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

PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441593

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 sychronous 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 sychronization 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 mini-batches 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 recomputes 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.

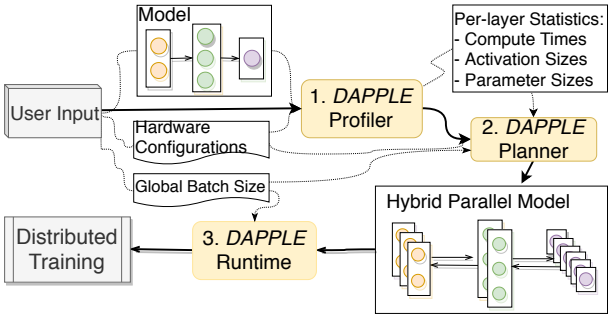


Figure 1. *DAPPLE* framework overview.

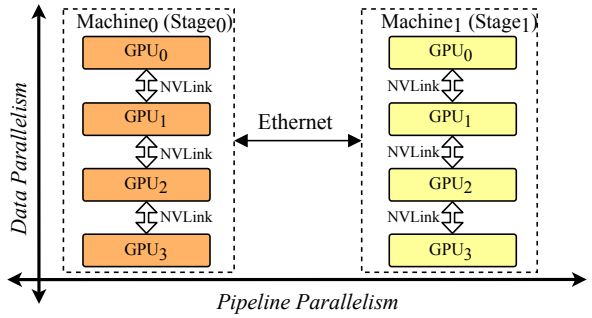


Figure 2. Device mapping on hierarchical interconnects.

2 The *DAPPLE* Approach Overview

Fig. 1 shows high-level workflow of *DAPPLE* which features a *profiler*, a *planner* and a *runtime* system. Overall, *DAPPLE profiler* takes one DNN model as input, and profiles execution time, activation sizes and parameter sizes for each layer. 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 *recomputation* 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.

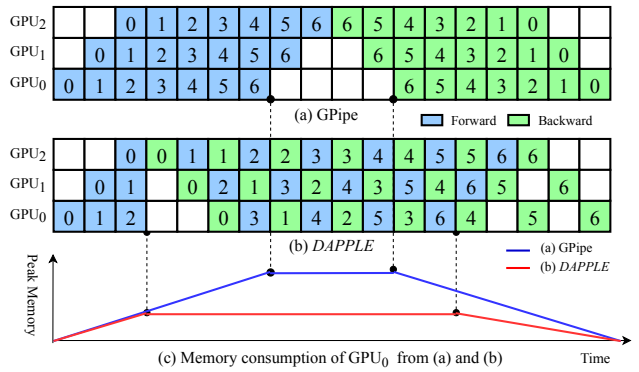


Figure 3. The different scheduling between GPipe(a) and DAPPLE(b) and their memory consumptions.

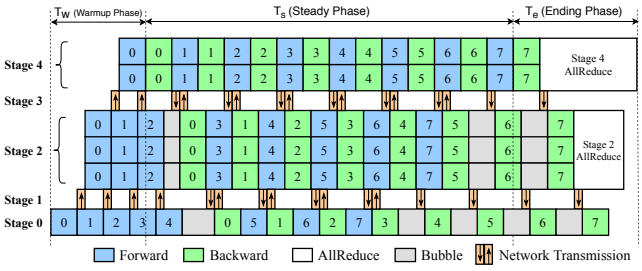


Figure 4. *DAPPLE* pipeline example. The numbers in the cells represent micro-batch ids and Stage-4 is the *pivot* stage.

4.1 The Optimization Objective

For synchronous training, we use the execution time of a single global batch as our performance metric, which we call *pipeline latency*. The optimization objective is to minimize pipeline latency L with the consideration of all solution spaces of data parallelism and pipeline parallelism.

