CAPSlog: Scalable Memory-Centric Partitioning for Pipeline Parallelism

Henk Dreuning*†, Anna Badia Liokouras,†, Xiaowei Ouyang†, Henri E. Bal† and Rob V. van Nieuwpoort‡
* University of Amsterdam, Amsterdam, The Netherlands
Email: h.h.h.dreuning@uva.nl
† Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
Email: h.h.h.dreuning@vu.nl, annabadia8@gmail.com, xiaowei.ouyang0@hotmail.com, h.e.bal@vu.nl
‡ Leiden University, Leiden, The Netherlands
Email: r.v.van.nieuwpoort@liacs.leidenuniv.nl

***Abstract*—Pipeline-parallel training has emerged as a popular method to train large Deep Neural Networks (DNNs), as it allows the use of the combined compute power and memory capacity of multiple Graphics Processing Units (GPUs). However, with the sustaining increase in Deep Learning (DL) model sizes, pipeline parallelism provides only a partial solution to the memory bottleneck in large-scale DNN training. Careful partitioning of the DL model over the available GPUs based on memory usage is required to further alleviate the memory bottleneck and train larger DNNs. mCAP is such a memory-oriented partitioning approach for pipeline parallel systems, but it does not scale to models with many layers and very large hardware setups, as it requires extensive profiling and fails to efficiently navigate the partitioning space to find the most memory-friendly partitioning.**

In this work, we propose CAPSlog, a scalable memory-centric partitioning approach that can recommend model partitionings for larger and more heterogeneous DL models and for larger hardware setups than existing approaches. CAPSlog introduces a new profiling method and a new, much more scalable algorithm for recommending memory-efficient partitionings. CAPSlog reduces the profiling time by 67% compared to existing approaches, searches the partitioning space for the optimal solution orders of magnitude faster and can train significantly larger models.

***Index Terms*—Deep Learning, Pipeline Parallelism, Memory**

I. Introduction

The sustaining increase in Deep Learning (DL) model sizes has provided state-of-the-art results in many application domains, but also increases the computational demand and memory requirements for training Deep Neural Networks (DNNs). The memory bottleneck is especially apparent when training on Graphics Processing Units (GPUs), due to their limited memory size. Pipeline-parallel training systems have been developed that allow the use of the combined compute power and memory capacity of multiple GPUs. However, pipeline parallelism only provides a partial solution to the memory bottleneck in large-scale DNN training.

Careful partitioning of the DL model over the available GPUs based on memory usage can further alleviate the memory bottleneck. Most partitioning approaches for pipeline parallel DNN training focus on achieving high computational throughput while partitioning the model, but do not address the memory bottleneck. To the best of our knowledge, our previous work mCAP, [1] is the only memory-centric partitioning approach that enables training of larger models by balancing memory usage between workers, which reduces the overall peak memory usage. The extra memory headroom can be used to train larger models.

However, mCAP does not scale to very large and heterogeneous (in terms of per-layer memory usage) models. To recommend a partitioning for a given model, it profiles the DNN layers' memory usage by performing a few training iterations with a number of specifically selected model partitionings, called *profiling partitionings*. It keeps the full DL model in memory when profiling the layers, which limits the size (in terms of memory usage) of models that can be profiled. Also, a high variation in memory usage between the model layers can cause the profiler to run out of memory.

Moreover, mCAP cannot scale to DL models with hundreds of layers nor very large hardware setups. It provides two search strategies to find the best balanced partitioning from the full partitioning space. The default brute-force strategy has combinatorial time complexity in terms of the number of layers in the model (L) and the number of GPUs in the hardware setup (G), while the faster bayesian optimization strategy does not guarantee optimal search results. We found that for large G or models with large L, the search time of the brute-force strategy increases to infeasible levels, while the bayesian optimization approach fails to find a sufficiently balanced partitioning to achieve significant memory gain.

In this work, we introduce CAPSlog, a memory-centric partitioning approach that is scalable in terms of hardware size, number of model layers and heterogeneity of the model layers. CAPSlog introduces a new method to profile DL models based on *model trimming* and provides a new approach to generate the set of *profiling partitionings*. It also introduces a new and much more scalable *binary search*-based algorithm to find a memory-friendly partitioning from the full partitioning space.

CAPSlog's objective and main approach correspond to that of mCAP: it increases trainable model size by partitioning DL models in a way that reduces the *overall* peak memory usage, e.g. the peak memory usage across all GPUs. It finds a partitioning with low overall memory usage, creating more memory headroom that can be used to train larger, more accurate, models. The extra memory headroom can also be used to train on smaller hardware setups or to increase the

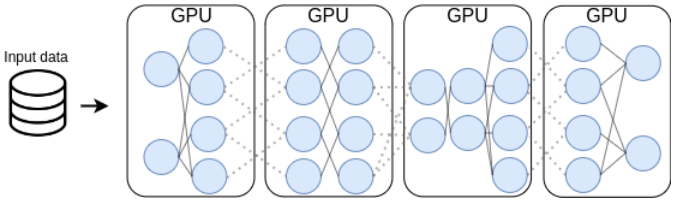

Fig. 1. A DNN partitioned over multiple GPUs for pipeline-parallel training.

batch size, both of which can improve hardware utilization. Unlike mCAP, CAPSlog scales to models with hundreds of layers and very large hardware setups.

CAPSlog applies *model trimming* to remove parts of the model that are (temporarily) not required in memory for profiling and provides a new method to generate the set of *profiling partitionings*. Combining these mechanisms eliminates the risk to run out of memory during profiling and reduces the time and hardware resources required for profiling. Hence, CAPSlog can profile larger and more heterogeneous models.

CAPSlog's new, *binary search*-based recommendation algorithm has logarithmic time complexity (after which CAPSlog is named), which searches for the lowest possible peak memory usage achievable for a given DL model by iteratively lowering a virtual memory threshold. The algorithm reduces the recommendation time from hours to milliseconds and can recommend memory-friendly partitionings for models with hundreds of layers and for large hardware setups.

