Reducing GPU Memory Fragmentation via Spatio-Temporal Planning for Efficient Large-Scale Model Training

Zixiao Huang*
Tsinghua University
Infinigence AI

Junhao Hu*
Infinigence AI

Hao Lin
Tsinghua University
Infinigence AI

Chunyang Zhu
Infinigence AI

Yueran Tang
Infinigence AI

Quanlu Zhang
Infinigence AI

Zhen Guo
Infinigence AI

Zhenhua Li
Tsinghua University

Shengen Yan
Tsinghua University
Infinigence AI

Zhenhua Zhu
Tsinghua University

Guohao Dai
Shanghai Jiao Tong University
Infinigence AI

Yu Wang
Tsinghua University

Abstract

The rapid scaling of large language models (LLMs) has significantly increased GPU memory pressure, which is further aggravated by training optimization techniques such as virtual pipeline and recomputation that disrupt tensor lifespans and introduce considerable memory fragmentation. Default GPU memory allocators of popular deep learning frameworks like PyTorch use online strategies without knowledge of tensor lifespans, which can waste up to 43% of memory and cause out-of-memory errors, rendering optimization techniques ineffective or even unusable.

To address this, we introduce STWeaver, a GPU memory allocator for deep learning frameworks that reduces fragmentation by exploiting the spatial and temporal regularity in memory allocation behaviors of training workloads. STWeaver introduces a novel paradigm that combines offline planning with online allocation. The offline planning leverages spatio-temporal regularities to generate a near-optimal allocation plan, while the online allocation handles complex and dynamic models such as Mixture-of-Experts (MoE). Built as a pluggable PyTorch allocator, STWeaver reduces fragmentation ratio on average by 79.2% (up to 100%) across both dense and sparse models, with negligible overhead. This enables more efficient, high-throughput training configurations and improves performance by up to 32.5%.

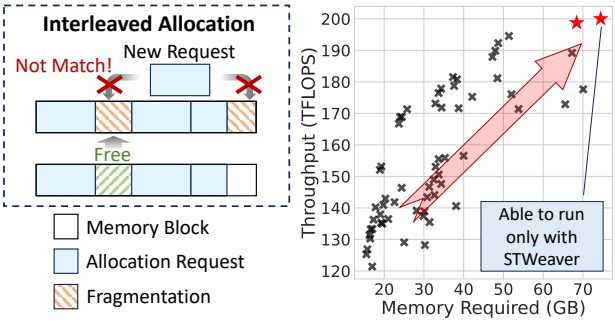


Figure 1. (a) Memory fragmentation in interleaved allocation. (b) Memory and training throughput of different training configurations for Llama2-7B on 8 NVIDIA A800 GPUs.

1 Introduction

In recent years, large-scale models, particularly large language models (LLMs) [1, 4, 15, 24, 44, 45, 47, 53], have demonstrated extraordinary performance in language comprehension, problem reasoning, code generation, etc. The scaling law [17] dictates that such powerful capabilities stem from the models' massive parameters and training data. As a result, nowadays even a medium-sized model such as Llama-3 [9] with 70 billion parameters requires more than 1 TB GPU/accelerator memory for training, placing heavy demands on the scarce and expensive GPU memory resource.

Additionally, current large-scale model training often employs a combination of various optimization techniques to enhance overall training efficiency. Such optimization techniques serve to either boost training throughput [31, 34] or reduce the theoretical GPU memory demand of the training [6, 18, 20, 27, 36]. For instance, the Virtual Pipeline [31] partitions a conventional pipeline parallel stage into several virtual stages, thereby minimizing idle periods (i.e., pipeline

*Both authors contributed equally to this research.

bubbles) inherent in pipeline parallelism. Furthermore, memory optimization techniques such as recomputation [6], tensor offloading [20, 27], and ZeRO [36] trade additional computation or transmission for reduced GPU memory usage.

However, the application of these training optimization techniques alters GPU memory allocation patterns. First, the number of allocation requests increases significantly compared to the training configuration without these techniques (e.g., 30% increase). Second, the allocation pattern shifts from a regular sequence of allocations followed by deallocations (e.g., activation tensors reserved for backward computation) to a more complex, interleaved pattern with frequent alternation between the two.

Unfortunately, the memory allocators in current deep learning frameworks, such as PyTorch [32], struggle to efficiently handle such complex allocation patterns, leading to severe memory fragmentation (up to 43% in typical scenarios). Consequently, the actual memory consumption during training significantly exceeds the theoretical allocation requirements. The root cause of fragmentation lies in the online best-fit allocation policy adopted by the allocator in popular deep learning frameworks (e.g., PyTorch). This policy allocates a requested tensor of a certain size to the most suitable memory slot without considering the tensor's lifespan, which is unknown to the allocator. Unpredictable deallocations lead to a discontinuous memory space, making it difficult to fit new tensors, as illustrated in Figure 1(a). Over time, this increases fragmentation as free space becomes scattered and less reusable for larger requests.

More critically, the increased GPU memory consumption caused by fragmentation can slow down model training. In large-scale training, configurations with higher throughput often require more GPU memory, as shown in Figure 1(b), where each point represents a different setup, i.e., using different optimization techniques. Fragmentation reduces the amount of available GPU memory, limiting the feasibility of high-throughput configurations. When such configurations are used, fragmentation can cause actual memory usage to far exceed theoretical estimates, leading to out-of-memory (OOM) errors. As a result, model developers are forced to revert to less efficient configurations, thus reducing training efficiency (e.g., up to 24.5%).

To address these problems, we propose STWeaver, a novel GPU memory allocator for deep learning frameworks to reduce fragmentation. Our approach is based on the observation that GPU memory requests exhibit strong consistency across training iterations. Therefore, by pre-assigning memory addresses before training, we can reduce fragmentation caused by online allocation in current allocators.

However, optimizing memory allocation requests ahead of training meets two challenges. First, offline allocation planning is NP-hard, known as Dynamic Storage Allocation problem [51]. In large-scale model training, the number of memory requests can exceed 10^5, making direct optimization intractable. To obtain a near-optimal solution within an acceptable time, we extract spatio-temporal regularities from memory allocation during training and use them to guide a grouping-based optimization. This grouping approach decomposes the time and space characteristics of memory requests, significantly reducing the complexity of the optimization problem.

Second, the recent emergence of sparse models of Mixture-of-Experts (MoE) models [15, 23, 24] introduces dynamics in memory allocation patterns compared to dense models. MoE models replace MLP layers with expert layers, and decide which experts to use for each token at runtime, which results in the dynamic nature of allocation request sizes. Consequently, we cannot rely on planning of certain address for the allocation requests. To address the challenge of dynamic request sizes, we propose a hybrid paradigm that combines offline planning with online allocation. By identifying reusable regions for dynamic requests before training and performing online allocation at runtime, STWeaver supports the dynamicity of allocation requests while maintaining a low fragmentation rate.

We implement STWeaver as a pluggable memory allocator for PyTorch and evaluate it across over 48 training configurations on 3 different testbeds. These configurations combine diverse dense and sparse models, model sizes, optimization techniques, microbatch sizes, and training frameworks. STWeaver reduces fragmentation memory by an average of 79.2% (up to 100%), saving up to 56.3GB GPU memory with negligible impact on end-to-end training throughput. By reducing peak GPU memory usage, it enables efficient training configurations that would otherwise trigger Out-of-Memory errors, resulting in an up to 32.5% throughput improvement. We will open source STWeaver to support more developers' efficient large-scale training.

This paper makes three main contributions:

- We conduct an in-depth analysis of the memory allocation characteristics and fragmentation problem of large model training, identifying spatial and temporal regularity in the allocation pattern.
- We propose a memory allocation paradigm for large-scale model training that combines offline planning with online allocation. STWeaver is capable of generating a near-optimal allocation plan based on spatio-temporal regularities, while effectively accommodating the dynamicity of allocation requests at runtime.
- We comprehensively evaluated STWeaver using diverse training configurations on different testbeds, demonstrating its wide applicability and effectiveness. It also enables more efficient model training.

2 Background and Motivation

2.1 Memory-driven Parallelism and Optimization

The evolution of distributed training parallelism has been driven by the critical need to fit increasingly large models into limited GPU memory. Early data parallelism (DP) strategies, which replicate the entire model, became infeasible for large-scale training. This led to model parallelism techniques such as tensor parallelism (TP) [39], which partitions weights, and pipeline parallelism (PP) [29–31, 34], which distributes layers, each with distinct memory trade-offs. To address the escalating memory demands from models like Mixture-of-Experts (MoE) or those with long sequences, more advanced methods emerged. These include expert parallelism (EP) [21] for distributing experts, sequence parallelism (SP) [18] for sharding activations, and ZeRO [36] optimizations, which partition optimizer states, gradients, and even weights. This progression clearly demonstrates that efficient GPU memory usage is the central consideration shaping the design of modern parallel training systems.

