PROTRAIN: EFFICIENT LLM TRAINING VIA AUTOMATIC MEMORY MANAGEMENT

Hanmei Yang [1] **Jin Zhou** [1] **Yao Fu** [2] **Xiaoqun Wang** [2] **Ramine Roane** [2] **Hui Guan** [1] **Tongping Liu** [1]

ABSTRACT

Memory pressure has emerged as a dominant constraint in scaling the training of large language models (LLMs), particularly in resource-constrained environments. While modern frameworks incorporate various memory-saving techniques, they often expose low-level configuration knobs that require manual tuning and specialized system expertise. This not only adds engineering overhead but also risks suboptimal hardware utilization when misconfigured. This paper introduces ProTrain, a novel training system that automatically tailors memory management policies to the model architecture and underlying hardware resources, eliminating the need for manual intervention. The core of ProTrain is its automated memory management that abstracts complex memory management strategies into a few tunable configuration parameters, allowing searches for optimal parameter settings using cost models. ProTrain is equipped with a runtime profiler that provides precise estimates of latency, memory usage, and I/O bandwidth to build high-fidelity cost models. ProTrain does not change the training algorithm and thus does not compromise accuracy. Experiments show that ProTrain improves training throughput by 1.43× to 2.71× compared to the state-of-the-art training systems.

1 INTRODUCTION

Large Language Models (LLMs) have achieved remarkable success in various fields. Inspired by the scaling law (Kaplan et al., 2020) that LLM perplexity often improves logarithmically with the number of parameters, there has been a trend towards increasing parameter size. However, the significant growth in parameter size leads to a substantial increase in memory demands. According to existing studies (Ren et al., 2021), each unit increase in parameters generally requires 16× more memory to store the model states (e.g., fp16 and fp32 parameters, fp16 gradients, fp32 momentum and variances), not to mention the increased memory demand for activations due to larger model sizes. Consequently, memory has become the dominant bottleneck in LLM training.

Limitations of Existing Work Numerous memory management strategies have been proposed to alleviate the memory pressure, including parallelization (Li et al., 2020; Shoeybi et al., 2019; Huang et al., 2019), gradient checkpointing (Chen et al., 2016; Jain et al., 2020; Herrmann et al., 2019; Zhao et al., 2023a; Korthikanti et al., 2023), and tensor swapping (Rhu et al., 2016; Le et al., 2018; Huang et al., 2020; Ren et al., 2021; Sun et al., 2022). These approaches are often implemented in LLM training frameworks (Rasley et al., 2020; Li et al., 2023; Zhao et al., 2023b), yet each exposes a set of low-level configuration knobs that require extensive manual tuning.

The tuning challenge arises not only from the large number of parameters, but also from the lack of proper abstractions to manage their complex interdependencies. First, *techniques are often mutually exclusive in their applicability*. Consider activation memory management: checkpointing reduces memory usage by recomputing activations, while swapping trades I/O overhead for memory savings by offloading them to CPU. When computation can effectively hide this I/O overhead, swapping outperforms checkpointing, yet most practical systems favor checkpointing for its simplicity. The optimal choice thus depends on runtime dynamics rather than static heuristics. Second, *even complementary techniques compete for shared hardware resources*. For example, activation swapping and parameter prefetching both consume CPU-GPU bandwidth. When swapping is enabled to reduce activation memory, it can saturate the bandwidth, blocking parameter prefetches and stalling computation. Balancing these techniques requires coordinating their usage based on available hardware resources, yet such hidden dependencies make optimal configurations difficult to identify manually. Together, these interactions create a search space that is both large and non-decomposable, making manual tuning infeasible without deep system expertise.

This challenge is particularly pronounced in systems such

[1]University of Massachusetts Amherst [2]Advanced Micro Devices, Inc.. Correspondence to: Hanmei Yang <hanmeiyang@umass.edu>.

Proceedings of the 9th MLSys Conference, Bellevue, WA, USA, 2026. Copyright 2026 by the author(s).

as DeepSpeed, which exposes over 18 tunable parameters spanning ZeRO partitioning, tensor swapping, and gradient checkpointing that are tightly coupled. For instance, `stage3_max_reuse_distance` caches recently used parameters on the GPU to avoid reloading them during the backward pass, while `stage3_max_live_parameters` limits the total number of parameters allowed to reside in GPU memory. This creates a conflict: aggressive caching consumes the available parameter budget, leaving insufficient room for prefetching upcoming parameters and delaying computation. Balancing these configurations requires runtime-aware tuning based on hardware resources, and the tight coupling among these knobs makes independent tuning ineffective, requiring joint exploration of an exponentially large search space. Without such tuning, performance is suboptimal. Our evaluation shows that training a 10B GPT-2 model on RTX 3090 GPUs with default configurations utilizes only 35.6% of GPU memory and runs $1.18\times$ slower than an optimized setup. Moreover, configurations optimized for RTX 3090 GPUs fail to fully utilize A100 GPUs, while those tailored for A100s often cause out-of-memory (OOM) errors on RTX 3090s, requiring users to manually re-tune for every hardware change.

Our Contribution To address these challenges, we present ProTrain, an efficient LLM training system that automatically tailors memory management policies for data-parallel training to the target model and hardware configuration. The core idea behind ProTrain is to unify disparate memory-saving techniques, such as ZeRO-style model state partitioning, tensor swapping, and gradient checkpointing, into a structured configuration space. Instead of exposing users to fragmented and low-level tuning knobs, ProTrain abstracts these strategies into a compact set of high-level parameters that capture the trade-offs and constraints across techniques. These parameters can then be automatically optimized using runtime and memory cost models informed by profiling. ProTrain makes three core technical contributions:

- *Structured Memory Strategies:* We design two coordinated strategies that provide a unified view of memory management techniques. First, *hierarchical chunk management* handles model states at both inter- and intra-chunk levels to promote memory reuse and overlap computation with communication. Second, *interleaved block management* manages activations by interleaving tensor swapping and gradient checkpointing at the transformer block level, improving scheduling flexibility and minimizing memory fragmentation.
- *Memory-Aware Profiler:* We introduce a memory-aware profiler that captures memory usage patterns often missed by conventional tools, such as transient allocations and memory from unhookable operations. The profiler reconstructs peak memory usage under varied scheduling policies and supplies detailed statistics on latency, memory footprint, and I/O bandwidth—providing critical inputs to our cost models.
- *Automatic Memory Management:* We formulate memory management as a constrained optimization problem: minimize per-iteration runtime while staying within GPU memory capacity. Leveraging the structured search space and profiling data, ProTrain builds cost models that capture how different configurations affect runtime and memory usage. While memory optimization behaviors, such as asynchronous tensor swapping, are challenging to model due to runtime variability, we redesign the memory strategies to expose deterministic scheduling structures. This makes their interactions more predictable and enables the cost models to reason about compound effects, such as bandwidth contention and overlapping behaviors, ultimately guiding automatic and efficient configuration searches.

