A Cost Models

A.1 Modeling Runtime

The total iteration time in ProTrain is determined by the duration of the forward pass, backward pass, and parameter updates, as defined in Equation 2. To estimate the forward computation time, ProTrain adopts a chunk-based approach, as most operations in Figure 1 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:

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

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 detailed 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:

$$T_{\text{BWD}} = \sum_{i=1}^{N_{\text{chunk}}+1} \max\left(T_{\text{comp}}^{\text{BWD}}(i) + T_{\text{recomp}}(i), T_{\text{comm}}^{\text{BWD\_prefetch}}(i-1), T_{\text{comm}}^{\text{reduce-offload}}(i+1)\right). \tag{5}$$

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:

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

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 both the GPU and CPU, depending on the chunk placement. For CPU-based updates, ProTrain employs the fast CPU Adam optimizer Ren et al. (2021), while GPU updates use the FusedAdam optimizer NVIDIA (2018). ProTrain models performance for both updates based on parameter size.

A.2 Modeling Memory Consumption

Accurately estimating peak memory usage is essential for efficient 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 to $M_{\text{FWD}} + \sum_{i=1}^{N_{\text{op}}} M_{\text{Act}}^{\text{Op}}(i)$. These values are iteratively updated for each operator using Equation 8 and 9:

$$M_{\text{Cur}}(i) = M_{\text{Cur}}(i-1) + \Delta M_{\text{Cur}}^{\text{PriorOp}}(i) + \Delta M_{\text{Cur}}^{\text{Op}}(i) - M_{\text{Act}}^{\text{Op}}(i), \tag{8}$$

$$\begin{aligned} M_{\text{Peak}}(i) = max\{&M_{\text{Peak}}(i-1), M_{\text{Cur}}(i-1) + \Delta M_{\text{Peak}}^{\text{PriorOp}}(i), \\ &M_{\text{Cur}}(i-1) + \Delta M_{\text{Cur}}^{\text{PriorOp}}(i) + \Delta M_{\text{Peak}}^{\text{Op}}(i)\}. \end{aligned} \tag{9}$$

This iterative, operator-wise approach allows us to recover the peak memory usage by accounting for both the transient nature of temporary tensors, which are typically confined to individual operators, and the longer life cycle of activations, which span across multiple operations depending on the execution order. The final value obtained from Equation 9, denoted as $M_{\text{Peak}}^{\text{Base}}$, serves as the foundational baseline for estimating peak memory usage across various configurations. Building on this, the final peak memory for any specific configuration is computed as:

$$\begin{aligned} M_{\text{Peak}} = M_{\text{Peak}}^{\text{Base}} + M_{\text{persist}} \cdot n_{\text{persist}} + M_{\text{buffer}} \cdot n_{\text{buffer}} - M_{\text{swap}} \cdot n_{\text{swap}} \\ - M_{\text{checkpoint}} \cdot n_{\text{checkpoint}} + \begin{cases} M_{\text{checkpoint}}, & \text{if } n_{\text{checkpoint}} + n_{\text{swap}} = N_{\text{block}}, \\ 0, & \text{otherwise}, \end{cases} \end{aligned} \tag{10}$$

where M_{persist} and M_{buffer} represent the memory allocated for a single persistent chunk and chunk buffer, and M_{swap} and $M_{\text{checkpoint}}$ reflect the memory savings from activation swapping and gradient checkpointing for a single transformer block, respectively. When all blocks are involved in either swapping or gradient checkpointing, recomputation during the backward pass is inevitable, leading to an increase in memory consumption. Furthermore, actual memory usage is typically higher than estimates due to memory fragmentation, so we include a fragmentation factor in the final estimation.

