2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-3766-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPS54959.2023.00025

Exploiting Input Tensor Dynamics in Activation Checkpointing for Efficient Training on GPU

Jianjin Liao[1†], Mingzhen Li[1†], Hailong Yang[1✉], Qingxiao Sun[1], Biao Sun[1], Jiwei Hao[1], Tianyu Feng[1]
Fengwei Yu[2], Shengdong Chen[2], Ye Tao[2], Zicheng Zhang[2], Zhongzhi Luan[1], Depei Qian[1]
[1] *Beihang University*, Beijing, China
[2] *SenseTime Research*, Beijing, China
{liaojianjin,lmzhhh,hailong.yang,qingxiaosun,biaosun,jiweihao,ty_feng,07680,depeiq}@buaa.edu.cn
{yufengwei,chenshengdong,taoye1,zhangzicheng}@sensetime.com

***Abstract*—Larger deep learning models usually lead to higher model quality, however with an ever-increasing GPU memory footprint. Although several tensor checkpointing techniques have been proposed to enable training under a restricted GPU memory budget, they fail to exploit the input tensor dynamics due to diverse datasets and subsequent data augmentation, and thus leave the training optimization on table. In this paper, we propose *Mimose*, an input-aware tensor checkpointing planner respecting the memory budget while enabling efficient model training on GPU. *Mimose* builds a lightweight but accurate prediction model of GPU memory usage online, without pre-analyzing the model. It generates a tensor checkpointing plan based on per-layer memory prediction and applies it to the training process on the fly. Our experiments show that *Mimose* achieves superior training throughput compared to state-of-the-art checkpointing frameworks under the same GPU memory budgets.**

***Index Terms*—model training, GPU memory, tensor checkpointing, input dynamics**

I. Introduction

Deep learning (DL) models are important and indispensable building blocks in various fields, such as image classification, object detection, natural language processing (NLP) and etc. DL models are prevalently becoming larger to achieve higher quality, with such trend expected to continue [1], [2]. Training large models necessitate an ever-increasing GPU memory footprint, which can hardly be satisfied by the slow growth rate of GPU memory capacity. The unsatisfied demand for training large models due to limited GPU memory prevents DL practitioners from experimenting and innovating cutting-edge models, thereby impeding the rapid advance of the DL field. Previous works [3]–[6] have reported that, the GPU memory usage during model training is dominated by the intermediate activation tensors (activations in short).

Activations are intermediate outputs generated by DL operators in the forward pass, and then kept in GPU memory until consumed to calculate the gradients in the backward pass. To reduce the memory occupancy of activations, a large number of techniques have been proposed. These techniques can be classified into three categories such as compressing [7]–[12], swapping [13]–[15], and checkpointing [5], [6], [16]–[18]. Compressing attempts to convert activation into its low-bit counterpart, and thus may affect the convergence and the model quality, because the iterative nature of training may lead to uncontrollable error propagation. Swapping offloads the activations from GPU memory to CPU DRAM in the forward pass, and asynchronously copies them back in the backward pass. Unfortunately, the copying overhead is quite high due to the limited PCIe bandwidth. Checkpointing allows dropping the activations in the forward pass and re-generating them by replaying the forward computation (i.e., re-computation) in the backward pass. In general, due to the lower overhead of re-computation than data transmission between GPU and CPU, checkpointing is widely adopted by popular DL frameworks such as TensorFlow [5] and PyTorch [16], [18] to reduce GPU memory consumption during model training.

The fundamental of a checkpointing technique is the GPU memory planner that decides when and where to drop and re-compute the activations. Depending on whether it requires prior knowledge of the model structure, the GPU memory planner can be further divided into static planners (e.g., Checkmate [5]) and dynamic planners (e.g., DTR [16]). The static planners commonly adopt a conservative plan regarding the largest input tensor to avoid GPU memory over-subscription. Whereas, the dynamic planners reactively checkpoint the activations with the lowest re-computing costs in a greedy manner, when running out of GPU memory. However, both types of planners fail to consider input dynamics during training, thus unnecessarily sacrificing the training performance for reduced GPU memory occupancy. For example, when the input size is small and the GPU memory is sufficient, the conservative checkpoint decisions of static planners regarding largest input size result in massive redundant computation, and thus unnecessarily degrade training performance. Whereas, due to the lack of holistic information about model structure and training process, dynamic planners may fail to generate optimal checkpointing plans, and thus achieve low training performance (details in Section III-B).

The input dynamics with changing activation sizes exist during training due to the diverse datasets and subsequent data augmentation, which results in changing GPU memory footprint during model training (details in Section III-A). Regarding the datasets, object detection datasets (e.g., COCO) contain many images with varying aspect ratios [19], whereas NLP datasets (e.g., SWAG) contain many multiple-choice

† Contributed equally. ✉ Corresponding author.

1530-2075/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPS54959.2023.00025

questions with varying text sequence lengths. Regarding the data augmentation, an image can be resized to a random size [20]–[22] to improve the model robustness. Whereas, a text sequence can be broken down into a sequence of word tokens during tokenization. Besides, several images/texts are then collated into a mini-batch after padding and truncation. However, the above input dynamics can hardly be exploited by current checkpointing planners to improve training performance without over-subscribing the GPU memory. To leverage the input dynamics, the checkpointing plans need to be determined during runtime adapting to the input dynamics, and then applied to the training process on the fly in order to further improve the performance.

Moreover, in many production scenarios, the DL models need to be frequently fine-tuned to fit the latest collected dataset to avoid the so-called "concept drift" [23] and improve the serving quality continuously. In such case, it is infeasible to obtain the input size distribution of the dataset and perform checkpointing plan in advance. In addition, considering the training pipeline, the checkpointing planner has no prior knowledge of input size distribution, data augmentation process, model structure, and model parameters, and thereby cannot predict the GPU memory usage of the model accurately. In sum, to exploit the performance opportunity of input tensor dynamics during model training, a checkpointing planner should address the following challenges: *1)* it should collect the GPU memory usage online without prior knowledge, *2)* it should predict the per-layer GPU memory usage accurately given arbitrary input tensor, and *3)* it should generate and apply checkpointing plans adaptively during runtime based on the prediction of GPU memory usage. Putting the above together, the introduced overhead should be trivial without offsetting the performance benefit of exploiting input dynamics.

In this paper, we propose *Mimose*[1], an input-aware checkpointing planner respecting the memory budget, while enabling efficient model training on GPU. The key feature of *Mimose* is that it dynamically adjusts the checkpointing plan according to the predicted memory usage of current input tensor, in order to maximize GPU memory utilization and minimize the performance overhead. *Mimose* builds a lightweight but accurate prediction model of GPU memory usage online without pre-analyzing the model, to achieve the sub-millisecond-level checkpointing planning for each input tensor. It generates a tensor checkpointing plan and applies the plan on the fly during the training process. By exploiting the input tensor dynamics, our experiments show that *Mimose* achieves superior training performance compared to state-of-the-art checkpoint planners under the same GPU memory budget.