We implement CAPSlog for Varuna [2], a state-of-the-art pipeline-parallel framework, and evaluate its effectiveness on DETR [3], an object detection model with a heterogeneous model structure and up to 48 layers. We evaluate CAPSlog on DETR because it provides a single, scalable model that has been shown to achieve higher statistical performance when scaled in various dimensions. We show that CAPSlog's scalable profiling and recommendation methods are required to train configurations of DETR that have been scaled in multiple dimensions simultaneously on a distributed hardware setup. Note that training with a different model partitioning does not alter the training process mathematically. Hence, we do not explicitly evaluate the accuracy achieved by DETR when trained with CAPSlog's recommended partitioning compared to other partitioning approaches.

Concretely, we make the following contributions:

- We introduce CAPSlog, a memory-centric partitioning approach for pipeline-parallel DNN training that is scalable in terms of the number of DNN layers and hardware setup size and supports larger and more heterogeneous DL models than existing memory-centric partitioners.
- We introduce a novel profiling approach based on *model trimming* that reduces profiling time and supports larger models, as well as a new *binary search*-based method that reduces the recommendation time to milliseconds.
- We demonstrate CAPSlog's effectiveness in recommending a memory-friendly partitioning by applying it to the DETR object detection model.

The rest of this paper is structured as follows: Section II provides background information and an overview of related work. Section III provides an overview of CAPSlog and discusses its components and design. Section IV evaluates CAPSlog's effectiveness and scalability, Section V concludes the work and provides suggestions for future research.

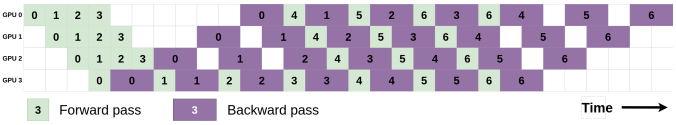

Fig. 2. 1F1B pipelining schedule.

II. Background and Related Work

A DNN is trained by feeding samples of input data into the model. For each input sample three operations are performed: a forward pass, in which the model produces a prediction for the given input; a backward pass that, based on the prediction and the expected output, calculates a gradient for each weight in the model that dictates how the weight should be adjusted to improve the prediction for the given input; and an update step, in which the weights are updated based on the gradients. During the forward pass, each layer produces intermediate outputs (activations) that flow through the neural network from the first to the last layer. In the backward pass, gradients flow through the network in the opposite direction. Input samples are typically processed in batches, referred to as *minibatches*.

In **pipeline-parallel** training [4]–[7], the DL model is partitioned over multiple workers (GPUs), as illustrated in Fig. 1. In this work, we focus on *intra-batch* or *synchronous* pipelining, where a minibatch is split into multiple microbatches that are fed into the model in a pipelined manner. Each worker performs forward and backward passes for all the microbatches in a minibatch and then updates its layers' weights. Only after the weight update, the next minibatch is fed into the pipeline.

The order in which the forward and backward passes are performed is determined by the *pipeline schedule*. In the GPipe schedule [4], the forward passes for all microbatches in one minibatch are performed before all backward passes. The 1F1B schedule [5] alternates between forward and backward passes, so that activations can be released from memory earlier on GPUs later in the pipeline (see Fig. 2).

Varuna [2] is a pipelining framework that targets the training of large DL models on cheap preemptible cloud resources (e.g. spot-VMs), to make training of massive DL models more affordable. Instead of assuming the availability of expensive and specialised GPUs and networking infrastructure, Varuna targets commodity hardware. It uses an adjusted 1F1B pipelining schedule to add resilience to network jitter and applies "morphing" to deal with preemption: when the number of available GPUs decreases during runtime (due to preemption), it chooses a new model partitioning and continues to train on a smaller hardware setup. It performs intra-batch pipelining and supports per-stage data parallelism, although we focus on pipeline-parallel only training in this work. The authors of Varuna opt not to use per-stage intra-operator parallelism because the communication overhead that

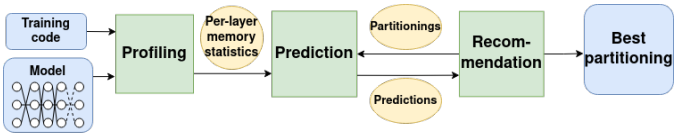

Fig. 3. Overview of mCAP. Figure adapted from [1].

comes with splitting layers across devices is suboptimal for low-bandwidth commodity networks. Varuna is built on the PyTorch DL framework [8]. Other pipelining systems, such as GPipe [4], Merak [9] and Alpa [10] focus on training on specialized clusters with stable networks and no preemption.

CAPSlog is based on **mCAP**, our memory-centric partitioning approach for pipeline-parallel training. It partitions DL models in such a way that the memory usage is balanced across GPUs. Balancing the memory usage reduces the *overall* (across all GPUs) peak memory usage and enables the training of larger DL models. mCAP consists of a profiling, prediction and recommendation stage, as illustrated in Fig. 3.

In the profiling stage, mCAP collects two metrics for each layer in the neural network: the peak memory usage when training that layer on a GPU in isolation (with no other layers placed on the same GPU), and the change in peak memory usage caused by adding that layer to an existing set of layers on a GPU. Both metrics need to be recorded, because the memory usage differs significantly in both scenarios due to memory optimizations in DL and pipelining frameworks and the presence of communication buffers and passthrough variables. The metrics are recorded by performing a few training iterations with a specifically selected set of partitionings, called *profiling partitionings*.

The metrics collected in the profiling stage are input to the predictor, which predicts the memory usage for any partitioning requested by the recommender. The recommender searches for a well-balanced partitioning of the model over the available GPUs, using one of two search strategies. The brute-force strategy requests memory predictions for all possible partitionings and recommends the one with the lowest peak memory usage. The bayesian optimization strategy finds a partitioning within a fixed number of iterations, but does not guarantee to find the optimal solution.