We evaluate ProTrain on popular LLM architectures including GPT-2, OPT, Mistral, and LLaMA, and compare against state-of-the-art frameworks such as DeepSpeed, Colossal-AI, and FSDP. ProTrain trains models up to $2.47\times$ larger than DeepSpeed and $1.48\times$ larger than Colossal-AI on RTX 3090 GPUs, and up to $7.5\times$ larger than FSDP on A100 GPUs, while delivering $1.43\times$ to $2.71\times$ higher training throughput across model and hardware configurations. In addition to throughput, ProTrain demonstrates strong scalability with available compute resources and consistently outperforms existing frameworks across a range of batch sizes. Its runtime and memory estimators achieve prediction errors under 4%, enabling reliable automatic strategy selection. These results highlight ProTrain's effectiveness in delivering efficient and scalable memory management for LLM training under diverse deployment scenarios.

2 Background and Related Work

Model training consists of three key stages: forward propagation (FWD), backward propagation (BWD), and parameter updates (OPTIM). LLM training typically employs mixed-precision training (Micikevicius et al., 2017), where FWD and BWD are performed in reduced-precision formats (e.g., fp16 or bf16), while parameter updates use higher precision (e.g., fp32) to ensure accuracy. Memory consumption during training arises from two main sources: *model states* and *residual states*. Model states include parameters, gradients, and optimizer states, while residual states consist of activations and temporary tensors.

Among the many memory-saving techniques developed for LLM training, this work focuses on three widely adopted approaches. (a) *Zero Redundancy Optimizer (ZeRO)*, a representative of sharded data parallelism, partitions model

states across GPUs to eliminate redundancy and utilize aggregate device memory. We adopt this strategy as the parallelism foundation of our system due to its simplicity and widespread use. (b) *Gradient checkpointing* reduces activation memory usage by discarding selected tensors during the forward pass and recomputing them during the backward pass. (c) *Tensor swapping*[1] offloads tensors, either model states or activations, to external memory such as CPU DRAM or NVMe devices. These techniques target different tensor types and offer complementary trade-offs between memory efficiency and computation overhead. A more detailed discussion of related work is provided in Appendix D.

3 Overview of ProTrain

ProTrain comprises three core components, as illustrated in Figure 1: (1) **Structured Memory Strategies** (Section 3.1), which manage model states and activations using strategies suited to their specific optimization techniques. (2) a **Memory-Aware Profiler** (Section 3.2) that collects runtime and memory data to guide memory management decisions, and (3) an **Automatic Memory Management** module (Section 3.3) that dynamically identifies the optimal memory optimization strategies for training the target LLM on the given hardware.

3.1 Structured Memory Strategies

ProTrain applies different strategies to manage model states and activations based on their unique optimization needs. For model states, ZeRO and offloading are often used to alleviate GPU memory pressure. Activations, on the other hand, can be handled via gradient checkpointing and offloading to reduce memory footprint. These strategies unify the configurations of ZeRO, offloading, and checkpointing through a compact and structured design that enables predictable scheduling and effective automated tuning. In the following, we elaborate the specific approaches for managing them.

3.1.1 Hierarchical Chunk Management for Model States

ProTrain proposes a **hierarchical chunk management** system to manage model states that include parameters, gradients, and optimizer states. The system borrows the idea of organizing model states into chunks from prior work (Fang et al., 2022; Li et al., 2023; Zhao et al., 2023b), enabling grouped transfers to improve I/O throughput. However, existing chunk-based systems face the following limitations:

- **Ineffective Overlapping:** The transfer of model states may not effectively overlap with computation if all model states have to be transferred. In addition, inefficient use of heterogeneous hardware leaves some overlapping opportunities unexploited.
- **Ping-Pong Access Pattern:** Existing systems organize chunks by model-defined order rather than execution order, resulting in back-and-forth access that increases peak memory usage, causes cache inefficiencies, and triggers unnecessary transfers due to evictions.
- **Frequent Memory Allocations:** Existing systems typically allocate memory dynamically to hold chunks. This approach fails to exploit the repetitive patterns of the training process, resulting in unnecessary allocation overhead and increased memory fragmentation.

ProTrain overcomes these limitations through a two-level hierarchical approach: managing model states at both inter-chunk and intra-chunk levels. At the inter-chunk level, *ProTrain addresses the ineffective overlapping issue by dividing model states into persistent and non-persistent chunks. Persistent chunks* reside on the GPU, eliminating data transfers and enabling direct GPU parameter updates. *Non-persistent chunks* are offloaded to CPU memory or other ranks, requiring parameters to be gathered on the GPU for computation and gradients to be reduced and offloaded back to the CPU for updates. This design unifies ZeRO sharding and offloading: non-persistent chunks are partitioned to reduce communication overhead, while persistent chunks remain unpartitioned for fast GPU access. By adjusting the number of persistent chunks through its Automatic Memory Management module, ProTrain achieves a simple yet effective trade-off between memory usage and communication cost, without requiring complex coordination.

To further improve overlapping efficiency, ProTrain overlaps CPU parameter updates for non-persistent chunks with GPU backward computations, making effective use of idle CPU cycles. Once the number of persistent chunks is determined, *ProTrain allocates them sequentially from the beginning of the model*, while the remaining chunks are marked as non-persistent. This strategy is motivated by two observations: (1) The forward pass computation can often hide the overhead of model states prefetching for later layers, but not the first few layers. (2) The first few layers, being processed last during the backward pass, do not have enough time to fully hide the latency of CPU parameter updates.

At the intra-chunk level, *ProTrain addresses the ping-pong access issue by organizing model states within each chunk according to their execution order.* By leveraging the repeatable access patterns in training, ProTrain registers runtime hooks for forward and backward computations to capture operator execution order, eliminating back-and-forth transfers and releasing memory earlier to prefetch future chunks.

To reduce memory allocations, ProTrain introduces pre-allocated chunk buffers for non-persistent chunks, allowing upcoming chunks to be prefetched ahead of computation.

[1]As we consider swapping to CPU memory, we use *swapping* and *CPU offloading* interchangeably.

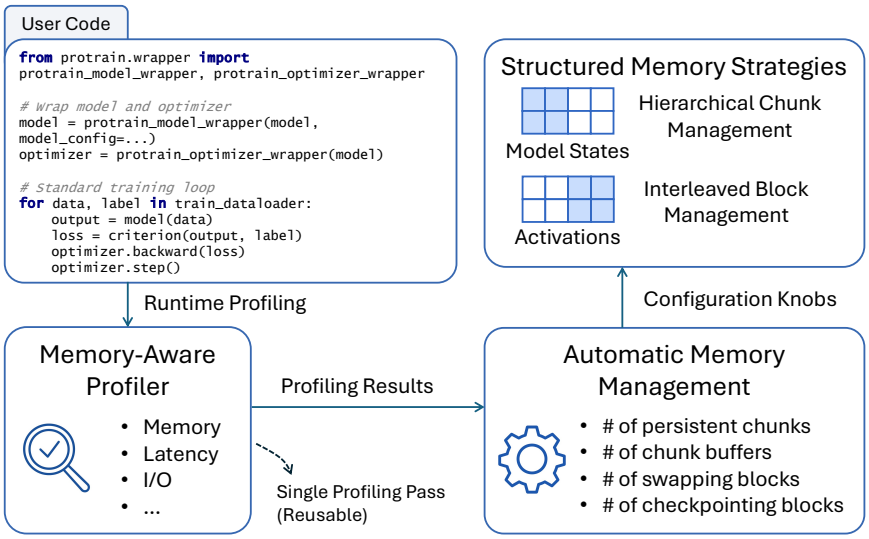

Figure 1. System overview of ProTrain.