While parallelism strategies help distribute memory load across GPUs, the number of available GPUs is often limited. To make training fit within GPU memory, recomputation [6] and tensor offloading [20, 27, 37] are commonly used to reduce GPU memory usage, at the cost of slower training. Recomputation involves recalculating activation tensors within model layers during backpropagation rather than storing them, allowing for memory savings. The tensor offloading technique temporarily shifts tensors to CPU memory and retrieves them back when needed. Unfortunately, even with careful and reasonable combinations of parallelism, recomputation, and offloading, the desired training configuration often encounters the out-of-memory (OOM) error due to less effective usage of GPU memory, thus falling back to a less efficient training configuration.

2.2 Low Memory Efficiency in LLM Training

When training large models on GPUs, operators generate tensors of varying sizes and lifespans, which must be managed in GPU memory. These allocation requests pose significant challenges for memory allocators of current deep learning frameworks (e.g., PyTorch). Lacking prior knowledge of allocation patterns, allocators typically use online allocation strategy [32]. To reduce system call overhead (e.g., `cudaMalloc`), they often pre-allocate large caching blocks and slice out chunks based on best fit policy [40]. Over time, this leads to fragmented memory, i.e., unallocated regions too small or scattered to satisfy new requests, known as memory fragmentation. For clarity, we define *memory efficiency* as the ratio of the actual allocated tensor size to the reserved GPU memory size, which is $E = \frac{M_a}{M_r}$, where M_a is the size of allocated memory, representing the theoretical memory required under current training configuration; M_r is the total memory reserved by the allocator, representing the actual memory usage.

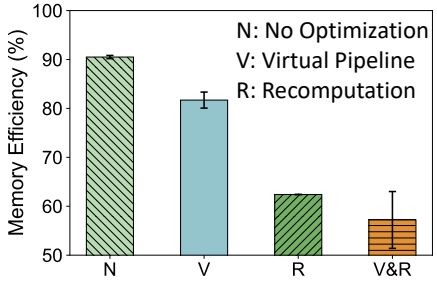


Figure 2. Comparison of PyTorch memory efficiency with no optimizations, recomputation, and Virtual Pipeline.

Large model training often leads to severe GPU memory fragmentation, especially when complex parallelism strategies are combined with memory optimization techniques (e.g., recomputation). Figure 2 shows the memory efficiency of GPT-2 (345M parameters) [35] trained on 8 NVIDIA A800-80G GPUs under different training configurations. The baseline uses 1F1B pipeline parallelism, achieves acceptable 90% memory efficiency with 52.1 GB of reserved memory. Using Virtual Pipeline Parallelism (VPP) [31] can improve training throughput. However, the utilization of VPP increases the allocated memory to 51.8 GB, and complicates the memory activities, reducing memory efficiency to 80%, leading to 59.9 GB of reserved memory. This higher memory usage can lead to OOM errors in some training scenarios. Recomputation is often used to mitigate memory requirement; however, while it reduces allocated memory, it also drops memory efficiency around 60%, causing significant memory waste. Therefore, memory fragmentation prevents logically effective approaches from achieving the expected memory reduction. Not only GPT-2, we found that many popular large models (e.g., Llama [47], Qwen [4]) suffer from serious memory fragmentation in training (see § 8).

Low Memory Efficiency Slows Training. Low memory efficiency often prevents more efficient parallelism strategies from fitting within available GPUs, which is a common challenge in large model training. As a case, we trained Qwen2.5-14B on 16 NVIDIA H200 GPUs, requiring at least 2-way tensor parallelism ($TP = 2$) to fit. We selected 2-way pipeline parallelism ($PP = 2$), 4-way data parallelism ($DP = 4$), enabled VPP to reduce bubble ratio for better training speed, but encountered OOM. To adapt, we tried three alternatives: (1) replacing VPP with 1F1B, but still occurs OOM, (2) enabling recomputation, and (3) increasing TP from 2 to 4. We also applied STWeaver to $TP = 2$, $PP = 2$, and $DP = 4$ with virtual pipeline enabled, which successfully fit in memory. The the alternative training configurations degrade training speed by 24.5% and 7.1% compared to STWeaver respectively, highlighting the critical role of memory efficiency in enabling high-performance parallelism strategies.

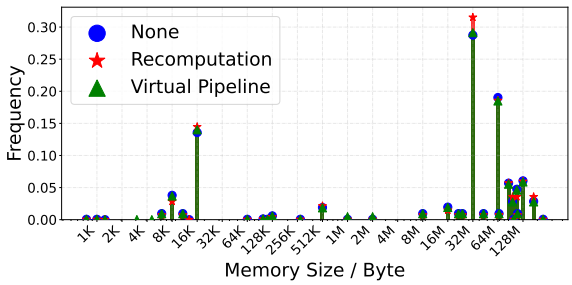


Figure 3. Allocation size distribution during training. As shown in the figure, there are only around 32 distinct tensor sizes among different training configurations.

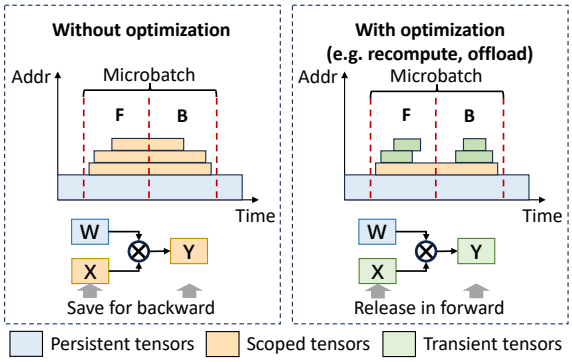


Figure 4. Allocation classification based on temporal characteristic. The temporal characteristic of activation tensors are influenced by training optimization techniques.

2.3 Memory Behavior Insights in LLM Training

Low memory efficiency stems from complex allocation and deallocation requests, making it difficult for online allocators to minimize fragmentation. While defragmentation techniques such as block merging [32] and virtual memory stitching [12] can help, it is either suboptimal or introduce performance overhead (see §8.3). Fortunately, large model training presents an opportunity to address this challenge. We observe that it generates a largely predictable and periodic pattern of allocation requests in both spatial and temporal dimensions, which we term *allocation regularity*. Although optimization techniques such as virtual pipeline and recomputation add complexity, the overall allocation behavior remains regular. This regularity can be proactively exploited by allocators to create low-fragmentation plans in advance. Notably, we identify regularity across both spatial and temporal dimensions, as detailed below.

Spatial Regularity. Modern large models are comprised of a stack of Transformer layers or identical sub-networks [35]. Consequently, the size of activation tensors generated during a training iteration exhibits significant repetition, which we call *spatial regularity*. As shown in the Figure 3, among over 50,000 tensor allocations with >512-byte size in a single training iteration of Llama2-7B, there are only 32 distinct tensor sizes. Notably, with optimizations like recomputation and virtual pipeline, the regularity still persists—around 32 different sizes for >512-byte tensor allocations.

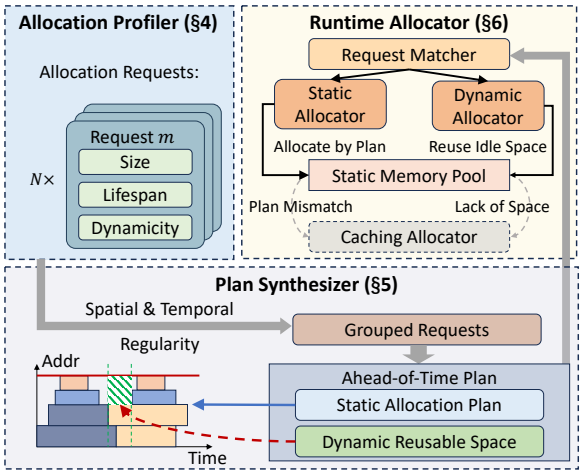


Figure 5. Workflow of STWeaver.

Temporal Regularity. We observe that tensor lifespans during language model training exhibit regular patterns, which can be categorized into three types as shown in Figure 4. *Persistent tensors*, such as model weights, gradients, and optimizer states, are allocated at the beginning of training and remain in GPU memory throughout the training process. *Scoped tensors* are allocated in one computation phase (the forward pass or backward pass of one microbatch) and released in another. This type of tensor is mainly activation tensors of forward computation and is used in backward computation. As shown in Figure 4, scoped tensors are allocated sequentially in the forward computation and released in reverse order during the backward pass. *Transient tensors*, such as intermediate input to unary operators (e.g., `ReLU`, `swiglu`), and activation tensors when training with optimization techniques like recomputation and offload, have very short lifespans and are released immediately after use, as they are not needed for backward computation. These temporal regularities can be effectively exploited in memory pre-planning to reduce inefficiencies caused by online decisions of allocation.

