# mCAP: Memory-Centric Partitioning for Large-Scale Pipeline-Parallel DNN Training

Henk Dreuning[1,2], Henri E. Bal[2], and Rob V. van Nieuwpoort[1,3]

[1] University of Amsterdam, The Netherlands
{h.h.h.dreuning, R.V.vanNieuwpoort}@uva.nl
[2] Vrije Universiteit Amsterdam, The Netherlands
bal@cs.vu.nl
[3] Netherlands eScience Center, The Netherlands

**Abstract.** Memory usage is becoming an increasingly pressing bottleneck in the training process of Deep Neural Networks (DNNs), especially when training on Graphics Processing Units (GPUs). Existing solutions for multi-GPU training setups partition the neural network over the GPUs in a way that favors training throughput over memory usage, and thus maximum trainable network size.
We propose mCAP, a partitioning solution for pipeline-parallel DNN training that focuses specifically on memory usage. It evenly distributes Deep Learning models over the available resources with respect to per-device peak memory usage. Our partitioning approach uses a novel *incremental profiling* strategy to extract per-layer memory usage statistics. A *model-based predictor* uses the profiling data to recommend a partitioning that balances peak memory usage. Our approach is DL-framework agnostic and orthogonal to existing memory optimizations found in large-scale DNN training systems. Our results show that our approach enables training of neural networks that are 1.55 times larger than existing partitioning solutions in terms of the number of parameters.

**Keywords:** Deep Learning · Pipeline Parallelism · HPC.

## 1 Introduction

Deep Learning (DL) has facilitated breakthroughs in many application domains, including video analysis, natural language processing and speech recognition. The popularity of neural networks in these domains can be attributed partly to the development of new methods and algorithms, and partly to an increase in available compute power. Increasing the "depth" of neural networks, i.e. the number of hidden layers, often improves the performance of the models, as a deeper network can learn more complex input-output relations. Increased compute power has enabled training of deeper networks and has shortened the development times of neural network architectures.

However, as DL models, training datasets and individual training samples continue to grow in size, memory usage becomes an increasingly pressing bottleneck in Deep Neural Network (DNN) training. This bottleneck is especially

apparent when training on Graphics Processing Units (GPUs), due to their limited memory capacity. To limit memory usage, developers are forced to resort to measures that severely reduce the effectiveness of their solutions, such as downsampling input data, reducing training batch sizes or shrinking DL model sizes. In some cases models cannot even be trained with such measures in place. Examples can be found in research areas such as high resolution image- and video-processing [4, 5, 14] and natural-language processing [12, 15].

In this work, we present mCAP (memory-Centric Approach for Partitioning), a partitioning approach for multi-GPU pipeline-parallel DNN training. Existing pipelined training solutions, such as GPipe [6, 8], PipeDream [10, 12] and DAPPLE [4] prioritize training throughput when partitioning the model. This creates an imbalance in peak memory usage between devices, leading to a smaller trainable model size. Our partitioning solution uses novel methods, incremental profiling and model-based prediction, to evenly distribute DL models over the available resources with respect to per-device peak memory usage, thus focusing on the maximum trainable model size instead of other objectives. Our partitioning scheme targets intra-batch pipeline parallel training solutions and can be adjusted to work with inter-batch pipelining systems as well. mCAP is orthogonal to memory optimizations found in pipeline-parallel systems, such as efficient scheduling of forward and backward passes [4, 12] and more generic optimizations, such as compression, recomputation and swapping of intermediate data to host memory [2, 9, 18].

Most existing partitioning and placement approaches aim to optimize achieved throughput and do not consider per-GPU memory usage. PipeDream and DAPPLE's planners only focus on equal per-GPU processing time and high throughput. GPipe leaves the task of partitioning the model to the programmer. However, TorchGPipe [8] (an implementation of GPipe in PyTorch) contains an automatic partitioner that optimizes throughput based on measurements of the execution time of the forward pass for each layer of the DL model.

Accurate predictions of per-GPU peak memory consumption are needed for automatic, memory-balanced partitioning. Predicting peak memory consumption is complex, because it is influenced by memory optimizations implemented at different levels of the DNN training software stack. Analytical modeling based on static analysis of the DL model does not capture the effects of these optimizations, making it infeasible to reach accurate predictions of peak memory usage using static techniques. mCAP uses novel profiling and prediction methods to recommend a partitioning with a balanced per-GPU peak memory usage, while automatically capturing the effects of a wide range of memory optimizations at the levels of the DL framework and pipeline-parallel training system.

mCAP **does not affect the convergence speed and accuracy achieved by the DL model** compared to other partitioning approaches, because the performed learning operations are identical, regardless of the selected partitioning.

