DAPPLE: A Pipelined Data Parallel Approach for Training Large Models

Shiqing Fan  
Alibaba Group  
shiqing.fsq@alibaba-inc.com

Yi Rong  
Alibaba Group  
rongyi.ry@alibaba-inc.com

Chen Meng  
Alibaba Group  
mc119496@alibaba-inc.com

Zongyan Cao  
Alibaba Group  
zongyan.cao@alibaba-inc.com

Siyu Wang  
Alibaba Group  
siyu.wsy@alibaba-inc.com

Zhen Zheng  
Alibaba Group  
james.zz@alibaba-inc.com

Chuan Wu  
The University of Hong Kong  
cwu@cs.hku.hk

Guoping Long  
Alibaba Group  
guopinglong.lgp@alibaba-inc.com

Jun Yang  
Alibaba Group  
muzhuo.yj@alibaba-inc.com

Lixue Xia  
Alibaba Group  
lixue.xlx@alibaba-inc.com

Lansong Diao  
Alibaba Group  
lansong.dls@alibaba-inc.com

Xiaoyong Liu  
Alibaba Group  
xiaoyong.liu@alibaba-inc.com

Wei Lin  
Alibaba Group  
weilin.lw@alibaba-inc.com

Abstract
It is a challenging task to train large DNN models on sophisticated GPU platforms with diversified interconnect capabilities. Recently, pipelined training has been proposed as an effective approach for improving device utilization. However, there are still several tricky issues to address: improving computing efficiency while ensuring convergence, and reducing memory usage without incurring additional computing costs. We propose DAPPLE, a synchronous training framework which combines data parallelism and pipeline parallelism for large DNN models. It features a novel parallelization strategy planner to solve the partition and placement problems, and explores the optimal hybrid strategies of data and pipeline parallelism. We also propose a new runtime scheduling algorithm to reduce device memory usage, which is orthogonal to re-computation approach and does not come at the expense of training throughput. Experiments show that DAPPLE planner consistently outperforms strategies generated by PipeDream’s planner by up to 3.23× speedup under synchronous training scenarios, and DAPPLE runtime outperforms GPipe by 1.6× speedup of training throughput and saves 12% of memory consumption at the same time.

CCS Concepts: • Computing methodologies → Massively parallel algorithms.

Keywords: deep learning, data parallelism, pipeline parallelism, hybrid parallelism

1 Introduction
The artificial intelligence research community has a long history of harnessing computing power to achieve significant breakthroughs [44]. For deep learning, a trend has been increasing the model scale up to the limit of modern AI hardware. Many state-of-the-art DNN models (e.g., NLP[39], search/recommendation systems[13, 45]) have billions of parameters, demanding tens to hundreds of GBs of device memory for training. A critical challenge is how to train such large DNN models on hardware accelerators, such as GPUs, with diversified interconnect capabilities [18, 29, 34].

A common approach is synchronous data parallel (DP) training. Multiple workers each performs complete model computation and synchronizes gradients periodically to ensure proper model convergence. DP is simple to implement and friendly in terms of load balance, but the gradients synchronization overhead can be a major factor preventing linear scalability. While the performance issue can be alleviated by optimizations such as local gradients accumulation [1, 5, 6] or computation and communication overlap techniques [27, 43],
aggressive DP typically requires large training batch sizes, which makes model tuning harder from the perspective of retaining convergence [20].

Recently, pipeline parallelism [24, 36, 52] has been proposed as a promising approach for training large DNN models. The idea is to partition model layers into multiple stages and place them on a set of inter-connected devices. During training, each input batch is further divided into multiple micro-batches, which are scheduled to run over multiple devices in a pipelined manner. Prior research on pipeline training generally falls into two categories. One is on optimizing pipeline parallelism for synchronous (sync) training [24, 52]. This approach requires necessary gradient synchronizations between adjacent training iterations to ensure convergence. At runtime, it schedules as many concurrent pipe stages as possible in order to maximize device utilization. In practice, this scheduling policy can incur notable peak memory consumption. To remedy this issue, re-computation [12, 26] can be introduced to trade redundant computation costs for reduced memory usage. The other category is asynchronous (async) pipeline training [36]. This manner inserts minibatches into pipeline continuously and discards the original sync operations to achieve maximum throughput.

Although these efforts have made good contributions to advance pipelined training techniques, they have some serious limitations. While PipeDream [23] made progress in improving the time-to-accuracy for some benchmarks with async pipeline parallelism, async training is not a common practice in important industry application domains due to convergence concerns. This is reflected in a characterization study [47] of widely diversified and fast evolving workloads in industry scale clusters. In addition, the async approach requires the storage of multiple versions of model parameters. This, while friendly for increasing parallelism, further exacerbates the already critical memory consumption issue. As for sync training, current approach [24] still requires notable memory consumption, because no backward processing (BW) can be scheduled until the forward processing (FW) of all micro-batches is finished. GPipe [24] proposes to discard some intermediate results to free the memory and re-computes them during BW when needed. But it introduces additional re-computation overhead [4].

