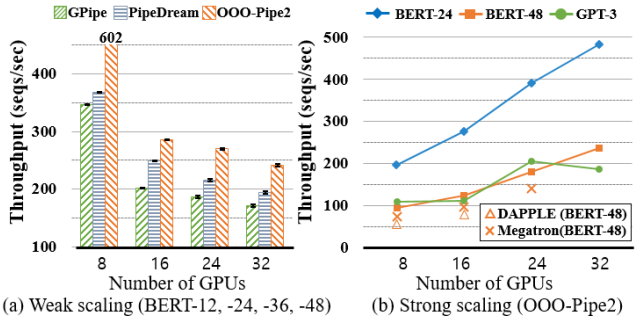

Figure 13. Scalability of pipeline-parallel training.

or increases (compared to PipeDream). This demonstrates that the performance gain by modulo allocation outweighs the overhead of the increased communication for the evaluated cluster setting. For training BERT-48 in PipeDream, we set the maximum number of parameter versions to be 32 as it gives the maximum training throughput. Training with high staleness level of 32 may negatively affect the learning efficiency and slow down the convergence [14**?**].

Now we describe the strong scaling experiments for OOO-Pipe2. We trained BERT-{24,48} with 8–32 GPUs in the same way as the previous experiments. For GPT-3, the size of its last embedding layer is large due to its large sequence length and vocabulary size, and thus we separately assign four GPUs to the layer, which is split in the output neuron dimension; the GPUs compute the last word embedding and the first embedding lookup operations. Figure 13 (b) shows the results. The performance of training BERT-24 and BERT-48 scales fairly well, with the number of GPUs for the training increasing from 8 to 32; with 4× GPUs, the throughput for the two BERT models increased by 2.5×.

For comparison we also evaluated DAPPLE for the pipeline-parallel training of BERT-48 with 8 and 16 GPUs; DAPPLE is previously evaluated with 16 GPUs [21] and their published system does not support larger GPU clusters out-of-the-box [2]. With 8 and 16 GPUs, OOO-Pipe2 is 1.47× and 1.29× faster than DAPPLE respectively. When we applied both data- and pipeline-parallel training to DAPPLE and OOO-Pipe2, the performance of the two systems similarly improved by 30–35% (we leave hybrid-parallel training with ooo backprop as future work). We also evaluated Megatron 2 for BERT-48 with 8, 16, and 24 GPUs; Megatron failed to run with 32 GPUs as it requires that the number of transformers is divisible by the number of GPUs. Compared to Megatron 2, the performance of OOO-Pipe2 is 13.6–29.2% faster. Moreover, when we solely apply gradient fast-forwarding to Megatron 2, its performance is improved by average 20.4% and maximum 27.5%. For GPT-3, we used 12 to 36 GPUs with the extra 4 GPUs for the word embedding layer. The performance for GPT-3 scales in a limited manner because it consists of 24 transformer decoders. That is, the 24 transformers cannot be evenly assigned to 16 or 32 GPUs, hence its scalability from 8 to 16 or from 24 to 32 GPUs is limited.

9 Related Work

Optimization of single-GPU training. Kernel fusion reduces the execution overhead of short-running kernels by combining multiple consecutive kernels into a single one. The technique is applied by optimizing compilers such as XLA [5, 12, 49, 65]. In our evaluation, we show that our optimizations can be applied to XLA to further speedup the performance. Running two gradient computations of a layer concurrently in a GPU or TPU is recently proposed [4, 68]; but without executing them in an out-of-order manner, the performance impact is very limited as we discussed in our evaluation. The technique of pre-compiling a group of kernel issues has been recently proposed to reduce the kernel issue overhead [11, 41, 49]. Particularly, Nimble [41] also applies multiple GPU streams for the neural network models that have parallel blocks such as Inception blocks [62] to compute those blocks in parallel. However, Nimble (and other systems supporting multiple GPU streams [19, 53, 73]) do not concurrently execute weight and output gradient computations in an out-of-order manner as our proposed techniques.

Data-parallel training. Hao et al. proposed wait-free backpropagation for data-parallel training [74], which overlaps the parameter synchronization of a layer with the prior layer's gradient computations. More recently, the technique of prioritizing parameter communication has been proposed to improve the performance by giving higher priority to the parameter communications in critical path [28, 36, 38, 44, 56]. Out-of-order backprop further reorders the critical gradient computations as well as the critical synchronizations to more efficiently overlap the gradient computations with their communications. Our scheduling algorithm based on out-of-order backprop largely outperforms BytePS that prioritizes the parameter communications [38].