Specifically, this paper makes the following contributions:

- We propose an online GPU memory estimator that predicts the memory usage of activation tensors for given input size. The estimator is constructed during model training without prior knowledge of the model structure or the input size. After negligible training iterations, the estimator can offer accurate enough memory prediction to facilitate generating checkpointing plans.
- We propose an effective checkpointing scheduler that generates and applies the checkpointing plans based on the memory prediction during runtime. In addition, it adopts a caching strategy to avoid re-generating the checkpointing plans for repeated input sizes redundantly.
- We develop the *Mimose* framework with the input-aware checkpointing planner for efficient training on GPU. *Mimose* works entirely online, and does not rely on either model pre-analysis or ahead-of-time memory planning. Our experiment results demonstrate that *Mimose* can achieve better training performance than state-of-the-art checkpointing frameworks.

[1]*Mimose* is open source at https://github.com/buaa-hipo/mimose-mmdet and https://github.com/buaa-hipo/mimose-transformers.

II. Background

A. Training Pipeline

Deep learning training often includes many iterations, with each iteration processing a mini-batch containing a few samples. In each iteration, samples are processed in pipeline as shown in Figure 1. Samples are first loaded from the training dataset and preprocessed through data augmentation and collation to form a mini-batch input tensor. During the *data augmentation phase*, for NLP tasks, text samples are collected from various sources with diverse text lengths. After tokenizing, samples are split as tokens and converted into sequences (e.g., *input_ids*). The sequence length varies across samples. For object detection tasks, image samples are collected with different sizes and aspect ratios. And the state-of-the-art models, such as DETR [20], Sparse R-CNN [21], and Swin Transformer [22], leverage the multi-scale resizing to improve the robustness, which randomly resizes the image samples such that the shorter side is between 480 and 800 while the longer side is at most 1,333. Besides, the scaling keeps the aspect ratios unchanged. Then during the *data collation phase*, smaller samples in a mini-batch are padded to match the largest sample, whereas the samples too large to be handled are truncated smaller. The pre-processed samples with uniform shapes are then collated to an input tensor. Note that the input tensor sizes can fluctuate across iterations due to the diversity of datasets and the flexibility of data augmentation.

During the *training phase*, the input tensor is used in conjunction with the model parameters to produce a set of scores, a process known as the *forward pass*. During the forward pass, the activation tensors are generated successively and stored in GPU memory for reuse in the backward pass. At the end of the forward pass, a train loss is derived by comparing the produced scores with the desired scores. After that, the loss is propagated through the entire model to compute the gradients of the model parameters, a process known as the *backward pass*. During the backward pass, the activation tensors are deallocated immediately once corresponding gradients are derived. Eventually, the gradients are scaled by a learning rate and used to *update* the model parameters.

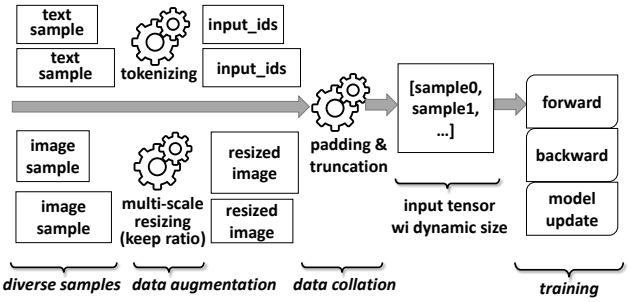


Fig. 1: Input tensor dynamics in the training pipeline, which is mainly caused by the dataset and data augmentation.

B. Checkpointing Planners

Mainstream memory planners reduce GPU memory footprint by techniques including compressing, swapping, and checkpointing. Compressing can have a significant impact on convergence speed and model accuracy [14]. The time cost incurred by swapping is more than 2× the computation time for most layers, which may slow down the training process [24]. Due to the above drawbacks, we leverage checkpointing for efficient model training on GPU while respecting the memory budget. Figure 2 shows the major difference across existing checkpointing planners, where the red arrow indicates the timing for generating checkpointing plan. Different from existing checkpointing planners, our approach (i.e., Mimose) generates the checkpointing plans at the start of each forward pass, which can better exploit the input tensor dynamics (Section IV). The detailed comparison between our approach and existing checkpointing and hybrid (checkpointing+swapping) planners is shown in Table I.

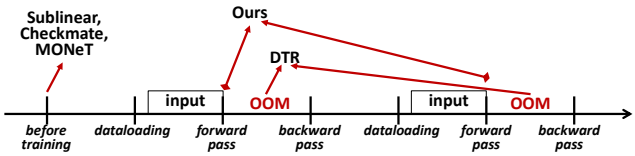


Fig. 2: Comparison across different checkpointing planners, where x-axis indicates the timeline. The red arrow indicates the timing for generating checkpointing plan.

III. Motivation

A. Memory Impact of Dynamic Input Size

The input size is represented by the number of elements in the input tensor for each mini-batch. The dynamic of input size comes from two aspects: dataset and data augmentation. Figure 3 shows the input size distribution when training Bert-base on SWAG, SQuAD, and GLUE-QQP datasets (batch size of 16, 12, and 32, respectively), and T5-base on UN_PC dataset (batch size of 8). Among different datasets, the range of input size is quite large, with 35∼141, 153∼512, 30∼332, and 17∼460 for the above four datasets, respectively. And the input size tends to follow a certain probability distribution, such as normal distribution and power-law distribution. Besides, the data augmentation technique of multi-scale resizing (e.g., in object detection tasks) may enforce pre-defined probability distributions.

Consequently, the dynamic of input size can greatly affect the GPU memory footprint of activation tensors. The memory footprint during each training iteration consists of model parameters, gradients, optimizer states, and activation tensors, where the first three are constant regarding the model structure and do not change across different input sizes. Figure 3 shows the GPU memory usage under various input sizes. With input size increasing, the memory usage increases accordingly. Besides, the GPU memory usage curve is quite smooth, revealing the possibility for accurate memory prediction with analytical models.

B. Inefficiency of Current Checkpointing Planners

Static checkpointing planners conservatively preserve memory for the largest input size and thus lead to low training throughput. Training a large DL model successfully on GPU means no OOM exception happens throughout all training iterations. Thereby with the dynamic of input size, static checkpointing planners have to conservatively generate checkpointing plans regarding the peak memory usage of the largest input size. Figure 4 illustrates the GPU memory usage of training Bert-base model on GLUE-QQP dataset (with batch size of 32) using *Sublinear* [6] under the GPU memory budget of 3 GB. *Sublinear* generates the checkpointing plan targeting the maximum input tensor size (e.g., with *seqlen*=300) to conservatively avoid OOM exception. Unfortunately, considering *Sublinear* applies the above plan to a smaller input tensor (e.g., with *seqlen*=55), it unnecessarily leaves 1.2 GB memory budget unused, which unnecessarily sacrifices training throughput. It also can be seen that even without checkpointing, the peak memory usage of a smaller input tensor may still be within the memory budget. Due to the conservatism of static planner, it fails to exploit the performance opportunity of input tensor dynamics for increasing training throughput. As shown in Figure 4, the degraded training throughput caused by *Sublinear* is non-trivial, which can be as large as 35%.