Concretely, the contributions of this paper are as follows:

- We introduce a novel approach to DNN partitioning for multi-GPU training, focusing purely on reducing peak memory usage to enable the training of

larger DL models. By focusing on maximum model size instead of training throughput, we make a different trade-off than existing approaches. We study the effects of this trade-off on training throughput.
– We provide an overview of memory-optimization techniques present in modern DL frameworks and explain the shortcomings of existing attempts to reach balanced peak memory usage across GPUs during DNN training.
– We use a novel profiling method, incremental profiling, for our partitioning.
– We present a prediction based partitioning algorithm that uses the profiling data to obtain balanced peak memory usage between GPUs.
– We demonstrate that our approach enables the training of DL networks that are up to 1.55 times larger than existing partitioning approaches.

## 2 Background and Related Work

**Neural network training:** the DNN training process consists of iterations in which a forward pass and backward pass are performed for a single batch of input data (a minibatch). When the forward and backward passes have been performed, the weights of the model are updated, which concludes the iteration.

As DNNs continue to grow in size and computational demand, DNN training is now moving towards high-performance computing infrastructures where multiple GPUs can be used simultaneously to train a DNN. State-of-the-art software solutions for multi-GPU training perform pipeline-parallel training.

Figure 1 shows the three layers of the software stack: the DL framework that implements the training operations, the pipelining system that implements pipeline parallelism, and the application layer.

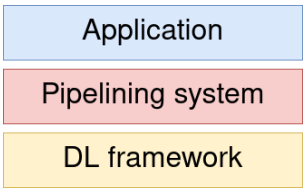



Fig. 1: Pipeline-parallel DNN training software stack

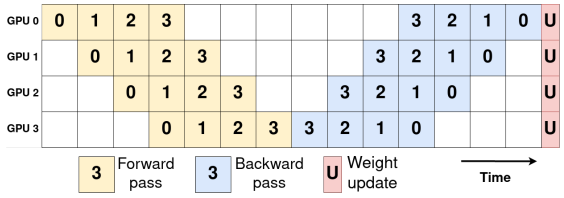



Fig. 2: Pipelined DNN training as performed by GPipe. Numbers indicate microbatch ids.

**Existing memory optimizations:** modern DL frameworks like PyTorch [13] and TensorFlow [1] include optimizations that reduce the memory usage on GPUs. With *early deallocation* of memory, the memory used for forward activations and gradients is released immediately after the backward pass and weight update have been performed for a given layer, instead of for the complete neural network. Another optimization at the level of the DL framework is *activation memory reuse*, in which the memory where activations are stored during the forward pass is re-used to store gradients in the backward pass.

An optimization that is not implemented at the level of the DL framework, but at the level of the pipelining system, is *activation recomputation*. Here forward activations are not kept in memory in between the forward and backward

pass of a given layer, but discarded and recomputed again when they are needed during the backward pass.

Several other works have proposed methods to reduce GPU memory usage during training. In [18,21,25], the authors propose methods for memory pooling and swapping (temporarily) unused data (like activations) in GPU memory to main memory. In [2], the authors use Unified Memory capabilities to leverage host memory for out-of-core DNN training. Our approach is orthogonal to such approaches and can be used on top of training systems that implement these optimizations at the level of the DL-framework or the pipelining system.

**Pipeline Parallelism:** in pipeline-parallel training, the neural network is partitioned over the workers. For each minibatch, each worker performs the forward and backward pass for their part of the network, and activations and gradients are communicated between workers. As a result, the model size is no longer limited by the memory size of a single worker. To increase throughput minibatches are processed in a pipelined fashion. Multiple minibatches (or slices thereof) are consecutively fed into a pipeline, and workers perform forward- and backward-passes on these minibatches.

There are two types of pipeline parallelism. In intra-batch pipelining (such as implemented by GPipe [6, 8] and DAPPLE [4]) a single minibatch is split into multiple micro-batches, and the forward passes of these micro-batches are fed into a processing pipeline. When all forward passes have completed, the corresponding backward passes are performed. Finally, all compute nodes update their parameters (see Figure 2). Intra-batch pipelining does not introduce staleness of weights and has a memory usage that is inversely proportional to the number of workers. A disadvantage is the existence of a synchronization point, which causes a "bubble" in the pipeline and idle time for the workers.

In inter-batch pipelining (such as implemented by PipeDream [10, 12]) complete minibatches are fed into the pipeline without splitting them into smaller entities (see Figure 4). Multiple copies of the model's parameters are kept in memory to make sure forward and backward passes on a particular minibatch are performed with the same parameters. Inter-batch pipelining does not suffer from idle time because there is no system-wide synchronization point. Despite several improvements [12, 24], staleness and increased memory usage caused by the need for multiple parameter versions remain disadvantages.