In this paper, we propose DAPPLE, a distributed training scheme which combines pipeline parallelism and data parallelism to ensure both training convergence and memory efficiency. DAPPLE adopts sync training to guarantee convergence, while avoiding the storage of multiple versions of parameters in async approach. Specifically, we address two design challenges. The first challenge is how to determine an optimal parallelization strategy given model structure and hardware configurations. The target optimization space includes DP, pipelined parallelism, and hybrid approaches combining both. Current state-of-the-art pipeline partitioning algorithm [36] is not applicable for sync training effectively. Some other work [4, 24] rely on empirical and manual optimizations, and lack consideration of some parallelism dimensions. We introduce a sync pipeline planner, which generates optimal parallelization strategies automatically by minimizing execution time of training iterations. Our planner combines pipeline and data parallelism (via stage-level replication) together while partitioning layers into multiple stages. Besides pipeline planning, for those models that can fit into a single device and with high computation/communication ratio, the planner is also capable of producing DP strategies directly for runtime execution.

The second challenge is how to schedule pipeline stage computations, in order to achieve a balance among parallelism, memory consumption and execution efficiency. We introduce DAPPLE schedule, a novel pipeline stage scheduling algorithm which achieves decent execution efficiency with reasonably low peak memory consumption. A key feature of our algorithm is to schedule forward and backward stages in a deterministic and interleaved manner to release the memory of each pipeline task as early as possible.

We evaluate DAPPLE on six benchmarks over three representative application domains (i.e., image classification, machine translation and language modeling). For all benchmarks, experiments show that our planner can consistently produce optimal hybrid parallelization strategies combining data and pipeline parallelism on three typical GPU hardware environments in industry. Besides large models, DAPPLE also works well for medium scale models with relatively large weights yet small activations (i.e. VGG-19).

The contributions of DAPPLE are summarized as follows:

- We systematically explore hybrid of data and pipeline parallelism with a pipeline stage partition algorithm for sync training, incorporating a topology-aware device assignment mechanism given model graphs and hardware configurations. This facilitates large model training and reduces communication overhead of sync training, which is friendly for model convergence.
- We feature a novel parallelization strategy DAPPLE planner to solve the partition and placement problems and explore the optimal hybrid strategies of data and pipeline parallelism, which consistently outperforms SOTA planner’s strategies in sync training scenes.
- We eliminate the need of storing multiple versions of parameters. DAPPLE introduces a pipeline task scheduling approach to further reduce memory consumption. This method is orthogonal to re-computation approach and does not come at the expense of training throughput. Experiments show that DAPPLE can further save about 20% of device memory on the basis of enabling re-computation optimization.
Taking profiling results as input, DAPPLE planner generates an optimized (hybrid) parallelization plan on a given global batch size. Both DAPPLE profiler and planner are offline and can be completed within a few seconds for all our benchmark models (Table 1). Finally DAPPLE runtime takes the planner’s results, and transforms the original model graph into a pipelined parallel graph. At this stage, global batch size is further split into multiple micro-batches and then been scheduled for execution by DAPPLE runtime.

We also explore the mapping of a single stage onto multiple devices. With the replication of pipeline stages on multiple devices, DAPPLE processes training with the hybrid of data and pipeline parallelism. In practice, this hybrid strategy can exploit hierarchical interconnects effectively. Fig. 2 gives an example where a model is partitioned into two stages and each stage is replicated on four devices within the same server(NVLink connections within server), while inter-stage communication goes over the Ethernet. This mapping exploits workload characteristics by leveraging the high-speed NVLink for heavy gradients sync, while using the slow Ethernet bandwidth for small activations communication. We discuss details of our planner in Section 4.

3 DAPPLE Schedule

3.1 Limitations of GPipe Schedule
To improve pipeline training efficiency, GPipe[24] proposes to split global batch into multiple micro-batches and injects them into the pipeline concurrently (Fig. 3 (a)). However, this scheduling pattern alone is not memory-friendly and will not scale well with large batch. The activations produced by forward tasks have to be kept for all micro-batches until corresponding backward tasks start, thus leads to the memory demand to be proportional (O(M)) to the number of concurrently scheduled micro-batches (M). GPipe adopts re-computation to save memory while brings approximately 20% extra computation. In DAPPLE, we propose early backward scheduling to reduce memory consumptions while achieving good pipeline training efficiency (Fig. 3 (b)).

3.2 Early backward scheduling
The main idea is to schedule backward tasks(BW) earlier and hence free the memory used for storing activations produced by corresponding forward tasks(FW). Fig. 3(b) shows DAPPLE’s scheduling mechanism, compared to GPipe in Fig. 3 (a), where the numbers in the cells represent micro-batch ids.

Firstly, instead of injecting all M micro-batches at once, we propose to inject K micro-batches (K < M) at the beginning to release memory pressure while retaining high pipeline efficiency. Secondly, we schedule one FW of a micro-batch followed by one BW strictly to guarantee that BW can be scheduled earlier. Fig. 3 (c) shows how the memory consumptions change over time in GPipe and DAPPLE. At the beginning, the memory usage in DAPPLE increases with time and is the same as GPipe’s until K micro-batches are injected, then it reaches the maximum due to the early BW scheduling. Specifically, with strictly controlling the execution order of FW and BW, the occupied memory for activations produced by the FW of a micro-batch will be freed after the corresponding BW so that it can be reused by the next injected micro-batch. In comparison, GPipe’s peak memory consumptions increases continuously and has no opportunity for early release. Moreover, DAPPLE does not sacrifice in pipeline training efficiency. Actually, DAPPLE introduces the exact same bubble time as GPipe when given the same stage partition, micro-batches and device mapping. We will present the details in section 5.3.

