# GMLake: Efficient and Transparent GPU Memory Defragmentation for Large-scale DNN Training with Virtual Memory Stitching

Cong Guo*
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute

Rui Zhang*
Ant Group

Jiale Xu
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute

Jingwen Leng†
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute

Zihan Liu
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute

Ziyu Huang
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute

Minyi Guo†
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute

Hao Wu
Ant Group

Shouren Zhao
Ant Group

Junping Zhao†
Ant Group

Ke Zhang
Ant Group

## Abstract

Large-scale deep neural networks (DNNs), such as large language models (LLMs), have revolutionized the artificial intelligence (AI) field and become increasingly popular. However, training or fine-tuning such models requires substantial computational power and resources, where the memory capacity of a single acceleration device like a GPU is one of the most important bottlenecks. Owing to the prohibitively large overhead (e.g., 10×) of GPUs' native memory allocator, DNN frameworks like PyTorch and TensorFlow adopt a caching allocator that maintains a memory pool with a splitting mechanism for fast memory (de)allocation. Unfortunately, the caching allocator's efficiency degrades quickly for popular memory reduction techniques such as recomputation, offloading, distributed training, and low-rank adaptation. The primary reason is that those memory reduction techniques introduce frequent and irregular memory (de)allocation requests, leading to severe fragmentation problems for the splitting-based caching allocator. To mitigate this fragmentation problem, we propose a novel memory allocation framework based on low-level GPU virtual memory management called GPU memory lake (**GMLake**). GMLake employs a novel **virtual memory stitching** (VMS) mechanism, which can fuse or combine non-contiguous memory blocks with a virtual memory address mapping. GMLake can reduce average of 9.2 GB (up to 25 GB) GPU memory usage and 15% (up to 33% ) fragmentation among eight LLM models on GPU A100 with 80 GB memory. GMLake is completely transparent to the DNN models and memory reduction techniques and ensures the seamless execution of resource-intensive deep-learning tasks. We have open-sourced GMLake at https://github.com/intelligent-machine-learning/glake/tree/main/GMLake.

***CCS Concepts:*** • **Computing methodologies** → *Machine learning*; • **Software and its engineering** → **Virtual memory management schemes**.

***Keywords:*** Memory Defragmentation, GPU, Deep Learning, Virtual Memory Stitching

**ACM Reference Format:**
Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. 2024. GMLake: Efficient and Transparent GPU Memory Defragmentation for Large-scale DNN Training with Virtual Memory Stitching. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640423

*Equal contribution
†Corresponding authors


*ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA*

ACM ISBN 979-8-4007-0385-0/24/04…$15.00
https://doi.org/10.1145/3620665.3640423

## 1 Introduction

Large-scale deep neural network (DNN) models, specifically Large Language Models (LLMs), have revolutionized natural language processing (NLP) and artificial intelligence (AI) research [88]. LLMs, such as the GPT-3 [8] architecture, are sophisticated DNN models with remarkable capabilities in understanding, generating, and processing human language. These models leverage vast amounts of textual data and employ Transformer-based architectures characterized by attention mechanisms [78] to achieve state-of-the-art performance on various NLP tasks. However, the widespread adoption of LLMs comes with significant computational challenges, as training or fine-tuning such models requires substantial computational power and resources. For example, OPT [87], with 175 billion parameters, needs 34 days on 1,024 A100 GPUs, and 65B-parameter LLaMA [77] processes 1.4T tokens on 2048 A100 GPUs taking approximately 21 days.

Therefore, deep learning (DL) frameworks, e.g., PyTorch [63] and TensorFlow [76], have emerged as the fundamental infrastructure for DNN models due to their flexibility and computational efficiency. Those DL frameworks have enabled the training of increasingly large and complex neural network models. Meanwhile, the GPU architecture [12, 36, 60] has become the most widely used hardware to support the high-performance execution of DNN models. On the other side, the growing scale and complexity of DNN models poses new challenges to GPU memory management. For instance, using the CUDA's native memory allocation APIs like `cudaMalloc` and `cudaFree` incurs a large overhead. To improve the efficiency of GPU memory allocation, DL frameworks opt to implement a caching allocator that maintains a memory pool with the best fit with coalescing (BFC) algorithm [76]. Our experiments show that the caching allocator outperforms the native memory allocator by almost 10×.

On the other side, the rapid growth in the memory requirements of large-scale DNN models [2, 75] has sparked the development of methods at the system- and algorithm-level to alleviate memory demands. Examples for these methods include recomputation [40, 86], offloading [69], distributed training [28, 30, 42, 45, 72, 83] and low-rank adaptation [29]. Even though these optimizations can effectively reduce memory footprint for training or fine-tuning large-scale DNN models, they may lead to poor memory utilization. The reason is that they also introduce a significant amount of regularity and dynamicity in the memory allocation requests, which results in up to 30% GPU memory fragmentation.

As shown in Figure 1 left, DL frameworks manage the memory allocation within the memory pool. They adopt the "splitting" method to split the memory pool to fit the DNN tensors' arbitrary size and boost the utilization of the memory pool. However, that will cause severe memory fragmentation for some new allocations. For example, the framework splits the third line to store the new allocation of Block 4.

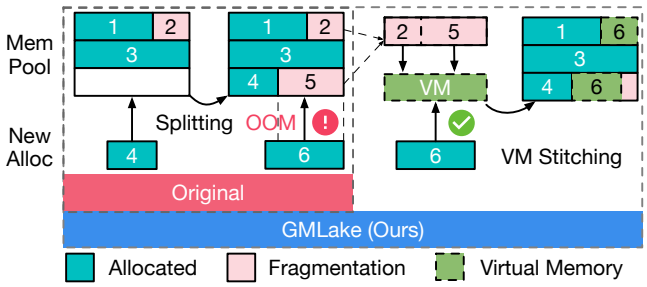



**Figure 1.** Representative example of memory allocation problem. The original splitting method can boost GPU memory utilization but cause fragmentation. Our proposed virtual memory stitching can complement and optimize the memory fragmentation issues.

But the memory pool cannot hold Block 6 because the size of Block 6 is larger than Block 5, which cannot be exploited and becomes fragmented. Finally, the framework will report the out-of-memory (OOM) error, one of the most common issues in the training processing for DNN models. The aforementioned memory reduction techniques like recomputation and offloading can mitigate the OOM issue, but also lead to more frequent and irregular memory allocation and deallocation requests, exacerbating the fragmentation problem.

To mitigate GPU memory fragmentation and improve efficient memory utilization, this study focuses on exploring the causes of GPU memory fragmentation and proposes a novel memory allocation framework based on low-level GPU virtual memory management, called GPU memory lake (**GMLake**), to optimize GPU memory management with low overhead. As shown in Figure 1 right, GMLake employs a novel **virtual memory stitching** (VMS) mechanism, which is seemingly a reverse behavior to the splitting. Compared to the original framework, it can fuse or combine non-contiguous memory blocks with a virtual memory address mapping. For example, the VMS can map Block 6 to the Block 2 and 5 stitched block by a virtual memory address, then store Block 6 in the physical memory chunks of Block 2 and 5. Obviously, virtual memory stitching effectively reduces memory fragmentation and improves memory utilization. We implement the GMLake on the low level of the DL framework and replace the original memory allocation API of DNN training. GMLake is completely transparent to the DNN models and other memory optimization methods, e.g., recomputation and offloading, ensuring the seamless execution of resource-intensive deep-learning tasks.

Overall, this work makes the following contributions:

- We perform a characterization study to show that the caching allocator used in existing DL frameworks suffers from up to 30% memory fragmentation when running large-scale DNN models with various memory reduction techniques such as recomputation, offloading, distributed training, and low-rank adaptation.

- We design and implement GMLake, a novel memory allocator that effectively reduces memory fragmentation and improves memory utilization. GMLake is transparent to the above existing memory reduction techniques. It integrates the virtual memory stitching mechanism by using the low-level CUDA virtual memory management knobs.
- We evaluate GMLake on multiple prominent LLM optimization platforms with a set of representative open-source LLMs to demonstrate its effectiveness, efficiency, and robustness. In the best case, we can reduce GPU memory usage by 33%, translating to 25 GB memory saving comparing to native caching allocator in PyTorch on an A100 GPU with 80 GB HBM memory.

## 2 Background and Motivation

In this section, we provide essential background concerning the memory management of the DL framework and outline the motivation behind conducting this study through our experimental observations. To begin, we provide a concise overview of the increasing trend of large-scale DNN such as LLM. Subsequently, we conduct a comparative analysis of various memory management methods. Following the comparison, we uncover significant fragmentation challenges that arise in the context of LLMs during distributed training and memory-efficient optimization strategies. As a result, it is imperative for us to develop a novel and efficient memory allocator that can effectively address these challenges.

### 2.1 Large-scale DNNs

LLM, such as OpenAI's GPT series [33, 57, 88], represent the success of large-scale DNNs and have led to significant advancements in various language processing tasks. GPT-2 represents a significant advancement over its predecessor, offering a ten-fold increase in model size and complexity [66], followed by GPT-3 [7] with 175 billion parameters, and ChatGPT [49], fine-tuned for conversations.

However, the size and complexity of these models pose considerable challenges in training and deployment. The requirements for vast computational resources, enormous data, and extensive time (e.g., OPT-175B [87] taking 34 days with 1024 A100 GPU) have intensified focus on efficient training optimization. Therefore, this study emphasizes the importance of efficient memory management for LLM training.

