CAPTURE: Memory-Centric Partitioning for Distributed DNN Training with Hybrid Parallelism

Henk Dreuning
University of Amsterdam
Amsterdam, The Netherlands
h.h.h.dreuning@uva.nl

Kees Verstoep, Henri E. Bal
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
{c.verstoep, h.e.bal}@vu.nl

Rob V. van Nieuwpoort
Leiden University
Leiden, The Netherlands
r.v.van.nieuwpoort@liacs.leidenuniv.nl

***Abstract*—Deep Learning (DL) model sizes are increasing at a rapid pace, as larger models typically offer better statistical performance. Modern Large Language Models (LLMs) and image processing models contain billions of trainable parameters. Training such massive neural networks incurs significant memory requirements and financial cost. Hybrid-parallel training approaches have emerged that combine pipelining with data and tensor parallelism to facilitate the training of large DL models on distributed hardware setups. However, existing approaches to design a hybrid-parallel partitioning and parallelization plan for DL models focus on achieving high throughput and not on minimizing memory usage and financial cost.**

We introduce CAPTURE, a partitioning and parallelization approach for hybrid parallelism that minimizes peak memory usage. CAPTURE combines a profiling-based approach with statistical modeling to recommend a partitioning and parallelization plan that minimizes the peak memory usage across all the Graphics Processing Units (GPUs) in the hardware setup. Our results show a reduction in memory usage of up to 43.9% compared to partitioners in state-of-the-art hybrid-parallel training systems. The reduced memory footprint enables the training of larger DL models on the same hardware resources and training with larger batch sizes. CAPTURE can also train a given model on a smaller hardware setup than other approaches, reducing the financial cost of training massive DL models.

***Index Terms*—HPC, Deep Learning, Hybrid Parallelism, GPU, Memory**

I. Introduction

Scaling up Deep Learning (DL) model sizes has proven to be an effective way to increase a model's statistical performance for many application domains. Deep Neural Networks (DNNs) for natural language processing (NLP) and image processing typically show increased accuracy when the model is scaled up. Training such networks is not only a compute-intensive task, but also incurs significant memory requirements and financial cost. Training large DNNs requires an abundance of hardware resources and state-of-the-art distributed training techniques such as pipeline parallelism to meet the memory requirements. However, pipeline parallelism only provides a partial solution to the performance and memory bottleneck in large-scale DNN training. As DNN sizes keep increasing, new hybrid forms of parallelism are being developed that aim to further increase the trainable model size and training throughput. To facilitate the training of large DNNs, state-of-the-art hybrid-parallel training systems aim to (1) increase the trainable model size by enabling scaling to more hardware resources and (2) reduce the cost of training large DNNs by making effective use of the available resources and enabling training on cheap, preemptible cloud instances.

In this work, we introduce CAPTURE, a new approach to partition DNN models for hybrid parallelism, focusing specifically on memory usage. CAPTURE is inspired by mCAP [1], our existing partitioning approach for pipeline-parallel DNN training that balances peak memory usage between workers. Similar to mCAP, CAPTURE uses a *profiling-based* approach to collect memory statistics about the DL model. Unlike mCAP, CAPTURE uses those memory statistics and applies *statistical modeling* techniques to predict the memory usage for various hybrid-parallel partitioning plans and is thus not limited to pipeline parallelism only. Based on the predictions, it recommends a *partitioning and parallelization plan* that minimizes peak memory usage across workers (GPUs).

Hybrid parallelism is a combination of pipelining with stage-wise data- and/or tensor parallelism. Applying data- and/or tensor-parallelism within a pipeline stage can help to balance the memory usage across GPUs, but also has an effect on the total memory requirement: the memory required to process a DNN layer using data- or tensor parallelism depends, amongst others, on the layer's weight to activation ratio.

CAPTURE determines how to partition a DL model into pipeline stages and what form (data- and/or tensor) and degree of parallelism to apply to each pipeline stage. CAPTURE recommends a partitioning and parallelization plan with a low and well-balanced peak memory usage across all GPUs. Minimizing the peak memory usage enables training of larger DL models or training with a smaller hardware setup. Because CAPTURE is profiling-based, it can capture the effects of memory optimizations present in DL and pipeline- or hybrid-parallel frameworks on memory usage. By combining profiling with statistical modeling, it can predict the peak memory usage of any model partitioning and hybrid parallelization plan and recommend a memory-friendly plan for the target hardware configuration, but also for smaller or larger hardware setups.

Existing partitioners in hybrid-parallel systems typically suggest a partitioning and parallelization plan that optimizes the achieved computational throughput. However, optimizing the partitioning for throughput can be disadvantageous when the goal is to maximize trainable model size or minimize financial cost. CAPTURE finds a partitioning and paralleliza-

tion plan with a low and well-balanced (peak) memory usage, which enables (1) training of larger DNNs with the same hardware, or (2) training a given model with fewer resources, which can reduce the financial cost of training large DNNs.

CAPTURE consists of three stages: a profiling stage that gathers per-layer memory statistics, a prediction stage that models/predicts per-GPU peak memory usage for different pipeline-parallel model partitionings as well as different per-stage data- and tensor-parallel parallelization strategies, and a recommendation stage that finds a memory-efficient and well-balanced partitioning. The recommendation stage uses predictions of the memory usage for different partitioning plans made by the predictor. Various optimizations, such as a knowledge-guided layer grouping approach, limit the time required in the profiling and prediction stages.

A major advantage of our method is that it is DL-framework agnostic. We have implemented it for two state-of-the-art hybrid-parallel training systems with different underlying DL frameworks and software stacks: Alpa [2] and Varuna [3], though in this paper we focus on Alpa. We evaluate CAPTURE's effectiveness on a distributed scale.

We demonstrate that our approach reduces the peak memory usage by up to 43.9% compared to other partitioning approaches for hybrid parallelism. It can train larger DNNs on a given hardware setup and it can train a given model on more than two times smaller hardware setups.

Concretely, the contributions of this paper are:

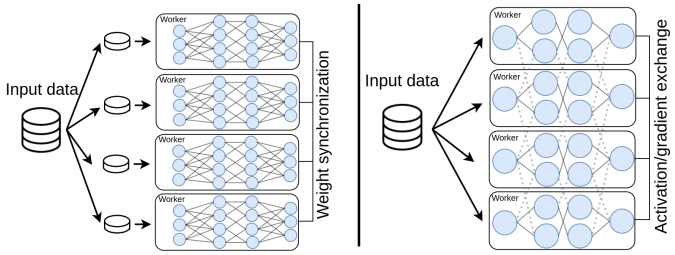
- We introduce CAPTURE, the first memory-centric partitioning and parallelization approach for hybrid parallelism.
- We provide a method based on *profiling* and *statistical modeling* that can predict the memory usage of hybrid-parallel training plans with per-stage data- or tensor parallelism for hardware setups of any scale.
- We demonstrate that CAPTURE is DL-framework agnostic and is generically applicable to hybrid-parallel systems by implementing it for two state-of-the-art hybrid-parallel systems, Alpa and Varuna.
- We evaluate the effectiveness of CAPTURE for pipeline- and hybrid-parallel training on a distributed scale.
- We evaluate the impact of using a memory-centric partitioning approach over a throughput-oriented partitioner on computational performance.

The rest of the paper is structured as follows: Section II provides background information on hybrid-parallel DNN training, existing systems and partitioning/parallelization approaches. Section III describes CAPTURE's design and components, Section IV contains our evaluation and Section V contains conclusions and directions for future work.

II. Background and Related Work

A. DNN training

A DNN is a self-learning model that can be trained by feeding minibatches of training samples into the model. For each minibatch, a forward pass is performed that makes a prediction (e.g. a classification) for that input data. The forward pass is followed by a backward pass, which calculates updates (gradients) for each weight in the neural network that improve the predictions for the current inputs. After the backward pass the updates are applied to the weights in the neural network. During the forward pass, activations are computed, which flow through the neural network from the first to the last layer. During the backward pass, gradients flow from the last to the first layer in the model.



