MemFerry: A Fast and Memory Efficient Offload Training Framework with Hybrid GPU Computation

Zhiyi Yao[1], Zuning Liang[1], Yuedong Xu[1], Jin Zhao[1], Jessie Hui Wang[2], Tong Li[3]
[1]Fudan University, [2]Tsinghua University, [3]Renmin University of China

***Abstract*—With the ever-growing size of deep learning models, GPU memory is prone to be insufficient during training. A prominent approach is ZeRO-Offload which moves the optimizer states to CPU memory and performs parameter update using CPU. However, the deficiencies of ZeRO-Offload include low GPU utilization, imperfect overlapping of communication and computation, and inflexible offloading. In this paper, we leverage Direct Host Access (DHA) in GPU that can compute data on CPU memory to form a novel hybrid on-GPU and DHA. We design and implement MemFerry consisting of an execution scheduler and a shadow model. The scheduler strategically chooses layers of parameters for DHA computation and transmits the remaining parameters to GPU memory simultaneously to shorten forward propagation time, and further loads DHA parameters to GPU memory for reducing backward propagation time. The shadow model presents a unified memory abstraction for the parameter partitions stored separately in GPU and CPU memories. To further reduce GPU memory usage, we present GO-MemFerry along with its dynamic programming algorithm that offloads gradients to CPU memory via DHA. Our experiments show that MemFerry trains up to $1.68\times$ faster and GO-MemFerry could train $1.52\times$ larger model compared to ZeRO-Offload on a single GPU, and increase training speed by at least 28.1% when scaling to data parallelism on 8 GPUs.**

I. Introduction

With the penetration of large-scale deep learning such as ChatGPT (GPT-3.5 and GPT-4) [1] in our daily life and production activities, how to train deep learning models efficiently has attracted great interest. In recent years, the size of deep learning models has increased rapidly. For instance, BERT-base [2] developed in 2018 contains 110 million parameters, while one year after GPT-2 [3] contains 15 Billion parameters, and today's LLaMA [4] and GPT-3 [5] contain 70 Billion and 175 Billion parameters, respectively. However, GPU as the mainstream computing accelerator has very limited device memory. The latest Nvidia A100, H100 and H800 have $80G$ Bytes of high-bandwidth memory (HBM), and the frequently used V100 has only $32G$ Bytes of memory. This imposes a significant challenge on large-scale model training.

To reduce GPU memory requirements, two key technical routines have surfaced. One is to segment the parameters, gradients, optimizer states and activation (also called training content) on multiple GPUs such as ZeRO [6] for data parallelism (DP) and GPipe [7] for pipeline parallelism. The other is offload training [8]–[11]. The basic idea is to move partial training content from GPU memory to CPU memory [12] that is usually an order of magnitude larger, or even to secondary storages [13], [14]. However, hosting large models is achieved at the cost of reduced training speed. The most widely used offload training system is ZeRO-Offload [11]. In ZeRO-Offload, the optimizer states are placed in CPU memory and the parameter update is performed by CPU, where the regular forward propagation (FP) and backward propagation (BP) are performed in GPU and its memory. Therefore, in each iteration, the updated parameter will be transmitted to GPU memory for FP and the gradients are returned to CPU memory for update. The limited PCIe bandwidth between GPU and CPU throttles the effective GPU utilization.

In this paper, we conduct a preliminary evaluation on the DeepSpeed implementation of ZeRO-Offload with three key observations. (1) GPU streaming processors (SMs) are severely under-utilized compared to non-offloading (e.g. 63% vs 99% in BERT-base on Nvidia A100). (2) Communication and computation are not well overlapped. The GPU idleness becomes prominent when the parameters are enormous and unbalancedly distributed on different layers. (3) The memory offloading is not flexible, especially that no further training content can be moved to CPU memory. Hence, we raise a critical question: "Does there exist a new or improved training paradigm that can achieve higher training speed over ZeRO-Offload and reduce GPU memory usage at the same time?"

We propose to integrate the Direct-Host-Access (DHA) functionality provided by mainstream GPUs in offload training. With DHA, GPU SMs access data in CPU memory, bypassing CPU cores and GPU memory. DeepPlan [15] is the first work to introduce DHA into model *inference*. Inspired by this, we evaluate the FP and BP times under the on-GPU and DHA execution modes of offload training. In FP, DHA outperforms on-GPU on certain large layers considerably. In BP, DHA is inferior to on-GPUs if the resulting gradients are stored in GPU memory, but their gaps are greatly reduced if the gradient transmission times of on-GPU are considered. Therefore, fusing on-GPU and DHA into a novel hybrid training paradigm has the potential to achieve our objectives.

We design MemFerry, a novel hybrid training framework that exploits the benefit of using DHA in FP and meantime

This work was supported in part by the Natural Science Foundation of China under Grant 62072117; in part by the Natural Science Foundation of China under Grant 62202473; in part by the Shanghai Natural Science Foundation under Grant 22ZR1407000; in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B010166003; in part by the Huawei project on New Network Transport Layer; Yuedong Xu is the corresponding author. Emails: zyyao23@m.fudan.edu.cn, ydxu@fudan.edu.cn

mitigates its side effect in BP as much as possible. MemFerry consists of an *execution scheduler* and a *shadow model*. The execution scheduler leverages DHA to compute certain layers of parameters while loading the remainder to GPU memory in FP for overlapping communication and computation, and such decisions are made for each layer. In the period between the end of parameter loading and the beginning of gradient transmission, MemFerry loads the DHA parameters from CPU memory to GPU memory that can reduce the BP time most. After mitigating bubbles in both FP and BP, the per-iteration time is reduced consequently. The *shadow model* is a unified memory abstraction that consists of three parameter partitions spanning CPU and GPU memories: the DHA parameters never entering GPU memory, the GFGB (GPU-FP-GPU-BP) parameters computed in GPU memory, and the DFGB (DHA-FP-GPU-BP) parameters that are computed via DHA but shipped to GPU memory for more efficient BP operations. By utilizing dynamic address mapping, GPU computes the data in the shadow model without considering its physical space and without additional CPU data copy.

To achieve flexible GPU memory offloading, we propose GO-MemFerry that uses DHA to write gradients directly to CPU memory, thus further reducing GPU memory usage. To the best of our knowledge, the gradient offloading by DHA has not been reported before. With the user-specified gradient reserving factor, we model the optimal gradient offloading as a dynamic programming problem so as to minimize its impact on the training speed. With GO-MemFerry, we are eligible to train even larger models and balance the tradeoff between fast training speed and low GPU memory usage.

