Parallel Training of Pre-Trained Models via Chunk-Based Dynamic Memory Management

Jiarui Fang, Zilin Zhu, Shenggui Li, Hui Su, Yang Yu, Jie Zhou, and Yang You

Abstract—The pre-trained model (PTM) is revolutionizing Artificial Intelligence (AI) technology. However, the hardware requirement of PTM training is prohibitively high, making it a game for a small proportion of people. Therefore, we proposed PatrickStar system to lower the hardware requirements of PTMs and make them accessible to everyone. PatrickStar uses the CPU-GPU heterogeneous memory space to store the model data. Different from existing works, we organize the model data in memory chunks and dynamically distribute them in the heterogeneous memory. Guided by the runtime memory statistics collected in a warm-up iteration, chunks are orchestrated efficiently in heterogeneous memory and generate lower CPU-GPU data transmission volume and higher bandwidth utilization. Symbiosis with the Zero Redundancy Optimizer, PatrickStar scales to multiple GPUs on multiple nodes. The system can train tasks on bigger models and larger batch sizes, which cannot be accomplished by existing works. Experimental results show that PatrickStar extends model scales 2.27 and 2.5 times of DeepSpeed, and exhibits significantly higher execution speed. PatrickStar also successfully runs the 175B GPT3 training task on a 32 GPU cluster. Our code is available at https://github.com/Tencent/PatrickStar.

Index Terms—Deep learning, parallel computing, distributed system, NLP

1 INTRODUCTION

The pre-trained models (PTMs) such as ViT [1], BERT [2], GPT [3] and have become a milestone in the field of Artificial Intelligence (AI). Although successful pre-training efforts exist on models of trillions of parameters [4], [5], [6], it still has a long way to go before making PTM accessible to everyone. The application of PTM has two phases: pre-training and fine-tuning. Although the computation patterns are the same, there is a huge gap in the accessible hardware quality between the two stages. The pre-training phase is trained from scratch on a super large-scale dataset with countless iterations and is highly time-consuming. Therefore, it is usually conducted on supercomputers using hundreds of GPU nodes connected with high-speed network fabric. Such high-quality hardware is only affordable for a small proportion of people in the AI community. The PTMs are often fine-tuned on much smaller downstream application-specific datasets in the fine-tuning phase. Most people can access the hardware quality in this phase is much lower than in the pre-training phase, usually a single node equipped with multiple GPUs. Therefore, improving the model size available on these accessible low-end small-scale hardware is the critical path to the democratization of PTM. Unfortunately, the existing work pays little attention to bridging the hardware gap between these two stages.

Heterogeneous training [6], [7], [8] is the most promising solution to lower the hardware requirements of fine-tuning phase. This approach exploits the heterogeneous memory space, including GPU and CPU memory, and only moves data to the required device if necessary. However, all of these works reported their results on multiple nodes of the DGX-2 supercomputer. Each node supplies sufficient memory resources, i.e., 8x32GB GPU memory 1.5TB CPU memory and 3.84TB NVMe SSDs. The memory configuration far exceeds the average standard in data centers and cloud computing platforms. The performance is largely compromised when adopting these systems on commonly accessible hardware.

We observe two types of tensors managed during PTM training: the model data consists of parameters, gradients, and optimizer states whose footprints are related by the model structure definition; and the non-model data consists of the intermediate tensors generated by operators. The non-model data dynamically change according to the configuration of training tasks, such as batch size. Model data and non-model data compete for GPU memory with each other. Without considering non-model data volume changing inside an iteration, the existing solutions [6], [8] statically partition the model data between CPU and GPU memory in advance of the training, and the partition scheme is constant to various training configurations. Such a static partition strategy leads to several problems. First, the system will crash when the GPU or the CPU memory is insufficient for its corresponding model data requirements, even if there is still memory available on the other devices at the time. Second, communication is inefficient when transferring data among memory spaces in the granularity of the tensor.

PatrickStar overcomes these shortcomings by dynamically managing model data in a chunk manner to use the heterogeneous memory. First, instead of partitioning the
heterogeneous space layout of model data before training and then keeping it unchanged during training. It adjusts the layout in real-time according to runtime information before operator execution during training. For this purpose, it adopts a runtime memory tracing method and a smart eviction strategy to minimize dynamic data transfer. Second, we organize the model data tensors in chunks which are blocks of continuous memory of the same element size. Data transmission in the system is based on chunks leading to high utilization of CPU-GPU and inter-GPU bandwidth. When scaling to multiple GPUs, chunk-based memory management can be symbiotic with data parallelism using zero redundancy optimizer [5] (ZeRO-DP).

Our main contributions include: 1) We built an innovative DNN training system from scratch called PatrickStar, consisting of two modules: chunk-based memory manager and dynamic memory manager. 2) The chunk-based memory management is naturally symbiotic with zero redundancy optimizer data parallelism. The chunk-based communication pattern leads to higher CPU-GPU and inter-GPU bandwidth utilization. 3) The dynamic memory management can improve memory efficiency and reduce CPU-GPU communication overhead. 4) We evaluate our system on two GPU clusters, an 8x V100 GPUs 240 GB memory node and an 8x A100 1TB memory node. PatrickStar trains GPT-like models 2.27 and 2.5 times the maximal model scale of deepspeed and trains faster under the same model scale. 5) PatrickStar has higher computing and memory efficiency than DeepSpeed and achieves superlinear scalability on 8x GPUs. 6) Without significant code modifications, PatrickStar successfully runs the 175B GPT3 training task on 32 GPUs using ZeRO-DP only.

2 BACKGROUND OF PTM

Nowadays, PTMs use transformer [9] structures due to their superior performance. And to speed up convergence, adaptive gradient-based optimizer ADAM are the de facto choice in transformer model training. Taking advantage of the tensor-core component on GPUs, PTM training is usually conducted in a mixed-precision way, which requires that the parameters and gradients of the model use the half-precision floating-point (fp16) format in forward propagation (FWD) and backward propagation (BWD), and the single-precision floating-point (fp32) format during the parameter updating [10]. We list the types of tensors used in PTM training. **Param fp16**: the model parameters of fp16 type used in FWD and BWD. **Grads fp16**: the gradients of param fp16, generated in BWD. **Optimizer states (OS)**: state parameters required by ADAM optimizer, including momentum fp32 and variance fp32. **Activations**: intermediate output tensors of the operators, also known as feature maps, produced in FWD and BWD. **Temporary data**: buffers used during operator computation and memory overhead of the software framework. We refer to activations and temporary data as non-model data, and the others as model data.