Note that the combination of early backward scheduling and re-combination allows further exploitation in memory usage. We present detailed performance comparisons of DAPPLE and GPipe in Section 6.4.

4 DAPPLE Planner
DAPPLE Planner generates an optimal hybrid parallelism execution plan given profiling results of DAPPLE profiler, hardware configurations and a global training batch size.
A pipeline training iteration consists of three phases, namely warmup phase, steady phase and ending phase. As an example shown in Fig. 4 where the pivot stage is the last stage. Pivot stage dominates steady phase. We call the execution period from the start to pivot stage’s first forward micro-batch as warmup phase in a pipeline iteration, the period from pivot stage’s last backward micro-batch to the end as ending phase. Pipeline latency $L$ is the sum of these three phases. The optimization objective for estimating $L$ is as follows:

$$T_w = \sum_{s=0}^{Q} F_s$$

$$T_s = (M - 1) \times (F_Q + B_Q)$$  \hspace{1cm} (1)

$$T_e = \max_{s=0}^{S-1}(AR(P_s, g_s) + \left\{ \begin{array}{ll} -\sum_{a=Q}^{s} B_a & s > Q \\ \sum_{a=0}^{Q} B_a & s \leq Q \end{array} \right\})$$  \hspace{1cm} (2)

$$L = T_w + T_s + T_e$$

$T_w$ denotes the execution time of warmup phase, which is the sum of forward execution time of stages till $Q$ for one micro-batch. $T_s$ denotes the steady phase, which includes both forward and backward time of the pivot stage $Q$ for all micro-batches except for the one contributing to warmup and ending phase, respectively. $T_e$ corresponds to the ending phase. $T_e$ includes allreduce overhead and thus considers stages both before and after $Q$. Note that some stages before $Q$ may contribute to $T_e$ with allreduce cost. $M, S, F_s$, and $B_s$ denote the total number of micro-batches, the number of stages (computation stages + network stages), forward and backward computation time of stage $s$, respectively. $AR(P_s, g_s)$ represents the gradients synchronization (AllReduce) time for stage $s$, with its parameter set $P_s$ on the device set $g_s$.

Note here we consider inter-stage communication as an independent stage alongside the computation stages, e.g., the Stage 1 and Stage 3 in Fig. 4. The AllReduce time $AR(P_s, g_s)$ is always 0 for these inter-stage communication stages. Moreover, we define $F_s$ and $B_s$ for a communication stages as its following forward and backward communication time.

In practice, synchronous pipelines in some cases include bubbles in the pivot stage $Q$, which may contribute a small fraction of additional delay to the pipeline latency $L$. This objective does not model those internal bubbles, and thus is an approximation to the true pipeline latency. But it works practically very well for all our benchmarks (Section 6).

### 4.2 Device Assignment

Device assignment affects communication efficiency and computing resource utilization. Previous work [36] uses hierarchical planning and works well for asynchronous training. However, it lacks consideration of synchronous pipeline training, in which the latency of the whole pipeline, rather than of a single stage, matters to overall performance. It cannot be used to efficiently estimate the whole pipeline latency.
Meanwhile, it does not allow stages to be placed on arbitrary devices. Our approach essentially allows a specific stage to be mapped to any set of devices, and therefore is able to handle more placement cases, at a reasonable searching cost.

Instead of enumerating all possibilities of placement plans using brute force, we designed three policies (Fig. 5), and explore their compositions to form the final placement plan.

**Fresh First** allocates GPUs from a fresh machine. It tends to put tasks within a stage onto the same machine, which can leverage high-speed NVLink [8] for intra-stage communication. A problem of this policy is that, it can cause fragmentation. It also can leverage high-speed NVLink [8] for intra-stage communication. This policy could also serve as an intermediate state to allocate GPU with minimal fragmentation.

**Append First** allocates from machines that already have GPUs occupied. It helps to reduce fragmentation. It also largely implies that the stage is likely to be within the same machine.

**Scatter First** tries to use available GPUs equally from all used machines, or use GPUs equally from all machines if they are all fresh. It is suitable for those stages that have negligible weights compared to activation sizes (less intra-stage communication). This policy could also serve as an intermediate state to allocate GPU with minimal fragmentation.

The overall device placement policies reduce search space effectively down to less than $O(2^S)$, while retaining room for potential performance gain.

In the Formulation section 4.3, we will use a Pseudo function $D(gids, n)$ to denote the returned results of our device placement policies, when requested $n$ GPUs from the states $gids$.

### 4.3 Planning Algorithm

Our planning algorithm use Dynamic Programming to find the optimal partition, replication and placement strategy, so that the pipeline latency $L$ (as defined in formula 2) is minimized. Here we first present how to update the pivot stage id $Q$ along the planning process, and then the formulation of our algorithm.

**Determining The Pivot Stage $Q$.** It is vital to select a proper pivot stage $Q$ for the estimation of $L$. The insight is to find the stage with minimum *bubbles*, which dominates steady phase. We use a heuristic to determine $Q$.