The number of profiling runs required by mCAP is linear to the number of layers in the DL model: $L-G+1$ (where G is the number of GPUs in the hardware setup, and $L >> G$ typically holds). Profiling takes in the order of minutes per run. The search strategies in mCAP's recommender take seconds to hours, depending on L and G, making them unsuitable to perform live recommendations for Varuna's morphing.

CAPSlog reduces the time needed for profiling and can partition larger models. It finds a memory-friendly partitioning in milliseconds, fast enough for live morphing.

Several other works balance memory usage between workers in pipeline-parallel systems. BPipe [11] moves activations between workers to mitigate the memory imbalance introduced by the 1F1B schedule. Bidirectional pipelining systems, such as Chimera [12] and MixPipe [13] adopt novel pipelining schedules and adjust the number of microbatches that are fed into the pipeline to increase device utilization and throughput. While these approaches lead to more balanced memory usage, they do not reduce the overall peak usage and do not increase trainable model size. They are specialized to models with a repetitive structure, such as Large Language Models, and assume a simplistic model partitioning. Our work provides a scalable method to increase trainable model size for heterogeneous DL models through memory-oriented partitioning.

III. Method

We introduce CAPSlog, a memory-centric partitioning approach that is scalable in terms of hardware size and number of model layers and can profile heterogeneous models[1]. CAPSlog is a partitioner that recommends a partitioning (mapping of DL model layers to GPUs in the pipeline) that has a low peak memory usage. We specifically look at the highest peak memory usage across all GPUs, as that determines how much memory headroom there is (how far the peak memory is from the memory capacity of the GPUs). The more memory headroom, the further the model can be scaled up. Scaling the model typically increases AI performance and can also improve achieved computational throughput. Alternatively, the memory headroom can be used to train with a larger batch size (increasing throughput) or the model can be trained on a smaller hardware setup without changing any parameters (increasing throughput and saving financial cost).

CAPSlog recommends a memory-centric partitioning in two steps, and thus consists of two components: a profiler and a recommender. The profiler performs a number of profiling runs to collect two metrics about each layer in the neural network. The predictor component of the recommender can use these metrics to predict what the memory usage will be for any given model partitioning. The recommender performs a binary search and uses the predictor component to predict the memory usage for a number of partitionings needed for the search. It then outputs the recommended partitioning. We will discuss each of CAPSlog's components in detail in this section.

A. Profiling and Model Trimming

CAPSlog collects two metrics for each layer in the neural network, M_i and M_a, as proposed by mCAP. M_i is the memory that a DL model layer occupies in memory when it is placed on a GPU in isolation (as the only layer), while M_a is the increase in memory usage caused by adding a layer to an already existing set of layers on a GPU. The recommender uses these statistics to search a model partitioning with a low overall peak memory usage (see Section III-B).

Like mCAP, CAPSlog generates a number of partitionings of the model that are used to profile the memory usage of the DNN layers, called the *profiling partitionings*. We run a few training iterations (forward pass, backward pass and update step) with the profiling partitionings and record the memory usage of the GPUs in the setup, so that we can extract M_i

[1]CAPSlog's source code is available at https://github.com/henkdr/CAPSlog.

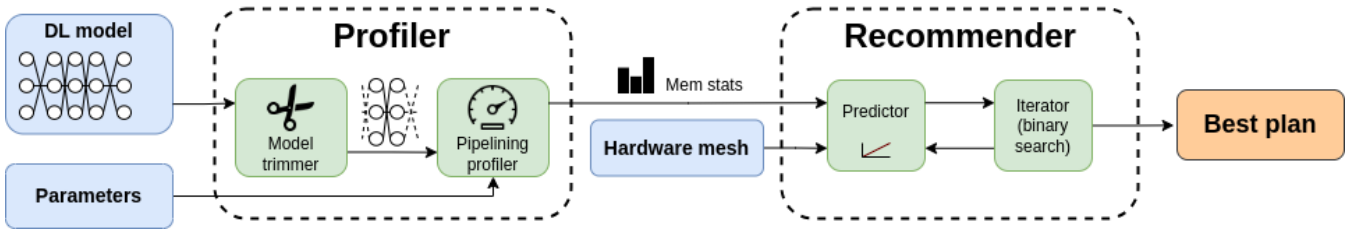

Fig. 4. Overview of CAPSlog.

and M_a for each layer in the DL model. Since the profiling runs are performed in the same setup (hardware configuration and model (hyper)parameters) as the target training run, the profiling data captures the effects of memory optimizations, pipelining and framework characteristics on memory usage. It also captures all components of memory usage: model weights, intermediate activations, gradients and optimizer state.

The set of profiling partitionings P must meet three criteria to ensure that M_i and M_a can be extracted from the profiling data for each layer l of the model. For each layer l, each of the following criteria must be met by at least one partitioning in P:

C1: l is isolated on a GPU.
C2: There is a set of layers $\{l-n, ..., l-1\}$ on a GPU
C3: There is a set of layers $\{l-n, ..., l\}$ on a GPU

Criterion C1 is necessary to obtain M_i, while fulfilling Criteria C2 and C3 allows for calculating M_a. Note that for the first layer of a model, only Criterion C1 has to be fulfilled, as it does not make sense to compute M_a for this layer.

Unlike mCAP, CAPSlog generates a set of profiling partitionings in such a way that the number of profiling runs is minimized and at the same time, the runtime of each profiling run is minimized. The strategy used to minimize the runtime of each individual profiling run is based on the assumption that we can perform *model trimming*: we can remove those parts of the DL model that are not of interest in a given profiling run, omit performing computations for those parts of the model and replace their in- and output-data by dummy values. We can do this, because the memory usage of the layers that we are profiling does not depend on the actual values of their inputs and outputs. CAPSlog assumes the possibility of model trimming to minimize the time taken for profiling. Furthermore, criteria C2 and C3 require profiling runs with at least 2 consecutive layers of the model placed on a single GPU (when $n = 1$). This results in the following two concrete assumptions for CAPSlog's profiling stage:

A1: Any two consecutive layers in the model fit in the memory of a single GPU.
A2: The model can be *trimmed*.