As parameters within each chunk are arranged in execution order, evictions triggered by buffer shortages follow a predefined sequence, eliminating the need for complex dynamic eviction policies. With pre-allocated buffers and deterministic prefetching, ProTrain exhibits stable runtime behavior, enabling accurate modeling of both execution overhead and peak memory usage (Section 3.3). Moreover, non-persistent chunks that remain in the buffer after the forward pass can be reused during the backward pass if not evicted, eliminating the need for reloading and enhancing performance.

3.1.2 *Interleaved Block Management for Activations*

Activation memory usage can be greatly reduced using techniques such as tensor swapping and gradient checkpointing. While these approaches are widely employed in model training, their application to LLMs presents distinct challenges:

- **Granularity Selection:** Existing approaches often rely on either coarse-grained techniques, such as uniformly applying gradient checkpointing to all transformer blocks (Li et al., 2023; Rasley et al., 2020), which can underutilize GPU memory and unnecessarily slow down training, or fine-grained tensor-level management (Peng et al., 2020; Beaumont et al., 2021), which adds implementation complexity and scalability challenges due to the large search space created by the numerous activation tensors in LLMs.
- **Efficiency in Scheduling:** Even with appropriate granularity, inefficient scheduling between activation swapping and gradient checkpointing can result in higher memory requirements and degraded performance.

To balance flexibility and scalability, *ProTrain proposes a block-wise activation management strategy that applies memory optimization techniques at the transformer block level.* Unlike coarse-grained strategies that enforce a uniform policy across all blocks, this approach allows each block to adopt the most suitable optimization scheme. Each block can choose among three activation handling strategies: activation swapping, gradient checkpointing, or no optimization (i.e., neither swapping nor checkpointing). Compared to tensor-level management, the proposed block-wise approach adapts to existing transformer implementations without requiring computation graph reconstruction. It also significantly reduces the search space and scheduling overhead of managing individual activation tensors, offering a more scalable solution for LLMs.

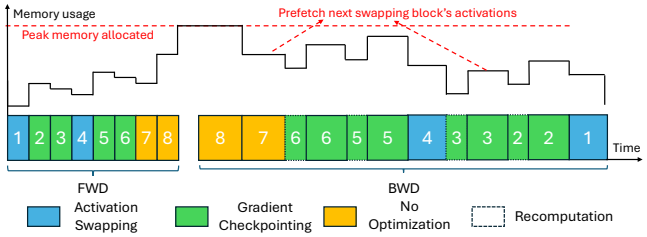

Figure 2. Interleaved Block Management Layout and Memory Usage Trend

ProTrain introduces an interleaved block management layout to coordinate activation swapping and gradient checkpointing, as illustrated in Figure 2. The figure shows an 8-block transformer where blocks 1 and 4 use activation swapping, blocks 2, 3, 5, and 6 use gradient checkpointing, and the remaining blocks are left unoptimized. This layout offers several benefits: (1) Placing swapping blocks earlier increases opportunities to overlap swapping with

computation. (2) Interleaving swapping and checkpointing blocks prevents activation accumulation, reducing OOM risks caused by slower swapping and minimizing peak memory usage, as shown in the upper part of Figure 2. (3) Placing unoptimized blocks in later layers ensures their activations are consumed earlier, enabling timely prefetching for swapping blocks.

3.2 Memory-Aware Profiler

ProTrain introduces a memory-aware profiler to determine the optimal settings for the tunable parameters defined in Section 3.1. The profiler focuses on estimating peak memory usage and understanding the runtime behavior of the model. In addition, it separately collects hardware metrics, such as memory transfer bandwidth and collective communication operation durations. The runtime profiler utilizes a single GPU to perform sampling under the user-defined batch size, which has the following challenges:

- **Underestimation of Memory Usage:** Traditional profiling techniques, including static profiling (Patil et al., 2022) and layer-wise runtime profiling (Beaumont et al., 2021), fail to capture memory usage from two sources: (1) transient memory for temporary tensors, which are intermediate results created within an operator that increase the peak memory footprint, and (2) memory usage from unhookable operators, typically PyTorch operators invoked via the functional API calls (e.g., `nn.functional.softmax` or `nn.functional.layer_norm`) instead of `nn.Module` instances and are thus bypassed by layer-wise profiling hooks. These overlooked components contribute about 17.2% (3.06 GB) of peak memory in the 10B GPT-2 model with batch size 16. This underestimation leads to inaccurate memory planning and increases the risk of OOM errors during model execution.
- **Accurate Profiling under Memory Constraints:** Profiling LLMs is inherently constrained by the limited memory capacity of a single GPU, often necessitating the use of memory-saving techniques to prevent OOM errors. However, these techniques change memory allocation patterns, making it difficult to accurately predict the peak memory usage of the model.

To accurately estimate peak memory usage, ProTrain proposes to profile operators within a complete model execution trace, rather than profiling them individually in isolation. This trace-based approach captures two types of memory deltas to handle both transient temporary tensors and unhookable operators. The **intra-operator memory delta** measures the peak memory usage[2] during an operator's execution minus the allocated memory just before it starts (i.e., the extra memory spike caused by intermediate results created within an operator). The **inter-operator memory delta** measures the peak memory usage between consecutive hooked modules minus the allocated memory at the end of the first hooked module, and is used to infer the memory cost of unhookable operators that fall between them. Together, these two measurements allow ProTrain to accurately estimate the full peak memory footprint of the model.

However, running a complete trace for large models is often infeasible on a single GPU. For example, a 70B model in FP16 requires 140 GB for model weights alone, exceeding the 80 GB capacity of an A100 GPU. To address this, ProTrain employs on-demand tensor management: tensors are allocated just before use and released immediately after, reducing peak memory to that of the largest single operator rather than the entire model. This represents the most aggressive memory-saving strategy; in real training, however, model states and activations persist in GPU memory across operators, resulting in a much higher memory baseline. This raises the challenge of reconstructing the actual peak memory under any target configuration from a profiled trace that does not reflect real memory persistence patterns.

To enable peak memory estimation under altered usage patterns, we present a memory-aware profiling approach. During profiling, ProTrain temporarily drops model states (i.e., parameters and gradients) and activations to reduce memory usage, as their fixed and predictable sizes allow their contributions to peak memory to be reconstructed through static analysis. Specifically, model states are pre-allocated in ProTrain and persist throughout training, contributing a constant memory cost that can be directly included in the estimation. In contrast, activations, managed by the training framework and tightly coupled to the operator execution order, require operator-by-operator analysis to quantify their impact on peak memory, as detailed in Section A.2. By combining static analysis with operator-level memory tracking, our profiler provides accurate peak memory estimates across diverse configurations.

