DELTA: Dynamically Optimizing GPU Memory beyond Tensor Recomputation

Yu Tang\textsuperscript{1}, Chenyu Wang\textsuperscript{2}, Yufan Zhang\textsuperscript{3}, Yuliang Liu\textsuperscript{3}, Xingcheng Zhang\textsuperscript{3}, Zhiquan Lai\textsuperscript{1}, Linbo Qiao\textsuperscript{1*,}\textsuperscript{*} Dongsheng Li\textsuperscript{1*}
\textsuperscript{1} Science and Technology on Parallel and Distributed Processing Laboratory
\textsuperscript{2} SenseTime Research
\textsuperscript{3} Shanghai Artificial Intelligence Laboratory

Abstract—Training activations of deep neural networks occupy plenty of GPU memory, especially for large-scale deep neural networks. However, the further development of deep neural networks is hampered by the limited GPU memory resource. Therefore, the optimal utilization of GPU memory resources is highly demanded. Swapping and recomputation are commonly applied to make better use of GPU memory in deep learning. As an emerging domain, several dilemmas remain: 1) The efficiency of recomputation is limited and swapping between GPU and CPU costs severe time delay; 2) There still lacks a dynamic runtime memory manager of tensor swapping and tensor recomputation nowadays; 3) Manually decisions for activations of training deep neural network require professional priors and experience. To remedy the above issues, we propose a novel memory manager named DELTA (Dynamic Tensor offLoading and recompuTAtion). To the best of our knowledge, we are the first to propose a reasonable dynamic runtime manager on the combination of tensor swapping and tensor recomputation without user oversight. In DELTA, we firstly propose a filter algorithm to select the optimal tensors to be released out of GPU memory and secondly present a director algorithm to select a proper action for each of these tensors. Furthermore, prefetching and overlapping are deliberately considered to overcome the time cost caused by swapping and recomputing tensors. Experimental results show that DELTA not only saves \textit{40\% - 70\%} of GPU memory, surpassing the state-of-the-art method to a great extent, but also gets comparable convergence results as the baseline with acceptable time delay. Also, DELTA gains $2.04 \times$ maximum batchsize when training ResNet-50 and $2.25 \times$ when training ResNet-101 compared with the baseline. Besides, comparisons between the swapping cost and recomputation cost in our experiments demonstrate the importance of \textit{making a reasonable decision on tensor swapping and tensor recomputation}, which refutes the arguments in some related work that swapping should be the first and best choice.

Index Terms—DELTA, dynamic tensor swapping and recomputation, prefetching, overlapping, memory management.

1 INTRODUCTION

Deep Neural Networks (DNN) have gained significant improvement in plenty of domains, such as image classification \cite{1, 2}, object detection \cite{3, 4}, text classification \cite{5, 6}, and machine translation \cite{7, 8}. It has been proven that larger models with more parameters come with stronger performance. However, training large-scale models \cite{8, 9, 10} with massive parameters requires large amount of GPU memory, and the memory of a single GPU cannot meet this requirement, and multi-GPU training needs a majority of resources. Besides, the scale of deep neural networks increases exponentially while GPU memory cannot keep pace with it and limits the further development of large scale deep neural networks. There is called \textit{GPU memory wall} \cite{11, 12}, as shown in Fig. 1. So there has been an urgent need to optimize GPU memory recently.

Memory management is another concern in deep learning and distributed systems \cite{13, 14, 15, 16}. The activations of training deep neural networks consume a lot of GPU memory. Recomputation and swapping are two common methods in the field of memory management. In the field of tensor recomputation, Chen \textit{et al.} \cite{17} proposed checkpointing and achieved training deep neural networks with sublinear memory cost. However, checkpointing is a static method requiring researchers to understand the models and checkpoint layers manually, which needs much efforts of

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig1.pdf}
\caption{The GPU memory wall problem \cite{11}. It is obvious that the transformer size increases much faster than that of GPU memory. GPU memory limits the further development of large-scale models.}
\end{figure}


1 more information about \textit{GPU memory wall} could be referred at https://github.com/amirgholami/ai_and_memory_wall

\textsuperscript{*}: corresponding author: \{qiao.linbo, dsli\}@nudt.edu.cn
researchers. Besides, there is an upper bound of memory saving for checkpointing. Dynamic Tensor Rematerialization (DTR) [18] targeted at tensors and adopted a dynamic strategy through which tensors are evicted based on heuristic functions and are reconstructed when needed. Checkmate [19] analyzes deep neural networks and automatically drops some operators at an appropriate time. On the other hand, Swapping is another method to reduce GPU memory consumption by offloading tensors or parameters to the CPU. SwapAdvisor [13] obtains the computation graph and optimize this graph by SwapPlanner. ZeRO-offload [20] offloads the parameters of models to CPU rather than the weights of tensors. Despite plenty of efforts in optimizing GPU memory, there are still several dilemmas remain: 1) The efficiency of recomputation is limited. Swapping between GPU and CPU costs severe time delay, which is even impractical in the training process; 2) There still lacks a dynamic runtime memory manager of tensor swapping and tensor recomputation nowadays; 3) Manually decisions for activations of training deep neural network require professional priors and much experience.