**Partitioning algorithms for pipeline parallelism:** PipeDream [10] proposes a *planner* specific to inter-batch pipelining that outputs a balanced pipeline in terms of per-stage (GPU) computation time. This is achieved by profiling computation time per layer and estimating communication times with an analytical model. GPipe [6] leaves the task of partitioning the DL model to the programmer. TorchGPipe [8], an implementation of GPipe in PyTorch, provides automatic partitioning based on profiling the computation time needed for the forward pass of each layer. DAPPLE's [4] partitioner tries to achieve high throughput by minimizing the pipeline latency, which is determined by the latency of processing a single minibatch. RaNNC [23] is an intra-batch pipelining

framework that performs automatic partitioning using atomic-level, block-level and stage-level partitioning.

All of these partitioning approaches focus on finding the partitioning that achieves the highest throughput. Contrary to existing work, our work proposes a partitioning approach that aims for a balanced pipeline in terms of memory usage, enabling training of larger models. To that end, it models the effects of memory optimizations implemented in the pipelining system, such as activation recomputation. Our approach currently targets intra-batch pipeline parallelism.

**Other large-scale DNN training paradigms:** MeshTensorflow [19] and Megatron-LM [20] are systems that partition individual tensor operations over multiple accelerators as opposed to layers. Megatron-LM does not provide automatic partitioning and only supports transformer models. In [11], the authors combine tensor partitioning with pipeline parallelism, but do not improve the memory footprint over existing approaches. ZeRO [16] partitions model states over workers to save memory, but focuses on data and model parallelism.

## 3 Method

We propose mCAP, a partitioning approach for pipeline-parallel DNN training that determines how the DL model is partitioned over the workers (GPUs). Our partitioning approach focuses on achieving balanced peak memory usage across all GPUs during DNN training, to enable the training of larger DL models. By using a novel profiling strategy called *incremental profiling* mCAP automatically captures and models the effects of a wide range of memory optimizations that are present in DL frameworks, such as the ones described in Section 2 (early deallocation and activation memory reuse). This makes our approach agnostic to which framework is used at the DL framework layer. Moreover, by combining the incremental profiling strategy with our *model-based prediction algorithm*, mCAP models the memory usage of intra-batch pipelining systems and the optimizations they implement (like activation recomputation).

This section discusses mCAP's design and shows how the combination of *incremental profiling* and our *model-based prediction algorithm* is capable of capturing the memory behavior of intra-batch pipeline-parallel training systems.

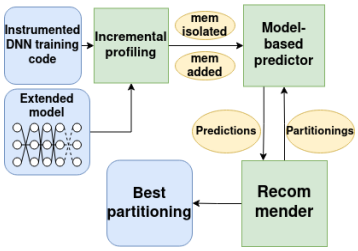



Fig. 3: Overview of mCAP.

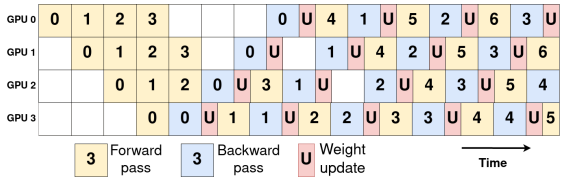



Fig. 4: Pipelined DNN training as performed by PipeDream. Numbers indicate minibatch ids.

Figure 3 shows an overview of our partitioning approach. The approach consists of three parts: incremental profiling, prediction, and recommendation. In the profiling phase, we collect data about the peak memory usage of each GPU

during training for several, specifically selected partitionings. From this profiling data, we extract two statistics for each layer in the neural network. In the recommendation phase, these statistics form the input to the predictor, which accurately predicts the memory usage for a set of partitionings generated by the recommender. The recommender applies a search strategy to find the partitioning(s) with the lowest peak memory usage across the GPUs based on the predictions. We explain the workflow step-by-step.

### 3.1 Profiling

The profiling stage collects per-layer statistics that can be used in the prediction stage. We instrumented pipeline parallel DNN training code to monitor the peak memory usage for each GPU during the training process. Short profiling runs are then performed in the same setup (DL framework, pipelining system, hyperparameters and hardware) as in the final training run for which we are finding a memory-balanced partitioning. In these profiling runs all training stages are executed: forward passes, backward passes and update steps. Therefore, they automatically capture the effects of memory optimizations at the DL framework level for *all training stages*.

The profiling runs are performed with a specifically selected set of partitionings. The selected set of partitionings is such that for each layer $l$, we run (with $l > n >= 0$):

(a) a partitioning where layer $l$ is the only one on a GPU, and;
(b) a partitioning where layers $n$ to $l-1$ are placed on a GPU, and;
(c) a partitioning where layers $n$ to $l$ are placed on a GPU.

Figure 5 shows examples of partitionings described by requirements (a), (b) and (c). During selection of these partitionings, we keep $n$ as small as possible.

