流水线，d2d(压力大到压力小，生存间隔短),c2g,后几层优先考虑重计算
给每个tensor赋予一种内存技术
2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)
先赋予c2g(生存间隔长可重叠)、重计算（计算开销小于等于c2g不能被重叠的开销）再替换成d2d

MPress: Democratizing Billion-Scale Model Training on Multi-GPU Servers via Memory-Saving Inter-Operator Parallelism

Quan Zhou[1], Haiquan Wang[1], Xiaoyan Yu[1], Cheng Li[1,2], Youhui Bai[1], Feng Yan[3], Yinlong Xu[1,2]
[1]University of Science and Technology of China [2]Anhui Province Key Laboratory of High Performance Computing [3]University of Houston

2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071077

***Abstract*—It remains challenging to train billion-scale DNN models on a single modern multi-GPU server due to the GPU memory wall. Unfortunately, existing memory-saving techniques such as GPU-CPU swap, recomputation, and ZeRO-Series come at the price of extra computation, communication overhead, or limited memory reduction.**

We present `MPress`, a new single-server multi-GPU system that breaks the GPU memory wall of billion-scale model training while minimizing extra cost. `MPress` first discusses the trade-offs of various memory-saving techniques and offers a holistic solution, which alternatively chooses the inter-operator parallelism with low cross-GPU communication traffics, and combines with recomputation and swap, to balance training performance and sustained model sizes. Additionally, `MPress` employs a novel, fast D2D swap technique, which simultaneously utilizes multiple high-bandwidth NVLink to swap tensors to light-load GPUs, based on a key observation that inter-operator parallel training may result in imbalanced GPU memory utilization and spare memory space from least used devices plus the high-end interconnects among them have the opportunity to support low-overhead swapping. Finally, we integrate `MPress` with PipeDream and DAPPLE, two representative inter-operator parallel training systems. Experimental results with two popular DNN models, Bert, and GPT, on two modern GPU servers from the DGX-1 and DGX-2 generation, equipped with 8 V100 or A100 cards, respectively, demonstrate that `MPress` significantly improves the training throughput over ZeRO-Series with the identical memory reduction, while being able to train larger models than the recomputation baseline.

***Index Terms*—Inter-Operator Parallelism, DNN Training, Swap, Recomputation**

I. Introduction

To support today's AI revolution, deep neural networks (DNNs) are becoming larger with the number of parameters increasing from million-scale to billion-scale [3], [20], [62]. The model size is expected to continue growing in the future, as recent studies prove that the model accuracy and resource efficiency have been improved with the growth of model size [37], [51]. Along with this trend, there is an unprecedented growth of GPU memory demands, far beyond a single GPU's memory capacity (typically a few tens of GBs). Thus, it has been a norm to break the GPU memory wall by parallelizing large DNN model training over multiple GPU accelerators.

In this paper, we focus on scaling up billion-scale model training on a single modern multi-GPU server for the following reasons. First, billion-scale DNN models have already covered a wide range of applications [30], [57]. Second, enhanced single-server performance can be the building block for accelerating cross-server giant model training [50]. However, this is challenging because of the GPU memory bottleneck. For example, a high-end 8-V100 GPU server (AWS EC2 p3dn.24xlarge instance), offering 256 GB GPU memory in total (32 GB per GPU card), still can not sustain the training of a GPT model with over 10.3 billion parameters, which, however, requires 325 GB GPU memory. The huge memory space is used to accommodate not only model parameters, but also the dynamically generated data alongside computation, such as activations, optimizer states, gradients, etc.

To alleviate GPU memory limitation, intra- and inter-operator parallelism split DNN models and distribute partitions with lower GPU memory footprints to multiple GPUs [64]. However, the stand-alone model partitioning solution is insufficient to fulfill the goal. For instance, even using the leading inter-operator parallel training system PipeDream [43], we observe out-of-memory errors when the Bert model with over 640 million parameters, on the above 8-V100 GPU server.

There are also various memory-saving techniques [31], [33], [38], [49]. For instance, recomputation drops activations generated by the forward pass and recovers them when needed in the backward pass by triggering the original computation again [33]. However, it results in extra computation overhead and is not applicable to data other than activation. In comparison, a more general approach is to swap model data between GPU and CPU memory during computation [31]. However, existing GPU-CPU swap solutions suffer non-negligible performance loss, mainly due to the limited PCIe bandwidth, precluding the opportunities to overlap computation and swapping. Most recently, ZeRO-Series eliminate data redundancy in data parallel training and combine both recomputation and swap to reduce GPU memory consumption, at the price of introducing cross-GPU communication overhead for exchanging all data except activations.

To break the GPU memory wall and address the above challenges, we propose `MPress`, a single-server multi-GPU system that enables billion-scale model training with improved performance. Note that `MPress` outperforms ZeRO-Series w.r.t the same-sized large models because we choose inter-operator parallelism as the underlying parallel training strategy, which introduces extremely lower cross-GPU communication than data parallel that the latter systems rely on.

More importantly, `MPress` strategically saves the GPU

memory usage of inter-operator training by considering new hardware features and its unique GPU memory consumption. First, high-end GPU machines such as NVIDIA DGX Systems [14] bring advanced interconnects between GPUs with very high bandwidth, e.g., up to 50 GB/s between two GPUs, 212.5% higher than the bandwidth of PCIe 3.0 x 16. In addition, with up to 6 high-bandwidth in/outbound links per GPU, one can achieve even higher aggregated bandwidth. Second, we identify the ill-imbalanced GPU memory consumption in inter-operator parallel training, where the memory utilization of GPUs that host early stages of the whole pipeline is significantly higher than the rest (e.g., up to 7.9 $\times$ gap between the most and least used GPU memory).

The two findings motivate us to propose a new GPU memory swapping methodology (named *D2D swap*), which swaps tensors from GPUs with high memory pressure to their peers with spare memory resources. It further enables fast, parallel data transfer between GPU devices by weighted data stripping and stage-device mapping to better leverage the spare GPU memory resources and available NVLink interconnects. Ultimately, our new D2D swap technique can accelerate the speed and bandwidth of swapping and improve the performance and scale of inter-operator parallel training.

Despite being fast, the size of spare GPU memory is limited. Therefore, `MPress` only applies the D2D swap method to reduce short-lived tensors' memory footprint. This is difficult to be achieved by existing techniques as the overhead of swapping short-lived tensors can only be compensated by a very fast swapping speed. To further improve the available memory capacity, we also leverage the large capacity of CPU memory and the recomputation technique. Different optimizations present different trade-offs between memory-saving amounts and extra overheads, depending on several factors such as tensor sizes, computation complexity, operator dependency and scheduling, and swapping speed. Therefore, to support billion-scale models, while delivering reasonable training throughput, `MPress` carefully combines D2D swap, GPU-CPU swap, and recomputation together with proper configurations. To do so, `MPress` employs a decision maker that compares the benefits and costs of D2D swap, GPU-CPU swap, and recomputation; uses key factors such as data-flow computation graph, operator dependencies, live interval analysis, and tensor sizes to strategically optimize configurations for tensors produced by different stages.

Finally, we integrate `MPress` into PipeDream and DAPPLE, two representative inter-operator parallel training systems, and train two widely-used DNN models, Bert and GPT, from the natural language processing (NLP) field using two high-end GPU servers following the DGX-1 or DGX-2 architecture. They both have 8 GPUs connected via NVLinks. However, the DGX-2 server uses A100 GPUs (40GB memory per each) and the symmetric GPU connection topology, while DGX-1 uses V100 (32GB per each) and asymmetric connections. Experimental results show that `MPress` supports Bert with up to 6.2 billion parameters and GPT with up to 25.5 billion parameters, 3.7$\times$ and 1.7$\times$ of the counterparts achieved by the Recomputation baseline, respectively. When executing training jobs with the same large model, under GPU memory pressure, `MPress` introduces 1.4 - 2.3$\times$ speedups of training throughput, compared to the ZeRO-Series baselines.

The rest of the paper is organized as follows: we introduce the background and preliminary studies to motivate our work in Section II. We outline the design overview and key techniques of `MPress` in Section III. We analyze experimental results in Section IV, position our work in comparison to existing proposals in Section VI and conclude in Section VII.

II. Background and Motivation

A. Parallel DNN Training

As DNN computation is resource-hungry, it is a norm to parallelize the model training jobs across multiple GPU devices to leverage the massive parallelism. There are three main parallel training methods, each corresponding to a different partitioning strategy, namely, partitioning by input samples (data parallelism), by network structure (model parallelism), and by layer (pipeline parallelism). As it is easy to misunderstand the latter two parallelisms, we follow the catalog used by the recent Alpa work [64] to categorize existing solutions into two orthogonal directions, namely, intra-operator and inter-operator parallelism.