![Figure 5. Device assignment examples: applying for 6 devices using three different strategies respectively from (a).](image)

![Algorithm 1 Iteratively update Q](image)

**Algorithm 1** Iteratively update $Q$

1. let $Q = S-1$, $s = Q-1$
2. while $s \geq 0$ do
3. \hspace{1cm} let $l_1 = (M - 1) \times (F_x + B_x)$
4. \hspace{1cm} let $l_2 = (M - 1) \times (F_Q + B_Q)$
5. \hspace{1cm} for $s' = s + 1; s' < Q; s' = s' + 1$
6. \hspace{2cm} $l_2 = l_1 + F_x + B_x$
7. \hspace{1cm} end for
8. \hspace{1cm} if $l_1 > l_2$ then
9. \hspace{2cm} $Q = s$
10. \hspace{1cm} end if
11. \hspace{1cm} $s = s - 1$
12. end while

Formula 3 along with Algorithm 1 describe how to update stage $Q$ iteratively from stage $S - 1$ to stage 0. The initial $Q$ is set to $S - 1$. $T^j_{st} = (M - 1) \times (F_x + B_x)$ means the duration of steady phase, without bubbles, suppose pivot stage is $j$. For a stage $s < Q$, if $T^j_{st}$ is larger than the sum of $T^s_{st}$ and corresponding forward/backward costs between the stage $s$ and current stage $Q$, it means the *steady phase* will have less bubbles if pivot stage is set to $s$ other than current $Q$. $Q$ will then be updated to $s$.

**Algorithm Formulation.** We define the estimated pipeline latency $T_{PL}(j, m, g)$ as the subproblem, for which we have planned the strategy for the first $j$ layers using $m$ GPUs (with device id set $g$). The unplanned layers forms the last stage and replicates on the other $(G - m)$ GPUs. Our objective is to solve for $T_{PL}(N, G, \mathcal{G})$, $\mathcal{G} = \{0, 1, \ldots, G - 1\}$. $N$, $G$ and $\mathcal{G}$ denote the number of layers, number of GPUs and GPU set, respectively. Formula 4 describes the algorithm.

$$T_{PL}(N, G, \mathcal{G}) = \min_{1 \leq j < N} \min_{1 \leq m < G \in \mathcal{G}} \min_{\mathcal{G} \in \mathcal{G}} T_{PL}(j, m, g) \quad (4)$$

Our DP algorithm (Formula 4) tries to split the layers into two parts, with the second part being a single stage and recursively partition the first part. For each split, the algorithm enumerates the number of GPUs allocated to the last stage, and use the three strategies (section 4.2) for device placement.
4.4 Contributions over previous work

This section highlights our contributions of planning for hybrid parallelism. The resulting strategies and performance gain on real-world models are demonstrated in Section 6.6.

Uneven Pipeline Partitioning with Fewer Stages. In sync pipeline parallelism scenarios, we find two insights that could provide an additional performance improvements. The first one is to partition the model into as few stages as possible to minimize the bubble overhead under the same number of micro-batches. This conclusion is also mentioned in GPipe. The second one is that partitioning the model in a slightly uneven way yields much higher performance than a perfectly even split, like the example in Fig. 7.

Versatile Device Placement. DAPPLE device assignment strategy covers a broad solution space for stage placement, and is a strict superset of PipeDream’s hierarchical recursive partitioning approach. This allows us to handle various real world models. For example, for models that have layers with huge activations compared to their weights, DAPPLE allows such a layer to be replicated across multiple machines (Scatter First) to utilize high-speed NVLink for activation communication and low-speed Ethernet for AllReduce.

5 DAPPLE Runtime

5.1 Overview

We design and implement DAPPLE runtime in Tensorflow[9] (TF) 1.12, which employs a graph based execution paradigm. As common practices, TF takes a precise and complete computation graph (DAG), schedules and executes graph nodes respecting all data/control dependencies.

DAPPLE runtime takes a user model and its planning results as input, transforms the model graph into a pipelined parallel graph and executes on multiple distributed devices (as shown in Fig. 1). It first builds forward/backward graphs separately for each pipeline stage. Then additional split/concat nodes are introduced between adjacent stages for activation comm. Finally, it builds a subgraph to perform weights update for sync training. In this work, we implement this pipelined graph construction process by manually manipulating computation nodes of user models. This graph transformation can be done automatically, which is left as our future work.

Section 5.2 presents how to build basic Tensorflow[9] graph units for a single micro-batch. Section 5.3 discusses how to chain multiple such units using control dependencies to facilitate DAPPLE execution.

5.2 Building Micro-batch Units

5.2.1 Forward/Backward Stages. In order to enforce execution orders with control dependencies between stages, we need to build forward and backward graphs stage by stage to deal with the boundary output tensors such as activations.

Specifically, we first construct the forward graph of each stage. As common practices, TF takes a precise and complete computation graph (DAG), schedules and executes graph nodes respecting all data/control dependencies.

DAPPLE runtime takes a user model and its planning results as input, transforms the model graph into a pipelined parallel graph and executes on multiple distributed devices (as shown in Fig. 1). It first builds forward/backward graphs separately for each pipeline stage. Then additional split/concat nodes are introduced between adjacent stages for activation comm. Finally, it builds a subgraph to perform weights update for sync training. In this work, we implement this pipelined graph construction process by manually manipulating computation nodes of user models. This graph transformation can be done automatically, which is left as our future work.