3 STWeaver Design Overview

STWeaver comprises three components (Figure 5): Allocation Profiler (§4), Plan Synthesizer (§5), and Runtime Allocator (§6). To generate an ahead-of-time GPU memory allocation plan, the initial step is to use the Allocation Profiler to capture the temporal (lifespan), spatial (size), and dynamicity information of all memory (allocation or free) requests within a

training iteration. The request information is then fed to the Plan Synthesizer to generate an allocation plan. To this end, the Plan Synthesizer first groups the requests to reduce planning complexity based on their spatio-temporal regularities. For static requests with fixed allocation size and lifespan, a *Static Allocation Plan* that minimizes memory fragmentation is generated leveraging the grouping results. To handle dynamic requests with unpredictable allocation pattern, the Plan Synthesizer finds idle spaces (termed *Dynamic Reusable Space*) within the *Static Allocation Plan* that can be reused by dynamic requests later at runtime to further reduce fragmentation. During training, the Runtime Allocator is used to perform the actual memory allocation, which consists of a Static Allocator and a Dynamic Allocator. The Static Allocator handles static requests based on the *Static Allocation Plan*, while the Dynamic Allocator attempts to allocate dynamic requests within the *Dynamic Reusable Space* if possible. For dynamic requests that cannot be accommodated by the *Dynamic Reusable Space*, and any unexpected requests, the Runtime Allocator falls back to a caching allocator.

4 Allocation Profiler

As described in §3, the Allocation Profiler traces each torch-level memory allocation and free request to capture its spatial, temporal, and dynamicity information. Notably, apart from the basic information like request timestamp, address, size, and dynamicity, the Allocation Profiler also records training-level information including the current computation phase (forward or backward), micro-batch ID, and the module name that issues the request to facilitate the identification of spatio-temporal regularities.

Formally, we organize an allocation request and its associated free request into a memory request event m, which is defined as $m := (s, t_s, t_e, p_s, p_e, d)$. Here, s represents the request size; t_s and t_e are the allocation and free timestamps of the memory chunk, respectively; p_s and p_e identify the computation phases of allocation and free, respectively; and d is a boolean flag indicating if the request originates from a dynamic layer (e.g., a MoE expert layer). For requests from dynamic layer (where $m.d = True$), two additional elements l_s and l_e are recorded, which are the originating module name for the allocation and free, respectively. This additional information allows us to group dynamic requests based on their temporal regularity, further details are provided in §5.2. Upon completion, the profiler outputs a list $\mathcal{M}$ of these characterized allocation requests, which is the primary input of the Plan Synthesizer.

5 Plan Synthesizer

The goal of the plan synthesizer is to produce a low fragmentation memory allocation plan that maximizes memory efficiency E as defined in §2.2. Since allocated memory M_a is fixed for a specific training configuration, the goal is then to minimize reserved memory M_r. To this end, the input of the synthesizer $\mathcal{M}$ is first partitioned into two subsets $\mathcal{M}_s := \{m | m \in \mathcal{M}, m.d = False\}$ and $\mathcal{M}_d := \{m | m \in \mathcal{M}, m.d = True\}$ according to their dynamicity. For $\mathcal{M}_s$ containing static request events, we perform static allocation planning to generate the *Static Allocation Plan*. Next, for dynamic request events $\mathcal{M}_d$, we find idle space in the plan (called *Dynamic Reusable Space*) that can be used to handle dynamic requests at runtime.

5.1 Static Allocation Planning

The *Static Allocation Plan*, denoted as $\mathcal{D}_s$, consists of a list of allocation decisions. Each allocation decision $d \in \mathcal{D}_s$ incorporates the six attributes of m and is augmented with an additional attribute, a, which denotes the start address of the allocated memory chunk, i.e., $d := m + (a) = (s, t_s, t_e, p_s, p_e, d, a)$. The allocation planning is then under the constraint that for any two allocation decisions d_i and d_j, they cannot simultaneously have conflicting lifespans and conflicting address ranges. Otherwise, they will have intersecting memory and result in memory stomping.

Since finding the optimal allocation plan is NP-hard and involves a large input scale as described in §1, brute-force methods or any pruning techniques that do not fundamentally reduce the complexity of the original search space [25, 28] are infeasible. Inspired by the spatio-temporal regularity we uncover in §2.3, our idea is to decouple the searching in the space (i.e., memory address and size) and time (allocation time and free time) dimensions. At a high level, we first group memory allocation requests with the same temporal or spatial characteristics, devise a local plan for them to reduce the problem space, and then perform a global planning to reach the final plan. During the local and global planning, we greedily combine the optimal local memory layout for each request size to approximate the global optimal solution. We devise special abstractions for the temporal and spatial groups in the above algorithmic workflow, namely *HomoPhase Group* and *HomoSize Group*, respectively.

HomoPhase Group Planning. A *HomoPhase Group* $\mathcal{M}_g$ contains allocation requests that start and end in the same computation phases, which is: $\mathcal{M}_g := \{m \in \mathcal{M}_s \mid m.p_s = P_s,\ m.p_e = P_e\}$. Here, P_s and P_e denote a pair of computation phases (e.g., forward/backward passes), meaning all requests in $\mathcal{M}_g$ share similar lifespans.

Since their lifespans overlap, packing them contiguously into a single memory block achieves local optimal. However, since their lifespans are only partially aligned, some memory may remain unused during parts of the timeline—these gaps are called *spatio-temporal bubbles*, causing memory fragmentation.

To reduce such bubbles, we fuse adjacent groups when the end phase of one matches the start phase of another. The merged group can better reuse memory across phase

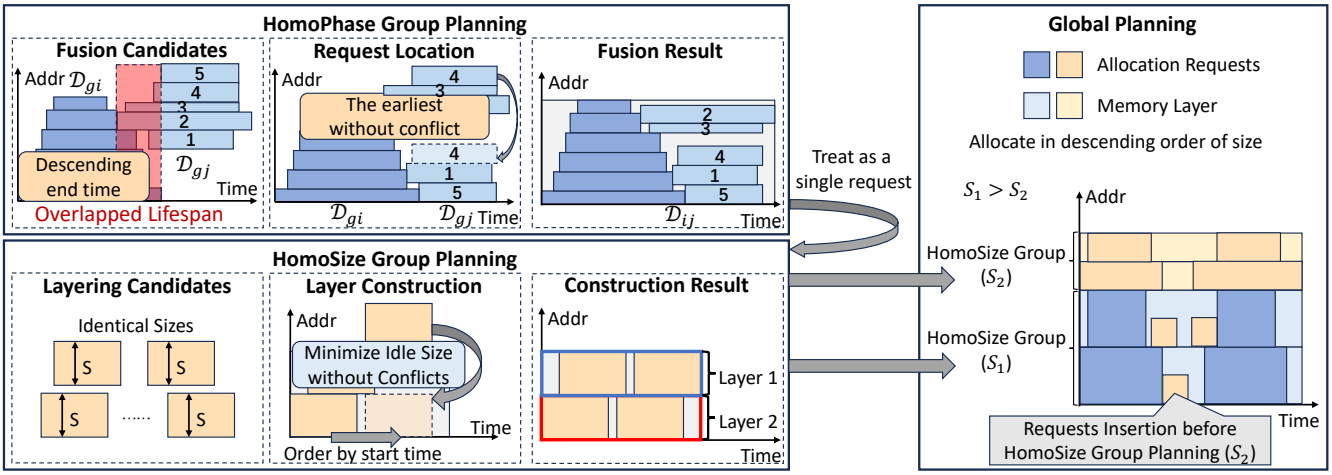


Figure 6. Static Allocation Planning of allocation requests. Allocation requests are first grouped based on temporal characteristics into *HomoPhase Groups* for intra-group planning (upper left), and then further grouped based on spatial characteristics into *HomoSize Groups* (bottom left). During global planning, *HomoSize Groups* are inserted and placed in descending order of allocation size (right).

boundaries. We evaluate memory efficiency using the *time-memory product* (*TMP*) [40]:

$$\begin{aligned} TMP &= \frac{\sum_{d\in\mathcal{D}_g} d.s\cdot(d.t_e - d.t_s)}{\mathcal{D}_g.s\cdot(\mathcal{D}_g.t_e - \mathcal{D}_g.t_s)}, \\ \mathcal{D}_g.s &= \max_{d\in\mathcal{D}_g}(d.a + d.s), \\ \mathcal{D}_g.t_s &= \min_{d\in\mathcal{D}_g} d.t_s, \quad \mathcal{D}_g.t_e = \max_{d\in\mathcal{D}_g} d.t_e. \end{aligned} \tag{1}$$

The numerator measures actual memory-time usage; the denominator reflects the reserved memory over time. A higher *TMP* indicates fewer bubbles.