In this paper, we propose DELTA, Dynamic tEnson offLoading and recompuTAion, a novel memory manager involving dynamic tensor swapping and dynamic tensor recomputation simultaneously. For convenience, we present a new definition: Tensor-Releasing, which consists of tensor swapping and tensor recomputation. We show the position of DELTA within the context of typical deep learning frameworks in Fig. 2a and show its architecture details in Fig. 2b.

As stated in [24, 25], a good strategy combining swapping and recomputation requires overlapping between communication and computation as much as possible. Therefore, we design a prefetching algorithm to decline the time cost caused by frequent tensor swapping and recomputation. In DELTA, we adopt several overlapping policies to realize considerable overlapping of swapping and computation, which is another feature of DELTA. DTR [18] only uses recomputation to tensors while DELTA combines tensor swapping and tensor recomputation.

As it turns out, swapping is much more time-consuming than recomputation. In this way, we want to clarify whether we should recompute or reload those offloaded tensors when reusing them because swapping may cause tensors waiting in the computation process. Therefore, quite different from [24], we demonstrate the importance of making a more proper dynamic scheduling on tensor swapping and tensor recomputation case by case rather than adopting swapping as the first choice for GPU memory optimization. In our experiments, we evaluate DELTA on one and eight GPUs respectively, which means DELTA is also suitable for distributed training. Experimental results also evaluate the efficiency of DELTA.

The main contributions of this work are four-fold:

1) We combine dynamic tensor swapping together with dynamic tensor rematerialization rather than activation layers in [26], which is much more fine-grained. We propose a new dynamic memory manager, DELTA. DELTA achieves a higher memory saving level than DTR and the baseline automatically. With DELTA, researchers could spare more time to their professions.
2) To overcome the time delay of swapping and recomputation in DELTA, we design a prefetching strategy and several efficient overlapping policies. These overlapping policies help decrease the time cost caused by the frequent tensor swapping between GPU and CPU.
3) We conduct numerical experiments to evaluate the efficiency of DELTA. Experimental results show that
DETDA achieves nearly 40%-70% memory reduction compared with the baseline. DETDA could also save at least 30% GPU memory for a given batchsize compared with DTR.

4) Further experimental results demonstrate that reloading is not the best choice for those offloaded tensors and we should make proper decisions for those offloaded tensors when we try to get them back into GPU. This is distinct from previous work. Our experimental results about the comparison of reloading and recomputation provide convincing proof for our point of view.

The remainder of the paper is organized as follows: Section 2 summarizes some research related to this topic. Section 3 illustrates some assumptions and notations in our work. In Section 4, we present our core design and algorithms of DETDA. In Section 5, we give our experimental results and make some discussions according to these results. Section 6 concludes this paper by summarizing this research and providing our future work.

2 RELATED WORK

There is a large body of work on the optimization of GPU memory. Checkpoint [17] was proposed to train DNNs with sublinear memory cost by checkpointing every $\sqrt{n}$ layers, where $n$ is the number of layers in the model. However, it requires researchers to divide deep neural networks into several parts manually and requires professional experience. Moreover, Checkpoint cannot be applied into non-linear networks, such as Inception [27], ResNet [28], and U-Net [29]. In order to get rid of the weakness of Checkpoint, Kirisame et al. [18] suggests dynamic tensor rematerialization which finds an optimal tensor that has the minimum recomputation cost to evict. In [18], they propose some heuristic functions to pick out the optimal tensors to evict in the training process. Though it avoids checkpointing activations manually, it has a limit of memory saving. Jain et al. [19] regards tensor recomputation as a constrained optimization problem and discovers the best tensor recomputation strategy for deep neural networks.

Some other works resort to saving GPU memory through swapping tensors between GPU and CPU. SwapAdvisor [13], consisting of a scheduler, a swap planner, a simulator, and a sample selector, takes a dataflow graph as input and returns an augmented dataflow graph. SwapAdvisor focuses on tensors in the Deep Learning framework (MXNet [23] and proposes a fine-grained memory-optimization strategy. However, swapping tensors between GPU and CPU frequently results in an inevitably longer training time. SuperNeurons [25] adopt swapping and recomputation at the same time but only swaps convolution tensors. Actually, many tensors could be considered but SuperNeurons ignores them. It turns out that SuperNeurons needs more improvements. Rather than tensors, ZeRO-Offload [20] tries to offload model parameters to CPU and save significant GPU memory when training large-scale language models. Rhu et al. [30] provided a memory runtime manager, which makes the use of GPU and CPU memory virtual for each layer. There is a common shortcoming in these swapping methods. Frequent data swapping results in considerable training time costs. Bulo et al. [31] replaces ReLU [32] and Batch Normalization [33] layers with invertible variants, which saves memory consumption up to 50%. Capuchin [24], brought out by Microsoft in 2020, is another work combining swapping and recomputation. Capuchin achieves continuous execution even in the case that Out Of Memory (OOM) or Access Failure occurs. Besides, Capuchin considers prefetching swapped tensors on-demand and define MSPS. But Capuchin suggests swapping tensors comes first when both make decisions. In our experiments, we find that swapping tensors takes more time than recomputation. This motivates us to make reasonable decisions when integrating offloading together with recomputation. Beaumont et al. [26] also combines swapping and recomputation together but targets at activation layers in the training process, which is coarse-grained compared with tensor-level methods, such as DTR and DETDA. Moreover, another drawback is that it relies on the possibility of linearizing networks.