The PTM training is viewed as a directed graph. As shown in Fig. 1, the circular nodes represent tensor sets (param fp16, grad fp16, param fp32, momentum fp32, variance fp32), and rectangular nodes represent operator sets, including FWD and BWD. Parameter Updating. Edges of the graph represent the data flow, and the weight of the edge is the overall amount of data (in bytes), assuming the model has M parameters. Let us first theoretically estimate the memory requirements of PTM training. For a M-parameter model, the FWD and BWD require 4M bytes space for param fp16 and grad fp16 tensors. For the Param updating, 4M (momentum fp32) + 4M (variance fp32) + 4M (param fp32) = 12M bytes are required. The updated param fp32 is converted to param fp16 to participate in the computation of the next iteration. In addition to grad fp16, the overall training process requires 2+2+14=18M bytes of memory space. For a GPT model with 2 billion (2B) parameters, 36 GB of GPU memory is required, more than 32 GB of V100 overall GPU memory. And we have not taken the memory usage of non-model data during the training process into account yet. The GPU memory is often more scarce.

3 RELATED WORKS

Here list the three major lines of work of PTM training.

*Parallel Training*. The most popular parallel training method is data parallelism [11], [12] (DP), which keeps a copy of the model on each GPU and partitions inputs and activations among multiple GPUs. DP fits the entire model data in the GPU memory, which is no longer valid in the PTM training. Parallel training PTM, therefore, has to partition the model data across multiple GPUs. A set of works are proposed to adapt DP on PTM training. The Zero Redundancy Optimizer (ZeRO-DP) [5] eliminates memory redundancies of model data by partitioning them in layers among multiple GPUs, therefore using the total aggregate GPU memory capacity of a cluster. It applies extra communications to collect remote device parameters.

Model parallelism (MP) [4], [13] is also revisited in PTM training recently. MP splits the multi-dimensional tensors of model data along one or more specific dimensions and then distributes them on multiple devices. In this case, the activations have to be accommodated on each device redundantly. MP has poorer computational efficiency than DP due to additional communication of activations in FWD and BWD. As for memory efficiency, MP reduces the model data footprint proportional to the number of workers while not reducing the non-model data footprint. In addition, model parallelism’s computing and communication patterns are specific to model structures and require a tremendous initial engineering effort for different models.
Pipeline Parallelism (PP) \[14, 15, 16\] is another approach for distributed training that divides the model’s layers into stages so that the model can process in a pipeline manner. As one stage completes the FWD or BWD for a micro-batch, the activation tensors are communicated to the next stage, and the gradients of activations are passed to the previous stage in the pipeline. Although PP communicates less volume than the DP and MP, it has pipeline starting overheads and load imbalance issues.

**Heterogeneous Training.** Different from distributing the model data on the homogeneous GPU memory space, heterogeneous training utilizes heterogeneous memory space consisting of both CPU and GPU memory. L2L \[7\] accommodates the model data in CPU by default, and the GPU memory is populated only with the executing layer’s model data at any given moment in training. However, it leads to frequent CPU-GPU data movement, in units of tensors. Moreover, L2L is implemented on one GPU system. ZeRO-Offload \[8\] advanced the ZeRO-DP by offloading the OS data and grad fp16 on CPU and executes ADAM on CPU and transferred grad fp16 and param fp16 tensors between CPU and GPU during training.

**Activation Checkpointing and Offloading.** In addition to reducing GPU model data, a set of work reduces the GPU memory consumption of activations. The activation checkpointing, a.k.a. activation rematerization, first proposed by Chen et al. \[17\] and widely investigated in work \[18, 19\] to trade off activation memory with more computation and is implemented in PyTorch \[20\]. Activation offloading \[21, 22, 23, 24\] is an alternative approach that consists of off-loading some of forward activations from GPU to CPU.

### 4 Motivations

PTM goes through two stages: pre-training and fine-tuning. They share the same computation pattern but different application scenarios. In the pre-training stage, AI supercomputers are widely applied. Parallel training work of Section 3 is mainly focused on this stage. The hardware settings of related works \[4, 6\] are the latest AI supercomputer with 32 high-end DGX-2H nodes. While expanding the pre-training model scale is only a game for a small proportion of people, the fine-tuning stage must involve most people playing with their datasets. AI developers commonly use equipment from data centers or self-purchased small-scale servers. Their computing scale and storage capability are lower than AI supercomputers. Besides improving hardware qualities, developing a software system to bridge the computing power gap between pre-training and fine-tuning is the key to democratizing PTM.

The most successful effort to facilitate PTM fine-tuning is the heterogeneous training approach. DeepSpeed \[25\] integrated with optimizations of ZeRO-Offload and Zero-DP \[5, 8\] trains a 13B model on a single V100 of a DGX-2H server. They statically manage model data in heterogeneous memory space such that the param fp16 data are stored in GPU while the grad fp16 and OS data are stored in CPU. The param fp16 and grad fp16 data, overall 4M bytes, are moved between CPU and GPU during each iteration. It is worth noting that the experiment environment of ZeRO-Offload has 1.5TB CPU memory, which undertakes most of the model data storage. However, when decreasing the amount of CPU memory, the maximum model scale drops a lot, e.g., deploying it with 240 GB CPU memory, the maximum model scale is lowered to 4B. Moreover, its computing efficiency decreases as the model scale increases, pointing to the direction for further improvements. For example, 47 Tflops can be achieved on a 1B model whereas only 33 Tflops achievable on a 4B model.

ZeRO-Offload is relatively fragile and inefficient on lower-quality computing hardware. During FWD and BWD, GPU memory has to accommodate peak non-model data in addition to overall param fp16. During param updating (ADAM), the CPU has to accommodate all of the model data. The memory of model-data depends on the configuration of the model structure, which we call model-related configuration. However, the memory of non-model data is not only related to the model-related configuration but also related to the task-related configuration, e.g., batch size, the activation optimization scheme. It should be noted that the memory consumption of non-model data cannot be ignored. As shown in the Fig. 2, even if the checkpointing and offloading optimization are used, there is still a peak memory consumption of nearly 5GB for this task, and activation offloading has been slowing down the overall training speed. Using the above static partition strategy, the model-related configuration is strongly affected by the task-related configuration. Once the param fp16 volume exceeds the size of GPU memory limitation subtracted by peak non-model volume, the system will crash with OOM, even if the memory shortage only lasts a short period around the non-model memory peaking moment. Another issue comes from bandwidth utilization. ZeRO-Offload and L2L \[7\] transfer data among different memory spaces in the granularity of the tensor. Bandwidth is sometimes wasted when moving tensors of small message sizes \[26\]. Bandwidth underutilization, therefore, results in lower transmission efficiency and longer execution time. This phenomenon is especially obvious for PCI-e where the CPU-GPU is connected.