As shown in Figure 6, we fuse two local plans $\mathcal{D}_{gi}$ and $\mathcal{D}_{gj}$ by inserting the smaller one into the larger. Assume $\mathcal{D}_{gi}.s > \mathcal{D}_{gj}.s$; we sort $\mathcal{D}_{gi}$ by end time in descending order and try placing each $d_j \in \mathcal{D}_{gj}$ at the lowest available address *addr*, starting from: $addr = \min_{d_i\in\mathcal{D}_{gi}} d_i.a$. At each step:

1. Choose the earliest-starting d_j that fits without conflict and place it at *addr*. Update $addr \leftarrow addr + d_j.s$.
2. If no fit is found, move *addr* to the next $d_i.a$ in $\mathcal{D}_{gi}$:

$$addr \leftarrow \min_{d_i\in\mathcal{D}_{gi},\ d_i.a>addr} d_i.a.$$

The fusion is accepted if the new *TMP* improves over the weighted average of the originals, meaning fewer bubbles.

Each (possibly fused) group then forms a local plan $\mathcal{D}_g$, where requests are given relative addresses. We treat this plan as a single large request m_g for global planning:

$$m_g.s = \mathcal{D}_g.s, \quad m_g.t_s = \mathcal{D}_g.t_s, \quad m_g.t_e = \mathcal{D}_g.t_e.$$

HomoSize Group Planning. Allocation requests exhibit strong repetitiveness, with many requests having identical allocation sizes and differing only in their lifespan. This observation continues to hold true after *HomoPhase Group* planning, primarily because each microbatch exhibits identical behavior during training. Therefore, the *HomoPhase Group* formed through temporal grouping and fusion also possesses the characteristic that multiple such groups are identical in size, differing only in their lifespan.

Based on this observation, we propose a new abstraction termed *HomoSize Group*, which aggregates allocation requests of the same size property. For requests of a specific size S, there are only differences in their lifespan. Therefore, any subset of these requests with non-overlapping lifetimes can reuse the same space in GPU memory. In the time-space coordinate system, this shared space can be regarded as a layer within the memory space, referred to as a memory layer. To obtain a local optimal allocation plan for memory requests within a *HomoSize Group*, we need to minimize the number of memory layers required to allocate all requests.

Algorithm 1 describes the procedure of constructing memory layers for *HomoSize Groups* of a specific size S. To begin with, the allocation requests of size S are included in a *HomoSize Groups* $\mathcal{M}_s$, and are sorted by their allocation time. Next, for each allocation request $m \in \mathcal{M}_s$, we try to find a memory layer whose last allocation request's free time is closest to but smaller than m's allocation time, so as to minimize the idle time of the layer while avoiding conflicting lifespans. If we can find such a layer, m is appended to the layer's tail. Otherwise, a new layer is constructed and is populated by m. In this way, we minimize both the intra-layer gaps and the total number of layers.

Global Planning. In the local planning, memory requests are grouped based on their temporal and spatial regularities into *HomoPhase Groups* and *HomoSize Groups*, and generate a local memory allocation plan for each group. These local

Algorithm 1: Memory Layer Construction for *HomoSize Group*.

Input : Request Set $\mathcal{M}_s = \{m | m.s = S, m \in \mathcal{M}\}$
Output: MemoryLayer List $\mathcal{L} = (\mathcal{M}_{l1}, \mathcal{M}_{l2}, ...)$

$\mathcal{L} \leftarrow [\,]$;
$\mathcal{M}_s$.sort(key=$m.t_s$);
for $m \in \mathcal{M}_s$ **do**
  $\mathcal{M}_l \leftarrow \max_{L.end}\{L \in \mathcal{L} : L.end < m.t_s\}$;
  if $\mathcal{M}_l$ *is None* **then**
    $\mathcal{M}_l \leftarrow$ new MemoryLayer(size=$m.s$);
    $\mathcal{L}$.append($\mathcal{M}_l$);
  end
  $\mathcal{M}_l$.append(m);
  $\mathcal{M}_l.end \leftarrow m.t_e$;
end

plans can then be regarded as building blocks of the global memory allocation plan, as shown in Figure 6.

Specifically, the first step in generating the global allocation plan is to partition all memory requests within one training iteration into different *HomoPhase Groups* based on their temporal characteristics. Adjacent *HomoPhase Groups* are then merged to produce local plans, denoted as $\mathcal{D}_g$. These local plans are subsequently treated as unified memory requests in the next stage of spatial grouping, where they are classified into different *HomoSize Groups* according to their allocation sizes. Each *HomoSize Group* then constructs memory layers within the group. We execute the construction in descending order of the request size within the *HomoSize Group*, since smaller memory requests may fit into the unused intervals of memory layers allocated for larger requests, thus improving overall memory efficiency. Therefore, before constructing memory layers for size S_i (at which point all larger groups have already been processed), we first attempt to insert the requests of size S_i into the free intervals of existing larger memory layers. Any remaining requests of size S_i that cannot be placed in the larger layers are then assigned new memory layers by executing the Algorithm 1. Finally, after all layers have been constructed, each memory request is assigned a specific address within the global allocation plan.

5.2 Locating Dynamic Reusable Space

Dynamic allocation requests ($\mathcal{M}_d$) are characterized by sizes determined only at runtime, necessitating online allocation. A key insight from our profiler analysis is that the peak memory usages of static and dynamic allocations typically do not occur simultaneously. Therefore, managing static and dynamic allocations in separate regions—where each region is provisioned for its respective peak—results in memory fragmentation. For instance, when dynamic memory usage peaks, the dedicated static memory region often contains substantial idle capacity. This observation of underutilized memory, a direct consequence of misaligned peak demands under segregated management, forms the primary motivation for our proposed dynamic planning strategy.

To improve memory efficiency and reduce peak memory consumption during training, dynamic requests should reuse idle spaces within the *Static Allocation Plan* as much as possible. However, allocating dynamic requests directly within the available spaces of the *Static Allocation Plan* at runtime may lead to memory stomping. This occurs because the current dynamic allocation request might overlap in address space with subsequent static allocations that are already planned. We observe that, although the sizes of dynamic allocations are unpredictable, their lifespan are relatively fixed. Leveraging this temporal regularity, we can identify reusable regions within the *Static Allocation Plan* before training, providing guidance for online allocation at runtime.

Leveraging the predictable lifetimes of dynamic memory allocations, our approach proactively identifies reusable memory regions within the *Static Allocation Plan* before runtime. To achieve fine-grained temporal precision for these dynamic requests—in contrast to the computation-phase granularity used for static allocations—we operate at the model layer level. This refined granularity enables a more precise interrogation of *Dynamic Reusable Space*. Such accurately identified regions within the typically shorter layer-level intervals tend to be more effectively utilized, maximizing opportunities for dynamic memory reuse and thereby reducing the peak GPU memory footprint. We characterize each dynamic request by its malloc layer, l_s, and its free layer, l_e (profiling methodology in §4). This (l_s, l_e) pair establishes a bounding temporal interval, from l_s's start to l_e's end, which contains the lifespan of the dynamic allocation. To systematically manage these lifetimes, we classify all dynamic allocation requests into distinct groups, called *HomoLayer Group*, where each group $\mathcal{G}$ comprises requests sharing identical (l_s, l_e) pairs:

$$\mathcal{G}(a, b) := \{m | m.l_s = a, m.l_e = b\} \tag{2}$$

where a and b represent for the dynamic layers in the model. For every such group of dynamic requests $\mathcal{G}(a, b)$, and its corresponding temporal range $\mathcal{T}(a, b) = [a.start, b.end]$, we then interrogate the pre-established *Static Allocation Plan* $\mathcal{D}_s$. The objective of this interrogation is to identify all contiguous memory segments that remain idle throughout the entirety of this specific temporal range. In the *Static Allocation Plan* $\mathcal{D}_s$, each decision d contains a static allocation request m and its allocate address a, indicating the spatial and temporal occupation space for d is $R_s(d) = [d.a, d.a + d.s]$ and $R_t(d) = [d.t_s, d.t_e]$ respectively. The occupied address ranges for $\mathcal{T}(a, b)$ can be represented as:

$$\mathcal{A}_o(a, b) = \bigcup_{d \in \mathcal{D}_s,\ R_t(d) \cap \mathcal{T}(a,b) \neq \emptyset} R_s(d) \tag{3}$$

The *Dynamic Reusable Space* $\mathcal{A}_i$ ranges during $\mathcal{T}(a, b)$ are the complement of all addresses $\mathcal{A}$ occupied in the allocation plan, as shown in Eq. 4 and Eq. 5.

$$\mathcal{A} = [\min_{d \in \mathcal{D}_s}(d.a), \max_{d \in \mathcal{D}_s}(d.a + d.s)] \tag{4}$$

$$\mathcal{A}_i(a, b) = \mathcal{A} \setminus \mathcal{A}_o(a, b) \tag{5}$$

The identified *Dynamic Reusable Space* $\mathcal{A}_i$ are subsequently designated as candidate reusable regions. At runtime, when a dynamic allocation request belonging to a particular (l_s,l_e) group arises, the allocator can preferentially utilize these pre-vetted regions, thereby ensuring that dynamic allocations are placed in memory spaces that will not conflict with future, planned static allocations.