Pudipeddi et al. [34] proposed a layer-to-layer algorithm in Transformer-based models. Layer-to-layer executes a variant of ResNet, which is reversible. The activations in every layer could be computed from their following reversible layer’s activations. The memory cost of the Reversible Residual Network is independent of the number of layers. However, these methods requires manually design and professional researchers and our method avoids this problem.

3 PROBLEM FORMALIZATION

For simplicity, we clarify specific actions in DETDA, which are listed in Table I. Based on these definition, we make the following assumptions in DETDA:

Assumption 1. All the tensors in the training process are obtained by operators. These tensors are either constant or calculated by operators. Within each iteration, there is a sequence of tensor operations.

Assumption 2. Operations of offloading and eviction are acted in a synchronous manner. All the eviction and offloading operations must complete before the computation of the loss. In the backward pass, we only consider the recomputation and reloading operations. That is to say, no release operations will be performed in the backward pass.

Assumption 3. The communication process and computation process could be completely overlapped on an ideal condition.

Based on these assumptions, we aim to establish a new GPU memory optimization manager, settling the optimization limitation of dynamic tensor recomputation and the time delay of offloading tensors at the same time. Besides, we could make use of the communication time of tensor swapping to overlap with the computation process.
To address the issues above and design a dynamic memory manager, the first challenge is how to select out the optimal tensors to perform Release? The second challenge is to identify the appropriate action that should be performed once we obtain the most suitable tensors among all the tensors. After that, the most vexing issue is how to overlap the communication and computation to reduce the time delay caused by Tensor-Releasing. To be more precisely, our goal is to perform Offload with no or just a slight drag on the training time.

We clarify some actions in DELTA and for the sake of convenience, we present them and their definition in Table 1.

| Actions  | Meaning                                      |
|----------|----------------------------------------------|
| Release  | evict or offload tensors from GPU            |
| Evict    | evict tensors from GPU                       |
| Recompute| recompute tensors                            |
| Offload  | move tensors from GPU to CPU                 |
| Reload   | move tensors from CPU to GPU                 |

### 4 DELTA

As shown in Fig. 2b DELTA consists of three components, namely Filter, Director, and Prefetcher. The Filter is responsible for selecting the most suitable tensors and feeding them to the Director, which is in charge of determining actions for each tensor. Prefetcher acts as a role of reloading or recomputing tensors which reduces time delay. In this section, we present the core design and detailed algorithms in DELTA, which is summarized in Algorithm 1.

#### 4.1 Filter and Director

In order to determine which tensors to perform Release from GPU, we are supposed to design a mechanism to select out the optimal tensors among all the tensors on GPU. Inspired by [18], we define a filtering heuristic function regarding the memory consumption and the staleness time of each tensor.

Intuitively, no matter Evict or Offload is adopted for a tensor, one with enormous memory consumption and the longest staleness time is the most suitable one to be released from GPU. For a tensor, the more significant memory it occupies and the longer the staleness time it has, the more suitable it is to perform Release from GPU if ignoring the swapping cost and the recomputation cost. Thus, we arrive at our filtering heuristic function, as suggested in Eq. (1).

$$H^\text{base}_\text{filter}(t) = \frac{1}{m(t) \times s(t)},$$

where $m(t)$ and $s(t)$ are the memory cost and the staleness time of tensor $t$ respectively.

Beside our base filtering heuristic function, we also adopt two other filtering heuristic functions, $H^\text{filter}_\text{RU}$ based on the “least-recently used” policy and $H^\text{filter}_\text{GREEDY}$ based on the “greedy” strategy [44], namely Eq. (2) and Eq. (3) respectively. These heuristic functions are also compared in Section 5.

$$H^\text{RU}_\text{filter}(t) = \frac{1}{s(t)},$$

$$H^\text{GREEDY}_\text{filter}(t) = \frac{1}{m(t)}.$$  

After filtering the optimal tensors to release from GPU, we next set out to determine whether to perform Evict or Offload for these tensors. Then we define a decision function, Eq. (4), to make the decision,

$$F_\text{decision}(t) = \frac{h_r(t)}{h_o(t)},$$

where $h_r(t)$ is calculated by Eq. (5),

$$h_r(t) = \frac{c_r(t)}{m(t) \times s(t)},$$

where $c_r(t)$ is the recomputation cost of tensor $t$, including its parent tensors, defined in [18]. On the other hand, $h_o(t)$ is defined in Eq. (6),

$$h_o(t) = \frac{c_o(t)}{m(t) \times s(t)} = \frac{1}{b \times s(t)},$$

where $b$ is the bandwidth of the system. Empirically, only 35% of the bandwidth is used. As pointed out previously, $c_r(t)$ and $c_o(t)$ stand for the recomputation cost and swapping cost for tensor $t$ respectively. Thus, Eq. (4) can be
Algorithm 2 Filter: finding the optimal tensors to perform Escape at current iteration.

Input: Heuristic function \( h_t \), current iteration number \( t \).
Output: optimal tensor \( t \) from Filter.

1: Get tensor set \( O_t \) which stay on GPU now.
2: for each tensor \( t \) in \( O_t \) do
3: Compute \( s_i = h(t_i) \);
4: Append \( s_i \) into score set \( S \).
5: end for
6: find the minimum \( s_i \) in \( S \) and corresponding tensor \( t \)
7: return \( t \)

Algorithm 3 Director: making decisions to the optimal tensor \( t \) from Filter.