### 2.2 Memory Management of DL Framework

Frameworks like PyTorch [61] and TensorFlow [1] play a crucial role in DNN model training and inference. Concurrently, GPU [12, 36, 60] has become an important hardware for high-performance model execution. This study focuses on the memory management optimization for those popular frameworks on GPUs. We compare the three types of memory management: GPU native allocator, caching allocator, and virtual memory (VM) allocator. We conduct multiple experiments to show the efficiency associated overhead for each allocator.

***Native Allocator.*** As depicted in Figure 2(a), the native allocator is provided through GPU-vendor-supplied APIs, i.e., `cudaMalloc` and `cudaFree`, which need device synchronizations. The native allocator has a simplistic design that lacks flexibility. This makes it unsuitable for applications that need dynamic and resizable memory structures or complex memory management, especially in the context of DL. If the DL framework implements the native GPU allocator without proper synchronization optimization, it may cause unacceptable overhead for training the DNN models.

Our experimental results have quantified the overhead of the native allocator. We disable the PyTorch caching allocator (presented in the next paragraph) to train the OPT-1.3B model [87] on four A100-80G GPUs. The native allocator in PyTorch provide identical programming model for users, who can change the environment variables to set the allocator. The throughput of the GPU native allocator is 9.7× lower than the original PyTorch allocator. Therefore, an efficient memory management design should be one of the most critical components of the DL framework.

***Caching Allocator.*** DL frameworks usually use the caching allocator with a memory pool for fast memory allocation and deallocation without device synchronizations. Figure 2(b) depicts the BFC algorithm [76] in the caching allocator employed by PyTorch and TensorFlow. The BFC implementations of PyTorch and TensorFlow are almost the same, with minor differences in their data structures.

There are four main operations in the BFC algorithm. ❶ It begins with searching for the most suitable allocated but inactive memory block, known as the "best fit". If there is no suitable allocated memory block candidate, the caching allocator invokes native GPU allocator APIs to allocate new memory blocks. ❷ If the requested memory is smaller than the best-fit block, the algorithm splits the block into two blocks to boost memory utilization. One of the split blocks is allocated to fulfill the memory request, while the other remains in the memory pool for future reallocation. To effectively manage the memory, these two blocks are interconnected through a bidirectional link, with each block monitoring the availability status of its adjacent block. ❸ For the free (deallocation) operation, the algorithm does not invoke the native GPU API `cuMemFree` but only releases the assignment (pointer) to the block and sets the block to inactive. ❹ Finally, the caching allocator examines whether the blocks adjacent to the left or right are also inactive. If so, the caching allocator would merge those adjacent inactive blocks into a single block.

Obviously, the caching allocator can significantly reduce the invocations of native GPU memory allocator APIs. In the best case, all memory blocks are allocated and deallocated only once through the native allocator. As such, the caching

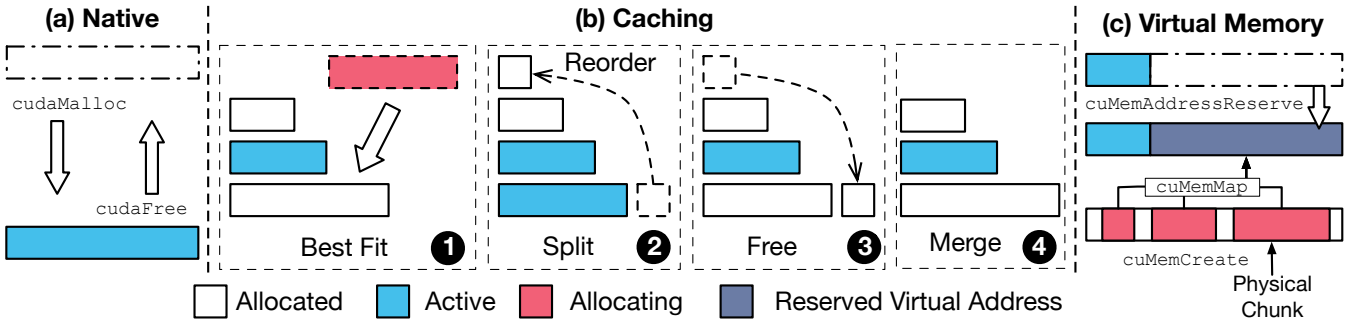



**Figure 2.** Three memory management strategies.

allocator can be much more efficient than the native GPU allocator and is widely adopted in existing DL frameworks. On the other hand, the "splitting" mechanism will provoke a lot of possible memory fragmentation issues when the memory allocation requests are irregular and their sizes are significantly different from each other. This fragmentation issue is not notable previously because the models are usually regular and not large enough. For example, Transformer-based model [78] is a stack of multiple identical layers with the same size of tensor, leading to minor memory fragmentation.

However, as the size of LLM grows, the fragmentation issue significantly deteriorates due to the distributed training and complex training memory strategy, leading to limited batching and inefficient memory management and training. In the next subsections, we observe fragmentation issues become challenging in these complex training scenarios.

### 2.3 Memory-efficient Optimization

The rapid growth in the memory requirements of large-scale DNN models [2, 75] has sparked the development of methods at the system- and algorithm-level to alleviate memory demands. Examples for these methods include recomputation [40], offloading [69], and low-rank adaptation [29].

Recomputation [40, 86], also known as checkpointing, involves recalculating specific layer outputs during back-propagation rather than storing them, allowing for memory savings. Offloading (also known as swap), such as ZeRO-Offload [69] shifts optimizer memory and computation from GPU to CPU, enabling training of large models on a single GPU and scaling across multiple GPUs. In addition to these system-level approaches, algorithmic methods can also effectively reduce memory requirements. For example, LoRA [29], designed for large-scale models, introduces rank-decomposition matrices to minimize both trainable parameters and GPU memory requirements, achieving accuracy similar to full model fine-tuning with reduced cost and time.

However, even though these optimizations can effectively reduce memory footprint in GPU memory, they may sometimes lead to poor memory utilization. As depicted in Figure 3, we train the OPT-1.3B model on four A100-80G GPUs with different optimization method combinations. Using only PyTorch (P) achieves high memory utilization, whereas employing techniques such as LoRA (L), Recomputation (R), or Offload (O) significantly reduces memory utilization. According to our investigations, combining these techniques results in high memory fragmentation. The reason is that these memory optimization techniques inherently incur dynamic and irregular allocation requests.

To explore the origin of this irregularity, we present the memory footprint of a training process on the GPT-NeoX-20B model. As shown in Figure 5, the left figure shows the footprint on the original PyTorch, and the right figure is collected from PyTorch with LR (LoRA and Recomputation) optimization. Obviously, the right figure shows more irregularity than the left because of the usage strategy like recomputation. Statistically, the left figure makes 46 thousand allocations with a size of 93 MB on average, while the right

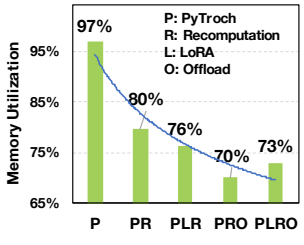



**Figure 3.** Memory utilization with five method combinations.

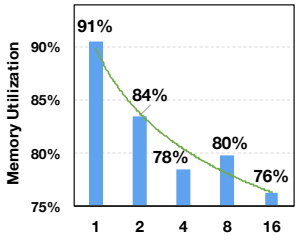



**Figure 4.** Memory utilization with different GPU numbers.

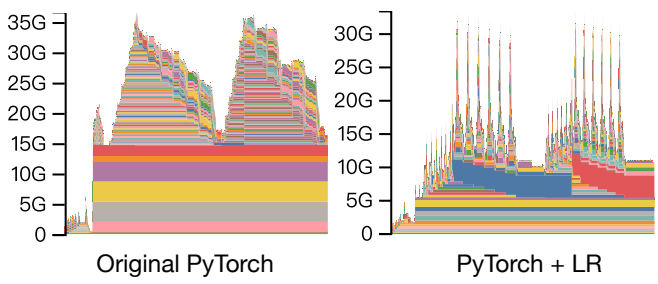



**Figure 5.** Memory footprint of GPT-Neox-20B training.

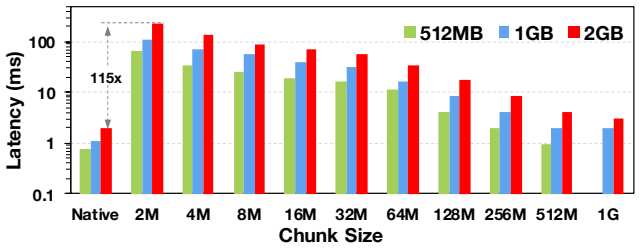



**Figure 6.** The allocation latency of native allocator (the first one) and virtual memory allocator.

| Chunk Size | 2 MB | 128 MB | 1024 MB |
|---|---|---|---|
| cuMemReserve | 0.003 | 0.003 | 0.002 |
| cuMemCreate | 18.1 | 0.89 | 0.79 |
| cuMemMap | 0.70 | 0.01 | 0.002 |
| cuMemSetAccess | 96.8 | 8.2 | 0.7 |
| Total | 115.4 | 9.1 | 1.5 |

**Table 1.** The VMM API execution time breakdown normalized by cuMalloc execution time.

figure has 76 thousand allocations with 85 MB on average, indicating that complex strategies lead to more frequent and smaller allocations thus causing fragmentation.

The results motivate us to address the memory fragmentation issues for more efficient and scalable large-scale DNN model training.

***Observation 1**: The more complex and irregular memory optimization strategies used, the more fragmentation there will be.*

### 2.4 Distributed Training

With the increasing complexity of DNN models, distributed training has become vital, especially for LLMs. Data parallelism, supported by PyTorch distributed data parallel (DDP) [43], duplicates the setup to process different portions of data simultaneously, synchronizing after each training step. Model parallelism splits the model across multiple GPUs, each handling different stages. The model parallelism includes two categories: pipeline parallelism [42, 72, 83], placing individual layers on single GPUs, and tensor parallelism [28, 30, 45], dividing each tensor into chunks for specific GPUs.