6 Runtime Allocation

The runtime allocator manages the GPU memory and serves allocation requests based on the allocation plan generated by the plan synthesizer. It consists of two main components, a static allocator that handles allocation requests *without* runtime dynamics (i.e., $m.d == False$), and a dynamic allocator that handles allocations *with* runtime dynamics (i.e., $m.d == True$). During runtime, when an allocation request is received by the Runtime Allocator, the Request Matcher routes the request to an appropriate allocator based on the dynamic characteristics of the current model layer (detail shows in §7). Furthermore, to address scenarios such as potential mismatch between actual runtime allocation requests and the *Static Allocation Plan*, or instances of inadequate *Dynamic Reusable Space* for dynamic requests, STWeaver's runtime allocation further incorporates a caching allocator. This component is designed to manage these exceptional cases, thereby guaranteeing the overall robustness of the system.

6.1 Static Allocator

The static allocator, guided by the *Static Allocation Plan*, reserves a static memory pool prior to training, where the size of the memory pool is fixed, defined by the result of *Static Allocation Plan*. At runtime, it efficiently serves static requests by providing pre-planned memory addresses sequentially. This eliminates the need for online allocation searches found in systems like PyTorch.

6.2 Dynamic Allocator

Certain models, such as the Mixture-of-Experts (MoE), exhibit non-deterministic memory patterns, making it impossible to pre-plan memory addresses for all tensors. To handle these cases, we employ a dynamic allocator that assigns memory at runtime.

The primary strategy of the dynamic allocator is to prioritize reusing memory from the static memory pool, which was pre-allocated for predictable requests. To prevent conflicts, STWeaver meticulously tracks all currently available address intervals ($\mathcal{A}_a$) in this pool. When any memory block is allocated or freed, $\mathcal{A}_a$ is updated accordingly.

When a dynamic request m arrives, the allocation process begins by first identifying the *Dynamic Reusable Space* $\mathcal{A}_i$, which is the available space in static memory pool for the *HomoLayer Group* contains m. Since prior allocations may have already occupied parts of this space, we must identify the portions that are still free. To find the actual memory available for allocation, STWeaver computes the intersection of this potential space $\mathcal{A}_i$ with the currently free intervals $\mathcal{A}_a$. This calculation yields a set of candidate intervals, $\mathcal{A}_c(m)$:

$$\mathcal{A}_c(m) = \mathcal{A}_a \cap \mathcal{A}_i \tag{6}$$

From these candidate intervals, we apply the best-fit policy to select the most suitable one for the request. Once an interval is chosen and the memory is assigned, the system updates the list of available intervals $\mathcal{A}_a$ to reflect the allocation.

If no candidate interval in the static pool can satisfy the request, the system falls back to the caching allocator as a secondary option.

7 Implementation

STWeaver is implemented for PyTorch using about 5700 lines of Python and 3700 lines of C++. The plan synthesizer is implemented as a standalone tool, while the profiler and allocator are implemented as PyTorch's PluggableAllocator [33], which can be loaded before training to take over the `malloc` and `free` API calls. This means that STWeaver is compatible with any PyTorch version and GPU platform that supports the PluggableAllocator interface. To capture temporal and spatial characteristics, STWeaver employs monkey patching for lightweight instrumentation, requiring no more than five lines of code in the original training framework.

Allocation Profiler. The profiler is designed to log tensor allocation requests made by PyTorch-based model training frameworks. It interfaces directly with native GPU memory allocation APIs, such as `cudaMalloc` and `cudaFree` for NVIDIA GPUs. This approach ensures that memory is allocated precisely as required, thereby almost entirely obviating memory fragmentation under these conditions. Consequently, the profiler can trace GPU memory for training configurations that would lead to out-of-memory (OOM) errors with PyTorch's default allocator. If an OOM error occurs even when using these native GPU APIs for profiling, it indicates that the configuration's theoretical memory demand inherently surpasses the GPU's memory capacity, rendering it impossible to execute irrespective of fragmentation.

Runtime Allocator. At runtime, the allocator performs memory allocation according to the allocation plan. During training initialization, STWeaver uses native GPU memory allocation APIs to preallocate a contiguous memory block equal in size to the static memory pool and also initializes a caching allocator as a fallback. The runtime allocator then assigns address ranges within the static memory pool without

issuing additional GPU memory API calls, thereby avoiding extra runtime overhead. To identify the current model layer during execution and route memory requests to the appropriate allocator, STWeaver leverages PyTorch's hook APIs to track the execution of model modules. When a memory request arrives at runtime, the Request Matcher in the Runtime Allocator uses the current module information to determine whether the request should be handled by the static allocator according to the allocation plan, or by the dynamic allocator for online allocation.

8 Evaluation

To gain an in-depth understanding of STWeaver, we focus on the following aspects in the evaluation. **(1) Performance.** We show that STWeaver can reduce fragmentation memory by 79.2% on average (up to 100%), saving up to 56.3GB GPU memory across dense/sparse models trained with a variety of frameworks, configurations, and scales. **(2) Overhead.** We demonstrate that STWeaver's impact on end-to-end training throughput is negligible in all cases, and our plan synthesizer can efficiently produce an allocation plan in minutes even under complex allocation requests. **(3) Performance Breakdown.** We study the individual performance of the static and dynamic allocators, and show their impacts on the final performance of STWeaver.

8.1 Experimental Setup

Testbed. STWeaver is evaluated on both NVIDIA and AMD GPU platforms. One configuration consists of 1 node equipped with an Intel Xeon Platinum 8358 128-Core CPU and 8 NVIDIA A800-80GB GPUs, which is used to evaluate various training optimization setups. The other has up to 16 nodes, each equipped with an Intel Xeon Platinum 8558 192-Core CPU and 8 NVIDIA H200-141GB GPUs, and is used for scalability evaluation. The AMD GPU platform has 8 nodes, each of which is equipped with AMD EPYC 7K62 48-Core Processor and 8 AMD MI210-64GB GPUs.

Models. We evaluate STWeaver on 7 representative large-scale dense and sparse Mixture-of-Expert (MoE) models. For dense models, we choose GPT-2 [35] and Llama2-7B [47] for experiments on multiple training configurations. We use four models of varying sizes (including 7B, 14B, 32B, 72B) from the Qwen2.5 [52] series to demonstrate the scalability of our approach with respect to both model size and cluster size. For sparse models, we choose Qwen1.5-MoE-A2.7B [46], a MoE model with 16 billion parameters to evaluate the efficiency of STWeaver on both multiple configurations as well as scalability and extendability on AMD platform.

Training Setup. We evaluate STWeaver with multiple training setups, in terms of training frameworks and training optimization techniques. For training frameworks, we choose the popular Megatron-LM [39], Megatron-DeepSpeed [42], and Colossal-AI [22]. For training optimizations, we choose the pipeline parallelism schedule of Pipedream-1F1B [29], Virtual Pipeline [31] as parallelism-based optimizations. For non-parallelism-based optimizations, we consider activation recomputation [6], offloading [38], and distributed optimizer (ZeRO [36]), which contains all kinds of memory optimizations [8].