Input: Decision function \( h_d \), the optimal tensor \( t \) at iteration number \( t \).
Output: decision action.

1: if \( t \) is evicted_pinned then
2: return Offload;
3: else if \( t \) is offload_pinned then
4: return Evict;
5: else
6: Compute decision score \( s_d = F_d(t) \).
7: if \( s_d \leq 1 \) then
8: return Evict;
9: else
10: return Offload;
11: end if
12: end if

Algorithm 4 Prefetcher: Prefetching incoming tensors.

Input: Offloaded tensor queue \( Q \), the max reloading number \( n \).

1: while memory is enough or reload_count \( n \) do
2: Find the first tensor \( o \) in \( Q \).
3: perform Reload(o).
4: delete \( o \) in queue \( Q \).
5: end while

regarded as a comparison between the recomputation cost and swapping cost of the same tensor. If the computation cost is more negligible than its swapping cost, we perform Evict to these tensors. Otherwise, we apply Offload to them.

In general, Eq. (4) has the same meaning as memory saving per second (MSPS) [24].

If \( h_o(t) \) in Eq. (4) is ignored, we would get the heuristic function defined in DTR [18], which only performs Recomputation to the tensors. DELTA takes swapping into consideration beyond recomputation and clarifies the decision of tensors through Eq. (4).

4.2 Prefetcher and Overlapping Strategies

It is worth mentioning that both Evict and Offload in Tensor-Releasing get GPU memory-optimized at the cost of training time. As pointed out by [24], a suitable method involving swapping and recomputation should be largely overlapped and proceed with a minor time delay. This section discusses our design about prefetching and overlapping in DELTA.

The prefetching algorithm is summarized in Algorithm 4. In detail, when there is enough memory and the tensor required to be reloaded, we reload the tensors in the offloaded queue as many as possible. However, we observe that if we reload plenty of tensors into GPU, there is a high probability that these tensors are chosen to Offload again, which incurs extra time cost beyond our purpose. In order to remedy this problem, we set a threshold to limit the number of reloaded tensors. In Algorithm 4, we only perform Reload for all offloaded tensors rather than performing Recompute to those which are not uncomputable, this is another overlapping strategy making full use of the swapping time cost.

There are still offloaded tensors that are not prefetched into GPU memory and will be executed. As we have observed that for the same tensor, the swapping cost is higher than the recomputation cost, it will severely impact the training process if we reload those offloaded tensor only when this tensor is going to be conducted. As stated in Section 4.3, we adopt Recompute to some tensors and reduce the time cost to a great extent. Moreover, we set an early start strategy when offloading tensors as shown in Algorithm 4. Our overlapping strategies also consist of copying memory asynchronously from GPU to CPU and placing copy into CUDA memory stream out of the CUDA computation stream [15]. These strategies ensure that no severe extra time delay occurs in the whole training process.

4.3 Implementation and Algorithms

In this section, we mainly display some core codes and algorithms of DELTA. DELTA is implemented in C++. We define a struct called DELTATensor, consisting of several attributes for each tensor. The details of the code is shown below:

```cpp
struct DELTATensor{
    bool evicted;
    bool evict_pinned;
    bool todelete;
    bool offload_pinned;
    bool onGPU;
    bool swapout;
    bool uncomputable;
    uint64_t compute_cost;
    uint64_t swap_cost;
    uint64_t memory;
    Opcode opcode_;
    std::shared_ptr<CachedStorage> cpuStorage;
    std::vector<std::weak_ptr<DELTATensor>> parents;
    std::vector<std::weak_ptr<DELTATensor>> children;
}
```

In each DELTATensor, we set some variables to judge each state of this tensor and list them and their meanings as follows.
For simplicity and clarity, we use these flags in our algorithms. These variables get changed adaptively in the runtime of the training process. The algorithms of Filter, Director, and Prefetcher are summarized in Algorithm 2 and Algorithm 3 respectively.

We use Algorithm 2 to pick out the most suitable tensor to perform Evict or Offload if the GPU memory is not sufficient when training deep neural networks. The heuristic function $F_R$ in Algorithm 2 is discussed in Section 4.1. Algorithm 2 selects the corresponding tensor, which has the minimum heuristic function score.

The decision algorithm is displayed in Algorithm 3. Algorithm 3 takes the result tensor $o_t$ of Algorithm 2 as the input and outputs a decision for it. To be more specific, if $o_t$ is evict_pinned, which means that we cannot perform Evict to this tensor, Algorithm 3 returns Offload. Similarly, if $o_t$ is offload_pinned, Algorithm 3 returns Evict. Otherwise, we feed this tensor into Eq. (4) and compute its decision score $s_d$. Then if $s_d$ is less than 1, which means performing Evict costs less than performing Offloading, $o_t$ will be performed Evict in DELTA. If not, we perform Offload with regard to this tensor.