(a) Data parallelism (b) Tensor parallelism

Fig. 1. Different types of parallelism.

B. Parallel and distributed DNN training

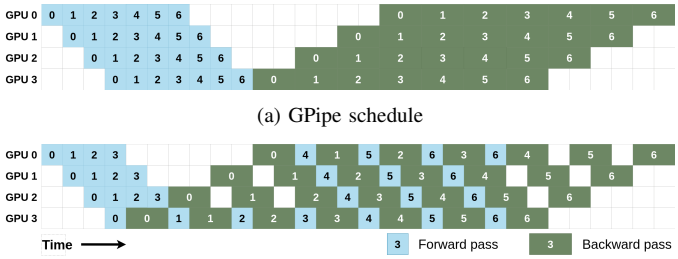
In **data-parallel** DNN training, the DL model is replicated on all workers. Each worker trains its model on a different part of the input data. Thus, each worker processes a separate batch of input data in each training iteration and updates its own weights. The weights are periodically exchanged and synchronized. Figure 1a illustrates this principle.

In **tensor-** or **operator parallelism** the operations that constitute the DL model (matrix multiplications) are partitioned over multiple workers. Multiple workers collectively process the same input data for the same operator. Conceptually, a single DNN layer or a group of layers is distributed over multiple workers, as illustrated in Figure 1b.

In **pipeline-parallel** training [4]–[7], the DL model is partitioned over the workers. Each worker trains a group of DNN layers and the input data is fed into the model in a pipelined fashion. After the full batch has been processed, a weight update is applied, after which the next batch of input data is fed into the pipeline. In this paper, we focus on *intra-batch* or *synchronous* pipelining, in which an input batch is split into multiple microbatches that are fed into the pipeline.

The pipelining schedule determines the order in which the forward and backward passes of the microbatches are executed. The GPipe schedule (Fig. 2a) first processes all forward passes, followed by all backward passes, and then the update step. The 1F1B schedule (Fig. 2b) interleaves forward and backward passes to enable earlier release of memory for activations. As a result, the 1F1B schedule is more memory efficient for GPUs at the end of the pipeline, but does not improve pipeline latency or training throughput.

CAPTURE can be applied to both pipelining schedules, but motivated by its generic nature, does not explicitly adjust its memory predictions to the variations caused by early release of memory in the 1F1B schedule. We study CAPTURE's effectiveness in recommending a memory-efficient partitioning for both pipelining schedules in our evaluation.



(a) GPipe schedule

(b) 1F1B schedule

Fig. 2. Pipelining schedules.

Hybrid parallelism is the combination of pipeline-, data- and/or tensor parallelism. Existing frameworks that implement hybrid parallelism partition a DL model into multiple pipeline stages. Each pipeline stage can then be processed by multiple workers using data- or tensor-parallelism, as illustrated in Figure 3. In this scenario the input data that is partitioned over the workers consists of the input to the layer (group) that is parallelized using data- and/or tensor parallelism, i.e. the input activations and gradients to the layer or layer group.

The partitioning of layers into pipeline stages, together with the type and degree of parallelism chosen for each pipeline stage is called the *partitioning and parallelization plan*. This plan determines the computational demand per worker, the amount of communication between the workers and the memory requirements for each worker within a stage. For example, processing a pipeline stage in a data-parallel fashion requires an allreduce operation after the update step that synchronizes the weights across the workers. When processing a pipeline stage in a tensor-parallel fashion, two allreduce operations per forward and backward pass are required. Each allreduce operation communicates the (output) activations/gradients generated for the layers in that pipeline stage. Hence, the partitioning of the model into pipeline stages combined with the parallelization strategy per stage directly impacts the latency of the full pipeline, the achieved training throughput and the overall (peak) memory usage.

Note that simply reducing the batch size is ineffective to solve the memory bottleneck in hybrid-parallel training, and often impossible. A sufficiently large batch size is required to form microbatches for pipelining and for data-parallel execution of pipeline stages. It is also needed to provide enough parallelism to achieve reasonable hardware utilization, because hybrid parallel systems rely on parallel and pipelined execution of training steps and on overlapping of computation and communication. Moreover, the batch size influences the number of activations and gradients generated during training, but not the size of the model's parameters. Thus, depending on the model, lowering the batch size may not sufficiently reduce memory usage to resolve the bottleneck. Finally, some models generate so many activations and gradients that they cannot be trained even with the minimum possible batch size without running out of memory.

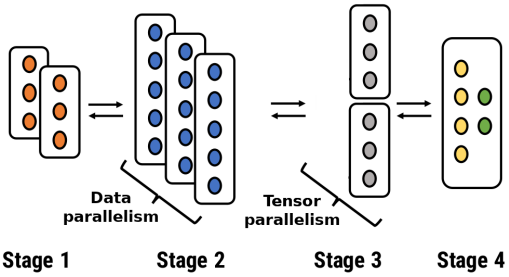


Fig. 3. Example of hybrid parallelism. Figure adapted from [5].

C. Partitioning approaches for hybrid parallelism

A partitioner for hybrid-parallelism recommends a *partitioning and parallelization plan*, which consists of the following aspects: a mapping of layers to pipeline stages (the partitioning), the type of parallelism applied to each stage (data- and/or tensor parallelism) and the number of workers assigned to each stage (the degree of parallelism). Existing partitioning approaches for hybrid parallelism focus on achieving high training throughput. High throughput is achieved by considering: the latency of computing forward and backward passes for a pipeline stage, the communication required between stages and inside stages for data- and/or tensor parallelism and the available communication bandwidth between GPUs and nodes. Existing partitioners do not consider the imbalance in memory usage that may exist when a DL model is partitioned for achieving high throughput.

D. Hybrid-parallel training systems

Different hybrid-parallel frameworks support different combinations of pipeline-, data-, and tensor parallelism. Varuna supports pipelining combined with data parallelism, but not tensor parallelism, a design decision motivated by their target usage scenario. The allreduces in tensor-parallel stages are communication-intensive, which requires a fast interconnect between GPUs and compute nodes. However, Varuna targets training on cheap spot-VMs on cloud resources, where such high-bandwidth, low-latency interconnects are typically not available. Varuna also processes each pipeline stage with the same degree of data parallelism and is limited in the types of DL models it supports. In this work, we focus on generic DNN training with all three forms of parallelism.

DeepSpeed [8] and Megatron-LM [9], [10] support hybrid pipeline-, data- and tensor parallelism and divide the DL model into pipeline stages based on simple metrics, such as the number of parameters or layers. Merak [11] is a hybrid-parallel system that performs throughput-oriented model partitioning. In all three systems, the degrees of data- and tensor parallelism are fixed across stages (or the full pipeline is processed in data-parallel fashion, which results in a similar plan as with a fixed parallel degree across stages). Neither of the systems provide an automated approach to generate a full partitioning and parallelization plan for hybrid parallelism, as we do in this work. Moreover, Megatron-LM and Merak are specialized for Transformer models, while we focus on generic DNN training.

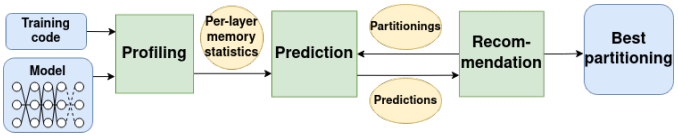


Fig. 4. Overview of mCAP. Figure adapted from [1].