The above selection requirements ensure that we can extract two metrics for each layer $l$ in the DL model from the profiled data: (1) the peak memory usage when layer $l$ is the only one on a GPU and (2) the effect on peak memory of adding layer $l$ to an existing set of layers on a GPU. We extract the former directly from the partitionings described by requirement (a) and call this metric $mem\_isolated(l)$, while we extract the latter from the difference in peak memory usage between the partitionings described by requirements (b) and (c), and call it $mem\_added(l)$. These two per-layer metrics capture the data needed to accurately predict the memory usage for all possible partitionings.

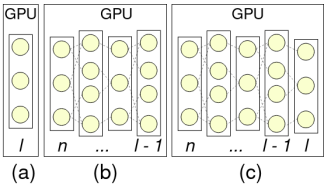



Fig. 5: Examples of partitionings described by requirements (a), (b) and (c).

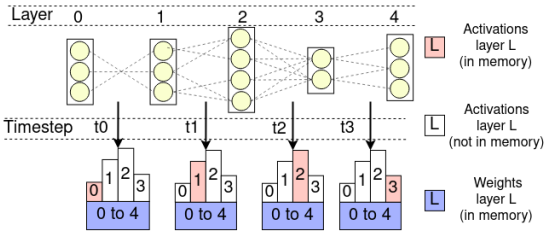



Fig. 6: GPU memory usage for weights and activations during the forward pass.

It is important to make the distinction between $mem\_isolated(l)$ and $mem\_added(l)$ because a layer's contribution to peak memory usage depends on its position on the GPU it is assigned to. Inspection of these metrics for AmoebaNet-D(36, 544) showed that the difference between the two metrics is significant, ranging to hundreds of Megabytes per layer. The difference is caused by optimizations such as activation recomputation and the presence of communication buffers to send the activations of the last layer on a GPU to the next GPU in the pipeline. The next subsections show how our predictor models the effects of such optimizations on memory usage, using both metrics as input.

### 3.2 Peak memory usage for intra-batch pipelining with activation recomputation

This subsection describes how the recomputation optimization implemented in most intra-batch pipelining systems (including TorchGPipe) affects the peak memory consumption. Optimizations such as recomputation do not simply lower the peak memory consumption for each layer individually, but influence peak memory in a more complex manner. We created our prediction algorithm to model such influences of recomputation and other (potentially future) optimizations in intra-batch pipelining systems on peak memory usage.

The peak memory consumption on a GPU in a pipelined system is constituted by two main factors: the memory needed for the weights of the layers hosted by the GPU, and the memory needed for the activations generated by those layers. In intra-batch pipelining, the memory needed for a layer's weights is constant. Recall from Section 2 that with activation recomputation, a layer's activations are discarded as soon as the activations for the next layer have been computed. The activations for the former layer are then recomputed when they are needed again in the backward pass. As a result, the amount of memory required to store activations fluctuates during the forward and backward passes of a single microbatch. Hence, the *peak* memory consumption of the GPU is determined by the one layer on the GPU that requires the most memory for its activations.

Figure 6 illustrates this principle of fluctuating memory usage in the forward pass in a simplified situation, where an example neural network of 5 layers is trained on a single GPU. At time $t_0$, memory is used to store the weights of all layers and the activations generated by layer $l_0$. At time $t_1$, the activations of layer $l_0$ are discarded, and the activations of layer $l_1$ are stored (which consume more memory). In this example, the layer that generates the most activations is layer $l_2$, which means the peak memory consumption for this neural network during the forward pass is dictated by that layer. This principle extends to the backward pass in similar fashion.

### 3.3 Prediction

To find the partitioning with the lowest peak memory usage across all GPUs, we predict the peak memory usage for a set of partitionings. This set is determined by the recommender (see section 3.4) and forms the input to the predictor, together with the per-layer data described in Section 3.1. The predictor estimates the peak memory usage for each partitioning in the set as shown in listing 1.1.

Our partitioning algorithm automatically models the effects that different layers have on the peak memory consumption on a GPU, as described in Section 3.2, by means of its design. When predicting the memory usage for a given partitioning, it first considers the peak memory usage that is obtained when only the first layer that is assigned to the GPU, is placed (which corresponds to the *mem_isolated* statistic). Figure 7.a illustrates this situation. It then models the changes in peak memory consumption that are caused by adding the remaining layers to the GPU, by adding the *mem_added* statistic layer-by-layer. Two scenarios exist for each added layer: it generates less activations than the preceding one (Figure 7.b) and the peak memory of the GPU is only affected by the added layer's weights, or it outputs more activations than the preceding layer (Figure 7.c), and the peak memory usage increases due to the added layer's weights *and* activations.

```
for gpu in GPUs:
  layers = RetrieveLayers(p, gpu)
  GPU_peak_mem =
    mem_isolated(layers[0])
  for layer in remaining layers:
    GPU_peak_mem +=
      mem_added(layer)
  StorePeak(p, gpu, GPU_peak_mem)

per_GPU_peaks = RetrievePeaks(p)
overall_peak = max(per_GPU_peaks)
```

Listing 1.1: Peak memory prediction for a given partitioning.