The main algorithm is outlined in Algorithm 1. DELTA proceeds with the following steps. Firstly, we check the state of the current tensor $o$. If it is evicted, we perform Recompute. Considering those offloaded tensors, we have two choices to get them into GPU again. If $o$ stays uncomputable, it cannot be recomputed anymore. Therefore, Reload is the only way we can do to move it to GPU. In contrast, DELTA recomputes this tensor which mitigates the drawbacks of the extra time cost of swapping tensors between GPU and CPU. This is a great improvement considering the experimental conclusion in Section 5.5. Secondly, we find the suitable tensor to be released from GPU through Algorithm 2 and get the decision through Algorithm 3. According to the decision, we choose different actions for the output tensor of Algorithm 2. Repeat the process above until the memory is enough to continue the training process. As for prefetching and overlapping, we will discuss them later in Section 4.2.

In practice, DELTA gets started when three-quarters of the memory budget is used rather than when the allocated memory is beyond the budget to increase the overlapping degree of DELTA. This is also one of our approaches to achieve overlapping.

5 EXPERIMENTS

5.1 Setup

To evaluate the efficiency of DELTA, we compare our method with DTR and baseline, which means training deep neural networks without any memory optimization methods. Our experiments are conducted on a cluster equipped with eight NVIDIA Tesla A100 GPUs within one node, together with CUDA Toolkit 11.2 and cudnn 8.0. This system is equipped with stable PCIe 4.0, whose bandwidth is generally 64GB/s. In our experiments, we evaluate DELTA on both one GPU and 8 GPUs in a distributed way respectively, which gains nearly the same results. We choose Adam [46] as our optimizer and train deep neural networks on ImageNet [47] in our experiments. What we want to compare is the performance of DELTA and DTR on the same condition, including the same budget and the same hardware environment. So in our experiments, the budgets of DELTA and DTR are also set to be the same under the same batchsize setting. We don’t need to seek a minimal budget setting for both DTR and DELTA.

5.2 Memory Saving

In this section, we mainly compare the memory saving among different methods to evaluate the efficiency of our method. Fig. 3 shows the memory consumption of ResNet-50 utilizing DELTA, DTR and baseline respectively with different batchsizes. We use $\text{torch.cuda.max_memory_allocated}$ to track the maximum memory consumption in the training process. We set the same budget for DTR and DELTA to make it a fair comparison. From Fig. 3 we spot that when the batchsize is 1814, both baseline and DTR get stuck into the out-of-memory (OOM) problem but DELTA could train ResNet-50 with 62.6 GB memory cost. DELTA saves more GPU memory compared with DTR and baseline. When the batchsize is 256, DELTA trimmed the memory consumption from 21.3 GB to 8.4 GB, saving GPU memory about 60.56% for ResNet-50. DELTA achieves the maximum memory saving when batchsize is 128, reducing memory consumption from 11.2 GB to 4.11 GB, getting 63.3% spare GPU memory. However, DTR only saves 6.25% from 11.2 GB to 10.5 GB under the same condition. It is clear that despite DTR improves the training batchsize to some degree compared with baseline, DELTA saves more memory and achieves a larger training batchsize. In conclusion, DELTA achieves about 40%-63% memory saving compared with baseline.

In Fig. 3 we observe that the memory consumption of the baseline is nearly linear. The memory consumption of DTR is not linearly increasing. For DTR, when batchsize increases from 256 to 512, even 1072, the ratio of memory increasing declines. This is because DTR creates severe memory fragments. DELTA focuses on the combination of tensor swapping and tensor recomputation, which gains a more significant optimization performance under the same memory budget setting. We find that when the batchsize increases from 128 to 512, the memory consumption of DELTA is nearly linear. However, when batchsize changes from 512
For ResNet-50, we compare the memory consumption when the batch size varies from 128 to 1814.

### TABLE 2: The memory consumption comparison of some typical models with different methods (MB).

| Models     | ResNet-101 | ResNext-50 | BERT         |
|------------|------------|------------|--------------|
| baseline   | 16553      | 14726      | 19845.4      |
| DTR        | 16059      | 13988      | 19071.3      |
| DELTA      | 4577       | 5356       | 19023.5      |

In addition, we compare the memory consumption of some other typical models, consisting of ResNet-101, and ResNext-50, with different methods and show the results in Table 2. Table 2 illustrates that compared with baseline, DELTA reduces the memory cost from 16553 MB to 4577 MB, saving about 72.3% of the GPU memory for ResNet-101. As for ResNext-50, DELTA also gets a GPU memory saving of 63.6%. Besides, we train BERT-base on SQuAD with the pre-trained model using a batch size of 32. Compared with baseline, DELTA reduces more than 800 MB for the BERT-base model, saving nearly 7.5% of the memory.

### 5.3 Maximum Batchsize

In this section, we try to explore the maximum batch size with DELTA on ResNet-50 and ResNet-101. In this experiment, we make full use of the whole GPU memory and summarize the results in Table 3 and Table 4. With DTR, we could train ResNet-50 with a batch size of 1072, which is nearly 1.2× compared with the baseline. As for DELTA, the max batch size reaches 1814 for ResNet-50, earning a growth of 2.04×.

DELTA behaves nearly the same in training ResNet-50 and ResNet-101. The max batch size of training ResNet-101 is 1336 using DELTA, obtaining an increment of 1.28× and 2.25× compared with baseline and DTR, respectively.

Besides, we observe that for ResNet-50 and ResNet-101, DTR only uses 32.8GB, which is far less than 80GB. We assume that this bottleneck mainly results from plenty of memory fragmentation caused by DTR. DELTA consumes 62.6 GB and 45.05 GB when training ResNet-50 and ResNet-101 respectively, which is much lower than DTR and the baseline. It turns out that DELTA alleviates the harmful effects of memory fragmentation brought about by DTR. However, there still needs much effort to remedy this issue completely.