Alpa supports the most elaborate form of hybrid parallelism with pipelining and per-stage data- and/or tensor parallelism, and can assign a different parallel degree to each stage. It thus has the largest number of possible partitioning and parallelization plans of all of today's training systems. Therefore, we perform our evaluation of CAPTURE with Alpa. All other (simpler) forms of hybrid parallelism found in today's hybrid-parallel systems can be considered a subset of Alpa's capabilities, and each of those forms will reduce the number of possible plans and simplify the decision making process to determine a suitable plan. Alpa focuses on throughput when determining a parallelization plan, while we focus on achieving low peak memory usage.

E. mCAP

CAPTURE is inspired by mCAP. mCAP is our memory-centric partitioning approach for pipeline-parallel-only training. It partitions DL models over GPUs to balance peak memory usage. By balancing the peak memory usage across the GPUs, it enables the training of larger neural networks. mCAP consists of 3 stages (see Fig. 4).

mCAP's profiler performs a number of profiling runs to collect two metrics for each layer in the DL model: the (peak) memory usage of the layer when it is the only layer placed on a GPU, and the additional memory usage caused by the layer when it is placed on a GPU together with a set of preceding DNN layers. It is important to record both metrics, because memory optimizations implemented in DL and pipelining frameworks, as well as buffers for communication and passthrough variables, cause the memory usage to be significantly different in both scenarios.

The metrics collected in the profiling stage are used by the predictor, which can predict the memory usage for any given partitioning. The recommender navigates the search space of all the possible partitionings for a given DL model over a given number of GPUs. It supports two search strategies: *mCAP-BF* and *mCAP-BO*. *mCAP-BF* (brute force) iterates over all possible partitionings and requests predictions for them. It keeps track of the prediction with the lowest peak memory usage across the GPUs and recommends that as the best partitioning. The $mCAP-BO$ search strategy uses Bayesian optimization to navigate the search space and find the (near) optimal partitioning in a fixed number of iterations.

mCAP is limited to pipeline parallelism only and cannot generate partitioning and parallelization plans for hybrid parallelism. CAPTURE supports both pure pipelining and hybrid parallelism and, unlike mCAP, can recommend a memory-friendly parallelization plan for any given target batch size.

F. Memory optimizations

Several recent papers propose methods to address memory limitations during DNN training. STR [12] optimizes the scheduling of CPU-GPU memory swapping and recomputation operations to reduce memory usage, but focuses on single GPU training. Harmony [13] addresses the scheduling and placement of compute and swapping operations for multi-GPU training, but focuses on pipeline- or data-parallelism. Our work is orthogonal to these optimizations, and focuses on recommending a partitioning and parallelization plan that minimizes peak memory usage for hybrid parallelism.

III. Method

We introduce CAPTURE, a partitioner for hybrid-parallel DNN training. It partitions the DL model into pipeline stages, determines the parallelization method (data- or tensor-parallel) for each stage and assigns a number of workers (GPUs) to each stage. The recommended *partitioning and parallelization plan* minimizes the peak memory usage across GPUs.

CAPTURE is designed to be DL-framework agnostic and easily applicable to any hybrid-parallel training system. Therefore, a *profiling-based* approach and *statistical modeling* are favored over *analytical* modeling. CAPTURE uses profiling to capture the effects of memory optimizations present in DL- and hybrid-parallel training frameworks on memory usage. It then applies statistical modeling to the profiling data to predict the memory usage for any data- or tensor-parallel pipeline stage. Fig. 5 shows an overview of CAPTURE. This section discusses CAPTURE's design and components.

A. Profiler

CAPTURE's profiling stage performs a number of short profiling runs with a specifically selected set of partitionings and parallelization plans for pipeline- and hybrid-parallel training. Through these runs, two metrics about the memory usage for each layer in the DNN are collected: M_i (memory isolated) and M_a (memory added). $M_i(l)$ is the peak memory usage observed for layer l when that layer is the only one placed on a GPU. $M_a(l)$ is the increase (or decrease) in memory usage observed when layer l is added to an existing set of layers on a GPU. These metrics are based on mCAP, which collects the same metrics for pipeline parallel-only training.

In CAPTURE, the metrics are collected multiple times, for three different training scenarios, namely: training with pipeline parallelism only, training with data- or tensor parallelism and training with different batch sizes. By collecting the memory statistics for different scenarios, we collect the data that is later needed in the prediction stage to accurately predict the memory usage for any partitioning and parallelization plan, as well as any training batch size.

The profiling runs are performed in the same setup as the target training job: the same software stack and hyperparameters are used, except for the batch size. The profiling runs perform two training iterations, in which all stages of the training process are performed (forward pass, backward pass and update step). Because profiling is performed in the

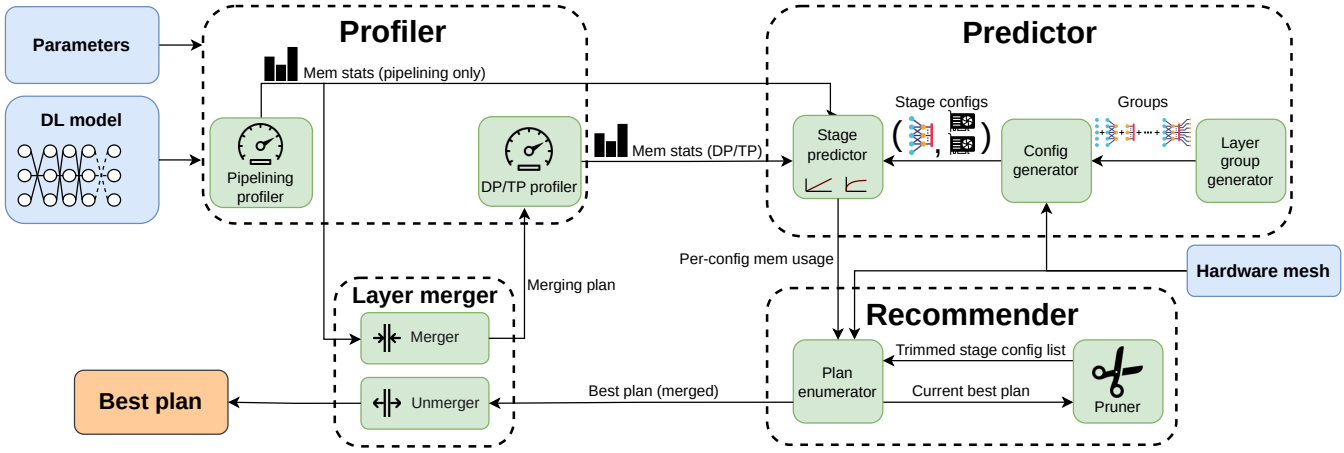


Fig. 5. Overview of CAPTURE.

same setup as the target run, the profiling data captures the effects of memory optimizations in the DL- and hybrid-parallel frameworks on peak memory usage.

1) The profiling partitionings: We demonstrate how the set of profiling partitionings for pipeline parallel-only training is generated for a DNN with L layers on G GPUs, in the idealized case where L is divisible by G. The aim is to collect M_i and M_a for each layer in the DNN. To collect M_i for layer l, we need to perform a profiling run where layer l is the only one on a GPU. To collect M_a, we need a run with some set of layers $l-n, ..., l-1$ (called neighbors) on a GPU, and a run with layers $\{l-n, ..., l\}$ on a GPU. The memory statistics are then calculated as follows:

$$M_i(l) = PM(l)$$

$$M_a(l) = PM(l-n, l) - PM(l-n, l-1)$$

where $PM(l, l+n)$ indicates the profiled peak memory usage of a GPU with layers l to $l+n$ placed on it. To limit the number of required profiling runs, $M_a(l)$ is collected once for each layer, for a single value of n (set of neighbors).