A number of studies proposed relaxed schemes for parameter synchronization [31, 47, 75]. For example in AD-PSGD, a worker synchronizes its parameters with only one other worker in each training iteration [47]. While these techniques reduce the synchronization overhead, they are likely to incur accuracy loss in practice [27, 71]. Our optimizations do not change the semantics of the training and thus they can be applied without incurring accuracy loss.

Pipeline-parallel training. For the training of large neural networks that do not fit in a single GPU, pipeline-parallel training is proposed [20, 21, 35, 45, 51, 54]. The systems commonly apply the micro-batch technique that splits a mini-batch into multiple micro-batches to pipeline their executions. Our optimizations are applied on top of the micro-batch technique to further improve the training performance, as we show in our evaluation. Megatron 2 [54] recently proposed *interleaved pipeline scheduling*, which assigns multiple groups of contiguous layers to a GPU instead of one group of contiguous layers. This is similar to our modulo allocation to some extent, but without ooo backprop (i.e., gradient fast-forwarding) their scheduling technique has very limited performance impact because of the increased communication overhead. Because modulo allocation is applied together with the fast-forwarding, it can effectively improve the overlapping of the weight and output gradient computations. In other words, gradient fast-forwarding makes it better for performance to partition and assign neural network layers in an extremely fine-grained manner, which resulted in our modulo allocation. PipeDream [51] and FTPipe [20] support asynchronous pipeline-parallel training, which incurs staleness and thus changes the training semantics similarly as asynchronous data-parallel training.

10 Conclusion

This paper proposes out-of-order backprop, an effective scheduling technique for neural network training. By exploiting the dependencies of gradient computations, ooo backprop reorders gradient computations to speed up neural network training. We proposed scheduling algorithms based on ooo backprop for single-GPU, data-parallel, and pipeline-parallel training. In single-GPU training, we schedule with multi-stream ooo computation to mask the kernel execution overhead. In data-parallel training, we reorder the gradient computations to maximize the overlapping of computation and parameter communication. In pipeline-parallel training, we prioritize the execution of gradient computations to reduce the pipeline stalls; we also apply modulo allocation, an optimized layer allocation policy. We implemented ooo backprop and our scheduling algorithms in TensorFlow and BytePS and evaluated the performance with twelve neural networks. Compared to the respective state of the art training systems, our algorithms improve the training throughput by 1.03–1.58× for single-GPU training, by 1.10–1.27× for data-parallel training, and by 1.41–1.99× for pipeline-parallel training.

Acknowledgement

This work is supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2013-0-00109, WiseKB: Big data based self-evolving knowledge base and reasoning platform, No.2020-0-01373, Artificial Intelligence Graduate School Program (Hanyang University), No.2021-0-00310, Development of SW Framework for Server to Improve AI Training/Inference Efficiency, and No.2021-0-01817, Development of Next-Generation Computing Techniques for Hyper-Composable Datacenters), and by the "Research on the Optimization of the Distributed Training of Large-Scale NLP Models" project funded by KT (KT award B210001432). We thank Gunjoo Ahn for the preliminary experiments and anonymous reviewers for their feedback. Jiwon Seo is the corresponding author who mainly designed the scheduling algorithms and conceived the concept of out-of-order backprop.

References

[1] BytePS github repository. URL: https://github.com/bytedance/byteps.

[2] DAPPLE github repository. URL: https://github.com/AlibabaPAI/DAPPLE.

[3] FTPipe github repository. URL: https://github.com/saareliad/FTPipe.

[4] MXNet dual stream convolution. URL: https://github.com/apache/incubator-mxnet/pull/14006.

[5] NVIDIA Corporation, FasterTransformer. URL: https://github.com/NVIDIA/FasterTransformer.

[6] NVIDIA Corporation, TensorRT. URL: https://developer.nvidia.com/tensorrt.

[7] The repository for ooo backprop experiments. URL: https://github.com/mlsys-seo/ooo-backprop.

[8] Torchscript. URL: https://pytorch.org/docs/stable/jit.html.

[9] 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.

