DeepCompile: A Compiler-Driven Approach to Optimizing Distributed Deep Learning Training

Masahiro Tanaka
Microsoft
mtanaka@microsoft.com

Du Li
Microsoft
duli.personal@gmail.com

Umesh Chand*
Microsoft
Umesh.Chand@amd.com

Ali Zafar
University of Virginia
mzw2cu@virginia.edu

Haiying Shen
University of Virginia
hshen@virginia.edu

Olatunji Ruwase
Microsoft
olruwase@microsoft.com

Abstract

The increasing scale of deep learning models has led to the development of various parallelization strategies for distributed training across accelerators. For example, fully sharded approaches like DeepSpeed ZeRO-3 and FSDP partition the parameters of each layer across multiple GPUs and gather them through communication when needed. These methods rely on optimizations such as prefetching, which initiates communication early to overlap it with computation and reduce communication overhead, and unsharding, which retains as many parameters in their unsharded form as possible to reduce communication volume. Although the timing of prefetching should be adjusted in response to dynamic memory usage during execution, these systems lack the flexibility to control it, which limits the benefits of prefetching. Moreover, they cannot anticipate how memory usage will change after prefetching is applied, making it difficult to combine it effectively with other optimizations such as unsharding. We present DeepCompile, which compiles user-defined models into computation graphs and applies a sequence of profiling-guided optimization passes for distributed training. Taking dynamic memory usage into account, these passes flexibly insert, reorder, or remove operations to improve communication–computation overlap, reduce memory pressure, and coordinate multiple optimizations in a unified manner. To evaluate the effectiveness of this design, we implemented a fully sharded approach like ZeRO-3 and FSDP on top of DeepCompile, along with three optimizations: proactive prefetching, selective unsharding, and adaptive offloading. We evaluate DeepCompile on the training of Llama 3 70B and Mixtral 8×7B MoE models. DeepCompile achieves up to 1.28× and 1.54× performance improvements over ZeRO-3 and FSDP baselines, respectively, and up to a 7.01× throughput increase in settings with limited GPU resources, using offloading.

Keywords

deep learning, distributed training

*Work done while at Microsoft. Now at AMD.

1 Introduction

The rapid growth of deep learning has led to the emergence of increasingly large models. Modern architectures often contain billions of parameters, resulting in immense computational demands. In this context, efficient parallelization across multiple accelerator devices, such as GPUs, has become essential for achieving feasible training times and manageable costs.

There are several parallelization strategies for training large models: In data parallelism, all model parameters are replicated across multiple GPUs. This approach is natively supported in many deep learning frameworks, including PyTorch [21]. Pipeline parallelism [11] instead splits the model by layers, assigning each block to a different GPU. Tensor parallelism [25] divides parameter tensors within a layer across GPUs, where each device computes a partial result that is later aggregated. In addition to these, the *fully sharded approach*, which is implemented in DeepSpeed ZeRO-3 [22] (hereafter ZeRO-3) and Fully Sharded Data Parallel (FSDP) [27], partitions the parameters of each layer across multiple GPUs and gathers them through communication before each layer's computation begins.

While these strategies provide scalability, efficiently managing memory and overlapping communication with computation to reduce communication overhead [31, 32] remain open challenges. Existing systems rely on various optimization techniques to address these issues. In the context of the fully sharded approach, such as ZeRO-3 and FSDP, *prefetching* initiates communication early to gather partitioned parameters and overlap it with computation; *unsharding* keeps parameters in their full form to reduce communication when memory permits; and *offloading*, which moves data from GPU memory to host memory, is often used in combination when GPU memory is insufficient.

Although these optimizations are effective in principle, applying them efficiently in practice is not straightforward. Take prefetching as an example. In the fully sharded approach, parameters partitioned across multiple GPUs must

be gathered via all-gather communication, which requires a large buffer to hold the gathered parameters. Initiating the all-gather earlier increases opportunities to overlap communication with computation. However, it also extends the buffer's lifetime and increases memory pressure, since the buffer cannot be released until all computations that use the parameters have completed. Similarly, the overhead of offloading can be mitigated by minimizing data transfers between host and GPU memory and overlapping them with computation. To make these optimizations effective, it is essential to flexibly adjust the timing of all-gather and offloading operations, monitor memory usage trends, and analyze data dependencies across broad scopes such as the forward and backward passes.

Moreover, when multiple optimizations are applied together, their interactions must be carefully coordinated. For instance, unsharding can improve performance when communication is on the critical path, but when combined with prefetching, it must also account for the additional memory pressure introduced by early optimization.