While CAPTURE records the same metrics for the model's layers as mCAP for this profiling scenario (pipelining only), the set of profiling partitionings that CAPTURE generates to record the statistics is different. The partitionings are summarized in Description 1, and the partitionings required for the first L/G layers are described in lines 1-4. These partitionings allow us to extract M_i for layers 0 to $L/G+2$, because all of those layers have been placed alone on a GPU at least once. We can extract M_a for layers 1 to L/G. To extract M_a also for layer $L/G+1$, we add one extra partitioning, described by line 5. We then fix the number of layers for GPU 0 to L/G and repeat the same procedure for GPU 1 (lines 7-11, note that the partitioning on line 4 is duplicated on line 7 for clarity reasons), to add the runs for layers $L/G+1$ to $L/G*2+1$. The procedure repeats until the partitioning on line 14 is reached. A final partitioning is added to record M_a for the last layer (line 15).

Using this method of generating the profiling partitionings for pipeline parallelism, the maximum number of required profiling partitionings (and runs) is $L+1$, where L is the number of layers in the DNN. We perform these runs with two different batch sizes (see Section III-A2), which brings the number of profiling runs to $2*(L+1)$.

Description 1: The profiling partitionings for pipeline parallelism. The numbers in a partitioning denote the number of DNN layers placed on a GPU. eq(k, g) equally distributes k layers over g GPUs by number.

1 $[1, 1, \text{eq}(L-2, G-2)]$
2 $[2, 1, \text{eq}(L-3, G-2)]$
3 ...
4 $[\frac{L}{G}, 1, 1, \text{eq}(L-(\frac{L}{G}+2), G-3)]$
5 $[\frac{L}{G}+1, 1, 1, \text{eq}(L-(\frac{L}{G}+2), G-3)]$
6 ...
7 $[\frac{L}{G}, 1, 1, \text{eq}(L-(\frac{L}{G}+2), G-3)]$
8 $[\frac{L}{G}, 2, 1, \text{eq}(L-(\frac{L}{G}+3), G-3)]$
9 ...
10 $[\frac{L}{G}, \frac{L}{G}, 1, 1, \text{eq}(L-(2*\frac{L}{G}+2, G-4))]$
11 $[\frac{L}{G}, \frac{L}{G}+1, 1, 1, \text{eq}(L-(2*\frac{L}{G}+3, G-4))]$
12 ...
13 ...
14 $[\frac{L}{G}, ..., \frac{L}{G}, 1, 1]$
15 $[\frac{L}{G}, ..., \frac{L}{G}-1, 1, 2]$

The set of profiling partitionings in Description 1 are the runs that collect the memory statistics for pipeline-parallel-only training. Additional runs are performed to collect profiling data for data- or tensor-parallel training of pipeline stages. Four sets of profiling runs are performed for those scenarios: sets for 2-way and 4-way data-parallelism, as well as for 2- and 4-way tensor parallelism. In those runs, the GPUs are grouped together into S sub-meshes of size k, where k is the degree of data- or tensor parallelism (2 or 4). The generation of the set of runs is similar to Description 1, but the L DNN layers are now divided over S sub-meshes instead of G individual GPUs. Each layer or set of layers is trained with k-way parallelism.

The profiling runs are only performed with 2- and 4-way data- and tensor parallelism to limit the time required for profiling. The predictor applies statistical modeling techniques to predict the memory usage for stages with parallel degrees higher than 4 (see Section III-B).

2) Scaling the batch size: The profiling runs cannot be performed with the same batch size as the target run, because some of the profiling runs have an imbalanced memory usage and could go out-of-memory with the full batch size. Thus, we perform the profiling runs with a smaller batch size. Because the total memory usage (and hence the collected memory statistics) grows linearly with the batch size, but with a different rate for each layer, we perform the profiling runs with two batch sizes (by default half and quarter of the target batch size). The predictor uses the statistics collected with both batch sizes to scale the memory statistics to the target batch size (see Section III-B). The total number of required profiling runs for pipeline-parallel stages, 2- and 4- way data- and tensor-parallel stages, each for 2 batch sizes, is $10*(L-1)$.

3) Reducing profiling time: CAPTURE reduces the time required for profiling in three ways. First, profiling runs are performed with a reduced batch size, as described previously. Scaling down the batch size reduces the time required for the profiling runs. The default batch sizes chosen for profiling can be decreased further to trade accuracy of the memory predictions for reduced profiling time. Second, instead of performing data- and tensor-parallel profiling runs with multiple batch sizes, the memory statistics of parallel stages can be scaled to the target batch size based on the pipeline-parallel profiling data. This eliminates the need to profile with two batch sizes for data- and tensor-parallelism, but could affect the accuracy of the memory predictions. Finally, CAPTURE includes a *layer merger*, which can be used to reduce the number of profiling runs for the data- and tensor-parallel scenarios.

Layer merger: After performing the profiling pass for pipeline parallel stages, the layer merger can link ("merge") layers together based on their combined memory usage. After merging, multiple merged layers are treated as a single DNN layer during the profiling phase for data- and tensor parallelism and during the prediction phase, reducing the number of profiling runs. Merging layers will also reduce the number of predictions made by the predictor.

The merger links layers in a DNN until a user-specified number of layers (L_m) remain. It merges layers based on their combined memory usage when processed without data- or tensor-parallelism. It iteratively chooses the two DNN layers that result in the lowest M_i value for the merged layer. The metrics (M_i and M_a) for the new layer are calculated as follows (merging layers l and $l+1$):

$$M_i(l, l+1) = M_i(l) + M_a(l+1)$$

$$M_a(l) = M_a(l) + M_a(l+1)$$

B. Predictor

The predictor uses the profiling data and statistical modeling to predict the per-GPU peak memory usage for any hybrid-parallel pipeline stage, hardware setup, and batch size. A pipeline stage is defined by three components: (1) the group of DNN layers in the stage, (2) the type of parallelism used to process the stage and (3) the degree of parallelism (number of GPUs assigned to the stage). We call a tuple of (layer group, type of parallelism, parallel degree) a *stage config*. Note that a stage with parallelism degree 1 does not use data- or tensor-parallelism and is referred to as a "pipeline-parallel stage".

1) Predicting memory usage for pipeline-parallel stages: The predictor predicts the memory usage for pipeline-parallel stages as follows:

$$M_p(l, l+n, 1) = M_i(l) + \sum_{k=l+1}^{l+n} M_a(k)$$

where $M_p(l, l+n, 1)$ is the predicted per-GPU peak memory usage of a pipeline stage containing layers l to $l+n$ and parallel degree 1 (no parallelism, so a pipeline-parallel stage).

2) Predicting for the target batch size: Recall that CAPTURE collects the memory statistics for two batch sizes, which are smaller than the target batch size (by default half and quarter of the target batch size). The predictor scales the predicted memory usage for a pipeline stage to match the target batch size. It fits a straight line through the predictions for the two profiled batch sizes and samples the fitted function at the target batch size, as depicted in Description 2.

Description 2: Scaling memory usage to the target batch size. bs_prof_high and bs_prof_low denote the lowest and highest batch size used during profiling, $target\_batch\_size$ denotes the target batch size.

$x = [bs\_prof\_low, bs\_prof\_high]$
$y = [M_p(l, l+n, 1, bs\_prof\_low),$
$\quad M_p(l, l+n, 1, bs\_prof\_high)$
$line = fit\_staight\_line(x, y)$
$M_p(l, l+n, 1, target\_batch\_size) =$
$\quad sample(line(target\_batch\_size))$

3) Statistical modeling for hybrid parallelism: The profiling data contains M_i and M_a measurements for each layer for 2 and 4-way data- and tensor parallelism. For 2- and 4-way parallel stages, we can therefore predict the per-GPU memory usage for hybrid parallel stages with parallel degrees 2 and 4 in similar fashion as for pipeline-parallel stages, including the scaling to the target batch size.

For parallel degrees larger than 4, we predict the memory usage through statistical modeling: during prediction, we fit a logarithmic function through the predicted memory usage of a given stage for the two degrees of parallelism used during profiling and sample the function at the degree of parallelism that we are predicting the memory usage for.