Obviously, using more GPUs would cause more fragmentation due to its irregular memory allocation and deallocation. To examine this effect, we conduct a test implemented on OPT-13B on PyTorch. As depicted in Figure 4, when the number of GPUs is one, the memory utilization is >90%. However, as the number of GPUs grows to 16, the memory utilization declines to 76%, even though multi-GPU parallelism is essential for training large models. Such fragmentation wastes memory resources and limits the batch size of LLM training.

***Observation 2**: As the number of GPUs scales up, the issue of memory fragmentation is likely to become more pronounced.*

### 2.5 Low-level Virtual Memory (VM) Management

Recognizing the growing need among applications to manage memory quickly and efficiently, CUDA has introduced a new feature called low-level virtual memory management [62], similar to Windows's VirtualAlloc [56] and Linux's mmap [48]. This feature breaks the malloc-like abstractions, and offers primitive operations such as reserve and map to manipulate the virtual address space. In our work, we show that this low-level VM management can be used to reduce memory fragmentation and improve memory utilization for large-scale DNN training, which we call as virtual memory allocator.

Figure 2(c) illustrates the basic idea of using this low-level virtual memory management. The cuMemAddressReserve function reserves a virtual memory address for the new memory allocation, and cuMemCreate allocates physical memory chunks on GPU. It is not revealed how the underlying system translates memory in the physical address space. Furthermore, there is no guarantee of contiguity of physical chunks. The cuMemMap function bridges the physical and virtual memory, mapping the physical handle to the reserved address. CUDA also offers a suite of memory deallocation functions such as cuMemUnmap, cuMemAddressFree, and cuMemRelease. Obviously, the advantage of low-level VM API is that it can allocate and map the non-contiguous physical chunks, which can tackle GPU memory fragmentation issues. However, the virtual memory allocator costs much more expensive overhead than native GPU allocator.

To verify the overhead of the VM allocator, we have the experiments on memory allocation with three different sizes: 512 MB, 1 GB, and 2 GB, which are total allocated block sizes. Figure 6 illustrates the comparative results between the native memory allocator and the virtual memory allocator. The y-axis represents the allocation latency, taken on a logarithmic scale. On the x-axis, 2 MB, 4 MB, ..., and 1024 MB represent the sizes of internal physical chunks that construct the allocation block. For example, the 1 GB allocation block needs to map 512 chunks with 2 MB size. Finally, the result in Figure 6 shows that the latency of virtual memory is exceedingly high. Specifically, if the virtual memory block is partitioned into 2 MB chunks, it would be over 100× slower than the native allocator, which is totally unacceptable.

To further explore the bottleneck of the VMM API, we provide the execution time breakdown of allocation. Table 1 shows the latency breakdown of the VMM API with 2 GB GPU memory allocation. All latency is normalized to the cuMalloc. Each allocation in GMLake needs only one cuMemAddressReserve but multiple cuMemCreate, cuMemMap, and cuMemSetAccess for each physical chunk. cuMemSetAccess is a special function to make the map available provided by

VMM. We can see that using a 2MB small chunk to allocate 2 GB of memory is 115× slower than native `cuMalloc`.

***Observation 3**: Although virtual memory can reduce memory fragmentation, the original virtual memory allocator on GPU still presents many challenges and needs further optimization.*

## 3 GMLake

In this work, we introduce GMLake, an efficient memory allocation method specifically designed for GPU memory (i.e., GPU memory lake). GMLake leverages CUDA's low-level VM management features to expedite memory allocation and deallocation processes. Figure 7 provides an overview of GMLake, which provides the same memory (de)allocation interfaces to the existing caching allocator but internally integrates the virtual memory stitching (VMS) mechanism. This integration is achieved through the precise utilization of CUDA's low-level VM management APIs.

The original caching allocator in DL frameworks adopt the BFC algorithm, as explained in Section 2.2 and shown in Figure 2(b). To avoid synchronization and improve memory management efficiency, those frameworks internally manage the memory pool to handle the (de)allocation instead of directly using the native APIs. Following this approach, our GMLake also has the three following components:

**Virtual memory API**: This refers to the low-level APIs employed to instruct the GPU to allocate and free memory using virtual memory addresses, a process characterized by significant overhead if not fully optimized.

**Virtual memory pool**: Serving as the foundational data structure, this is designed for caching virtual memory. Its implementation is crucial for enhancing efficiency.

**GMLake allocator**: This includes all functions, algorithms, and strategies essential for managing the VM pool.

In this section, we describe the design and details for the three integral components and assemble the GMLake framework, proceeding from the foundational to the topmost layer.

### 3.1 Virtual Memory API

As introduced in Section 2.5 and illustrated in Figure 2(c), the low-level VMM APIs serve as the fundamental interface between the GPU and the application. As depicted in the bottom of Figure 8, we utilize the VMM API to build the primitive block (pBlock), a critical data structure for the GMLake allocator. The operations within the pBlock include:

**`AddrReserve`**: Initially, the allocation of a primitive block necessitates specifying the allocation size and reserving the corresponding virtual address (VA).

**`Create`**: Following that, the primitive block creates the physical chunks where the data is stored physically.

**`Map`**: Lastly, the primitive block maps all physical chunks to the virtual address, enabling seamless access for the tensors.

To optimize defragmentation, we apply a uniform chunk size of 2 MB across all chunks. While the overhead of this

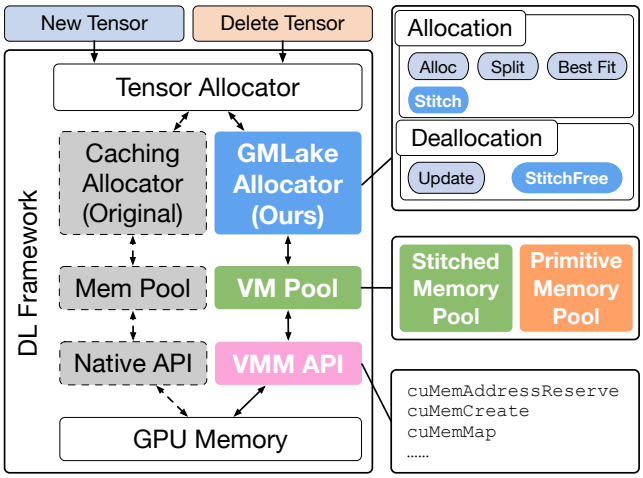



**Figure 7.** The overview of caching allocator and GMLake.

2 MB chunk size is considerable (Section 2.5), it can be mitigated through an efficient data structure and a well-designed stitching strategy that achieves the best defragmentation effect and the best compatibility with PyTorch code base. Meanwhile, we use the following DNN-specific optimization to reduce the frequency of stitching so that its end-to-end overhead is negligible. GMLake uses VMM to tackle allocation larger than 2MB. For memory allocation less than 2MB, we use the original PyTorch splitting method of the caching allocator to deal with its internal fragmentation issues. Moreover, allocation < 2MB is rare in LLM training.

The map operation serves as the basic operation for virtual memory stitching, allowing us to concatenate multiple physically continuous chunks that may not be contiguous in physical memory. Employing the VMM APIs, we can orchestrate the virtual memory into the virtual memory pool, the underlying data structure for the GMLake allocator. The details of VM pool are presented as follows.

### 3.2 Virtual Memory Pool

Given that original VMM APIs are time-consuming, it is crucial to reduce their usages to achieve high efficiency for GMLake. Drawing inspiration from the caching allocators, we have designed our virtual memory pool (VMP) with caching capability, thereby markedly reducing the instances of physical memory (de)allocations. Shown in Figure 8, we differentiate between two types of memory pools: the primitive memory pool (pPool) and the stitched memory pool (sPool).

***pPool and pBlock.*** The data structure of pPool utilizes a sorted set to store the pBlocks. For each pBlock, pPool begins by constructing a structure to document the pointer to the pBlock, including essential basic attributes, such as the active state of the pBlock. Subsequently, the newly allocated pBlock is inserted into the set, with all pBlocks sorted by block size in descending order. The pBlock, serving as the

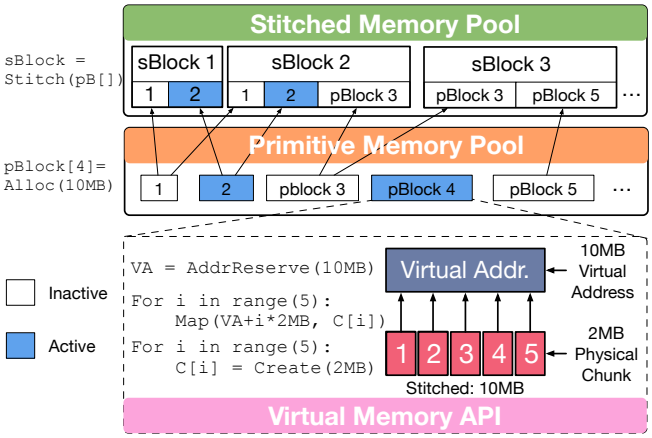



**Figure 8.** The data structure of primitive and stitched memory pool.

primitive block, represents the smallest unit accessible to high-level tensors, and it functions as a basic data structure that can be stitched and pointed to by multiple sBlocks.

