A COST MODELS

A.1 Modeling Runtime

We formulate automatic memory management as a constrained optimization problem and aim to minimize the total runtime of the training process. Since training consists of repeated iterations, minimizing the total training time is equivalent to minimizing the runtime of a single iteration, denoted as $T_{\text{Iteration}}$:

$$\min_{\textit{configs}} T_{\text{Iteration}} \quad s.t.\ M_{\text{Peak}} < M_{\text{Capacity}}, \tag{1}$$

where M_{Peak} represents the peak memory usage, and M_{Capacity} is the total GPU memory capacity. The set of tunable configuration parameters, *configs*, that determines the memory optimization strategy is $\{n_{\text{persist}}, n_{\text{buffer}}, n_{\text{swap}}, n_{\text{checkpoint}}\}$.

A single training iteration consists of three phases: the forward pass, the backward pass, and parameter updates. In ProTrain, forward and backward passes run on the GPU, while parameter updates can be performed on either the GPU or CPU. CPU updates execute concurrently with GPU computations, which include the backward pass and GPU-based updates. However, if CPU updates do not fully overlap with GPU operations, the total iteration time is constrained by the longer CPU update phase. The runtime cost model is formulated as follows:

$$T_{\text{Iteration}} = T_{\text{FWD}} + \max\{T_{\text{BWD}} + T_{\text{GPU\_OPTIM}}, T_{\text{CPU\_OPTIM}}\}, \tag{2}$$

where T_{FWD} and T_{BWD} are modeled as a function of the configuration parameters. For parameter update of the persistent chunks ($T_{\text{GPU\_OPTIM}}$) and non-persistent chunks ($T_{\text{CPU\_OPTIM}}$), ProTrain models runtimes predictably based on parameter size.

To estimate the runtime of the forward pass, ProTrain adopts a chunk-based approach, as most operations operate at the chunk level. By comparing the computation and communication overheads for each chunk, the estimator identifies whether the chunk is compute-bound or communication-bound, using the larger value as its runtime estimate:

$$T_{\text{FWD}} = \sum_{i=1}^{N_{\text{chunk}}+1} \max\left(T_{\text{comp}}^{\text{FWD}}(i-1), T_{\text{comm}}^{\text{FWD-prefetch}}(i)\right), \tag{3}$$

where $T_{\text{comp}}^{\text{FWD}}$ represents the forward computation time of a chunk, which aggregates the runtimes of individual operators within the chunk. $T_{\text{comm}}^{\text{FWD-prefetch}}$ represents the communication time required to prefetch parameters for the next chunk during the forward pass, which is calculated as follows:

$$T_{\text{comm}}^{\text{FWD-prefetch}}(i) = \begin{cases} 0, & \text{if } i \le n_{\text{persist}} \\ & \text{or } i > N_{\text{chunk}}, \\ T_{\text{comm}}^{\text{gather}}(i) + T_{\text{comm}}^{\text{upload}}(i), & \text{otherwise}, \end{cases} \tag{4}$$

where $T_{\text{comm}}^{\text{gather}}$ is the time to gather parameter chunks from multiple GPUs, and $T_{\text{comm}}^{\text{upload}}$ is the time to transfer non-persistent chunks from CPU to GPU. To estimate $T_{\text{comm}}^{\text{gather}}$ and $T_{\text{comm}}^{\text{upload}}$, ProTrain uses a separate hardware profiling to accurately model their runtime. In contrast to conventional approaches that assume a fixed bandwidth for memory transfers, ProTrain simulates various overlapping scenarios to capture the effects of bandwidth contention. For instance, when activation swapping is enabled, we estimate the swapping time, identify the affected chunks, and use the reduced bandwidth instead. The activation swapping time is excluded from the forward pass calculation, as ProTrain carefully controls n_{swap} to ensure its overhead is fully overlapped with computation.