Section 5.2 presents how to build basic Tensorflow[9] graph units for a single micro-batch. Section 5.3 discusses how to chain multiple such units using control dependencies to facilitate DAPPLE execution.

5.2.2 Cross Stage Communication. DAPPLE replicates some stages such that the number of nodes running a stage can be different between adjacent stages, and the communication patterns between them are different from straight pipeline design. We introduce special split-concat operations between these stages.

Figure 6. Uneven pipeline minimum example.

Figure 7. Uneven pipeline minimum example.

Figure 8. Efficiency of two-stage replication approaches. Stage0 consumes twice as much time as stage1 for a micro-batch.
Fig. 8(a) shows the replication in DAPPLE for a 2-stage pipeline, whose first stage consumes twice as much time as the second stage for a micro-batch and thus is replicated on two devices. For the first stage, we split the micro-batch further into 2 even slices, and assign each to a device. An alternative approach[36] (Fig. 8(b)) is not to split, but to schedule an entire micro-batch to two devices in round robin manner. However, the second approach has lower pipeline efficiency due to tail effect[15]. Though the second approach does not involve extra split-concat operations, the overhead of tail effect is larger than split-concat in practice. We hence use the first approach with large enough micro-batch size setting to ensure device efficiency.

5.2.3 Synchronous Weights Update. Weights updates in DAPPLE is different with naive training as there are multiple micro-batches injected concurrently. Meanwhile, the replication makes weights updating more complex. As shown in Fig. 9, each device produces and accumulates gradients for all micro-batches. There is an AllReduce operation to synchronize gradients among all replicas, if exists. A normal Apply operation updates weights with averaged gradients eventually.

5.3 Micro-batch Unit Scheduling

The early backward scheduling strikes a trade-off between micro-batch level parallelism and peak memory consumption: feeding more micro-batches into pipeline at once implies higher parallelism, but may lead to more memory usage. DAPPLE scheduler enforces special execution orders between micro-batches to reduce memory usage. For the first stage, we suppose K micro-batches are scheduled concurrently at the beginning for forward computation. Specifically, Kᵢ is the number of scheduled micro-batches at the beginning for stage i. The overall execution follows a round robin order with interleaving FW and BW.

Fig. 10 shows how up to three micro-batches are connected via control dependencies to implement the schedule for a two stage pipeline. Control dependency is not required when there is only one micro-batch (Fig. 10(a)). With two micro-batches (Fig. 10(b)), two control edges (red dotted arrow) are introduced. The situation with three micro-batches (Fig. 10(c)) is similar. An appropriate Kᵢ is essential as it indicates the peak memory consumption for stage i. There are two primary factors for deciding Kᵢ: memory demand for one micro-batch execution, and the ratio between cross stage communication latency and the average FW/BW computation time (referred as activation communication ratio, ACR in short). The former determines how many forward batches can be scheduled concurrently at most.

We implement two policies to set Kᵢ in practice. Policy A (P_A): Kᵢ = min(S – i, D). P_A works well when ACR is small, i.e. the impact of cross stage communication overhead is negligible. Policy B (P_B): Kᵢ = min(2 * (S – i) – 1, D). Here we schedule twice the number of forward micro-batches than P_A. The underlying intuition is that in some workloads, the cross stage communication overhead is comparable with forward/backward computations and thus more micro-batches is needed to saturate the pipeline.

In our experiments, we do observe two workloads (VGG-19 and AmoebaNet-36 in section 6.4), for which P_B works better. In both polices, we keep Kᵢ at O(S). In practice, the number of pipeline stages (S) produced by the DAPPLE planner is typically much smaller than the number of micro-batches (M). This is important to constrain peak memory consumption. A further implication is that DAPPLE encourages relatively large granularity of stage computations, thus achieving decent execution efficiency.

6 Evaluation

6.1 Experimental Setup

Benchmarks. Table 1 summarizes all the six representative DNN models that we use as benchmarks in this section. The datasets applied for the three tasks are WMT16 En-De [42], SQuAD2.0 [40] and ImageNet[41], respectively.

Hardware Configurations. Table 2 summarizes three common hardware environments for DNN training in our experiments, where hierarchical and flat interconnections are both covered. In general, hierarchical interconnection is popular in industry GPU data centers. We also consider flat Ethernet networks interconnections because NVLink may not be available and GPU resources are highly fragmented in
when keeping global batch size fixed, and we have tested all
Normalized training throughput speedup of sched-
Table 4 summarizes

<table>
<thead>
<tr>
<th>Task</th>
<th>Model</th>
<th># of Params</th>
<th>(Profile Batch Size, Memory Cost)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>GNMT-16</td>
<td>291M</td>
<td>(64, 3.9GB)</td>
</tr>
<tr>
<td>Language</td>
<td>BERT-48</td>
<td>640M</td>
<td>(2, 11.4GB)</td>
</tr>
<tr>
<td>Model</td>
<td>XLLNet-36 [50]</td>
<td>500M</td>
<td>(1, 12GB)</td>
</tr>
<tr>
<td>Image</td>
<td>ResNet-50</td>
<td>24.5M</td>
<td>(128, 1GB)</td>
</tr>
<tr>
<td>Classification</td>
<td>VGG-19</td>
<td>137M</td>
<td>(32, 5.6GB)</td>
</tr>
<tr>
<td></td>
<td>AmoebaNet-36</td>
<td>933M</td>
<td>(1, 20GB)</td>
</tr>
</tbody>
</table>

Table 2. Hardware configurations.