The evaluation results on clusters consisting of Nvidia V100 and A100 show that MemFerry has faster training speed and lower memory usage compared to the SOTA system during offload training. MemFerry improves the training speed by $1.32\times$ on average and up to $1.68\times$. The GPU utilization of MemFerry and GO-MemFerry is higher than the baseline by up to $1.27\times$. At the same time, MemFerry reduces training memory usage by up to 61.7%. GO-MemFerry reduces the memory usage by 20.1% on average and up to 74.4%. With GO-MemFerry, we could train $1.52\times$ larger models compared to the baseline system. Furthermore, MemFerry is scalable and compatible with distributed training, with at least $1.28\times$ speedup when we integrate it into data parallelism.

II. Background

A. Memory Usage of DNN Training

We first investigate the memory usage of standard DL training on the premise of sufficient GPU High Bandwidth Memory (HBM). All the involved data is usually in the form of tensors. The data that needs to be maintained on GPU memory (interchangeable with GPU HBM) during training includes: 1) **Parameters.** Before the training starts, the parameters of a model need to be loaded into GPU memory. This memory usage is exactly the model size. 2) **Gradients.** During the backward propagation (BP) stage, the DL framework will generate gradient tensors of the same size as the parameters

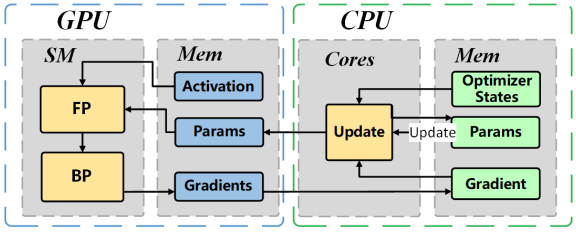


Fig. 1. Training process of ZeRO-Offload.

on the GPU memory. 3) **Optimizer States.** Optimization algorithms in DL usually rely on first-order and second-order momentum to accelerate convergence. For example, the optimizer states of Adam [16] include momentum and variance, each of which is two times the model size. 4) **Activations.** The activations include the intermediate computing results of each model layer and the loss during BP. If the model input is complex, especially in large language models, activations will occupy a large portion of GPU memory. Fortunately, the recomputation [17] technique discards intermediate activations when the computation completes so that only the peak size of activations is counted.

B. ZeRO-Offload Training Paradigm

To reduce GPU memory requirement, ZeRO-Offload [11] presents a new training framework that uses CPU memory to supplement the limited GPU memory. The core idea of ZeRO-offload is to offload the expensive optimizer states and corresponding update computation to CPU. ZeRO-Offload is officially implemented in DeepSpeed [18], and its training process is shown in Figure 1.

Before the training starts, the training system stores model parameters and optimizer states in CPU memory, and allocates space for computation in GPU memory. At the beginning of each iteration, CPU explicitly copies the latest model parameters from CPU memory to GPU memory and then performs forward and backward computations on GPU. During the BP process, the gradients will be asynchronously transferred to the CPU memory for model update. Once the BP process is finished, the optimizer updates the parameters in the CPU. In this way, the GPU memory consumption of model training is reduced to the size of activation, parameters, and gradients.

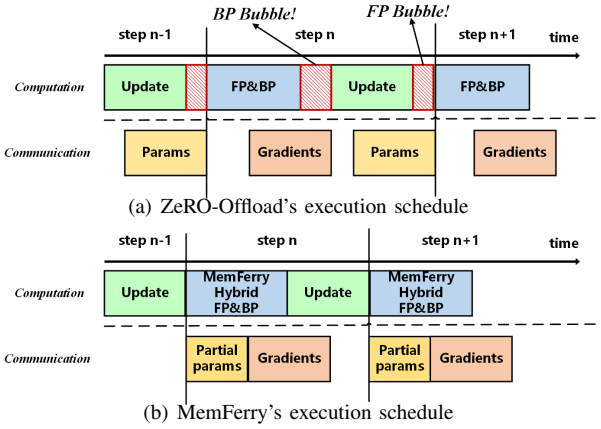
III. Motivation

In this section, we first present key observations on GPU offloaded training and Direct-Host-Access (DHA). Then we state our research problem and technical challenges.

A. Limitations of Offload Training

We evaluate the performance of GPU offload training in DeepSpeed, a widely-used distributed DL framework [19], and observe three key limitations that motivate our study.

Low GPU utilization. In ZeRO-Offload, GPU does not perform computation at every moment. Instead, it resides in the *idle* state for a period of time at each iteration. Take the training of GPT-2 as an example, We find that the idle state occupies around one-third of the GPU runtime, confining the average GPU utilization to be merely 63.1%. In contrast,



(a) ZeRO-Offload's execution schedule

(b) MemFerry's execution schedule

Fig. 2. Training schedule: ZeRO-Offload vs MemFerry.

when the training is carried out in GPU completely, the GPU utilization is always higher than 80% in every iteration. The huge utilization gap comes from the data shipment of ZeRO-Offload, where the GPU utilization is near zero. This indicates that exchanging data between GPU and CPU is quite slow, leading to the waste of expensive GPU resources.

Insufficient communication overlapping. We delve into the execution schedule of ZeRO-Offload to understand the root cause of GPU under-utilization. ZeRO-Offload leverages the overlapping of communication and computation to reduce the offloading overhead. As shown in Figure 2(a), the backward propagation (labeled as 'BP') at GPU partially overlaps the gradient synchronization from GPU to CPU (labeled as 'Gradients'), and the parameter update (labeled as 'Update') partially overlaps the parameter synchronization from CPU to GPU (labeled as 'Params'). However, such overlapping is imperfect with two bubbles observed in each iteration. Owing to the layer-wise execution of FP and BP, the communication of a layer always takes place after the completion of its computation, generating bubbles in the figure. Even worse, during BP, the final computed layer is often the embedding layer in large language models, which have large model sizes and long transmission latency.

Inflexible GPU offload. The offloading policy of ZeRO-Offload is inelastic, i.e. a fixed proportion of training contents is moved to CPU memory. However, the GPU memory budget needs to be utilized as much as possible to improve the training throughput. There are contents such as gradients that do not have to be stored in the GPU memory in its entirety, an offload training system could offload them when the GPU memory budget is small. Therefore, memory offloading needs to be flexible, catering to versatile memory sizes.