Dynamic checkpointing planners lack holistic model information and rely on runtime data collection, which leads to non-trivial runtime overhead and sub-optimal plans. Figure 5 shows the training time breakdown of Bert-base on SWAG dataset using *DTR* [16] under the GPU memory budget of 4.2/4.5/5/5.5 GB. However, the actual memory usage of *DTR* is 6.7/7/7.5/8 GB due to the severe memory fragmentation. In addition, *DTR* collects and maintains the checkpointing cost of all participated tensors during runtime. The overhead of maintaining the checkpointing cost takes 26.0% of the overall iteration time on average. With a lower memory budget, the overhead can reach 40.1% at most. The overhead of checkpoint planning also increases significantly when given tighter memory budgets, which can reach 11.9% at most and 7.2% on average. We notice such overhead exists even without any activation tensor dropped. Moreover, the

TABLE I: Comparison between *Mimose* and other checkpointing planners.

	Mimose	DTR [16]	Sublinear [6]	Checkmate [5]	MONet [17]	Capuchin [4]	MegTaiChi [25]	HOME [24]	STR [26]
Swapping	✗	✗	✗	✗	✗	✓	✓	✓	✓
Checkpointing	✓	✓	✓	✓	✓	✓	✓	✓	✓
Dynamic input	✓	✓	✗	✗	✗	✗	✗	✗	✗
Dynamic graph	✗	✓	✗	✗	✗	✗	✓	✗	✗
Mem. fragmentation avoidance	side-effect	✗	✗	✗	✗	✗	tensor partition	✗	✗
Granularity	block	tensor	layer	layer	tensor	tensor	tensor	tensor	tensor
Timing for generating plan	runtime	runtime	offline	offline	offline	runtime	runtime	offline	offline
Search space	holistic	currently traced tensors	segments	reduced	holistic	holistic	currently traced tensors	holistic	holistic
Search algorithm	greedy	greedy	greedy	MILP+approx.	MILP	greedy	greedy	PSO algo.	MILP+approx.
Solving time	short	short	short	<1 hour	hours	short	short	short	minutes

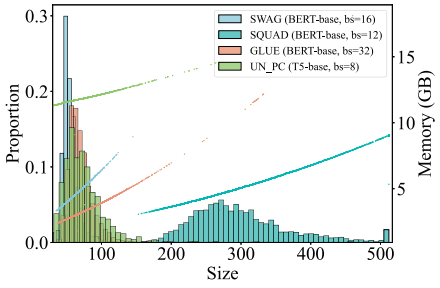


Fig. 3: Input size distributions of different datasets (left y-axis) and GPU memory footprints (right y-axis) when training Bert-base/T5-base with batch size set to 16, 12, 32, 8, respectively.

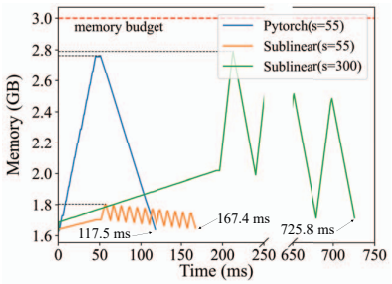


Fig. 4: Memory footprint curves of training Bert-base on GLUE-QQP dataset using static checkpointing planner, *Sublinear*. The memory budget is set to 3 GB.

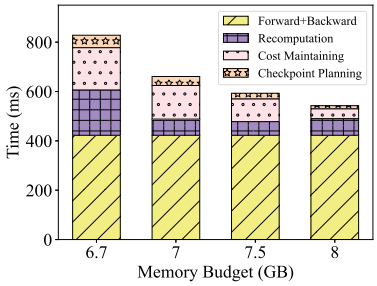


Fig. 5: Training time breakdown of Roberta-base on SWAG dataset using dynamic checkpointing planner, *DTR*. The memory budget is set to 4.2/4.5/5/5.5 GB (actually 6.7/7/7.5/8 GB used).

memory fragmentation can also reduce the available memory capacity, which makes the planning overhead further unpredictable. Another drawback of *DTR* is that it makes checkpointing decisions greedily when OOM exceptions happen, which prevents generating global optimal checkpointing plans, and thus hurts the training throughput.

IV. Design

A. Design Overview

Mimose is designed to be dynamic and agile in generating and applying checkpointing plans according to the input size on the fly. Due to this design philosophy, *Mimose* exists on the critical path of the training pipeline. Therefore, lightweight modules are designed for *Mimose* to ensure low overhead.

Specifically, *Mimose* is mainly composed of a shuttling online collector, a lightning memory estimator, and a responsive memory scheduler, as shown in Figure 6. The **shuttling online collector** collects the memory usage and forward computation time of each layer in a given DL model (e.g., encoder, attention). It can perform collection online without pre-analyzing the model even under insufficient GPU memory. The **lightning memory estimator** builds a memory prediction model based on the collected data and estimates the per-layer memory usage for each unknown input tensor size. The **responsive memory scheduler** is responsible for exploring a near-optimal checkpointing plan based on the estimated memory consumption and the computation time and then scheduling the activation tensors with negligible overhead.

Mimose divides the whole training process into two phases. During **1) sheltered execution**, *Mimose* leverages the shuttling online collector, which considers the DL model as a sequence of building blocks (e.g., encoder block, attention block). It modifies the forward calculation per block and executes it twice in a training iteration. At the end of this phase, the memory consumption data is fed to the memory estimator to train the memory estimation model. In our experience, this phase requires only 10∼30 iterations. During **2) responsive execution**, *Mimose* passes the augmented input tensor to the responsive memory scheduler. If there is a checkpointing plan of similar input size in its cache, the plan can be picked up directly. Otherwise, when cache miss occurs, the scheduler, together with the memory estimator, can derive the near-optimal checkpointing plan in less than a millisecond. In this phase, the online collector is frozen, and no additional knowledge is required.

B. Shuttling Online Collector

If with unlimited GPU memory, we can directly profile the model during the forward pass, for instance, compare the timestamp and memory footprint in different stages (e.g., before and after the forward computation of some layers) so

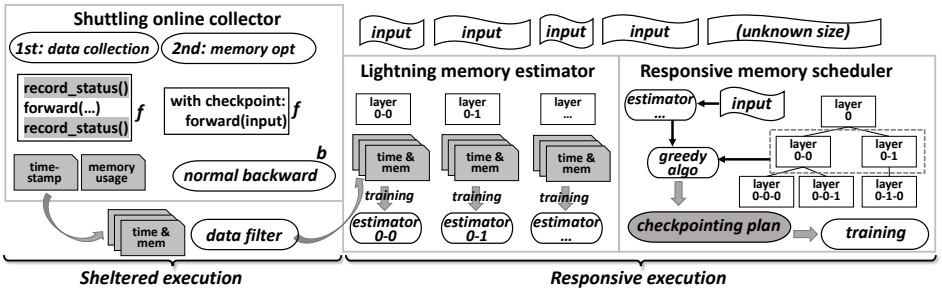


Fig. 6: Design overview of *Mimose*.