B Implementation Details

B.1 Adaptive Chunk Size

ProTrain employs a dynamic search mechanism to determine the optimal chunk size for model training, which organizes parameters according to their execution order and ensures that all parameters within a block are grouped in a single chunk. For transformers that share parameters across layers, ProTrain uses the parameter's first occurrence as the ordering criterion. To find the most efficient chunk size, ProTrain conducts a grid search, simulating memory waste across various chunk sizes to identify the size that minimizes waste.

B.2 Memory Optimizations

Proactive Memory Allocation ProTrain preallocates memory for tensors that persist until training completes, including early allocation of persistent chunks for parameters and optimizer states, as well as GPU chunk buffers. This proactive strategy reduces the number of memory allocations and mitigates fragmentation by grouping long-lived tensors together, ensuring a more organized and efficient memory layout.

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 Experiment Settings

C.1 Model Configurations

The model configurations used in the experiment are shown in Table 2. The underlying model implementation is from the HuggingFace library.

Table 2: Model Configuration

Model	Parameter Size	Hidden Size	# of Layers	# 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

C.2 Hardware Configurations

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.

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.

C.3 Baseline Configurations

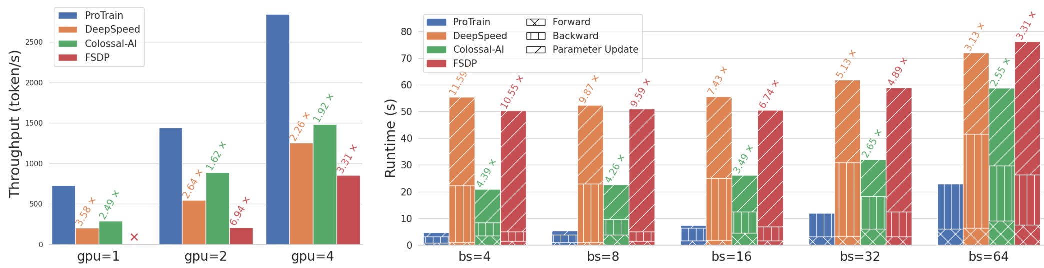
For our experiments, we utilized DeepSpeed-0.12.1, enabling ZeRO-3 alongside offloading of both parameters and optimizer states. The configuration was fine-tuned for optimal performance, with key settings including `stage3_max_live_parameter`, `stage3_max_reuse_distance`, `stage3_prefetch_bucket_size` and `reduce_bucket_size`.

In the case of Colossal-AI, we leveraged version 0.3.3 along with the Gemini Plugin to facilitate chunk-based memory management. This setup featured a static placement policy and also enabled the offloading of parameters and optimizer states to make large models trainable.

For Fully Sharded Data Parallel (FSDP) which is integrated within PyTorch-2.0.1, we employed the `transformer_auto_wrap_policy` to ensure that each transformer block was encapsulated within a single `FlatParameter`. We also enable CPU offloading to accommodate the training of larger models.

D Full Experiment Results

D.1 Throughput Scalability on A100 GPUs



(a) (b)

Figure 6: Scalability of performance on A100 GPUs (a) Maximum throughput across different numbers of GPUs (b) Step time breakdown for different batch sizes

Figure 6(a) presents the scalability performance of ProTrain for LLaMA 34B on four A100 GPUs compared to other frameworks. ProTrain demonstrates superior scalability, achieving a $2.49\times$ to $3.58\times$ speedup over a single GPU setup. The increased performance on A100 GPUs, compared to RTX 3090 GPUs, can be attributed to ProTrain's advanced memory management, which maximizes the utilization of the A100's larger memory capacity and higher bandwidth. This allows ProTrain to effectively scale with larger batch sizes, fully leveraging the additional resources to improve the training throughput.