In synchronous pipeline training, computations and cross-stage communication of all stages usually form a *trapezoid*. Fig.4 shows a pipelined training example with well designed task scheduling arrangement, where network communications are arranged as individual stages. We use blue and green blocks to indicate forward and backward computations, respectively, with numbers in them represent micro-batch ids. Gray blocks indicate *bubble* overheads, which refer to some idle time per accelerator introduced by stage partitions. Such shape shown above is formed due to the nature of DNN training: for example, the forward block of micro-batch i at stage s must be performed before the forward block of micro-batch i at stage $s+1$ as well as the forward block of micro-batch $i+1$ at stage s. We denote the stage with the least bubble overhead as *pivot stage*, which will be the dominant factor in calculating pipeline latency L. Let its stage id be Q. How to choose the pivot stage is discussed is Section 4.3.

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:

$$
\begin{aligned}
T_w &= \sum_{s=0}^{Q} F_s \\
T_s &= (M-1) \times (F_Q + B_Q) \\
T_e &= \max_{s=0}^{S-1}(AR(P_s, g_s) + \begin{cases} -\sum_{a=Q}^{s} B_a & s > Q \\ \sum_{a=s}^{Q} B_a & s \le Q \end{cases})
\end{aligned} \tag{1}
$$

$$
L = T_w + T_s + T_e \tag{2}
$$

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.

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

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 if the stage cannot fully occupy the machine.

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.

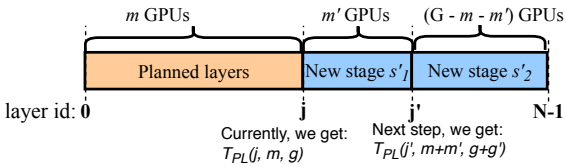


Figure 6. Planning process for j'.

$$Q = \arg\max_{s=S-1}^{0} \max\left(T_{st}^{Q} + \sum_{s'=s+1}^{Q-1}(F_{s'} + B_{s'}), T_{st}^{s}\right) \quad (3)$$