Intra-operator parallelism relies on the fact that an operator works on tensors with multiple dimensions to partition tensors along some dimensions and assign the resulting sliced operators to multiple devices [19], [59]. Data parallelism, as the simplest intra-operator parallelism, partitions the input tensors, which makes operators unchanged, and distributes data shards to GPU devices for training the shared, replicated model [5], [19], [42], [54], [62]. Unlike them, an inter-operator parallel training partitions the target DNN model into disjoint stages, each corresponding to a consecutive set of model layers and mapped to a separate GPU for its computation [21], [32], [43]. The minibatch training data is processed through stages in a pipeline manner.

Unfortunately, all the above parallel strategies face the GPU memory bottleneck for supporting billion-scaling single-server training. However, we choose inter-operator parallelism as our starting point for the following reasons. First, compared to the other two methods, the data parallel training results in the heaviest memory footprints and cross-GPU communication, as every GPU replicates the same amount of model data and periodically exchanges gradients, equal in size to the model parameters. Thus, data parallelism alone is difficult to meet the massive memory demands of the rapidly growing model sizes [34], [44], [46], [54]. Intra-operator parallelism splits operators into smaller ones, requiring heavy communication to gather and aggregate partial results to trigger the subsequent computation, which is sitting on the critical path of training.

In comparison, inter-operator parallel training introduces the least communication overhead, as for large NLP models, only activations are transferred between stages, and they are often small. For instance, concerning the Bert-0.64B model, only $microbatch\_size$ $\times$ 1.5 MB data are exchanged between

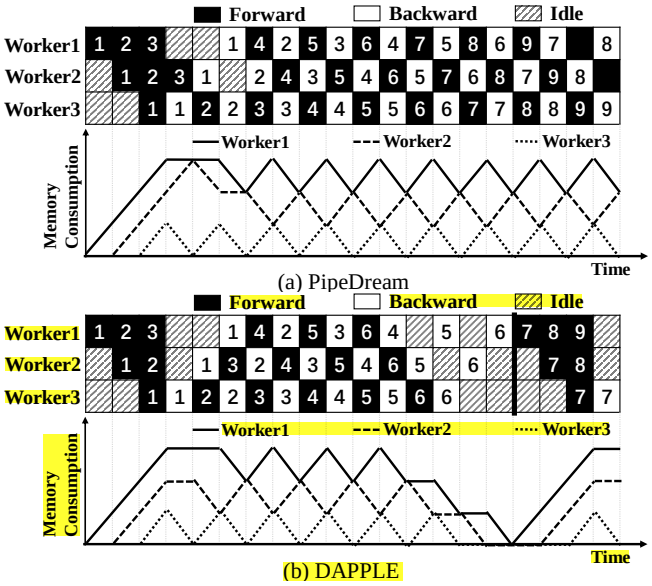


Fig. 1: The training workflow and timeline diagram of inter-operator parallel training in PipeDream and DAPPLE. Each minibatch consists of 6 microbatches. Microbatches 1-6 belong to the first minibatch, while 7-9 are part of the second minibatch. Black and white boxes represent the forward and backward computation pass of each microbatch, respectively. We also show the evolution of the per-device GPU memory consumption over time.

two GPUs that host two consecutive stages. Furthermore, inter-operator parallelism has received increasing attentions from both industry and academia [21], [44], [50], [61], [63]. Many training systems like PipeDream [43], DAPPLE [21], GPipe [32], DeepSpeed [50] and Megatron-LM [44] have already incorporated it. Thus, we focus on enabling fast billion-scale model training with inter-operator parallelism.

B. Inter-Operator Parallelism

Figure 1 showcases the inter-operator parallel DNN training workflow. Each of the three GPU devices (a.k.a workers) is responsible for training a disjoint set of consecutive model layers. Each minibatch of training data is further divided into six microbatches to feed into the whole execution pipeline. Worker 1 starts the forward pass of the first minibatch by consuming its first microbatch and passes the computation to Worker 2 to kick off the second stage. Meanwhile, the second microbatch is being processed by Worker 1. The same process will be applied to Worker 3. When Worker 3 finishes the forward pass computation of the first microbatch, the corresponding backward pass will be immediately started and flowed back from Worker 3 to Worker 1.

There are two ways to schedule the execution between adjacent minibatches, namely, asynchronous and synchronous. The asynchronous mode used in PipeDream [43] allows the forward pass of the second minibatch starts in parallel with the backward pass of the first one. For instance, the seventh

	Activation	Optimizer states	Parameters & Gradients
Bert-0.64B	39%	46%	15%
GPT-5.3B	42%	44%	14%

TABLE I: The GPU memory consumption contributed by different types of model data for various trainable models (measured in percentage). B stands for billion.

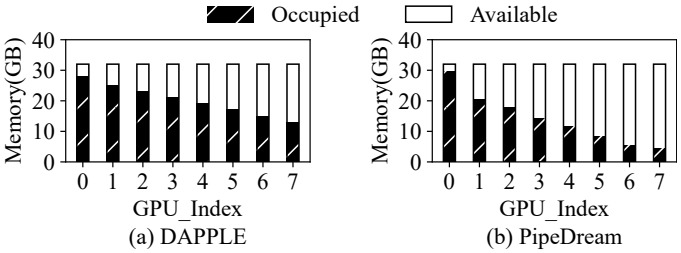


Fig. 2: Imbalanced per-device GPU memory consumption when training Bert (1.67B parameters) in PipeDream and DAPPLE with the batch sizes set to be 2 and 12, respectively.

microbatch belonging to the second minibatch is executed by Worker 1 immediately after the backward pass of the fourth microbatch completes (Figure 1(a)). In contrast, the synchronous mode used in GPipe [32] and DAPPLE [21] imposes a constraint that the computation of different minibatches is serialized (see the vertical bold black line in Figure 1(b)).

C. Problems of GPU Memory Consumption

To explore the GPU memory utilization of inter-operator parallelism, we train two popular DNN models, Bert and GPT, in PipeDream [13], [43] and DAPPLE [21], two representative inter-operator training systems. We deploy experiments on an AWS EC2 p3dn.24xlarge GPU server, which has 8 V100 GPUs (each has 32GB GPU memory), and set the largest sustainable model sizes. Here, we use the recommended stage partition strategy to balance the computation time for each stage within the two systems and assign the stages to GPU-(0-7) in order. In more detail, the partitioning strategies focus on balancing per-stage computation.

Largest sustainable model sizes. With the microbatch size of 12, PipeDream is able to support the training of a Bert model with up to 600 million parameters, beyond which out of GPU memory errors will surface. When shrinking the microbatch size to 2, the largest trainable Bert model with PipeDream is made up of 2 billion parameters. With the same hardware setup, DAPPLE can train GPT models with a maximum of 5.3B parameters and a microbatch size of 2. The reason for the sustainable model size gap between PipeDream and DAPPLE is that the former system using asynchronous scheduling requires stashing multiple versions of model data. Table I summarizes the percentage of GPU memory occupied by different types of model data. Concerning GPT-5.3B, among all its memory-resident tensors, activation accounts for 42%, optimizer states for 44%, and parameters and gradients for 14%. The key to train even larger models with preserved

performance is compacting the memory usage of these model data while minimizing the associated overhead.

Imbalanced GPU memory consumption. Figure 2 reports the per-GPU memory footprint of the two training jobs. We observe an ill-balanced GPU consumption across all cases, where the GPUs with lower indices that hold earlier stages of the corresponding pipeline execution use a significantly larger amount of GPU memory than the rest. In particular, the most used GPU memory space is up to 7.9 × of the least used one. When the batch size or model size is slightly increased for the experiments with Bert variants, both training jobs fail to run due to the out-of-memory (OOM) errors reported by the GPUs that suffer high GPU memory pressure.

The imbalanced GPU consumption is an inherent problem of inter-operator parallelism due to the following reasons. Activation states are generated by running the forward pass computation of each microbatch and then kept till being used by the corresponding backward pass. Therefore, as shown in Figure 1, the GPUs hosting the computation of the early stages accumulate more activation states than the rest. For instance, in the workflows of both PipeDream and DAPPLE, Worker 1 holds three copies of activation states before the backward pass of the first microbatch starts (white box), while Worker 3 keeps only a single copy, which can be further released after the corresponding backward pass completes. Additionally, when using asynchronous scheduling such as PipeDream, multiple versions of parameters must be kept to ensure convergence. The curves at the bottom of each subfigure of Figure 1 show the evolution of per-GPU memory consumption with a decreasing trend from Worker 1 to 3 all the time.

D. Memory-saving Optimizations and Their Limitations

Here, we iterate a list of state-of-the-art GPU memory-related optimizations that share the same goal of breaking the GPU memory wall as ours and discuss their benefits and limitations when being applied to inter-operator parallel training.