3.3 Automatic Memory Management

ProTrain introduces an Automatic Memory Management module that dynamically optimizes memory configurations to adapt to different models and hardware setups, eliminating the need for tedious manual tuning while ensuring optimal efficiency. Automating memory management presents two key challenges:

- **Modeling runtime and memory usage**: Identifying optimal configurations requires precise modeling of how memory optimization strategies affect runtime and memory usage, which can be complex. For instance, while offloading reduces GPU memory demands, it

[2] ProTrain leverages PyTorch's CUDA caching allocator statistics and uses pre/post operator hooks to track memory changes.

introduces communication delays and intricate overlapping patterns that complicate runtime estimation. Predicting peak memory usage becomes difficult as swapping and checkpointing dynamically modify memory allocation patterns during execution.

- **Complications from Interactions Between Optimization Strategies**: Memory optimization strategies do not operate independently; rather, they influence and constrain each other. For example, parameter prefetching and activation swapping compete for bandwidth, potentially causing delays and inefficiencies when used together. Therefore, their interactions must be carefully evaluated rather than considered in isolation.

ProTrain addresses the modeling challenge by exploiting the structured search space defined by Structured Memory Strategies (Section 3.1). The search space is abstracted into a compact set of tunable parameters, making their impact on runtime and memory usage more tractable to model. Specifically, we abstract memory strategies into the following tunable parameters. For model states, we define (1) **chunk size** (S_{chunk}) – detailed in Section B.1, (2) **number of chunks** (N_{chunk}), and (3) **number of persistent chunks** (n_{persist}) **and pre-allocated chunk buffers** (n_{buffer}) – both bounded by N_{chunk}. For activations, we configure (1) **swapping interval** (N_{interval}) – determined by the computation time required to swap out a block, (2) **number of blocks** (N_{block}), and (3) **number of blocks designated for activation swapping** (n_{swap}) **and gradient checkpointing** ($n_{\text{checkpoint}}$) – both bounded by N_{block}. The parameters S_{chunk}, N_{chunk}, N_{interval}, and N_{block} are determined independently before the optimal configuration search. During configuration search, ProTrain tunes n_{persist}, n_{buffer}, n_{swap}, and $n_{\text{checkpoint}}$ to identify the optimal setup.

ProTrain further formulates memory management as a constrained search problem that considers the interplay between different strategies. The objective is to minimize the per-iteration runtime, while ensuring that peak memory usage remains within the available GPU memory capacity. To guide the search, ProTrain builds two cost models—one for runtime and one for peak memory usage—both defined as functions of the configuration parameters $n_{\text{persist}}, n_{\text{buffer}}, n_{\text{swap}}, n_{\text{checkpoint}}$. Unlike approaches that require running actual training iterations for each candidate configuration, ProTrain builds these cost models from a single profiling pass and uses them to analytically evaluate all configurations in the search space, making the search tractable even when the configuration space is large. Detailed formulations are in Appendix A.1 and Appendix A.2.

Building on the formulated cost models, ProTrain explores the configuration space through an exhaustive search while applying pruning strategies to improve efficiency. First, n_{swap} is constrained by the swapping interval N_{interval} and available bandwidth, which limits the number of feasible values to a small set. Second, configurations are evaluated in order of increasing memory usage, and those exceeding the GPU memory capacity are discarded early to avoid unnecessary evaluations. After traversing the search space, ProTrain selects the configuration parameters $\{n_{\text{persist}}, n_{\text{buffer}}, n_{\text{swap}}, n_{\text{checkpoint}}\}$ that achieves the shortest iteration runtime while satisfying memory constraints as the final setup.

4 IMPLEMENTATION

We implemented ProTrain on top of PyTorch with about 7,600 lines of code. It is designed for easy adoption and requires minimal changes to existing training code. To use ProTrain, users only need to wrap the model and optimizer with the provided interfaces and the training loop remains unchanged. An example usage is shown in the top-left corner of Figure 1. Internally, ProTrain uses runtime hooks to trace operator execution and tensor usage, enabling memory management without modifying the original model code. This design allows ProTrain to support any model that follows PyTorch's standard forward API. The detailed implementation components, including chunk organization and two low-level memory optimizations, are described in Appendix B.

5 EVALUATION

5.1 Experiment Setup

Workloads. We evaluate ProTrain on four popular LLM architectures: GPT-2 (Radford et al., 2019), OPT (Zhang et al., 2022), Mistral (Jiang et al., 2023), and LLaMA (Touvron et al., 2023), using model implementations from the HuggingFace library. To cover a range of model sizes, we vary the hidden size, number of transformer blocks, and attention heads, as summarized in Table 1. The sequence length is fixed at 1024. [3]

Table 1. Model configuration

Model	Parameter Size	Hidden Size	# of Blocks	# of Heads
Mistral	7B	4096	32	32
GPT-2	10B	4096	48	32
OPT, LLaMA	13B	5120	40	40
GPT-2	15B, 20B, 30B, 40B	8192	18, 24, 36, 50	64
OPT	30B	7168	48	56
LLaMA	34B	8192	48	64

[3]We use 1024 for controlled comparisons, a standard setting when this study began. The methodology remains effective for longer sequences (e.g., the 75B model is trainable up to 7168 on an A100), as our search automatically adapts to the increased activation memory.

Table 2. Maximum trainable model size (unit: billion)

Backend	RTX 3090*1	RTX 3090*4	A100*1	A100*4
ProTrain	**34B**	**37B**	**75B**	**87B**
DeepSpeed	15B	15B	34B	37B
Colossal-AI	25B	25B	53B	53B
FSDP	3B	15B	10B	55B

Testbed. Experiments are conducted in two environments: (1) **4× RTX 3090**: The system contains four NVIDIA GeForce RTX 3090 GPUs with 24GB memory. It is powered by Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz with 24 cores. The CPU DRAM size is 384GB. The PCIe version is 3 with 15.8GB/s bandwidth. NVLink is not available in this setup. (2) **4× A100**: The system contains four NVIDIA A100 GPUs with 80GB memory. It is powered by Intel(R) Xeon(R) Platinum 8480+ with 112 cores. The CPU DRAM size is 1TB. The PCIe version is 4 with 31.5GB/s bandwidth. GPUs are fully connected by NVLink 3.0 with 300GB/s bandwidth.