The above phenomena motivate us toward a new heterogeneous training system design that focuses on 1) improving communication bandwidth utilization efficiency, and 2) optimizing memory utilization. First, instead of statically splitting the model data into GPU and CPU before training, we dynamically arrange them into GPU and CPU before the execution of each operator. Briefly, when there are too much non-model data, the
param fp16 tensors not used in the upcoming operators can be moved to the CPU, so that the memory use efficiency is optimized and the trainable model scale is enlarged. Second, in replacement of small tensor transmission, we adopt a large granularity memory chunk transmission method to optimize both the CPU-GPU and inter-GPU transmitting bandwidth, which has traditionally been the bottle-neck of heterogeneous training systems [7], [8]. According to the PyTorch Team’s observation [12], communications are more efficient on large tensors (to saturate the bandwidth, it is suggested to transmit over 20-M elements each time), but with ZeRO-DP, 97% of the parameter tensors are less than 2.6-M when training a 4B GPT [27] model on 8 GPUs. Moreover, introducing MP further reduce the model tensor sizes and lower the bandwidth utilization. To address this issue, we came up with chunk-based memory management as inspired by PyTorch-DPP [12], which uses Gradient Bucketing to concat gradient tensors into a memory chunk to improve the inter-GPU bandwidth utilization.

5 DESIGN OVERVIEW

As shown in Fig. 3, PatrickStar is the middleware of PyTorch and heterogeneous memory space. The software consists of two modules: a Chunk-based Memory Manager (CMM) and a Dynamic Memory Manager (DMM). CMM organizes PyTorch tensors into chunks to improve communication bandwidth (elaborated in Section 7). The Communication Executor of DMM dynamically arranges the layout of ChunkLists in CPU and GPU during training and data transmission volume is reduced and model scale is improved (elaborated in Section 6). The modules not in black color are newly introduced by PatrickStar and they bring almost no extra overhead (elaborated in Section 8). The Tensor-Chunk Mapper (blue color) is built during the PyTorch model initialization before training and remains unchanged during training. The Runtime Memory Tracer (yellow color) is executed only once during the first training iteration. The states of Stateful Tensors (red color) of CMM (yellow color) is executed only once during the first training iteration. The states of Stateful Tensors (red color) of CMM (yellow color) is executed only once during the first training iteration. The states of Stateful Tensors (red color) of CMM (yellow color) is executed only once during the first training iteration. The state is that activation tensors whose life cycles do not overlap may reuse the same memory footprint in the first iteration. We call the first iteration as the warm-up iteration as the warm-up iteration and the remaining iterations of the training phase are working iterations. At the time an operator starts, which is called as moment $i$ in this paper, DMM obtains the real-time GPU memory consumption $R_i$. This can be obtained with almost no overhead via the API provided by DL frameworks, i.e., cuda.memory_allocated in PyTorch. Meanwhile, the system can accurately know the volume of model-data memory $C_i$ in use, therefore, the non-model data memory consumption $N_i$ is figured out with $R_i - C_i$. This traced memory consumption $N_i$ obtained during the warm-up iteration will be used later in working iterations because the computing patterns are the same among iterations. Fig. 2 is depicted using the proposed tracer.

6 DYNAMIC MEMORY MANAGEMENT

This section introduces the DMM module mentioned in Fig. 3. It rearranges model data layout around heterogeneous spaces before operator execution. Therefore, the system can be used to train a larger-size model. Besides from the improvement of the model scale, the DMM consists of three innovations that can further reduce the CPU-GPU communication overhead as much as possible. First, DMM designs a runtime tracer to collect memory statistics (Section 6.1). Second, we can switch the computing device instead of CPU-GPU data moving by a device-aware operator placement optimization (Section 6.2). Third, for the data must be moved, DMM uses a smart eviction strategy (Section 6.3) to reduce transmission volume.

6.1 Runtime Memory Tracer

DMM obtains the statistics of non-model data memory by a runtime memory tracer. The runtime tracing is necessary since it is hard to get the footprint of non-model data via theoretical estimations. Even if the shape of each activation tensor is precisely available, accumulation of all activation tensor sizes is not equal to realistic memory consumption. First, GPU memory fragmentation and the CUDA context memory are hard to measure before execution. Second, the dynamic framework PyTorch does not provide an interface to capture the life cycle of every activation tensor before execution. Although there is research [28] on this topic, the method proposed is coupled with the specific memory allocator design of the target frameworks. Further complicating the theoretical estimations is that activation tensors whose life cycles do not overlap may reuse the same memory space [29], [30]. Third, the estimation method is affected largely by activation checkpointing and offloading.

Fig. 4 shows how we measure non-model data memory footprint in the first iteration. We call the first iteration as the warm-up iteration and the remaining iterations of the training phase are working iterations. At the time an operator starts, which is called as moment $i$ in this paper, DMM obtains the real-time GPU memory consumption $R_i$. This can be obtained with almost no overhead via the API provided by DL frameworks, i.e., cuda.memory_allocated in PyTorch. Meanwhile, the system can accurately know the volume of model-data memory $C_i$ in use, therefore, the non-model data memory consumption $N_i$ is figured out with $R_i - C_i$. This traced memory consumption $N_i$ obtained during the warm-up iteration will be used later in working iterations because the computing patterns are the same among iterations. Fig. 2 is depicted using the proposed tracer.
6.2 Device-Aware Operator Placement

PTM Operators exhibit different computing and memory access characteristics. For example, the computing-intensive operator like the Linear must be executed on GPU. Memory-intensive operators like the element-wise ones in ADAM can be executed on both CPU and GPU. Note that Memory-intensive operators take up a small part of the overall time. An intelligent layout can improve end-to-end performance by reducing the CPU-GPU communication volume, even if it slightly increases their own part’s execution overhead. PatrickStar designs a device-aware placement for optimizer states (OS). GPU margin space is the remaining space after removing peak non-model data and param fp16 from the overall GPU memory. The peak non-model data volume is provided by the memory tracer. We design an optimal eviction strategy to achieve minimal CPU-GPU data transmission. In the warm-up iteration, the tracer records a visit list for each tensor which contains both the moment and target device of the access. During working iteration, at the moment \( j \), we can search the visit lists to find tensors not needed on this computing device from moment \( j \) to \( j+1 \). A greedy algorithm picks the longest future reference tensors on the computing device from the list whose volume is just larger than the eviction volume. It is implemented in \( O(\log^2 N) \) by traversing the visit list of all tensors and binary searching the next moment to be used, where \( N \) is the tensor count, and \( T \) is the moment count. Our proposed strategy can be viewed as Belady’s OPT algorithm [31], [32], which replaces the buffer page with the longest future reference distance as the cache replacement policy. We will extend the eviction strategy from tensor to chunk in CMM (Section 7.1.2).