[10] Mauro Cettolo, Jan Niehues, Sebastian Stüker, Luisa Bentivogli, R. Cattoni, and Marcello Federico. The IWSLT 2015 evaluation campaign.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594, 2018.

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

[14] Wei Dai, Yi Zhou, Nanqing Dong, Hao Zhang, and Eric Xing. Toward understanding the impact of staleness in distributed machine learning. In *International Conference on Learning Representations*, 2018.

[15] Zihang Dai, Guokun Lai, Yiming Yang, and Quoc Le. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. In *NeurIPS*, 2020.

[16] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. Number BOOK. McGraw Hill, 1994.

[17] Julien Demouth. CUDA Pro Tip: Minimize the tail effect, NVIDIA Developer Blog, 2015.

[18] 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.

[19] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. Ios: Inter-operator scheduler for cnn acceleration. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 167–180, 2021. URL: https://proceedings.mlsys.org/paper/2021/file/38b3eff8baf56627478ec76a704e9b52-Paper.pdf.

[20] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In *2021 USENIX Annual Technical Conference (USENIXATC '21)*, pages 381–396, 2021.

[21] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: a pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[22] Michael R Garey, David S Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[23] Samuel Gehman, Suchin Gururangan, Maarten Sap, Yejin Choi, and Noah A Smith. Realtoxicityprompts: Evaluating neural toxic degeneration in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3356–3369, 2020.

[24] Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. *http://Skylion007.github.io/OpenWebTextCorpus*, 2019.

[25] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal(BSTJ)*, 45(9):1563–1581, 1966.

[26] Alan Gray. Getting started with cuda graphs. *https://developer.nvidia.com/blog/cuda-graphs/*, NVIDIA Developer Blog, 2019.

[27] Suyog Gupta, Wei Zhang, and Fei Wang. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In *IEEE 16th International Conference on Data Mining (ICDM)*, pages 171–180. IEEE, 2016.

[28] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition(CVPR)*, pages 770–778, 2016.

[30] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention. In *International Conference on Learning Representations*, 2020.

[31] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B Gibbons, Garth A Gibson, Gregory R Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1223–1231, 2013.

[32] Dorit S Hochbaum and David B Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1):144–162, 1987.

[33] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[34] 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*, pages 4700–4708, 2017.

[35] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems(NeurIPS)*, pages 103–112, 2019.

[36] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. *arXiv preprint arXiv:1905.03960*, 2019.

[37] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 947–960, 2019.

[38] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/jiang.

[39] Yunyong Ko, Kibong Choi, Jiwon Seo, and Sang-Wook Kim. An in-depth analysis of distributed training of deep neural networks. *2021 IEEE International Parallel & Distributed Processing Symposium*, 2021.

[40] Krizhevsky et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[41] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. *Advances in Neural Information Processing Systems*, 33, 2020.

[42] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328, 1988.

[43] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2019.

[44] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12).

[45] Shigang Li and Torsten Hoefler. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines. *arXiv preprint arXiv:2107.06925*, 2021.

[46] Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. Performance analysis of gpu-based convolutional neural networks. In *45th International Conference on Parallel Processing (ICPP)*, pages 67–76. IEEE, 2016.

[47] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In *International Conference on Machine Learning (ICML)*, 2018.

[48] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[49] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 881–897, 2020.

[50] Paulius Micikevicius. GPU performance analysis and optimization. *https://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf* , GTC, 2012.

[51] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, 2019.

[52] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. *arXiv preprint arXiv:2006.09503*, 2020.

[53] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. Accelerating deep learning workloads through efficient multi-model execution. In *NeurIPS Workshop on Systems for Machine Learning*, page 20, 2018.

[54] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters. *arXiv preprint arXiv:2104.04473*, 2021.

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 8024–8035, 2019.

[56] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[57] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[58] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[59] Kaitao Song, Hao Sun, Xu Tan, Tao Qin, Jianfeng Lu, Hongzhi Liu, and Tie-Yan Liu. Lightpaff: A two-stage distillation framework for pre-training and fine-tuning. *arXiv preprint arXiv:2004.12817*, 2020.

[60] Mingcong Song, Yang Hu, Yunlong Xu, Chao Li, Huixiang Chen, Jingling Yuan, and Tao Li. Bridging the semantic gaps of gpu acceleration for scale-out cnn-based big data processing: Think big, see small. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 315–326, 2016.

[61] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2158–2170, 2020.