***sPool and sBlock.*** The sPool is also organized into a sorted set, similar to pPool. Its elements comprise the stitched block structure, which integrates multiple pBlocks. For instance, as illustrated in Figure 8, sBlock 3 contains pBlocks 3 and 5, which are stitched together, and pBlock 3 may also be pointed to and stitched by sBlock 2. Consequently, the attributes of the sBlock are influenced by the pBlocks, such that if even one pBlock is active, all corresponding sBlocks are labeled as active. In practice, the sBlock remaps virtual memory to all physical chunks of the pointed pBlocks, making it accessible by high-level tensors. To streamline the process, we stipulate that sBlock can only be allocated or assigned to tensor allocations that match the sBlock size. Further details on this constraint are provided in the subsequent section.

Comparing the physical chunk, pBlock, and sBlock reveals their roles as data structures at different levels. While physical chunk is controlled by the low-level API and remains transparent to the high-level tensors, pBlock and sBlock reside in the pPool and sPool, respectively, offering the virtual memory address for access by high-level tensors. Moreover, sBlock operates at a more advanced level, consisting of multiple pBlocks. Next, we describe how GMLake uses those data structures to achieve efficient memory management.

### 3.3 Allocator

The allocator includes all the essential functions and algorithms for the memory allocation and deallocation. Due to the space limit, we only briefly explain the most important functions used in the allocation and deallocation modules.

**3.3.1 Allocation Module.** The **`Alloc`** function is responsible for allocating a new pBlock and inserting it into pPool, as detailed in Figure 8. It serves as the exclusive interface for allocating new physical chunks and incrementing the allocated GPU memory.

The **`Split`** function divides a pBlock (primitive block) into two smaller pBlocks, similar to the "Split" operation depicted in Figure 2, but with an entirely distinct underlying implementation. Specifically, the `Split` function in GMLake operates based on the pBlock structure, resulting in two new pBlocks with corresponding virtual memory addresses and remapped physical chunks. The previous pBlock structure is subsequently removed from the pPool set.

The **`Stitch`** function is the sole mechanism to create an sBlock and insert it into the sPool, as shown in Figure 8 top. This function is an integral component of our allocator and can stitch together multiple pBlocks into a single sBlock. We use the VMM API, i.e., the low-level API provided by NVIDIA specifically for Virtual Memory Management (VMM), to "stitch" two pBlocks, as shown in Figure 2 (c). Assume we have two pBlocks, $p_1$ (1GB) and $p_2$ (2GB). We adopt the VMM API `cuMemCreate` to create corresponding physical chunks and reserve virtual address (VA) by `cuMemAddressReserve`. Then, VA maps PA using `cuMemMap`. Actually, we do not need to unmap the original VA-PA mapping for $p_1$ and $p_2$, as the PA in VMM can be pointed by multiple VAs. Therefore, we only reserve a 3GB VA of the sBlock $s_1$ using `cuMemAddressReserve`. For all sBlocks, they never create `cuMemCreate` new physical chunks. We use the API `cuMemMap` to map the VA of $s_1$ (3GB) to the physical chunks of $p_1$ and $p_2$. Since multiple sBlocks can contain the same physical chunks, we need more attributes to ensure that each physical chunk is used by only a single tensor, such as the active state of the pBlock.

The **`BestFit`** function identify the most suitable pBlock or sBlock for memory allocation, returning the state and candidate blocks for subsequent processing. As detailed in Algorithm 1, we have designed four states, covering all scenarios GMLake may face. It operates on the assumption that both sPool and pPool are sorted in descending order of size.

- **Exact match** (Line 2-4): This state arises when the size of a candidate block matches the allocation size. The block may be either an sBlock from sPool or pBlock from pPool. This is the sole situation where an sBlock can be assigned for new allocation. All other states exclusively involve pBlock.
- **Single block** (Line 12): Here, `BestFit` identifies the best-fit (minimum) pBlock that is larger than the requested allocation size.
- **Multiple blocks** (Line 14): In scenarios where all pBlocks are smaller than the required allocation size while their total size satisifies allocation requirment, the `BestFit` function greedily seeks multiple candidate pBlocks to stitch together.

**Algorithm 1:** The `BestFit` Function.

```
Input: Allocating block size: bSize;
       Inactive sBlocks and pBlocks: sBlocks, pBlocks.
Output: State, state; Candidate pBlocks, CB.
1  def BestFit(bSize, sBlocks, pBlocks):
       // SBlock only for S1: Exact match.
2      foreach block ∈ sBlocks ∪ pBlocks do
3          if block.size == bSize then
4              return 1, [block]
5      CB=[]
6      CBSize = 0
7      foreach block ∈ pBlocks do
8          if block.size ≥ bSize then
9              CB = [block]
10             CBSize = block.size
11         else if CBSize < bSize then
12             CB.append(block)
13             CBSize += block.size
14         else
15             break
16     if CB.length == 1 & CBSize > bSize then
17         return 2, CB // S2: Single block.
18     else if CBSize ≥ bSize then
19         return 3, CB // S3: Multiple blocks.
20     else
21         return 4, CB // S4: Insufficient blocks.
```

- **Insufficient blocks** (Line 16): This state occurs when there is no enough pBlocks to meet the requested allocation size, though `BestFit` still returns a block list.

GMLake substitutes several of the internal functions of the PyTorch caching allocator module. The "stitch" operation is completely user-transparent and doesn't burden user to modify their code. The CUDA VMM API refers to the low-level API provided by NVIDIA specifically for Virtual Memory Management (VMM).The CUDA API we adpot not only includes the VMM-related, but also comprises regular ones like cuMemAlloc. The implementation of GMLake APIs leverages CUDA APIs to achieve fine-grained memory stitching and reusing, thus they belong to distinct levels and serve corresponding functionalities.

**3.3.2 Deallocation Module.** The deallocation module refrains from actively deallocating physical GPU memory using the low-level VMM API. Instead, it only updates or restores the stitched virtual memory blocks.

**`Update`.** Upon receiving a deallocation request for a high-level tensor, we substitute the original VMM deallocation function with the `Update` function. This function alternates the state of active pBlocks and sBlocks, thereby effecting the removal of links and assignments between tensors and

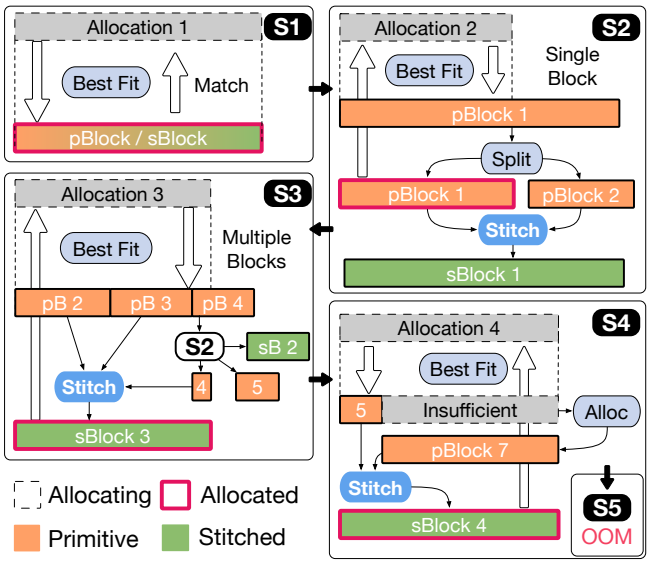



**Figure 9.** Memory allocation strategy in GMLake.

blocks. Throughout the program runtime, actual physical memory remains under the control of corresponding pBlock.

**`StitchFree`.** The purpose of this function is to release the Least Recently Used (LRU) sBlocks held within the sPool. Due to space limitations, we skip intricate details here. We have implemented complete algorithms and data structures to support the LRU-based `StitchFree`. Notably, we only release the inactive sBlock structure from sPool.

We have presented details for various components in GMLake, including low-level APIs, data structures, and high-level functions. Leveraging these thoughtfully engineered features, we are equipped to execute efficient allocation strategies within GMLake, which effectively tackle memory defragmentation challenges in large-scale DNN training.

## 4 Defragmentation Strategy

In this section, we present the strategies to reduce the memory fragmentation issue. We first propose a sophisticated algorithm that theoretically eliminates all fragmentation based on GMLake allocator. We then discuss and describe our optimizations to guarantee its efficiency and robustness.

### 4.1 Memory Allocation Strategy

Figure 9 shows the GMLake allocation strategy to allocate or assign a new memory block to a new tensor allocation request. This strategy is based on the four states provided by `BestFit` module, for each of which we design a post-processing step.

In state S1, an immediate return of existing pBlock or sBlock is made for Allocation 1. If an exact matching block is not found within the inactive sPool and pPool, we progress to state S2, guided by a single block produced by the `BestFit` function. This requires the partitioning (via `Split`) of the

larger pBlock 1 into two distinct pBlocks, both inserted into the pPool. The newly created pBlock 1 replaces its predecessor and is allocated for Allocation 2. Simultaneously, GMLake executes a `Stitch` function, merging pBlock 1 and pBlock 2 into sBlock 1, which is then added to the sPool.

In state S3, GMLake engages in merging (`Stitch`) multiple pBlocks to create consolidated sBlock 3 for Allocation 3. If needed, the final pBlock can be subdivided (`Split`) similar to the procedure in S2. The stage concludes with the introduction of new pBlocks, specifically pBlock 4 and pBlock 5, into pPool, while sBlock 2 and sBlock 3 are added to sPool.

In state S4, it is possible that available blocks for stitching and allocation are insufficient for Allocation 4. GMLake then triggers the `Alloc` function, using the low-level API to create pBlock 7 with new physical chunks and corresponding virtual memory addresses. This new pBlock is added to the pPool. Additionally, we merge (via `Stitch`) pBlock 5 and pBlock 7 into a new sBlock 4, which is returned for Allocation 4 and added to the sPool. If no eligible candidate blocks are present (i.e., the absence of pBlock 5), GMLake directly allocates pBlock 7, without using the `Stitch` function. If the `Alloc` function call fails, GMLake immediately reports an Out-of-Memory (OOM) error in state S5.