In fully sharded approaches like ZeRO-3 and FSDP, all-gather operations are implemented via runtime hooks inserted into the user-level code (typically in Python). Optimizations such as prefetching and unsharding are supported, but this design lacks the flexibility to adjust the timing of all-gather or offloading operations based on runtime memory usage over broader scopes, such as an entire forward or backward pass. Moreover, there is no clear mechanism to coordinate multiple optimizations in a unified manner.

Other systems have been proposed to automatically combine multiple optimizations for distributed training based on profiling. Examples include Alpa [19], FlexFlow [12], and Unity [29], which aim to determine effective combinations of parallelization strategies such as data, tensor, and pipeline parallelism, as well as to apply kernel-level optimizations like kernel fusion. However, these systems primarily focus on static planning of parallelization strategies and kernel-level optimizations. They do not take into account runtime dynamics such as changes in memory usage during execution. As a result, they lack mechanisms to adjust communication schedules, adapt to memory availability at runtime, or coordinate optimizations such as prefetching, unsharding, and offloading.

To address these limitations, we introduce DeepCompile, a compiler-based system that uses an existing compiler to convert the model into a computation graph and then transforms it for optimization. DeepCompile applies a series of *optimization passes*, where each pass targets a specific goal—such as improving communication overlap—by inserting, removing, or reordering operations in the graph. These passes make decisions based on profiling such as operator execution time and memory usage trends over broader scopes, including entire forward and backward passes. Since each pass produces a modified graph, subsequent passes can analyze the result and flexibly adapt to the effects of earlier transformations.

To evaluate the effectiveness of this design, we implemented the fully sharded approach, which is similar to ZeRO-3 and FSDP, as a series of optimization passes within DeepCompile. In addition, we implemented the following key optimizations as separate passes.

- **Proactive prefetching.** To maximize overlap between communication and computation, this optimization pass initiates all-gather as early as possible, considering how available memory changes as the forward and backward passes progresses.
- **Selective unsharding.** This pass keeps as many parameters unsharded as possible to reduce communication overhead caused by all-gather communication, and decides which parameters to unshard based on operator-level memory profiling.
- **Adaptive offloading.** DeepCompile offloads optimizer states such as momentum and variance used by Adam [14] to CPU memory when GPU memory is insufficient. To reduce data transfer overhead, it offloads only the amount of data that exceeds the memory limit and schedules transfers to overlap with computation.

SimpleFSDP [33] takes a similar approach to DeepCompile by using compilation to optimize the placement of all-gather operations in the fully sharded approach. However, it focuses specifically on prefetching and does not address other optimizations such as unsharding or offloading, which leads to notable differences in design. In SimpleFSDP, communication operations are inserted into the user-level code (Python) before compilation, whereas DeepCompile first compiles the model into a computation graph and then applies transformations. In SimpleFSDP's design, the compiler must correctly lower these manually inserted communication operations into graph-level operators. While the PyTorch compiler currently supports lowering all-gather operations used by SimpleFSDP, it may fail to handle other optimizations, such as offloading, if they are implemented at the user level. This design limits the extensibility of SimpleFSDP in supporting and combining a broader range of optimizations.

We evaluated DeepCompile on large-scale distributed training tasks using Llama 3 70B and Mixtral 8x7B MoE models. Across all configurations, DeepCompile consistently improved training efficiency compared to baselines, including ZeRO-3, FSDP, and setups using PyTorch compiler optimizations. It achieved up to 1.28× improvement on Llama 3 70B and up to 1.54× on the Mixtral 8x7B MoE model. Additionally, it demonstrated a 7.01× increase in throughput when using fewer GPUs with offloading.

2 Background and Motivating Example

2.1 Distributed Training

The emergence of Transformer architectures [30], which enabled breakthroughs like BERT [6], has significantly accelerated the growth of modern deep learning models, such as GPT-3 [3], Llama-2 [28], Llama-3 [10], Phi-3 [1], PaLM [5], and OPT [34]. This trend has led to a tremendous increase in computational demands. While accelerator devices such as GPUs have become the standard, cutting-edge models now require distributed training across multiple GPUs to be feasible. For example, Llama-3 [10] was trained on 16,384 NVIDIA H100 GPUs over 54 days. Even fine-tuning, which adapts pre-trained models to specific downstream tasks, often involves dozens of GPUs running for hours or days.

In distributed training, where multiple GPUs are used in parallel, reducing overhead from both inter-device communication and memory management is critical for efficiently training large-scale models.

2.2 Parallelization Strategies

Various parallelization strategies have been proposed to improve the efficiency of distributed training.