B. GPU Computation on CPU Memory

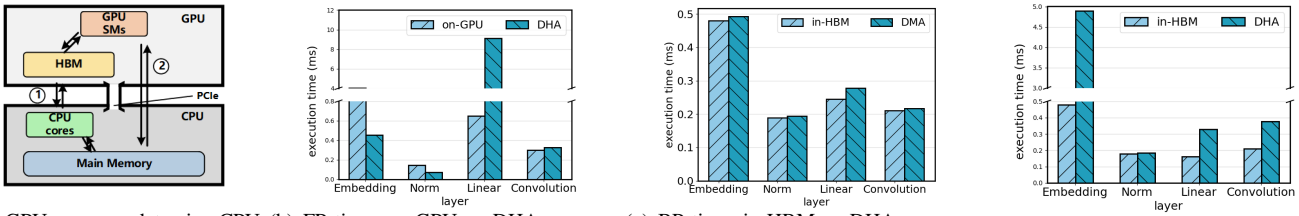
Commodity GPU streaming multiprocessors (SMs) possess two computing approaches with the data stored in CPU memory [20]. In Figure 3(a), we illustrate these two approaches with Nvidia GPU as an example. ① **on-GPU**: the data is ferried from CPU memory to GPU memory (i.e. GPU HBM) before it is computed by GPU SMs. The advantage of on-GPU is the high bandwidth between GPU memory and SMs, while the cost of CPU memory to GPU memory transmission is non-trivial. ② **Direct Host Access (DHA)**: The *CudaHostAlloc* function is called to allocate the space in CPU memory and register it to the GPU. In this way, the GPU SMs can access data in the registered space in the CPU similar to that in GPU memory, without bypassing the GPU memory. Naturally, the computation via DHA is slower compared with on-GPU. However, the DHA computation streams and the CPU-GPU data transfer streams in GPU SM are independent, so they can be executed in parallel.

Existing ML frameworks such as Pytorch [19] do not support DHA-enabled training. Even though ZeRO-Offload partitions the data in both CPU and GPU memories, the FP and BP operations are entirely processed via the on-GPU mode. DeepPlan [15] introduces DHA computation into model *inference*. With DHA, GPU can load several layers of model parameters from CPU memory to GPU memory, and perform FP computation on other layers in CPU memory in parallel. To explore the potential of DHA in modeling training, we profile the efficiency of both on-GPU and DHA approaches using *Bert-base* as an example. The results below are measured from a machine with an Nvidia A100 and 8 CPU cores interconnected via $100Gbps$ PCIe switch. We next show the training performance when we offload part of the training contents (e.g. parameters and gradients) to CPU memory and enforce GPU to operate them via DHA.

DHA on CPU parameters in FP. We first measure the FP times of on-GPU and DHA execution for the parameters stored in CPU memory in Figure 3(b). Note that both the transmission time and the computation time are included in the FP time of on-GPU, while they are inseparable in that of DHA. For the *Embedding* and *LayerNorm* layers, DHA exhibits much shorter FP time than on-GPU. This is because the transmission time of these layers is longer than the computation time, and the on-GPU execution is blocked by the transmission. In contrast, the FP times of the *Linear* and *Convolution* layers are longer in DHA than in on-GPU. Obviously, for these layers with high computational loads but small in size, they are more suitable to be executed via the on-GPU approach.

DHA on CPU parameters in BP. If FP uses on-GPU approach (load parameters from CPU memory), then the model parameters are already in GPU HBM during the BP process. Then, GPU is able to directly use the in-HBM parameters to perform BP. If DHA is used to perform FP, the parameters are still in CPU memory which means GPU still needs to use DHA to perform BP. In Figure 3(c), we show the BP time of these two approaches. Overall, DHA is slower than on-GPU. While compared to FP, the performance degradation of DHA BP is not severe $(5-10\%)$. This is because deep learning frameworks have activation retention mechanisms. However, completely ignoring the performance degradation of DHA BP can also lead to poor training efficiency.

DHA on CPU gradients in BP. During the traditional BP process of offload training, gradients will be generated on the GPU HBM and then transmitted to CPU memory. However, we can make use of DHA to generate gradients directly



(a) GPU access data in CPU memory (b) FP time: on-GPU vs DHA parameters (c) BP time: in-HBM vs DHA parameters (d) BP time: in-HBM vs DHA gradients

Fig. 3. DHA and on-GPU execution.

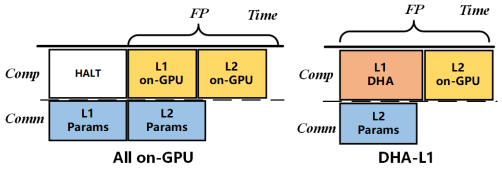


Fig. 4. An example of hybrid execution.

on CPU memory (leveraging DHA write). So we propose BP integrating DHA gradients as an additional BP execution approach. Figure 3(d) shows the measurement of BP time with DHA gradients. Compared to in-HBM BP which directly generates gradients in GPU HBM, BP with DHA gradients executes slower. The reason is that the former's SM can write gradients to HBM with high bandwidth, but the latter needs to write gradients to CPU memory through the low-bandwidth PCIe switch. However, DHA gradients do not need any GPU HBM buffer to save gradients, which means the GPU memory requirements for BP are significantly reduced.

C. An Ideal Execution Schedule

The above observations manifest that DHA enriches the FP and BP execution approaches so as to possibly reduce the data ferry time between the CPU memory and the GPU memory. Here, we elaborate on an "ideal" scheduling policy for offloaded training (in Figure 2(b)) and show how to utilize the advantages of the hybrid on-GPU and DHA.

To minimize GPU idleness, the execution schedule could leverage hybrid DHA and on-GPU computation. Recall in Figure 2(a), two bubbles exist in each iteration of ZeRO-Offload. The FP bubble is caused by the layer-by-layer parameter transmission from CPU memory to GPU memory. Hence, we can simultaneously load the parameters of some layers to GPU memory and perform FP computation on other layers in CPU memory via DHA. Figure 4 compares the execution modes of a two-layer model. When the two layers are both performed via on-GPU (labeled as "All on-GPU"), there is a long computation halt of waiting for parameters transmission. But when layer-1 FP is conducted via DHA and layer-2 is on-GPU, the transmission of parameters is completely hidden and computation starts with no startup latency. In Figure 2(b), one can see that "Partial params" transmission executes in parallel with "Hybrid FP&BP".

The BP bubble (in Figure 2(a)) appears in the gradient transmission between BP and update. The gradient is always generated after BP even if we use a fine-grained schedule. Hence, unlike hybrid execution in FP, we cannot transmit the gradient of a layer before it is generated. However, if gradient transmission could be carefully partitioned, the gradient transmission could perform with parameters update in parallel. This is reflected in Figure 2(b) that the "gradients" transmission is partly execution parallel with "BP" and "update".

Achieving such an ideal runtime scheduler with flexible GPU memory offloading has three major challenges.

- **Complex training decisions in hybrid training.** Different layers of DL models have different affinities for the execution approaches. We need to design a fine-grained execution schedule leveraging the advantages of hybrid execution training to minimize training time.
- **Unified management of GPU and CPU memory.** Existing frameworks lack efficient storage support to manage normal and DHA training contents on GPU memory and CPU memory during training. If we blindly copy data even under an optimal schedule, the additional overhead may offset the performance gain.
- **Flexible GPU memory offloading.** The current offloading training designates fixed contents to be offloaded to CPU memory, which is not flexible and makes peak GPU memory usage unchangeable. Further offloading data to CPU memory with a gentle sacrifice of training speed remains a challenge.

IV. System Design

In this section, we present an overview of MemFerry with three key aspects, including hybrid training modes, efficient memory management and flexible offloading.