Baselines. We compare STWeaver with state-of-the-art baselines, including:

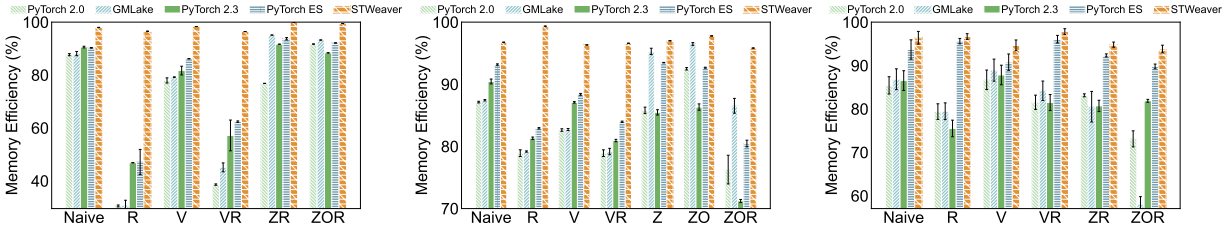
- **PyTorch [32].** PyTorch employs a caching memory allocator for GPU memory management. It reduces the overhead of frequent native GPU API calls by reusing previously freed memory blocks, improving performance and memory efficiency.
- **PyTorch expandable_segments (PyTorch ES) [33].** The expandable_segments allocator in PyTorch introduces support for virtual memory, allowing memory segments to grow dynamically as needed. This feature is only available in PyTorch versions 2.1 and above.
- **GMLake [12].** GMLake leverages virtual memory stitching to unify non-contiguous memory blocks into a single virtual space for defragmenetation. We deployed it using the official Docker image provided in its repository [11], whose PyTorch version is 2.0.

Metrics. We evaluate the performance of STWeaver using three key metrics. First, memory efficiency is the ratio of the max allocated memory to the max reserved memory as explained in §2.2. Building on this, the fragmentation ratio represents the proportion of reserved memory that is not actually utilized, which equals to (1 - memory efficiency). To measure the end-to-end throughput of training and evaluate the overhead of STWeaver, we choose FLOPS (floating point operations per second) as the throughput metric, which is calculated by training frameworks per training iteration.

8.2 Memory Efficiency and Defragmentation

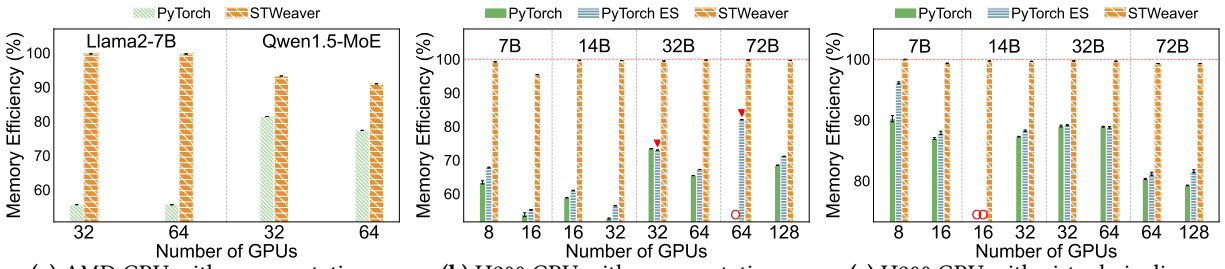
Models and Optimization Techniques. For the model and optimization combination test, the micro-batch sizes are set to the maximum feasible size that will not cause OOM following common practices [31], i.e., 128, 4, and 8 for GPT-2, Llama2-7B, and Qwen1.5-MoE-A2.7B, respectively.

Our experiments used two training frameworks: Megatron-LM and Megatron-DeepSpeed. Megatron-LM is employed for configurations with pipeline parallelism (PP); for dense models, we set tensor parallelism (TP) to 4 and PP to 2, while MoE models used $TP = 4$, $PP = 2$, and expert parallelism $EP = 4$ to hold the model. In contrast, Megatron-DeepSpeed is used to evaluate techniques of ZeRO and ZeRO Offload. Due to architectural incompatibility of the Qwen1.5-MoE-A2.7B model with Megatron-DeepSpeed for ZeRO-like evaluations, we instead use Megatron-LM's distributed optimizer and optimizer offload features as a substitute. Finally, since evaluating ZeRO and ZeRO Offload requires $DP > 1$, the settings on GPT-2 and Qwen1.5-MoE-A2.7B model exceed



(a) GPT-2. **(b)** Llama2-7B. **(c)** Qwen1.5-MoE-A2.7B.

Figure 7. Comparison of memory efficiency on the 3 models among different allocators, using different combinations of optimizations, namely recomputation (R), Virtual Pipeline (V), ZeRO (Z), and offload (O).



(a) AMD GPU with recomputation. **(b)** H200 GPU with recomputation. **(c)** H200 GPU with virtual pipeline.

Figure 8. Comparison of memory efficiency on different cluster scales and model sizes using optimization of recomputation or virtual pipeline. The red "O" means the case occurs OOM error, the red triangle means distinct throughput decrease.

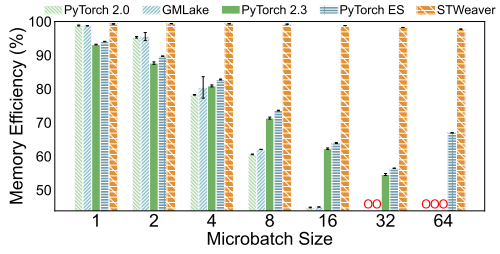


Figure 9. Memory efficiency under different micro-batch sizes when training Llama2-7B with recomputation.

the eight A800 GPU memory capacity, we tested ZeRO with recomputation as an alternative for these cases.

Figure 7 shows the memory efficiency comparison. We can observe that for dense models that do not have dynamic layers, STWeaver can achieve >95% (up to 100%) memory efficiency (i.e., fragmentation ratio < 5%) in all cases, demonstrating the effectiveness of our spatio-temporal planning mechanism. In comparison, PyTorch 2.3 produces 57.2% to 93.8% memory efficiency, GMLake produces 53.6% to 96.5% memory efficiency, and PyTorch ES yields 62.4% to 94.9% memory efficiency. Compared to the baselines, STWeaver reduces fragmentation memory by 88.1%, 77.1%, and 76.0% on average, up to 100%, reducing reserved memory up to 11GB (i.e., 13.8% of GPU memory). The most significant fragmentation reduction appears in the case of GPT-2 with ZeRO-3 and recomputation, which is because the weight size of GPT-2 is relatively small compared to the other 2 models, and thus the proportion of activation tensors (whose lifecycle is affected by recomputation) among all the tensors is considerably larger than that of Llama2-7B.

For the MoE model with dynamic layers, STWeaver still shows 93.8% to 97.8% memory efficiency in the evaluated cases, reducing the fragmentation ratio to 4.3% on average. Compared to PyTorch, GMLake, and PyTorch ES, whose fragmentation ratios are 17.7%, 20.3%, and 6.9%, respectively. STWeaver occurs less fragmentation memory of 77.1%, 78.3%, and 39.4%, respectively. In the MoE test, tuning the default GMLake defragmentation threshold (`fragLimit`) from 512 MB to 64 MB increased memory efficiency to 97.73% but reduced training performance by 56.4% over 50 iterations. The 64 MB threshold caused unstable virtual memory pools under MoE's dynamic allocations, leading to frequent virtual memory operations (up to 1500 times per iteration, each taking around 30ms). A 512 MB threshold optimally balances memory efficiency and training performance.

Training Scales. We demonstrate the scalability of STWeaver on the two different GPU platforms. On the AMD platform, we train the Llama2-7B and Qwen1.5-MoE-A2.7B models on 4 nodes (32 GPUs) and 8 nodes (64 GPUs), respectively. We excluded GMLake and PyTorch ES from this study, as GMLake does not support AMD GPUs, and the features of PyTorch ES are unavailable in our platform's PyTorch version (2.0). All the training experiments are conducted with recomputation. As shown in Figure 8a, STWeaver scales well for both the dense and MoE models. The memory efficiency on both models achieves over 90%, and up to 99.7%. In contrast, the PyTorch caching allocator exhibits memory efficiency below

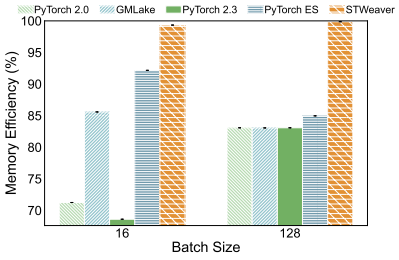


Figure 10. Memory efficiency comparison on Colossal-AI.