Data parallelism replicates the entire model on each GPU, and each GPU processes a different portion of the training data. A key advantage is that it can be used with any model architecture without requiring code changes, and the communication overhead is relatively low. However, because each GPU must store a full copy of the model, large models may exceed the memory capacity.

Unlike data parallelism, *pipeline parallelism* partitions the model to fit within GPU memory by grouping layers into coarser-grained stages and placing each stage on a different GPU. The training batch is split into smaller fragments and processed in a pipelined fashion. To fully utilize all GPUs, the compute time of each stage should be approximately balanced. In addition, the choice of partition points affects the communication volume between GPUs. These factors often require manual tuning to achieve optimal performance.

Tensor parallelism is primarily applied to the multi-head attention and MLP layers in Transformer models. It splits specific parameter tensors along designated dimensions across GPUs, performs computations locally, and aggregates results using all-reduce communication. For example, in an MLP block, the first weight matrix is typically split horizontally, and the second one vertically; each GPU computes partial results, and an all-reduce is applied after the second linear transformation to combine them. In addition, although well-tested implementations exist for standard architectures such as GPT [3], adapting tensor parallelism to model variants such as multi-query attention (MQA) [23], grouped-query attention (GQA) [20], or multi-head latent attention [16] requires additional engineering effort. In each case, developers must manually identify which parameters can be split, choose appropriate dimensions for partitioning, and ensure correct synchronization across devices.

An alternative to pipeline and tensor parallelism is the *fully sharded approach*, which partitions the parameters of each layer across multiple GPUs. Frameworks such as Fully Sharded Data Parallel (FSDP) and ZeRO-3 adopt this strategy. During training, each GPU gathers its required parameter shards using all-gather communication before computing each layer. To reduce peak memory usage, the gathered parameters are discarded immediately after the layer's computation finishes. This approach is widely adopted because it can be applied to arbitrary model architectures without requiring manual restructuring. In this work, we focus on optimizing this fully sharded approach.

2.3 Deep Learning Compiler

Deep learning compilers have rapidly advanced in recent years [2, 4, 9]. They translate high-level models into computation graphs, which enables graph-level analysis and flexible optimizations such as operator reordering, fusion, and memory reuse.

Major frameworks have been integrating compiler capabilities into their ecosystems. For example, PyTorch now includes its own compiler infrastructure to enable graph-based optimizations [2]. Since these compilers can be applied to the vast number of existing model implementations, they are now widely used in practice.

2.4 Motivating Example

The fully sharded approach, as implemented in systems like ZeRO-3 and FSDP, employs runtime optimizations such as prefetching and unsharding. However, these optimizations alone are not sufficient to fully improve training efficiency. Let us take prefetching as an example. Prefetching aims to reduce communication overhead by initiating all-gather operations earlier than the layer where the parameters are actually needed, thereby overlapping communication with computation. Because the fully sharded approach allocates a large buffer to store the gathered parameters, initiating all-gather earlier extends the buffer's lifetime and increases memory pressure. Therefore, prefetching needs to be scheduled based on memory usage patterns throughout the forward and backward passes.

Figure 1 illustrates a typical pattern of GPU memory usage during a forward and backward pass. Memory usage gradually increases during the forward pass as activations are stored, and decreases during the backward pass as activations are released. At the beginning of the forward pass,

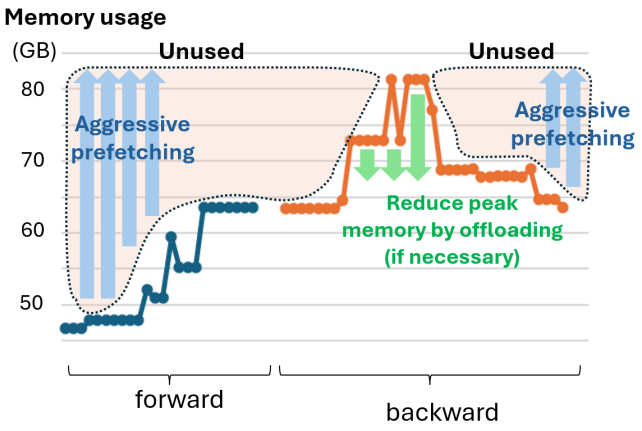


Figure 1: Memory usage trends and scheduling opportunities for prefetching and offloading. Profile of several final layers with a sequence length of 4096 and a vocabulary size of 128k. Significant memory spikes are observed in the log-softmax and negative log-likelihood loss layers.