Algorithm 1 Iteratively update Q

```
let Q = S-1, s = Q-1
while s ≥ 0 do
    let l1 = (M − 1) × (Fs + Bs)
    let l2 = (M − 1) × (FQ + BQ)
    for s' = s + 1; s' < Q; s' = s' + 1 do
        l2 += Fs' + Bs'
    end for
    if l1 > l2 then
        Q = s
    end if
    s = s − 1
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 $S-1$. $T_{st}^{j} = (M-1) \times (F_j + B_j)$ means the duration of steady phase, without bubbles, suppose pivot stage is j. For a stage $s < Q$, if T_{st}^{s} is larger than the sum of T_{st}^{Q} 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 $T_{PL}(N, G, \mathcal{G})$, $\mathcal{G} = \{0, 1, ..., G-1\}$. N, G and $\mathcal{G}$ denote the number of layers, number of GPUs and GPU set, respectively. Formula 4 describes the algorithm.

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

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.

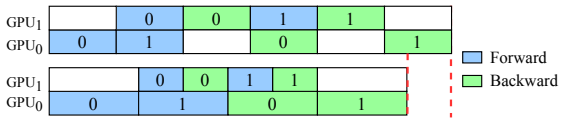


Figure 7. Uneven pipeline minimum example.

Fig. 6 describes the iterative planning process. Suppose we have already planned for the first j ($0 \le j < N$) layers and have the estimation $T_{PL}(j, m, g)$ as pipeline latency. The layers after j forms a stage s'. Meanwhile, we get the optimal Q for current strategy along with the cost of F_Q and B_Q for stage Q. Next step, we try to add one more partition in stage s', supposing after layer j' ($j < j' \le N$), and split s' into two new stages s'_1 and s'_2. We assign m' GPUs for s'_1 and $(G - m - m')$ GPUs for s'_2, and estimate $T_{PL}(j', m + m', g + g')$ according to formula 5. Note *DAPPLE* enumerates the three strategies in section 4.2 for device placement of stage s'_1.

$$T_{PL}(j', m + m', g + g') = L \tag{5}$$

Here, L is the same with that in formula 2. The key for the estimation of L in formula 5 is to find Q of subproblem $T_{PL}(j', m + m', g + g')$. In the sample in Fig. 6, we get Q_j for $T_{PL}(j, m, g)$. We apply formula 3 to get $Q_{j'}$ for $T_{PL}(j, m + m', g + g')$ with the help of Q_j: if Q_j is not s', we do not need to iterate all stages before j, but use Q_j for all stages before layer j instead in the iterative process.

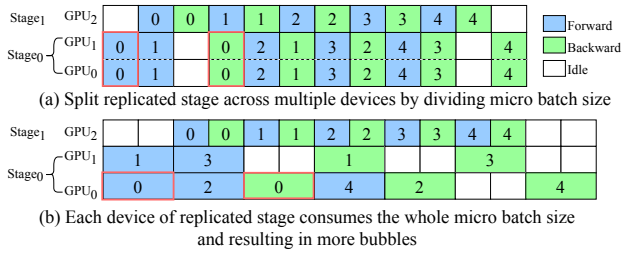
Along the above process, we record the current best split, replication and placement for each point in our solution space using memorized search.

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.



(a) Split replicated stage across multiple devices by dividing micro batch size

(b) Each device of replicated stage consumes the whole micro batch size and resulting in more bubbles

Figure 8. Efficiency of two-stage replication approaches. $Stage_0$ consumes twice as much time as $stage_1$ for a micro-batch.

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 in sequence and record the boundary tensors. No backward graphs should be built until all forward graphs are ready. Second, backward graphs will be built in reverse order for each stage.

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.

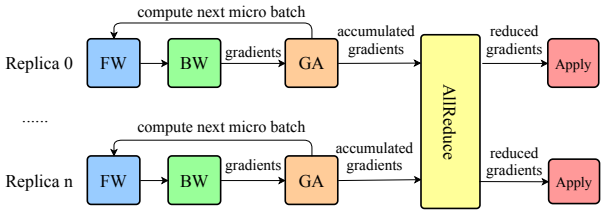


Figure 9. Weights update. GA means gradient accumulation[6].

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 updating 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, K_i 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 K_i is essential as it indicates the peak memory consumption for stage i. There are two