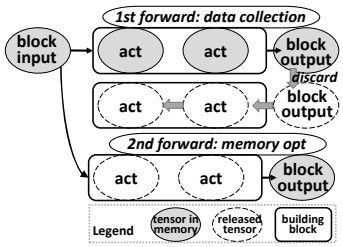


Fig. 7: Two forward passes of the shuttling online collector.

that we can get the memory occupation and computation time of each layer. However, given random-sized input tensors, the activation tensors can consume large memory and cause OOM exception. To ensure that the model can be trained properly, we need to apply the conservative checkpointing (e.g., *Sublinear*) during memory/time data collection. Besides, there is a contradiction between applying checkpointing and inspecting activation tensors: checkpointing means discarding the activation tensors instantly, and thus these tensors can be neither revisited nor inspected.

Therefore, we propose the shuttling forwarding, where the forward pass of each layer will be executed twice, as in Figure 7. A DL model is split as a sequence of building blocks (e.g., encoder block, residual block). The first forward computation is conducted as normal, but the final output tensors inside this block are discarded, and the activation tensors are dropped consequently. The second forward computation is conducted oppositely, with all activation tensors inside this block dropped instantly, except checkpointing the output tensor, so as to minimize the memory usage and prepare for the data collection for the next block. Note that the shuttling forwarding is conducted block by block, with the activation tensor between blocks kept in GPU memory, which also means that its memory footprint is the same as that of *Sublinear* planner [6].

As for the time overhead, compared to *Sublinear*, shuttling collector only repeats the forward pass in each training iteration. Additionally, compared to normal training without any memory planner, it takes more time for recomputation in each iteration. Since the time of holistic forward pass is generally shorter than that of backward pass, the overhead of shuttling collector is at most twice that of normal training iteration. Although its overhead per iteration seems large, *Mimose* requires a trivial number of iterations for collection. *Mimose* uses shuttling collector only when meeting new input size, so the overhead can be reduced to $O(\frac{n}{N})$ throughout the training process, where n represents the types of input size and N represents the total iterations. The data provided by the shuttling collector is used to train the memory estimator model (Section IV-C). If combined with a lightweight but accurate memory estimating model, the collector overhead can be reduced to a very low level.

C. Lightning Memory Estimator

The memory estimating model lies on the critical path of the training pipeline with *Mimose*. Therefore, it should satisfy the following rules.

- It should require less training data, since its training data has to be collected online during the sheltered execution.
- Its prediction should be fast enough, since its prediction is the prerequisite for generating checkpointing plan.
- It should be accurate enough to provide memory usage information to the subsequent checkpointing scheduler for reasonable plans.

The activation tensors in model training are actually composed of the output tensors of all operators in the model. Given a static model, the number of tensors forming the activation is constant, so we should focus on the size of each tensor. To construct the memory estimator model between input tensors and activation tensors, we study the relationship between them in representative DL operators, including both layers and neutral network structures, and we classify them into four categories, as shown in Figure 8.

Elementwise operators, such as `ReLU` and `add`, perform individual operations on each element of the input tensor. Therefore, the output tensor shares the same size as the input.

Fixed-output-sized operators, convert the input tensor to an output tensor with fixed size. For example, the `AdaptiveAvgPool` operator applies an adaptive average pooling over an input tensor and can output a tensor with a pre-defined size.

Operators with implicit reductions, which contain reduction operations as part of the operators, such as `Linear`, `GEMM`, `Convolution`, and `maxPool`. Specifically, as for `Linear`, `GEMM`, the iteration input tensor only determines their input tensor shape in only one dimension. While in other dimensions, their shapes are carefully designed by the DL experts after substantial hyper-parameter tuning, and thus are specially fixed during training. Therefore, the input-output tensor shapes have a deterministically linear correlation. As for `Conv`, `maxPool` and other operators with shape changes in multiple dimensions, this relationship is slightly more complicated. But these dimension sizes have deterministic relationships with extra variables that are also specially fixed,

such as stride, kernel size, and padding size, and thus the resulting output tensor size still has a linear correlation.

Typical structure with a set of operators can bring more complex memory usage relationship, such as `attention`, as shown in Figure 8. The input Q, K, and V has exactly the same shapes, i.e., $(seqlen, hidden\_size)$, where $hidden\_dims$ is specifically fixed in model, but $seqlen$ is linearly proportional to the iteration input size. After the first `Matmul` operation, $Q \times K^T$, a tensor with shape of $(seqlen, seqlen)$ is generated, which will increase the memory usage by $seqlen \times seqlen$. Later, after the `Scale` and `Softmax` operations, another two intermediate tensors with size of $(seqlen, seqlen)$ are generated. Therefore, the sizes of these intermediate tensors are quadratically correlated to the iteration input size. And one of them is multiplied with input tensor Q whose shape is $(seqlen, hidden\_size)$, to get a tensor with shape of $(seqlen, hidden\_size)$ as the final output. It is obvious that the output tensor size also has a proportional relationship with the input tensor, Q, K, or V. For this reason, the input tensor of other subsequent layers is still linearly correlated to the iteration input tensor, therefore, avoiding the size explosion due to function compositions.

From the above analysis, we find that the sizes of activation tensors in a general DL model are almost polynomially correlated to the size of input tensors, and it is at most quadratic in most cases. Moreover, the input size of each operator (both individual operators and structures) should be linearly correlated to the input tensor size of the mini-batch (i.e., current iteration) so that the memory usage of activation tensors can be abstracted a lightweight polynomial function.

Based on the above study, we finally choose the polynomial regression model in the memory estimator. Compared with other complex algorithms, such as XGBoost [27], it achieves a satisfying trade-off between accuracy and efficiency, and we will evaluate a variety of fitting algorithms in Section VI-E.

Although the polynomial model works well for NLP models, there are still some structures that do not conform to polynomial correlation especially for object detection models. For example, in the 2-stage object detection, the number of generated anchors/proposals is not fixed, because they are mainly related to the content (e.g., how many people in an image sample) of the input tensor. Since this fluctuation is unpredictable, it could have a negative impact on the memory estimator. Therefore, we leave the support of objective detection models for our future work. We perform memory reservation for these model structure by now and plan to apply some adaptive algorithms to the memory estimator.

D. Adaptive Memory Scheduler

The same structures in a model could have different memory usages. For simplicity, we define the *stage*, which represents a set of layers corresponding to the user-written model code structures, and we regard *stages* as natural separators to locate layers in the model. Take Swin Transformer for example, the patch merging structure on the boundary of each stage reduces the output tensor size of the previous stage by 50%, which

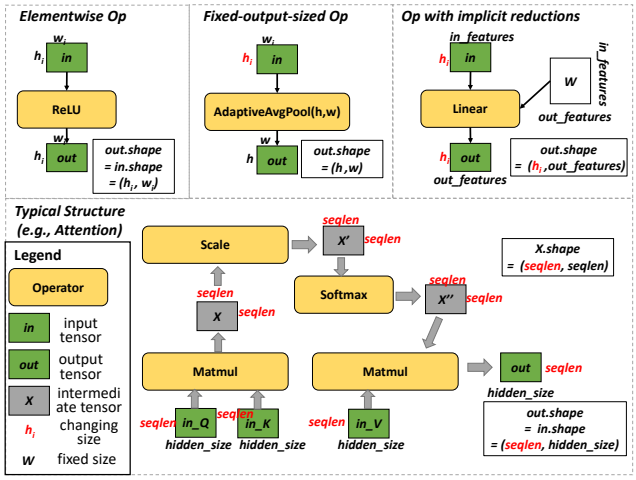


Fig. 8: Relationship between input tensor shape and output tensor shape of four representative operators.