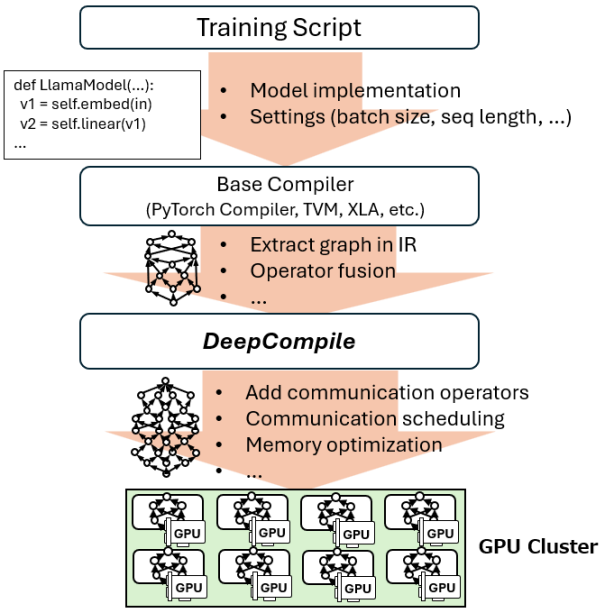


Figure 2: Workflow of compilation and optimization with DeepCompile

a substantial amount of unused memory is available, providing an opportunity to prefetch aggressively and increase computation-communication overlap. In contrast, near the end of the forward pass and throughout the backward pass, memory usage is high, limiting the opportunity for prefetching due to reduced memory availability. As the backward pass progresses and memory is gradually freed, more aggressive prefetching becomes possible again. Existing systems such as ZeRO-3 and FSDP typically allow users to specify a fixed buffer size for prefetching. However, this static configuration cannot adapt to changes in available memory as computation progresses, limiting the potential benefits of prefetching.

The same issue arises with offloading, which transfers data such as optimizer states (e.g., momentum and variance in Adam optimizer [14]) to host memory to reduce GPU memory usage. While offloading can significantly reduce peak GPU memory, the associated data transfers introduce overhead. A naive strategy is to offload states before starting the forward pass, but this leads to significant overhead due to idle GPU time. Instead, data transfers and computation can be overlapped. By initiating transfers at the beginning of the forward pass and synchronizing their completion just before memory usage rises to its limit, it is possible to hide the cost of data movement. In the backward pass, the opposite strategy can be applied: as memory usage decreases, transfers from host memory to GPU can be initiated.

These observations highlight the limitations of existing approaches. To address these limitations, we introduce DeepCompile, a compiler-based approach designed to systematically optimize fully sharded training.

3 System Design

We now describe the design of DeepCompile, a compiler-based system that enables graph-level transformations to support flexible and coordinated optimization of distributed training.

Figure 2 shows the overall workflow. Starting from a user-defined training script written in a framework such as PyTorch, a base compiler (e.g., the PyTorch compiler) lowers the model into an intermediate representation (IR) as a computation graph. DeepCompile takes this graph as input, transforms it by adding communication operations for distributed training, and applies optimizations. The resulting graph is then deployed to a runtime engine running on GPU servers.

Note that DeepCompile does not assume that distributed training logic is manually written in the model code. Instead, it targets standard, framework-native implementations, such as those commonly found on the HuggingFace Model Hub[1], and programmatically adds necessary operators including communication and synchronization operations to the graph.

[1]https://huggingface.co/models

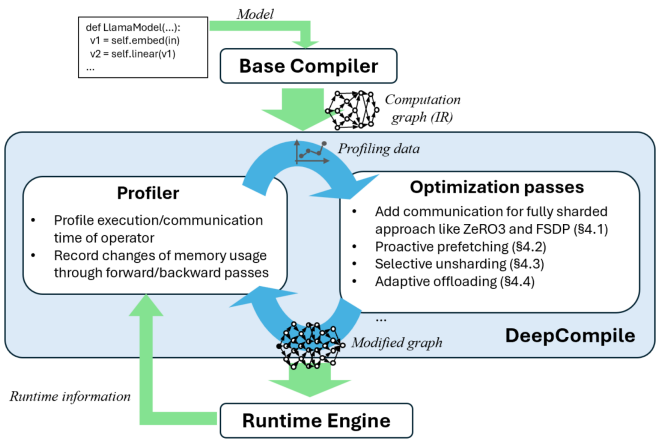


Figure 3: Optimization and profiling loop in DeepCompile

To support dynamic and memory-aware optimizations, DeepCompile organizes its graph transformations as a sequence of *optimization passes*. Each pass rewrites the computation graph based on a specific strategy, such as inserting communication operators or reordering memory-intensive computations. After applying each pass, DeepCompile executes the modified graph to collect runtime profiling data, including operator execution times, communication overhead, and memory usage trends. This information is then used to guide the next optimization pass. By repeating this process, DeepCompile incrementally refines the graph with awareness of runtime behavior.