A. System Overview

We briefly describe our technical measures toward the key challenges raised in Section III, followed by the workflow.

1) Scheduler for hybrid training. (Section IV-B) We design a new offloading execution scheduling with hybrid training that overlaps parameter loading time and gradient transmission time with DHA computation. Our schedule proposes three execution modes, GFGB, DHA and DFGB that are automatically chosen for different layers.

2) Runtime shared memory management. (Section IV-C) We propose "shadow model", a logical model on GPU where its underlying storage space spans both GPU memory and CPU DHA memory. The shadow model allows all the tensors to seamlessly switch the underlying storage address between DHA memory and GPU memory, avoiding unnecessary data copy at runtime.

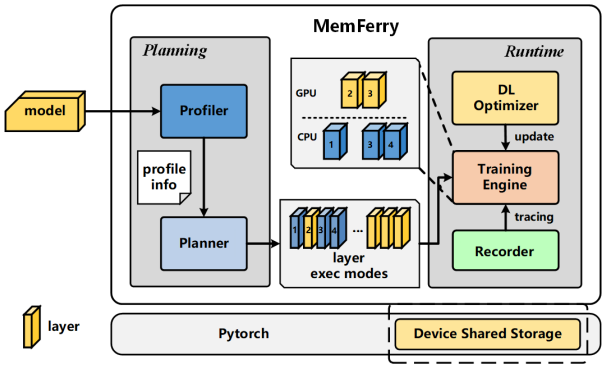


Fig. 5. Overview of MemFerry System.

3) On-demand offloading of GPU memory. (Section IV-D) With DHA, not only the parameter but also the gradient can be offloaded to CPU memory. The gradient offloading could further reduce GPU memory usage at the cost of the gentle increase in per-iteration training time. This means that an elastic choice in pursuit of fast training speed or low GPU memory usage is allowed.

Workflow Figure 5 shows the overall architecture of MemFerry, including the static planning components (*Profiler* and *Planner*), runtime memory management and execution logic components (*Training Engine*, *DL Optimizer*, *Recorder*). When a new model training configuration initiates, MemFerry first passes the training configuration into *Profiler*. The *Profiler* uses different training configurations to evaluate the intra/inter-layer(s) computation times and the data transmission times. Then the *Planner* calls the scheduling algorithm (See in Section IV-B) based on the profiled information to determine the specific execution mode for each layer.

Once the planning procedure finishes, MemFerry Runtime modules start the model training. The *Training Engine* is responsible for memory management and data transmission. At the beginning of training, it distributes parameters to the corresponding devices and constructs logical and underlying storage models (including *shadow model*). During the training process, it schedules the execution to behave as planned. The *DL Optimizer* performs parallel partial gradient transmission support (DHA parameters) and parameter update by user-specific training optimizer.

B. Hybrid Training Schedule

We hereby propose the layer-level scheduling algorithm of MemFerry's scheduler for general foundation models. The design goal is to overlap parameter transmission with DHA computation as much as possible in FP and mitigate the adverse impact of enabling DHA in BP. We create three execution modes for the parameter of each layer: **GFGB** (GPU Forward GPU Backward), **DHA**, and **DFGB** (GPU DHA Forward GPU Backward). Let us take a layer of parameters as a unit. In GFGB, the parameters are transmitted from CPU memory to GPU memory before computation; in DHA, the parameters in CPU memory are computed by GPU SMs directly and no further transmission happens; in DFGB, the layer is transmitted to GPU memory after its DHA FP execution.

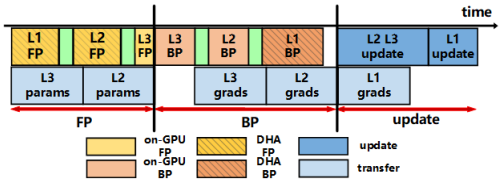


Fig. 6. Hybrid training schedule of MemFerry.

DFGB helps to reduce BP time because the on-GPU BP is faster than the DHA BP (see in Section III-B) so it can be regarded as a remedy of DHA for BP.

Fig.6 illustrates a training schedule example for a three-layer model. We stipulate the execution modes of 1-3 layers to be DHA, DFGB and GFGB respectively. Due to DHA computation without any transmission, layer 1 and 2 start FP computation at the iteration beginning. At the same time, the parameter transmission of layer 3 also starts. When transmission of layer 3 is completed, MemFerry load parameter of layer 2 from CPU to GPU for on-GPU BP, which avoids prolonging the BP time of layer 2. As we hide the gradient transmission with both BP and update, update does not wait for all the gradients to be transmitted.

To realize the above schedule, our first step is to profile the training information for MemFerry's decision-making. During the profiling process, we evaluate the parameter transfer time, FP and BP computation time for each fine-grained layer. The profiling is executed only once, thus bringing negligible overhead. We summarize the profiled variable and their notations below. Here, L: the number of layers; l: the layer id of model; $A_{l,li+1}$: computation between layer l and l+1; $T_{trans}(l)$: parameter transfer time of layer l; $T_{FP}(l)$: on GPU forward time of layer l; $T_{BP}(l)$: on GPU backward time of layer l; $T^H_{FP}(l)$: DHA forward time of layer l; $T^H_{BP}(l)$: DHA backward time of layer l; $T_{FP}(A_{l,l+1})$: forward computation time $A_{l,l+1}$; $T_{BP}(A_{l,l+1})$: backward computation time of $A_{l,l+1}$; S_l: size of layer l.

With the profiled information, we introduce our scheduling algorithm. We logically separate the entire training algorithm into the forward and backward algorithms.

Forward Algorithm (Alg. 1). Our goal is to minimize the overall execution time and hide transmission time as much as possible in computation. The execution plan is denoted as P and the overall execution time as T_{all}. We notice that computation can provide some overlapping time for communication. This overlapping time is called the reserved overlap time and denoted as T_{res}. From the previous observations, we learn that loading time is often larger than computation time. Based on this, we make a simple criterion for judging each layer's execution mode. When the DHA computation time is less than on-GPU computation with loading time, using DHA can reserve more overlapping time for other layers to load parameters (i.e. T_{res} increases) and reduce execution time. On the contrary, using GFGB mode is intuitively better.

As the above intuitive methods do not guarantee the optimality, we design a backtracking function. A layer that is decided to execute using GFGB mode may have two cases.