### 4.2 Strategy Analysis

We analyze the GMLake memory allocation strategy from aspects of effectiveness, efficiency, and robustness.

**4.2.1 Effectiveness.** The GMLake allocation strategy effectively ensures that nearly all fragmentation is eliminated from the GPU system.

***Interface.*** The effectiveness of this strategy is aided by our interface design, which consolidates all operations into three main functions: `Alloc`, `Split`, and `Stitch`. `Alloc` is the only function that can create a new pBlock, and only `Stitch` can generate a new sBlock, while `Split` does not increase the allocated memory.

***Data Structure.*** The pPool represents a strict one-to-one mapping of GPU memory, with each pBlock being distinct from others. It is an allocated GPU memory set devoid of duplicate elements and overlapping addresses. The sPool is designed to store sBlocks, reserving links and pointers to the pBlocks in a manner similar to a soft link mechanism. GMLake prohibits the splitting of sBlocks as it may affect the pBlocks. The sPool is considered a subset of the pPool.

In the end, each time the program reaches a new peak in GPU memory usage, for example, when calling the `Alloc` function, the pPool may not provide enough blocks for stitching and allocation, resulting in full memory utilization without fragmentation. This contrasts with the original caching allocator, which may leave many sub-blocks unused.

**4.2.2 Efficiency.** We incorporates several methods to achieve high efficiency. Initially, the algorithmic problem in S3 represents a classic NP-hard packing problem [55]. Yet, through the application of the `Split` and `Stitch` functions, an exactly-matched block is generated to fit the allocation, resulting in linear complexity.

Secondly, the union set comprising sPool and pPool chronicles all sizes and corresponding blocks for every tensor allocation, akin to a tape recording the tensor allocation pattern for DNN models. Fortunately, DNN model training is highly regularized, as each iteration processes identical model parameters and input data sizes. Therefore, after a few iterations, GMLake will no longer execute S2, S3, and S4. GMLake will only utilize the S1 "exact match" strategy for the remainder of the training, contrasting with the original caching allocator, which continuously requires splitting and merging operations.

**4.2.3 Robustness.** In practice, stitching and creating new sBlocks cannot occur infinitely due to the total capacity limitation on the GPU. Moreover, excessive stitching operations can impair the efficiency of the GMLake allocator when running allocation modules on sPool, such as `BestFit`. When the total capacity surpasses this limitation or threshold, GMLake employs `StitchFree` to release the LRU sBlocks and clear the pattern tape, thereby serving as a fallback mechanism for robustness.

Furthermore, when DNN training exhibits an extremely irregular pattern, it may generate numerous small blocks leading to frequent splits and stitches, causing early attainment of the limitation. To avoid unnecessary performance loss, a minimal fragmentation limit is established. If a block is smaller than this limit, GMLake will avoid stitching or splitting it. Hence, all algorithms and modules adhere to the fragmentation limit (e.g., 128 MB), to ensure high efficiency and robustness in DNN training.

## 5 Evaluation

We implement GMLake with 5000 lines of C++ code and integrate it into the caching allocator of PyTorch. We have adapted GMLake to different versions of PyTorch, such as PyTorch-1.13.1 and PyTorch-2.0.

We evaluate the performance of GMLake on fine-tuning several popular LLMs. We compare GMLake virtual memory allocator and PyTorch's caching allocator under diverse conditions, considering distinct training frameworks, GPU scalability, and combinations of optimization strategies. Through this analysis, we demonstrate the scalability of GMLake to adapt seamlessly to complex environments and its effectiveness in resolving memory fragmentation issues. Collectively, GMLake achieves a significant reduction in the fragmentation ratio of 15% on average and up to 33%, as well as a decrease in reserved GPU memory of 9.2GB on average and

| Model | Strategies | DDP Framework |
|---|---|---|
| **OPT-1.3b [87]** | L [29] R [86] O [69] | Deepspeed [15] |
| **GPT-2 [66]** | R O | Colossal-AI [44] |
| **GLM-10b [17]** | R O | FSDP [89] |
| **OPT-13b** [87] | L R O | Deepspeed |
| **Vicuna-13b [10]** | L R O | Deepspeed |
| **GPT-NeoX-20b [6]** | L R O | Deepspeed |

**Table 2.** Benchmark specifications. (L: LoRA; R: Recomputation; O: Offload; FSDP: Fully Sharded Data Parallel.)

up to **25GB**, which is obtained from 76 workloads within 8 different models .

### 5.1 Evaluation Methodology

***Testbed.*** The evaluation of GMLake is conducted utilizing two distinct setups: single-node multi-GPU and multi-node multi-GPU experiments. The single-node evaluations are performed on a server equipped with an Intel Xeon Platinum 8369B CPU boasting 1 TB of DRAM and eight NVIDIA A100 GPUs (80 GB memory for each). These GPUs are interlinked via NVLink, running on CUDA 11.4 and cuDNN 8.5. Correspondingly, the multi-node evaluations encompass two servers, each mirroring the configuration of the single-node server. In the finetuning phase, these two nodes engage in distributed training facilitated by RDMA.

***Training Scenarios.*** We evaluate GMLake across multiple prominent LLM optimization platforms, including Deepspeed [68] and FSDP [89], encompassing diverse optimization strategies such as LORA [29], gradient-checkpointing(i.e. Recomputation) [70], and offload [4, 69]. We focus on the fine-tuning scenario.

***Models and Datasets.*** We conduct evaluation on a set of representative open-source LLMs from their official examples. The exhaustive list of evaluated models, datasets, distributed data parallel (DDP) frameworks, and optimization strategies employed during the finetuning stage is presented in Table 2. Notably, the selection of default datasets from open-source LLM applications is based on the consideration that memory usage during finetuning remains unaffected by dataset quality.

***Baselines.*** We compare GMLake against PyTorch 2.0 with various LLMs training, e.g., Deepspeed [68] and Colossal-AI [44]. To our best knowledge, PyTorch caching allocator represents a state-of-the-art, application-agnostic memory allocator that demonstrates versatile compatibility across a range of deep learning applications. Notably, the transition between GMLake and PyTorch's original allocator is notably convenient by switching certain configurations. Additionally, GMLake is equipped with a specific set of hyper-parameters, empirically configured to achieve optimal performance outcomes through best practices.

***Metric.*** Different from the fragmentation metric FMFI [18, 41] for the virtual memory page with fixed length, the blocks used in GMLake can be split with arbitrary size. Therefore, we empirically define a specific fragmentation ratio for GMLake, which equals (1 − utilization ratio). To measure the memory fragmentation, we first calculate the memory utilization ratio, which equals peak active memory divided by peak reserved memory. The term "active memory" refers to the cumulative memory occupied by all active blocks, currently allocated by high-level tensors and utilized within DNN computations. On the other hand, "reserved memory" pertains to the total memory allocation set aside by both PyTorch and GMLake. These metrics are recorded at their respective peak values. To depict the relationship before and after the application of GMLake in terms of memory reduction, we calculate the arithmetic average result of the memory reduction ratio for multiple workloads. The formula to calculate the arithmetic average is

$$MemReductionRatio = \frac{\sum Reserved - \sum GMLakeReserved}{\sum Reserved}$$

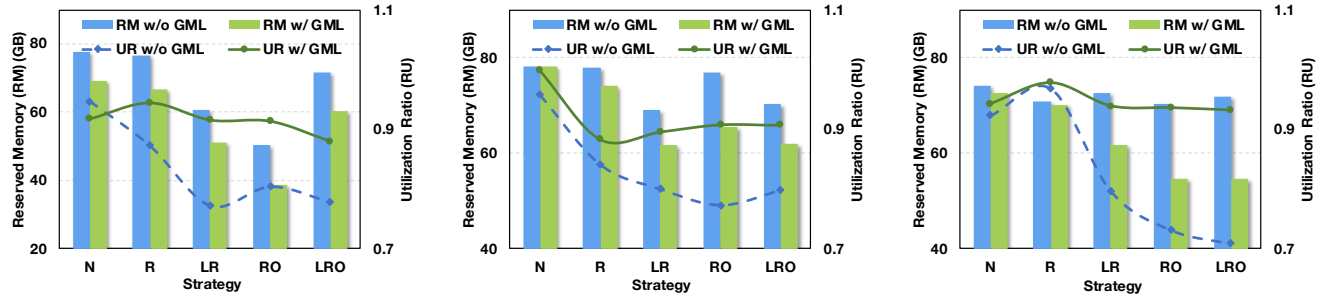
where "Reserved" and "GMLakeReserved" refer to the reserved memory of PyTorch and GMLake, respectively. While the reserved memory typically stabilizes as the training process runs, fluctuations in active memory are more substantial due to tensor (de)allocations during model execution. Given the caching mechanism, the peak active memory is pertinent for utilization or fragmentation analysis. We also employ throughput (expressed in samples per second) to quantify the speed of DNN training.

### 5.2 Scalability of GMLake

Firstly, we conduct serveral experiments to evaluate the scalability of GMLake across distinct memory-efficient strategies, GPU scale-out scenarios, and optimization platforms to address our observations in Section 2.