In addition, to account for memory usage changes that arise during actual training, DeepCompile periodically runs short training iterations between groups of optimization passes. While DeepCompile focuses primarily on the forward and backward passes, other parts of the training process, such as parameter updates, can significantly affect memory usage. For example, the Adam optimizer typically allocates a large buffer after the first backward pass completes. To capture such changes, DeepCompile runs several training iterations to reflect the updated memory dynamics, then applies another round of optimization passes adapted to the new conditions.

Figure 3 illustrates this two-level loop: an inner loop of optimization and profiling, and an outer loop that periodically runs training to reflect changes in the runtime environment. By alternating between these loops, DeepCompile adapts its transformations to the evolving memory and execution characteristics of the model. This enables coordinated application of optimizations such as prefetching, unsharding, and offloading in a unified and data-driven manner.

While DeepCompile is applicable to a wide range of optimization problems, this paper focuses on one representative use case: the fully sharded approach, as implemented in ZeRO-3 and FSDP. We take advantage of DeepCompile's graph-level transformation and profiling infrastructure to implement this use case. Specifically, we incorporate a series of optimization passes that target key performance bottlenecks, including proactive prefetching to overlap communication with computation, selective unsharding to reduce redundant data communication, and adaptive offloading to mitigate GPU memory pressure.

In the following sections, we describe the implementation of this fully sharded training strategy using DeepCompile and detail the design and impact of each optimization pass.

4 Optimizations

DeepCompile enables a broad range of optimizations by leveraging a computational graph extracted by the base compiler. In this paper, we first demonstrate how to implement the fully-sharded approach. We then describe additional optimizations to reduce communication overhead and reduce memory requirements.

4.1 Fully-sharded approach

In the fully sharded approach, each parameter tensor is evenly partitioned across all GPUs. Before a layer is computed, each GPU gathers the required parameter shards via an *all-gather* communication, reconstructing the full parameters locally. Once the layer computation is complete, the gathered parameters are discarded to free memory.

Existing systems such as ZeRO-3 and FSDP insert all-gather and memory release operations around each layer that uses a given parameter. When a model has many layers with small parameter tensors, this results in frequent all-gather operations on small fragments. Since all-gather communication is inefficient for small data sizes (Fig. 5(c)), this leads to significant communication overhead. To mitigate this, both ZeRO-3 and FSDP allow users to manually group multiple layers into a larger block, reducing the number of all-gather operations. However, this increases the lifetime of the buffers holding gathered parameters. Furthermore, none of the layers in the block can begin computation until all of their associated all-gather operations have completed, often resulting in significant idle time. As a result, manually tuning the block size for optimal performance is difficult.

DeepCompile operates by modifying the computation graph to insert all-gather and release operations in a similar manner to existing frameworks. However, unlike systems that rely on layer boundaries or manually defined blocks, DeepCompile uses operator-level dependency analysis to determine the optimal placement of these operations.

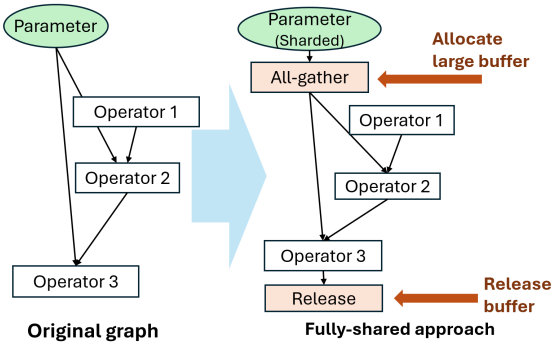


Figure 4: Graph modification

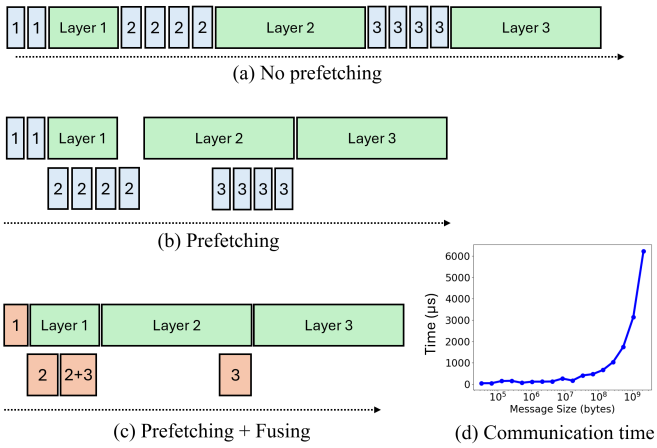


Figure 5: Parameter sharding and prefetching

As a first step toward enabling the fully sharded approach, DeepCompile schedules all-gather operations just before each parameter's first use, and release operations immediately after its last use, minimizing the lifetime of each buffer. Figure 4 shows an example of this process. If a parameter is used by multiple operators, DeepCompile analyzes their dependencies to ensure that the buffer is released only after its final use.