Algorithm 1: MemFerry FP Algorithm

Input: $T_{trans},\ T_{FP},\ T^H_{FP},\ T_{FP}(A)$
Output: P

$T_{all},\ T_{res},\ tag \leftarrow 0,\ 0,\ -1$;
for $l \leftarrow 1$ **to** L **do**
  if $T^H_{FP}(l) \le T_{FP}(l) + \max(T_{trans}(l), 0)$ **then**
    $P_l \leftarrow DHA$;
    $update(T^H_{FP}(l), T^H_{FP}(l))$;
  else
    if $T_{trans}(l) < T_{res}$ **then**
      if $tag = -1$ **then** $tag \leftarrow l$;
      $update(T_{FP}(l), T_{FP}(l) - T_{trans})$;
    else
      $backtrack(l, tag, T_{all}, T_{res})$;
    end
    $P_l \leftarrow GFGB$;
  end
  $update(T_{FP}(A_{l,l+1}), T_{FP}(A_{l,l+1}))$;
end
return $P,\ T_{res},\ tag$;

Function `backtrack` $(l, bt, T^b_{all}, T^b_{res})$ **:**
  $base \leftarrow T^b_{all} + T_{trans}(l) + T_{FP}(l) - T^b_{res}$;
  for $i \leftarrow bt$ **to** l **do**
    $gain \leftarrow T_{trans}(i) + T^H_{FP}(i) - T_{FP}(i)$;
    $T^i_{all} \leftarrow \max(T_{trans}(l) - T^b_{res} - gain, 0) + T^b_{all} + (T^H_{FP}(i) - T_{FP}(i)) + T_{FP}(l)$;
    if $T^i_{all} < baseline$ **then**
      $P_i,\ tag \leftarrow DHA,\ i+1$;
      $base,\ T^b_{res} \leftarrow T^i_{all},\ T^b_{res} + gain$;
      if $T^b_{res} \ge T_{trans}(l)$ **then break** ;
    end
  end
  $update(base, \max(T^b_{res} - T_{trans}(l), 0) + T_{FP}(l))$;

Function `update` (t_o, t_r) **:**
  $T_{all}, T_{res} \leftarrow T_{all} + t_o,\ T_{res} + t_r$;

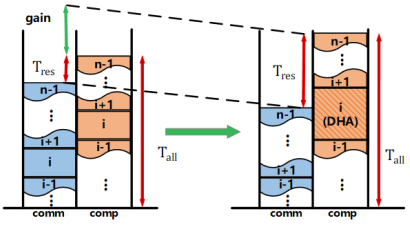


Fig. 7. Backtrack to turn execution mode of layer i to DHA.

When the reserve time T_{res} is greater than its loading time T_{trans}, it has sufficient time to load its parameters and then compute. When T_{res} is insufficient to cover its loading time, we backtrack the previous GFGB layers and modify their execution mode to DHA mode, which provides more overlap time for the layer to load parameters.

Figure 7 shows the effect on T_{res} and T_{all} when we modify a GFGB mode layer i to DHA mode. After execution mode changing, T_{res}, T_{all} will increase and we labeled them T^i_{res}, T^i_{all}. As in the figure, the gain of T_{res} could be computed as $T^i_{res} - T_{res} = T_{all} + T^H_{FP}(i) - T^H_{FP}(i) + T_{trans}(i)$. In the backtracking process for a new GFGB layer l, after changing the execution mode of a previous layer i, if the $T_{trans}(i) - T^i_{res} - T_{res}$ is larger than $T^i_{all} - T_{all}$, then the mode change of layer i successfully hides more parameter load time of layer l without prolonging overall execution time.

Backward Algorithm. The PCIe communication is idle for a certain period between the end of parameter loading from CPU memory and the beginning of gradient transmission back to CPU memory. To fully utilize this idleness to reduce the negative impact of DHA computing, we present the DFGB execution mode that loads several layers of parameters (after FP) still in CPU memory into GPU memory. For these layers, their BP can be performed on GPU so as to reduce the time of obtaining the gradients. We design a greedy algorithm to decide whether the layers are executed in DFGB mode.

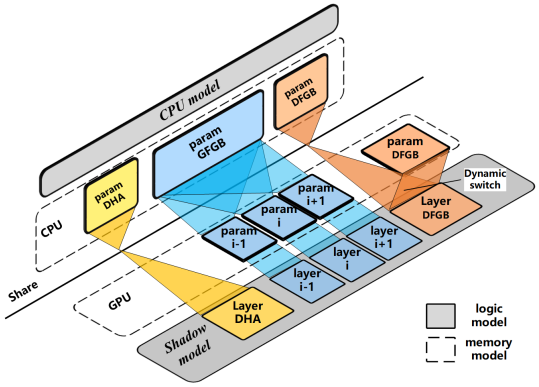


Fig. 8. The shadow memory management in MemFerry.

C. Memory Management

The hybrid training requires MemFerry to be capable of computing data stored in both the GPU memory and the CPU memory. This signifies that MemFerry will stipulate the storage space for different partitions of the model and their respective access patterns. We then design an abstract model named *shadow model* with a set of merits: *shared memory of GPU and CPU*, *minimal memory copy* and *compatible to ML frameworks*. We next explain where the model partitions are stored and how they interact with the shadow model.

We firstly separate the models of the logical level and the underlying storage level. In *Training Engine*, a logical CPU model (for updates), a logical GPU model (for FP&BP) and CPU-GPU parameters storage are maintained, as the Figure 8 shows. In the CPU memory, the model parameters are partitioned into three parts corresponding to the three execution modes in Section III-C: 1) GFGB parameters that will be transmitted to the GPU memory; 2) DHA parameters that will be computed in situ by GPU; 3) DFGB parameters that will be computed by GPU first and be transmitted to GPU memory later on. To maximize the parallel computing power of the CPU SIMD, we aggregate the GFGB parameters into a long parameter vector. Besides, the CPU memory also hosts the gradient and the optimizer state just like ZeRO-Offload. The GPU runtime memory stores the GFGB parameters buffer and the DFGB parameters buffer in line with the corresponding parameters on CPU memory.

The logical GPU model used in FP&BP is called the ***shadow model***. MemFerry adopts a dynamic mapping mechanism for the underlying storage of shadow model based on the execution plan. The parameters of the shadow model are not necessarily located in the GPU memory but may also be mapped from the CPU parameters. For example, the parameters executed by the GFGB mode are projected and

transmitted from the long parameter vector to GPU memory; for the parameters executed by DHA mode, MemFerry directly delivers their storage pointers to the shadow model, which could access them directly. As shown in Figure 8, We use three colors to differentiate the execution modes of GFGB (*blue*), DHA (*yellow*) and DFGB (*orange*). The DHA parameters are directly mapped from CPU memory to the shadow model in GPU. The GFGB parameters are coordinated to a large contiguous area in CPU memory, and mapped separately to GPU memory layer by layer when being loaded. The DFGB parameters have both the DHA storage and the GPU memory storage, whose pointer can be switched before BP starts.

The shadow model abstraction brings two advantages: 1) **Reducing space allocation and memory copy.** by unified storage of logical parameters. We directly share some of the DHA parameters of the CPU model with the GPU model, avoiding redundancy and copying in the memory buffer. 2) **Facilitating dynamic storage switching in DFGB.** MemFerry dynamically modifies the mapping of the physical storage of the DFGB layer after FP and change it back after BP without additional burden or affecting the learning procedure.