Baselines. We compare ProTrain with three widely used open-source LLM training frameworks. We focus on comparable features—ZeRO-3, CPU offloading, and gradient checkpointing. While implementations differ, these mechanisms provide similar functionality for fair comparison: (1) **DeepSpeed** (Rasley et al., 2020) (v0.12.1): ZeRO-3 is enabled with offloading of both parameters and optimizer states. Thresholds such as `stage3_max_live_parameter` and `stage3_prefetch_bucket_size` are tuned for best performance. (2) **Colossal-AI** (Li et al., 2023) (0.3.3): We use the Gemini Plugin with chunk-based ZeRO-3 memory management. Both parameters and optimizer states are offloaded using a static placement strategy. (3) **FSDP** (Zhao et al., 2023b) (PyTorch 2.0.1): The native PyTorch implementation of ZeRO-3 with CPU offloading and selective gradient checkpointing. The `transformer_auto_wrap_policy` is used to encapsulate each transformer block into a `FlatParameter`. Gradient checkpointing is either applied to all transformer blocks or entirely disabled, depending on whether memory constraints require it. For FSDP, we additionally evaluate selective checkpointing in Section C.2.1.

5.2 Training Performance

5.2.1 Maximum Trainable Model Size

Table 2 reports the maximum trainable model sizes across frameworks using GPT-2 as a benchmark. ProTrain supports models up to 34B parameters on a single RTX 3090 GPU and scales to 37B with four GPUs. On A100 GPUs, ProTrain trains models as large as 75B on a single GPU and 87B with four GPUs, outperforming Colossal-AI and DeepSpeed by 1.64× and 2.35×, respectively, in the four-GPU setup. In contrast, FSDP underperforms in the single GPU setting, handling significantly smaller models. Some frameworks also struggle to scale with multiple GPUs due to inefficiencies in model initialization. These results demonstrate ProTrain's ability to unlock larger model sizes on constrained hardware, making LLM training more accessible.

5.2.2 Training Throughput

Figure 3 presents the maximum training throughput for various models on four RTX 3090 and A100 GPUs, measured in tokens per second. Throughput is obtained by testing each model at different batch sizes to find the highest achievable value. The results show that ProTrain consistently outperforms other frameworks across diverse hardware and models. On RTX 3090 GPUs, ProTrain achieves an average throughput of 2090 tokens per second, 1.77 to 2.71× higher than other frameworks. On A100 GPUs, ProTrain improves the throughput of DeepSpeed, Colossal-AI, and FSDP by 1.85×, 1.43×, and 2.22×, respectively. While training throughput typically decreases as model sizes grow and memory demands increase, ProTrain maintains robust performance relative to other frameworks. For example, ProTrain achieves 5.05× the training speed of 15B GPT-2 on RTX 3090 GPUs and 2.78× that of 34B LLaMA on A100 GPUs compared to FSDP. In contrast, other frameworks often fail to train larger models with feasible batch sizes or resort to inefficient memory optimization techniques. Overall, ProTrain delivers substantial performance improvements, achieving up to 2.71× the throughput of other frameworks on average and significantly enhancing the efficiency of LLM training.

5.2.3 Performance Scalability

Figure 4(a) shows the maximum throughput of 10B GPT-2 with varying numbers of RTX 3090 GPUs. ProTrain scales efficiently, achieving 2493 tokens/s with four GPUs, a 3.5× improvement over the single-GPU baseline. While DeepSpeed and Colossal-AI also show increased throughput with more GPUs, their performance gains do not match those of ProTrain. Figure 7(a) further presents the scalability results for 34B LLaMA on four A100 GPUs. ProTrain continues to outperform the baselines, delivering a 2.49× to 3.58× speedup over a single GPU. This enhanced performance on A100s, compared to RTX 3090s, is largely due to ProTrain's advanced memory management, which fully exploits the A100's larger memory and higher bandwidth. As a result, ProTrain is able to scale to larger batch sizes and better utilize the available hardware resources to boost training throughput.

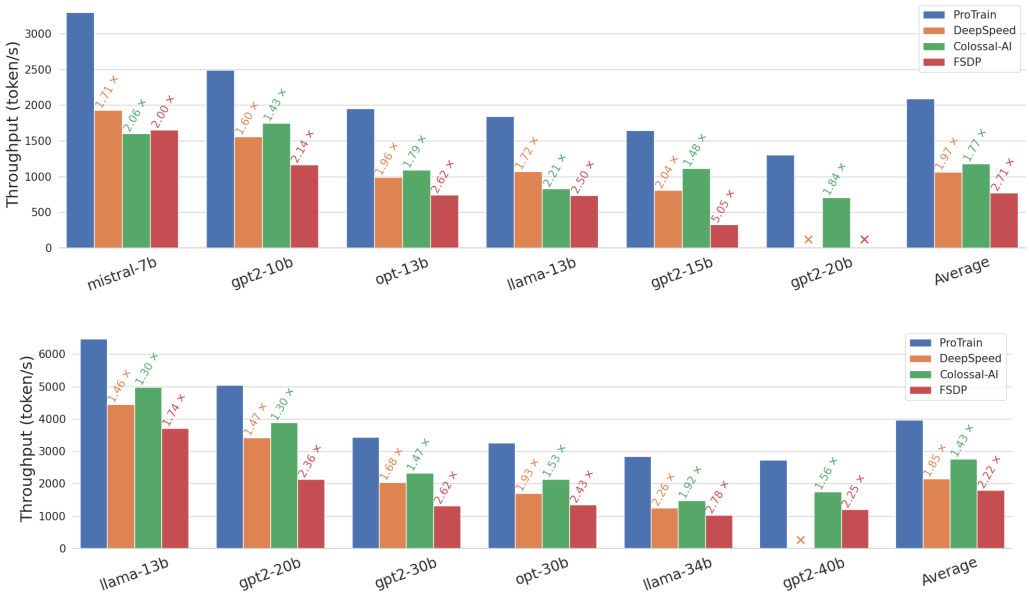

Figure 3. Maximum Training Throughput on four RTX 3090 GPUs (top) and A100 GPUs (bottom). The "×" symbol indicates training failure due to Out-Of-Memory errors.

5.2.4 *Performance Breakdown*

Figure 4(b) provides a detailed breakdown of one iteration time into forward, backward, and parameter update phases for training 10B GPT-2 across varying batch sizes on four RTX 3090 GPUs. Detailed experimental results on A100 GPUs are provided in Appendix C.1.

On RTX 3090 GPUs, ProTrain achieves notable speedups for two main reasons. First, it efficiently implements memory strategies through hierarchical chunk management and interleaved block management to optimize data movement and enhance I/O-compute overlap, with the benefits particularly evident during the backward pass due to its higher communication demands. Meanwhile, CPU parameter updates, executed concurrently with GPU backward computations, are effectively overlapped, making their contribution to the total iteration time nearly negligible. Second, ProTrain's automatic memory management dynamically adjusts optimization strategies to balance performance and memory efficiency. As batch sizes grow, increasing memory demands prompt ProTrain to intensify memory-saving techniques, resulting in performance gains primarily driven by ProTrain's efficient implementation.