Although this scheduling improves memory efficiency, it does not necessarily improve execution speed. DeepCompile addresses this by applying a series of optimization passes. The initial pass focuses on ensuring the model fits within available memory, and subsequent passes improve training efficiency by adjusting the timing of communication, avoiding unnecessary sharding, and overlapping data transfers with computation.

4.2 Proactive prefetching

The initial graph transformation for the fully sharded approach in DeepCompile produces behavior similar to that of ZeRO-3 or FSDP operating in their most fine-grained configuration, where each layer triggers an all-gather before its computation and a release immediately after. However, because DeepCompile expresses this logic at the level of the computation graph, it enables detailed analysis of operator dependencies and memory usage patterns. This graph-based representation provides a foundation for more precise and globally coordinated optimizations, such as prefetching, which go beyond what is possible with heuristic or manually configured approaches.

With the fully sharded approach implemented in the computation graph, DeepCompile can build on top of it by introducing prefetching. Prefetching is a commonly used technique in fully sharded training systems, and both ZeRO-3 and FSDP include built-in support for it. The goal is to improve training efficiency by overlapping communication with computation, reducing the impact of all-gather latency.

Figure 5(a) illustrates the timeline of a naive behavior of fully sharded approach, where green boxes represent computation and blue boxes indicate all-gather communication. The number in each blue box corresponds to the respective layer. Since all-gather communication is the primary source of overhead, both ZeRO-3 and FSDP employ prefetching to mitigate its impact. By launching all-gather earlier than the parameters are used, these methods increase opportunities to overlap communication and computation (Fig. 5(b)).

In existing frameworks, prefetching is implemented in a static and coarse-grained manner. ZeRO-3 issues prefetches based on a predefined buffer size, launching all-gather operations for as many upcoming parameters as can fit in that buffer. Additionally, by grouping multiple layers into a larger block, ZeRO-3 and FSDP can fuse several all-gather operations. This improves communication efficiency, particularly for small data sizes. However, as discussed earlier, it can also delay the computation of earlier layers because the system must wait until all parameters involved in the fused communication are gathered.

Furthermore, ZeRO-3 and FSDP are not designed to account for dynamic changes in memory availability. As shown in Figure 1, memory usage fluctuates throughout the forward and backward passes. More memory tends to be available in the early stages of the forward pass and the later stages of the backward pass. However, existing methods do not effectively take advantage of these patterns. As a result, prefetching opportunities are often underutilized, leading to suboptimal efficiency.

To address this limitation, we propose an optimization pass for *proactive prefetching*. Figure 6 illustrates the memory usage trend and scheduling behavior with and without proactive prefetching. The goal of this optimization pass is

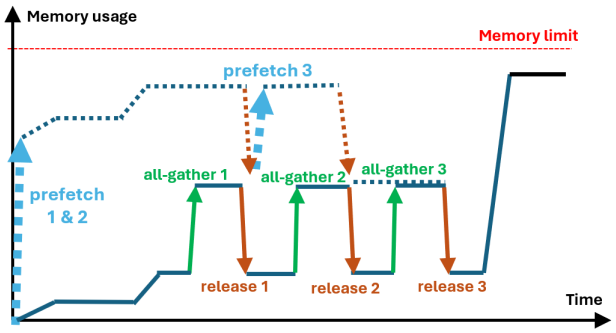


Figure 6: Memory usage and prefetch scheduling decisions. Solid lines indicate memory usage without prefetching; dotted lines show usage with proactive prefetching. All-gather and release operations are labeled accordingly.

to move all-gather operations as early as possible while ensuring that memory constraints are not violated. Since the memory usage of each operator and the size of the all-gather buffers are known from profiling, it is possible to estimate the total memory footprint if an all-gather is scheduled earlier than usual.

In Fig. 6, the dotted line shows the effect of proactive prefetching. In this example, there are three all-gather operations. Based on profiling results, the system determines that prefetching the first two can be safely done at the beginning without exceeding the memory limit. The third all-gather, however, would exceed the memory budget if issued immediately. Therefore, it is scheduled only after the buffer from the first parameter has been released. This scheduling strategy ensures efficient memory usage and allows prefetching to be applied as aggressively as possible within the memory budget.