Figure 6(b) breaks down the runtime per iteration into forward, backward, and parameter update phases across various batch sizes on A100 GPUs. ProTrain consistently outperforms other frameworks due to its efficient memory management and overlapping strategies. One of the most significant improvements comes from its ability to overlap CPU parameter updates with backward computations, effectively hiding the update time and reducing it to nearly zero. This optimization ensures that parameter updates do not become a bottleneck, where other other frameworks experience significant slowdowns. For instance, FSDP spends considerable time in the parameter update phase due to its use of the default Adam optimizer, which is less efficient than the optimized variants used by ProTrain. On the other hand, ProTrain significantly reduces backward execution time compared to DeepSpeed, which relies on multiple thresholds for parameter prefetching and eviction, similar to a sliding window. In DeepSpeed's approach, parameters can only be evicted after full usage, and new ones are prefetched only if they fit into the freed memory, leading to inefficient bandwidth utilization. Overall, ProTrain delivers an average speedup of $3.47\times$ to $7.43\times$ compared to other frameworks, showcasing its superior performance across various setups.

D.2 Training Throughput w/ and w/o Offloading

Although ProTrain is designed for scenarios where the model cannot fully fit into GPU memory (requiring offloading), it also delivers excellent performance compared to baselines in non-offloading scenarios. As shown in Table 3, when DeepSpeed and Colossal-AI operate without offloading,

Table 3: Maximum Training Throughput on four A100 GPUs w/ and w/o Offloading (Unit: token/s)

Model		Mistral 7B	GPT-2 10B	LLaMA 13B	GPT-2 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

their training throughput improves for smaller models. However, as model size increases, the batch size that can be trained without offloading decreases, diminishing the performance advantage. For instance, Colossal-AI's performance on LLaMA 13B is 15% slower without offloading compared to with offloading. Overall, regardless of whether the baselines use offloading or not, ProTrain consistently achieves the best performance, showing its versatility and adaptability across different training scenarios.

D.3 Training Performance on AMD MI300X GPUs

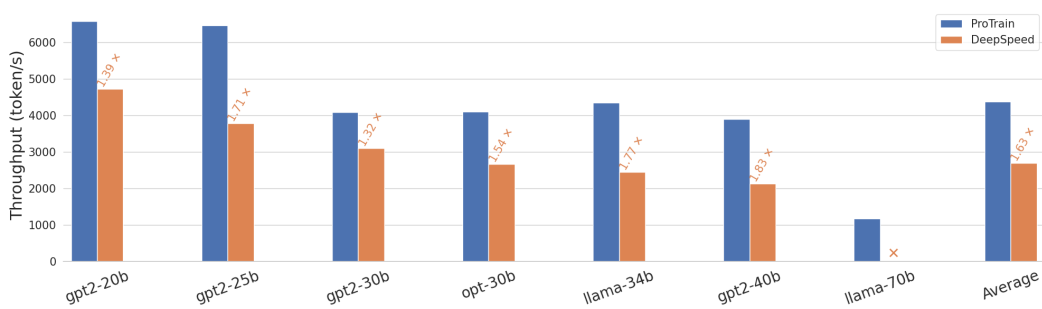


Figure 7: Maximum Training Throughput on four AMD MI300X GPUs

Figure 7 presents the throughput comparison between ProTrain and DeepSpeed across various model sizes on AMD Instinct™ MI300X GPUs, which feature 192 GB of HBM3 memory and provide 5.3 TB/s peak memory bandwidth. This extensive memory capacity and bandwidth, along with Infinity Fabric interconnect technology, enables superior multi-GPU scaling compared to RTX 3090 and A100 GPUs, making it especially advantageous for training larger models. As demonstrated in the results, ProTrain consistently surpasses DeepSpeed, with speedups ranging from 1.39× to 1.83× across all model configurations. This performance improvement highlights ProTrain's ability to leverage the high memory bandwidth and capacity, resulting in better hardware utilization and overall performance.