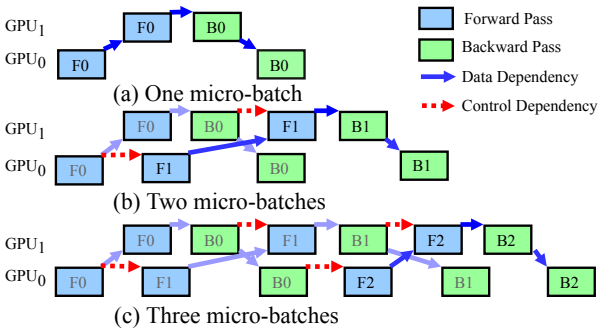


Figure 10. Micro-batches scheduling. The solid blue and dotted red arrows denote data and control dependencies, respectively.

primary factors for deciding K_i: 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 K_i in practice. *Policy A* (P_A): $K_i = min(S - i, D)$. P_A works well when ACR is small, i.e. the impact of cross stage communication overhead is negligible. *Policy B* (P_B): $K_i = min(2 * (S - i) - 1, D)$. 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 K_i 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

Table 1. Benchmark models.

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

Table 2. Hardware configurations.

Config	GPU(s) per server(N_s)	Intra-server connnections	Inter-server connections
A	8x V100	NVLink	25 Gbps
B	1x V100	N/A	25 Gbps
C	1x V100	N/A	10 Gbps

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

Model	Bert-48	XLNet-36	VGG-19	GNMT-16
Speedup	1.0	1.02	1.1	1.31

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.

Table 4. *DAPPLE* planning results.

Model (*GBS*)	$\#Servers \times N_s$	Output Plan	Split Position	*ACR*
ResNet-50 (2048)	2×8 (A)	DP	-	-
	16×1 (B)	DP	-	-
	16×1 (C)	DP	-	-
VGG-19 (2048)	2×8 (A)	DP	-	-
	16×1 (B)	DP	-	-
	16×1 (C)	15 : 1	13 : 6	0.40
GNMT-16 (1024)	2×8 (A)	8 : 8	9 : 7	0.10
	16×1 (B)	8 : 8	9 : 7	0.10
	16×1 (C)	Straight	-	3.75
BERT-48 (64)	2×8 (A)	8 : 8	23 : 25	0.06
	16×1 (B)	Straight	-	0.50
	16×1 (C)	Straight	-	1.25
XLNet-36 (128)	2×8 (A)	8 : 8	18 : 18	0.03
	16×1 (B)	8 : 8	18 : 18	0.03
	16×1 (C)	Straight	-	0.67
AmoebaNet-36 (128)	2×8 (A)	8 : 8	24 : 12	0.18
	16×1 (B)	11 : 5	27 : 9	0.14
	16×1 (C)	11 : 5	27 : 9	0.35

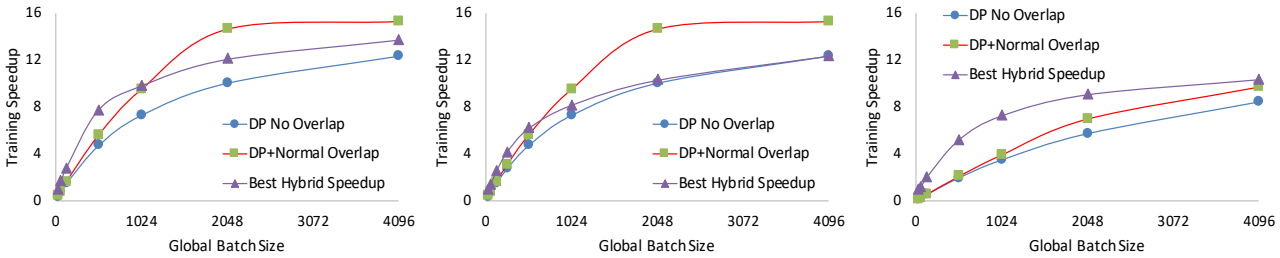
A *straight* plan denotes pipelines with no replication.

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

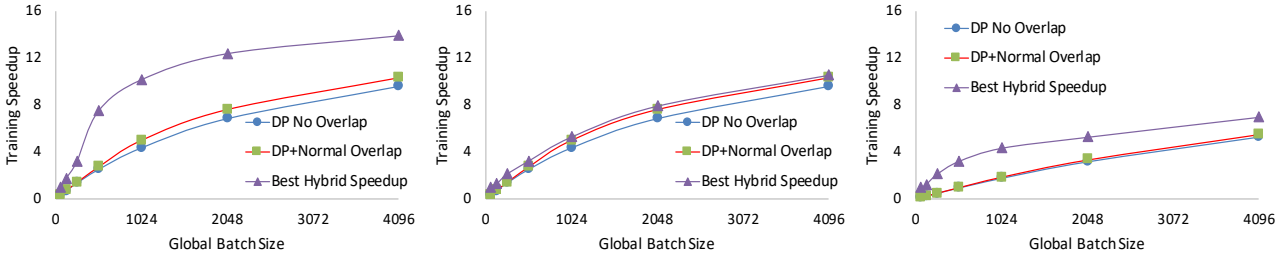
The *Split Position* column of Table 4 shows the stage partition point of each model for the corresponding pipeline plan. The *ACR* column of the table shows the *averaged* ratio of cross-stage comm latency (i.e. comm of both activations in FW and gradients in BW) and stage computation time.

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.

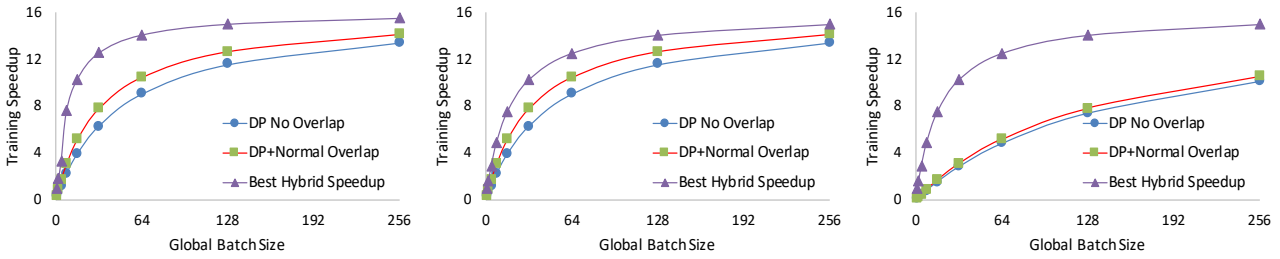
VGG-19. Best plans in config *A* and *B* are also *DP* (Fig. 11 (a) and(b)), due to the moderate model size (548MB), relatively fast interconnects (25 Gbps), and the overlapping in *DP*. The weights and computation distributions of VGG19 are also considered overlapping-friendly, since most of the weights 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)).



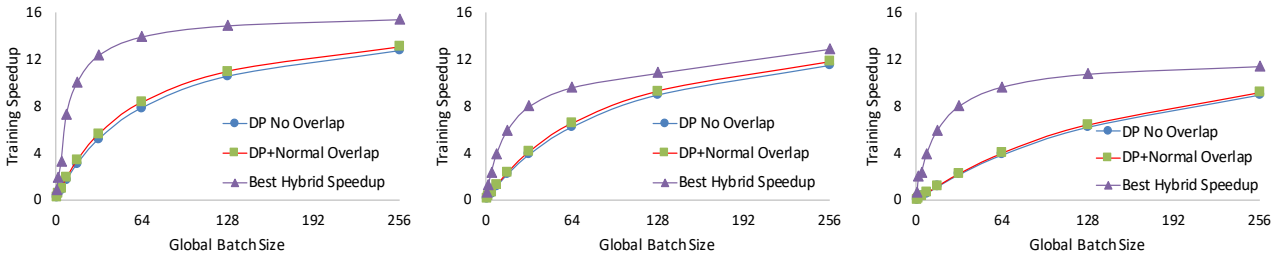
(a) VGG-19 on config *A* **(b)** VGG-19 on config *B* **(c)** VGG-19 on config *C*



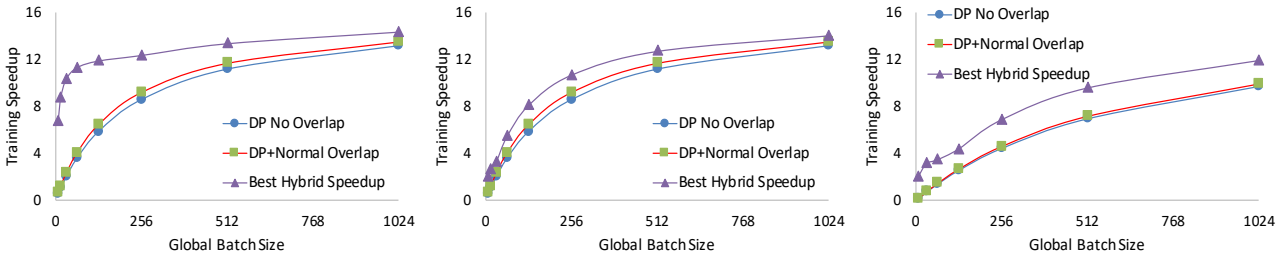
(d) GNMT-16 on config *A* **(e)** GNMT-16 on config *B* **(f)** GNMT-16 on config *C*



(g) BERT-48 on config *A* **(h)** BERT-48 on config *B* **(i)** BERT-48 on config *C*



(j) XLNet-36 on config *A* **(k)** XLNet-36 on config *B* **(l)** XLNet-36 on config *C*



(m) AmoebaNet-36 on config *A* **(n)** AmoebaNet-36 on config *B* **(o)** AmoebaNet-36 on config *C*

Figure 11. Speedups on configurations with hierarchical/flat interconnects.