leads to the step-down of memory usage in different stages. However, the first stage of ResNet has a different structure from the other stages, which does not show the same trend. In addition, even for stages with the same memory usage, checkpointing at different stages can still lead to different peak memory usages. Bert-base mainly contains 12 encoders. Figure 9 shows the peak memory usages of checkpointing different encoder structures under various input tensor sizes. Due to the characteristic of backward computation, if the checkpointed encoder is the last encoder in the model, this encoder's activation has to be restored instantly once the backward computation starts. At the same time, activation tensors of other encoders are not released in order to participate in the subsequent backward computation. It leads to a high peak memory usage, which is similar to the scenario without any checkpointing at all. Therefore, we prefer checkpoint layers/structures with earlier timestamps in forward pass when their activation tensors have similar sizes.

Based on the above observations, we adopt a greedy algorithm for checkpointing scheduling, as shown in Algorithm 1. We first derive the estimated memory usage of the given input tensor, leveraging the lightning memory estimator (line 1). And we assign the layers with similar estimated memory usage ($\pm 10\%$ in our implementation) to a bucket and sort them according to the execution sequence in forward pass (line 4$\sim$12). Then we select the layers that need to be recomputed one by one according to their activation sizes (line 13$\sim$21). When the excess memory (i.e., the memory usage that is out of the memory budget) cannot be covered by one layer, the remaining layer with the largest activation is selected as soon as possible (line 17). Otherwise, the layer whose activation size is nearest to the excess memory is selected (line 19). Note that we prefer to select layers with earlier timestamps within a bucket in order to further reduce the peak memory footprint. The selection procedure loops until the activation

size of selected layers exceeds the targeted excess memory. Our experiments shows that the greedy algorithm is simple but effective. However, *Mimose* still reserves a flexible interface for users to experiment with other scheduling algorithms, such as the Knapsack optimization and other DL approaches. If better algorithms are explored , we can quickly replace the current one.

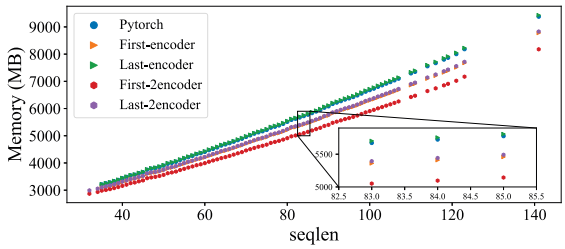


Fig. 9: Peak memory usages of checkpointing different encoders in Bert-base, which contains 12 encoders in total.

Algorithm 1 Greedy scheduling of the responsive scheduler.

Input: Memory budget M, input tensor size x, layer set L
Output: Set of dropped/recomputed layers L'

```
est_mem ← MemoryEstimator(x)
buckets ← empty list    // Buckets of layers
sorted(L, key=<estimated activation size>, order=desc)
while ! L.is_empty() do
    l ← L.top()
    L.remove(l)
    initialize new bucket with l
    while est_mem[L.top()] > est_mem[l] × 0.9 do
        l' ← L.top()
        bucket.append(l') and L.remove(l')
    sorted(bucket, key=<forward timestamp>, order=asc)
    buckets.append(bucket)
excess_mem ← (∑est_mem - M)
while excess_mem > 0 do
    bucket_candidates ← ∀b ∈ buckets, s.t. max(est_mem[b]) > excess_mem
    if bucket_candidates.empty() then
        l ← buckets.top().top() // layer with largest activation
    else
        l ← bucket_candidates.top().top()
    L'.append(l) and remove l from buckets
    excess_mem ← excess_mem - est_mem[l]
```

V. Implementation

The implementation of *Mimose* is on the basis of the checkpointing API (i.e., `torch.utils.checkpoint`) provided by PyTorch (since v0.4.0), so that *Mimose* is compatible with broad range of training codes written with PyTorch. Although the above choice makes it difficult to achieve tensor-level memory planning, it brings considerable performance benefits to *Mimose* in generating and applying ever-changing checkpointing plans. It is essential in scenarios with input tensor size dynamics, because *Mimose* is lying on the critical path of the training pipeline.

The data collection in *Mimose* is only performed by the shuttling collector during the sheltered execution phase (Figure 6). It collects samples of different input sizes during the first ten iterations (discussed in Section VI-E), which are used to train the memory estimator. After that, data collection is no longer required.

In sheltered execution, the data collector needs to collect per-layer memory usage and per-layer forward computation time during the model training online. And it wraps the forward pass and instruments before and after the forward computation of each layer. In such case, the memory usage and computation time can be derived by comparing the state differences.

In responsive execution, the memory scheduler holds a cache to store the generated checkpointing plans and uses the input tensor size as the indexing key. Whenever an input size is encountered, the scheduler firstly searches in cache, so as to avoid the overhead of generating plans repeatedly. In addition, the memory usages of similar input sizes are similar, and the generated plans are also similar. Therefore, they can also be the plans of each other. During the forward pass, *Mimose* looks for whether the ID of current layer exists in the previously generated checkpointing plan and determines whether to checkpoint accordingly, whose overhead is negligible.

VI. Evaluation

A. Experimental Setup

Hardware and software configurations. We conduct the experiments on a platform equipped with two-socket Intel Xeon E5-2680v4 CPUs (28 cores in total) and two NVIDIA V100 GPUs. The software environment contains Ubuntu 20.04 operating system, CUDA 11.3, cuDNN v8.2.0, PyTorch v1.11, HuggingFace transformers v4.18.0, and MMDetection v2.11.0.
Tasks, datasets, and models. We evaluate *Mimose* on four NLP tasks and two object detection tasks. Their datasets, models, and batch size settings are shown in Table II. Note that the NLP models involves Roberta-Base, T5-Base, and Bert-Base, which contains 125, 220, and 110 million parameters.
Comparison methods. We compare *Mimose* with the static checkpointing planners, *Sublinear* [6], *Checkmate* [5], and *MONeT* [17], and the dynamic checkpointing planner, *DTR* [16], under various GPU memory budgets. And we adopt the original PyTorch without checkpointing as the *baseline*. The original *DTR* implementation lacks support for several critical operators and fails to execute the above tasks. Therefore, we have extended *DTR* with a few operators, such as `cumsum`, `split`, `masked_fill`, etc.

As for *MONeT*, its author claims that it cannot work on NLP models because of lacking `embedding` and `gelu` implementations (refer to this issue[2]). We have tried to plan only an `encoder` with it and have tackled extra operators (e.g., `view`, `size`, `add_`), which takes one author up for two weeks. However, the memory consumption turns to be abnormal, thus we omit it on NLP tasks. We have also modified *MONeT* to

[2]https://github.com/utsaslab/MONeT/issues/2

support the ResNet backbones in MMDetection. Since *MONeT* requires offline checkpointing planning, we have allocated 8/12 hours to solve the ResNet50/101 backbones, respectively. According to its paper [17], 8 hours' solving is enough to reach 5% close to optimal solution.