Similarly, the runtime of the backward pass is calculated at the chunk level:

$$\begin{aligned} T_{\text{BWD}} = \sum_{i=1}^{N_{\text{chunk}}+1} \max(&T_{\text{comp}}^{\text{BWD}}(i) + T_{\text{recomp}}(i), \\ &T_{\text{comm}}^{\text{BWD-prefetch}}(i-1), T_{\text{comm}}^{\text{reduce-offload}}(i+1)). \end{aligned} \tag{5}$$

In contrast to the forward pass, the backward computation includes additional recomputation overheads from gradient checkpointing, represented by $T_{\text{recomp}}(i)$. The value is calculated as the aggregated forward computation time for the checkpointed blocks within chunk i, following the block-to-chunk mapping in the interleaved organization. Another key distinction from the forward pass is the overhead related to gradient reduce and offloading during the backward pass, represented by $T_{\text{comm}}^{\text{reduce-offload}}$, which is defined as:

$$T_{\text{comm}}^{\text{reduce-offload}}(i) = \begin{cases} T_{\text{comm}}^{\text{reduce}}(i), & \text{if } i \le n_{\text{persist}}, \\ 0, & \text{if } i > N_{\text{chunk}}, \\ T_{\text{comm}}^{\text{reduce}}(i) + T_{\text{comm}}^{\text{offload}}(i), & \text{otherwise}. \end{cases} \tag{6}$$

As with $T_{\text{comm}}^{\text{FWD-prefetch}}$, the performance of $T_{\text{comm}}^{\text{reduce-offload}}$ is directly influenced by the number of persistent chunks, as persistent chunks avoid parameter prefetching and only involve gradient reduce. However, $T_{\text{comm}}^{\text{BWD-prefetch}}$ differs in its estimation from $T_{\text{comm}}^{\text{FWD-prefetch}}$, and is defined as:

$$T_{\text{comm}}^{\text{BWD-prefetch}}(i) = \begin{cases} 0, & \text{if } i \leq n_{\text{persist}} \\ & \text{or } i > N_{\text{chunk}} - n_{\text{buffer}}, \\ T_{\text{comm}}^{\text{gather}}(i) + T_{\text{comm}}^{\text{upload}}(i), & \text{otherwise.} \end{cases} \tag{7}$$

This difference arises because of the presence of chunk buffers, which cache the parameter loaded and gathered during the forward pass, eliminating the need for re-loading and re-gathering in the backward pass. As a result, uploading and gathering are only required for chunks that were evicted due to limited buffer capacity.

Following the backward pass, parameter updates are executed on either the GPU or CPU, depending on the chunk type. ProTrain employs a fast CPU Adam optimizer (Ren et al., 2021) for CPU updates and the FusedAdam optimizer (NVIDIA, 2018) for GPU updates. The runtime of both is modeled based on parameter size.

A.2 Modeling Memory Consumption

Accurate peak memory estimation is essential for effective memory management, particularly in LLMs, where memory constraints require careful data handling to prevent exceeding capacity. Our estimator relies on the data collected by the profiler (detailed in Section 3.2) to compute memory usage precisely. The profiled data includes the changes in current memory usage, $\Delta M_{\text{Cur}}^{\text{PriorOp}}$, and peak memory usage, $\Delta M_{\text{Peak}}^{\text{PriorOp}}$, before each operation, as well as $\Delta M_{\text{Cur}}^{\text{Op}}$ and $\Delta M_{\text{Peak}}^{\text{Op}}$ during each operation. Additionally, the profiler tracks the activation memory usage for each operator, $M_{\text{Act}}^{\text{Op}}$, and the memory usage at the end of the forward pass, M_{FWD}. Since memory usage typically peaks during the backward pass, our focus is on identifying the peak memory usage in that phase.

To estimate peak memory usage, we define two key variables: the current memory usage, M_{Cur}, and the peak memory usage, M_{Peak}. Initially, M_{Cur} is set as:

$$M_{\text{Cur}} = M_{\text{FWD}} + \sum_{i=1}^{N_{\text{op}}} M_{\text{Act}}^{\text{Op}}(i) - M_{\text{swap}} \cdot n_{\text{swap}} - M_{\text{checkpoint}} \cdot n_{\text{checkpoint}} \tag{8}$$

Here, M_{swap} and $M_{\text{checkpoint}}$ represent the per-block memory savings from activation swapping and gradient checkpointing. Then, for each operator, we iteratively update M_{Cur} and M_{Peak} as follows:

$$M_{\text{Cur}}(i) = M_{\text{Cur}}(i-1) + \Delta M_{\text{Cur}}^{\text{PriorOp}}(i) + \Delta M_{\text{Cur}}^{\text{Op}}(i) - \begin{cases} 0, & \text{if Op}(i) \text{ uses swapping or checkpointing} \\ M_{\text{Act}}^{\text{Op}}(i), & \text{if no optimization is applied} \end{cases} \tag{9}$$