The shape of the fitted function is motivated by high-level analytical models of the per-GPU memory usage for data and

tensor parallelism. For data parallelism, the memory usage is determined by the following components:

$$Mtotal = Mweights + \frac{Mactivations}{nGPUs}$$
$$+Buffer_{stage\_comm}+Buffer_{allreduce}+Buffer_{passthrough}$$

All the components of the memory usage either scale linearly with the number of GPUs, or are fixed regardless of the degree of data parallelism. The passthrough buffers scale linearly with number of GPUs or are fixed (depending on the implementation in the hybrid-parallel framework). The memory for the activations also scales linearly. The other components use a fixed amount of memory. Given that the per-GPU memory usage has a component that is fixed in size and a component that scales linearly (is reduced) when the degree of parallelism increases, the sum of those components scales logarithmically with the degree of parallelism. Hence, we fit a logarithmic function through the collected datapoints.

For tensor parallelism, the various components of the (per-GPU) memory usage are expected to scale differently than for data parallelism, but all components also scale linearly with the number of GPUs, or are fixed. Hence, we also fit a logarithmic function through the collected datapoints for tensor parallelism. The analytical model for per-GPU memory usage of tensor parallel pipeline stages looks like:

$$Mtotal = \frac{Mweights + Mactivations}{nGPUs} + buffers$$

C. Recommender

The recommender finds a memory-efficient partitioning and parallelization plan based on the predicted peak memory usages of pipeline stages provided by the predictor. The recommender takes the number of DNN layers (L), the (target) hardware mesh (number of nodes N and GPUs G) as input. Given L, it generates the list of $\frac{L*(L+1)}{2}$ layer groups that can form a pipeline stage, which is as described in Description 3.

Description 3: All possible layer groups.

1 $[1], [1, 2], [1, 2, 3]...[1, 2, 3, ..., L]$
2 $[2], [2, 3], ...[2, 3...L]$
3 ...
4 $[L-1][L-1, L]$
5 $[L]$

In the target run, each layer group can be processed on a submesh (set of GPUs) of size 1 to (maximum) $k = \frac{G}{N}$, using either data- or tensor parallelism. Recall that each tuple of (layer group, type of parallelism, parallel degree) is called a *stage config*. The recommender determines the possible degrees of data- or tensor parallelism and enumerates all possible stage configs. CAPTURE limits parallelism degrees for data- and tensor parallelism to powers of 2. Thus, if $k = 2^d$, a single stage can be processed with $2 * d$ parallel configurations, where $d = log_2(k)$. The case of 1-way data- or tensor parallelism is a duplicate, because it corresponds to pure pipeline parallelism. After correction, the total number of possible parallel configurations is: $2 * log_2(k) - 1$. After generating all possible stage configs, the recommender obtains the predicted memory usage for each config from the predictor.

The recommender then enumerates all possible placements of stage configs onto the hardware mesh. While placing layer groups onto sub-meshes, it ensures that (1) all the layers of the DNN should be placed and no GPUs should be left unused and (2) the layer groups should fit on the hardware mesh. Concretely, this means that (1) the layer groups should cover all the layers of the DNN, in-order and without duplicates and (2) the degree of data- or tensor parallelism with which a group can be processed is limited by the submesh size k (number of GPUs/node) and by the other layer groups placed on the same submesh: the sum of the degrees of parallelism for all groups on a submesh should be equal to k.

For each valid placement of groups onto the hardware (a *plan*), the recommender tracks the peak memory usage across all GPUs in the plan, using the predicted memory usage of each *stage config* in the plan. After enumerating all possible placements, it recommends the partitioning and parallelization plan with the lowest peak memory usage across all the GPUs. Algorithm 4 summarizes the algorithm for enumerating all possible partitioning and parallelization plans.

Algorithm 4: Algorithm for hybrid-parallel placement of layer groups on the hardware mesh.

```
1  Function place_layer_group(mesh, plan, group) is
2    if not enough groups left for remaining GPUs
3      return
4    if all layer groups placed
       /* Plan complete. Check mem usage.    */
5      if peak_mem(plan) < lowest_peak_mem
6        lowest_peak_mem = peak_mem(plan)
7    else
       /* Try to place all possible stage
          configs for this group.            */
8      foreach parallelism in [DP, TP]
9        foreach degree in range(1, submesh_size)
10         /* Check if config fits on submesh.*/
11         if fits(group, parallelism, degree, mesh)
             /* Place group.                 */
12           mesh = update(mesh, group)
13           plan = update(plan, group)
             /*Proceed to next group         */
14           foreach next_group starting with last layer of
               current group
15             place_layer_group(mesh, plan, next_group);
```

If multiple plans are found with the same peak memory usage across all GPUs, a tie-breaking rule is applied among the remaining plans: the GPU(s) with the highest predicted memory usage are excluded from the memory predictions, and the plan with the lowest memory usage across the remaining GPUs is chosen as the winner. This procedure is repeated until a winner is found or no GPUs remain. If no GPUs remain, the remaining plans are equal and a plan is chosen at random. This tie-breaking rule increases the chance that the best plan is elected as the winner if the memory usage of the excluded GPUs was mispredicted.

Reducing the time needed for recommendation: CAPTURE's recommender applies a number of optimizations to reduce the time needed for the recommendation phase. Applying the *layer merger* reduces the number of layers in the DNN model and thus the number of possible plans for both pipelined and hybrid parallelism. Further, the recommender applies a pruning strategy that reduces the number of possible plans based on the best recommendation found so far (the one with the lowest memory usage): if the predicted memory usage for a given *stage config* is higher than the peak memory usage across all GPUs of the best plan found so far, that stage config is omitted from placement (so not considered in lines 12-13 in Algorithm 4).

IV. Experiments

We evaluate CAPTURE for Alpa, the current state-of-the-art framework that supports all hybrid forms of parallelism (pipelining with per-stage data and tensor parallelism). We experiment with the GPipe and the 1F1B schedule, which are both supported by Alpa. We perform experiments with three large DL models, GPT-3 [14], GShard Mixture-of-experts [15] and Wide-ResNet [16]. Table I shows the different model configurations used in our evaluation. Our hardware setup consists of 4 compute nodes, each containing 4 NVIDIA A100 GPUs with 40 GB HBM2 memory. The nodes are connected through 100Gbps InfiniBand interconnects. We use Alpa 0.2.3, JAX 0.3.22 and CUDA 11.3.

The statistical performance of a model is not dependent on the chosen partitioning and parallelization plan. Thus, we do not explicitly evaluate the statistical performance of the models for the plans chosen by Alpa and CAPTURE. The (profiling) runs in our experiments last for two training iterations and we use randomly generated input data, as provided in Alpa's benchmarking suite. The dimensions of the input samples correspond to those typically used in other research. The contents of the data has very limited influence on our results, as they only affect optimizations such as compression of communicated activations. We use randomly generated data, since it serves as a worst-case scenario for such optimizations in terms of performance and memory usage.

As CAPTURE is based on mCAP and extends its approach to hybrid parallelism, it does not fundamentally differ from mCAP for pipelining only and yields the same result. Hence, we do not explicitly compare the two systems for pipelining only. Section IV-A does compare the results for pipelining only and hybrid parallelism with CAPTURE.

TABLE I. Model configurations used for the evaluation.

Model	# Parameters	Batch size	# micro-batches	# layers
GPT-3	2.6B	1024	64	32
GPT-3	6.7B	1024	32	32
MoE	2.4B	1024	32	16
MoE	7.1B	1024	32	16
Wide-ResNet	1B	768	24	32
Wide-ResNet	2B	768, 1536	24	32

A. End-to-end results / Memory reduction

Fig. 6 compares the peak memory usage of all 16 GPUs in our hardware setup for 12 parallel training scenarios. The figure contains results for the 3 different models in 5 different configurations, both pipelining schedules and 2 types of parallelism (hybrid parallelism and pipelining only).