Assumption A1 is a minimal assumption necessary to calculate M_a for each layer of the DL model. Assumption A2 removes the constraint that the entire model must fit on the available GPUs during profiling.

We generate the set of profiling partitionings to collect M_i for each layer of a DL model inductively. To profile a model with L layers on G GPUs, the first partitioning is:

$$Pi_0 : [1, 1, ..., 1, L-(G-1)]$$

Let the previously generated partitioning be $[X, 1, 1, ..., 1, Y]$, and let Z be the sum of all but the last value in this partitioning: $Z = X + (G-2)$. The next partitioning is then:

$$[Z, 1, 1, ..., R]$$

where R is the number of remaining layers: $R = L - Z - (G-2)$. The final partitioning is

$$[Z, 1, 1, ..., 1]$$

where Z is defined as above.

The set of profiling partitionings that CAPSlog generates is illustrated by Fig. 5. In the first partitioning P_0, it places the first $G-1$ layers of the model on the first $G-1$ GPUs, while the remaining layers are placed on the last GPU. In the next profiling runs, all layers that have already been profiled in isolation are placed together on GPU 0 and the consecutive $G-2$ layers are placed in isolation on GPUs 1 to $G-2$. The remaining layers are placed on the last GPU. This process is repeated until all layers have been profiled in isolation. Hence, in the last run (run 2 in Fig. 5), most of the model's layers are placed on GPU 0 and the remaining $G-1$ layers are placed in isolation on the other GPUs. Note that fewer than G GPUs may be used for the final profiling runs in case there are not enough layers to place at least one layer on each GPU.

To reduce the number of required profiling runs, we fix $n = 1$ in the definition of Criteria 2 and 3. As a result, a single profiling partitioning can fulfil both criteria C1 and C2: the partitioning that satisfies C1 for layer $l-1$, also fulfills C2 for layer l. To compute M_a for each layer, we are only missing profiling runs with the layers $\{l-1, l\}$ placed together on a GPU. We generate those partitionings inductively, in similar fashion as for M_i, but place 2 layers on each relevant GPU at a time.

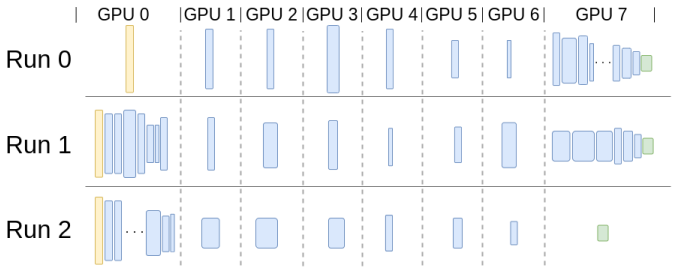

Fig. 5. Profiling partitionings to extract M_i for each layer of the DL model.

The first two partitionings are:

$$[2, 2, ..., 2, L - 2(G-1)] \text{ and } [1, 2, 2, ..., 2, L - 2(G-2) - 1]$$

Let the two previously generated partitionings be $[X, 2, 2, ..., 2, Y]$ and $[X - 1, 2, 2, ..., 2, Y + 1]$ and let Z be the sum of all but the last value in the first of these partitionings: $Z = X + 2(G-2)$. The next two partitionings are then:

$$[Z, 2, 2, ..., 2, R] \text{ and } [Z - 1, 2, 2, ..., 2, R + 1]$$

where R is the number of remaining layers: $R = L - Z - 2(G-2)$. The final partitionings are:

$$[Z, 2, 2, ..., 2] \text{ and } [Z - 1, 2, 2, ..., 2]$$

where Z is defined as above. Note that the final stage in either of the two final partitionings may only hold 1 layer, depending on the parity of the remaining number of layers. Model trimming is applied on the same GPUs as before.

Model trimming is performed by removing the irrelevant layers of the DL model before the model is moved to the GPUs' memories. However, the GPUs that host relevant layers are expecting to receive input activations from the downstream GPU and send their output activations to the upstream GPU. Moreover, they calculate gradients based on input received from the upstream GPU. Since the layers on the downstream and upstream GPUs are trimmed, the activations and gradients generated by those layers are not available. Instead, CAPSlog generates tensors with dummy data of the correct size on the next downstream and upstream GPU, and communicates those instead. This ensures that a situation is emulated in which the profiling setting does not differ from the setting of the target training run, leading to correct profiling data and predictions of peak memory usage in the recommendation stage.

Model trimming allows CAPSlog to generate a set of profiling partitionings that captures M_i and M_a for each model layer, without making assumptions about how many layers can fit on a single GPU (beyond the minimum requirement set by assumption A1). This makes CAPSlog less prone to running out of memory during profiling than mCAP and makes it more scalable in terms of the number of model layers and the heterogeneity of the model structure.

Applying model trimming also increases the space of possible profiling partitionings, as the entire model no longer needs to fit on the available GPUs. Without this constraint, more relevant data can be extracted from a single profiling run. Hence, CAPSlog can generate and perform a smaller set of profiling runs than mCAP, cutting down the profiling time.

Furthermore, no computations (forward pass, backward pass and update step) are performed for the trimmed layers during profiling. This significantly reduces the runtime of each individual profiling run compared to mCAP's approach. We evaluate the gain in profiling time in Section IV.

B. Recommender

The recommender uses the profiling data generated to find a partitioning of the DL model that has a low peak memory usage across all GPUs. It consists of two subcomponents, the *predictor* and the *iterator*. The iterator performs a binary search to determine what the lowest achievable peak memory usage is given the hardware setup and DL model. It requests partitionings and predictions of memory usage for each GPU from the predictor for each of the values in the binary search. The predictor generates a partitionining and predicts the corresponding memory usage for each GPU in the setup using a *fill-first* strategy.