Memory-balance partitioning strategies within inter-operator parallel training could address the above GPU memory imbalance. However, we've verified that adopting the memory-balance partitioning strategy is not a good option as the price paid to break the memory wall is to make the computation time costs imbalanced across different stages, which leads to 34% training performance loss, compared to the computation-balance partitioning ones.

Recomputation trades the computation redundancy for saving the memory usage of activations, and has been combined with all parallel training strategies by many mainstream systems such as Megatron, DeepSpeed, and Alpa [44], [50], [64]. However, the stand-alone recomputation solution has the following two major drawbacks. First, it is not applicable to reduce the memory consumption of the remaining three types of model data, namely, optimizer states, parameters, and gradients, which together account for almost another half of the total GPU memory utilization, e.g., 58-61% in Table I. Second, re-executing the forward computations would contend

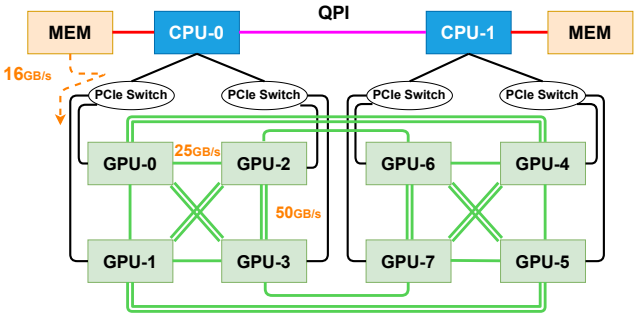


Fig. 3: The hardware configurations of a DGX-1 server with NVLink interconnects.

GPU resources with the backward ones and introduce extra delay, which can lead to up to 33% longer training time.

GPU-CPU swap leverages the large capacity of CPU memory or even storage devices like NVMe SSDs to extend the GPU memory space and is a general solution applicable to all model data. Nevertheless, it has not been applied in inter-operator training yet. Furthermore, the stand-alone GPU-CPU swap results in a significant training throughput loss, due to the tension between the huge amount of tensors that demand swapping and the limited PCI-e bandwidth between GPU and CPU. For instance, when training a Bert model with 640 million parameters via PipeDream, applying GPU-CPU swap to 39% of the model data in the first stage results in a 67% reduction in training throughput (measured as sentences processed per second), compared to the ideal counterparts with sufficient GPU memory supply and no swap.

Zero-Series cover a family of memory optimizations towards breaking the GPU memory wall for training DNN models with data parallelism rather than inter-operator parallelism. The first work, Zero Redundancy Optimizer (ZeRO) [48], eliminates the data redundancy of data parallelism by partitioning and distributing parameters, gradients, and optimizer states among multiple GPUs. The memory space saving comes at the cost of imposing extra communication overhead for gathering partitioned model data. ZeRO-Offload [51] further offloads optimizer states and the associated computation from GPU to CPU using a CPU-based Adam optimizer implementation. However, offloading cannot work with the asynchronous scheduling as each microbatch execution requires transferring parameters and gradients between GPU and CPU, demanding higher bandwidth than PCI-e's capacity. Most recently, ZeRO-Infinity [49] additionally incorporates GPU-CPU swap and leverages storage media like NVMe SSDs. It is worth mentioning that Zero-Series can significantly increase the trainable, single-server model size, but they introduce non-negligible training performance loss, due to the cross-GPU and GPU-CPU communication overhead.

E. New Hardware Trends and Opportunities

With the rapid hardware development, modern GPU servers have already incorporated ultra-high-speed links interconnecting GPU devices. In 2016, NVDIA P100 GPU adopts the first

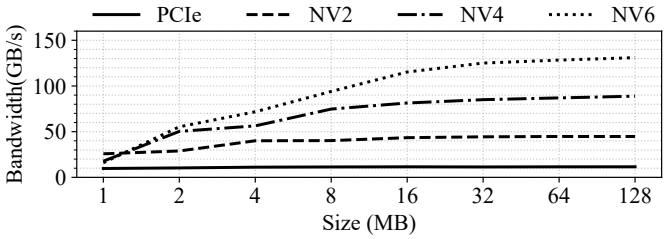


Fig. 4: The unidirectional aggregation bandwidth of links from a single GPU perspective w.r.t different data sizes. PCIe indicates the connection traversing a single PCIe. NV# indicates the connection traversing the number of NVLinks.

generation NVLink, which offers up to 160 GB/s data transfer bandwidth (bi-directional), nearly 5× of the bandwidth of PCIe Gen3x16. Figure-3 illustrates the internals of a DGX-1 machine, which uses NVLink 2.0 (white lines), which, in theory, can reach up to 300GB/s bidirectional bandwidth.

To explore the bandwidth of NVLink, we run experiments to transfer data using different number of links between a GPU and its neighbors on an AWS EC2 p3dn.24xlarge GPU instance (hardware details listed in Section IV-A). Note that this EC2 instance is similar to the NVIDIA DGX-1V system [52]. We compare NVLink interconnects between GPUs against PCIe links between GPU and CPU. As shown in Figure 4, the aggregated unidirectional bandwidth has been amplified from 45 to 146 GB/s when increasing the number of NVLinks from 2 to 6, which is 3.9-12.5× of the PCIe bandwidth (corresponding to the bandwidth of GPU-CPU swap).

In this paper, we propose a new GPU D2D swap (or short, D2D swap) to alleviate the GPU memory limitation presented above. D2D swap offloads tensors from GPUs that host early stages of inter-operator parallel training with high GPU memory pressure to peers with spare GPU memory resource via high-speed NVLink, and swaps in for subsequent uses. Our D2D swap technique has the following unique benefits. First, Figure 4 illustrates that swapping between GPUs is significantly faster than the GPU-CPU swap, and has the potential to avoid the significant training throughput loss imposed by GPU-CPU swap. Second, as opposed to recomputation, which drops activation states to release memory space and re-executes the corresponding forward pass before the dropped data is needed, D2D swap does not consume GPU computation resources and can be better overlapped with the backward computation.

III. MPress Internals

A. Design Rationale

We propose MPress, an efficient inter-operator parallel DNN training system using heterogeneous memory reduction optimizations to address the well-known GPU memory wall challenge. MPress employs the proposed *D2D swap* to offload model data from GPUs with high memory consumption pressure to the light-loaded ones via multiple high-bandwidth NVLink interconnects. As the GPU resources are limited, we choose to apply D2D swap only to a small fraction of model data, whose live intervals (i.e., the time between its generation/previous usage and the next usage) are too short to be tolerated by either *recomputation* or *GPU-CPU swap*.

The fast speed of D2D swap is crucial to avoid paying high extra performance loss associated under *recomputation* and *GPU-CPU swap* optimizations, and allows D2D swap to better overlap with DNN computation. By carefully mapping pipelined stages to GPU devices, we can aggressively fulfill the out/in bandwidth demands of each GPU. To overcome the limited overall swapping space of GPUs, MPress further employs *recomputation* and *GPU-CPU swap* techniques to reduce the GPU memory consumption.

B. Overview of MPress and its Workflow

Figure 5 gives a high-level view of MPress system architecture, which spans its logic across both static and runtime parts. The mission of the static part is to generate the memory saving plan, which determines the memory-resident tensor candidates for applying memory reduction optimizations under memory pressure; which optimization to apply; and when to perform the determined optimization or recover reduced tensors. To achieve this, in the static part of MPress, a *profiler* trains a target DNN model with the trainable model and batch size to obtain the basic stats ranging from tensor sizes to the latencies of the forward/backward computation for each tensor (① - ②). Table III showcases an example of the collected stats. Note that new models [22] may have dynamic memory consumption. However, most SOTA models that industry use show static memory usage [39], [48], [49], [55]. We will adapt MPress to leverage dynamically evolving memory space in future work. Then, the *planner* explores a possible configuration, which assigns proper memory-saving strategies to various tensors (③). At this step, *planner* is driven by a simple cost model, which compares the time cost of different optimizations to choose the one with the least performance penalty.

Following that, the generated tentative plan is fed into the *rewriter*, which further instruments the input data flow graph to incorporate these assigned strategies in proper places to respect the operator dependencies (④). We then employ an *emulator* to run a single training iteration according to the revised data flow graph with the target model size or batch size, and to collect the amount of saved GPU memory and incurred overhead. Finally, the feedback will be sent from *emulator* to *planner* to determine if the currently exercised configuration approximates the optimal one by comparing it against previous runs (⑤). The interaction among *planner*, *rewriter* and *emulator* may run throughout a series of iterations to converge to the final configuration as the input that is passed through the runtime part. Table IV demonstrates the strategies produced by MPress Static.

Note that MPress Static runs offline and thus does not incur runtime overhead. The time cost for the offline analysis is moderate as *emulator* only needs to exercise a limited number of training steps, each of which just runs one iteration. However, as a comparison, the actual training may require running millions of iterations towards model convergence [60].