5.3 Ablation Study

5.3.1 *Impact of Optimization Strategies*

Figure 5 analyzes the performance impact of disabling key optimizations in ProTrain when training a 10B GPT-2 model on four RTX 3090 GPUs. Each bar reports the runtime ratio of ProTrain without the corresponding optimization to that with the optimization enabled. Disabling hierarchical chunk management, where persistent chunks are replaced by three chunk buffers, leads to a slowdown of $1.02\times$ to $1.19\times$. The importance of overlapped parameter updates grows with batch size, as larger batches require more parameters to be offloaded to the CPU. When overlapping is enabled, these CPU updates are effectively hidden behind GPU computations, improving overall efficiency; disabling this overlap, however, prevents such hiding and results in an average slowdown of $1.22\times$.

Removing interleaved block management and instead applying gradient checkpointing to all transformer blocks results in an average slowdown of $1.04\times$. The limited gain in this setting is expected. On RTX 3090 GPUs with PCIe 3.0 bandwidth of 15.8 GB/s, training is communication-bound, and the available bandwidth is already saturated by parameter prefetching, leaving little room for activation swapping to overlap with computation. As a result, the automatic search favors checkpointing over swapping, producing a configuration already close to the ablation baseline of applying checkpointing to all blocks. On hardware with higher CPU-GPU bandwidth, such as the GH200 with NVLink-C2C (450 GB/s, $\sim$14.3$\times$ higher than A100 with PCIe 4.0), the relative cost of swapping decreases substantially. For example, for a 40B GPT-2 model, swapping activations per block takes 1.9s on A100 but an estimated 0.13s on GH200, while recomputation takes 0.95s on A100 and an estimated

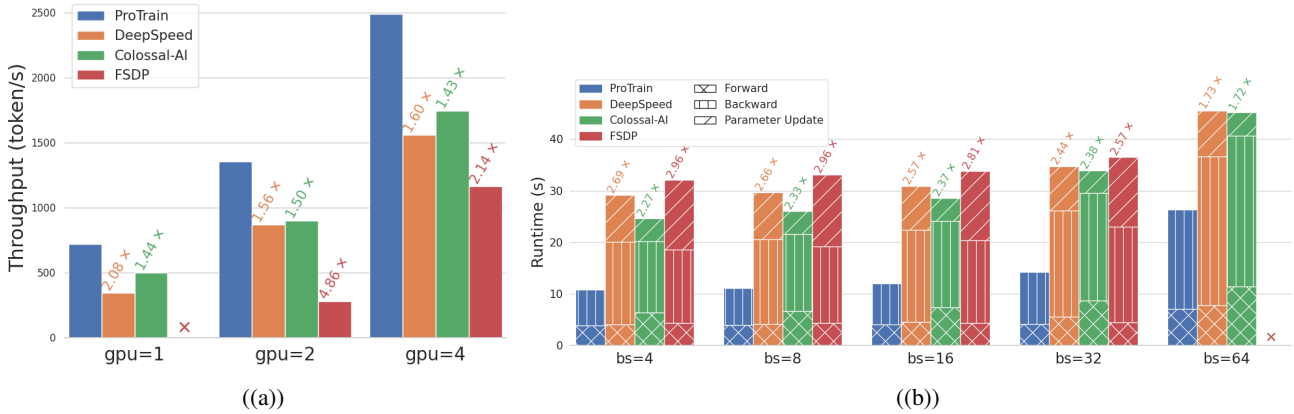

Figure 4. Scalability of performance on RTX 3090 GPUs (a) Maximum throughput across different numbers of GPUs (b) Step time breakdown for different batch sizes

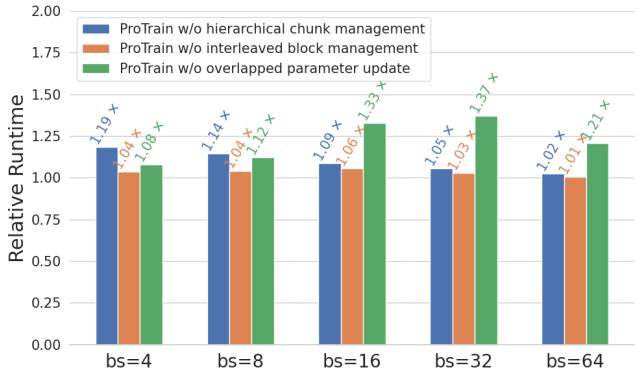

Figure 5. Impact of optimization strategies

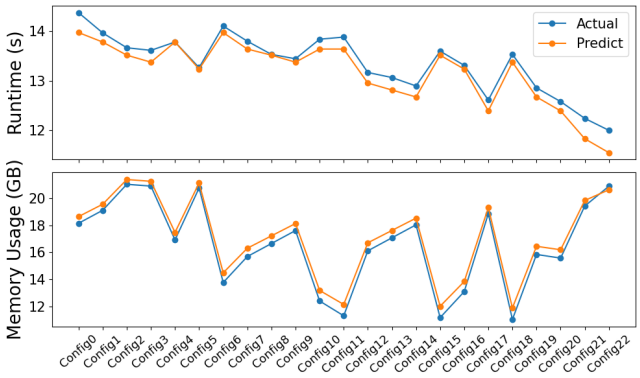

Figure 6. Predicted vs. actual runtime and peak memory across different configurations

0.3s on GH200, scaled by their FP16 Tensor Core TFLOPS ratio. This shift would lead ProTrain to favor swapping over checkpointing on GH200, yielding potentially larger gains from interleaved block management.

These results characterize the contribution of each optimization for a 10B GPT-2 model on RTX 3090 GPUs, and illustrate that their effectiveness varies across models and hardware. Section 5.4 further analyzes the optimal strategies under varying conditions, underscoring the necessity of ProTrain's automatic approach.

5.3.2 *Effectiveness of the Runtime and Peak Memory Usage Estimators*

Figure 6 (top) compares the predicted and actual runtimes across a range of memory configurations when training a 10B GPT-2 model. The estimator not only closely tracks the overall runtime trend, but also achieves high accuracy, with deviations within 4%. This fidelity enables ProTrain to reliably identify the configuration with the lowest runtime, demonstrating its robustness in navigating diverse memory strategies.

Figure 6 (bottom) shows the predicted versus actual peak memory usage across a wide range of configurations for training a 10B GPT-2 model. The estimator closely tracks the memory usage trend, with a maximum error within 4%. Appendix C.2 further validates the estimator's reliability across different models and batch sizes.