We use the model proposed by mCAP to make per-GPU memory predictions, but add the assumption of *monotonicity* of the memory usage of DL model layers. This means that adding an extra layer to the end of a pipeline stage will result in an increase or no change (but no decrease) of the total memory usage of that pipeline stage. This assumption makes our binary search-based recommendation method possible and in turn enables a scalable method to find memory-friendly partitionings for pipeline-parallel training.

1) Iterator: The iterator performs a binary search to find the smallest possible peak memory usage (across all GPUs) that is achievable for a given hardware setup and DL model. The search starts with a lower bound l of 0 and an upper bound r that corresponds to the memory capacity of a single GPU (in bytes). We define the *current threshold* as $\frac{l+r}{2}$ and request a partitioning from the predictor for this memory threshold (which will use a *fill-first* strategy to generate a partitioning, see Section III-B2). If the predictor generates a partitioning with a predicted memory usage below the current threshold, we lower the upper bound of the search to the current threshold -1. Otherwise, we raise the lower bound of the search to the threshold $+1$. The process is then repeated with the new bounds, until l and r reach the same value. The partitioning generated for that value is the partitioning recommended by CAPSlog. Algorithm 1 summarizes this search strategy.

Algorithm 1 Binary-search recommendation

```
l ← 0, r ← gpu_mem_capacity
while l ≤ r do
    threshold ← ⌊(l+r)/2⌋
    p, predicted_peak ← FILL_FIRST(G, L, threshold)
    if predicted_peak ≤ threshold then
        recommended_partitioning ← p
        r ← threshold − 1
    else
        l ← threshold + 1
return recommended_partitioning
```

2) Predictor: The predictor uses the profiling data to generate a partitioning for a given memory threshold using a fill-first strategy, as illustrated in Algorithm 2. We first place as many layers as possible on the first GPU without exceeding the current memory threshold. When GPU 0 has been "filled", we continue to fill GPUs 1 to $G-2$ in the same manner. We ensure that at least one layer is placed on each GPU. All remaining layers are placed on the last GPU and the partitioning and predicted per-GPU memory usage are returned to the iterator.

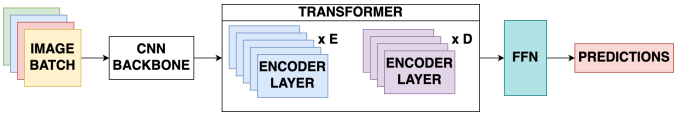

Fig. 6. The DETR architecture. The input is a batch of images, which passes through a CNN backbone, a transformer, and an FNN that outputs the predictions. The transformer has E encoder layers, and D decoder layers.

The memory usage of the last GPU will determine how the binary search continues: the iterator will lower r if the predicted memory usage does not exceed the threshold, but raise l if the memory threshold is exceeded.

The predictor uses the model proposed by mCAP to predict the memory usage of a set of layers on a GPU, namely:

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

Where $M_p(l,m)$ is the predicted memory usage of a GPU when layers l to m are placed on it, $M_i(x)$ is the value of M_i for layer x and $M_a(x)$ is the value of M_a for layer x.

The time complexity of the search performed by the iterator is $O(\log_2(memory\_capacity))$, while generating a partitioning for a given memory threshold has linear time complexity: $O(L+G)$. This means our recommendation method has a total time complexity of $O((L+G)\log_2(memory\_capacity))$. In contrast, mCAP's brute-force recommendation method performs memory predictions for all possible model partitionings and has combinatorial time complexity: $\binom{L-1}{G-1}$. This makes our method considerably more scalable in terms of the number of DL layers and GPUs in the hardware setup. For example, to recommend a partitioning for a 50-layer model on 16 GPUs with 16GB of memory each, mCAP' brute-force method makes 1.57 billion predictions, while CAPSlog makes only 34.

Algorithm 2 Fill-first layer placement and memory prediction

```
function FILL_FIRST(G, L, threshold)
    partitioning ← {},  prediction ← {},  m ← 0
    for gpu_id = 0 to G − 1 do
        l ← m,  m ← m + 1
        while m ≤ L + gpu_id − G + 1 do
            if M_p(l, m) ≤ threshold then
                m ← m + 1
            else
                break
        partitioning.append([l, m − 1])
        prediction.append(M_p(l, m − 1))
    partitioning.append([m, L − 1])
    prediction.append(M_p(m, L − 1))
    return partitioning, max(prediction)
```

Since our recommender performs a binary search to find the lowest achievable peak memory usage across GPUs, CAPSlog will recommend a partitioning with the same overall peak memory usage as mCAP. However, as CAPSlog uses a fill-first strategy to generate the partitioning, the memory usage of the GPUs that do not constitute the overall peak can differ from that of mCAPs recommended partitioning. As the overall peak memory usage dictates the memory headroom and trainable model size, we achieve the same memory gain as mCAP.

Name	CNN Backbone	Dilation	Encoders	CutPoints	Parameters
RN50-E6	ResNet-50	NO	6	24	41M
RN50-D-E6	ResNet-50	YES	6	24	41M
RN101-E6	ResNet-101	NO	6	41	60M
RN101-D-E6	ResNet-101	YES	6	41	60M
RN101-E12	ResNet-101	NO	12	47	68M
RN101-D-E12	ResNet-101	YES	12	47	68M

Fig. 7. DETR configurations.