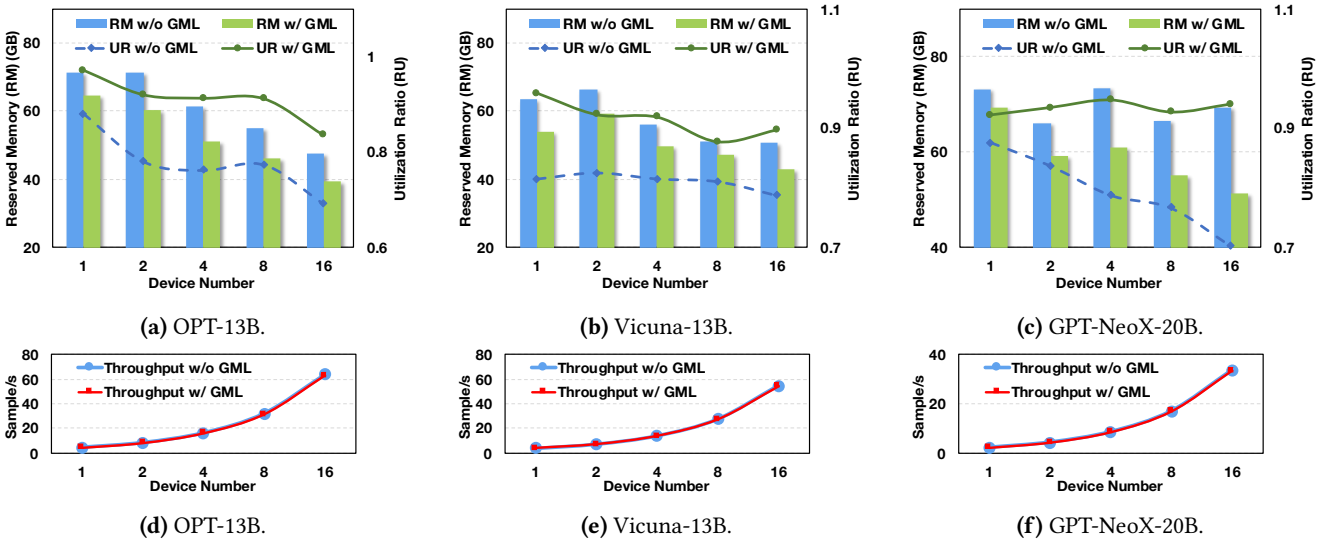
#### 5.2.1 Scalability on Memory-efficient Strategy.

To explore the scalability of GMLake in terms of memory-efficient strategies, we conduct finetuning experiments on OPT-13B [87], Vicuna-13B [10], and GPT-NeoX-20B [6] models using Deepspeed Zero3 [15] with four NVIDIA A100 (80 GB) GPUs, all under a common batch size. Notably, our evaluation entails influential memory-efficient strategies, including LoRA, gradient-checkpointing (recomputation), and offload. Thus, we systematically employ combinations of these strategies during our assessment. We label the no strategy scenario as **N**, recomputation as **R**, recomputation coupled with LoRA as **LR**, recomputation with offload as **RO**, and the joint utilization of recomputation, LoRA, and offload as **LRO**.




**(a)** OPT-13B. **(b)** Vicuna-13B. **(c)** GPT-NeoX-20B.

**Figure 10.** Comparison of memory utilization ratio on memory-efficient strategy combinations.




**(a)** OPT-13B. **(b)** Vicuna-13B. **(c)** GPT-NeoX-20B.

**(d)** OPT-13B. **(e)** Vicuna-13B. **(f)** GPT-NeoX-20B.

**Figure 11.** Comparison of memory utilization ratio on GPU scale-out.

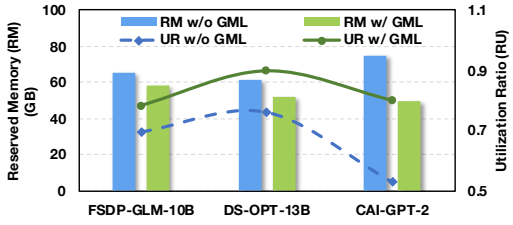
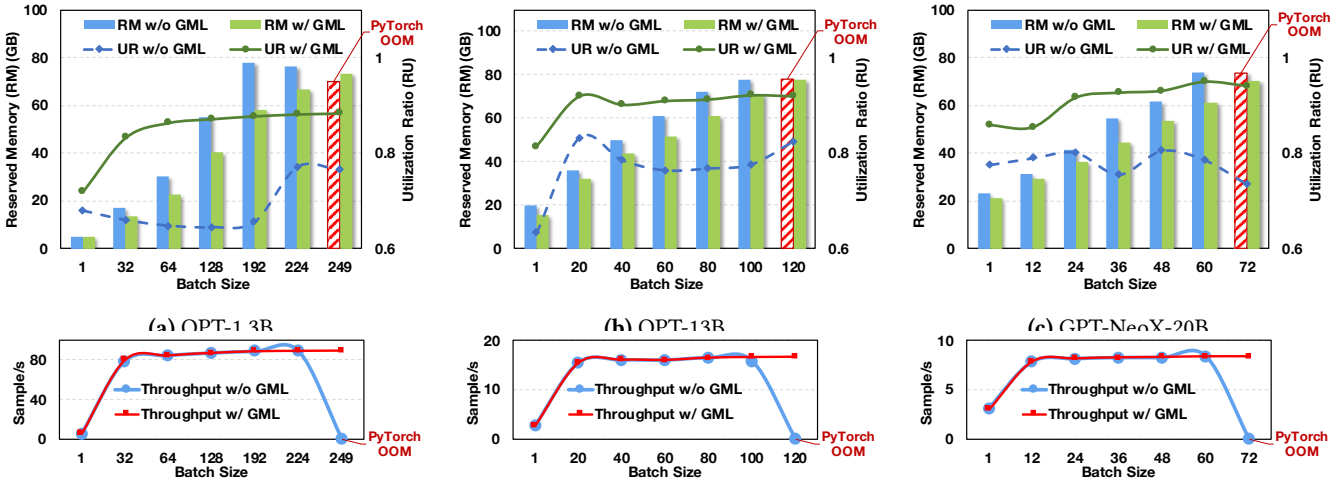



**Figure 12.** Memory utilization ratio comparison on various platforms.

***Complicated strategies lead to fragmentation.*** Figure 10 illustrates the utilization ratio and reserved memory consumption. In a comprehensive overview, the increase in utilization ratio and reduction in reserved memory, when contrasted with PyTorch, ranges from approximately 5% to 24% (or is around 10 GB and up to 17 GB), respectively. In contrast, GMLake effectively reduce this fragmentation ratio to 5% to 10%, mitigating the fragmentation issue.

When complex optimization strategies are employed, the consequent fragmentation ratio on PyTorch can exceed 20%. The contrast in utilization ratio between the application of these optimization strategies and their absence becomes evident. To illustrate, consider the recomputation strategy: it discards a portion of the activation tensor during the forward pass. This introduces a higher frequency of small memory allocation and deallocation operations, ultimately leading to fragmentation. A similar scenario arises with the offload strategy, where tensors frequently swap in and out between the CPU memory and the GPU memory, further increasing the frequency of memory allocation and deallocation operations. This situation highlights the need and features of GMLake: its design is transparent and effective with those increasingly complex optimization strategies.

**5.2.2 Scalability on GPU Scale-out.** Employing LR strategies along with the Deepspeed platform, we proceed to evaluate GMLake's scalability in the context of GPU scale-out. This evaluation entails scaling from 1 GPU to 16 GPUs incrementally. As depicted in Figure 11, GMLake consistently exhibits lower fragmentation ratios (high utilization ratio) and reserved memory consumption. Particularly noteworthy is the case of GPT-NeoX-20B in Figure 11c, where the utilization ratio and reserved memory reduction can be as




(a) OPT-1.3B (b) OPT-13B (c) GPT-NeoX-20B

(d) OPT-1.3B (e) OPT-13B (f) GPT-NeoX-20B

**Figure 13.** Comparison of memory utilization ratio and throughput on end-to-end effectiveness, utilizing varying batch sizes.

substantial as 23% or 17 GB, respectively. Moreover, the trend becomes evident that, as the number of GPUs increases, GMLake effectively maintains a utilization ratio of approximately 90% (i.e., fragmentation ratio less than 10%).

***GPU scale-out leads to fragmentation.*** While tremendous clusters speed up the training process, they also bring about more and more GPU memory fragmentation. Figure 11 presents the utilization ratio gradually decreases when the GPU number scales up from 1 to 16. Despite certain outliers, the Figure shows that the fragmentation ratio is enlarged regarding PyTorch. To put it in detail, the distributed data-parallel strategy causes this trend. DeepSpeed ZeRO-3 [67] means partitioning the optimization states, gradients, and weights. When more GPU is involved in training, the above tensor size will be smaller. It will cause many large blocks to split and cause more memory defragmentation. As shown in Figure 11 bottom, GMLake also maintains high throughput, similar to the original PyTorch. That indicates GMLake has excellent scalability with very low overhead.

**5.2.3 Scalability on Various Platforms.** We utilize various platforms including Deepspeed [68], FSDP [89], and Colossal-AI [44] to conduct finetuning on the OPT-13B [87], GLM-10B [17], and GPT-2 [66] models, respectively. This process employs static optimizing strategies, specifically LoRA and recomputation, and involves the use of four NVIDIA A100 (80 GB) GPUs. As illustrated in Figure **??**, the results demonstrate a noteworthy decrease in both fragmentation and reserved memory, with reductions ranging from approximately 9% to 33%, and from 7 GB to 25 GB, respectively. These results confirm the high scalability exhibited by GMLake on various optimized training platforms.