Alpa's recommended partitioning for hybrid parallelism (as used in Fig. 6a to 6f) is obtained by using its autostaging functionality, which automatically clusters model layers into pipeline stages and recommends a parallelization plan to optimize throughput. In all of these scenarios Alpa is given complete freedom in choosing a plan, except for Wide-ResNet (2B). Since Alpa recommends a plan for that model configuration that fails to run on our hardware setup, we restrict the search space for the degrees of per-stage parallelism ("logical mesh shapes" in Alpa) to match the layout of the real hardware ("physical mesh shapes"). This setting provides the best performing and most memory-friendly alternative plan that successfully runs on our hardware setup.

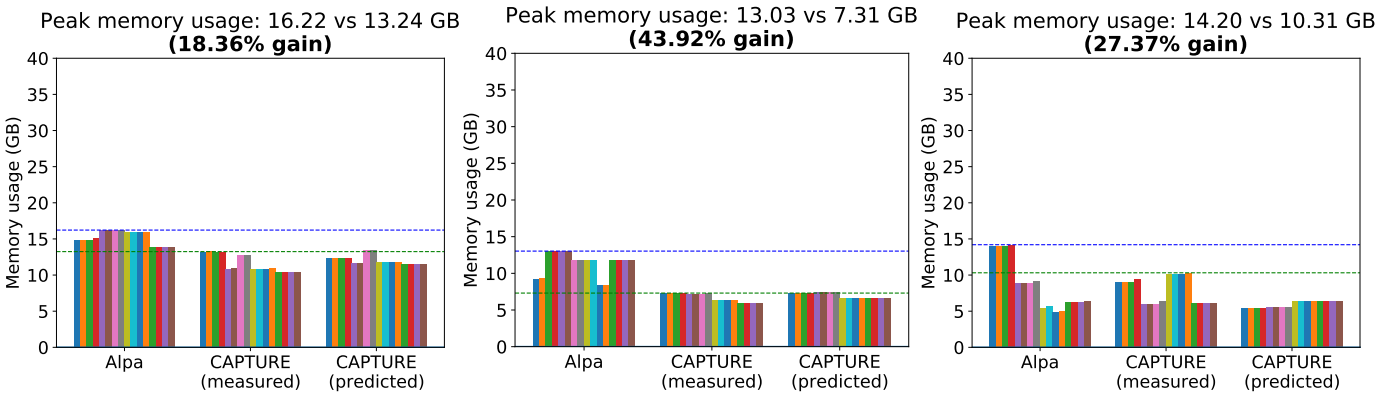
Alpa's recommended partitioning for pipelining only (as used in Fig. 6g to 6l) is obtained by requesting 16 layers from Alpa's "autolayering" component, which splits the DL model in x equally compute intensive layers, where x is a programmer-defined number. We request 16 such layers and assign one layer to each pipeline stage/GPU. Note that the maximum possible value of x depends on the DL model.

Fig. 6a to 6c show the results for hybrid parallelism using the 1F1B schedule. The observed reduction in peak memory usage is between 18.36% and 43.92% compared to Alpa. Not only does CAPTURE recommend a parallelization plan that has a lower peak memory across the GPUs than Alpa for GPT, MoE and Wide-ResNet, its plan also has a lower combined memory footprint. Since CAPTURE does not explicitly adjust the memory usage predictions for the 1F1B schedule, it underpredicts the memory usage for Wide-ResNet.

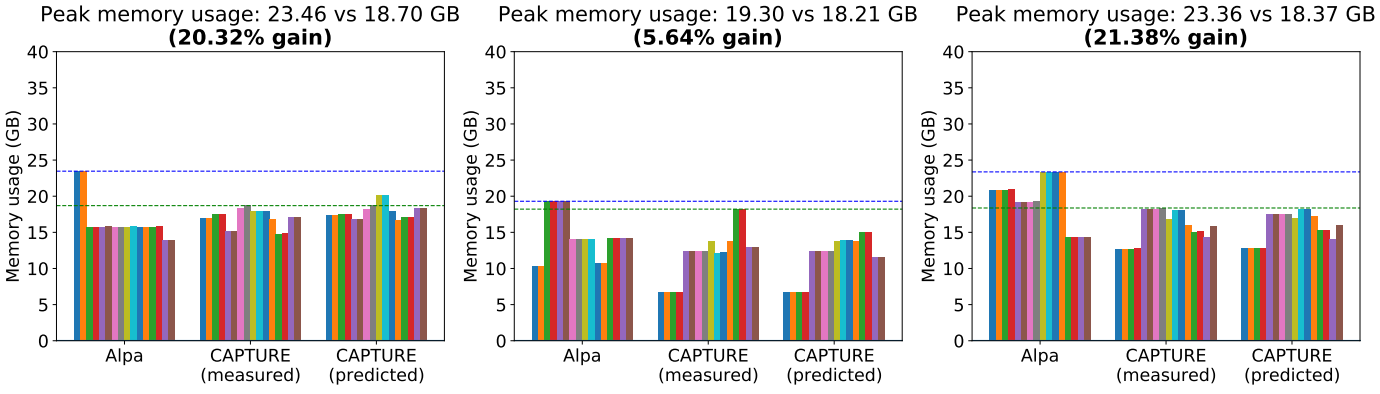
Fig. 6d to 6f show the results for hybrid parallelism using the GPipe schedule. Compared to Alpa, CAPTURE reduces the peak memory usage with 5.64% to 21.38% and in most cases accurately predicts the peak memory usage. As expected, the observed memory usage is higher than for the 1F1B schedule for all model configurations.

Fig. 6g to 6i show the results for pipeline parallel-only training, using the 1F1B schedule. As MoE has only 16 layers (the maximum value of x), only a single partitioning is possible with pipelining only on 16 GPUs. Reducing or better balancing the memory usage requires the use of hybrid parallelism (Fig. 6b and 6e). Also, scaling to more GPUs is not possible without hybrid parallelism, as there are not enough model layers to do so. CAPTURE establishes a memory gain of 19.38% to 25.26% for the other two models, despite some mispredictions related to the 1F1B schedule.

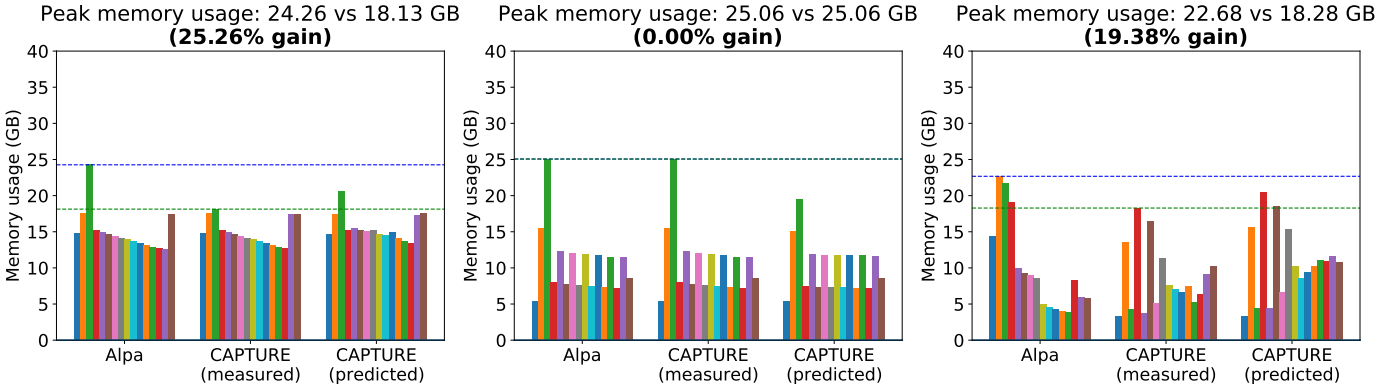
Fig. 6j to 6l show the results for pipeline parallelism only, using the GPipe schedule. As for 1F1B, MoE can only be trained with a single partitioning and shows a large imbalance in memory usage between the GPUs, motivating the use of hybrid parallelism. CAPTURE chooses the same partitioning