D.4 Effect of Runtime/Peak Memory Usage Estimator

Figure 8 compares predicted versus actual runtime and peak memory usage using ProTrain's chosen configuration on four RTX 3090 GPUs. The top chart shows the runtime prediction error does not exceed 5%, reflecting the high accuracy of the runtime estimator across different models and batch sizes. The bottom chart compares the predicted and actual peak memory usage, measured using `max_memory_allocated`. Prediction error increases slightly with larger batch sizes, typically overestimating by no more than 10%. This conservative estimation helps mitigate the risk of out-of-memory errors by accounting for memory fragmentation, thus ensuring reliable performance in diverse training conditions. Overall, these results validate ProTrain's estimators for both runtime and memory, confirming their reliability in automatic memory management.

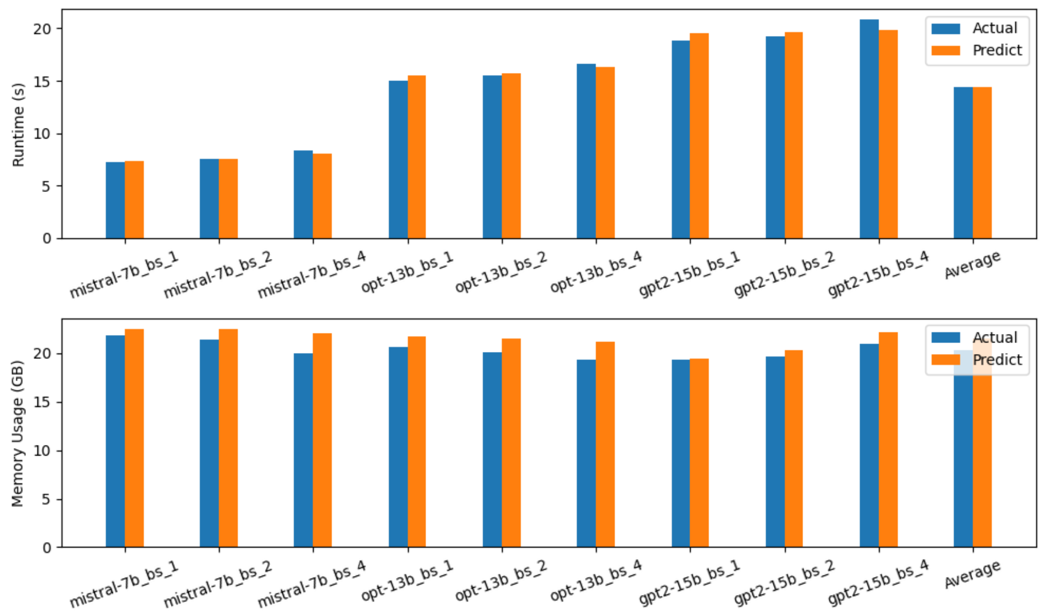


Figure 8: Comparison of Predicted vs. Actual Runtime and Peak Memory Usage for Various Models

E Related Work

Swapping and Recomputation Swapping Rhu et al. (2016); Le et al. (2018); Huang et al. (2020); Ren et al. (2021); Sun et al. (2022) is a commonly employed technique which leverages external memory such as CPU memory to offload tensors, thereby expanding the available memory for training. Traditional swapping methods mainly focus on offloading activations, SwapAdvisor Huang et al. (2020) extends it to parameters and ZeRO-offload Ren et al. (2021) further extends it to optimizer states. Recomputation Chen et al. (2016); Jain et al. (2020); Herrmann et al. (2019); Zhao et al. (2023a); Korthikanti et al. (2023), also known as gradient checkpointing, is another widely used technique that trades additional recompute time during backward pass for reduced memory usage of activations. Initially, Chen et al. Chen et al. (2016) focuses on homogeneous sequential networks, and subsequent studies Jain et al. (2020); Herrmann et al. (2019) extended its applicability to heterogeneous networks. Considering the scale and complexity of Transformers, which often contain numerous layers, previous approaches become less efficient. Therefore, Rockmate Zhao et al. (2023a) optimizes the plan generation by partitioning models into fine-grained blocks. NVIDIA further proposes selective activation recomputation which checkpoints and recomputes parts of layers Korthikanti et al. (2023). To get the best of both worlds, some works Peng et al. (2020); Beaumont et al. (2021); Nie et al. (2022) jointly optimize swapping and recomputation, whereas ProTrain differentiates itself by tailoring to fit the specific structure of transformers.