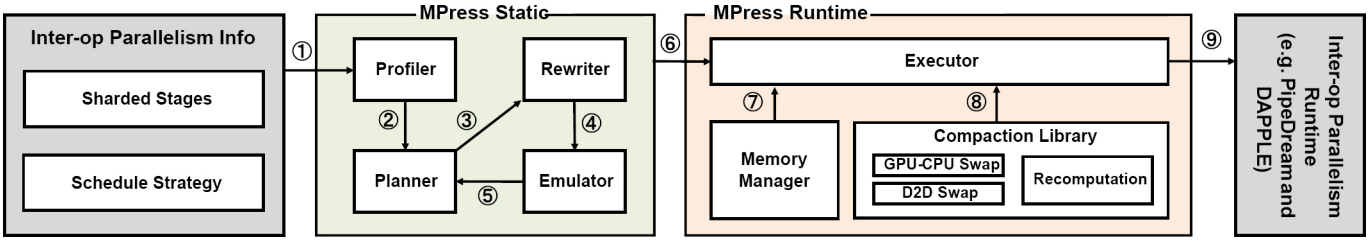


Fig. 5: MPress training system overview.

The *MPress Runtime* co-locates with the inter-operator parallel training framework runtime, and consists of three key system components, namely, *executor*, *memory manager*, and *compaction library*. First, the *compaction library* offers the efficient implementation of the three supported memory saving optimizations, where we include our novel D2D swap (details are in Section III-C), while re-using the built-in recomputation code from the training systems and implementing GPU-CPU swap by following the rich literature [31]. The Runtime workflow is as follows. The *executor* takes the instrumented data flow graph from *MPress Static* as the input (⑥), and triggers the memory compaction-enabled inter-operator parallel training. The ordinary operators except for memory-saving ones directly go through the underlying training framework runtime as usual (⑨). The *executor* executes the memory saving operators (i.e., swap-out, drop) to release used GPU memory and state recovering ones (i.e., swap-in, recompute) for fulfilling their next usage (⑧). During the execution, the *memory manager* takes over the memory allocation/deallocation by following the executor's commands (⑦).

C. D2D Swap

As the major challenge of designing our novel D2D swap technique among GPUs, we have to jointly take into account of (1) the possible hardware heterogeneity of the total number of links between swap out/in pairs and the per-pair bandwidth, and (2) the diversity of the memory swap-out demands and the spare GPU memory space of a target inter-operator parallel training job. To this end, we introduce the following two key techniques for efficient D2D swap optimization.

Data stripping. It is possible for a GPU device to swap tensors to many other NVLink reachable peers for fast data transfer. Therefore, we introduce **data striping** technique, which partitions a target tensor into several sub-blocks, and transmit them in parallel through disjoint links. For the symmetric NVLink topology used in new DGX-2 architecture and beyond, where GPUs are fully connected via homogenous links, we make the sub-blocks equally sized and its total number be the number of target importer GPU devices.

Unlike this, the older DGX-1 architecture uses an asymmetric topology, where the bandwidth between GPU pairs can vary. For example, in Figure 3, GPU0 can transfer data to GPU3 at a speed of 50GB/s because of the two NVLink interconnects, which have twice the bandwidth of GPU1. To realize this difference, we further evolve the **data striping** technique to incorporate the weighted bandwdiths among reachable NVLink connections, in which the sizes of such sub-blocks are proportional to the corresponding link bandwidths. We also batch tensors to fully use the high bandwidths.

```
1   #spare mem assignment from the view of a single GPU
2   def assign_mem(gpu, dev_map):
3       spare_amount = MEM_CAP − MEM_USE[gpu]
4       set nbhs = all NVLink neighbors of gpu in dev_map
5       set exporters = overflowed gpus
6       set candidates = nbhs ∩ exporters
7       set plans = all possible ways to distribute mem of
              spare_amount to candidates
8       return plans

10  def device_mapping_search():
11      best_score = 0
12      best_dev_map = None
13      set all_map = enumeration with no mem constrains
14      for dev_map in all_map:
15          all_plans = []
16          for g in all_gpus:
17              if g has spare mem:
18                  all_plans.add(assign_mem(g, dev_map))
19          #combining single gpu's plans
20          concat_plans = permutation(all_plans)
21          for plan in concat_plans:
22              score = ratio of revenue to cost
23              if score > best_score:
24                  best_score = score
25                  best_dev_map = dev_map
26      return best_dev_map
```

Fig. 6: Device mapping algorithm

In addition, we manage a metadata table to keep track of the states of tensors that go through our D2D swap. For each tensor, we record the following information before the execution of the swap-out operator, namely, the number of sub-blocks, the sizes of each sub-block, and the indices of target GPU devices. This information is used to guide the execution of the latter swap-in operator and updated when it completes.

Device mapping. Figure 6 illustrates the simple device-stage mapping algorithm, which properly assigns pipeline stages to GPU devices so that the light-loaded peers are the neighbors of GPUs with high memory pressure, and the swap-out/in bandwidths of the latter GPUs are maximized. It first enumerates all possible device mappings, and then for each mapping, identifies all possible spare memory allocation schemes from a

single GPU's point of view, and combine these schemes to the global swapping plans (line 14 - 20). Ultimately, we choose the optimal plan out of all the candidates by evaluating them using a scoring function (lines 21 - 25). This score function evaluates the effectiveness and efficiency of the pair of device mapping and spare GPU memory allocation from light-loaded GPUs to the ones with high memory pressure. Here, we compute the score as the reciprocal of the maximal time cost of D2D swap for a fixed assignment. Obviously, when the score is higher, the overall performance of D2D swap for offloading model data from high-loaded GPUs is better. Finally, we identify the best device mapping plus the spare GPU memory assignment as the one having the highest score. Note that for symmetric GPU connections, this algorithm simply skips the above logic, instead, randomly maps stages to devices and aggressively uses all NVLinks to its neighbors with spare memory. We evaluate the performance implications and report the time costs of adopting optimal device mapping in Section IV-D.

D. Memory Compaction Planning

Exploring the best configuration to combine D2D swap, GPU-CPU swap, and Recomputation to maximize the GPU memory saving while minimizing the extra delay imposed on the pipelined inter-operator parallel DNN training is very challenging. To address this challenge, we propose approximating this searching problem using the following key observations.

- Compared to Recomputation, the benefits of the two swapping methods are (1) they do not consume GPU computation resources; and (2) they can run in parallel with the forward and backward computation on GPUs. When appropriately applied, i.e., the live interval [1] of the target tensors is longer than the time cost of either of the two swapping methods, no extra time delay is introduced.
- We should prioritize Recomputation to alleviate the memory limitations of the latter layers of the target DNN model. The reasons are two-fold. First, their backward passes are started firstly in the second half of the pipelined execution; Second, the inevitably extra recomputation delay could enlarge the live intervals of the earlier layers, thus leaving more rooms for applying GPU-CPU swap.
- It is preferable to apply Recomputation to reduce tensors from consecutive layers. This choice can further reduce the memory consumption of the inputs of operators except the first one, which are the outputs of their preceding ones in the data-flow graph.
- As the spare GPU memory is scarce and D2D swap is much faster than GPU-CPU swap, to unleash its full potential, we should only apply it to performance-critical cases to minimize the extra delay imposed by both GPU-CPU swap and Recomputation.

Driven by the above observations and trade-offs, we introduce an approximated search algorithm, which first aggressively assigns GPU-CPU swap and Recomputation optimizations to proper tensors, and then optimizes the assignment by gradually replacing some GPU-CPU swap and Recomputation operators with D2D swap throughout a sequence of tuning iterations. Specifically, we first perform a live variable analysis [23] to compute the per tensor live intervals. Then we construct the initial tentative assignment as follows. We assign GPU-CPU swap to tensors with extremely long live intervals, while applying Recomputation to activation tensors when the imposed extra latency by itself is smaller to GPU-CPU swap [2]. Finally, we assign GPU-CPU swap to the remaining tensors to meet the target memory-saving goals.

Our algorithm goes through a few iterative steps to gradually update the memory reduction optimization assignment. At each step, we use our emulator to run the latest assignment (only one training iteration is sufficient) to filter out a set of reduced tensors, whose assigned optimizations introduce the most extra overhead. For those tensors, we try to use D2D swap to reduce their memory limitations whenever possible, e.g., when there is spare GPU memory. We accept the new assignment if its performance is better than its ancestor. This algorithm terminates if the subsequent assignment brings non-visible performance gains over the previous one.