[62] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[63] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. *ACM SIGARCH Computer Architecture News*, 42(3):193–204, 2014.

[64] The TensorFlow Team. Disable multi-stream support in TensorFlow. *https://github.com/tensorflow/tensorflow/blob/v2.5.0/tensorflow/compiler/xla/service/gpu/stream_assignment.cc#L78*.

[65] The XLA team. XLA - tensorflow, compiled. *https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html*, 2017.

[66] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[67] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[68] Nanda K Unnikrishnan and Keshab K Parhi. Layerpipe: Accelerating deep neural network training by intra-layer and inter-layer gradient pipelining and multiprocessor scheduling. *arXiv preprint arXiv:2108.06629*, 2021.

[69] Adina Williams, Nikita Nangia, and Samuel R Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL HLT 2018*, pages 1112–1122. Association for Computational Linguistics (ACL), 2018.

[70] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. IEEE, 2010.

[71] Arissa Wongpanich, Yang You, and James Demmel. Rethinking the value of asynchronous solvers for distributed deep learning. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 52–60, 2020.

[72] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: generalized autoregressive pretraining for language understanding. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages

5753–5763, 2019.

[73] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications. *MLSys' 20*, 2020.

[74] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 181–193, 2017.

[75] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 685–693, 2015.

[76] Keren Zhou, Guangming Tan, Xiuxia Zhang, Chaowei Wang, and Ninghui Sun. A performance analysis framework for exploiting gpu microarchitectural capability. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 1–10, 2017.

A Artifact Appendix

A.1 Summary

We proposed out-of-order backprop (ooo backprop in short) and optimized scheduling algorithms for training deep neural networks. The artifact helps to reproduce the results of the following experiments in the paper:

1. single-GPU training experiments (Section 7.2),
2. data-parallel training experiments (Section 7.3),
3. pipeline-parallel training experiments (Section 7.4).

For these experiments, the throughput is measured and reported both for the baseline execution and for our optimized execution with ooo backprop. The baseline for the experiments are TensorFlow XLA, BytePS, and GPipe respectively.

The artifact of this paper mainly consists of the following.

1. source code of TensorFlow and BytePS,
2. our modifications to TensorFlow and BytePS,
3. source code for training the evaluated models,
4. instructions for compiling the systems and running the experiments,
5. our pre-assembled Docker containers.

The artifact is stored in the github repository: https://github.com/mlsys-seo/ooo-backprop. The doi is https://doi.org/10.5281/zenodo.6345869.

A.2 Requirements

We have prepared and tested the artifact in the following hardware/software settings.

- A Linux machine with kernel 5.4.0-1063-aws (Ubuntu 18.04).
- Intel Xeon E5-2686 v4 (2.3 GHz) and NVIDIA V100 GPU.
- CUDA v11.0 and GPU driver version 450.142.00
- TensorFlow v2.4 and BytePS v0.2.5
- A GPU cluster consisting of V100 GPUs with NVLink inter-GPU interconnect and 10 or 25Gb inter-node interconnect (i.e., AWS p3.8xlarge or p3.16xlarge).
- For simply running the experiments, any Linux machine that runs TensorFlow/BytePS is sufficient.

Table A.1. Summary of the major experiments, their AWS instance types, and the repository location in the artifact.

Experiment	Section	Inst. Type	Repo. Location
Single-GPU	7.2 (Fig. 6)	p3.2xlarge	expr/single_gpu/
Data-Parallel	7.3 (Fig. 8)	p3.8xlarge	expr/data_par/
Pipeline-Parallel	7.4 (Fig. 9 & 11)	p3.16xlarge	expr/pipe_par/

Although CPU or Linux kernel should not affect our optimizations, we suggest to use the above settings to reproduce our experimental results. We have prepared Docker containers that have the above settings. Hence using the containers (via our scripts) is the simplest way to run the experiments.

A.3 Artifact Check-list

Experiments. For the single-GPU experiments, the artifact helps to reproduce the results in Figure 6. We prepared the scripts for training DenseNet-121 and MobileNet V3 on XLA and OOO-XLA and reporting the training throughput.

For the data-parallel experiments, the artifact helps to reproduce the results in Figure 8. We prepared the scripts for training ResNet-{50,101} on BytePS and OOO-BytePs with 1–48 V100 GPUs (i.e., up to twelve AWS p3.8xlarge instances).