Algorithm 1 formalizes this procedure. It examines each all-gather operation and determines whether it can be moved earlier in the schedule without exceeding the memory limit, based on the profiled memory usage and buffer size. The algorithm iterates over the operators in the initial schedule S_0 in reverse order. If the operator o_i is an all-gather, it checks whether the operator can be moved earlier without violating the memory constraint.

If the estimated memory usage remains below the limit, the all-gather operator o_i is added to a group U of unscheduled all-gather operators that can be moved earlier. If the memory usage exceeds the limit, the algorithm schedules the all-gather operators in U at the current position, fusing them where appropriate using the `Fuse` function. This strategy ensures that memory usage stays within the limit M.

Table 1: Notations

Symbol	Description
M	Total memory usage limit
M_{prefetch}	Size limit for the buffer size of a prefetch group
N	Number of operators
S_0	Initial schedule of operators $[o_1, o_2, \ldots, o_N]$
OS	Fragments of optimizer states $[os_1, os_2, \ldots]$
$\mathrm{P}_{\text{mem}}(o)$	Profiled memory usage just before operator o
$\mathrm{B}_{\text{ag}}(o)$	Buffer size allocated by all-gather operator o
$\mathrm{B}_{\text{os}}(os_i)$	Size of optimizer state fragment os_i
$Fuse(U)$	Produce operator fusing all-gathers in U
$T_c(V)$	Communication time for V
α	Fusion threshold parameter

Algorithm 1 Proactive prefetching

Require: Initial schedule S_0, Memory limit M, Size limit for prefetch group M_{prefetch}.

$S \leftarrow []$ *// Operators in new schedule*
$U \leftarrow []$ *// Unscheduled all-gathers*
for $i = N$ **to** 2 **do**
  if o_i is *allgather* **then**
    $\tilde{m}_U \leftarrow \sum_{o \in U \cup \{o_i\}} \mathrm{B}_{\text{ag}}(o)$
    $\tilde{m}_{i-1} \leftarrow \mathrm{P}_{\text{mem}}(o_{i-1}) + \tilde{m}_U$
    if $\tilde{m}_{i-1} < M$ and $\tilde{m}_U < M_{\text{prefetch}}$ **then**
      $U \leftarrow U \cup \{o_i\}$ *// Add o_i to prefetch list*
    else
      $S_f \leftarrow \text{Fuse}(U), U \leftarrow []$
      Append S_f to S *// Schedule fused allgathers*
    end if
  else
    Append o_i to S
  end if
end for
$S_f \leftarrow \text{Fuse}(U)$
Append S_f to S *// Schedule all remaining allgathers*
return S

We also set the limit M_{prefetch} on the prefetch group, as moving too many allgather operations earlier does not provide any additional benefit. We can set this limit to a significantly larger value than the predefined prefetch buffer size used in existing implementations such as ZeRO-3 because we also ensure the total memory usage is less than the limit M. By imposing this limit, we ensure that some memory remains available for selective unsharding, which is discussed in the next section.

When the data size is small, the communication time remains nearly constant, even as the data size increases.

Therefore, fusing all-gather operators reduces communication overhead (Fig. 5(c)(d)), especially for small data sizes. The criterion for fusing two all-gather operations is as follows: if $T_c(V_1) + T_c(V_2) > \alpha \cdot T_c(V_1 + V_2)$, where V_1 and V_2 are the data sizes of two operations, fusion is beneficial and the operations are merged. Otherwise, they are left separate. Here, $T_c(V)$ is profiled at runtime, and α is a tunable parameter, set to 1.5 in our experiments.

SimpleFSDP [33] takes a similar approach to ours by optimizing prefetching for the fully sharded approach using a compiled computation graph. It performs memory-aware fusion of all-gather operations to reduce communication overhead. However, SimpleFSDP does not consider how memory usage dynamically changes during the forward and backward passes to determine how early each all-gather can be issued. In contrast, our proactive prefetching algorithm uses profiling data to make this decision, allowing communication to be overlapped with computation while staying within memory constraints.

4.3 Selective unsharding

After applying the optimization pass for proactive prefetching, some free memory may remain, since the prefetching size is deliberately limited. Selective unsharding leverages this remaining memory by keeping some parameters unsharded after they are gathered. This approach can significantly reduce communication overhead, although it requires additional memory.

In the fully sharded approach, parameters are distributed across all GPUs, and all-gather operations are required after each parameter update. Selective unsharding, however, allows gathered parameters to remain unsharded after the forward pass, eliminating the need for all-gather in the backward pass.