5.3.3 *Throughput Comparison with and without Offloading*

ProTrain integrates CPU offloading as one of the key techniques to improve scalability, which may raise concerns about its practicality in real-world deployments due to limited bandwidth. To address this, we compare ProTrain against existing frameworks that do not rely on offloading. As shown in Table 3, frameworks such as DeepSpeed and Colossal-AI exhibit competitive performance without offloading on smaller models (e.g., 7B Mistral, 10B GPT-2). However, their advantages diminish on larger models,

Table 3. Training throughput on four A100 GPUs with and without offloading (tokens/s)

Model		Mistral-7B	GPT2-10B	LLaMA-13B	GPT2-20B
ProTrain	automatic	11060.92	8266.40	6471.32	5043.75
DeepSpeed	w/	7708.30 (1.43×)	6447.70 (1.28×)	4446.43 (1.46×)	3420.90 (1.47×)
	w/o	9748.03 (1.13×)	7320.50 (1.13×)	5234.92 (1.24×)	OOM
Colossal-AI	w/	7279.76 (1.52×)	6848.47 (1.21×)	4980.91 (1.30×)	3892.95 (1.30×)
	w/o	8447.30 (1.31×)	7855.46 (1.05×)	4404.30 (1.47×)	2084.74 (2.42×)
FSDP	w/	5315.81 (2.08×)	4666.03 (1.77×)	3715.12 (1.74×)	2136.16 (2.36×)
	w/o	OOM	OOM	OOM	OOM

where limited GPU memory restricts the achievable batch size. For instance, on 13B LLaMA, Colossal-AI can support a batch size of up to 220 with offloading, but only 32 without it, resulting in a 13% slowdown when offloading is disabled. In comparison, ProTrain supports a batch size of 228 through more efficient memory strategy implementation and better overlapping mechanisms, achieving a $1.3\times$ higher throughput than the best-performing Colossal-AI configuration. These results demonstrate that ProTrain consistently outperforms all baselines, both with and without offloading. More broadly, they highlight that offloading, when properly integrated, can serve as a scalable and cost-effective alternative to parallelism, enabling larger batch sizes and higher throughput under fixed hardware budgets, without requiring additional compute nodes.

5.3.4 Profiling and Search Overhead

ProTrain performs efficient profiling with overhead that scales linearly with a single training iteration. For instance, profiling 7B Mistral with batch size 4 on RTX 3090 GPUs takes 3.09 seconds, while 20B GPT-2 requires 5.38 seconds. The optimal configuration search is even faster, averaging just 0.06 seconds on the same hardware. These results demonstrate the low profiling and search overhead of ProTrain, enabling fast identification of optimal configurations with minimal cost.

5.4 Analysis of Searched Configurations

Table 4 summarizes the optimal memory management configurations automatically selected by ProTrain for different models, batch sizes, and hardware setups. The following discussion examines how these factors influence configuration choices.

Batch Size Impact. Increasing the batch size from 8 (row A) to 64 (row B) on RTX 3090 GPUs leads to the following configuration adjustments: n_{swap} increases from 0 to 2, $n_{\text{checkpoint}}$ increases from 0 to 24, n_{persist} decreases from 12 to 2, and n_{buffer} increases from 0 to 3. These changes align with runtime execution patterns, where a larger batch size amplifies computational intensity, allowing more data loading to be fully hidden within the computation. For 1B GPT-2 with batch size 64, ProTrain requires only three chunk buffers, each dedicated to prefetching, execution, and offloading. Since parameter loading overhead is fully hidden within the computation, the training remains compute-bound, eliminating the need for additional chunk buffers. The remaining bandwidth is then allocated to activation swapping, and ProTrain selects two swapping blocks whose overhead is entirely overlapped with computation without disrupting parameter loading. If bandwidth is insufficient for activation swapping in other setups, ProTrain prioritizes $n_{\text{checkpoint}}$ to minimize memory usage while maintaining performance.

Hardware Impact. When training 1B GPT-2 with batch size 8 (row A), both A100 and RTX 3090 GPUs have sufficient memory, eliminating the need for offloading or activation checkpointing, resulting in identical configurations. However, with batch size 64 (rows B and C), the increased memory demand necessitates offloading and checkpointing on RTX 3090, while A100 can handle the workload without these optimizations due to larger memory capacity. For 10B GPT-2 with batch size 8 (rows D and E), both GPUs face memory constraints but adopt different strategies based on runtime characteristics. RTX 3090 GPUs, lacking NVLink and experiencing communication bottlenecks in NCCL operations, apply checkpointing to all blocks to free up space for larger n_{persist} and n_{buffer}, mitigating parameter gathering overhead that cannot be fully hidden by computation. In contrast, A100 GPUs leverage NVLink's high bandwidth, making training compute-bound, which results in retaining all activations and using three chunk buffers as the optimal configuration.

Model Size Impact. The table shows that different model sizes require different configuration parameters. Larger models generally require more offloading (smaller n_{persist} and n_{buffer}, and larger n_{swap}) and more gradient checkpointing (larger $n_{\text{checkpoint}}$). These adjustments optimize memory usage, allowing efficient training and fine-tuning of larger models within hardware constraints.

Table 4. Automatically searched configurations with the best performance

ID	Model	Batch Size	Hardware	N_{block}	$n_{\text{checkpoint}}$	n_{swap}	N_{chunk}	n_{persist}	n_{buffer}
A	GPT2-1B	8	4 RTX 3090 or A100 GPUs	32	0	0	12	12	0
B	GPT2-1B	64	4 RTX 3090 GPUs	32	24	2	12	2	3
C	GPT2-1B	64	4 A100 GPUs	32	0	0	12	12	0
D	GPT2-10B	8	4 RTX 3090 GPUs	48	48	0	49	3	46
E	GPT2-10B	8	4 A100 GPUs	48	0	0	49	15	3

5.5 Multi-Node Scalability

To assess the scalability of ProTrain in large-scale training, we conduct a multi-node experiment on a 175B GPT model with batch size 256, deployed on 16 A100 GPUs across 4 nodes connected via a 100Gb InfiniBand network. The intra-node setup mirrors the configuration described in Section 5.1. ProTrain achieves a throughput of 2218.70 tokens/s, outperforming DeepSpeed by $2.58\times$ and Colossal-AI by $2.09\times$, while FSDP fails due to OOM errors. These results demonstrate ProTrain's ability to generalize across nodes and scale effectively to larger clusters.

6 Discussion

While offloading can alleviate memory pressure, its practicality in real-world LLM training is often questioned due to potential data transfer overhead. As shown in Section 5.3.3, we compare ProTrain against frameworks both with and without offloading. Unlike prior systems that rely on fixed or manually tuned thresholds, ProTrain automatically evaluates the cost-benefit trade-off and applies offloading only when it improves performance. This adaptive decision-making allows ProTrain to maintain high throughput and consistently outperform other frameworks across diverse hardware settings.

ProTrain focuses on data-parallel training with ZeRO, which is complementary to parallelism-focused systems (Zheng et al., 2022; Miao et al., 2023; Lin et al., 2024) that optimize parallelism strategies but largely overlook fine-grained memory management. A closely related system is Mist (Zhu et al., 2025), which co-optimizes parallelism (DP, TP, PP) jointly with memory optimizations including checkpointing, ZeRO, and offloading. Mist automates a broader scope than ProTrain by jointly searching parallelism strategies and memory optimizations. It relieves users from choosing the parallelism configuration, a step that ProTrain still requires. However, this broader search space comes with trade-offs. ProTrain takes a narrower but deeper approach. By fixing the parallelism strategy, it can afford more precise memory modeling at the operator level, capturing intra-operator temporary tensors and unhookable operators that symbolic or coarser-grained models overlook. This also yields a more tractable search space, enabling fine-grained memory optimization that would be prohibitively expensive in a joint parallelism-memory search. In addition, ProTrain's memory management layer operates on local tensors within each rank and is orthogonal to parallelism decisions, meaning it can in principle be composed with parallelism planners like Mist to combine broad automation with deep memory optimization.