For the pipeline-parallel experiments, the artifact helps to reproduce the results in Figure 9 and Figure 11. We prepared the scripts for training BERT-{12,24,36,48} on GPipe and OOO-Pipe2 with 8–32 V100 GPUs (i.e., up to four AWS p3.16xlarge instances). The experiments are summarized in Table A.1.

Repository Structure. The artifact is stored in the github repository: https://github.com/mlsys-seo/ooo-backprop. The structure of the repository is as following.

- TensorFlow/: Source code of TensorFlow (v2.4) modified to (optionally) run with ooo backprop.
- BytePS/: Source code of BytePS (v0.2.5) modified to (optionally) run with ooo backprop.
- expr/: Python scripts for defining and training the evaluated models. Three sub-directories contain the code for the three sets of experiments.
- AWS-doc/: Documentation for setting AWS instances for the experiments.
- scripts/: Bash scripts for running all the experiments.

A.4 Installation and Compilation

We make it possible to run the experiments either on our Docker containers or directly on any modern Linux servers. Running the experiments on our container simply requires git-cloning the repository and running the scripts as described in Section A.6; you can run the experiments on AWS (as described in Section A.5) to reproduce the experimental results in the paper. Because we have compiled TensorFlow and BytePS in the prepared containers, no further compilation is required.

However, running the experiments on the users' Linux servers requires compiling TensorFlow and BytePS after git-cloning the repository. The compilation instructions and the prerequisites are described in https://github.com/mlsys-seo/ooo-backprop#install-guide. Because ooo backprop does not require any prerequisite, only those that are required by TensorFlow and BytePS need to be installed.

A.5 AWS Cloud Settings

We tested the artifact on three AWS instances (i.e., p3.2xlarge, p3.8xlarge, and p3.16xlarge), respectively for the single-GPU, data-parallel, and pipeline-parallel training experiments. To setup an AWS instance for the experiments, one needs to take the follow steps.

1. In the AWS Console, choose to launch Deep Learning AMI (Ubuntu 18.04) Version 56.0.
2. Select the instance type for the experiments, e.g. p3.2xlarge for the single-GPU experiments; see Table A.1.
3. Configure security group for the instances. Open TCP port 1234 for all instances for the worker communication.

The above steps are also described in the AWS-doc/ directory of the repository.

After launching the AWS instances, one needs to install CUDA and compile TensorFlow (and BytePS) to run the experiments. We have prepared Docker containers that are installed with all the requirements for each set of experiments. To use the containers, one simply needs to use the scripts we provided. This is described next in Section A.6.

A.6 Running Experiments

Running on Containers. To run the experiments in our prepared containers, one needs to git-clone the code repository and run the scripts as following.

```
$ git clone https://github.com/mlsys-seo/ooo-backprop.git
$ scripts/single_gpu/single_gpu_densenet_k12_b32_base.sh
```

The above command executes single_gpu_densenet_k12_b32_base.sh, which performs the training of DenseNet (k=12 and batch=32) on the baseline (in this case XLA without ooo backprop) with a single GPU. When the script runs first time, it downloads the container for the single-GPU experiments.

For other experiments, the directory scripts/ contains all the scripts for running the experiments. For the data-parallel and pipeline-parallel experiments, one needs to git-clone the repository and run the script only in the master node (denoted by MASTER_NODE in the scripts which needs to be set up); this is also described in README.md under Quickstart (https://github.com/mlsys-seo/ooo-backprop#quickstart) in the repository. The information of the experiments is specified in the names of the scripts and it is also printed at the beginning of the execution. The scripts also print out the training throughput.

Note that for the pipeline-parallel training with 32 GPUs, it may take up to one hour for TensorFlow to prepare the computation graph (for synchronization, optimization, and compilation). This overhead is caused by TensorFlow itself and not by our implementation; DAPPLE [2] takes similar amount of time for the computation graph preparation.

Running on Custom Linux Machines. After compiling our modified TensorFlow in our repository, the following scripts are used instead to run the experiments.

- Single-GPU Training: expr/single_gpu/scripts/run_*.sh
- Data-Parallel Training: expr/data_par/code/run_node_resnet.sh
- Pipeline-Parallel Training: expr/pipe_par/code/run_node.sh

The above scripts require arguments, which are described at the beginning of the scripts. Other than using these scripts, the steps for running the experiments are the same as that for containers above.