### 5.3 End-to-End Effectiveness of GMLake

In this study, we conduct a comparative analysis between GMLake and the PyTorch caching allocator through end-to-end fine-tuning of LLMs, utilizing varying batch sizes. This evaluation uses four A100 GPUs and enables LoRA, recomputation, and Zero3 optimizations on both frameworks. Figure 13 shows that GMLake consistently demonstrates a substantial reduction in peak memory consumption across a range of model sizes from 1.3 to 20 billion parameters. Notably, this memory consumption mitigation exhibits scalability with increasing model sizes while maintaining a consistent batch size.

The efficiency of memory usage is also demonstrated by the trends presented in Figure 13. Significantly enhanced performance is evident in comparison to the baseline. Notably, as the model size increases, memory efficiency reaches levels exceeding 95% (as seen with the 13 billion and 20 billion parameter models), indicating minimal fragmentation and waste. This starkly contrasts the baseline approach, which struggles to achieve an 80% efficiency rate.

GMLake can reduce the framgmentaion and provide more memory to run LLM without facing OOM errors. Especialy, in fig:eval:end, we can see OPT-1.3B, OPT-13B and GPT-NeoX-20B perform well with GMLake while encountering OOM errors with PyTorch's CUDA caching allocator in large batch scenarios, indicating effectiveness of GMLake.

Furthermore, we quantify the overhead of defragmentation logic overhead using the end-to-end throughput. Figure 13 bottom shows that GMLake effectively maintains comparable throughput to the baseline approach. Interestingly, in scenarios involving exceedingly large batch sizes, GMLake exhibits the potential to achieve even higher throughput than the PyTorch baseline due to its adept memory management and reduced frequency of (de)allocation operations.

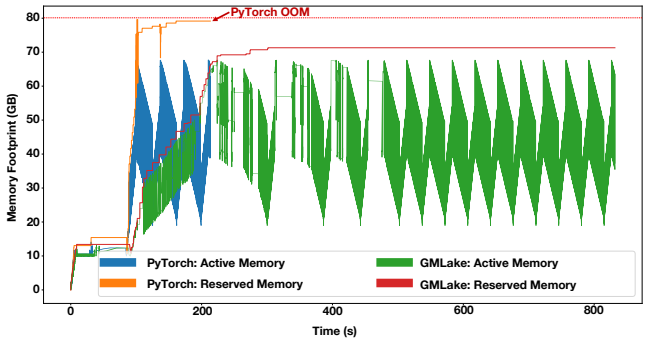



**Figure 14.** Memory trace on GPT-NeoX with batch size 72.

### 5.4 Memory Trace Analysis

Finally, we track the memory allocation behaviors on GPT-NeoX-20B on 4 GPUs with LoRA and recomputation strategies for PyTorch and GMLake, as shown in Figure 14.

There are three noteworthy points that highlights the advantages of GMLake. Firstly, PyTorch terminates at about 200 s owing to the OOM exception, while GMLake functions correctly for this batch size. Secondly, although the active memory of GMLake and PyTorch is of the same level, their reserved memory differs greatly, indicating the large fragmentation issue in PyTorch.

Thirdly, during the 100s to 400s, the active memory of both PyTorch and GMLake fluctuates regularly, showing the memory pattern in the LLM fine-tuning stage, especially in forward and backward passes. The rise of active memory represents the forward pass memory allocation, while the decrease of active memory stands for the backward pass. The interval between two similar patterns reflects the time cost of one single iteration.

Lastly, after four iterations, GMLake reaches stability and achieves the same throughput as PyTorch. That indicates that the allocation strategy used in GMLake can quickly adapt to the memory fluctuation in the forward/backward training passes, and find the best caching and stitching strategy.

We exploit the periodical nature of DNN training. Figure 14 shows that DNN training has a stable period in the training, where similar VMM allocation requests repeat. This periodicity presents the opportunity to reuse stiched sBlocks, amortizing their cost. To achieve that, we design the stitched memory pool (sPool) in GMLake to overscribe the sBlocks. When a new sBlock is created, we add it to the sPool. When the sBlock is freed, we still keep it presence. Next time when the same sBlock needs to be created, it can directly reuse the previously created sBlock. As long as we maintain enough sPool instances, all allocations only search for its best-fit sBlock without creating a new sBlock. We call it convergence, e.g., after four iterations of Figure 14.

## 6 Related Works and Discussion

We compare GMLake with existing works in two aspects: memory defragmentation and memory optimizations of LLMs.

***Memory Defragmentation.*** The memory defragmentation has been extensively investigated in various contexts [3, 35, 38]. In an effort to address fragment-related challenges, an early literature introduced a straightforward approach involving fine-grained fixed-sized chunks [73]. While this approach eliminates data movement overhead, it introduces increased access overhead and limited flexibility. To enhance both efficiency and flexibility, researchers have proposed compaction-based strategies, such as those involving the consolidation of multiple small chunks into a larger, contiguous unit through data movement [59, 79]. Other defragmentation techniques, including copy-based garbage collection systems [31, 53, 74], reduce complexity in data movement logic at the expense of temporary memory wastage. While sharing some conceptual similarities with the consolidation of small chunks into larger ones, GMLake adopts a stitching-based technique, which minimizes the need for frequent data movement and copying, resulting in a significant enhancement of memory efficiency. Beyond conventional memory systems, recent research has also explored the defragmentation for persistent memories [84].

***Efficient LLM.*** For Transformer-based LLMs, memory has emerged as a paramount resource within computing systems. The quadratic nature of attention mechanisms has led to a substantial surge in memory consumption for LLMs, thereby magnifying the significance of effective memory management [8, 16, 58]. Researchers have proposed various algorithmic optimizations aimed at curbing memory consumption, including quantization techniques [22, 24–26, 32, 46, 47, 81, 94], pruning strategies [19–21, 27, 39, 52, 64, 65, 80, 93], and KV-cache compression approaches [5, 37], compilation [9, 34, 90–92] and scheduling [11, 23, 50, 51, 54, 85].

There are various system-level memory optimizations. The vLLM work leverages page-based virtual memory management techniques to substantially enhance resource efficiency and serving throughput [82]. FlashAttention employs tiling techniques to optimize attention computation and notably mitigate memory consumption [13, 14]. The FlexGen framework introduces an optimization strategy to determine optimal memory-computation arrangements for efficient pipeline execution [71]. In this landscape, GMLake emerges as a user-transparent memory management system that orchestrates intelligent and efficient memory reuse.

***Novelty against other works.*** GMLake works on a unique memory scope for DNN training, which is different from the vLLM [82] and vMalloc [56]/CUDA VMM [62]. Neither them can solve the problem that the GMLake solved.

The vLLM is an algorithm-based solution for the Self-Attention operation and acts inside a tensor. LLMs originally

| Method | Scope |
|---|---|
| vLLM [82] | Tensor |
| GMLake | Memory Pool |
| vMalloc/CUDA VMM [10] | Physical Memory |

**Table 3.** Technical differences to related work.

pad all sequences with variable-length tokens to the maximum length and cause significant redundancy. vLLM uses a lookup table to remove the padded tokens. As such, vLLM is specific to self-attention while GMLake targets DNN training with a wider scenario. CUDA VMM, similar to vMalloc, is a low-level system tool for defragmentation. However, CUDA VMM/vMalloc cannot be aware of the memory pool commonly used in the DL framework. Without the memory pool design, the performance will significantly degrade. Therefore, these system-based memory tools cannot be directly implemented on the DL framework.

## 7 Conclusion

This study introduces GMLake, a new memory allocator with high efficiency and low fragmentation. Built upon the low-level CUDA virtual memory management knob, it consolidates multiple non-contiguous physical memory blocks into a singular, contiguous entity, thus mitigating the fragmentation issue. To facilitate seamless integration to existing DL frameworks, we formulate a virtual memory management API. Furthermore, we propose a multi-staged defragmentation strategies to guarantee the allocation efficiency and robustness. Our evaluation shows that GMLake reduces the memory fragmentation to 5% ~ 10% while maintaining the same throughput.

## Acknowledgements

This work was supported by the National Key R&D Program of China under Grant 2021ZD0110104, the National Natural Science Foundation of China (NSFC) grant (62222210, U21B2017, and 62072297), and the ANT Group Research Fund. The authors would like to thank the anonymous reviewers for their constructive feedback for improving the work. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

## References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Byungmin Ahn, Jaehun Jang, Hanbyeul Na, Mankeun Seo, Hongrak Son, and Yong Ho Song. Ai accelerator embedded computational storage for large-scale dnn models. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 483–486. IEEE, 2022.

[3] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 451–469. ACM, 2015.

[4] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Efficient combination of rematerialization and offloading for training dnns. *Advances in Neural Information Processing Systems*, 34:23844–23857, 2021.

[5] Amanda Bertsch, Uri Alon, Graham Neubig, and Matthew R. Gormley. Unlimiformer: Long-range transformers with unlimited length input. *CoRR*, abs/2305.01625, 2023.

[6] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. Gpt-neox-20b: An open-source autoregressive language model, 2022.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association, 2018.

[10] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023.

[11] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. Lazy batching: An sla-aware batching system for cloud machine learning inference. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 493–506. IEEE, 2021.

[12] Jack Choquette and Wish Gandhi. Nvidia a100 gpu: Performance & innovation for gpu computing. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–43. IEEE Computer Society, 2020.

[13] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *CoRR*, abs/2307.08691, 2023.

[14] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *NeurIPS*, 2022.

[15] DeepSpeed. Zero documentation, 2023.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[17] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. Glm: General language model pretraining with autoregressive blank infilling. *arXiv preprint arXiv:2103.10360*, 2021.

[18] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In *Ottawa Linux Symposium*, volume 1, pages 369–384. Citeseer, 2006.