IV. Experiments

We evaluate CAPSlog's effectiveness in recommending a memory-friendly partitioning by applying it to the DEtection TRansformer, or DETR [3] model. DETR is a heterogeneous object detection model that consists of a convolutional neural network (CNN) [14], a transformer [15] and a feed-forward network (FNN) that outputs the final predictions. The DETR architecture is illustrated in Fig. 6. We scale DETR in various ways that have been shown to improve the accuracy of the model, use CAPSlog to find a memory-friendly partitioning for each model configuration and train the configurations with Varuna. To train DETR with Varuna, we make a number of necessary additions to the framework. We add support for sending multiple, dynamically sized tensors across pipeline stages and for sharing model parameters across more than two stages. We present the peak memory usage when training DETR in Varuna using CAPSlog's recommended partitioning compared with Varuna's own partitioning. We also present an overview of the required time for profiling and recommendation for CAPSlog and mCAP. **We do not report the accuracy of our trained models, as the model's statistical performance is not significantly affected by the partitioning.**

A. Experimental setup

We use the same default model configuration and hyperparameters as in the original DETR paper [3]. The default configuration uses ResNet-50 [16] as the CNN backbone and has 6 encoder and decoder layers. We also scale the model in various ways shown in [3] to improve the accuracy of the model. The first is to replace ResNet-50 by ResNet-101, a larger version with 101 layers. The second is to replace the 2×2 stride in the last stage of the CNN, which halves the feature resolution, with dilation, which keeps the feature resolution the same. This change causes more spatial information and local details to be captured in the CNN output. This doubles the computational cost in the encoder layers, while their memory footprints increase by a factor of ten. The final optimization is to scale up to 12 encoder layers, which increases the total number of layers and trainable parameters in the model. In our experiments, we combine these optimizations in various ways. The precise combinations are shown in Table 7. Apart from these optimizations, all model hyperparameters are kept consistent across the experiments and are assigned DETR's default values.

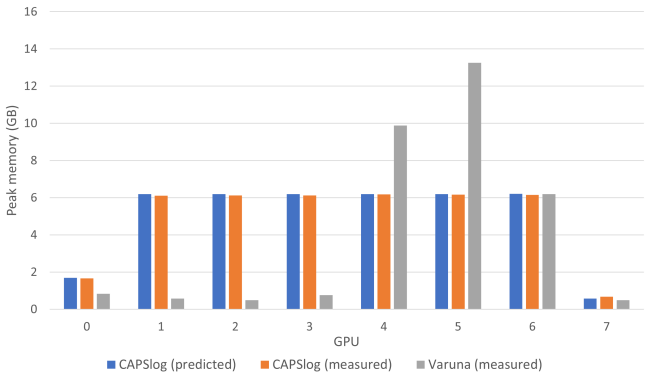

Fig. 8. Peak memory usage for CAPSlog (measured and predicted) and Varuna for RN50-D-E6. Note that CAPSlog gives a much better balanced memory usage over the 8 GPUs and achieves a lower peak memory usage.

We use the same dataset as [3] for our experiments: the COCO-2017 object detection image dataset [17], and use the same data augmentation techniques and random seed. We train all configurations with minibatch size 8 and microbatch size 1.

We place cutpoints in DETR in similar fashion as in the example models provided by Varuna. We place cutpoints between every encoder and decoder layer in DETR and in between the residual blocks in the CNN. We also place cutpoints in between each major component: between the CNN backbone and the first encoder, between the last encoder and the first decoder and between the last decoder and the FFN. We perform our experiments on nodes containing 4 NVidia RTX A4000 GPUs with 16 GB of memory.

B. Memory reduction

We first evaluate the reduction in peak memory usage obtained by CAPSlog compared to Varuna's partitioning. Fig. 9 shows the per-GPU peak memory usage for all model configurations when training with CAPSlog's and Varuna's recommended partitionings on 8 GPUs. CAPSlog achieves a lower peak memory usage for all model configurations, with a reduction in memory between 22.3% (for RN50-D-E6) and 53.2% (for RN50-D-E6). The two largest dilated configurations (RN101-D-E6 and RN101-D-E12) run out of memory when trained with Varuna's partitioning. CAPSlog successfully trains those model configurations, demonstrating how CAPSlog enables training of larger DL models.

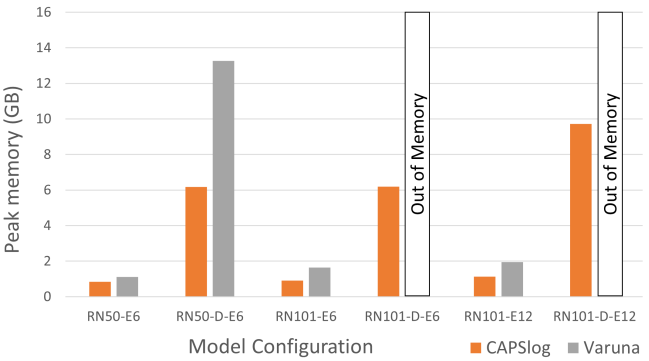

Fig. 9. Peak memory usage for CAPSlog's and Varuna's recommended partitionings for all model configurations.

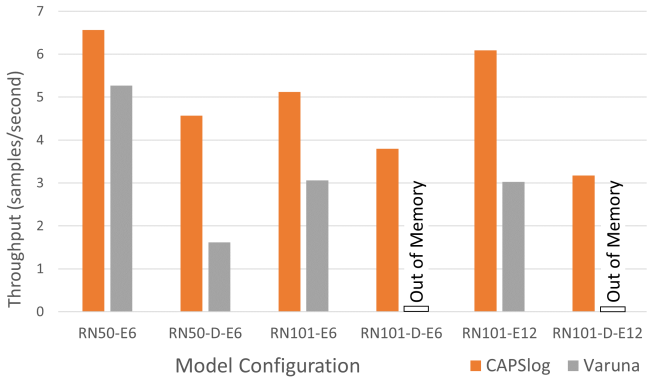

Fig. 10. Achieved throughput for CAPSlog's and Varuna's recommended partitionings for all model configurations.

Fig. 8 shows the per-GPU peak memory usage for RN50-D-E6 with CAPSlog's and Varuna's partitionings. For this configuration, CAPSlog achieves both a lower overall peak memory usage of (6.2 GB compared to 13.3 GB for Varuna) as well as a better balanced one, as GPUs 1 to 6 have approximately the same memory usage. In this dilated model, the CNN's output activations are larger than in the non-dilated model, resulting in an increased memory footprint of the encoder layers. CAPSlog automatically isolates the encoder layers as much as possible and puts only one on each GPU (1-6). Varuna's partitioning fails to spread the memory usage and places multiple encoder layers together on GPUs 4-6, causing high memory usage on those GPUs.