As for the warm-up iteration, there is no future non-model statistics at a specific moment. To avoid out of GPU memory, we use only a small proportion, by default 20%, of GPU memory to store model data. At this time, it simply evicts tensors in the order of the tensor list.

6.3 Eviction Strategy

In a working iteration, the eviction strategy dynamically determines how to move model data between CPU and GPU, depending on the runtime statistics of the tracer. As shown in Fig. 4, to let the operator execute during the period \( j \) to \( j+1 \), DMM has to prepare at least max \((N_{j+1}, N_j)\) memory space for non-model data. Consequently, the available memory space for model data \( A_{j+1} \) of the next moment \( j+1 \) is derive by GPU memory limitation \( L_{max}(N_{j+1}, N_j) \). Assuming the volume of model data not on GPU but required by the current operator is \( O_j \), then if \( C_j + O_j \) is larger than \( A_{j+1} \), \( A_{j+1} - C_j - O_j \) of data has to be evicted to CPU. We call the volume of data to be evicted as eviction volume.

We design an optimal eviction strategy to achieve minimal CPU-GPU data transmission. In the warm-up iteration, the tracer records a visit list for each tensor which contains both the moment and target device of the access. During working iteration, at the moment \( j \), we can search the visit lists to find tensors not needed on this computing device from moment \( j \) to \( j+1 \). A greedy algorithm picks the longest future reference tensors on the computing device from the list whose volume is just larger than the eviction volume. It is implemented in \( O(\log^2 N) \) by traversing the visit list of all tensors and binary searching the next moment to be used, where \( N \) is the tensor count, and \( T \) is the moment count. Our proposed strategy can be viewed as Belady’s OPT algorithm [31], [32], which replaces the buffer page with the longest future reference distance as the cache replacement policy. We will extend the eviction strategy from tensor to chunk in CMM (Section 7.1.2).

As for the warm-up iteration, there is no future non-model statistics at a specific moment. To avoid out of GPU memory, we use only a small proportion, by default 20%, of GPU memory to store model data. At this time, it simply evicts tensors in the order of the tensor list.

7 CHUNK-BASED MEMORY MANAGEMENT

The mechanism of CMM on a single GPU is shown in Fig. 5. PatrickStar improves the bandwidth utilization of existing heterogeneous training by managing the model data into chunks (details in Section 7.1). The circles in the figure represent the parameter elements and are arranged in memory chunks. When the GPU computation of an operator is triggered (the light-green part of the DNN in the right figure), PatrickStar moves the required chunks to GPU memory. Moreover, it is scalable to multiple GPUs combining with ZeRO-DP [5] (details in Section 7.2).

7.1 Design on a Single GPU

Before training, the CMM builds the Tensor-Chunk-Mapper by constructing the mapping schema between tensor and chunk based on the neural network structure. During training, it takes over the memory access of the PyTorch and directs it to the memory piece in ChunkLists.

7.1.1 Preprocessing Stage

We propose an efficient chunk-tensor mapping with three features: 1) increase the locality of tensor access. 2) reduce peak memory usage. 3) be parallel-friendly. Our mapping schema is derived from the following procedure. Chunks are categorized into four types according to types of tensors in model data, i.e., param fp16 list, param fp32 list, momentum list, and variance list, overall 14M bytes (\( M \) is parameter number). Chunks contain the same elements, so different chunks in the same type can reuse the same memory space. Especially, PatrickStar does not allocate a grad fp16 list. The grad fp16 tensors can reuse chunk space of the param fp16 list, as elaborated in Section 7.1.2, PatrickStar shrinks the memory footprint compared to ZeRO-Offload [8] whose minimal model-data memory footprint is 18M bytes as mentioned in Section 2. Additionally, it allocates extra GPU memory holding the gradients to be moved to the CPU, while PatrickStar eliminates this overhead.

We first build the param fp16 chunk list in the order of model initialization. In this way, when accessing a tensor in a chunk, the neighboring tensors used soon have already been in the same chunk. Therefore, the access locality is well guaranteed. For OS tensors, their corresponding chunk lists are mapped in the same way as param fp16. As a result, when scaling to multiple processes, four chunk lists are split at the same position and the chunks used by ADAM are in the local process and avoid cross-process communication. If there exist tensors breaking the locality with irregular access patterns, which are not common in PTM, PatrickStar supports labeling them not managed in chunks.
Algorithm 1. Access Tensor

1: Function FetchRemote(chunk_id)
2:   Require: Chunks (the chunk list), CachedChunks (the chunk cached), rank;
3:   if Chunks[chunk_id] in CachedChunks then
4:       return;
5:   end
6:   host_rank ← get_host_rank(chunk_id);
7:   if rank != host_rank then
8:       alloc_gpu_memory for Chunks[chunk_id].
9:   end
10:  move Chunks[chunk_id] to GPU if necessary;
11:   BCAST Chunks[chunk_id] from host_rank;
12:   CachedChunks.add(chunk_id);
13: end
14: Function Access(param, comp_dev)
15:   Require: Chunks (a vector of chunks);
16:   chunk_id = get_chunk_id(param);
17:   if is_distributed() then
18:       FetchRemoteChunk(chunk_id, comp_dev));
19:   end
20:   move Chunks[chunk_id] to comp_dev if necessary;
21:   param.data ← a piece of Chunks[chunk_id];
22:   old_state ← the state of param.data;
23:   if old_state is FREE then
24:       param.data.zero();
25:   end
26:   set the state of param.data as COMPUTE;
27: end

Algorithm 2. Release Tensor

1: Function ReleaseRemote(chunk_id, training_stage, status)
2:   Require: Chunks (the chunk list), CachedChunks (the chunk cached);
3:   host_rank = get_host_rank(chunk_id);
4:   if training_stage is FWD and at least a tensor in Chunks[chunk_id] is not HOLD_AFTER_FWD then
5:       move Chunks[chunk_id] to GPU;
6:       REDUCE(Chunks[chunk_id], AVG);
7:   end
8:   if rank != host_rank then
9:       release Chunks[chunk_id] and set its tensors to FREE;
10:  CachedChunks.remove(chunk_id);
11: end
12: Function Release(param, training_stage, target_state)
13:   chunk_id = get_chunk_id(param);
14:   set the state of param.data as target_state;
15:   if is_distributed() then
16:       ReleaseRemote(chunk_id, training_stage, target_state);
17:   end
18:   param.data ← dummy tensor
19: end