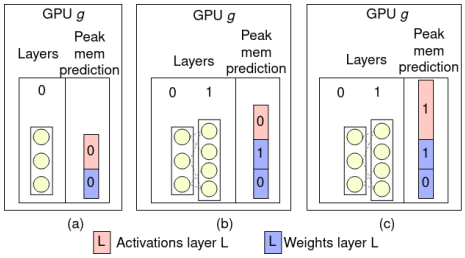



Fig. 7: *Model-based prediction* scenarios.

### 3.4 Recommendation

mCAP supports two mechanisms to search for the partitioning with the lowest peak memory usage across all GPUs: *Brute-Force (mCAP-BF)* and *Bayesian Optimization (mCAP-BO)*.

*mCAP-BF* predicts the peak memory usage for all possible $P_{all} = \binom{L-1}{G-1}$ partitionings (where $L$ is the number of layers to partition over $G$ GPUs), and selects the partitioning with the lowest peak memory usage from the prediction outcomes. We apply a tie-breaking rule if there are multiple partitionings with the same lowest predicted peak memory usage. For each remaining candidate, we exclude the GPU with the highest peak memory usage $GPU_{peak}$ in that partitioning and select the candidate that has the best balanced (lowest) peak memory usage across the remaining GPUs. This is a realistic alternative selection criterion if the prediction for $GPU_{peak}$ was inaccurate.

*mCAP-BO* applies Bayesian Optimization to search for the partitioning with the lowest predicted peak memory usage. Each GPU forms a dimension of the search space and the parameter range is determined by the number of layers that can be placed on the GPU.

The choice between *mCAP-BF* and *mCAP-BO* can be made based on the value of $P_{all}$. *mCAP-BF* is guaranteed to find the partitioning with the lowest

predicted peak memory usage in the search space and is cost efficient enough for limited values of $P_{all}$. When $P_{all}$ is large (because the DL model has many layers and/or many GPUs are used), it pays off to use *mCAP-BO*. While *mCAP-BO* is not guaranteed to find the partitioning with the lowest peak memory usage from the full search space, it is less expensive in terms of execution time for larger values of $P_{all}$ (see Section 4.3).

### 3.5 Implementation

We have implemented our partitioning approach for TorchGPipe. We have instrumented the training loop's code to capture the peak memory usage for each GPU during the training process using built-in facilities of PyTorch. Our code instrumentation has no impact on GPU memory usage, as the recorded data is stored in main memory. Moreover, the impact on training throughput is negligible, because recording the required data is a lightweight operation.

We chose to implement memory profiling at the level of the DL framework (PyTorch) because it allows us to differentiate between memory that is actually in use and memory that is reserved (cached) by PyTorch. Lower level tools (CUDA or other NVIDIA tools) would not allow us to make this distinction.

Currently the selection of the partitionings used for incremental profiling is partially automated and partially a manual process (if some configurations run out of memory, a different set is chosen). Additionally, we extend the network with dummy layers to enable running the partitionings described in Section 3.1 for each layer in the neural network. We plan the implementation of a fully automated version of this process for future work.

We use the DDLBench benchmarking framework [7] to run our experiments. We extended the code of the benchmarking suite and TorchGPipe to parameterize some variables of the training process, such as the partitioning to use. We use *scikit-optimize* to implement the Bayesian Optimization process for the *mCAP-BO* recommendation mode.

## 4 Experiments

### 4.1 Experimental Setup

We apply our partitioning approach to two DL models. First, we do experiments with a relatively small neural network, VGG11 [22], to evaluate to what extent mCAP is able to find an optimally memory-balanced partitioning. We then do experiments with a larger, scalable DL model, AmoebaNet-D [17], to see how far we can increase the size of the neural network without running out of memory. We perform this experiment for the partitioning recommended by mCAP and compare the results to the partitioning chosen by TorchGPipe's throughput-oriented partitioner.

We perform multiple experiments using a single DL model with an adjustable size, rather than with multiple different fixed-sized models, because we want to obtain a precise comparison between the maximum trainable model size with mCAP and TorchGPipe's partitioner. We compare to a throughput-oriented partitioner because, to the best of our knowledge, no other memory-oriented

partitioners exist for pipeline-parallel training. We do not compare to solutions that are orthogonal to pipeline-parallel training with our partitioning approach. Examples of orthogonal solutions are leveraging host memory to virtually increase the GPU memory size, through activation and weight swapping or Unified Memory techniques [2, 18, 21, 25].

We note that the statistical performance (how fast the neural network learns and achieved accuracy) is not affected by the choice of partitioning. The learning operations performed by the pipelining system are mathematically identical, regardless of the partitioning. We do therefore not explicitly evaluate the statistical performance of the partitioning selected by mCAP. We use randomly generated images of 224x224 pixels as training data in our experiments, consistent with images from the ImageNet dataset [3].