C. Throughput

Figure 10 shows the training throughput obtained for all model configurations when training with CAPSlog's and Varuna's recommended partitionings on 8 GPUs. CAPSlog outperforms Varuna for all model configurations, realizing an increase in throughput between 1.24x and 2.82x.

Varuna's partitioning approach is based on simple metrics about the DL model and cutpoints and fails to effectively balance the strongly varying computational load of the DL layers between the pipeline stages. CAPSlog is profiling-based, and while it partitions for low memory usage and does not guarantee optimal throughput, these results demonstrate that CAPSlog can achieve both lower memory usage and higher throughput than Varuna's partitioner for strongly heterogeneous DL models such as DETR.

D. Profiling and prediction time

Table 12 shows the times taken for profiling the different model configurations with mCAP's profiling approach, and CAPSlog's approach with model trimming. Thanks to CAPSlog's method of generating profiling partitionings, the number of required profiling runs (denoted by $|P|$) is lower than for mCAPs profiling method. mCAP requires at least 3 times the number of profiling runs that CAPSlog requires, for all model configurations. Even though the mean time required for a profiling run does not differ between both methods, the lower number of runs results in a significantly lower total

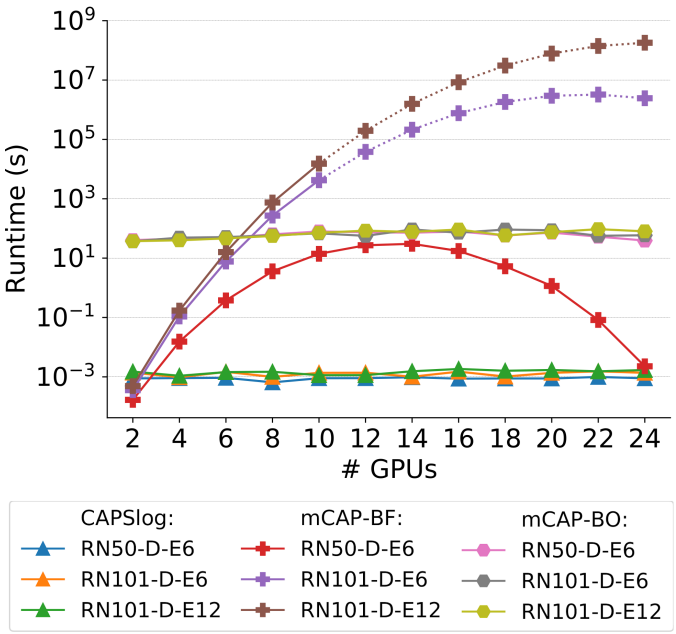

Fig. 11. Recommendation time in seconds (on a logarithmic scale) of CAPSlog and mCAP for all dilated model configurations, for 2-24 GPUs. The runtimes for mCAP-BF for for $\geq$ 10 GPUs are estimated based on the number of required predictions.

profiling time and limits the overhead caused by setting up runs, as reflected in the end-to-end (E2E) profiling times. The reduction in E2E profiling time is proportional to the reduction in the number of runs, at least $\frac{2}{3}$ in our experiments.

Next to reducing the number of profiling runs, model trimming enables profiling of larger models. Note that for all dilated model configurations, at least half of mCAP's profiling runs could not be executed due to running out of memory. CAPSlog successfully profiles all model configurations.

Fig. 11 shows the runtimes of CAPSlog's and mCAP's brute-force (BF) and bayesian optimization (BO) recommendation methods for all dilated model configurations. As mCAP-BF's and CAPSlog's recommendation times depend only on L and G, and that of mCAP-BO on the number of search iterations, the runtimes for the dilated and non-dilated configurations are the same. Hence, we only show the runtimes for the dilated configurations. Fig. 13 shows the predicted peak memory usage of the recommended partitioning for RN101-D-E12 for all recommendation methods. As the recommender's task is to navigate the partitioning space to find the partitioning with the lowest predicted memory usage, these results serve as a measure of the *quality* of the outcome of the search.

CAPSlog's binary search-based recommendation method has a runtime in the order of milliseconds and is faster than both of mCAP's recommenders in all scenarios. It scales well to larger models and hardware setups. It is guaranteed to find the partitioning with the lowest predicted peak memory usage, just as mCAP-BF, as shown in Fig. 13.

mCAP-BF's runtime grows rapidly when scaling from RN50-D-E6 to larger model configurations with more layers or when scaling to more GPUs (note the logarithmic scale on the y-axis in Fig. 11), due to its combinatorial time complexity.

Model	mCAP				CAPSlog			
Config.	$\|P\|$	Mean	Total	E2E	$\|P\|$	Mean	Total	E2E
RN50-E6	25	0:00:27	0:11:15	0:29:56	8	0:00:39	0:05:08	0:12:28
RN50-D-E6	25	*Out of Memory*			8	0:00:50	0:06:39	0:12:30
RN101-E6	42	0:00:43	0:29:58	1:01:10	15	0:00:39	0:09:51	0:21:00
RN101-D-E6	42	*Out of Memory*			15	0:00:45	0:11:18	0:22:28
RN101-E12	48	0:00:38	0:30:06	1:05:46	16	0:00:39	0:10:18	0:22:03
RN101-D-E12	48	*Out of Memory*			16	0:00:51	0:13:35	0:25:37

Fig. 12. CAPSlog's and mCAP's profiling time for all model configurations (denoted as hours:minutes:seconds). $|P|$ denotes the number of profiling runs. Mean denotes the mean runtime of all profiling runs (each performing 10 training iterations). Total is the cumulative total runtime for all $|P|$ profiling partitionings. The end-to-end (E2E) time includes time taken for launching runs, applying the model partitioning and other initialization overhead.