<table>
<thead>
<tr>
<th>Config</th>
<th>GPU(s) per server(Ni)</th>
<th>Intra-server connections</th>
<th>Inter-server connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8x V100</td>
<td>NVLink</td>
<td>25 Gbps</td>
</tr>
<tr>
<td>B</td>
<td>1x V100</td>
<td>N/A</td>
<td>25 Gbps</td>
</tr>
<tr>
<td>C</td>
<td>1x V100</td>
<td>N/A</td>
<td>10 Gbps</td>
</tr>
</tbody>
</table>

Table 3. Normalized training throughput speedup of scheduling policies P9 compared to P9.

<table>
<thead>
<tr>
<th>Model</th>
<th>Bert-48</th>
<th>XLLNet-36</th>
<th>VGG-19</th>
<th>GNMT-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1.0</td>
<td>1.02</td>
<td>1.1</td>
<td>1.31</td>
</tr>
</tbody>
</table>

some real-world production clusters. All servers run 64-bits CentOS 7.2 with CUDA 9.0, cuDNN v7.3, NCCL 2.4.2[7] and TF-1.12.

**Batch Size and Training Setup.** The batch sizes of offline profiling for the benchmarks are shown in the last column of Table 1 (profile batch size). As for AmoebaNet-36, it reaches OOM even if \( batch\_size = 1 \) on a single V100. Thus we extend to two V100s where \( batch\_size = 1 \) just works. We use large enough global (GBS) batch size for each benchmark to ensure high utilization on each device. All global batch sizes we use are consistent with common practices of the ML community. Note that all the pipeline latency optimizations proposed in this paper give equivalent gradients for training when keeping global batch size fixed, and we have tested all benchmarks with accuracy vs. epoch result recorded. Results show that DAPPLE can reach target accuracy consistently in a similar number of epochs as Data Parallel, which will not be further discussed for space constraints.

### 6.2 Planning Results

Table 4 summarizes DAPPLE planning results of six models in the three hardware environments, where the total number of available devices are all fixed at 16. The first column also gives the global batch size (GBS) correspondingly.

We use three notations to explain the output plans.

A plan of \( P : Q \) indicates a two stage pipeline, with the first stage and the second stages replicated on \( P \) and \( Q \) devices, respectively. For example, when \( P = 8 \) and \( Q = 8 \), we put each stage on one server, and replicate each stage on all 8 devices within the server(config-A). Besides, for plans where \( P > 8 \) or \( Q > 8 \) (e.g., \( 15 : 1 \)) where some stages are replicated across servers, it will most likely be chosen for configurations with flat interconnections such as Config-B or Config-C, since for Config-A replicating one stage across servers incurs additional inter-server communication overhead.

**A straight plan** denotes pipelines with no replication.

A **DP plan** means the optimal strategy is data-parallel. We treat DP and straight as special cases of general DAPPLE plans.

The **Split Position** column of Table 4 shows the stage partition point of each model for the corresponding pipeline plan. The **ACR** column of the table shows the **averaged** ratio of cross-stage comm latency (i.e. comm of both activations in FW and gradients in BW) and stage computation time.

**ResNet-50.** The best plan is consistently DP for all three hardware configurations. This is not surprising due to its relatively small model size (100MB) yet large computation density. Even with low speed interconnects Config-\( C \), DP with notably gradients accumulation and computation/communication overlap outperforms the pipelined approach.

**VGG-19.** Best plans in config A and B are also DP (Fig. 11 (a) and(b)), due to the moderate model size (548MB), relatively fast interconnects (25 Gbps), and the overlapping in DP. The weights and computation distributions of VGG-19 are also considered overlapping-friendly, since most of the weights are towards the end of the model while computations are at the beginning, allowing gradients aggregation to be overlapped during that computation-heavy phase. In the case of low speed interconnects (Config – C), a 15 : 1 pipelined outperforms DP (Fig. 11 (c)).
Figure 11. Speedups on configurations with hierarchical/flat interconnects.
6.3 Performance Analysis

In this work, we measure training speed-up as the ratio between the time executing all micro-batches sequentially on a single device and the time executing all micro-batches in parallel by all devices, with the same global batch size.

Fig. 11 shows training speed-ups for all models except ResNet-50 on config A, B and C. For ResNet-50, the planning results are obvious and we simply present it in Table 4. For the other models, we compare training speed-ups of three different implementations: (1) Best Hybrid Speedup, performance of the best hybrid plan of pipeline and data parallelism returned by DAPPLE planner; (2) DP No Overlap, performance of DP with gradients accumulation but without computation/communication overlap; (3) DP Overlap, performance of DP with both gradients accumulation and intra-iteration computation/comm. overlap between backward computation and gradients communication[53].

Overall analysis across these five models from Fig. 11, for fixed $GBS = 128$, we can find that the hybrid approaches from DAPPLE outperform the DP approach with best intra-batch overlapping with averaged $1.71x/1.37x/1.79x$ speedup for config-A, config-B and config-C, respectively. Specially, this speedup is up to $2.32x$ for GNMT-16 on config-C. Specific analysis for each model is given below.