D. Gradient Offloading

A crucial question regarding DHA-enabled training is whether the GPU memory requirement can be further relieved. With the ever-growing model size, MemFerry may still run out of GPU memory. To achieve the flexible usage of GPU memory and avoid violating the memory limit, we present GO-MemFerry based on the aforementioned scheduling algorithms and observations.

The GPU content of MemFerry includes the partial model parameters, the gradients generated by BP and intermediate activation. Further offloading model parameters will significantly decrease the training speed of layers that are not suitable for DHA. Offloading intermediate activation values involves a large number of reads and writes between GPU and CPU through slow PCIe. Therefore, GO-MemFerry chooses to offload the gradients. Specifically, GO-MemFerry designates a CPU memory as DHA memory, mapping it to the gradient storage of the shadow model. During the BP process, GO-MemFerry will directly generate the gradient in the CPU memory, avoiding the transferring and copying of GPU memory.

However, we do not expect gradient offloading to bring a significant side effect to training speed, and the amount of memory offloaded should be determined based on the needs of users. GO-MemFerry does not completely offload the gradients of the entire model but the gradients of partial parameters while meeting the user's memory needs. In Section III-B, we find that different model parameters have different performances on gradient offloading. Based on this phenomenon, GO-MemFerry automatically determines the layers for gradient offload to ensure maximum training speed under the user-specified memory requirement.

Gradient offload algorithm. We design a gradient offload algorithm based on dynamic programming. In this algorithm, users need to specify the total percentage S_{req} of gradient on GPU (named *reserve factor*), and the algorithm outputs the specific layers to offload as the gradient offload plan P^{GO}. We denote the normal on-GPU BP time for each layer as T_{CBP}, while the BP time after offloading the gradient by DHA is T_{GBP}. We default to offload gradients for all layers at the beginning and then judge the gain $T_{GBP}-T_{CBP}$ in time when adjusting a layer's gradient to on-GPU mode. If we define the problem A_l which means the maximum gain offload strategy of layer 1 to l. Then we could find that for a specific layer i, A_i is up to subproblem A_{i-1}. if layer i is choose to keep gradient offloading then $A_i = A_{i-1}$, else if layer i turn gradient to GPU then $A_i = A_{i-1}+T_{GBP}(i)-T_{CBP}(i)$. Based on this optimal substructure, we construct a dynamic programming table $D_{l,s}$ from two dimensions: layer id and space. The second axis of the table represents the size of the space occupied by the current plan, which is used to limit overall space. The content of the table is the execution time gain of the current plan (i.e. A). The recursion formula of D is given by:

$$D_{i,j} = \max(D_{i-1,j},\ D_{i-1,j-S_i} + T_{GBP}(i) - T_{CBP}(i)).$$

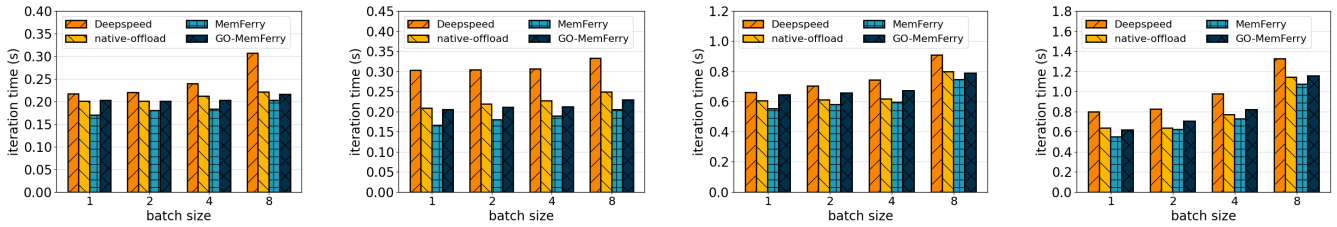
By using dynamic programming to identify the gradient offloading plan that has the least execution time, GO-MemFerry ensures the use of gradient offload meets the memory requirement with the least side effect. This allows users to adjust their memory requirements based on the actual memory size even after offloading some parameters and optimizer states during the various training processes. When the specified gradient is fully offloaded, the GPU memory required for GO-MemFerry is only the size of partial parameters and activations.

V. Performance Evaluation

A. Experimental Setup

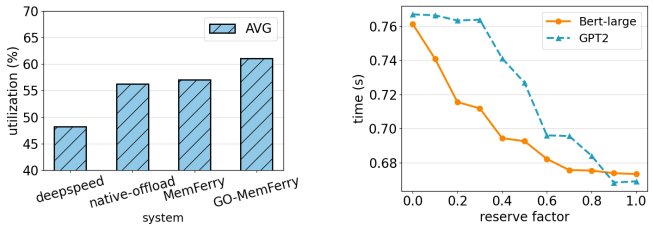
Testbed and DNN Models. Our evaluation is conducted on two cloud instances, one equipped with 128 CPU cores and 8 Nvidia A100 GPUs of $80GB$ memory, the other equipped with 12 CPU cores and an Nvidia V100 GPU of $32GB$ memory. Both of them are equipped with PCIe 3.0. The software we use includes Pytorch-1.9.0, Python-3.6 and CUDA-11.7. The foundation models in our experiments include Bert-base/Bert-large [2], Roberta [21], GPT-2-medium [3] models of transformers [22] library and transformerXL [23]. In transformerXL, we manually configure a set of models of different sizes within the range $0.7B$-$6.7B$ in order to evaluate system performance in a controlled manner. The embedding layer size ranges from 32768 to 130528, the hidden layer size is from 2048 to 4096 and the sequence length is from 378 to 4096.

Baselines and Metrics. Our baseline systems include DeepSpeed implementation of ZeRO-Offload (DeepSpeed), native ZeRO-Offload (Native-Offload), MemFerry and GO-MemFerry. The implementation methods influence the performance of ZeRO-Offload and MemFerry. In MemFerry, we do not destroy the gradients on GPU after they have been used while DeepSpeed does so. This brings considerable latency for DeepSpeed to dynamically construct and deconstruct the gradient buffer. For fair comparison, we implement a native ZeRO-Offload that is faithful to its original design. Note that



(a) Bert-base (b) Roberta (c) Bert-large (d) GPT-2

Fig. 9. Overall iteration time.



(a) Average GPU utilization (b) Reserve factors

Fig. 10. Utilization and reserve factor of GO-MemFerry.

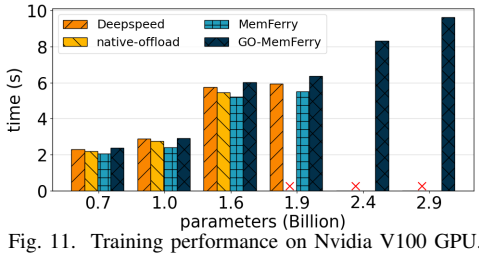


Fig. 11. Training performance on Nvidia V100 GPU.

for GO-MemFerry, the default configuration is to completely offload the gradients via DHA. The performance metrics considered in our evaluation are threefold: *per-iteration time*, *GPU utilization*, and *GPU memory usage*.