GNMT-16/BERT-48/XLNet-36. All three models have uniform layer structures, i.e., each layer has roughly the same scale of computations and parameters. And the parameter scales of these models vary from 1.2 GB up to 2.6 GB(Table 1). In *config-A* where all three models achieve low *ACR* values (0.10, 0.06 and 0.03, respectively, as shown in Table 4), a two stage 8 : 8 pipeline works best. Unlike VGG-19, the three models' layers are relatively uniformly distributed, thus a symmetric, evenly partitioning is more efficient. In config *C*, a straight pipeline works best for all three models. In this config, all devices have approximately the same workload. More importantly, no replication eliminates gradients sync overheads for relatively large models (1.2-2.6 GB) on a slow network (10 Gbps).

AmoebaNet-36. For AmoebaNet-36, *DP* is not available due to device memory limit. AmoebaNet-36 has more complex network patterns than other models we evaluated, and larger ACR in config *A* as well. Thus, more successive forward micro-batches are needed to saturate the pipeline. For all three configs, two-stage pipeline (8 : 8, 11 : 5 and 11 : 5, respectively) works best.

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 1.71×/1.37×/1.79× speedup for *config-A*, *config-B* and *config-C*, respectively. Specially, this speedup is up to 2.32× 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 spit 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 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.

Config	# of micro batch (M)	Throughput (samples/sec)	Average Peak Memory (GB)
GPipe	2	5.10	12.1
	3	–	*OOM*
GPipe + RC	2	4.00	9.9
	5	5.53	13.2
	8	–	*OOM*
DAPPLE	2	5.10	10.6
	8	7.60	10.6
	16	8.18	10.6
DAPPLE + RC	2	4.24	8.5
	8	6.23	8.5
	16	6.77	8.5

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_i 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.

Table 6. *DAPPLE* and PipeDream strategies comparison. in the form of (start layer, end layer)@[GPU IDs].

Model (Global Batch Size)	*DAPPLE*	PipeDream
VGG19 (1024)	(0, 16) @ [G0-G13] (17, 25) @ [G14,G15]	(0, 11) @ [G0-G7] (11, 17) @ [G8-G13] (17, 19) @ G14 (19, 25) @ G15
AmoebaNet-36 (128)	(0, 30) @ [G0-G7] (31, 43) @ [G8-G15]	straight
BERT Large (128)	(0, 13) @ [G0-G7] (14, 26) @ [G8-G15]	(0, 4) @ [G0,G1] (4, 13) @ [G2-G7] (13, 16) @ [G8, G9] (16, 19) @ [G10,G11] (19, 22) @ [G12,G13] (22, 26) @ [G14,G15]
XLNet-36 (128)	(0, 22) @ [G0-G7] (23, 41) @ [G8-G15]	straight

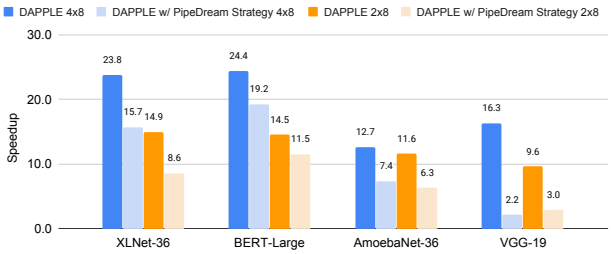


Figure 12. Performance comparison with PipeDream.

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.

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

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 hurts 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

[1] 2016. *A New Lightweight, Modular, and Scalable Deep Learning Framework.* https://caffe2.ai/.
[2] 2018. *Baidu-allreduce.* https://github.com/baidu-research/baidu-allreduce.
[3] 2019. *Byteps, A high performance and generic framework for distributed DNN training.* https://github.com/bytedance/byteps.
[4] 2019. *GpipeTalk.* https://www.youtube.com/watch?v=9s2cum25Kkc.
[5] 2019. *Gradients Accumulation-PyTorch.* https://gist.github.com/thomwolf/ac7a7da6b1888c2eeac8ac8b9b05d3d3.
[6] 2019. *Gradients Accumulation-Tensorflow.* https://github.com/tensorflow/tensorflow/pull/32576.