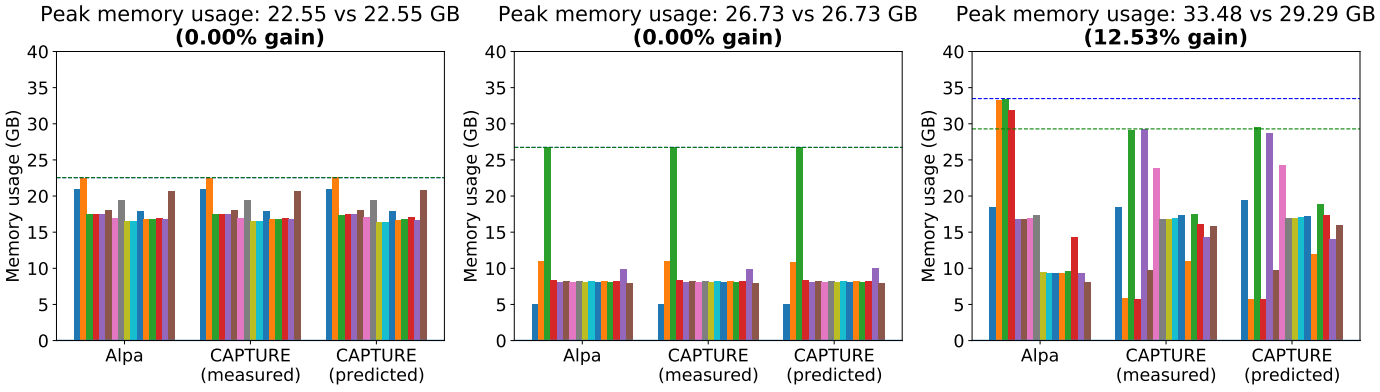
(a) GPT-3 (6.7B), 1F1B, hybrid parallelism (b) MoE (7.1B), 1F1B, hybrid parallelism (c) Wide-ResNet (2B, 1536), 1F1B, hybrid parallel



(d) GPT-3 (2.6B), GPipe, hybrid parallelism (e) MoE (7.1B), GPipe, hybrid parallelism (f) Wide-ResNet (2B, 768), GPipe, hybrid parallelism



(g) GPT-3 (6.7B), 1F1B, pipelining only (h) MoE (7.1B), 1F1B, pipelining only (i) Wide-ResNet (2B, 768), 1F1B, pipelining only



(j) GPT-3 (2.6B) GPipe, pipelining only (k) MoE (2.4B), GPipe, pipelining only (l) Wide-ResNet (2B, 768), GPipe, pipelining only

Fig. 6. Per-GPU memory usage for Alpa's and CAPTURE's parallelization plans for various configurations of GPT-3, MoE and Wide-ResNet. Results are shown for hybrid parallelism and pipeline parallelism only, as well as for the 1F1B and GPipe pipelining schedules. Each colored bar represents the memory usage of a single GPU. The dotted lines represent the peak memory usage across the GPUs of Alpa's (blue) and CAPTURE's (green) parallelization plan.

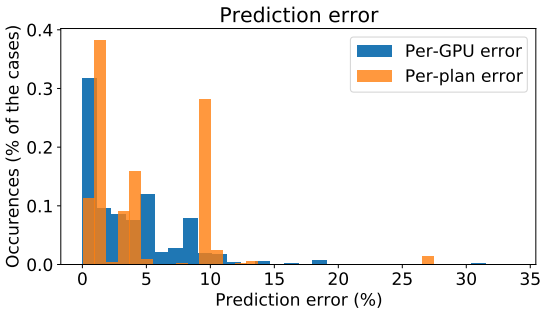


Fig. 7. Prediction error histogram for 500 plans for Wide-ResNet (1B).

as Alpa for GPT-3, leading to no memory gain. CAPTURE achieves 12.53% memory gain over Alpa for Wide-ResNet and accurately predicts the memory usage for all models. For all pipelining-only scenarios, the potential memory gain is limited, because further memory reduction and better balancing of the memory between workers requires the parallelization of layers to multiple GPUs (hybrid parallelism).

B. Prediction accuracy

To evaluate the accuracy of CAPTURE's peak memory predictions in more detail, we compare the predicted and real memory usage of 500 different partitioning and parallelization plans for Wide-ResNet (1B), using the GPipe schedule and hybrid parallelism. The plans are randomly sampled from the set of possible plans that the recommender generates. Plans that are predicted to run out of memory and pruned options are excluded from sampling. The memory usage is predicted for 500 plans, each using 16 GPUs, so the total number of per-GPU predictions made is 8000. Fig. 7 shows a histogram of the error for the predictions for individual GPUs and that of the peak across all GPUs for a single plan (per-plan error). 44.8% of the per-GPU predictions is within the 2% error margin, 65.5% within 5% and 97.1% within 11%. For the per-plan predictions that is 45.4%, 69.6% and 97.8%, respectively.

There are relatively more predictions within the 9%-11% error range for per-plan predictions than for per-GPU predictions, because the per-plan prediction only considers the highest memory usage across all the GPUs/pipeline stages. It is likely that the stage with the highest memory usage consists of a relatively large number of layers compared to the other stages in the pipeline. Having more layers in a pipeline increases the chance of prediction errors, because CAPTURE takes multiple aspects into account during prediction for each layer, such as the level of data- or tensor parallelism and scaling of the batch size and then combines the prediction for a layer with the predictions for other layers in the same stage.

We identify two possible causes for occasional mispredictions of per-GPU memory usage. In the 1F1B schedule, the memory usage of a *stage config* can vary based on the location of the config in the pipeline, as described in Section II-B. CAPTURE does not actively adjust its memory predictions to this behavior, causing occasional over- or under-estimations of peak memory usage, such as in Fig. 6c. Other mispredictions, such as for the GPipe schedule in Fig. 6e are caused by the limited number of values for n (neighbors) used to extract M_a for each layer (see Section III-A1). Profiling with multiple values of n can improve the accuracy of the predictions further, but requires more profiling runs.

TABLE II. Time (denoted as hours:minutes:seconds) required by CAPTURE and Alpa to recommend a partitioning and parallelization plan for hybrid parallelism. LM stands for layer merger, LM=16 denotes that the layer merger is applied to reduce the number of layers to 16. * Runtimes are estimated based on the number of remaining layers and the number of required profiling runs.

	CAPTURE				Alpa
Model	Profiling (No LM)	Planning (No LM)	Total (No LM)	Total (LM=16*)	Total
GPT-3 6.7B	09:20:07	00:31:18	**09:51:25**	**05:24:04**	**03:53:12**
MoE 7.1B	03:57:33	00:05:94	**04:04:07**	**04:04:07**	**03:07:10**
WResNet 2B	07:45:10	00:00:17	**08:02:25**	**04:24:20**	**03:04:39**

C. Profiling and planning time

Table II shows the time needed by CAPTURE and Alpa to generate a hybrid partitioning and parallelization plan for the three largest models in our setup, for the 1F1B schedule. CAPTURE's runtime consists of profiling (profiler) and planning (predictor and recommender). Alpa's runtime consists of profiling various computations and communication operations, profiling various parallel stage configurations and applying its autostaging functionality. The majority of the runtime is spent on profiling for both systems.