[19] Yue Guan, Jingwen Leng, Chao Li, Quan Chen, and Minyi Guo. How far does bert look at: Distance-based clustering and analysis of bert ' s attention. *arXiv preprint arXiv:2011.00943*, 2020.

[20] Yue Guan, Zhengyi Li, Jingwen Leng, Zhouhan Lin, and Minyi Guo. Transkimmer: Transformer learns to layer-wise skim. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 7275–7286. Association for Computational Linguistics, 2022.

[21] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. Accelerating sparse dnn models without hardware-support via tile-wise sparsity. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[22] Cong Guo, Yuxian Qiu, Jingwen Leng, Xiaotian Gao, Chen Zhang, Yunxin Liu, Fan Yang, Yuhao Zhu, and Minyi Guo. SQuant: On-the-fly data-free quantization via diagonal hessian approximation. In *International Conference on Learning Representations*, 2022.

[23] Cong Guo, Yuxian Qiu, Jingwen Leng, Chen Zhang, Ying Cao, Quanlu Zhang, Yunxin Liu, Fan Yang, and Minyi Guo. Nesting forward automatic differentiation for memory-efficient deep neural network training. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 738–745. IEEE, 2022.

[24] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. OliVe: Accelerating Large Language Models via Hardware-friendly Outlier-Victim Pair Quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2023.

[25] Cong Guo, Chen Zhang, Jingwen Leng, Zihan Liu, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. ANT: exploiting adaptive numerical data type for low-bit deep neural network quantization. In *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*, pages 1414–1433. IEEE, 2022.

[26] Cong Guo, Yangjie Zhou, Jingwen Leng, Yuhao Zhu, Zidong Du, Quan Chen, Chao Li, Bin Yao, and Minyi Guo. Balancing Efficiency and Flexibility for DNN Acceleration via Temporal GPU-Systolic Array Integration. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[27] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[28] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and Efficient Pipeline Parallel DNN Training. *arXiv:1806.03377*, 2018.

[29] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[30] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 2019.

[31] Richard L. Hudson and J. Eliot B. Moss. Sapphire: copying GC without stopping the world. In Denis Caromel, John Reynders, and Michael Philippsen, editors, *Proceedings of the ACM Java Grande Conference, Stanford University*. ACM, 2001.

[32] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[33] Bernard J Jansen, Soon-gyo Jung, and Joni Salminen. Employing large language models in survey research. *Natural Language Processing Journal*, 4:100020, 2023.

[34] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[35] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In Simon L. Peyton Jones and Richard E. Jones, editors, *International Symposium on Memory Management, ISMM '98, Vancouver, British Columbia, Canada, 17-19 October, 1998, Conference Proceedings*. ACM, 1998.

[36] Mikhail Khalilov and Alexey Timoveev. Performance analysis of cuda, openacc and openmp programming models on tesla v100 gpu. In *Journal of Physics: Conference Series*, volume 1740, page 012056. IOP Publishing, 2021.

[37] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[38] Rune Krauss, Mehran Goli, and Rolf Drechsler. EDDY: A multi-core BDD package with dynamic memory management and reduced fragmentation. In Atsushi Takahashi, editor, *Proceedings of the 28th Asia and South Pacific Design Automation Conference, ASPDAC 2023, Tokyo, Japan, January 16-19, 2023*, pages 423–428. ACM, 2023.

[39] Eldar Kurtic, Elias Frantar, and Dan Alistarh. Ziplm: Hardware-aware structured pruning of language models. *CoRR*, abs/2302.04089, 2023.

[40] Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. *Advances in Neural Information Processing Systems*, 32, 2019.

[41] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 705–721, 2016.

[42] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv:2006.16668*, 2020.

[43] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[44] Shenggui Li, Jiarui Fang, Zhengda Bian, Hongxin Liu, Yuliang Liu, Haichen Huang, Boxiang Wang, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. *arXiv preprint arXiv:2110.14883*, 2021.

[45] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[46] Zhengyi Li, Cong Guo, Zhanda Zhu, Yangjie Zhou, Yuxian Qiu, Xiaotian Gao, Jingwen Leng, and Minyi Guo. Efficient activation quantization via adaptive rounding border for post-training quantization. *arXiv preprint arXiv:2208.11945*, 2022.

[47] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. AWQ: activation-aware weight quantization for LLM compression and acceleration. *CoRR*, abs/2306.00978, 2023.

[48] Linux man-pages project. *mmap(2) — Linux manual page*.

[49] Yiheng Liu, Tianle Han, Siyuan Ma, Jiayue Zhang, Yuanyuan Yang, Jiaming Tian, Hao He, Antong Li, Mengshen He, Zhengliang Liu, Zihao

Wu, Dajiang Zhu, Xiang Li, Ning Qiang, Dingang Shen, Tianming Liu, and Bao Ge. Summary of chatgpt/gpt-4 research and perspective towards the future of large language models, 2023.

[50] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, 2022.

[51] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[52] Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. In *Advances in Neural Information Processing Systems*, 2023.

[53] Simon Marlow, Tim Harris, Roshan P. James, and Simon L. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In Richard E. Jones and Stephen M. Blackburn, editors, *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008*. ACM, 2008.

[54] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[55] Silvano Martello, Michele Monaci, and Daniele Vigo. An exact approach to the strip-packing problem. *INFORMS journal on Computing*, 15(3):310–319, 2003.

[56] Microsoft. Virtualalloc function (memoryapi.h), 7 2022.

[57] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. Recent advances in natural language processing via large pre-trained language models: A survey. *ACM Computing Surveys*, 2021.

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

[59] openjdk. The z garbage collector. https://github.com/openjdk/zgc, 2023.

[60] Nathan Otterness and James H Anderson. Amd gpus as an alternative to nvidia for supporting real-time workloads. In *32nd Euromicro conference on real-time systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[61] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 2019.

[62] Cory Perry and Nikolay Sakharnykh. Introducing low-level gpu virtual memory management, 4 2020.

[63] Automatic Differentiation In Pytorch. Pytorch, 2018.

[64] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70. IEEE, 2020.

[65] Yuxian Qiu, Jingwen Leng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. Adversarial defense through network profiling based path extraction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[66] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[67] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[68] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong 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*, pages 3505–3506, 2020.

[69] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale model training. In *2021 USENIX Annual Technical Conference (ATC)*, 2021.

[70] Tim Salimans and Yaroslav Bulatov. Gradient checkpointing, 2017.

[71] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark W. Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. High-throughput generative inference of large language models with a single GPU. *CoRR*, abs/2303.06865, 2023.

[72] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[73] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for java. In *Proceedings of the 2000 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2000, San Jose, California, USA, November 7-18, 2000*, pages 9–17. ACM, 2000.

[74] David Siegwart and Martin Hirzel. Improving locality with parallel hierarchical copying GC. In Erez Petrank and J. Eliot B. Moss, editors, *Proceedings of the 5th International Symposium on Memory Management, ISMM 2006, Ottawa, Ontario, Canada, June 10-11, 2006*, pages 52–63. ACM, 2006.

[75] Peng Sun, Yonggang Wen, Ruobing Han, Wansen Feng, and Shengen Yan. Gradientflow: Optimizing network performance for large-scale distributed dnn training. *IEEE Transactions on Big Data*, 2019.

[76] TensorFlow. Bfc allocator, 2022.

[77] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[78] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[79] Ronald Veldema and Michael Philippsen. Parallel memory defragmentation on a GPU. In Lixin Zhang and Onur Mutlu, editors, *Proceedings of the 2012 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '12, Beijing, China, June 16, 2012*, pages 38–47. ACM, 2012.

[80] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. Dual-side sparse tensor core. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1083–1095. IEEE, 2021.

[81] Ziwei Wang, Jiwen Lu, Chenxin Tao, Jie Zhou, and Qi Tian. Learning channel-wise interactions for binary convolutional neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 568–577, 2019.

[82] Kwon Woosuk, Li Zhuohan, Zhuang Siyuan, Sheng Ying, Zheng Lianmin, Yu Cody, Gonzalez Joey, Zhang Hao, and Stoica Ion. vllm: Easy, fast, and cheap llm serving with pagedattention. https://vllm.ai/, 2023.

[83] Qifan Xu and Yang You. An efficient 2d method for training super-large deep learning models. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 222–232. IEEE, 2023.

[84] Yuanchao Xu, Chencheng Ye, Yan Solihin, and Xipeng Shen. FFCCD: fence-free crash-consistent concurrent defragmentation for persistent memory. In Valentina Salapura, Mohamed Zahran, Fred Chong, and

Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 274–288. ACM, 2022.

[85] Hailong Yang, Alex D. Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: precise online qos management for increased utilization in warehouse scale computers. In *The 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[86] Yimin Yang, QM Jonathan Wu, Xiexing Feng, and Thangarajah Akilan. Recomputation of the dense layers for performance improvement of dcnn. *IEEE transactions on pattern analysis and machine intelligence*, 42(11):2912–2925, 2019.

[87] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, et al. Opt: Open pre-trained transformer language models, 2022.

[88] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[89] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[90] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[91] Yangjie Zhou, Jingwen Leng, Yaoxu Song, Shuwen Lu, Mian Wang, Chao Li, Minyi Guo, Wenting Shen, Yong Li, Wei Lin, et al. ugrapher: High-performance graph operator computation via unified abstraction for graph neural networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 878–891, 2023.

[92] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, 2022.

[93] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–371, 2019.

[94] Bohan Zhuang, Mingkui Tan, Jing Liu, Lingqiao Liu, Ian Reid, and Chunhua Shen. Effective training of convolutional neural networks with low-bitwidth weights and activations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.