[7] 2019. *NCCL.* https://developer.nvidia.com/nccl.
[8] 2019. *NVLink.* https://www.nvidia.com/en-us/data-center/nvlink/.
[9] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.
[10] Naveen Arivazhagan, Ankur Bapna, Orhan Firat, Dmitry Lepikhin, Melvin Johnson, Maxim Krikun, Mia Xu Chen, Yuan Cao, George Foster, Colin Cherry, et al. 2019. Massively multilingual neural machine translation in the wild: Findings and challenges. *arXiv preprint arXiv:1907.05019* (2019).
[11] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. 2018. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839* (2018).
[12] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174* (2016).
[13] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems, ACM, New York, NY, USA*. ACM.
[14] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large scale distributed deep networks. In *Advances*

in neural information processing systems. 1223–1231.

[15] Julien Demouth. 2015. CUDA Pro Tip: Minimize the Tail Effect. https://devblogs.nvidia.com/cuda-pro-tip-minimize-the-tail-effect/

[16] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Marc Snir, and Brian Van Essen. 2019. Channel and filter parallelism for large-scale CNN training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–20.

[17] Jinkun Geng, Dan Li, and Shuai Wang. 2019. Elasticpipe: An efficient and dynamic model-parallel solution to dnn training. In *Proceedings of the 10th Workshop on Scientific Cloud Computing*. ACM, 5–9.

[18] Jinkun Geng, Dan Li, and Shuai Wang. 2019. Horizontal or Vertical?: A Hybrid Approach to Large-Scale Distributed Machine Learning. In *Proceedings of the 10th Workshop on Scientific Cloud Computing*. ACM, 1–4.

[19] Jinkun Geng, Dan Li, and Shuai Wang. 2019. Rima: An RDMA-Accelerated Model-Parallelized Solution to Large-Scale Matrix Factorization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 100–111.

[20] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[21] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 485–500.

[22] Lei Guan, Wotao Yin, Dongsheng Li, and Xicheng Lu. 2019. XPipe: Efficient Pipeline Model Parallelism for Multi-GPU DNN Training. *arXiv preprint arXiv:1911.04610* (2019).

[23] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377* (2018).

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

[25] Zhouyuan Huo, Bin Gu, Qian Yang, and Heng Huang. 2018. Decoupled parallel backpropagation with convergence guarantee. *arXiv preprint arXiv:1804.10574* (2018).

[26] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems* 2 (2020), 497–511.

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

[28] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. 2018. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205* (2018).

[29] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. 2018. Exploring the Hidden Dimension in Accelerating Convolutional Neural Networks. (2018).

[30] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358* (2018).

[31] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. 2020. torchgpipe: On-the-fly Pipeline Parallelism for Training Giant Models. *arXiv preprint arXiv:2004.09910* (2020).

[32] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[33] Tung D Le, Taro Sekiyama, Yasushi Negishi, Haruki Imai, and Kiyokuni Kawachiya. 2018. Involving CPUs into Multi-GPU Deep Learning. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 56–67.