CAPTUREs planning takes between 17 seconds and 32 minutes (when the layer merger is not applied). The planning time is kept low by the pruner for MoE and WResNet, but is less effective for GPT-3 because of the repetitive nature of its model architecture: many layers exhibit similar memory usage, reducing the pruner's effectiveness.

Without applying the layer merger, CAPTURE's total runtime is between 4 and 10 hours, while Alpa's runtime is between 3 and 4 hours. Applying the layer merger reduces CAPTUREs runtime and brings it closer to Alpa's: merging to 16 layers reduces the number of profiling runs and plans traversed by the recommender and halves the runtime for MoE and GPT-3. WResNet's runtime does not change, as it already consists of 16 layers without applying the layer merger.

The time needed to recommend a plan is negligible compared to the runtime of the target training run: it takes weeks to fully train these models on our hardware setup. Hence, we consider the additional overhead introduced by CAPTURE compared to Alpa's throughput-oriented planner negligible.

D. Throughput

Fig. 8 summarizes the reduction in memory usage for all hybrid parallel scenarios in Fig. 6 and shows the loss in throughput that results from partitioning and parallelizing for memory usage. For the configurations using the 1F1B schedule the throughput loss is between 11.5 and 42.4%, while the gain in memory usage is between 18.4 and 43.9%. For the GPipe schedule, the case of MoE (7.1B) stands out, with a loss in throughput of over 60% and a memory gain of 5.6%. In contrast, GPT-3 records the lowest loss in throughput for this schedule with 26.2%, while obtaining 20.3% memory gain.

As demonstrated next, the loss in throughput can be compensated by taking advantage of the gain in memory usage,

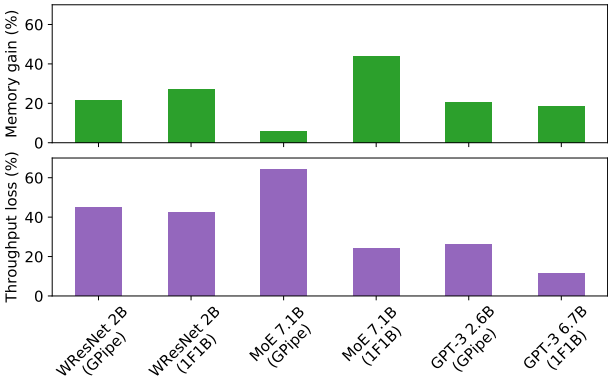


Fig. 8. Memory gain and throughput loss for all hybrid parallel scenarios.

by training on a smaller hardware setup. Alternatively, the extra memory headroom can be used to train a larger model or increase the batch size.

E. Smaller hardware setup

Fig. 9 shows the achieved throughput when training Wide-ResNet (1B) on 6, 8 and 16 GPUs with CAPTURE's and Alpa's parallelization plans for hybrid parallelism, normalized to Alpa's throughput on 16 GPUs. Alpa runs out of memory on 8 GPUs, while CAPTURE is able to train the model. Moreover, CAPTURE can scale the hardware setup down further to 6 GPUs without running out of memory. Training on less GPUs increases hardware utilization and reduces communication overhead, which results in an increase in achieved throughput.

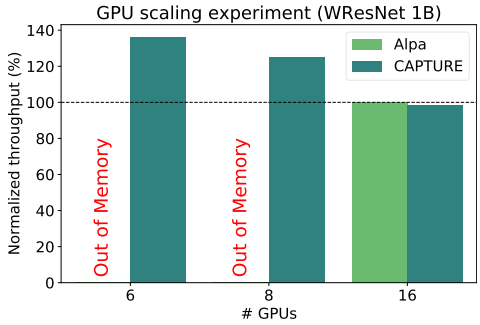


Fig. 9. Normalized throughput when scaling the hardware setup for training Wide-ResNet (1B) with the GPipe schedule and hybrid parallelism.

This experiment demonstrates how CAPTURE can train a given model in a more cost-effective way. By using the reduction in memory usage to train on a smaller hardware setup, less resources are used and all of the loss in throughput is compensated. In this experiment, CAPTURE trains a DL model on less than half the hardware resources while increasing training throughput by 36.3%, making the training significantly more cost-efficient.

V. Conclusion

We introduced CAPTURE, a method to generate memory-efficient partitioning and parallelization plans for hybrid parallel DNN training. CAPTURE combines profiling and statistical modeling to make accurate predictions of peak memory usage and recommends a plan that minimizes peak memory usage across GPUs. Our method can recommend a memory-efficient plan for any target batch size and hardware setup size.

By reducing the peak memory usage CAPTURE enables training of larger models, training on a smaller hardware setup and training in a more cost-effective way than existing approaches. CAPTURE provides the user with the flexibility to choose how to make use of the extra memory headroom, even when a model does not fully occupy the GPUs' memories. We demonstrated that CAPTURE reduces memory usage by up to 43.9% and can train DNNs on more than two times smaller hardware setups.

We suggest further improvement of memory prediction accuracy for future work, as well as alternative (e.g. throughput-based) tie-breaking rules for the recommender. Another option is to generalize our approach to optimize GPU utilization, energy consumption, throughput, or a combination of these objectives and memory usage when partitioning DL models. Finally, CAPTURE can be applied to large-scale inferencing.

VI. Acknowledgements

This work is part of the Efficient Deep Learning (EDL) programme (grant number P16-25), financed by the Dutch Research Council (NWO). We thank the anonymous reviewers for their valuable feedback. We thank SURF for the support in using the National Supercomputer Snellius.

References

[1] H. Dreuning *et al.*, "mCAP: Memory-Centric Partitioning for Large-Scale Pipeline-Parallel DNN Training," in *Euro-Par*. Springer, 2022, pp. 155–170.
[2] L. Zheng *et al.*, "Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning," in *OSDI*, 2022, pp. 559–578.
[3] S. Athlur *et al.*, "Varuna: Scalable, Low-cost Training of Massive Deep Learning Models," in *EuroSys*, 2022, pp. 472–487.
[4] Y. Huang *et al.*, "Gpipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," in *NeurIPS*, 2019, pp. 103–112.
[5] D. Narayanan *et al.*, "PipeDream: Generalized Pipeline Parallelism for DNN Training," in *SOSP*, 2019, pp. 1–15.
[6] S. Fan *et al.*, "DAPPLE: A Pipelined Data Parallel Approach for Training Large Models," in *PPoPP*, 2021, pp. 431–445.
[7] D. Narayanan *et al.*, "Memory-Efficient Pipeline-Parallel DNN Training," in *ICML*. PMLR, 2021, pp. 7937–7947.
[8] J. Rasley *et al.*, "DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters," in *KDD*, 2020, pp. 3505–3506.
[9] M. Shoeybi *et al.*, "Megatron-LM: Training Multi-Billion Parameter Language Models Using GPU Model Parallelism," *arXiv preprint arXiv:1909.08053*, 2019.
[10] D. Narayanan *et al.*, "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM," in *SC*, 2021, pp. 1–15.
[11] Z. Lai *et al.*, "Merak: An Efficient Distributed DNN Training Framework with Automated 3D Parallelism for Giant Foundation Models," *TPDS*, pp. 1466–1478, 2023.
[12] Z. Zong *et al.*, "STR: Hybrid Tensor Re-Generation to Break Memory Wall for DNN Training," *TPDS*, 2023.
[13] Y. Li *et al.*, "Harmony: Overcoming the Hurdles of GPU Memory Capacity to Train Massive DNN Models on Commodity Servers," *VLDB*, 2022.
[14] T. Brown *et al.*, "Language Models are Few-Shot Learners," *NeurIPS*, vol. 33, pp. 1877–1901, 2020.
[15] D. Lepikhin *et al.*, "GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding," *arXiv preprint arXiv:2006.16668*, 2020.
[16] S. Zagoruyko *et al.*, "Wide Residual Networks," *arXiv preprint arXiv:1605.07146*, 2016.