### 5.4 Overlapping

Taking all of the overlapping strategies into consideration mentioned in Section 4.2, we compare processes of DELTA, with and without overlapping strategies in detail. We use Nvidia’s Nsight Systems to analyze the timeline for each process within one iteration and show them in Fig. 4 and Fig. 5 respectively.

![Diagram](image)

Every block in both Fig. 4 and Fig. 5 represents an operator in the training process. `Memcpy` is one of the most used CUDA operator, which acts as a data transfer in DELTA. Fig. 4 tells us that without overlapping, each operator proceeds in sequential order. After we adopt these overlapping strategies, we get communication and computation overlapped in DELTA. In this way, we reduce the time by almost 20% per iteration. In Fig. 5, the purple blocks represent the tensor offloading process in the forward process and the cyan blocks represent the tensor reloading process in the backward process. We notice that there are nine offloading processes while there are only two reloading processes. This is because the other seven offloaded tensors that are recomputed into GPU to achieve overlapping as much as possible.

### 5.5 Reloading or Recomputing for Offloaded Tensors?

In practice, we are supposed to perform Reload to those tensors that offloaded to CPU in the training process. But we could perform either Reload or Recomputation back into GPU technically. Although [24] suggests that reloading is the first choice when faced with a tensor offloaded from GPU, we are still confused which action is better for offloaded tensors. In this section, we will answer this question according to our experimental results. No matter recomputing or reloading those offloaded tensor, each method has its own cost.

To make the results more clear, we conduct various experiments and compare the average swapping cost and recomputation cost on some familiar and typical tensors, such as BNForward, ConvForward, and PoolingForward.

### TABLE 3: The max batch size of different methods of ResNet-50 within one A100 GPU.

| Methods | max batchsize | Growth | MC (GB)* |
|---------|---------------|--------|----------|
| baseline | 891           | 1      | 71.34    |
| DTR     | 1072          | 1.20×  | 32.87    |
| DELTA   | 1814          | 2.04×  | 62.6     |

*MC is the short of memory consumption.

### TABLE 4: The max batch size of different methods of ResNet-101 within one A100 GPU.

| Methods | max batchsize | Growth | MC (GB)* |
|---------|---------------|--------|----------|
| baseline | 594           | 1      | 71.34    |
| DTR     | 761           | 1.28×  | 28.65    |
| DELTA   | 1336          | 2.25×  | 45.05    |

*MC is the short of memory consumption.
Fig. 4: The detailed timeline of each operator of training ResNet-50 without overlapping. There is no overlapping between the computation and communication process.

Fig. 5: The detailed timeline of each operator of training ResNet-50 with overlapping strategies. DELTA achieves overlapping during the offloading process and full overlapping during reloading process with our overlapping strategies.

TABLE 5: The comparison between the swapping cost ($c_s$) and the recomputation cost ($c_r$) of the same tensor.

| Tensor name         | $c_s$ | $c_r$ |
|---------------------|-------|-------|
| BNForward(4034)     | 96318 | 6     |
| BNForward(6963)     | 22230 | 4     |
| BNForward(6526)     | 22237 | 4     |
| BNForward(6590)     | 11148 | 4     |
| ConvForward(96640)  | 22805 | 69    |
| PoolingForward(33219)| 11641 | 25    |

Tensors in Table 5 are usually likely to be chosen to release from GPU in our experiments. The number in “()” is the relative unique id for them. $c_s$ and $c_r$ are the swapping cost and recomputation cost respectively. It turns out that for these tensors, the swapping cost is much higher than the recomputation cost, especially for BNForward. It is obvious that the recomputation cost is smaller from Table 5. Therefore, we prefer performing Recomputation rather than Reload for these offloaded tensors, which is quite different from Capuchin[24]. This view is helpful to achieve reducing time delay when we try to get some offloaded tensors back onto GPU again. Thus, we have evidence that swapping is not generally the best choice when we can either perform Evict or Offload to the same tensor.

In fact, from the timelines in Section 5.4 we can also tell thatMemcpy costs more time than other computation operators, which means that swapping results in severe time delay. These data in Table 5 are also consistent with this notion. Thus, we adopt Recomputation for those tensors which has been offloaded from GPU to reduce the time delay caused by swapping. In other words, Recomputation is our first choice on the occasion that we could perform both Recomputation and Reload according to the conclusion that recomputation cost is relatively more smaller than its swapping cost. This choice wildly differs from the choice in [24]. That is to say, we are supposed to make reasonable choices to offloaded tensors rather than choose to reload them directly.

Fig. 6: The comparison of convergence values and time consumption.

5.6 Convergence

In this section, we examined the convergence of DELTA. We utilize the Adam optimizer and MultiStep learning rate scheduler to train ResNet-50 with 8 GPUs on ImageNet [51] with batchsize 128 and 90 epochs in our experiments, and the training results are summarized in Fig 6. In our experiments, we mainly focus on the convergence of DELTA rather than the performance. Thus, we don’t use any special tricks in our experiments and these experiments are conducted under the same setting. The baseline process gets a final training loss on ImageNet [47] of 1.5947. DELTA achieves a training loss of 1.6054, nearly the same as the baseline. It is worth mentioning that DELTA almost produces the same performance as the baseline, even with a slight time delay. However, DELTA saves 60% of the GPU memory according to the results in Section 5.2. With all of our overlapping policies, DELTA costs 0.551 seconds each iteration, which is acceptable.