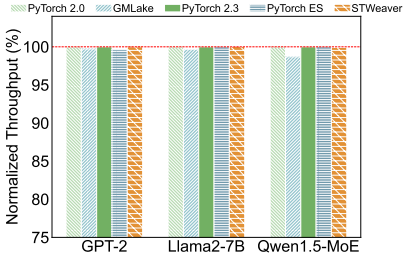


Figure 11. Training throughput comparison using different allocators.

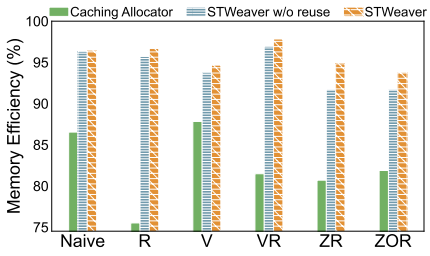


Figure 12. Performance breakdown under MoE model.

60% across all scales of the Llama2-7B model. This result shows that STWeaver can reduce fragmentation memory of 22.8 GB, which is 35.6% of GPU memory. Moreover, for MoE models, when the cluster size increases from 32 to 64 GPUs, the memory efficiency drops below 80%.

To further investigate the scalability of STWeaver's memory efficiency as model and cluster sizes are concurrently augmented, we use four models of varying sizes from the Qwen2.5 series, including 7B, 14B, 32B, and 72B, on 8 to 128 NVIDIA H200 GPUs. The training configurations are either recomputation, as a memory optimization technique, or virtual pipeline, as a parallelism optimization strategy, thereby demonstrating STWeaver's scalability across diverse scenarios. GMLake is not included since it does not support PyTorch 2.6 on the current platform.

Under recomputation settings (Figure 8b), STWeaver achieves 99.1% memory efficiency, reducing fragmentation by over 98.5% and 98.4% compared to PyTorch 2.6 and PyTorch ES, respectively, saving 37.9 GB GPU memory on average, up to 56.3 GB. PyTorch ES showed throughput degradation: 15.0% lower than PyTorch for the 32B model on 32 GPUs, while STWeaver's throughput matched PyTorch within 0.02%. For the 72B model on 64 GPUs, PyTorch faced OOM errors due to fragmentation, and PyTorch ES was 20.1% slower than STWeaver. PyTorch ES's overhead stems from frequent virtual memory API calls, whereas STWeaver maintains high efficiency with minimal runtime penalties.

Under virtual pipeline settings as shown in Figure 8c, STWeaver achieves memory efficiency over 99% in all cases; reduce fragmentation memory over 97.6% and 97.4% compared to PyTorch 2.6 and PyTorch ES, respectively, saving GPU memory of 15.7 GB on average. We find that with the scaling of model and cluster sizes, the memory efficiency of PyTorch and PyTorch ES declined by 10.9% and 15.0%, respectively, while STWeaver differs within 0.7%.

When training the 14B model on 16 GPUs, only STWeaver successfully completes the training without out-of-memory (OOM) error by reducing fragmentation. To avoid OOM, PyTorch and PyTorch ES require disabling virtual pipeline, increasing the tensor parallelism degree, or introducing recomputation. The original training configuration outperforms these adjustments in training throughput by 5.4% to 32.5%, as shown in Table 1. *This indicates that by reducing fragmentation, STWeaver enables more efficient training configurations and yields performance improvements.*

Table 1. Train Qwen2.5-14B with 16 GPUs using different configurations. The original uses only VPP with $TP = 2$.

Config	PyTorch	PyTorch ES	STWeaver	Throughput (TFLOPS)
Original	OOM	OOM	✓	464.3
Disable VPP	OOM	✓	✓	440.6
Recomputation	✓	✓	✓	350.4
$TP = 4$	✓	✓	✓	431.5

Micro-Batch Sizes. Given that activation memory usage during training is directly proportional to microbatch size, and that larger microbatch sizes typically enhance operator computational efficiency [31], we conducted further experiments across a range of microbatch sizes. We conduct the experiments for micro-batch sizes 1, 2, 4, 8, 16, 32, and 64, training Llama2-7B with recomputation on Megatron-LM. As shown in 9, STWeaver yields the best and similar (around 99%) memory efficiency regardless of the micro-batch size, while the other allocators generally performs worse as the micro-batch size increases, mostly because the increasing size of the activation tensors affected by recomputation. This proves STWeaver's robustness against memory-related training configurations in practice.

Training Frameworks. To evaluate STWeaver's generalizability across high-level training frameworks, we also apply STWeaver to Colossal-AI [22], another representative training framework shipped with a variety of memory optimizations. We train GPT-2 on Colossal-AI with tensor offload and ZeRO-3 [36] with two different batch sizes. As depicted in Figure 10, STWeaver still performs better than the other allocators, demonstrating STWeaver's general applicability across training frameworks.

8.3 Overhead Analysis

We next evaluate STWeaver's potential impact on the end-to-end training throughput, as well as the efficiency of the allocation profiler and plan synthesizer facing different numbers of allocation requests.

Overhead of Allocators in Training Throughput. Figure 11 shows the normalized end-to-end training throughput when training the 3 test models on Megatron-LM using different allocators. Specifically, GMLake is normalized against

Table 2. Profile and plan synthesis time in different training configuration. *Num* is the number of requests within one iteration. -N and -R represent the configuration without/with recomputation, respectively.

Config	Num	$T_{profile}(s)$	$T_{plan}(s)$
GPT-2-N	12785	78.82	24.36
GPT-2-R	16569	100.19	21.93
Llama2-7B-N	66529	204.73	104.96
Llama2-7B-R	86721	278.41	136.34
Qwen1.5-MoE-N	196759	273.74	374.18
Qwen1.5-MoE-R	281669	362.20	145.40

PyTorch 2.0, while PyTorch ES and STWeaver are normalized against PyTorch 2.3 for fairness. All the experiment settings adopt recomputation. We can see that none of the allocators incur noticeable throughput degradation. In particular, STWeaver's throughput difference with the vanilla PyTorch 2.3 is <0.05% in all cases, which are most likely due to hardware performance fluctuation. It is worth noting that virtual memory–based GPU memory allocation methods have shown significant drops in training throughput under specific scenarios as discussed in §8.2. GMLake exhibits such behavior in MoE models, and PyTorch ES demonstrates it in recomputation-heavy settings. While these approaches help reduce memory fragmentation, the runtime overhead introduced by virtual memory operations can become non-negligible, ultimately impacting training performance.

The above throughput comparison uses identical training configurations. Thanks to STWeaver's ability to reduce GPU memory usage without incurring extra runtime overhead, it enables the use of more memory-intensive configurations without triggering out-of-memory (OOM) errors. As a result, STWeaver can achieve higher training throughput.

Profiling and Plan Synthesis Time. To understand the efficiency of ahead-of-time planning, we further delve into the profile and plan synthesis time for different settings with varying complexity in terms of the number of total allocation requests that need to be planned per training iteration. As shown in Table 2, the Allocation Profiler utilizing CUDA malloc/free, requires a runtime for minutes for three iterations, approximately 10% to 30% of the speed using PyTorch caching allocator. Given that profiling requires only three iterations, this overhead is deemed acceptable. The plan synthesis time is around 2 minutes, up to 6 minutes for complex cases, and only around 20 seconds for simpler cases. In the case of MoE models, the plan synthesis time in the configuration without recomputation markedly exceeds that of the configuration with recomputation. This disparity occurs because recomputation leads to the immediate deallocation of activation tensors within the same dynamic layer following their forward pass allocation. Conversely, in the absence of recomputation, these activation tensors must be preserved from their forward pass allocation until the corresponding dynamic layer in the backward pass to be freed. Consequently, during the plan generation phase, the configuration without recomputation results in a larger number of *HomoLayer Groups* when classifying dynamic requests. This, in turn, increases the quantity of associated *Dynamic Reuse Space* that needs to be interrogated, thereby prolonging the plan synthesis time.

Table 3. Composition of allocation types.

Allocation type	None	R	V	VR	ZR	ZOR
Total (GB)	59.51	32.36	62.78	33.07	44.65	44.70
Static (GB)	44.68	31.39	46.10	31.83	44.62	44.40
Dynamic fallback w/o reuse (GB)	15.19	1.61	17.80	1.78	2.97	1.95
Dynamic fallback with reuse (GB)	15.19	1.12	17.22	1.70	1.65	1.55

8.4 Performance Breakdown

To understand the performance contribution of the static and dynamic allocators in STWeaver, we evaluate the performance breakdown of STWeaver when training the Qwen1.5-MoE-A2.7B model with the same setting in §8.2. To this end, we sequentially disable the dynamic allocator reusing *Static Allocation Plan* (mentioned as STWeaver w/o reuse), and the static allocator (mentioned as Caching Allocator, which is the vanilla PyTorch caching Allocator), and measure the corresponding memory efficiency in the above cases.