When the number of GPUs approaches the number of model layers, the runtime starts to decrease, since there are fewer possible model partitionings (with at least one layer per GPU). This occurs at about 14 GPUs for RN50-D-E6 and later for the larger models, as the turning point occurs later if a model has more layers. In a typical training setup, L will be significantly larger than G, making the runtimes around the turning point most relevant. When recommending a partitioning for more than 10 GPUs for the R101 models, mCAP-BF's runtime becomes infeasible: it takes over 10 hours for 12 GPUs and over a week for 16 GPUs. CAPSlog's recommendation method is orders of magnitude faster and scales well to large hardware setups. It finds the optimal partitioning for RN101-D-E12 on 24 GPUs in 1.7 milliseconds, while it would take mCAP-BF over 5 years ($1.8 * 10^8$ seconds), over 100 billion times longer.

mCAP-BO's recommendation time is consistent across model and hardware sizes, because it performs a fixed number of search iterations and is not dependent on L and G. However, unlike mCAP-BF and CAPSlog, it does not guarantee to find the partitioning with the lowest predicted memory usage from the full search space. For small L and G, mCAP-BO is outperformed by mCAP-BF both in terms of runtime and outcome of the search. For larger models, mCAP-BO's runtime is significantly lower than that of mCAP-BF. However, as shown in Fig. 13, the search outcome is significantly worse, as the predicted memory usage of the partitioning it finds is up to 93.2% higher than the one found by CAPSlog and mCAP-BF. CAPSlog's recommender is several orders of magnitude faster than mCAP-BO, scales well to higher L and G and provides optimal search results (like mCAP-BF), making CAPSlog the preferred method in all scenarios.

V. Conclusion

We introduced CAPSlog, a scalable memory-centric partitioning approach for pipeline-parallel training. CAPSlog's profiler uses *model trimming* to reduce the profiling time and supports larger and more heterogeneous DL models than existing approaches. CAPSlog's *binary search*-based recommender provides partitioning recommendations in milliseconds and scales well to large DL models and hardware setups.

We demonstrated that CAPSlog reduces the profiling time by 67% on average for various configurations of the DETR object detection model compared to existing approaches and

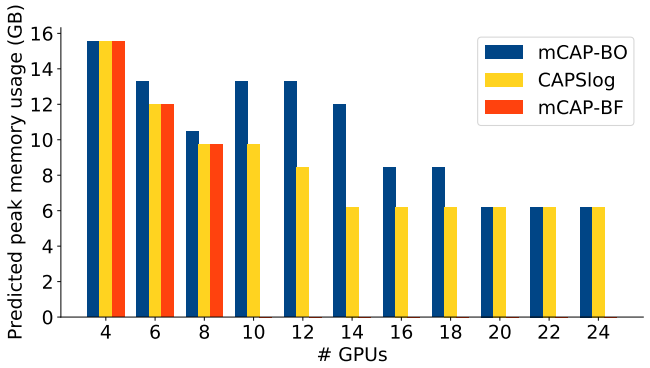

Fig. 13. Predicted peak memory usage for CAPSlog's and mCAP's recommended partitionings for RN101-D-E12 as a measure of the quality of the search outcome. Recommendations with mCAP-BF for $\geq$ 10 GPUs are omitted because of infeasible runtimes.

can find the partitioning with the lowest overall peak memory usage several orders of magnitude faster. It constitutes a reduction in peak memory usage of up to 53.2% for smaller model configurations and unlike existing work, is able to train the largest and most heterogeneous configuration of DETR with a dilated ResNet-101 backbone and 12 encoder layers on our hardware setup without running out of memory.

Through this significant reduction in profiling and recommendation times, CAPSlog enables memory-based partitioning and morphing at a much larger scale, allowing more flexible and dynamic scheduling of large distributed training runs.

We plan to integrate CAPSlog into a partitioning approach that combines multiple optimization objectives, such as throughput, memory and energy efficiency, in future work.

References

[1] H. Dreuning, H. E. Bal, and R. V. van Nieuwpoort, "mCAP: Memory-Centric Partitioning for Large-Scale Pipeline-Parallel DNN Training," in *European Conference on Parallel Processing (Euro-Par)*. Springer, 2022, pp. 155–170.

[2] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra, "Varuna: Scalable, Low-cost Training of Massive Deep Learning Models," in *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022, pp. 472–487.

[3] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-End Object Detection with Transformers," in *European Conference on Computer Vision (ECCV)*. Springer, 2020, pp. 213–229.

[4] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.

[5] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized Pipeline Parallelism for DNN Training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (OSDI)*, 2019, pp. 1–15.

[6] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia *et al.*, "DAPPLE: A Pipelined Data Parallel Approach for Training Large Models," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2021, pp. 431–445.

[7] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-Efficient Pipeline-Parallel DNN Training," in *International Conference on Machine Learning (ICML)*. PMLR, 2021, pp. 7937–7947.

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.

[9] Z. Lai, S. Li, X. Tang, K. Ge, W. Liu, Y. Duan, L. Qiao, and D. Li, "Merak: An Efficient Distributed DNN Training Framework with Automated 3D Parallelism for Giant Foundation Models," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 34, no. 5, pp. 1466–1478, 2023.

[10] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing *et al.*, "Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022, pp. 559–578.

[11] T. Kim, H. Kim, G.-I. Yu, and B.-G. Chun, "BPipe: Memory-Balanced Pipeline Parallelism for Training Large Language Models," in *International Conference on Machine Learning (ICML)*. PMLR, 2023, pp. 16 639–16 653.

[12] S. Li and T. Hoefler, "Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021, pp. 1–14.

[13] W. Zhang, B. Zhou, X. Tang, Z. Wang, and S. Hu, "MixPipe: Efficient Bidirectional Pipeline Parallelism for Training Large-Scale Models," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 25, 2012.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[17] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, 2014, pp. 740–755.