E. Implementation Details

We incorporate the aforementioned design principles into an open-source training system `MPress` [2], which has $2k$ lines of code in C++ and Python. To demonstrate its generality for optimizing GPU memory usage in inter-operator parallel training, we integrate `MPress` into PipeDream [13] and DAPPLE [21], two recent systems using either asynchronous or synchronous scheduling. The backend engine is PyTorch [10]. We improve PipeDream by upgrading its original PyTorch from version 1.1 to 1.2, and enabling its NCCL library to use NVLink to transfer data between stages. Note that `MPress` is general and can be applied to other inter-operator training systems such as GPipe. We will continue to make `MPress` work with those systems.

Memory management. Our memory manager is responsible for allocating and releasing the GPU/CPU memory space for tensors and monitoring the per-device memory usage. First, as for GPU memory allocation, the *manager* directly uses the native GPU memory allocator in PyTorch. Second, upon the host memory request, considering that the data transmission between pinned memory and GPU memory is faster than that with normal pageable memory, we decide to use pinned memory for swapping space. To avoid paying high cost for allocating and freeing pinned memory, we further build a host pinned memory pool that is not part of the PyTorch runtime.

Memory swapping. For D2D swap, the *executor* manages two extra threads for executing swap-in and swap-out tasks, respectively, which use different CUDA streams created by calling `cudaStreamCreate` upon the system bootstrapping. This design allows *executor* to launch tensor transferring

[1]Live interval of a tensor is the time duration between its generation and the subsequent usage. For instance, concerning activation tensors, their live interval is computed by the difference between the timestamps of its backward and forward passes.

[2]The extra time overhead of GPU-CPU swap is computed by subtracting the live interval of the target tensor from the swapping cost.

	Config.	Total	per-stage Max	per-stage Min
Bert+PipeDream	0.35B	108.8	24.7	3.7
	0.64B	227.0	50.6	6.4
	1.67B	345.9	78.0	8.8
	4.0B	578.70	128.3	16.3
	6.2B	1279.1	280.6	35.5
GPT+DAPPLE	5.3B	164.8	28.5	12.7
	10.3B	325.0	56.4	24.9
	15.4B	486.7	84.5	37.2
	20.4B	646.9	112.4	49.4
	25.5B	806.2	140.1	61.5

TABLE II: GPU memory demands (in GBs) measured for different training jobs of models with various parameter scales. Max and Min represent the most and the least memory consumption per stage, while total represents the total number of GPU memory for the whole model. The abbreviation "B" denotes billion. The microbatch sizes for Bert and GPT are 12 and 2, respectively.

tasks and check their status with DNN computation without blocking the main thread. Thus, data movement between GPUs can work asynchronously with DNN computation.

IV. Evaluation

Our evaluation aims to answer the following questions.

- Will MPress effectively alleviate GPU memory limitation to support large model sizes within inter-operator parallel training, while delivering better training performance than baselines?
- What are the performance implications of the stand-alone D2D swap, and device mapping and memory compaction strategies?
- What are the individual contributions of the three memory reduction optimizations to the overall GPU memory saving?

A. Experimental Setup

Machine configurations. We conduct experiments on both DGX-1 and DGX-2 GPU servers to evaluate MPress and baselines. The DGX-1 server is an AWS EC2 p3dn.24xlarge instance [7], which has 96 vCPUs, 8 NVIDIA Tesla V100 GPUs (32GB per-device memory, connected by asymmetric NVLink), and 768 GB CPU memory. We additionally use an DGX-2 server from another provider, as the quota of this type of high-end GPU servers is extremely limited on EC2 and our provisioning requests failed many times. It has 164 vCPUs, 8 NVIDIA Tesla A100 GPUs (40GB per-device memory, connected by symmetric NVLink), 948 GB CPU memory, and 6TB NVMe SSD. Both run software like Ubuntu 18.04, CUDA 11.7, NCCL 2.8.4, PyTorch 1.2.0, etc.

Models and datasets. We choose two widely-used DNN models Bert and GPT, both from the natural language processing field. We train Bert with the SQuAD v1.1 dataset [12], and GPT with the Wikipedia dataset [15].

We exercise MPress atop PipeDream using Bert and its variants with varied model sizes. By following the literature [1], we make Bert variants deeper and wider by adjusting the number of encoder layers and the value of hidden sizes. As shown in Table II, our Bert variants have 0.35 to 6.2 billion parameters. Bert-0.35B represents the smallest Bert model, with a total GPU memory requirement of 108.80 GB and a maximum requirement of 24.77 GB per stage. Clearly, its GPU memory demands can be fulfilled by our tested GPU server without memory reduction optimizations. Bert-0.64B is medium-sized with its maximum, and minimum per-stage GPU memory demands higher and lower than per-GPU memory capacity, respectively. Additionally, Bert models with 1.67 and 4B parameters denote the large models with all their per-stage memory consumption higher than per-GPU capacity. Finally, Bert-6.2B is the extra-large one with its total memory requirement being 5.0 $\times$ of the server GPU memory supply. We use 12 as the microbatch size, suggested by the repo [11].

Similarly, we use GPT and its variants with adjusted parameters to exercise MPress atop DAPPLE. As shown in Table II, among the five GPT configurations, the smallest model, GPT-5.3B, is trainable for the original DAPPLE. However, the remaining four GPT configurations demand a maximum requirement of 56.4-140.1 GB per-stage GPU memory, which already exceeds the single GPU's capacity. We set 2 as the microbatch size, corresponding to the smallest suggested batch size in the DAPPLE publication [21].

Baselines and system configurations. For inter-operator parallel training over PipeDream, we use the original PipeDream as the no-memory saving inter-operator training baseline. We further deploy two systems GPU-CPU Swap and Recomputation as the memory compaction-enabled baselines by enabling the memory swapping between GPU and CPU or the recomputation optimizations within PipeDream. We run two variants of MPress, where the one with D2D Swap only, while the other explores all three optimizations.

With regard to MPress based on DAPPLE, we run the original DAPPLE as the natural baseline with no memory optimizations. We further deploy DAPPLE with recomputation enabled. In addition, we run two state-of-the-art training systems, ZeRO-Offload [51] and ZeRO-Infinity [49], which can support large model training with data parallelism.

Furthermore, we use the computation-balanced stage partitioning strategies for both Bert and GPT, suggested by PipeDream and DAPPLE. For the recomputation baselines, we choose the set of tensors to drop by following the literature [16]. Finally, the MPress variants adopt the device mapping and memory saving plans generated by the aforementioned algorithms in Section III-C and Section III-D.

Metrics. We measure the total number of samples processed per second and floating point operations per second (FLOPS) as the training throughput, the time cost of D2D swap, GPU-CPU swap, and Recomputation, and the memory reduction breakdown among the three methods. Similar to other existing FLOPS calculation tools or methods [44], we measure the FLOPS of the forward pass of a model and estimate the FLOPS of the corresponding backward pass as two times that of the forward pass.

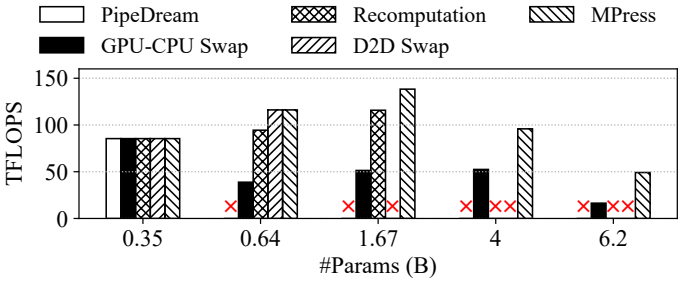


Fig. 7: The training performance comparison of variable-sized Bert models with different memory-saving optimizations. Red crossed marks indicate that the corresponding training jobs fail to run due to out-of-memory errors on GPUs. The based inter-operator parallel training system is PipeDream.

B. Performance with `MPress` atop PipeDream

Figure 7 compare the inter-operator parallel training performance measured in TFLOPS [3] of Bert variants with various sizes across five different system configurations. We analyze the results via the following categories.

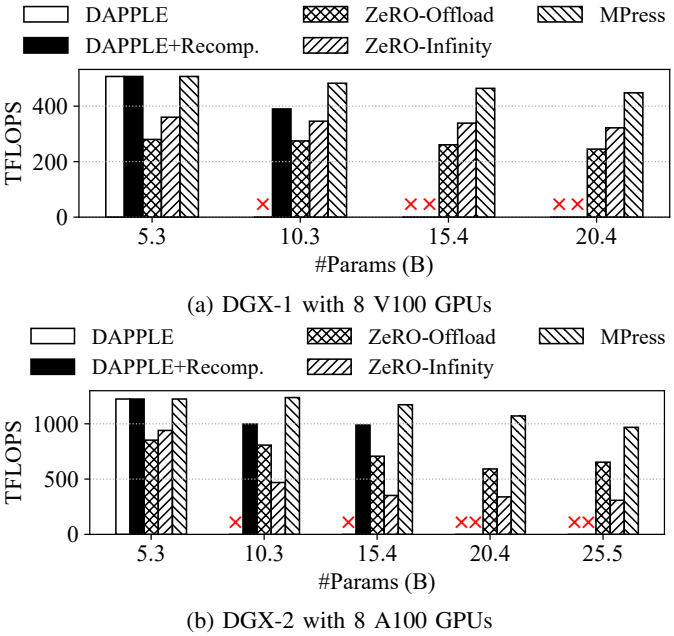
Small size. We begin our analysis with the smallest Bert with 0.35 billion parameters. All five systems successfully train this model and report identical performance numbers. This is because the ordinary inter-operator parallel training enabled by PipeDream can fulfill all its GPU memory demands, thus there is no need to trigger any memory compaction optimizations.