B. Training Efficiency of MemFerry

Speedup over various DNNs. We first evaluate the training speed of MemFerry with different large language models in Figure 9. Overall, the per-iteration time is consistently the highest in DeepSpeed, while it is always the shortest in MemFerry. In all the models training we performed, MemFerry has a 10% to 40% improvement in training speed compared to DeepSpeed and Native-Offload. When using the two smaller models (*Bert-base*, *Roberta*), the training speed of MemFerry improves by 21.7% to 68.7% compared to the baselines. As for larger models (*Bert-large*, *GPT-2*), the training speedup is reduced to 7.1% till 32.1%. The main reason is that they usually require large input such that the computation time accounts for a larger proportion of the overall training period.

GPU utilization improvement. The GPU utilization when training *GPT-2* is illustrated in Figure 10(a). MemFerry's GPU utilization is 18.5% higher than DeepSpeed and 1.5% higher than Native-Offload. This utilization improvement originates from MemFerry reducing the GPU idle time during communication and improving the utilization of GPU SMs through parallel DHA computation and transmission. Native-Offload has higher utilization than DeepSpeed because the latter needs time to allocate and reclaim gradient memory. GO-MemFerry has the highest utilization (27% higher than DeepSpeed), as it parallel executes gradient computation and transmission.

C. Training Memory Usage

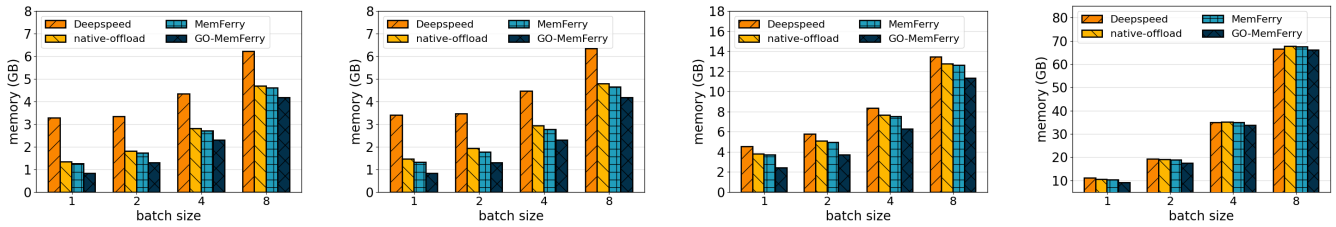
Larger model size limitation. To demonstrate the training performance of MemFerry and GO-MemFerry under limited training resources, we conduct LLM training on the V100 GPU machine. In Figure 11, we configure models of different sizes with a training batch size of 1 MemFerry is still the fastest in all training configurations, 1.12× faster than the baseline. When GO-MemFerry fully offload gradients in this figure, the training speed is the slowest, but only 3.4% lower than the baseline. This is because the lower PCIe bandwidth severely affects the write operations of DHA gradients. As the model size increases, Native-Offload, DeepSpeed, and MemFerry all experience memory overflow. After the model is larger than $2.4B$, only GO-MemFerry can train normally. Overall, GO-MemFerry can support models that are 1.52× larger than DeepSpeed ZeRO-Offload with limited memory.

Memory usage reduction over various DNNs. We then study the *peak* GPU memory usage in an entire training iteration. In Figure 12, MemFerry reduces GPU memory by average 40.1% compared with DeepSpeed when training smaller language models. Compared to baselines, MemFerry uses DHA computing to move some parameters from the GPU to the CPU. GO-MemFerry has a memory reduction of 17.6% even compared to MemFerry, as the gradients are completely removed from the GPU memory. When training *GPT-2*, MemFerry saves the GPU memory by an average of 1.4%. On average, MemFerry and GO-MemFerry reduce 25.4% and 41.6% GPU memory.

Flexibility of GO-MemFerry. While offloading gradients within the user-specified memory size based on the reserve factor, GO-MemFerry minimizes the degradation in the training speed. In Figure 10(b), we measure the iteration time of GO-MemFerry when the gradient reserve factor (larger means keep more gradients on GPU) varies. As the reserve factor increases, iteration time gradually decreases due to less DHA gradient write. The curves are convex, indicating that GO-MemFerry's gradient offload algorithm offloads the gradients with the smallest performance loss to meet users' needs. When the reserve factor increases, GO-MemFerry leans to MemFerry, resulting in a higher training speed.

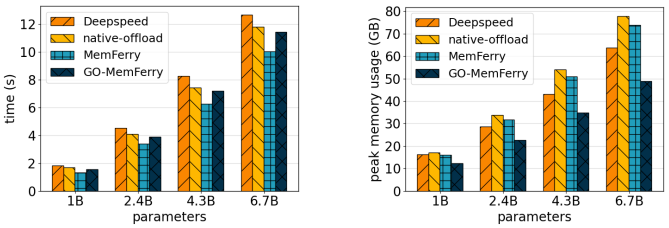
D. Scalability of MemFerry

Scalability to model size. Figure 13(a) shows the training speeds of *TransformerXL* with different parameter sizes where



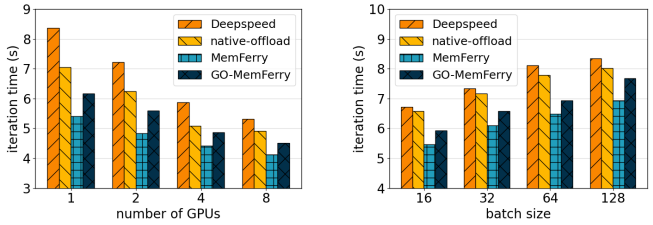
(a) Bert-base (110M) (b) Roberta (110M) (c) Bert-large (340M) (d) GPT-2 (355M)

Fig. 12. Maximum memory usage.



(a) Iteration time (b) Memory usage

Fig. 13. Training on large language models.



(a) Scalability to number of GPUs (batch size = 16) (b) Scalability to batch size (8 GPUs)

Fig. 14. Performance in multi-GPU training.

the batch size is set to 8. MemFerry always possesses the highest training speed, $1.32\times$ faster than DeepSpeed on average. As the model size increases, the speed advantage of MemFerry decreases. When training the $1B$ model, MemFerry has a training speed improvement of $1.37\times$ compared to DeepSpeed, while training the $6.7B$ model the improvement is $1.26\times$. This is because when the model is larger, the computation time during the training process also significantly increases. Figure 13(b) shows the memory usage. Overall, GO-MemFerry uses the least space, with an average reduction of 21.6% compared to DeepSpeed. As the model size increases, the maximum memory savings of GO-MemFerry remain between 18% and 22%. indicating that GO-MemFerry can reduce the memory requirements for large-scale model training.