7.1.2 Training Stage

To use the DMM in combination with CMM during training, we need to extend the eviction strategy from tensor-based to chunk-based. Tensors managed by CMM are stateful as shown in Table 1. It has five states; the last three are called HOLD-like states. The COMPUTE state indicates the tensor is about to be computed on a specific computing device, e.g., CPU or GPU. The HOLD-like states indicate that the tensor is not involved in computation right now, but its payload must be maintained in memory, either CPU or GPU. The state of the param fp16 tensor is set to HOLD after being initialized. The states of all its tensors determine a chunk’s location in heterogeneous space. When all tensors of a chunk are in the FREE state, the chunk’s memory space can be reused by other chunks or released. If any of the tensors of a chunk is in COMPUTE state, the chunk must be located on the required computing device. If none of its tensors is in COMPUTE and at least one of its tensors is in a HOLD-like state, the chunk may be in any place of the heterogeneous memory space. The chunk eviction strategy can be adapted from the eviction strategy proposed in Section 6.3. We build visit lists for each chunk, containing the access time and device for each access of each tensor inside the chunk. When the target memory is full, we evict the longest future reference chunks which only contain HOLD-like tensors in the memory.

The chunk eviction problem can be defined as the Optimal Cache Replacement problem: consider the management GPU memory space in T moments given advance knowledge of GPU chunk requests $R_0, R_1, ..., R_{T-1}$ from a set $\Omega$ of chunks stored in CPU memory. Let $S_t \subset \Omega$ denote the set of chunks stored in the GPU just before $R_t$ is requested. If $R_t \notin S_t$ then $S_{t+1} = S_t$. Otherwise, a decision must be made to evict one chunk from the cache, which is replaced by $R_t$. The objective is to maximize the number of hits: $\sum_{t=0}^{T-1} I(R_t \in S_t)$. A rigorous proof that eviction of the longest future reference chunk yields the optimal cache effectiveness can be
Fig. 6. The state transition diagram of a param fp16 tensor.

found in references [33], [34]. The time complexity of eviction is \(O(\log^2 |\Omega|)\), where \(|\Omega|\) is the chunk count.

Before operator execution, CMM uses Algorithm 1 to access the parameters from chunks and dynamically adjust the chunk-based model data layout using DMM. In line 20, the system must place the chunk containing the parameter in need on the target device, which may evict chunks from the device if its memory is full. After Access, the state of the tensor is converted to COMPUTE (Line 26). When the operation execution is finished, we release the tensor using Algorithm 2. During FWD, the training stage is set as FWD, and target_state is HOLD_AFTER_FWD. During FWD, the training stage is set as BWD, and target_state is HOLD_AFTER_BWD. After Release, the tensor states are converted to HOLD-like, so the tensor can be evicted to other devices if necessary. The state transfer of tensors has almost no overhead because it just needs to change the value of a few of variables.

Fig. 6 shows the state transition of a param fp16 tensor during training. After all operators finished FWD, the states of all param fp16 tensors are reset to HOLD to ensure the correct execution of BWD. This is because the active checkpointing optimization [17] will conduct FWD computation between two checkpoints during BWD. In this way, if the HOLD states generated by FWD and BWD are not distinguished, we cannot know whether all the tensors in the chunk have finished BWD. Before ADAM computing, OS tensors are set to COMPUTE. During computing, grad fp16 chunks are converted to fp32 on the fly to save memory. After computing, the updated param fp32 tensors and used OS tensors are set to HOLD. When all tensors in a param fp32 chunk are in the HOLD state, param fp32 chunks are copied into the corresponding param fp16 chunk.

CMM provides an optimization option to reuse chunks during BWD. Fig. 7 shows how to reuse param fp16 chunks for grad fp16 tensors. Before the BWD computing of an operator, we access the param fp16 tensors and change their states to COMPUTE. During the BWD computing of the operator, grad fp16 tensors are generated in a temporary memory space. After the BWD computing of the operator, since param fp16 is no longer needed, we copy the grad fp16 data from the temporary memory space to the memory space of the corresponding param fp16 tensors and change the tensor state to HOLD_AFTER_BWD. Given that multiple operators may share parameters, we use a reference counter to indicate the last time the tensor is accessed.

### 7.2 Scaling to Multiple GPUs

PatrickStar uses ZeRO-DP [5] to scale training to multi-GPU via multiple-processing. Assume the number of processes is \(nproc\). Each process takes care of a single GPU. In a multi-node system, where each node has one CPU and multiple GPUs, the processes inside a node share the CPU of the node. Therefore, the local memory space of a process consists of the memory of a single GPU and \(1/nproc\) of overall GPU memory space. ZeRO-DP reduces memory requirements by \(nproc\) times compared to DP by sharding chunks among processes. The chunks kept by the process are named local chunks and the rank of the process is host_rank. The chunks kept by the other processes are remote chunks. Before computing, a process may fetch the remote chunks from the other processes. In this way, the chunk is replicated on all processes. After it uses all parameters of the chunk for computing, the memory space of the remote chunk can be released. Distributed logic is added in Algorithm 1 (line 17) and Algorithm 2 (line 19).

The FetchRemote of Algorithm 1 shows the method of fetching remote chunks. If a process finds chunk to be fetched is a remote chunk, when its rank is not equal to the host_rank, it needs to allocate memory space for the chunk. Then the host_rank process broadcasts its local chunk data to other processes. We use CachedChunk to record the chunks currently replicated on all processes.

The ReleaseRemote of Algorithm 2 shows the method of releasing remote chunks. When the states of all tensors in a chunk are all HOLD_AFTER_FWD/BWD after FWD/BWD operator is finished, the tensors in the remote chunks are set to FREE, and the remote chunks are released and removed from CachedChunks. Especially during BWD, we need an extra reduce operation to average the gradients among processes and store the results on the target process. The CachedChunk is updated after chunk released.

Fig. 8 shows a snapshot before the FWD computing of layer 0 on 3 processes. The remote chunk is represented by dashed box, and the local chunk is represented by solid box. The green arrow in the figure indicates the direction of chunk transmission. The host_rank of chunk 0,1,2 is 0,1,2. Before FWD computing layer 0, process 1, and 2 find chunk 0 not resident in local memory space. Therefore, a broadcast is triggered to fetch remote chunks from rank 0. After collecting remote chunks, all processes will have their copy of chunk 0, and all tensor states in the remote chunks are set to HOLD. When we are working on L2 and L3, since chunk 0
is in the CachedChunks, FetchRemote will not be executed. After we finish computation on L3, chunk 0 will be released on process 1 and 2 and removed from CachedChunks. Later, before computation of L4, chunk 1 is transmitted from process 1 to process 0 and 2.