The training runs performed in the profiling stage only have to last a very limited number of epochs and can be performed with a small dataset size. This is because the peak memory usage is steady after the first epoch and the peak memory usage is not dependent on the size of the dataset. In our experiments, the time required for profiling is in the order of minutes per profiling run. The number of required profiling runs is $L-G+1$. The time needed by the prediction algorithm ranges from seconds to minutes, depending on the number of possible partitionings and the recommendation mode. We consider this overhead negligible, given that the final run for which we are searching a balanced partitioning typically lasts several days to weeks. Moreover, the aim of our approach is to enable training of larger models, not to increase throughput.

The Bayesian Optimization process of *mCAP-BO* performs 75 iterations, uses *scikit-optimize*'s *gp_hedge* acquisition function with the *sampling* acquisition optimizer, *xi* and *kappa* set to 1000 to favor exploration over exploitation and the default Matérn kernel.

Our experiments are performed on nodes containing 4 NVIDIA Titan RTX GPUs with 24 GB GDDR6 memory. The GPUs are connected to the host through PCIe 3.0 x16. We use PyTorch version 1.5.0 and TorchGPipe as the pipelining system.

### 4.2 VGG11

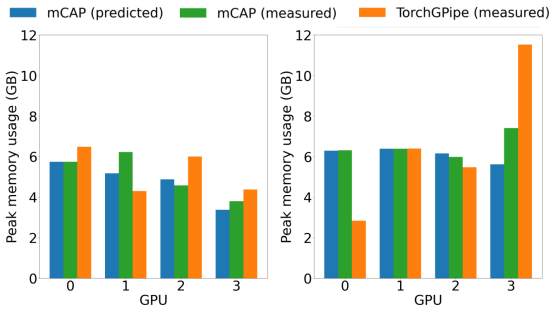
We perform experiments with VGG11 to evaluate how accurate our prediction algorithm is. We first use VGG11 because it is a relatively small network (132.9 million parameters) with a limited number of layers (30), so the amount of possible partitionings of the model over 4 GPUs is also limited (3654). It is therefore feasible to perform training runs for all possible partitionings, to get an overview of the memory usage and computational performance for *all datapoints in the partitioning space*. This experiment is only focused on validating that our approach selects a partitioning with a low peak memory usage from the full partitioning space.

Figure 10 shows the achieved peak memory usage and throughput of training VGG11 for all possible partitionings on a 4-GPU node, with TorchGPipe. Each run performs 2 training epochs with a training dataset size of 5000 samples, an overall batch size of 1104, consisting of 12 microbatches of 92 samples each, the

Stochastic Gradient Descent optimizer with a momentum of 0.9, a weight decay of $1 \times 10^{-4}$ and a learning rate of 0.1. The partitionings selected by mCAP and TorchGPipe's automatic partitioner are highlighted. Given the limited number of possible partitionings, we use the *mCAP-BF* recommendation mode.

Our approach selects the partitioning (3-3-5-19) with the 3rd-lowest peak memory usage of all partitionings. Our predictor slightly underestimates the peak memory usage of the selected partitioning and deems it equivalent to the partitionings with the lowest peak (the horizontal green line). The tie-breaking rule then works in favor of the selected partitioning. Its peak memory usage is 1.05x higher than the lowest peak, while that is 1.10x for the partitioning selected by TorchGPipe's partitioner. That partitioning (6-2-5-17) achieves 0.84x the throughput of the best performing partitioning, which is the one selected by our approach. These observations confirm that mCAP selects a partitioning from the full partitioning space that has relatively low peak memory usage.

Figure 8.a shows the per-GPU peak memory usage of mCAP's and TorchGPipe's partitioning (predicted and measured). The partitioning selected by mCAP, while being the one with the 3rd-lowest memory usage, is still relatively unbalanced (the standard deviation is 0.95). We attribute this to the small partitioning space of VGG11. Because the network is split at the level of layers and the number of layers in VGG11 is limited, a partitioning with an (almost) perfectly balanced memory usage simply does not exist (the standard deviation of the best performing partitioning amongst the ones with the absolute lowest peak memory usage is still 0.73). We study the memory gain in a more realistic scenario, with a larger network, in Section 4.3.




(a) VGG11 (b) AmoebaNet-D(36, 544)

Fig. 8: Per-GPU peak memory usage.

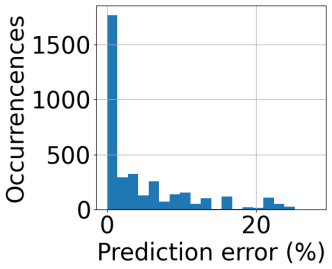



Fig. 9: Prediction error histogram.