As for *Checkmate*, it is implemented based on static graphs, and there are many problems in converting the models of the above tasks to static graphs. For example, the converted static graph fails to tackle the input tensor with dynamic size. Besides, it is built on TensorFlow and decided not to pursue PyTorch support due to the complexity of integration (refer to this issue[3]). For above reason, we use the *Checkmate* implementation from *MONeT* repository for comparison.

TABLE II: Training tasks for evaluation.

Abbr.	Task	Dataset	Model	Batch Size
MC-Roberta	Multiple Choice	SWAG	Roberta-B	16
TR-T5	Translation	UN_PC	T5	8
QA-Bert	Question Answering	SQuAD	Bert-B	12
TC-Bert	Text Classification	GLUE-QQP	Bert-B	32
OD-R50	Object Detection	COCO	ResNet50	8
OD-R101	Object Detection	COCO	ResNet101	6

B. Overall Performance

To comprehensively evaluate different planners, we present their execution times under different memory budgets. Figure 10 shows the execution times (10% samples of total dataset) for different planners normalized to *baseline* (original PyTorch without checkpointing). The "⋆" markers indicate the memory lower bound (when checkpointing all layers) and the memory upper bound (*baseline*, without checkpointing). As seen, *Mimose* significantly outperforms *Sublinear* with about 18.0% improvement. This is because *Sublinear* can only generate a static checkpointing plan based on the largest input tensor to avoid OOM, causing amounts of redundant recomputations for small input tensors. In contrast, *Mimose* can adaptively generate plans according to input tensors to minimize performance overhead.

Compared with the dynamic checkpoint planner *DTR*, *Mimose* improves the performance by about 15.0% on average. The reasons can be attributed to the following points. *1)* the cost maintaining time accounts for a high proportion of iteration time. *2) DTR* incurs a large planning overhead with limited time budget. When OOM exception happens, *DTR* inspects its currently maintained tensors and determines the tensors to be checkpointed greedily. Unfortunately, *DTR* is unaware of all tensors during each training iteration and thus cannot make global optimal plan. In contrast, thanks to the memory estimator, *Mimose* is aware of all participated tensors during each training iteration, and thus it can generate better plans than the greedy policy adopted in DTR. *3) DTR* generates lots of memory fragmentation at runtime, thus further impacting the quality of the plans. During each training iteration, *DTR* frequently releases/checkpoints tensors to accommodate new tensors. However, since the size of tensor varies, it is impossible to fully reuse the released non-sequential memory space, which leads to memory fragmentation. For the *MC-Roberta* task, *DTR*'s memory fragmentation reaches 2.5 GB with a 7 GB memory budget, whereas *Mimose*'s is only 0.5 GB. The memory fragmentation problem of *DTR* has also been observed in [25]. In contrast, *Mimose* generates and applies a checkpointing plan for each input tensor before the forward/backward computation. Therefore, the tensors that need to be checkpointed are scheduled in advance, without frequently releasing/allocating memory. In such case, *Mimose* can effectively mitigate memory fragmentation.

It can be observed that the performance of *Mimose* improves as the memory budget increases. For example, *Mimose* achieves only 2.6% slowdown compared to *baseline* under the memory budget of 8 GB. Furthermore, *Mimose* can still guarantee normal execution under a memory budget close to the lower bound (e.g., 3.36 GB for the *MC-Roberta* task). The above results indicate that *Mimose* can adjust the checkpointing plan for various input tensors and achieve lower overhead compared to other checkpointing planners.

On *OD-R50* and *OD-R101* tasks, we try to perform experiments under the memory budget of 14 GB, as solving *MONeT* and *Checkmate* plans for each budget can take about 20 hours on our platform. However, only *Mimose* and *Sublinear* can strictly obey the memory budget, while others exceed the memory budget. So we mark their peak memory consumption in the figures. Besides, on *OD-R101* task, *MONeT* and *Checkmate* are more than 2× slower that the *baseline*. Even though *Mimose* uses less memory, it still has lower overhead compared to *MONeT* and *Checkmate*, and we think they should take much more hours for solving satisfying plans.

C. Overhead Breakdown

We normalize the *Mimose* overhead to the duration of the training task executing one iteration. Table III shows the overhead breakdown under 6 GB memory budget when executing one epoch, where the total overhead is normalized to the single-iteration time. The *Mimose* overhead comes from three parts: data collector, memory estimator, and memory scheduler. Specifically, the data collector involves redundant computations caused by forwarding each layer twice. The memory estimator predicts the per-layer memory usage for each input tensor size. The memory scheduler generates a checkpointing plan based on the estimated memory usage and schedules the activation tensors. Note that *Mimose* will reuse the previous checkpointing plans without introducing overhead when executing iterations of the same input sizes.

It can be observed that the data collector accounts for about 12.0%~45.3% of the time within a single iteration due to forwarding twice and the slower first iteration. To this end, *Mimose* collects data by forwarding twice only in the first 10 iterations, and predict memory usage by memory estimator if necessary in the remaining iterations. In contrast, the overhead of memory estimator and scheduler is less than 1.25 ms,

[3]https://github.com/parasj/checkmate/issues/126

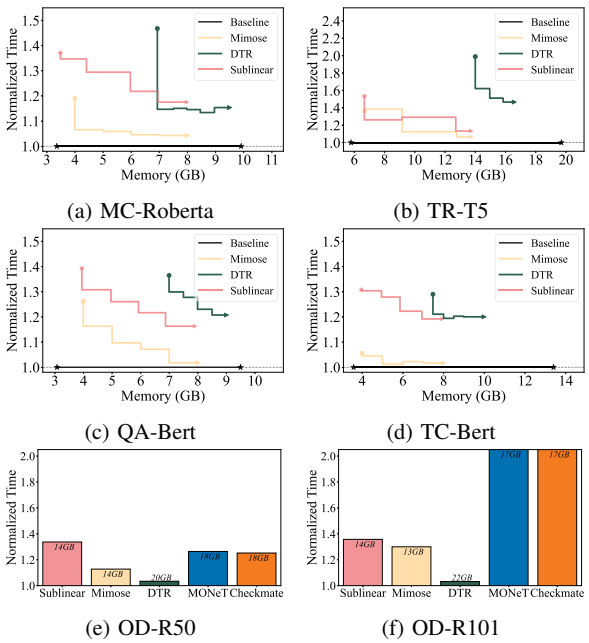


Fig. 10: Training times for different planners normalized to *Baseline* (original PyTorch without memory limit), where x-axis represents the memory budget. The "⋆" markers indicate the memory lower bound (when checkpointing all layers) and the memory upper bound (*baseline*, without checkpointing).

TABLE III: Overhead breakdown of *Mimose* under 6 GB memory budget when executing one epoch, where the total overhead is normalized to the single-iteration time.