Scalability to multiple GPUs. MemFerry supports data parallel training that "aggregates" the computing power of multiple GPUs (using Pytorch DataParallel). As shown in Figure 14(a), the iteration time decreases with the number of GPU increases when we train a 4.3B model with a fixed global batch size. Specifically, MemFerry takes 5.41s to complete an iteration on average using a GPU and 4.12s using 8 GPUs. As the training scale increases, inter-GPU communication becomes the new bottleneck that limits the performance of MemFerry. However, MemFerry still ensures a better performance when training with multi-GPUs. For example, when scaling to training with 8 GPUs, MemFerry increases the training speed by 28.1% compared to DeepSpeed and 19.1% compared to Native-Offload.

Scalability to batch size. As training can be conducted with different batch sizes for best performance, we study MemFerry by assigning larger batch sizes to a 6.7B model training in 8 GPUs. As shown in Figure 14(b), we observe that MemFerry always outperforms all the baselines when the batch size increases. Specifically, when the batch size is 128, MemFerry and GO-MemFerry trains faster than DeepSpeed by $1.21\times$ and $1.16\times$ respectively. This observation means that if resources are limited to training models and offloading is used, MemFerry could always improve the training efficiency.

VI. Related Work

Offload Training (or heterogeneous training). To train large-scale models with limited GPU memory [8]–[10], [24]–[28], previous works offloaded the model training content to devices with larger memory (such as CPU and NVMe [13], [14], [29]). L2L [12] offloaded parameters into CPU retrieving them as needed for computational processes during training. MemFerry continues the approach of CPU offload while introducing GPU DHA, into training, providing a new execution schedule and gradient offloading method.

DL Systems with Direct-Access. In model inference, DeepPlan [15] leveraged DHA to serve better inference, which achieves layer-wise communication overlap with computation. In deep learning training, systems [30], [31] leveraged DMA [20], a GPU feature for unified cross-device data management, to reduce data copying and achieve computation-communication overlap for specific models (e.g. CNNs, GNNs). MemFerry provides a training paradigm for general offload training by leveraging DHA not only to overlap transmission but also to reduce the training memory requirement.

VII. Conclusion

In this paper, we propose MemFerry, which leverages hybrid GPU execution approaches to accelerate offload training and reduce memory usage. Our key contribution is to prove that GPU's DHA feature can potentially benefit deep learning training. We integrate the GPU DHA feature into offload training and design hybrid training patterns that can fully overlap the transmission of offload training. We propose a unified management abstraction called the shadow model which removes redundant memory copying. We further propose a gradient offload strategy - GO-MemFerry which reduces training memory requirement on demand. Evaluations show that MemFerry can speed up offload training meanwhile significantly reducing the memory requirement for training.

REFERENCES

[1] OpenAI, "GPT-4 technical report," *CoRR*, vol. abs/2303.08774, 2023.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[3] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:160025533

[4] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," *CoRR*, vol. abs/2302.13971, 2023.

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *NeurIPS*, 2020.

[6] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: memory optimizations toward training trillion parameter models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*. IEEE/ACM, 2020, p. 20.

[7] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. X. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, 2019, pp. 103–112.

[8] C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the GPU memory limit via smart swapping," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, 2020, pp. 1341–1355.

[9] J. Ren, J. Luo, K. W., M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 2021, pp. 598–611.

[10] 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, PPoPP 2018, Vienna, Austria, February 24-28, 2018*. ACM, 2018, pp. 41–53.

[11] 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," in *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. USENIX Association, 2021, pp. 551–564.

[12] B. Pudipeddi, M. Mesmakhosroshahi, J. Xi, and S. Bharadwaj, "Training large neural networks with constant memory using a new execution algorithm," *CoRR*, vol. abs/2002.05645, 2020.

[13] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: breaking the GPU memory wall for extreme scale deep learning," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*. ACM, 2021, p. 59.

[14] X. Sun, W. Wang, S. Qiu, R. Yang, S. Huang, J. Xu, and Z. Wang, "STRONGHOLD: fast and affordable billion-scale deep learning model training," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*. IEEE, 2022, pp. 71:1–71:17.

[15] J. Jeong, S. Baek, and J. Ahn, "Fast and efficient model serving using multi-gpus with direct-host-access," in *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*. ACM, 2023, pp. 249–265.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[17] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *CoRR*, vol. abs/1604.06174, 2016.

[18] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*. ACM, 2020, pp. 3505–3506.

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, 2019, pp. 8024–8035.

[20] N. Corporation, "Cuda c++ programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2024.

[21] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019.

[22] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, EMNLP 2020 - Demos, Online, November 16-20, 2020*. Association for Computational Linguistics, 2020, pp. 38–45.

[23] Z. Dai, Z. Yang, Y. Yang, J. G. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive language models beyond a fixed-length context," in *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*. Association for Computational Linguistics, 2019, pp. 2978–2988.

[24] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. S., 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, PPoPP 2018, Vienna, Austria, February 24-28, 2018*. ACM, 2018, pp. 41–53.

[25] 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 *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, 2020, pp. 891–905.

[26] 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 *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 2021, pp. 598–611.

[27] C. Zhang, F. Zhang, X. Guo, B. He, X. Zhang, and X. Du, "imlbench: A machine learning benchmark suite for CPU-GPU integrated architectures," *IEEE Trans. Parallel Distributed Syst.*, vol. 32, no. 7, pp. 1740–1752, 2021.

[28] Z. Li, Q. Cao, Y. Chen, and W. Yan, "Cotrain: Efficient scheduling for large-model training upon GPU and CPU in parallel," in *Proceedings of the 52nd International Conference on Parallel Processing, ICPP 2023, Salt Lake City, UT, USA, August 7-10, 2023*. ACM, 2023, pp. 92–101.

[29] J. Bae, J. Lee, Y. Jin, S. Son, S. Kim, H. Jang, T. J. Ham, and J. W. Lee, "Flashneuron: Ssd-enabled large-batch training of very deep neural networks," in *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, M. K. Aguilera and G. Yadgar, Eds. USENIX Association, 2021, pp. 387–401.

[30] Z. Wang, Z. Wang, J. Liao, C. Chen, Y. Yang, B. Dong, W. Chen, W. Chen, M. Lei, W. Guo, R. Chen, Y. Peng, and Z. Yu, "CNN-DMA: A predictable and scalable direct memory access engine for convolutional neural network with sliding-window filtering," in *GLSVLSI '21: Great Lakes Symposium on VLSI 2021, Virtual Event, USA, June 22-25, 2021*. ACM, 2021, pp. 115–121.

[31] S. Min, K. Wu, S. Huang, M. Hidayetoglu, J. Xiong, E. Ebrahimi, D. Chen, and W. W. Hwu, "Large graph convolutional network training with gpu-oriented data communication architecture," *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 2087–2100, 2021.