$$M_{\text{Peak}}(i) = \max\big\{M_{\text{Peak}}(i-1), M_{\text{Cur}}(i-1) + \Delta M_{\text{Peak}}^{\text{PriorOp}}(i), M_{\text{Cur}}(i-1) + \Delta M_{\text{Cur}}^{\text{PriorOp}}(i) + \Delta M_{\text{Peak}}^{\text{Op}}(i) + I_{\text{checkpoint}}^{\text{Op}}(i) \cdot M_{\text{checkpoint}}\big\}. \tag{10}$$

where $I_{\text{checkpoint}}^{\text{Op}}(i)$ is an indicator function set to 1 if the operator belongs to a checkpointing block and is the first operator in that block; otherwise, it is 0. This reflects the runtime behavior where gradient checkpointing triggers a recomputation before the actual backward computation, increasing peak memory usage. For swapping blocks, activation prefetching is only performed when sufficient memory is available, ensuring it does not contribute to peak memory, as illustrated in the memory usage trend of Figure 2.

This iterative, operator-wise approach estimates peak memory by accounting for transient temporary tensors and short-lived activations, closely mirroring their allocation and deallocation behavior during runtime. Finally, the overall peak memory considering model states is computed as:

$$M_{\text{Peak}} = (M_{\text{Peak}} + M_{\text{persist}} \cdot n_{\text{persist}} + M_{\text{buffer}} \cdot n_{\text{buffer}}) \cdot \alpha \tag{11}$$

where M_{persist} and M_{buffer} represent the memory allocated for a single persistent chunk and chunk buffer, respectively, while α is a factor that accounts for potential memory inefficiencies due to memory fragmentation.

B Implementation Details

In the remainder of this section, we describe key implementation components, including chunk organization and size search (Section B.1) and two low-level memory optimizations (Section B.2) that improve memory efficiency.

B.1 Chunk Organization and Size Search

As discussed in Section 3.1.1, ProTrain organizes parameters according to their execution order rather than the initialization order used in Colossal-AI. For transformers that share parameters across layers, ProTrain uses the parameter's first occurrence as the ordering criterion. Additionally, ProTrain groups parameters from the same transformer block into one chunk, which minimizes memory accesses, especially when using gradient checkpointing that requires

revisiting parameters in reverse during the backward pass. To identify the most efficient chunk size for a given model, ProTrain conducts a grid search that simulates memory waste across various chunk sizes and selects the one that minimizes it.

B.2 Memory Optimizations

Single-Stream Memory Allocation ProTrain unifies memory allocations within the default stream to improve memory utilization. PyTorch's allocator adopts a multi-heap design where each stream has its own heap, limiting cross-heap memory reuse and necessitating the use of `record_stream()` to ensure correctness. By using a single stream for all allocations and directly managing deallocation synchronization ourselves, we effectively prevent misuse and reallocation conflicts, thereby improving memory efficiency.

Customized Pinned Memory Allocator We observe that the default pinned memory allocator (`CUDAHostAllocator`) often over-allocates by rounding up to the nearest power of two, leading to significant memory waste. To address this inefficiency, ProTrain developed a customized pinned memory allocator that leverages insights from automatic memory management to precisely determine pinned memory requirements, providing finer control and avoiding the excessive memory reservation of the default allocator.

C Full Experiment Results

C.1 Throughput Scalability on A100 GPUs

Figure 7(b) breaks down the per-iteration runtime into forward, backward, and parameter update phases when training 34B LLaMA on four A100 GPUs. On A100 GPUs, the performance advantage of ProTrain becomes more pronounced. It effectively overlaps CPU parameter updates with GPU computations, avoiding the update bottlenecks observed in frameworks such as FSDP, where the use of the default Adam optimizer leads to significant time spent in the update phase. Compared to DeepSpeed, ProTrain achieves lower backward pass latency by using fixed-size chunk buffers that enable timely parameter prefetching and reuse. In contrast, DeepSpeed relies on a threshold-based sliding window mechanism, where parameters are evicted only after their usage completes and new ones are prefetched only when enough memory is freed. This approach leads to poor bandwidth utilization and suboptimal overlapping.

C.2 Effect of Runtime/Peak Memory Usage Estimator

To further validate its effectiveness, Figure 8 (top) compares the predicted and actual runtimes of the configurations selected by ProTrain across different models and batch sizes on four RTX 3090 GPUs. Even across diverse workloads, the prediction error remains within 5%, confirming the estimator's generalization capability. With precise runtime estimation, ProTrain can effectively automate the selection of memory management strategies tailored to each training scenario.