Task	Collector	Estimator & Scheduler	Total
MC-Roberta (371.86 ms/iter)	145.99 ms (10 times)	0.26 ms˜0.32 ms (17 times)	1464.77 ms (3.93 iters)
TR-T5 (568.50 ms/iter)	257.22 ms (10 times)	0.47 ms˜0.64 ms (32 times)	2589.34 ms (4.55 iters)
QA-Bert (452.89 ms/iter)	118.96 ms (10 times)	0.27 ms˜0.38 ms (24 times)	1196.88 ms (2.64 iters)
TC-Bert (250.27 ms/iter)	157.73 ms (10 times)	0.27 ms˜0.34 ms (51 times)	1592.49 ms (6.36 iters)
OD-R50 (953.91 ms/iter)	114.95 ms (10 times)	0.29 ms˜0.70 ms (64 times)	1171.26 ms (1.23 iters)
OD-R101 (1080.16 ms/iter)	228.35 ms (10 times)	0.57 ms˜1.25 ms (120 times)	2368.39 ms (2.19 iters)

which is negligible compared to the single-iteration time (less than 0.2%), due to the lightweight estimation model and coarse-grained (stage-level) scheduling algorithm. Besides, the memory scheduler only needs to generate the checkpointing plan dozens of times during the entire epoch, as similar input sizes can share the same plan. The total overhead of *Mimose* is only 3.48 iterations on average, whereas the training of one epoch contains thousands of iterations. In sum, it is effective to improve performance under memory budgets by using *Mimose* to generate checkpointing plans for varying input sizes.

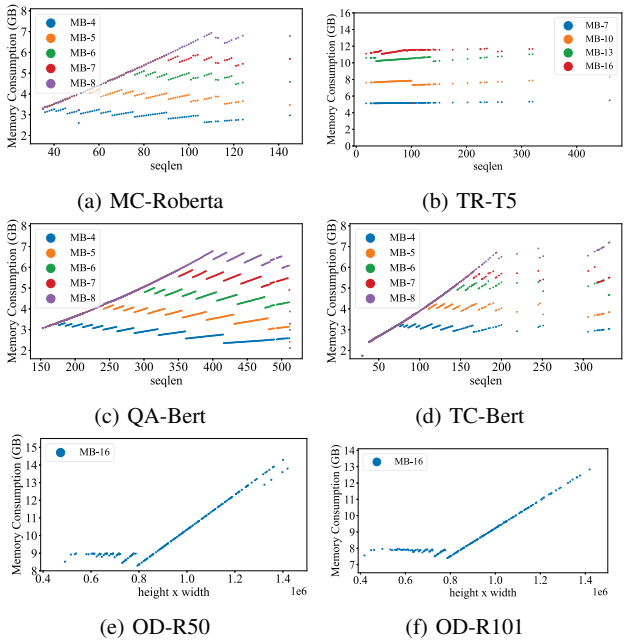
D. Memory Consumption

Mimose can adjust the checkpointing plan with the input size to minimize computational overhead. Figure 11 shows the memory consumption of *Mimose* processing varying sequence lengths, where MB-*X* refers to the memory budget of *X* GB. It can be observed that there is a small gap between the upper limit of memory consumption and the memory budget. This is because *Mimose* usually needs to reserve 0.5 GB˜1 GB of memory space to deal with possible memory fragmentation. In addition, there are a small number of points with particularly low memory consumption. The reason is that the data collector recomputes all modules in the first few iterations of the epoch to obtain layer-by-layer memory usage. In the next iterations, *Mimose* avoids redundant recomputations by predicting memory usage through the estimation model.

The memory consumption increases with the input size until the memory budget is reached. This indicates that for small input sizes, memory optimization is disabled to avoid introducing redundant computations. After reaching the memory budget, *Mimose* drops partial activation tensors through the checkpointing plan to reduce memory consumption. Since similar input sizes share the same checkpointing plan, the curve shows an upward trend of small segment separation with increasing input size. In addition, the curve of the latter segments under the same memory budget shows a downward trend. Consistent with other works [6], [16], the minimum recomputation unit of *Mimose* is a layer (or module in other literature), and the memory consumption of each layer is positively related to the input size. The above results demonstrate that *Mimose* can effectively utilize the memory budget by reducing memory fragmentation, thereby generating a near-optimal checkpoint plan with low computational overhead.

E. Memory Prediction

The memory estimator in *Mimose* can be formulated as predicting the memory usage of the model under the given input size. We evaluate six representative regression models as candidates for our memory estimator, including the polynomial regression model (with order $n = 1, 2, 3$), support vector machine (SVM), decision tree, and XGBoost. Specifically, we train the models using samples (i.e., per-layer memory usage under different input sizes) collected by the data collector, and compare the overall training time, prediction latency, and prediction error. The experimental results on TC-Bert are shown in Table IV. Specifically, for each layer of TC-Bert, the input of the prediction model is the input tensor size of the current iteration, and the output is the estimated memory usage of the layer. We sum up the estimated memory usage of all layers and compare it with the actual memory usage to calculate the relative prediction error. Except XGBoost, the training and prediction of the memory estimator candidates can be regarded as nearly zero-overhead compared to the per-iteration training time. For polynomial regression models with different orders ($n = 1, 2, 3$), the quadratic model achieves a very low prediction error (at the thousandth level), which demonstrates the correlation between the input sizes and



(a) MC-Roberta (b) TR-T5

(c) QA-Bert (d) TC-Bert

(e) OD-R50 (f) OD-R101

Fig. 11: The memory consumption of *Mimose* processing varying-sized inputs, where MB-*X* refers to the memory budget of *X* GB.

the memory usages conforms to the quadratic polynomial distribution. Other regression models fail to achieve the same level of prediction error, even with more training samples, due to their tendency to overfit. Furthermore, the experiment results of the quadratic polynomial model on four training tasks are shown in Table V. The quadratic polynomial model achieved low prediction errors (at the thousandth level) on all tasks, confirming that our observation in Section IV-C can be well generalized to NLP and CV tasks. Therefore, we adopt the quadratic polynomial regression models in our memory estimator, which are more lightweight and more accurate than other regression models.

TABLE IV: Prediction performance comparison of regression models on the text classification task, TC-Bert.

Regression Model	# Samples	Training Time (ms)	Prediction Latency (us)	Error
Polynomial (n=1)	10	0.90	14.78	4.04%
Polynomial (n=2)	**10**	**0.98**	**16.21**	**0.32%**
Polynomial (n=3)	10	1.01	17.88	0.32%
SVR	10	1.01	107.05	3.80%
SVR	50	2.70	110.39	3.56%
DecisionTree	10	3.98	82.97	5.67%
DecisionTree	50	21.15	82.25	1.50%
XGBoost	10	428.76	1348.26	5.13%
XGBoost	50	2504.11	1354.93	1.43%

VII. Related Work

DNN Model Compression – Since over-parameterization is a common property of DNN models, research works exploit compression techniques such as low precision, quantization, and pruning to reduce memory consumption. Compression techniques [7], [8], [28] usually reduce value redundancy by storing the compressed representation of feature maps and decompress them in the backward pass, however, they often require customized hardware designs to ensure (de)compression efficiency. Quantization [9], [10] and pruning [11], [12] techniques drop the unnecessary floating-point representations in model parameters, such as converting to lower-bit ones and removing useless ones, to reduce the memory footprint and computation intensity. However, these techniques have an unpredictable impact on model convergence.