During distributed heterogeneous training, data consistency is always guaranteed because there is no asynchronous write to duplicated model data. Model data is written in two cases: 1. gradient reduction (line 9 in Algorithm 2) and 2. param fp16 updating during ADAM. In the first case, duplicated grad fp16 are updated synchronously. In the second case, there is no duplicated data between multiple processes.

8 Discussion

As shown in Fig. 9, PatrickStar is competent in two situations where ZeRO-Offload fails. In the case of GPU not enough (left figure), for ZeRO-Offload, when the overall size of param fp16 tensors and peak size of non-model data exceeds GPU memory, unsatisfied GPU memory requirement crashes the system. With the help of the eviction strategy, PatrickStar makes it work by evicting chunks supposed to be in unsatisfied GPU memory to CPU efficiently. If CPU memory is not enough (right figure), there exist unsatisfied CPU memory requirements in ZeRO-Offload. In PatrickStar, the margin space on GPU can accommodate unsatisfied OS chunks to make the system work smoothly and efficiently.

It should be emphasized that PatrickStar does not introduce scheduling overhead to the training process. In Fig. 3, the Communication Executor, ChunkLists, and the green modules replace the tensor-based data transmission method in ZeRO-Offload, thus significantly increasing the system efficiency. As the tensor state updating has almost no cost, the additional overhead is mainly introduced from the runtime memory tracer of DMM. Due to the conservative use of GPU memory (20%) and a naive cache strategy, non-model data CPU-GPU transmission volume is higher than the working iterations. As shown in Fig. 10, on average the cost of warm-up iteration is 2.25x and 4.56x times larger than working iterations on two hardware platforms. For example, in the 10B (bs12)-8 GPU case, the warm-up iteration time is 19.0 sec, and the average working iteration time cost is 10.1 sec. Because the warm-up iteration also benefit from other modules' optimizations, its overhead is lower than the ZeRO-Offload average iteration time, which is 20.8 sec for this case. When training for 100 iterations, the extra cost brought by warm-up is only 0.9%. Since a training/fine-tuning process usually requires hundreds of thousands of iterations and lasts for hours to days, an extra cost brought by a single warm-up iteration is negligible in the entire training phase.

9 Evaluation

9.1 Evaluation Methodology

Testbed. We conduct our experiments on three GPU clusters. The first one is the cloud computing system called YARD. It is a node configured with 8x 32GB V100 GPU and a 12-core CPU with 240 GB CPU memory. The second one is called SuperPod [35]. We use a node consisting of 8x 40GB A100 GPU and 192-core CPU with 1TB CPU memory. For both clusters, GPUs are connected through NVLink. The third one is called HAI-PCIe. It is a node consists of 8x 40GB A100 and 128-core CPU with 512 GB CPU memory. GPUs are connected with PCI-e.

Workloads. Same as related work, the PTM tasks used for evaluation are GPT-2 like transformer-based models. The model configuration is shown in in Table 2.

Baselines. We compare the PatrickStar with PyTorch DDP [12] and the state-of-the-art (SOTA) heterogeneous training solution using DeepSpeed [25] v0.5.3, it has integrated optimizations from the works [4], [5], [8]. The configuration of Deepspeed is adopted from the official example. We use the zero3 stage, indicating using ZeRO-Offload/Infinity optimization. For all the three software, we apply activation checkpointing. We pick the best performance with and without activation CPU offloading for PatrickStar and DeepSpeed. Here performance is measured with throughput, which is
calculated by dividing the floating-point operation numbers with the average end to end execution time of a iteration. We update the parameters at every iteration, which is more practical in real scenarios. We test with various batch sizes per GPU, including 4, 8, 16, 32, 64. The throughput is computed using the overall floating-point operations numbers (#FLOP) dividing by the averaged end-to-end elapse of an iteration. The statistics of #FLOP is consistent between DeepSpeed and PatrickStar. Results are evaluated on 100 working iterations. The model is randomly initialized in pretrained mode. 

**Chunk Size Searching.** We use a light-weight method to search for the best chunk size before training. The chunk size searching script is executed offline on the CPU so that it runs quickly and does not actually allocate memory for parameters. This searching method builds the tensor chunk mapping schema by looking for the optimal chunk size that can host the overall model data in CPU+GPU from a size range of 128-M to 512-M with a step of 32-M. The optimized chunk sizes of the various models are listed in Table 3, where memory fragmentation is minimized (less than 10%) while chunk memory utilization ratio (UTIL.) is maximized.

### 9.2 Experimental Results

#### 9.2.1 Model Scale

Fig. 11 reported the model scale of three systems on YARD and SuperPod. Some entries miss values because of Out-Of-Memory (OOM) or the performance is unable to meet the efficiency bar, which means the throughput is less than 30, 50 Tflops per GPU on YARD and SuperPod, respectively. The ‘X’ indicates the number of GPUs. Our cluster requires each GPU to achieve minimal GPU efficiency to meet the energy requirements.

DeepS(eed)-DP is scaling ZeRO-Offload/Infinity [6], [8] with ZeRO-DP method. DeepS(speed)-MP is combined DP with different ways of MP using Megatron-LM [4]. For example, deeps-mp2 on 8 GPU indicates using MP is used in 2 GPU group and DP is used in a 4 GPU group. Since MP requires customized code modification for model definition, we do not apply MP on PatrickStar. But they can be used together in the future with extra programming effort.

On a YARD GPU node, PatrickStar improves model scale of DeepSpeed-DP 3x from 4B to 12B and improves PyTorch DDP 12x from 1B to 12B. When scaling to 8 GPUs, DeepSpeed-DP can improve model size to 6B on 2 and 4 GPUs. However, it decreases the size back to 4B on 8 GPUs. Compared with DeepSpeed+MP, PatrickStar still improves scale from 8B to 18B (2.25x). On a SuperPod Node, PatrickStar improves model scale 3.3x from 15B to 50B and improves PyTorch 50x from 1B to 50B. When scaling to 8 GPUs, PatrickStar can improve model size from 30B to 68B, 2.27x larger than DeepSpeed.

We analyze the impacts of system design on maximal model scale. PyTorch can only train the cases where model data and non-model data are fitted in the GPU. When the model size is 2B, the model data reaches 2×18=36 GB. In addition to non-model data, the GPU memory requirements exceeds the 32 GB V100 and 40 GB A100 capacity. DeepSpeed can train the case where param fp16 in addition to non-model data, the GPU memory requirements model size is 2B, the model data reaches 2×32 GB (20% comes from model-data GPU memory for warm-up from Section 6.3). The memory utilization is 252/291.2=86%. On 8 SuperPod GPUs, the memory utilization is 87.5%. We believe PatrickStar has reached the utilization limit of CPU+GPU heterogeneous memory space, considering the memory overhead of PyTorch and memory fragmentation of chunks.