Beyond performance gains and the scope discussed above, this work offers a new perspective on scaling LLM training. While recent efforts have largely focused on scaling through additional hardware or parallelism, we highlight that better utilizing existing CPU memory, an abundant yet often underused resource, can substantially increase model capacity and batch size without additional investment in large-memory GPUs. Given the significant cost gap between commodity CPU memory and high-end GPUs, ProTrain provides a practical path to maximizing training potential within a fixed infrastructure budget.

7 Conclusion

This paper introduced ProTrain, a novel training system that simplifies the training process through automatic memory management. ProTrain designs tailored memory strategies for model states and activations and abstracts them into configuration parameters. It highlights the significance of precise memory usage and runtime data gathered through memory-aware profiling to build high-fidelity cost models that drive automated configuration search. Through these innovations, ProTrain achieves up to $5\times$ the performance of state-of-the-art frameworks and enables training models with up to 75 billion parameters on a single A100 GPU. We hope this work benefits AI researchers and practitioners by enabling more efficient use of existing hardware, supporting the scaling of LLM training without proportional increases in infrastructure cost.

REFERENCES

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pp. 929–947, 2024.

Beaumont, O., Eyraud-Dubois, L., and Shilova, A. Efficient combination of rematerialization and offloading for training dnns. Advances in Neural Information Processing Systems, 34:23844–23857, 2021.

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

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. Advances in neural information processing systems, 35:16344–16359, 2022.

Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. Advances in neural information processing systems, 35:30318–30332, 2022.

Fang, J., Zhu, Z., Li, S., Su, H., Yu, Y., Zhou, J., and You, Y. Parallel training of pre-trained models via chunk-based dynamic memory management. IEEE Transactions on Parallel and Distributed Systems, 34(1):304–315, 2022.

Feng, Y., Xie, M., Tian, Z., Wang, S., Lu, Y., and Shu, J. Mobius: Fine tuning large-scale models on commodity gpu servers. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pp. 489–501, 2023.

Fishman, M., Chmiel, B., Banner, R., and Soudry, D. Scaling fp8 training to trillion-token llms. arXiv preprint arXiv:2409.12517, 2024.

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

Huang, C.-C., Jin, G., and Li, J. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 1341–1355, 2020.

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

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

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. Mistral 7b. arXiv preprint arXiv:2310.06825, 2023.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. arXiv preprint arXiv:2001.08361, 2020.

Korthikanti, V. A., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models. Proceedings of Machine Learning and Systems, 5, 2023.

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

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: Experiences on accelerating data parallel training. arXiv preprint arXiv:2006.15704, 2020.

Li, S., Liu, H., Bian, Z., Fang, J., Huang, H., Liu, Y., Wang, B., and You, Y. Colossal-ai: A unified deep learning system for large-scale parallel training. In Proceedings of the 52nd International Conference on Parallel Processing, pp. 766–775, 2023.

Li, Y., Phanishayee, A., Murray, D., Tarnawski, J., and Kim, N. S. Harmony: Overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers. arXiv preprint arXiv:2202.01306, 2022.

Lin, Z., Miao, Y., Zhang, Q., Yang, F., Zhu, Y., Li, C., Maleki, S., Cao, X., Shang, N., Yang, Y., et al. nnscaler: Constraint-guided parallelization plan generation for deep learning training. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pp. 347–363, 2024.

Miao, X., Wang, Y., Jiang, Y., Shi, C., Nie, X., Zhang, H., and Cui, B. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. Proc. VLDB Endow., 16(3):470–479, 2023.

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

Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R., Ha, S., Heinecke, A., Judd, P., Kamalu, J., et al. Fp8 formats for deep learning. arXiv preprint arXiv:2209.05433, 2022.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for dnn training. In Proceedings of the 27th ACM symposium on operating systems principles, pp. 1–15, 2019.

Nie, X., Miao, X., Yang, Z., and Cui, B. Tsplit: Fine-grained gpu memory management for efficient dnn training via tensor splitting. In 2022 IEEE 38th International Conference on Data Engineering (ICDE), pp. 2615–2628. IEEE, 2022.

NVIDIA. Api documentation of apex optimizers. https://nvidia.github.io/apex/optimizers.html, 2018.

Patil, S. G., Jain, P., Dutta, P., Stoica, I., and Gonzalez, J. Poet: Training neural networks on tiny devices with integrated rematerialization and paging. In International Conference on Machine Learning, pp. 17573–17583. PMLR, 2022.

Peng, X., Shi, X., Dai, H., Jin, H., Ma, W., Xiong, Q., Yang, F., and Qian, X. Capuchin: Tensor-based gpu memory management for deep learning. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, pp. 891–905, 2020.

Rabe, M. N. and Staats, C. Self-attention does not need $o(n2)$ memory. arXiv preprint arXiv:2112.05682, 2021.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. OpenAI blog, 1(8):9, 2019.

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

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–14, 2021.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. 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, pp. 3505–3506, 2020.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. Zero-offload: Democratizing billion-scale model training. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pp. 551–564, 2021.

Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1–13. IEEE, 2016.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053, 2019.

Sun, X., Wang, W., Qiu, S., Yang, R., Huang, S., Xu, J., and Wang, Z. Stronghold: fast and affordable billion-scale deep learning model training. In SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–17. IEEE, 2022.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971, 2023.

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

Yang, H., Deng, S., Nagpal, A., Naumov, M., Janani, M., Liu, T., and Guan, H. An empirical study of microscaling formats for low-precision llm training. In Proceedings of the 32nd IEEE International Symposium on Computer Arithmetic (ARITH), 2025.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. Opt: Open pre-trained transformer language models. arXiv preprint arXiv:2205.01068, 2022.

Zhao, J., Gao, X., Xia, R., Zhang, Z., Chen, D., Chen, L., Zhang, R., Geng, Z., Cheng, B., and Jin, X. Apollo: Automatic partition-based operator fusion through layer by layer optimization. Proceedings of Machine Learning and Systems, 4:1–19, 2022.

Zhao, X., Le Hellard, T., Eyraud-Dubois, L., Gusak, J., and Beaumont, O. Rockmate: an efficient, fast, automatic and generic tool for re-materialization in pytorch. In International Conference on Machine Learning, pp. 42018–42045. PMLR, 2023a.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. arXiv preprint arXiv:2304.11277, 2023b.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., et al. Alpa: Automating inter- and intra- operator parallelism for distributed deep learning. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pp. 559–578, 2022.

Zhu, Z., Giannoula, C., Andoorveedu, M., Su, Q., Mangalam, K., Zheng, B., and Pekhimenko, G. Mist: Efficient distributed training of large language models via memory-parallelism co-optimization. In Proceedings of the Twentieth European Conference on Computer Systems, pp. 1298–1316, 2025.