Figure 9 shows a histogram of the error between peak memory usage as predicted by our prediction algorithm and the actual peak memory consumption, for all 3654 possible partitionings. Our predictor is able to predict the peak with an error margin smaller than 14% error in 90% of the cases.

### 4.3 AmoebaNet-D

We now experiment with a larger network (AmoebaNet-D) to see how much reduction in peak memory usage and potential network growth our partitioner realistically achieves.

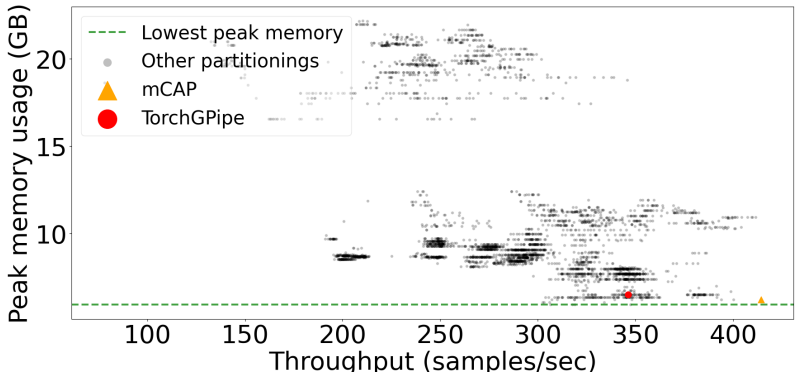



Fig. 10: Memory usage and throughput of all possible partitionings. The partitionings selected by mCAP (yellow) and TorchGPipe (red) are highlighted.

AmoebaNet-D($L$, $F$) has 2 parameters that determine the size of the neural network: $L$ and $F$ for layers and filters respectively. We first apply our partitioning approach to AmoebaNet-D(36, 544). With these parameters, the neural network has 1.06 billion trainable parameters.

We perform the training runs in this experiment with an overall batch size of 32, consisting of 4 microbatches of 8 samples each. The input data and remaining training- and hyperparameters are identical to the ones used before. We use the *mCAP-BO* recommendation mode (and *mCAP-BF* for reference).

Figure 8.b shows the per-GPU peak memory usage of AmoebaNet-D(36, 544) for mCAP's partitioning (predicted and measured) and TorchGPipe's partitioning. mCAP's partitioning is considerably more balanced in peak memory usage. The measured peak has a standard deviation of 0.53 across all GPUs, while that is 3.15 for TorchGPipe's partitioning. mCAP's partitioning reaches a 35% reduction in peak memory usage compared to TorchGPipe's partitioning.

Our experiments showed that *mCAP-BO* recommends the same partitioning for AmoebaNet-D(36, 544) as *mCAP-BF*, validating the effectiveness of *mCAP-BO* in navigating the search space. It also reduces the prediction time by 2.6x compared to *mCAP-BF* (from 99.5 to 38.8 seconds). We expect this gain to increase when more GPUs or DNNs with even more layers are used.

Next, we determine how far we can scale the network up with mCAP's partitioning. We use the $F$ (filters) parameter to increase the number of trainable parameters in AmoebaNet-D. To find the maximum value for $F$ that we can train with mCAP's partitioning, we perform a binary search.

Figure 11 shows the peak memory usage (and throughput) for each successful training run of the binary search, plotted against the network size of AmoebaNet-D. The size of the DL model is expressed in the number of trainable parameters, which is determined by $L$ and $F$. The peak memory usage is consistently higher for TorchGPipe's partitioning, and grows faster with network size than for mCAP's partitioning. The maximum trainable network size

for mCAP's partitioning is 1.55 times larger than for TorchGPipe's partitioning (3.61B vs 2.32B trainable parameters).

Figure 11 also shows the achieved throughput for each value of $F$ used in the scaling experiment. Although it is not our focus, mCAP's partitioning achieves 10.5% higher throughput on average than TorchGPipe's partitioning.

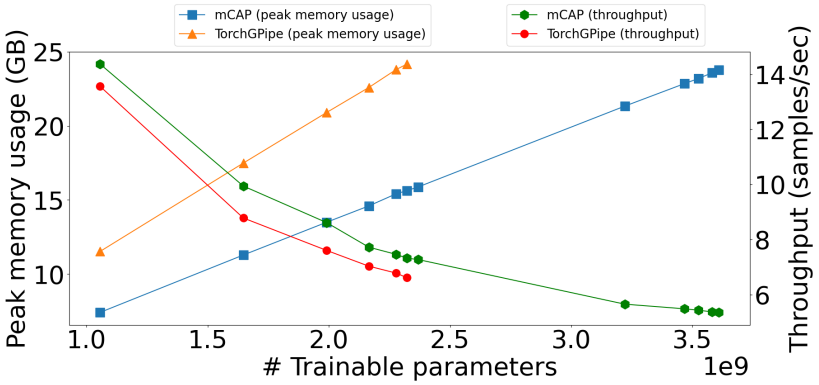



Fig. 11: Peak memory usage and achieved throughput for AmoebaNet-D.