#### 9.2.2 Computing Throughput on One GPU

Fig. 12 shows the performance of three systems on one V100 GPU in YARD and one A100 in SuperPod. The missing columns indicate DeepSpeed and PyTorch fail in these cases. The performance of PatrickStar and PyTorch is similar on the 1B model and higher than DeepSpeed. PatrickStar achieves higher performance and larger batch size than DeepSpeed in all the cases. Generally, a larger batch size leads to higher GPU utilization. Note the performance of PatrickStar is lower on batch size 48 than 32 on 6B and 15B model training on SuperPod. It is because that the large batch size cases use activation offloading and introduce more CPU-GPU data communications.

#### 9.2.3 Computing Throughput on Multiple GPUs

Figs. 13 and 14 presents the performance of 3 systems on 1, 2, 4, 8 GPUs. The y-axis is re-scaled by logarithmic. The

---

### Table 3: Chunk Size Searching Results

| Model     | YARD          | SuperPod      |
|-----------|---------------|---------------|
| SIZE (M)  | 10B 15B 18B 20B 40B 60B 68B | 252/291.2=86% |
| UTIL.(%)  | 94.47 92.62 91.5 90.5 90.6 92.2 97.4 | 87.5% |

---

**Fig. 11.** The max model scale on YARD and SuperPod of PyTorch, DeepSpeed and PatrickStar.
points represent the best results achieved with the best batch size. The value around the confidence interval bar indicates average throughput in Tflops. The deeps is DeepSpeed-DP, and deeps-mpX is DeepSpeed with X-way MP.

Fig. 13 presents results on YARD. PatrickStar is 1.37x faster than PyTorch on 8 GPUs and is similar to PyTorch in the 1,2,4 GPU cases for 1B model. In 8 GPU cases, the maximum batch size of PyTorch is 4 while PatrickStar is 64, and small-batch size decreases the efficiency of hardware utilization. On YARD, using the same Zero-DP strategy, PatrickStar is superior to DeepSpeed-DP in all of the cases and is the only solution to train model size between 8B and 18B with DP only. The improvement is significant (1.08x-1.47x, on average 1.23x), especially for small models. The reason is also that PatrickStar shrinks CPU-GPU data transmission volume. PatrickStar does not significantly decrease computational efficiency when increasing the model size. On 8 GPUs, the 419 Tflops performance on the 18B model is 94% of 444 Tflops on 1B. This shows that PatrickStar is very robust on the larger model scale with the help of efficient chunk orchestrating.

Fig. 14 presents the results on SuperPod. MP results are missing because they are always inferior to DP in the same test case. On SuperPod, the speedup to DeepSpeed is more significant than on YARD (1.07x-2.43x, on average 1.53x). There is also no significant degradation of performance as the model size scales. On 8 GPUs, the 857 Tflops performance on the 68B model is 73% of 1180 Tflops on 6B.

9.2.4 Optimization Analysis

We highlight the effect of CMM in Fig. 15, where SpongeBob (Sbob) is a system using only DMM without CMM. It is implemented with ColossalAI V0.1.3 [36], another open-sourced software developed by the authors. The figure contains computing parts (ADAM and FWD+BWD), inter-GPU communication parts (bcast and reduce) and the chunk moving parts. The gpu(cpu) → cpu(gpu) indicates the elapse of CPU-GPU chunk moving for FWD+BWD computing. The gpufp16(cpufp32) → cpufp32(gpufp16) indicate the elapse of CPU-GPU chunk moving and accompanying floating-point conversion for ADAM computing.

DeepSpeed execution time is presented as ‘DeepS’ in Fig. 15 and ‘DS’ in Fig. 16.

Fig. 16 demonstrates the effect of our proposed optimizations in DMM. Six representative cases were tested by using 1 and 8 GPU to train 10B and 50B models on SuperPod and 12B on YARD. In Fig. 16, ‘XgBase’ (X=1 or 8) represents the performance of the base PatrickStar on X GPU. ‘OSC’ represents the performance of fixing OS chunks on the CPU without Device-aware Operator Placement. ‘SP’ represents the performance using a static partition instead of the runtime memory tracer. In this case, we reserve 20% GPU memory to store the model data to avoid OOM. We can draw the following three conclusions.

First, Runtime memory tracer of Section 6.1 can significantly reduce the volume of CPU-GPU chunk moving during the FWD+BWD. As shown in the Fig. 16, compared...
with the SP plan, the base version almost eliminates the cpu (gpu)—cpu(cpu) cost. The resulting end-to-end speedup is impressive. For example, the 8gBase version is 6.9x faster than 8gSP on 10B SuperPod case.

Second, Device-aware operator placement of Section 6.2 reduces chunk movement and computing time of ADAM. Compared with OSC, the base version puts part of the OS on the GPU, thereby reducing the overhead of cufp32 (gpufp32)—gpufp16(cufp32). Additionally, the ADAM computation on GPU is faster than on CPU. It will benefit the cases with enough margin GPU space in Table 5. Table 4 reflects the margin/spilling space for two type of chunks. The last row shows memory space size counting in chunks. Positive numbers indicate the number of OS chunks that can be held in the margin GPU space. Negative numbers indicate the number of param fp16 chunks that have to be spilled to the CPU. For example, the 8gBase version is 1.3x faster than 8gOSC on the 12B YARD case.

Third, the CMM brings efficient inter-GPU and CPU-GPU communication. For SuperPod 10B, 50B, and YARD 12B 8 GPU cases, The base version’s communication overhead (bcast+reduce) are 5%, 11% and 9%, respectively. As shown in Table 5, the achieved bandwidth of communication in the base version on both clusters is above 75% of the saturated bandwidth.

PatrickStar also shows superlinear scalability when increasing GPU numbers, as shown in Fig. 17. Larger models have better scalability, since a more significant proportion of the communication volume has been transferred from PCI-e to NVLink. The latter has higher communication bandwidth.

### 9.3 Scaling to Multi-Node

PatrickStar is also evaluated on a multinode on SuperPod. Nodes are connected with IB-HAC, whose bi-directional bandwidth is 800 Gb/s. Fig. 18 shows superlinear multinode scalability for four test cases using batch size per GPU as 8. We succeed in running a 175B GPT3-scale model [27] on 4 nodes (32 GPUs) and achieved over 8 Pflops on 64 GPUs (41% of the system’s overall peak performance). It is worth noting that the PatrickStar do not make significant changes to the model code using MP.