[34] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).

[35] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2430–2439.

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

[37] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2020. Memory-Efficient Pipeline-Parallel DNN Training. *arXiv preprint arXiv:2006.09503* (2020).

[38] Saptadeep Pal, Eiman Ebrahimi, Arslan Zulfiqar, Yaosheng Fu, Victor Zhang, Szymon Migacz, David Nellans, and Puneet Gupta. 2019. Optimizing multi-GPU parallelization strategies for deep learning training. *IEEE Micro* 39, 5 (2019), 91–101.

[39] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and J. Peter Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *https://arxiv.org/abs/1910.10683* (2019).

[40] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know What You Don't Know: Unanswerable Questions for SQuAD. *arXiv preprint arXiv:1806.03822* (2018).

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

[42] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Edinburgh neural machine translation systems for WMT 16. *arXiv preprint arXiv:1606.02891* (2016).

[43] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[44] Rich Sutton. 2019. *The Bitter Lesson*. http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

[45] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 839–848.

[46] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[47] Mengdi Wang, Chen Meng, Guoping Long, Chuan Wu, Jun Yang, Wei Lin, and Yangqing Jia. 2019. Characterizing Deep Learning Training Workloads on Alibaba-PAI. *arXiv preprint arXiv:1910.05930* (2019).

[48] Siyu Wang, Yi Rong, Shiqing Fan, Zhen Zheng, LanSong Diao, Guoping Long, Jun Yang, Xiaoyong Liu, and Wei Lin. 2020. Auto-MAP: A DQN Framework for Exploring Distributed Execution Plans for DNN Workloads. *arXiv preprint arXiv:2007.04069* (2020).

[49] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R Aberger, and Christopher De Sa. 2019. PipeMare: Asynchronous Pipeline Parallel DNN Training. *arXiv preprint arXiv:1910.05124* (2019).

[50] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. *arXiv preprint arXiv:1906.08237* (2019).

[51] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2019. Large-batch training for LSTM and beyond. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[52] Jun Zhan and Jinghui Zhang. 2019. Pipe-Torch: Pipeline-Based Distributed Deep Learning in a GPU Cluster with Heterogeneous Networking. In *2019 Seventh International Conference on Advanced Cloud and Big Data (CBD)*. IEEE, 55–60.

[53] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xi, and Eric P. Xing. 2017. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 181–193. https://www.usenix.org/conference/atc17/technical-sessions/presentation/zhang

[54] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. 2020. Fusion-stitching: boosting memory intensive computations for deep learning workloads. *arXiv preprint arXiv:2009.10924* (2020).

A Artifact Appendix

A.1 Abstract

The artifact contains the code for the *DAPPLE* Planner, implemented in Python and Rust, and the *DAPPLE* Runtime based on TensorFlow-1.12. We provide instructions for using the planner to generate parallel strategies as well as scripts to run the actual training experiments.