## 5 Conclusion and Future Work

We proposed mCAP, a partitioning approach for multi-GPU pipeline-parallel DNN training that focuses purely on achieving balanced peak memory usage across GPUs. mCAP uses a combination of incremental profiling and model-based prediction. Through profiling our approach automatically captures the effects of memory optimizations implemented at the DL framework level and can thus (after re-implementation of the memory profiling) be applied in combination with any modern DL framework. mCAP's model-based predictor targets intra-batch pipelining systems and can be easily adjusted to support inter-batch pipelining systems as well. Applying mCAP does not affect the statistical performance compared to other partitioning approaches, because the performed learning operations are mathematically identical.

We demonstrated that mCAP recommends a partitioning with a low peak memory usage from the full partitioning space. mCAP provides the *brute-force* recommendation mode for limited search spaces and the *Bayesian Optimiztion* mode to efficiently find a memory-balanced partitioning in a large search space. mCAP can train neural networks that are 1.55 times larger than existing partitioning solutions. We plan to automate partitioning selection and network manipulation for the incremental profiling phase for future work. We also plan to port Bayesian Optimization to the GPU to further reduce *mCAP-BO*'s runtime.

## References

1. Abadi, M., et al.: TensorFlow: A System for Large-Scale Machine Learning. In: OSDI. pp. 265–283 (2016)

2. Awan, A.A., et al.: OC-DNN: Exploiting Advanced Unified Memory Capabilities in CUDA 9 and Volta GPUs for Out-of-Core DNN Training. In: HiPC. pp. 143–152. IEEE (2018)
3. Deng, J., et al.: Imagenet: A large-scale hierarchical image database. In: CVPR. pp. 248–255. IEEE (2009)
4. Fan, S., et al.: DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In: PPoPP. pp. 431–445 (2021)
5. Hara, K., et al.: Learning Spatio-Temporal Features with 3D Residual Networks for Action Recognition. In: ICCV Workshops. pp. 3154–3160 (2017)
6. Huang, Y., et al.: Gpipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In: NeurIPS. pp. 103–112 (2019)
7. Jansen, M., et al.: DDLBench: Towards a Scalable Benchmarking Infrastructure for Distributed Deep Learning. In: 2020 IEEE/ACM Fourth Workshop on Deep Learning on Supercomputers (DLS). pp. 31–39. IEEE (2020)
8. Kim, C., et al.: Torchgpipe: On-the-fly Pipeline Parallelism for Training Giant Models. arXiv preprint arXiv:2004.09910 (2020)
9. Mittal, S., Vaishay, S.: A Survey of Techniques for Optimizing Deep Learning on GPUs. Journal of Systems Architecture **99**, 101635 (2019)
10. Narayanan, D., et al.: PipeDream: Generalized Pipeline Parallelism for DNN Training. In: SOSP. pp. 1–15 (2019)
11. Narayanan, D., et al.: Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In: SC21. pp. 1–15 (2021)
12. Narayanan, D., et al.: Memory-Efficient Pipeline-Parallel DNN Training. In: ICML. pp. 7937–7947. PMLR (2021)
13. Paszke, A., et al.: PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: NeurIPS. pp. 8026–8037 (2019)
14. Pinckaers, H., Litjens, G.: Training Convolutional Neural Networks with Megapixel Images. arXiv preprint arXiv:1804.05712 (2018)
15. Raffel, C., et al.: Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv preprint arXiv:1910.10683 (2019)
16. Rajbhandari, S., et al.: ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In: SC20. pp. 1–16. IEEE (2020)
17. Real, E., et al.: Regularized evolution for image classifier architecture search. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 4780–4789 (2019)
18. Rhu, M., et al.: vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In: MICRO. pp. 1–13. IEEE (2016)
19. Shazeer, N., et al.: Mesh-Tensorflow: Deep Learning for Supercomputers. In: NeurIPS. pp. 10414–10423 (2018)
20. Shoeybi, M., et al.: Megatron-LM: Training Multi-Billion Parameter Language Models Using GPU Model Parallelism. arXiv preprint arXiv:1909.08053 (2019)
21. Shriram, S., et al.: Dynamic memory management for gpu-based training of deep neural networks. In: IPDPS. pp. 200–209. IEEE (2019)
22. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
23. Tanaka, M., et al.: Automatic Graph Partitioning for Very Large-scale Deep Learning. In: IPDPS. pp. 1004–1013. IEEE (2021)
24. Yang, B., et al.: PipeMare: Asynchronous Pipeline Parallel DNN Training. MLSys **3**, 269–296 (2021)
25. Zhang, J., et al.: Efficient Memory Management for GPU-based Deep Learning Systems. arXiv preprint arXiv:1903.06631 (2019)