ZeRO Techniques. ProTrain adopts ZeRO to manage model states. The Zero Redundancy Optimizer (ZeRO) Rajbhandari et al. (2020) distributes model states across multiple GPUs to reduce memory pressure of each GPU. ZeRO operates in three stages: ZeRO-1 partitions optimizer states across GPUs; ZeRO-2 extends this by also distributing gradients; and ZeRO-3 further divides the parameters, which are required to be gathered before forward/backward computation. The ZeRO techniques have been integrated into state-of-the-art frameworks such as DeepSpeed Rasley et al. (2020), FSDP Zhao et al. (2023b), and Colossal-AI Li et al. (2023), each differing in their parameter organization to optimize bandwidth utilization. Unlike DeepSpeed and FSDP, which require manual configuration for parameter grouping, Colossal-AI automatically groups parameters into chunks and dynamically adjusts their size according to the model's scale. This chunk-based method, inspired by PatrickStar Fang et al. (2022), is also adopted in ProTrain.

GPU Memory Management Deep learning frameworks, such as PyTorch Paszke et al. (2019) and TensorFlow Abadi et al. (2016), utilize caching allocators for efficient memory management.

However, these frameworks often face memory fragmentation issues, particularly when integrating memory-saving techniques like swapping, recomputation, and parallelization, which hurts allocation efficiency. To address this, two main approaches have been proposed. The first is profiling-guided optimization Sekiyama et al. (2018); Steiner et al. (2022; 2023), which leverages the repetitive and predictable nature of memory allocation patterns during training. This method traces and analyzes tensor allocations and deallocations to optimize tensor placement, thus improving memory efficiency. Alternatively, GMLake Guo et al. (2024) introduces Virtual Memory Stitching, a technique that merges non-contiguous memory blocks, thereby reducing memory fragmentation at the operating system level. These approaches are orthogonal to ProTrain's method. Angel-PTM Nie et al. (2023) adopts a page-based memory management strategy that partitions model states to reduce the memory fragmentation. In contrast, ProTrain designs a new chunk-based memory management inspired by PatrickStar Fang et al. (2022) grouping model states into chunks that align with the runtime execution order, which not only improves bandwidth utilization but also enhances memory locality.

Overlapping Computation and Communication There are numerous work on overlapping computation and communication, with many studies Mahajan et al. (2023); Hashemi et al. (2019); Peng et al. (2019); Jangda et al. (2022); Chen et al. (2024) focus on substituting, splitting, and scheduling complex operators to achieve fine-grained overlapping. CoCoNet Jangda et al. (2022) enhances lower-level operator optimization, while Centauri Chen et al. (2024) extends this to graph-level scheduling, offering a more hierarchical abstraction. Despite these advances, most research focuses on the optimization of collective communication operations in distributed cases. However, ProTrain also considers the communication between CPU and GPU under limited GPU memory conditions, making it orthogonal to existing research.

Training Frameworks for Transformers In response to the growing demand for efficient training of transformers, several specialized frameworks have been developed, each offering unique features 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 various parallelism strategies, swapping, and recomputation. Colossal-AI Li et al. (2023) from HPC-AI Tech, which offering similar features, distinguishes itself with a chunk-based memory management approach Fang et al. (2022), which our work adopts. Megatron-LM Shoeybi et al. (2019) by NVIDIA, on the other hand, specializes in model parallelism. These frameworks are designed for large-scale transformer training, complemented by academic efforts Sun et al. (2022); Li et al. (2022); Feng et al. (2023) to facilitate training on smaller systems.