VGG-19. For VGG-19, about 70% of model weights (about 400 MB) are in the last FC layer, while the activation size between any two adjacent layers gradually decreases from the first convolution (conv) layer to the last FC layer, varying dramatically from 384 MB to 3 MB for batch size of 32. Thus, the split between VGG-19’s conv layers and FC layers leads to very small activation (3MB), and only replicating all the conv layers other than FC layers greatly reduces communication overhead in case of relatively slow interconnects (Fig. 11 (c)).

GNMT-16. GNMT-16 prefers a two-stage pipeline on hierarchical network (config A) and flat network with relative high-speed connection (config B). And the corresponding split position is $9 : 7$ but not $8 : 8$, this is because the per-layer workloads of encoder and decoder of GNMT are unbalanced (about $1 : 1.45$), thus the split position of DAPPLE plan shifts one layer up into decoder for pursuit of better system load-balance. For low speed interconnection environments (config C), straight pipeline ranks first when $GBS = 1024$. Each device is assigned exactly one LSTM layers of GNMT, and the GBS is large enough to fill the 16-stage pipeline.

BERT-48/ XLNet-36. The best DAPPLE plan outperforms all DP variants for both models (Fig. 11 (g) to (l)) in all configurations. Compared to XLNet, the memory requirement for BERT is much smaller and thus allows more micro-batches on a single device. More computation per-step implies more backward computation time can be leveraged for overlapping comm overhead. As for config B and C, the slower the network is(from 25 Gbps to 10 Gbps), the higher the advantage of our approach has over DP variants. This is because the cross stage communication for both models is negligible with respect to gradients communication and the pipelined approach is more tolerant of slow network than DP.

AmoebaNet-36. The DAPPLE plan works best in all three configurations when $GBS = 128$. Unlike BERT-48 and XLNet-36, AmoebaNet has non uniform distributions of per layer parameters and computation density. The last third part of the model holds 73% of all parameters, and the per-layer computation time increases gradually for large layer id and the overall maximum increase is within 40%. As DAPPLE planner seeks for load-balanced staging scheme while considering the allreduce overhead across replicated stages, the split positions of pipelined staged approach for AmoebaNet-36 will obviously tilt to larger layer ID for better system efficiency.

6.4 Scheduling Policy

As discussed in Section 5.3, the number of successive forward micro-batches ($K_i$ for stage $i$) scheduled in the warm up phase is an important factor to pipeline efficiency. We implement two policies, $P_A$ and $P_B$, referring to smaller and larger $K_i$ numbers, respectively. Table 3 shows the normalized speedups for four benchmark models on hierarchical interconnects(config A), where all models’ stage partition and replication schemes are consistent with the planning results of 2 servers of config A as shown in Table 4.

For VGG-19 and GNMT-16 (as well as AmoebaNet-36, which is not given in this figure yet), where the ACR ratio is relative high ($0.16$, $0.10$, $0.18$, respectively), there exists notable performance difference between these two policies ($10\%, 31\%$ improvement from $P_A$ to $P_B$, respectively). Hence we choose a larger $K_i$ to maximize pipeline efficiency. For the other models (BERT-48, XLNet-36), whose ACRs are very
Table 5. DAPPLE vs. GPipe on BERT-48 with 2-stage pipeline when keeping micro-batch size fixed to 2 on Config-B. RC is short for re-computation.

<table>
<thead>
<tr>
<th>Config</th>
<th># of micro batch (M)</th>
<th>Throughput (samples/sec)</th>
<th>Average Peak Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPipe</td>
<td>2</td>
<td>5.10</td>
<td>12.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OOM</td>
</tr>
<tr>
<td>GPipe + RC</td>
<td>2</td>
<td>4.00</td>
<td>9.9</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5.53</td>
<td>13.2</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>–</td>
<td>OOM</td>
</tr>
<tr>
<td>DAPPLE</td>
<td>2</td>
<td>5.10</td>
<td>10.6</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7.60</td>
<td>10.6</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>8.18</td>
<td>10.6</td>
</tr>
<tr>
<td>DAPPLE + RC</td>
<td>2</td>
<td>4.24</td>
<td>8.5</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>6.23</td>
<td>8.5</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>6.77</td>
<td>8.5</td>
</tr>
</tbody>
</table>

small (0.06, 0.03, respectively), the cross stage communication overhead is negligible compared to intra-stage computation time, leading to little performance difference. In this case, we prefer a smaller $K_t$ to conserve memory consumption.

6.5 Comparison with GPipe

Table 5 shows the performance comparisons with GPipe. We focus on the throughput and peak memory usage on BERT-48 with a 2-stage pipeline in Config-B. To align with GPipe, we adopt the same re-computation strategy which stores activations only at the partition boundaries during forward pass [24]. Note that all the pipeline latency optimizations in DAPPLE give equivalent gradients for training when keeping global batch size fixed, thus convergence is safely preserved and tested, and will not be further analysed here.

When applying re-computation, both DAPPLE and GPipe save about 19% averaged peak memory at the expense of 20% on throughput when keeping $M = 2$ fixed.

When both without re-computation, DAPPLE gets 1.6× higher throughput with $M = 16$, and consumes 0.88× averaged peak memory compared to GPipe, which only supports up to 2 micro-batches. The speedup is mainly because higher $M$ leads to lower proportion of bubbles. Note DAPPLE allows more micro-batches as the peak memory requirement is independent of $M$ due to early backward scheduling.