TABLE V: Prediction performance of quadratic polynomial predictor on four training tasks.

Task	# Samples	Training Time (ms)	Prediction Latency (us)	Error
MC-Roberta	10	0.94	15.50	0.46%
TR-T5	10	0.99	14.54	0.10%
QA-Bert	10	1.18	16.45	0.33%
TC-Bert	10	0.98	16.21	0.32%
OD-R50	10	1.02	15.50	1.70%
OD-R101	10	1.11	18.36	2.29%

Model Checkpoint with Recomputation – Static checkpointing planners [5], [6], [17] collect model information and generate the checkpointing plan before training. In contrast, the dynamic planner (e.g., *DTR* [16]) can handle input dynamics by generating checkpointing plans on the fly. Its activation dropping decisions are triggered on demand by the OOM exception, which increases the checkpointing delay compared to static planners. Furthermore, a dynamic planner usually lacks holistic information about model training, leading to significant overhead. Note that *Mimose* differs from both the static and dynamic ones, and make a feature of input-aware checkpointing planning. *Mimose* can leverage the holistic model information for efficient planning, and can adjust the plans according to input tensors, so as to maximize GPU memory utilization and minimize the performance overhead.

Model Checkpoint with Swapping – Swapping techniques [13]–[15], [29] expand the scale of DNN training under limited memory capacity by offloading temporarily unneeded data to the CPU. There are also research works [4], [24]–[26], [30]–[33] combining swapping with checkpointing for hybrid memory optimization. However, swapping techniques may achieve high copy overhead due to limited PCIe bandwidth. Especially for varying input tensors, it is difficult to dynamically adjust swapping decisions to hide latencies by overlapping.

VIII. Conclusion

In this paper, we propose *Mimose*, an input-aware checkpointing planner for efficient GPU training under specific memory budgets. *Mimose* consists of a shuttling online collector, a lightning memory estimator and an adaptive memory scheduler, which together can dynamically and agilely adjust the checkpointing plan according to the predicted memory usage of the current input tensor to improve training throughput, while satisfying given GPU memory budgets. The experiment

results show that *Mimose* can achieve better training throughput for representative DL tasks compared to the-state-of-the-art checkpointing frameworks.

ACKNOWLEDGMENT

This work is supported by National Key R&D Program of China (No. 2020YFB1506703), National Natural Science Foundation of China (No. 62072018 and U22A2028), Special Fund for Basic Scientific Research of Central Universities, and SenseTime Research Fund. Hailong Yang is the corresponding author.

REFERENCES

[1] B. Ghorbani, O. Firat, M. Freitag, A. Bapna, M. Krikun, X. Garcia, C. Chelba, and C. Cherry, "Scaling laws for neural machine translation," in *International Conference on Learning Representations*, 2022.

[2] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[3] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[4] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, *Capuchin: Tensor-Based GPU Memory Management for Deep Learning*. New York, NY, USA: Association for Computing Machinery, 2020, p. 891–905.

[5] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica, "Checkmate: Breaking the memory wall with optimal tensor rematerialization," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 497–511.

[6] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

[7] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos, "Proteus: Exploiting numerical precision variability in deep neural networks," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12.

[8] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 776–789.

[9] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[10] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[11] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[12] S. Yang, W. Chen, X. Zhang, S. He, Y. Yin, and X.-H. Sun, "Auto-prune: automated dnn pruning and mapping for reram-based accelerator," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 304–315.

[13] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[14] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1341–1355.

[15] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 598–611.

[16] M. Kirisame, S. Lyubomirsky, A. Haan, J. Brennan, M. He, J. Roesch, T. Chen, and Z. Tatlock, "Dynamic tensor rematerialization," in *International Conference on Learning Representations*, 2021.

[17] A. Shah, C.-Y. Wu, J. Mohan, V. Chidambaram, and P. Kraehenbuehl, "Memory optimization for deep networks," in *International Conference on Learning Representations*, 2021.

[18] J. Feng and D. Huang, "Optimal gradient checkpoint search for arbitrary computation graphs," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 11 433–11 442.

[19] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal loss for dense object detection," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[20] 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*. Springer, 2020, pp. 213–229.

[21] P. Sun, R. Zhang, Y. Jiang, T. Kong, C. Xu, W. Zhan, M. Tomizuka, L. Li, Z. Yuan, C. Wang, and P. Luo, "Sparse r-cnn: End-to-end object detection with learnable proposals," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021, pp. 14 454–14 463.

[22] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," *arXiv preprint arXiv:2103.14030*, 2021.

[23] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under concept drift: A review," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346–2363, 2019.

[24] S. He, P. Chen, S. Chen, Z. Li, S. Yang, W. Chen, and L. Shou, "Home: A holistic gpu memory management framework for deep learning," *IEEE Transactions on Computers*, 2022.

[25] Z. Hu, J. Xiao, Z. Deng, M. Li, K. Zhang, X. Zhang, K. Meng, N. Sun, and G. Tan, "Megtaichi: dynamic tensor-based memory management optimization for dnn training," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–13.

[26] L. Wen, Z. Zong, L. Lin, and L. Lin, "A swap dominated tensor regeneration strategy for training deep learning models," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* IEEE, 2022, pp. 1–12.

[27] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794.

[28] J. Chen, L. Zheng, Z. Yao, D. Wang, I. Stoica, M. Mahoney, and J. Gonzalez, "Actnn: Reducing training memory footprint via 2-bit activation compressed training," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 1803–1813. [Online]. Available: https://proceedings.mlr.press/v139/chen21z.html

[29] J. Fang, Z. Zhu, S. Li, H. Su, Y. Yu, J. Zhou, and Y. You, "Parallel training of pre-trained models via chunk-based dynamic memory management," *IEEE Transactions on Parallel and Distributed Systems* vol. 34, no. 1, pp. 304–315, 2022.

[30] W. Jiang, Y. Ma, B. Liu, H. Liu, B. B. Zhou, J. Zhu, S. Wu, and H. Jin, "Layup: Layer-adaptive and multi-type intermediate-oriented memory optimization for gpu-based cnns," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1–23, 2019.

[31] S. G. Patil, P. Jain, P. Dutta, I. Stoica, and J. Gonzalez, "POET: Training neural networks on tiny devices with integrated rematerialization and paging," in *Proceedings of the 39th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., vol. 162. PMLR, 17–23 Jul 2022, pp. 17 573–17 583. [Online]. Available: https://proceedings.mlr.press/v162/patil22b.html

[32] M. Schuler, R. Membarth, and P. Slusallek, "Xengine: Optimal tensor rematerialization for neural networks in heterogeneous environments," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 1, dec 2022. [Online]. Available: https://doi.org/10.1145/3568956

[33] Y. Tang, C. Wang, Y. Zhang, Y. Liu, X. Zhang, L. Qiao, Z. Lai, and D. Li, "Delta: Dynamically optimizing gpu memory beyond tensor recomputation," *arXiv preprint arXiv:2203.15980*, 2022.