Medium size. When shifting our attention from Bert-0.35B to Bert-0.64B, PipeDream experiences out-of-GPU-memory errors, while the other four systems conduct successful executions. This is because, within PipeDream, the GPU memory consumption of stage 0 is 51 GB, higher than the GPU capacity. Consequently, we have to start applying the memory reduction optimizations from this model size. Among the four successful runs, GPU-CPU swap performs the worst. This is because the limited PCIe bandwidth makes swap operators slow, which further delays the corresponding DNN computation. Recomputation outperforms GPU-CPU swap by 143.4% because the extra latency imposed by re-executing the forward pass of dropped activations is often much lower than the corresponding GPU-CPU swapping counterparts. In comparison, the two `MPress` performs the best with identical performance. The reasons are two-fold. First, swapping tensors within GPUs is even faster than recomputation. Second, in this case, the stand-alone D2D swap is sufficient to alleviate the memory limitation. Thus `MPress` decides not to use the other two optimizations.

Large size. The continuation of increasing the model size to 1.67B leads the stand-alone D2D swap to fail. This is because, under high memory pressure, the spare GPU memory capacity cannot accommodate the tensors that would be offloaded from highly loaded GPUs. Similarly, Recomputation outperforms GPU-CPU swap by 125.4% but still performs 19.5% worse than `MPress`. In this case, `MPress` combines all strengths of the three optimizations so that both GPU-CPU swap and Recomputation can make more room for unleashing the potentials of D2D swap.

Interestingly, Recomputation fails to support Bert models with 4B parameters and above. This is as expected as recomputation can only save the memory consumed by activations generated by the forward pass, and fail to handle the remaining model data including parameters, gradients, etc, which use significantly more memory resources. In contrast, GPU-CPU swap is still functional as it can be applied to any kind of model data with sufficient host memory space. However, `MPress` introduces a 1.8 × speedup of training performance, compared to GPU-CPU swap, thanks to its prioritized adoption of the fastest D2D swap and the faster Recomputation whenever possible.

Extra-large size. Finally, we explore the performance implications by pushing the model size of Bert to 6.2B. Being consistent with the results of Bert-4B, only GPU-CPU swap and `MPress` survive this training task, while the other three systems fail to run. Though being able to support the same extra-large model, compared to GPU-CPU swap, by carefully choosing the optimal device mapping and exploring the combination of three optimizations, `MPress` introduces a 3.1 × speedup of training performance. In addition, `MPress` introduces a 2.7 × increase in model size compared to the state-of-the-art Recomputation.



(a) DGX-1 with 8 V100 GPUs

(b) DGX-2 with 8 A100 GPUs

Fig. 8: The training performance comparison of variable-sized GPT models among different memory optimization baselines. The based inter-operator parallel training system is DAPPLE.

C. Performance with `MPress` atop DAPPLE

We train another popular NLP model GPT with varied parameters to further stress `MPress` on both DGX-1 and DGX-2

[3] A TFLOPS (a.k.a teraFLOPS) refers to a GPU's capability to calculate one trillion (10^{12}) floating-point operations per second.

GPU servers. Here, we compare MPress with three strong baselines that demonstrated the application of the state-of-the-art memory saving techniques: (1) DAPPLE+Recomp which uses the high performant recomputation implementation; (2) ZeRO-Offload [51] with optimizer state offloading from GPU to CPU; and (3) ZeRO-Infinity [49], using the best-performing GPU-CPU swap, and extending the swapping space further from CPU memory to even larger NVMe device. Note that both ZeRO variants are from the DeepSpeed framework [4].We deploy tests on a high-end GPU server with the identical GPU setup as the above experiments but with much larger CPU memory and additional NVMe SSDs. We cannot use the above Amazon EC2 instance for this set of experiments because ZeRO-Infinity requires a large amount of CPU memory for initialization and the additional storage space with high I/O bandwidth for its tensor swapping.

Figure 8a summarizes the performance comparison with the DGX-1 GPU server. DAPPLE cannot scale to models with sizes beyond 5.3B, whose maximal per-GPU memory usage exceeds 32GB. Unlike this, the recomputation enabled by DAPPLE successfully runs the training job with up to 10.3B parameters, where it however observes 19.2% performance loss, compared to MPress. In contrast, the two ZeRO variants and MPress can support all four training jobs with model sizes ranging from 5.3B to 20.4B. ZeRO-Infinity outperforms ZeRO-Offload by 20.6-23.8% in terms of GPU computation efficiency. This is because offloading optimizer states results in frequent data movement between GPU and CPU per microbatch basis, and ZeRO-Infinity replaces it with the carefully designed GPU-CPU swap. However, MPress delivers constantly sustainable training performance, regardless of model sizes, thanks to the use of *D2D swap* and the combination of various memory compaction optimizations. In addition, MPress achieves 37.0-40.8% better performance than ZeRO-Infinity. This implies that the swapping solution excluding *D2D swap* may support very large-scale model training but at the price of sacrificing the performance speed. MPress actually can complement their limitations by further considering the spare GPU memory resources.

As illustrated in Figure 8b, we observe similar trends on the DGX-2 GPU server as DGX-1, but with the performance of all system configurations more than doubled, due to the higher computational density of the A100 GPU on DGX-2 over V100 on DGX-1. In addition, Recomputation can sustain the model sizes within 15.4B, higher than those achieved on DGX-1, because of the 40GB per-GPU memory, larger than 32GB of V100. Both ZeRO variants can scale to very large models with up to 25.5B parameters as MPress, but observe 30.4-44.8% and 23.2-70.0% reduction in training performance, compared to MPress. Interestingly, unlike the DGX-1 results, on larger models, ZeRO-Infinity performs worse than ZeRO-Offload. This is because the I/O bandwidth of SSDs of the rented DGX-2 server is significantly lower than DGX-1. However, it is impossible to find publicly accessible high-end GPU servers with scalable GPU computing power and storage capacity. Note that even with sufficient SSD bandwidth, ZeRO-Infinity shouldn't significantly outperform ZeRO-Offload, verified by its original publication [49] and results in Figure 8a.

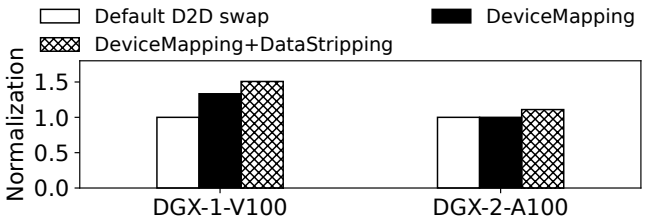


Fig. 9: Impacts of device mapping and data stripping on training performance of MPress with two different GPU topologies, namely, asymmetric DGX-1-V100 and symmetric DGX-2-A100.The training model is GPT-15.4B, and the microbatch sizes used is 2.

Model	Tensor name	Size (MB)	Live interval	Recomp.	GPU-CPU swap	D2D swap
Bert	t1	216	78	4	42	6
	t2	115	16	3	22	3
	t3	216	2	4	42	6
GPT	t4	384	214	8	74	9
	t5	384	50	8	74	9
	t6	1152	12	14	222	27

TABLE III: Time cost (in ms) comparison of three memory reduction optimizations when being applied to various tensors within Bert and GPT. Note that the D2D swap time cost corresponds to the usage of four NVLink interconnects.

Result gap between PipeDream and DAPPLE. Interestingly, there are big model size and performance gaps between the original PipeDream and DAPPLE, as well as the two MPress variants atop them. We've explained the reason for the model size difference in Section II-C. As for the performance gap, DAPPLE significantly outperforms PipeDream, since DAPPLE by default enables the low-precision training feature with FP16; additionally, DAPPLE was developed two years later than the release of PipeDream and has absorbed various optimizations from the deep learning community, for instance, better computation-communication overlapping.

D. Sensitivity Analysis

Here, to understand the strengths of MPress over baselines, we conduct a sensitivity analysis to explore the impacts of device mapping, the cost comparison among the three standalone optimizations (i.e., GPU-CPU swap, Recomputation, D2D swap), and the identified memory compaction plans.