5.7 Comparison among Different Heuristics in Filter

In this section, we mainly compare different filtering heuristic functions of Filter in Section 4.1 under different settings of memory budget. We conduct experiments on ResNet-50, EfficientNet and ResNeXt-50 and show their
In this paper, we propose a novel GPU memory manager named DELTA, Dynamic Tensor offLoading and recompuTAtion. DELTA is the first work combining dynamic tensor swapping and dynamic tensor recomputation to the best of our knowledge, which breaks the memory saving limitation of current recomputation methods and alleviates the severe time delay when only use swapping at the same time. Besides, DELTA achieves promising GPU memory saving without user insight. With DELTA, researcher could devote more efforts to their professions. Our work is inspired by DTR [18]. Nevertheless, the difference between our work and theirs is distinct. As a dynamic memory manager, DELTA has three components, including Filter, Director, and Prefetcher, each acting a vital role in the system. In detail, we firstly propose an optimal filter algorithm to select which tensors are the most suitable to perform Release among all the tensors which have been stored in GPU memory. Secondly, we present a director algorithm to choose a proper action for each tensor selected by our filter algorithm. Moreover, to reduce the time delay, we adopt prefetching and overlapping strategies to make communication and computation overlap as much as possible. Experimental results demonstrate the benefits of DELTA over baseline and DTR. With DELTA, we could save up to nearly 40%-63% of the GPU memory consumption while training ResNet-50, and we could train ResNet-50 with a max batch size of 1814, which is 2.04× compared with the baseline. We could also train ResNet-101 with a memory saving up to 70% with DELTA. Experimental results show that DELTA is effective in saving the memory of BERT as well. More importantly, DELTA also gets a comparable convergence result while optimizing GPU memory. Our experiments demonstrate the importance of making a good decision on tensor swapping and tensor recomputation rather than choosing swapping directly stated in [24]. Although DELTA reduces the memory fragmentation in training compared with DTR, how to use memory fragmentation is still worthy of inclusion in our future work.

6 Conclusion

In this paper, we propose a novel GPU memory manager named DELTA, Dynamic Tensor offLoading and recompuTAtion. DELTA is the first work combining dynamic tensor swapping and dynamic tensor recomputation to the best of our knowledge, which breaks the memory saving limitation of current recomputation methods and alleviates the severe time delay when only use swapping at the same time. Besides, DELTA achieves promising GPU memory saving without user insight. With DELTA, researcher could devote more efforts to their professions. Our work is inspired by DTR [18]. Nevertheless, the difference between our work and theirs is distinct. As a dynamic memory manager, DELTA has three components, including Filter, Director, and Prefetcher, each acting a vital role in the system. In detail, we firstly propose an optimal filter algorithm to select which tensors are the most suitable to perform Release among all the tensors which have been stored in GPU memory. Secondly, we present a director algorithm to choose a proper action for each tensor selected by our filter algorithm. Moreover, to reduce the time delay, we adopt prefetching and overlapping strategies to make communication and computation overlap as much as possible. Experimental results demonstrate the benefits of DELTA over baseline and DTR. With DELTA, we could save up to nearly 40%-63% of the GPU memory consumption while training ResNet-50, and we could train ResNet-50 with a max batch size of 1814, which is 2.04× compared with the baseline. We could also train ResNet-101 with a memory saving up to 70% with DELTA. Experimental results show that DELTA is effective in saving the memory of BERT as well. More importantly, DELTA also gets a comparable convergence result while optimizing GPU memory. Our experiments demonstrate the importance of making a good decision on tensor swapping and tensor recomputation rather than choosing swapping directly stated in [24]. Although DELTA reduces the memory fragmentation in training compared with DTR, how to use memory fragmentation is still worthy of inclusion in our future work.

References

[1] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” arXiv preprint arXiv:1704.04861, 2017.

[2] X. Zhang, X. Zhou, M. Lin, and J. Sun, “Shufflenet: An extremely efficient convolutional neural network for mobile devices,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 6848–6856.

[3] R. Girshick, “Fast r-cnn,” in Proceedings of the IEEE international conference on computer vision, 2015, pp. 1440–1448.

[4] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks,” Advances in neural information processing systems, vol. 28, pp. 91–99, 2015.
Fig. 8: The timeline comparison of different heuristics within one iteration. It is obvious that $h_{LRU}^{\text{filter}}$ incurs frequent tensor offloading compared with $h_{base}^{\text{filter}}$ and $h_{GREEDY}^{\text{filter}}$.

[5] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[6] J. Zhou, C. Ma, D. Long, G. Xu, N. Ding, H. Zhang, P. Xie, and G. Liu, “Hierarchy-aware global model for hierarchical text classification,” in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020, pp. 1106–1117.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in Advances in neural information processing systems, 2017, pp. 5998–6008.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” arXiv preprint arXiv:1810.04805, 2018.

[9] W. Fedus, B. Zoph, and N. Shazeer, “Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity,” arXiv preprint arXiv:2101.03961, 2021.

[10] 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.