The combination of DAPPLE scheduler and re-computation allows a further exploitation in memory usage. Compared with baseline GPipe (without re-computation), DAPPLE + RC achieves 0.70× memory consumption when $M = 16$, which allows us to handle larger micro-batch size or larger model.

6.6 Comparison with PipeDream

We compare the results of our planner with those of PipeDream’s under the synchronous training scenarios. We use the same configurations for both planners (e.g. same device topology, same interconnect and same profiling data), and evaluate both planners with DAPPLE Runtime. Table 6 shows the strategy results under a two-machine cluster of config-A. Fig. 12 shows the performance results for the strategies running in both 2 × 8 and 4 × 8 configurations.

6.7 Large Model Scalability

Table 7 shows the maximum model size that DAPPLE supports under reasonable input size with re-computation enabled. We scale the model by varying the numbers of layers. We are able to scale BERT to 5.5B on 8 V100s with NVLink. There is a slight reduction in average GPU utilization due to more bubbles introduced by longer pipeline. In this case, the maximum model size scales linearly due to the balanced distribution of model params over encoder layers in BERT.

7 Related Work

Large DNN models are increasingly computational intensive. It is a common practice to parallelize training by leveraging multiple GPUs[14, 18, 29, 38, 54].

Data parallelism, model parallelism and pipeline parallelism are common approaches for distributed training of DNN models. Note we discuss pipeline parallelism separately from model parallelism.
Table 7. Maximum model size of BERT supported by DAPPLE + re-computation on V100 (16GB each) on config-A. BERT-L: BERT model with L encoder layers. Each model parameter needs 16 bytes since we applied Adam optimizer.

<table>
<thead>
<tr>
<th>Config</th>
<th>BERT-L</th>
<th># of Model Params</th>
<th>Total Model Params</th>
<th>Mem</th>
<th>Avg. GPU Util</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native-1</td>
<td>48</td>
<td>640M</td>
<td>10.2GB</td>
<td>93%</td>
<td></td>
</tr>
<tr>
<td>Pipeline-2</td>
<td>106</td>
<td>1.4B</td>
<td>21.9GB</td>
<td>89%</td>
<td></td>
</tr>
<tr>
<td>Pipeline-4</td>
<td>215</td>
<td>2.7B</td>
<td>43.8GB</td>
<td>89%</td>
<td></td>
</tr>
<tr>
<td>Pipeline-8</td>
<td>428</td>
<td>5.5B</td>
<td>88.2GB</td>
<td>87%</td>
<td></td>
</tr>
</tbody>
</table>

Data Parallelism [32]. Some prior studies [2, 3, 10, 28, 43, 51] focus on reducing the comm overheads for data parallelism. As a commonly used performance optimization method, gradients accumulation[5, 6, 33] offers an effective approach to reduce comm-to-computation ratio. Another complementary approach is computation and comm overlap, with promising results reported in some CNN benchmarks[27, 53].

Model Parallelism. Model Parallelism[30] partitions DNN models among GPUs to mitigate comm overhead and memory bottlenecks for distributed training [11, 14, 16, 19, 23–25, 38, 48]. This paper focuses on model partition between layers, namely, pipeline parallelism.

The pipe-based model parallelism can benefit from: 1) overcoming the single node’s GPU memory limitation through partitioning large model and distributing to each device. 2) reducing communication overhead compared to data parallel, where only intermediate outputs (and corresponding gradients) of the boundary layers needs to transmit to its neighbours. However, this approach suffers from low resource utilization as only one device is active in the execution of pipeline workflow.

Pipeline parallelism. Pipeline Parallelism[17, 23, 24, 49, 52] has been recently proposed to train DNN in a pipelined manner. This approach achieves better overlap of communication and computation with each other, as communication and computation are executed in a finner granularity through the pipeline workflow.

GPipe[24, 31] explores synchronous pipeline approach to train large models with limited GPU memory. PipeDream[23] explores the hybrid approach of data and pipeline parallelism for asynchronous training. [11, 17, 22] make further optimization based on PipeDream. Pal et al. [38] evaluated the hybrid approach without thorough study. Some researchers have been seeking for the optimal placement strategy to assign operations in a DNN to different devices[21, 35, 37, 46] to further improve system efficiency.

8 Conclusion

In this paper, we propose DAPPLE framework for pipelined training of large DNN models. DAPPLE addresses the need for synchronous pipelined training and advances current state-of-the-art by novel pipeline planning and micro-batch scheduling approaches. On one hand, DAPPLE planner automatically determines an optimal parallelization strategy given model structure and hardware configurations as inputs. On the other hand, DAPPLE scheduler is capable of simultaneously achieving optimal training efficiency and moderate memory consumption, without storing multiple versions of parameters and getting rid of the strong demand of re-computation which hurs system efficiency at the same time. Experiments show that DAPPLE planner consistently outperforms strategies generated by PipeDream’s planner by up to 3.23× speedup under synchronous training scenarios, and DAPPLE scheduler outperforms GPipe by 1.6× speedup of training throughput and saves 12% of memory consumption at the same time.

References

in neural information processing systems. 1223–1231.