Static Allocator. The results in Figure 12 indicate that STWeaver with only the Static Allocation Plan reduces fragmentation memory by 70.2% compared to PyTorch Caching Allocator. This reduction in fragmentation memory accounts for 91% of the total fragmentation memory reduction achieved by the complete STWeaver system relative to PyTorch. Static planning accounts for the predominant share of the defragmentation result, primarily because static memory allocations form a substantial majority (from 73.4% to 99.3%) of the total memory allocation, as shown in Table 3.

Dynamic Allocator. Compared with STWeaver without dynamic reuse, the full STWeaver reduces memory fragmentation by an additional 22.9%, mainly by lowering fallback allocations to the caching allocator. As shown in Table 3, enabling dynamic reuse decreases the number of requests falling back to the caching allocator. This benefit is most evident under recomputation, where caching allocations drop by 24.9%. Without recomputation, the impact is smaller.

The difference stems from how recomputation affects memory lifespans. Without recomputation, activation memory is allocated during the forward pass and held until the backward pass, causing dynamic and static allocation requests' lifespans to fully overlap. This results in a peak memory usage close to the sum of both. With recomputation, activation memory is released immediately after the forward pass, so static and dynamic requests do not overlap in time. As a result, dynamic requests can reuse idle regions in the static pool, reducing overall peak usage.

9 Related Work

Online GPU Allocators. To reduce memory fragmentation and improve allocation efficiency, a plethora of online GPU memory allocators [2, 43, 49, 50] have been developed. Dynamic allocators operate atop the native GPU memory APIs (e.g., `cudaMalloc`) in a similar manner as the caching allocator of PyTorch. Differently, such allocators are meant to run on GPU threads alongside GPGPU applications, rather than managing GPU memory from the host like PyTorch and GMLake. To reduce fragmentation, they usually adopt sophisticated allocation policies such as the slab and buddy systems. Also, to achieve high-throughput allocations on GPUs, they have proposed scalable synchronization primitives across the massive threads of GPUs [10]. As a pluggable allocator of PyTorch, STWeaver also chooses to manage GPU memory from the host to improve usability and programmability.

Generic Memory Defragmentation Techniques. Memory defragmentation has been studied and discussed in various scenarios [3, 16, 19] beyond GPU applications. Previous work [13, 26, 41, 48] has proposed defragmentation strategy based on data movement or copying. These approaches are mainly deployed in the real-time system with unpredictable runtime behaviors, which result in complex defragmentation strategies with high runtime overhead.

Machine Learning Compilers. Machine Learning compilers that convert high-level computation graphs to GPU instructions must manage memory allocation for these graphs. The compilers pre-analyze control dependencies to optimize memory layouts before execution. Current compilers like TVM [5] and TFLite [7] use greedy heuristics for reasonable allocation, while Checkmate [14] employs solvers for optimal rematerialization to improve results. Unlike deep ML compilers that organize memory allocation and deallocation at the computation graph level, STWeaver manages memory requests at the level of the overall model execution. These approaches are complementary and orthogonal with our work.

10 Conclusion

This work presents the design, implementation, and evaluation of STWeaver, a novel memory allocation system that significantly improves the memory utilization of large-scale model training. STWeaver builds on our insight of spatio-temporal regularity in model training allocation requests to combine ahead-of-time memory layout planning with runtime profile-guided allocation. Extensive evaluations show that STWeaver significantly outperforms state-of-the-art solutions in terms of both effectiveness and efficiency. Complementary to existing runtime defragmentation methods, we believe STWeaver demonstrates the powerful potential of fusing proactive pre-runtime planning with reactive runtime decision-making.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Andrew V Adinetz and Dirk Pleiter. 2014. Halloc: a high-throughput dynamic memory allocator for GPGPU architectures. In *Proceedings of GTC*, Vol. 152.

[3] Martin Aigner, Christoph M Kirsch, Michael Lippautz, and Ana Sokolova. 2015. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. *ACM SIGPLAN Notices* 50, 10 (2015), 451–469.

[4] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *Proceedings of OSDI*. 578–594.

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[7] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. 2021. Tensorflow lite micro: Embedded machine learning for tinyml systems. In *Proceedings of MLSys*. 800–811.

[8] Jiangfei Duan, Shuo Zhang, Zerui Wang, Lijuan Jiang, Wenwen Qu, Qinghao Hu, Guoteng Wang, Qizhen Weng, Hang Yan, Xingcheng Zhang, et al. 2024. Efficient training of large language models on distributed infrastructures: a survey. *arXiv preprint arXiv:2407.20018* (2024).

[9] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[10] Isaac Gelado and Michael Garland. 2019. Throughput-oriented GPU memory allocation. In *Proceedings of PPoPP*. 27–37.

[11] Ant Group. 2023. GMLake: GPU Memory Lake for Large Model Training. https://github.com/antgroup/glake.

[12] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, et al. 2024. GMLake: Efficient and Transparent GPU Memory Defragmentation for Large-scale DNN Training with Virtual Memory Stitching. In *Proceedings of ASPLOS*. 450–466.

[13] Richard L Hudson and J Eliot B Moss. 2001. Sapphire: Copying GC without stopping the world. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*. 48–57.

[14] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of MLSys* 2 (2020), 497–511.

[15] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[16] Mark S Johnstone and Paul R Wilson. 1998. The memory fragmentation problem: Solved? *ACM Sigplan Notices* 34, 3 (1998), 26–36.

[17] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[18] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer

models. In *Proceedings of MLSys*. 341–353.

[19] Rune Krauss, Mehran Goli, and Rolf Drechsler. 2023. EDDY: A multi-core BDD package with dynamic memory management and reduced fragmentation. In *Proceedings of ASPDAC*. 423–428.

[20] Tung D Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. 2018. Tflms: Large model support in tensorflow by graph rewriting. *arXiv preprint arXiv:1807.02037* (2018).

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

[22] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. 2023. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of ICPP*. 766–775.

[23] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434* (2024).

[24] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[25] Martin Maas, Ulysse Beaugnon, Arun Chauhan, and Berkin Ilbeyi. 2022. Telamalloc: Efficient on-chip memory allocation for production machine learning accelerators. In *Proceedings of ASPLOS*. 123–137.

[26] Simon Marlow, Tim Harris, Roshan P James, and Simon Peyton Jones. 2008. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of ISMM*. 11–20.

[27] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. 2017. Training deeper models by GPU memory optimization on TensorFlow. In *Proceedings of ML Systems Workshop in NIPS*, Vol. 7.

[28] Michael D Moffitt. 2023. MiniMalloc: A lightweight memory allocator for hardware-accelerated machine learning. In *Proceedings of ASPLOS*. 238–252.

[29] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of SOSP*. 1–15.

[30] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel dnn training. In *Proceedings of ICML*. PMLR, 7937–7947.

[31] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of SC*. 1–15.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[33] PyTorch. 2025. PyTorch Documentation. https://pytorch.org/docs/stable/index.html.

[34] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. 2023. Zero bubble pipeline parallelism. *arXiv preprint arXiv:2401.10241* (2023).

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

[36] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of SC*. IEEE, 1–16.

[37] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of SC*. 1–14.

[38] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *Proceedings of USENIX ATC 21*. 551–564.

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

[40] John E Shore. 1975. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Commun. ACM* 18, 8 (1975), 433–440.

[41] David Siegwart and Martin Hirzel. 2006. Improving locality with parallel hierarchical copying GC. In *Proceedings of ISMM*. 52–63.

[42] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. 2022. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990* (2022).

[43] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. 2012. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *Proceedings of InPar*. IEEE, 1–10.

[44] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[45] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295* (2024).

[46] Qwen Team. 2024. Qwen1.5-MoE: Matching 7B Model Performance with 1/3 Activated Parameters. https://qwenlm.github.io/blog/qwen-moe/.

[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[48] Ronald Veldema and Michael Philippsen. 2012. Parallel memory defragmentation on a GPU. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. 38–47.

[49] Marek Vinkler, Vlastimil Havran, et al. 2014. Register efficient memory allocator for GPUs.. In *Proceedings of HPG*. 19–27.

[50] Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele. 2013. Fast dynamic memory allocator for massively parallel architectures. In *Proceedings of the 6th workshop on general purpose processor using graphics processing units*. 120–126.

[51] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. 1995. Dynamic storage allocation: A survey and critical review. In *Memory Management: International Workshop IWMM 95 Kinross, UK, September 27–29, 1995 Proceedings*. Springer, 1–116.

[52] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115* (2024).

[53] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. 2022. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414* (2022).