A.2 Artifact check-list (meta-information)

- **Algorithm:** The Planner uses Dynamic Programming to solve the MDPs in order to find the optimal parallel strategy that yields maximum training throughput. It also uses extensive multi-process parallel graph traversing to accelerate the search process.
- **Compilation:** The nightly Rust compiler is used to compile the code into a Rust library available to use as Planner Rust API. Python 3 interpreter is also required to wrap the Rust library to provide a Python API for the planner.
- **Run-time environment:** All of the Runtime experiments should be run under Linux environments with CUDA-9.0 and TensorFlow installed. Python and graphviz are also required for the Planner experiments.
- **Hardware:** Performance experiments should be measured in a cluster of Nvidia DGX-1 V100(16 GB RAM) machines with high-bandwidth, low-latency NVLink interconnects, and 25Gbps ethernet as inter-node network, in order to reproduce the paper's results. However, *DAPPLE* can be adapt to arbitrary environments thanks to the automatic planner as long as the profiling data is updated correspondingly.
- **Output:** Parallel execution plan for the given input model and hyper parameters.
- **Experiments:** Planner's output strategy and real-world performance for several models, such as BERT, VGG19, AmoebaNet.
- **How much disk space required (approximately)?:** 10 MB.
- **How much time is needed to prepare workflow (approximately)?:** 5 minutes.
- **How much time is needed to complete experiments (approximately)?:** 1 hour.
- **Publicly available?: Yes.**
- **Code licenses (if publicly available)?:** BSD-3-Clause.
- **Data licenses (if publicly available)?:** BSD-3-Clause (same as code).
- **Workflow framework used?:** TensorFlow-1.12, PyTorch, PyO3, Rust, Python.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.3887384.

A.3 Description

A.3.1 How to access. The *DAPPLE* artifacts is available on GitHub at https://github.com/AlibabaPAI/DAPPLE.

A.3.2 Hardware dependencies. The DAPPLE Planner requires a desktop operating system running on x86, x86_64, or aarch64 architecture. The DAPPLE runtime requires Nvidia V100 equipped Linux x86_64 machines, preferably with NVLink and Ethernet.

A.3.3 Software dependencies. The *DAPPLE* Planner depends on Python 3, Rust and a few cargo crates defined in cargo.toml, and is tested on Linux, Windows, and macOS. The *DAPPLE* Runtime needs TensorFlow, CUDA, NCCL to be present on the system. In our experiments, all servers run 64-bits CentOS 7.2 with CUDA 9.0, cuDNN v7.3, NCCL 2.4.2[7] and Tensorflow-1.12.

A.3.4 Data sets. The datasets applied for the three tasks (Table 1) are WMT16 En-De [42], SQuAD2.0 [40] and ImageNet[41], respectively.

A.3.5 Models. We provide *DAPPLE* reference implementation for VGG19, GNMT, BERT, NMT, and AmoebaNet. Details are summarized in Table 1.

A.4 Installation

We will provide detailed documentation on how to get started in our open source project at github: *DAPPLE*.

A.5 Evaluation and expected results

We will provide detailed documentation on how to reproduce all the experiments with our provided Docker container in our open source project at gitHub: *DAPPLE*.

A.6 Experiment customization