### REFERENCES

[1] A. Dosovitskiy et al., “An image is worth 16x16 words: Transformers for image recognition at scale,” 2020, arXiv:2010.11929.
[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” 2018, arXiv:1810.04805.
[3] T. Brown et al., “Language models are few-shot learners,” in Proc. Int. Conf. Neural Inf. Process. Syst., 2020, pp. 1877–1901.
[4] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-LM: Training multi-billion parameter language models using model parallelism,” 2019, arXiv:1909.08053.
[5] S. Rajhandari, J. Rasley, O. Ruwase, and Y. He, “ZeRO: Memory optimizations toward training trillion parameter models,” in Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal., 2020, pp. 1–16.
[6] S. Rajhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, “ZeRO-infinity: Breaking the GPU memory wall for extreme scale deep learning,” in Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal., 2021, pp. 1–14.
[7] B. Pudipeddi, M. Mesmakhosroshahi, J. Xi, and S. Bharadwaj, “Training large neural networks with constant memory using a new execution algorithm,” 2020, arXiv:2002.05645.
[8] J. Ren et al., “ZeRO-Offload: Democratizing billion-scale model training,” in Proc. USENIX Annu. Tech. Conf., 2021, pp. 551–564.
[9] A. Vaswani et al., “Attention is all you need,” in Proc. Int. Conf. Neural Inf. Process. Syst., 2017, pp. 5998–6008.
[10] P. Micikevicius et al., “Mixed precision training,” 2017, arXiv:1710.03740.
[11] J. Dean et al., “Large scale distributed deep networks,” in Proc. Int. Conf. Neural Inf. Process. Syst., 2012, pp. 1232–1240.
[12] S. Li et al., “Pytorch distributed: Experiences on accelerating data parallel training,” Proc. VLDB Endowment, vol. 13, no. 12, pp. 3005–3018, Aug. 2020.
[13] N. Shazeer et al., “Mesh-TensorFlow: Deep learning for supercomputers,” 2018, arXiv:1811.02084.
Y. Huang et al., “GPipe: Efficient training of giant neural networks using pipeline parallelism,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 103–112.

D. Narayanan et al., “PipeDream: Generalized pipeline parallelism for DNN training,” in *Proc. 27th ACM Symp. Operating Syst. Conf.*, 2019, pp. 1–15.

Z. Li et al., “TeraPipe: Token-level pipeline parallelism for training large-scale language models,” in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 6543–6552.

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

A. Gruslys, R. Munos, I. Danihelka, M. Lanciot, and A. Graves, “Memory-efficient backpropagation through time,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 4125–4133.

J. Herrmann, O. Beaumont, L. Eyraud-Dubois, J. Herrmann, A. Joly, and A. Shilova, “Optimal checkpointing for heterogeneous chains: How to train deep neural networks with limited memory,” 2019, arXiv:1911.13214.

A. Paszke et al., “Pytorch: An imperative style, high-performance deep learning library,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.

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 *Proc. IEEE/ACM 49th Annu. Int. Symp. Microarchit.*, 2016, pp. 1–13.

S. Shiriram, A. Garg, and P. Kulkarni, “Dynamic memory management for GPU-based training of deep neural networks,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2020, pp. 200–209.

O. Beaumont, L. Eyraud-Dubois, and A. Shilova, “Optimal GPU-CPU offloading strategies for deep neural network training,” in *Proc. Eur. Conf. Parallel Process.*, 2020, pp. 151–166.

O. Beaumont, L. Eyraud-Dubois, and A. Shilova, “Efficient combination of rematerialization and offloading for training DNNs,” in *Proc. 35th Conf. Neural Inf. Process. Syst.*, 2021, pp. 23844–23857.

Microsoft, “DeepSpeed open-source code,” 2021. Accessed: Dec. 2021. [Online]. Available: https://github.com/microsoft/DeepSpeed

A. Li et al., “Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPU direct,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 1, pp. 94–110, Jan. 2020.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” *OpenAI Blog*, vol. 1, no. 8, 2019, Art. no. 9.

Y. Gao et al., “Estimating GPU memory consumption of deep learning models,” in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1342–1352.

J. Lee et al., “On-device neural net inference with mobile GPUs,” 2019, arXiv:1907.01989.

J. Fang, Y. Yu, C. Zhao, and J. Zhou, “TurboTransformers: An efficient GPU serving system for transformer models,” in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 389–402.

L. A. Belady, “A study of replacement algorithms for a virtual-storage computer,” *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.

R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, “Evaluation techniques for storage hierarchies,” *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.

B. V. Roy, “A short proof of optimality for the min cache replacement algorithm,” *Inf. Process. Lett.*, vol. 102, no. 2/3, pp. 72–73, 2007.

L. A. Belady and F. P. Palermo, “On-line measurement of paging behavior by the multivalued min algorithm,” *IBM J. Res. Develop.*, vol. 18, no. 1, pp. 2–19, 1974.

NVIDIA, “NVIDIA DGX A100 the universal system for AI infrastructure,” 2020. Accessed: Dec. 2021. [Online]. Available: https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf

S. Li et al., “Colossal-AI: A unified deep learning system for large-scale parallel training,” 2021, arXiv:2110.14883.

**Zilin Zhu** received the MS degree from the Department of Computer Science, Columbia University, USA. He is a software developer with WeChat AI Lab, Tencent Inc. His research interests include compiler techniques and distributed system.

**Shenggui Li** received the BS degree from the Department of Computer Science, Nanyang Institute of Technology. He is the co-founder and VP of HPC-AI Technology. He was a research assistant with the National University of Singapore. His research interests include HPC and deep learning.

**Hui Su** received the MS degree from the Institute of Software, Chinese Academy of Sciences. He is a senior research scientist with WeChat AI Lab, Tencent Inc. His primary research interests lie in the general areas of natural language processing applications, and human-inspired language learning.

**Yang You** received the PhD degree in computer science from UC Berkeley. He is a presidential young professor with the National University of Singapore. His research interests include parallel/distributed algorithms, HPC, and ML. He is a winner of IPDPS 2015 and ICCP 2018 Best Paper Award. He is the founder and chairman of HPC-AI Technology.

**Yang Yu** received the MS degree from Tianjin University. He is a principle software engineer with WeChat AI Lab, Tencent Inc. Before joining Tencent, he worked in Baidu and was the main contributor to PaddlePaddle.

**Jie Zhou** received the PhD degree from the Institute of Theoretical Physics, Chinese Academy of Sciences. He is the director of WeChat AI Lab, Tencent Inc. His research interests include natural language processing, multimedia, and machine learning.

**For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdil.**

**Jiarui Fang** received the PhD degree from the Department of Computer Science and Technology, Tsinghua University. He is the co-founder and CTO of HPC-AI Technology. Previously, he was a senior research scientist with WeChat AI Lab, Tencent Inc. His research interests include HPC, advanced computing, and AI.