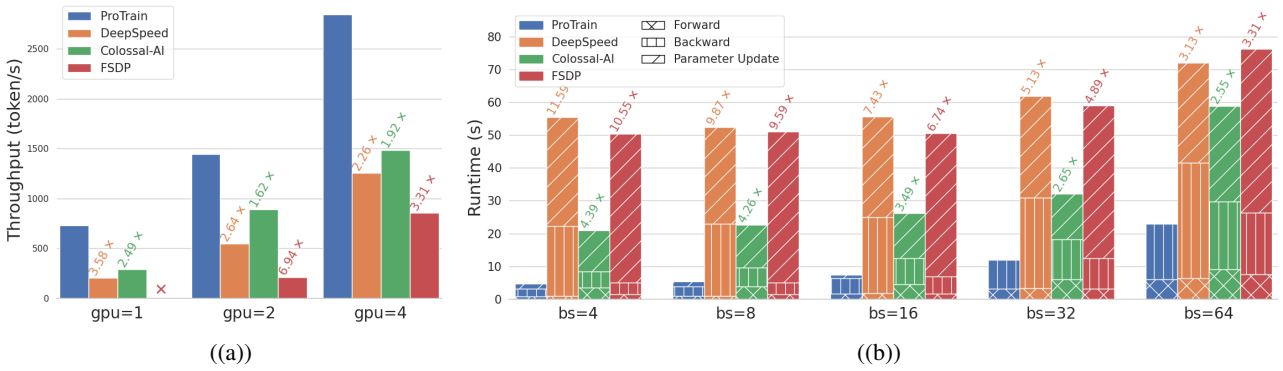
Figure 8 (bottom) extends this evaluation to the best configurations selected by ProTrain across various models and batch sizes. Although larger batch sizes generally require more memory, ProTrain reduces persistent chunks and chunk buffers under tight budgets, resulting in lower overall memory usage. The estimator conservatively overestimates peak memory by at most 10%, effectively accounting for fragmentation and reducing the risk of OOM, thereby ensuring robust performance across varied training scenarios.

C.2.1 Evaluation of FSDP with Selective Checkpointing

We re-evaluated FSDP using selective gradient checkpointing to compare its impact across hardware. On RTX 3090 GPUs, it does not improve throughput because execution is communication-bound, which limits the benefit of reduced recomputation. In contrast, on A100 GPUs, selective checkpointing improves throughput for all models except 40B GPT-2, which fails to scale due to OOM issues. Table 5 reports the maximum throughput on A100 GPUs for FSDP with and without selective checkpointing, as well as ProTrain. While selective checkpointing improves FSDP's performance, ProTrain consistently delivers higher throughput by jointly optimizing memory usage and hardware efficiency.

D Related Work

Tensor Swapping and Gradient Checkpointing Tensor swapping (Rhu et al., 2016; Le et al., 2018; Huang et al., 2020; Ren et al., 2021; Sun et al., 2022) expands memory capacity by offloading tensors to external memory such as CPU memory. Early work focused on activation swapping (Rhu et al., 2016; Le et al., 2018), while later efforts extended swapping to parameters (Huang et al., 2020) and optimizer states (Ren et al., 2021). Gradient checkpointing (Chen et al., 2016; Jain et al., 2020; Herrmann et al., 2019; Zhao et al., 2023a; Korthikanti et al., 2023) reduces activation memory by recomputing intermediate results during the backward pass. Several studies (Peng et al., 2020; Beaumont et al., 2021; Nie et al., 2022) explore joint optimization of swapping and checkpointing, but primarily target convolutional networks and offer limited scalability to larger transformer models. In contrast, ProTrain integrates these techniques through transformer-aware abstractions and cost-driven coordination, enabling automatic and scalable memory management for large-scale model training.



((a)) ((b))

Figure 7. Scalability of performance on A100 GPUs (a) Maximum throughput across different numbers of GPUs (b) Step time breakdown for different batch sizes

Table 5. Training throughput of FSDP with and without selective checkpointing vs. ProTrain (tokens/s)

Model	FSDP + Selective Checkpointing	FSDP - Selective Checkpointing	ProTrain
LLaMA-13B	3996.67 (1.00×)	3715.12 (0.93×)	6471.32 (1.62×)
GPT2-20B	2392.17 (1.00×)	2136.16 (0.89×)	5043.75 (2.11×)
GPT2-30B	1383.52 (1.00×)	1307.88 (0.95×)	3431.38 (2.48×)
OPT-30B	1621.85 (1.00×)	1342.40 (0.83×)	3266.02 (2.01×)
LLaMA-34B	1247.25 (1.00×)	1024.23 (0.82×)	2845.18 (2.28×)
GPT2-40B	1143.06 (1.00×)	1208.68 (1.06×)	2723.50 (2.38×)