[11] G. Amir, Y. Zhewei, K. Schoon, M. M. W, and K. Kurt, “AI and memory wall,” RiseLab Medium Post, 2021.

[12] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, “Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning,” arXiv preprint arXiv:2104.07857, 2021.

[13] 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.

[14] T. Beri, S. Bansal, and S. Kumar, “The unicorn runtime: Efficient distributed shared memory programming for hybrid cpu-gpu clusters,” IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 5, pp. 1518–1534, 2016.

[15] L. Liu, S. Yang, L. Peng, and X. Li, “Hierarchical hybrid memory management in os for tiered memory systems,” IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 10, pp. 2223–2236, 2019.

[16] P. Ghosh, S. Krishnamoorthy, and A. Kalyanaraman, “Pakman: A scalable algorithm for generating genomic contigs on distributed memory machines,” IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 5, pp. 1191–1209, 2020.

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

[18] M. Kirisame, S. Lyubomirsky, A. Haan, J. Brennan, M. He, J. Roesch, T. Chen, and Z. Tatlock, “Dynamic tensor rematerialization,” arXiv preprint arXiv:2006.09616, 2020.

[19] P. Jain, A. Jain, A. Nrusintha, A. Gholami, P. Abbeel, K. Keutzer, I. Stoica, and J. E. Gonzalez, “Checkmate: Breaking the memory wall with optimal tensor rematerialization,” arXiv preprint arXiv:1910.02653, 2019.

[20] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, “Zero-offload: Democratizing billion-scale model training,” arXiv preprint arXiv:2101.06840, 2021.

[21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., “Tensorflow: A system for large-scale machine
learning,” in 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), 2016, pp. 265–283.

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., “PyTorch: An imperative style, high-performance deep learning library,” Advances in neural information processing systems, vol. 32, pp. 8026–8037, 2019.

[23] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems,” arXiv preprint arXiv:1512.01274, 2015.

[24] 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,” in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 891–905.

[25] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, “Superneurons: Dynamic gpu memory management for training deep neural networks,” in Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming, 2018, pp. 41–53.

[26] O. Beaumont, L. Eyraud-Dubois, and A. Shilova, “Efficient combination of rematerialization and offloading for training dnns,” Advances in Neural Information Processing Systems, vol. 34, 2021.

[27] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, “Inception-v4, inception-resnet and the impact of residual connections on learning,” in Thirty-first AAAI conference on artificial intelligence, 2017.

[28] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[29] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in International Conference on Medical image computing and computer-assisted intervention. Springer, 2015, pp. 234–241.

[30] 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.

[31] S. R. Bulo, L. Porzi, and P. Kontschieder, “In-place activated batchnorm for memory-optimized training of dnns,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 5639–5647.

[32] J. Schmidt-Hieber, “Nonparametric regression using deep neural networks with relu activation function,” The Annals of Statistics, vol. 48, no. 4, pp. 1875–1897, 2020.

[33] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in International conference on machine learning. PMLR, 2015, pp. 448–456.

[34] B. Pudipeddi, M. Mesmakhosroshahi, J. Xi, and S. Bharadwaj, “Training large neural networks with constant memory using a new execution algorithm,” arXiv preprint arXiv:2002.05645, 2020.

[35] M. Courbariaux, Y. Bengio, and J.-P. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” in Advances in neural information processing systems, 2015, pp. 3123–3131.

[36] Y. Gong, L. Liu, M. Yang, and L. Bourdev, “Compressing deep convolutional networks using vector quantization,” arXiv preprint arXiv:1412.6115, 2014.

[37] 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.

[38] S. Han, J. Pool, J. Tran, and W. J. Dally, “Learning both weights and connections for efficient neural network,” in NIPS, 2015.

[39] M. Rhu, M. O’Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, “Compressing dma engine: Leveraging activation sparsity for training deep neural networks,” in 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2018, pp. 78–91.

[40] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in International conference on machine learning. PMLR, 2015, pp. 1737–1746.

[41] 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.

[42] A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse, “The reversible residual network: Backpropagation without storing activations,” Advances in neural information processing systems, vol. 30, 2017.

[43] J.-H. Jacobsen, A. Smeulders, and E. Oyallon, “i-revnet: Deep invertible networks,” in ICLR 2018-International Conference on Learning Representations, 2018.

[44] R. Kumar, M. Purohit, Z. Svitkina, E. Vee, and J. Wang, “Efficient rematerialization for deep networks,” Advances in Neural Information Processing Systems, vol. 32, 2019.

[45] S. Cook, CUDA programming: a developer’s guide to parallel computing with GPUs. Newnes, 2012.

[46] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” arXiv preprint arXiv:1412.6980, 2014.

[47] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in 2009 IEEE conference on computer vision and pattern recognition. Ieee, 2009, pp. 248–255.

[48] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, “Aggregated residual transformations for deep neural networks,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2017, pp. 1492–1500.

[49] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, “Squad: 100,000+ questions for machine comprehension of text,” in Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, 2016, pp. 2383–2392.

[50] M. Leinhauser, J. Young, S. Bastrakov, R. Widera,
R. Chatterjee, and S. Chandrasekaran, “Performance analysis of picongpu: Particle-in-cell on gpus using nvidia’s nsight systems and nsight compute,” Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), Tech. Rep., 2021.

[51] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” Advances in neural information processing systems, vol. 25, pp. 1097–1105, 2012.