Impacts of optimizations in *D2D swap*. Figure 9 summarizes the MPress performance achieved by gradually adding both device mapping and data stripping optimization, presented in Section III-C. The results are normalized to the default setting, where stages are mapped to devices via the DAPPLE's suggestion and *D2D swap* is enabled but with no data stripping. For DGX-1, device mapping and data stripping improve the performance of the default setting by 17.4% and 33.3%, respectively, as the former optimization enables

Model	Recomputation		GPU-CPU swap		D2D swap	
	Applied Stages	Saved GPU Mem.	Applied Stages	Saved GPU Mem.	Applied Stages	Saved GPU Mem.
Bert-1.67B	stage 0-6	121GB (76.6%)	N/A	0 (0%)	stage 0-3	37GB (23.4%)
Bert-6.2B	stage 0-7	972GB (90.6%)	stage 0-5	59GB (5.5%)	stage 0-4	42GB (3.9%)
GPT-10.3B	stage 0-7	156GB (82.5%)	stage 0-7	6GB (3.2%)	stage 0-3	27GB (14.3%)
GPT-20.4B	stage 0-7	294GB (51.2%)	stage 0-7	242GB (42.2%)	stage 0-3	38GB (6.6%)

TABLE IV: Strategies chosen by MPress for different stages with choices spanning across Recomputation, GPU-CPU swap and D2D swap, plus their individual contributions to the total memory saving

the swap-in/out pairs to transfer data via reachable NVLink connections, while the latter optimization further allows to maximize the data transferring bandwidth. However, unlike the DGX-1 results, device mapping introduces no optimization for DGX-2. This is as expected since symmetric, all-to-all NVLink connections among GPUs make every GPU have the identical number of NVLink interconnects. Contrary to the behavior of device mapping, data stripping makes MPress outperform the default setting by 11.0%. This is a direct consequence of leveraging aggregated bandwidth of multiple NVLinks for accelerating model data swap.

We also evaluate the time cost of running the device mapping algorithm. First, we composite an extreme case, which is much more complex than all experiments corresponding to Figure 7 and Figure 8, to stress our searching algorithm. Even with the single-threaded implementation, MPress identifies the optimal mapping within 47 seconds. In addition, for all cases in our evaluation, the device mapping searching takes a few seconds to complete. Therefore, we conclude that our searching algorithm does not incur heavy overhead. If needed, we can further improve it to be multi-threaded.

Memory compaction cost comparison. Table III reports the time cost of the three memory reduction optimizations for sampled, variable-sized tensors within Bert and GPT. Obviously, the three methods deliver significantly different performances, which plays a key role in determining their combination for the final memory-saving plan.

First, among the three tensors of Bert, t1 has the longest live internal, which can cover the time cost of both GPU-CPU swap and D2D swap. Therefore, MPress will prioritize the usage of GPU-CPU swap since its cost can be hidden while D2D swap can be saved for other tasks with more stringent demands. For t2, both GPU-CPU swap and Recomputation bring extra time overhead and would delay the pipelined inter-operator parallel training, since the former completes in a longer time than t2's live interval, while Recomputation brings an extra forward pass finishing in 3ms. Thus, MPress will choose D2D swap for this short-live tensor, whose time cost (3 ms) can be easily hidden. Finally, concerning t3, MPress discards the GPU-CPU swap due to its slowness, while prioritizing recomputation over D2D swap since both methods bring the same extra overhead (4ms) but Recomputation does not consume limited spare GPU memory, which perhaps has better use for other tensors.

Second, the above reasoning can also be transferred to GPT. Similarly, MPress assigns GPU-CPU swap to t4 thanks to t4's long live interval, while prioritizing D2D swap over the other two methods for t5, thanks to its superior data transferring performance, which improves that of GPU-CPU swap by 7.6 ×. Finally, for t6, Recomputation is chosen as this method introduces the least extra overhead, compared to the other two.

Strategies chosen by MPress. Table IV shows the optimal strategies generated by MPress for four inter-operator parallel training jobs under high GPU memory pressure, namely, Bert-1.67B, Bert-6.2B, GPT-10.3B, and GPT-20.4B. We also report the percentage of GPU memory reduction of each optimization. Among the three methods, the contribution of GPU-CPU swap to memory reduction is 0 - 42.2% of the total saved memory. Unlike this, Recomputation contributes the most, i.e., 51.2 - 90.6%. D2D swap saves 3.9 - 23.4% of GPU memory space, less than both recomputation and GPU-CPU swap for most cases, but still plays a key role in eliminating the extra overhead or GPU computation resource contention that would be imposed by both GPU-CPU swap and recomputation.

For Bert-1.67B, there is no GPU-CPU swap due to its unacceptably long swap-out/in cost. Contrary, D2D swap saves 23.4% GPU memory in total and was applied to transfer early stages (0-3) to the following stages (4-7). Recomputation saves GPU memory of stages (0-6) and introduces a 76.6% memory reduction. The combination of D2D swap and Recomputation brings the best performance, while D2D swap introduces a 19.5% speedup (Figure 7). The behavior of GPT-20.4B looks quite different. First, MPress chooses to apply GPU-CPU swap to transfer 242GB of model data from stages 0-7 to host memory, resulting in a 42.2% drop in GPU memory. Second, Recomputation reduces GPU memory by 51.2%. Third, D2D swap saves in total 38GB of GPU memory, the most across all the four sampled training jobs.

V. Hardware Insights

Even though GPU HBM provides extremely high bandwidth, it is challenging to meet the fast-growing model demands in the near future due to its expensive cost (e.g., the latest GPU has only 80GB HBM [9]). In comparison, CPU memory is cheaper and can scale better. Their price and capacity gap is due to the different manufacturing processes [6]. In fact, the new Grace-Hopper architecture already supports dedicated CPU-side memory for each GPU with high bandwidth (NVLink C2C) [8]. Therefore, MPress demonstrates the potential benefits and example usage of such architecture to address the memory wall challenge with low hardware cost.

To understand MPress' benefits under such new architecture, we devise a simple analysis which projects its ideal performance. We show that even with 96GB (HBM) + 512GB (Grace CPU memory) per-device memory of Grace-Hopper, training 175B GPT-3 still faces the OOM problem. However, MPress can address this problem. To completely hide the GPU-CPU swap cost, we expect the PCI-e bandwidth to exceed 140 GB/s for each GPU, more than double of Grace-Hopper's 64GB/s. Therefore, the D2D swap technique within MPress is still valid in this case for either saving 25% of wasted resources by Recomputation or avoiding paying 13% longer end-to-end training time introduced by swapping between the superchip and the PCI-e connected memory. Furthermore, it takes time to widely adopt new hardware technology. E.g., the latest GPU instance on AWS EC2 still uses DGX-2-A100 and its availability is very limited. Thus MPress can partially complement the limitations of contemporary hardware.

Finally, MPress helps the rethinking of memory architecture. Considering tensors' life spans and the cost/capacity/bandwidth trade-offs of memory technologies, it is beneficial to expand the memory hierarchy with more levels, where the fastest level stores the data immediately required for computation, while the slower levels store data with longer life spans. Each level can have different access bandwidths to further reduce the cost. In this case, MPress' planning algorithm III-D can be extended to judiciously assign various model tensors to proper levels.

VI. Related Work

There are a large body of related work aiming at addressing the GPU memory limitation of parallel DNN training. First, lossy data compression approaches that leverage quantization, sparsification, and mixed precision training can be a good candidate [17], [18], [24], [26]–[29], [36], [41], [45], but likely imposing negative impacts on training convergence and accuracy. Unlike this, `MPress` compacts the GPU memory usage and thus does not affect accuracy.

Similarly, ZeRO [48] removes the memory state redundancies across data-parallel processes by partitioning the model states instead of replicating them. Intra-operator parallelism (a.k.a model parallelism) splits the model vertically, partitioning the computation and parameters in each layer across multiple devices, requiring significant communication between each layer [59]. In contrast, `MPress` works with inter-operator parallelism, which supports larger model and batch sizes than data parallelism and demonstrates better usability than intra-operator parallelism. ZeRO-Offload [51] offloads optimizer computation and states from GPU to CPU, and is only suitable to computation-light optimizers such as Adam. However, `MPress` does not impose any constraint over training frameworks. ZeRO-Infinity [49] further extends the ZeRO-faimily by additionally leveraging the space of NVMe SSDs for supporting large models.

Recomputation proposals [16], [25], [33], [58] drop activation tensors after their last usage in the forward pass or when the GPU memory usage reaches a thresholds [38], [56] and later recompute them when needed in the backward pass. They fail to be applied to models with wider layers that occupy huge GPU memory or parameters whose recomputation is much expensive [31]. The more recent recomputation implementation has been demonstrated by state-of-the-art training systems such as GPipe [32] and ZeRO-Infinity [49].