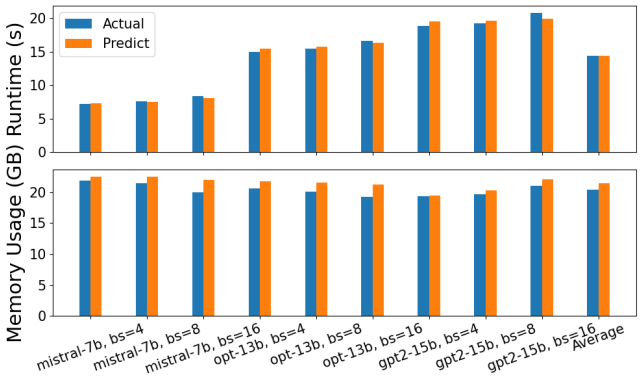


Figure 8. Predicted vs. actual runtime and peak memory usage across different models and batch sizes

Parallelization Techniques Parallelization is a widely adopted strategy to scale large model training by distributing computation and memory across multiple devices. Common approaches include: Data Parallelism (DP)(Li et al., 2020), which replicates the model and distributes input data across devices, can be enhanced by ZeRO (Rajbhandari et al., 2020; Zhao et al., 2023b) that partitions model states across GPUs to reduce memory redundancy; Tensor Parallelism (TP)(Shoeybi et al., 2019; Xu et al., 2021), which partitions tensors within operators; and Pipeline Parallelism (PP)(Huang et al., 2019; Narayanan et al., 2019), which splits the model into stages assigned to different devices for pipelined execution. Recent systems (Zheng et al., 2022; Lin et al., 2024; Miao et al., 2023) explore hybrid combinations of these parallelism strategies via automatic search. Mist (Zhu et al., 2025) goes further by co-optimizing parallelism with memory footprint reduction techniques including checkpointing, ZeRO, and offloading; we provide a detailed comparison in Section 6. While effective, these systems focus purely on parallel execution and overlook memory-saving strategies such as swapping and checkpointing. Our work complements them by integrating these techniques into the search space, enabling further scalability in model size or throughput under fixed resources. We focus on Data Parallelism with the ZeRO technique to demonstrate this integration in a practical and widely used setting, but our approach is orthogonal to other parallelisms and applicable to more complex training setups.

LLM Memory Optimization Techniques Beyond the aforementioned techniques, a range of system-level and algorithmic methods have been developed to further reduce memory consumption in LLM training. Compiler-level optimizations, such as operator fusion (Zhao et al., 2022; Ansel et al., 2024), eliminate intermediate tensors by merging operations during compilation. At the kernel level, memory-efficient attention mechanisms (Dao et al., 2022; Rabe &

Staats, 2021) reduce memory footprint by restructuring attention computation to avoid materializing the full attention matrix. Quantization methods reduce memory usage by compressing weights, activations, and gradients into lower-precision formats such as INT8 (Dettmers et al., 2022), FP8 (Fishman et al., 2024; Micikevicius et al., 2022), and FP4 (Yang et al., 2025). While orthogonal to our approach, these techniques are complementary and can be integrated into ProTrain to further improve scalability.

Training Frameworks for Transformers To meet the growing demand for efficient LLM training, numerous frameworks have been developed with distinct design goals and optimizations. DeepSpeed (Rasley et al., 2020) by Microsoft enhances training efficiency through ZeRO series techniques (Rajbhandari et al., 2020; Ren et al., 2021; Rajbhandari et al., 2021), and supports a wide range of parallelism strategies, tensor swapping, and so on. Colossal-AI (Li et al., 2023) from HPC-AI Tech, offers similar features but distinguishes itself with a chunk-based memory management design (Fang et al., 2022), which is also adopted in ProTrain. Megatron-LM (Shoeybi et al., 2019) by NVIDIA, on the other hand, focuses on model parallelism techniques. In parallel, recent academic efforts (Sun et al., 2022; Li et al., 2022; Feng et al., 2023) aim to enable efficient training on memory-constrained systems. ProTrain complements these frameworks by introducing automatic memory management that adapts to varying model and hardware configurations.