Some existing work have already explored the possibility to swap tensors from GPU to the CPU memory with much larger capacity [31], [35], [40], [47], [53], [58]. For instance, TFLMS [40] and vDNN [53] swap out/in only feature maps according to the topological ordering over computation graph, while SuperNeurons [58] only considers the data of the convolution operators. L2L [47] aggressively supports very deep Transformer networks by keeping only one Transformer block in GPU memory at each time. Layrub [35] also adopts GPU-CPU swap to support wider networks within data parallel training. More recently, SwapAdvisor [31] explores the optimal GPU-CPU swap configuration by jointly taking into account the operator scheduling and memory allocation.

`MPress` is complementary to all the above memory compaction techniques such as swapping, recomputation, and offloading, and we extend them by making the best usage of GPU spare memory and the high-speed interconnects in inter-operator parallel training, and strategically exploring the chances to further combine all these optimizations together.

VII. Conclusion

`MPress` supports fast billion-scale model training on a single multi-GPU server. It adopts a novel D2D swap technique, which utilizes multiple NVLink interconnects to swap tensors to GPUs with spare memory and whose cost can be potentially hidden. It then employs recomputation and GPU-CPU swap when possible, to increase the D2D swap chances or complement the limited GPU swap space. Finally, `MPress` balances these three optimizations for better performance and model/batch size trade-offs. Evaluation with Bert and GPT demonstrates that `MPress` can either train larger models or deliver better training efficiency than baselines.

Acknowledgment

We thank the anonymous reviewers for their insightful comments. We also thank Ruichuan Chen, Youshan Miao, Jinhui Yuan, Teng Su, Sen Wang, and Olatunji Ruwase for their valuable suggestions and feedback. This work is supported in part by the National Natural Science Foundation of China under Grant No.: 62141216, 62172382 and 61832011, the Open Fund of PDL under Grant No.: WDZC20215250115, the USTC Research Funds of the Double First-Class Initiative under Grant No.: YD2150002006, the University Synergy Innovation Program of Anhui Province under Grant No.: GXXT-2022-045, and National Science Foundation under Grant No.: CAREER-2048044. We also thank the technical support from Shanghai HPC-NOW Technologies Co., Ltd. Cheng Li is the corresponding author.

REFERENCES

[1] "Bert from Google Research," https://github.com/google-research/bert.

[2] "Code of MPress," https://gitlab.com/MPressX/mpress.

[3] "Convnet-burden," https://github.com/albanie/convnet-burden, [Online; accessed July-2022].

[4] "DeepSpeed Code Repository," https://github.com/microsoft/DeepSpeed.

[5] "Distributed training of deep learning models on Azure," https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/ai/training-deep-learning, [Online; accessed July-2022].

[6] "High Bandwidth Memory," https://en.wikipedia.org/wiki/High_Bandwidth_Memory, [Online; accessed July-2022].

[7] "Introducing to P3.24xlarge of AWS," https://aws.amazon.com/about-aws/whats-new/2019/10/introducing-amazon-sagemaker-mlp3dn24xlarge-instances/.

[8] "NVIDIA Grace-Hopper Superchip Architecture," https://nvdam.widen.net/s/qjzrmfdn2j/nvidia-grace-hopper-superchip-architecture-whitepaper-v1.0, [Online; accessed July-2022].

[9] "NVIDIA H100 Tensor Core GPU Architecture," https://www.hpctech.co.jp/catalog/gtc22-whitepaper-hopper_v1.01.pdf, [Online; accessed July-2022].

[10] "Pytorch Homepage," https://pytorch.org/, [Online; accessed July-2022].

[11] "PyTorch Pretrained Bert," https://github.com/Meelfy/pytorch_pretrained_BERT.

[12] "Stanford Question Answering Dataset v1.1," https://rajpurkar.github.io/SQuAD-explorer/explore/1.1/dev/.

[13] "The Code Repo for PipeDream: Pipeline Parallelism for DNN Training," https://github.com/msr-fiddle/pipedream, [Online; accessed July-2022].

[14] "White Paper for DGX-1 with Tesla V100," https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf.

[15] "Wikipedia Dataset," https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2.

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

[17] 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.

[18] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[19] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," in *Proceedings of Advances in neural information processing systems*, 2012, pp. 1223–1231.

[20] 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.

[21] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia *et al.*, "Dapple: A pipelined data parallel approach for training large models," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 431–445.

[22] W. Gao, X. Fan, J. Sun, K. Jia, W. Xiao, C. Wang, and X. Liu, "Deep retrieval: An end-to-end learnable structure model for large-scale recommendations," *CoRR*, vol. abs/2007.07203, 2020. [Online]. Available: https://arxiv.org/abs/2007.07203

[23] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang, "Estimating gpu memory consumption of deep learning models," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1342–1352.

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

[25] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, "Memory-efficient backpropagation through time," *Advances in Neural Information Processing Systems*, vol. 29, pp. 4125–4133, 2016.

[26] 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.

[27] 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.

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

[29] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *arXiv preprint arXiv:1506.02626*, 2015.

[30] X. Han, Z. Zhang, N. Ding, Y. Gu, X. Liu, Y. Huo, J. Qiu, Y. Yao, A. Zhang, L. Zhang *et al.*, "Pre-trained models: Past, present and future," *AI Open*, vol. 2, pp. 225–250, 2021.

[31] 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*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1341–1355. [Online]. Available: https://doi.org/10.1145/3373376.3378530

[32] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

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

[34] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 463–479.

[35] H. Jin, B. Liu, W. Jiang, Y. Ma, X. Shi, B. He, and S. Zhao, "Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 3, pp. 1–26, 2018.

[36] 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.

[37] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *CoRR*, vol. abs/2001.08361, 2020. [Online]. Available: https://arxiv.org/abs/2001.08361

[38] 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.

[39] V. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," *CoRR*, vol. abs/2205.05198, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2205.05198

[40] T. D. Le, H. Imai, Y. Negishi, and K. Kawachiya, "Tflms: Large model support in tensorflow by graph rewriting," *arXiv preprint arXiv:1807.02037*, 2018.

[41] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[42] H. Mikami, H. Suganuma, P. U.-Chupala, Y. Tanaka, and Y. Kageyama, "Imagenet/resnet-50 training in 224 seconds," *ArXiv*, vol. abs/1811.05233, 2018.

[43] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–15. [Online]. Available: https://doi.org/10.1145/3341301.3359646

[44] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[45] L. Ning and X. Shen, "Deep reuse: streamline cnn inference on the fly via coarse-grained computation reuse," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 438–448.

[46] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (ACM SOSP 2019), Huntsville, Ontario, Canada, October 27-30, 2019*, 2019.

[47] 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.

[48] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.

[49] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476205

[50] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3505–3506.

[51] 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.

[52] Y. Ren, S. Yoo, and A. Hoisie, "Performance analysis of deep learning workloads on leading-edge systems," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2019, pp. 103–113.

[53] 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.

[54] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *CoRR*, vol. abs/1802.05799, 2018. [Online]. Available: http://arxiv.org/abs/1802.05799

[55] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *CoRR*, vol. abs/1909.08053, 2019. [Online]. Available: http://arxiv.org/abs/1909.08053

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

[57] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, "Efficient Transformers: A Survey," *arXiv:2009.06732 [cs]*, Mar. 2022.

[58] 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.

[59] M. Wang, C.-c. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303953

[60] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 595–610. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/xiao

[61] Y. Xu, H. Lee, D. Chen, B. Hechtman, Y. Huang, R. Joshi, M. Krikun, D. Lepikhin, A. Ly, M. Maggioni *et al.*, "Gspmd: General and scalable parallelization for ml computation graphs," *arXiv preprint arXiv:2105.04663*, 2021.

[62] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *CoRR*, vol. abs/1906.08237, 2019. [Online]. Available: http://arxiv.org/abs/1906.08237

[63] W. Zeng, X. Ren, T. Su, H. Wang, Y. Liao, Z. Wang, X. Jiang, Z. Yang, K. Wang, X. Zhang, C. Li, Z. Gong, Y. Yao, X. Huang, J. Wang, J. Yu, Q. Guo, Y. Yu, Y. Zhang, J. Wang, H. Tao, D. Yan, Z. Yi, F. Peng, F. Jiang, H. Zhang, L. Deng, Y. Zhang, Z. Lin, C. Zhang, S. Zhang, M. Guo, S. Gu, G. Fan, Y. Wang, X. Jin, Q. Liu, and Y. Tian, "Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation," *CoRR*, vol. abs/2104.12369, 2021. [Online]. Available: https://arxiv.org/abs/2104.12369

[64] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," 2022.