This technique is especially beneficial when used with gradient accumulation, a common method to increase the effective batch size without increasing per-GPU memory usage. Given a gradient accumulation step of N, the model performs N forward and backward passes while accumulating gradients before applying a single parameter update. Since parameters are not updated during this period, they can remain unsharded across multiple forward and backward passes.

We run profiling after applying proactive prefetching to measure the peak memory usage. Then, we select as many parameters to keep unsharded as possible, ensuring that the total buffer size does not exceed the memory limit. The selection considers both the buffer size of each all-gather operator o, denoted as $\mathrm{B}(o)$, and its communication time $T_c(\mathrm{B}(o))$. Parameters are prioritized based on the ratio $\frac{T_c(\mathrm{B}(o))}{\mathrm{B}(o)}$, where a higher value indicates a greater reduction in communication time relative to memory cost. Following the trend shown in Fig. 5(d), smaller parameters are generally selected first.

Algorithm 2 Adaptive offloading (forward)

Require: Initial schedule S_0, Memory limit M, optimizer state fragments OS
  $S \leftarrow []$ *// List of operators in output schedule*
  $M^- \leftarrow 0$ *// Offloaded size*
  $M_{\text{peak}} \leftarrow max_{o_i \in S_0} \mathrm{P}_{\text{mem}}(o_i)$
  $M_{opt} \leftarrow \sum_{os_i \in OS} \mathrm{B}_{\text{os}}(os_i)$
  $OS_{\text{offload}} \leftarrow []$
  for all $os_i \in OS$ **do**
    if $M_{\text{peak}} + M_{\text{opt}} - \sum_{os_i \in OS_{\text{offload}}} \mathrm{B}_{\text{os}}(os_i) > M$ **then**
      $OS_{\text{offload}} \leftarrow OS_{\text{offload}} \cup os_i$
      Append offload operator for os_i to S
    end if
  end for
  for all $o_i \in S_0$ **do**
    while $\mathrm{P}_{\text{mem}}(o_i) + M_{\text{opt}} - M^- > M$ **do**
      Pop os_i from OS_{offload}
      Append operator to synchronize the copy of os_i to S and free memory of os_i
      $M^- \leftarrow M^- + \mathrm{B}_{\text{os}}(os_i)$
    end while
    Append o_i to S
  end for
  return S

4.4 Adaptive offloading

Although optimizer states such as momentum and variance used in Adam optimizer [14] are only needed for parameter updates and are not required during the forward and backward passes, they occupy a significant amount of memory throughout training. Existing frameworks, such as DeepSpeed, provide functionality to offload these states to CPU memory. In fully sharded approaches like ZeRO-3, both model parameters and optimizer states are partitioned across GPUs. This design reduces the memory footprint on each device and enables parallel data transfers between GPUs and CPU memory. As a result, optimizer state offloading is often used together with fully sharded training.

However, offloading and reloading optimizer states can typically significant overhead. To address this challenge, we propose an approach called *adaptive offloading*, which improves upon existing offloading mechanisms in two ways. First, by monitoring actual memory usage during training, it minimizes the amount of data that needs to be offloaded. Only the portion of optimizer states that would exceed the memory limit is transferred to the host. Second, it takes advantage of the characteristic memory usage patterns in training. Since memory usage tends to increase during the

forward pass and decrease during the backward pass, adaptive offloading schedules data transfers to overlap with computation. This overlap reduces the performance impact of moving data between GPU and CPU memory.

Algorithm 2 presents the pseudo-code for the forward pass with adaptive offloading of optimizer states. The optimizer states are divided into a large enough number of fragments. At the start of the forward pass, the algorithm schedules operators to initiate asynchronous offloading of fragments that exceed the memory limit. Since the copy operation is asynchronous, computation can run in parallel. The memory must only be freed after the copy operation completes. Therefore, the peak memory usage before each operator is checked based on profiling when scheduling. If the memory usage exceeds the limit, the algorithm adds operators to synchronize the copy and free memory for the fragments.

We also iterate over the operators in the backward pass schedule, checking the available memory. If sufficient memory is available from that operator to the end, we schedule a reload operation to transfer the optimizer states back to the GPU. This approach enables overlapping the backward computation with the data transfer from host memory to GPU memory while adhering to memory constraints. Since memory usage often increases during forward passes and decreases during backward passes, this strategy allows for efficient overlap of computation and offloading/reloading operations.

5 Evaluation

We conducted experiments to evaluate the performance of DeepCompile across multiple dimensions, including computational efficiency, memory utilization, and correctness.

5.1 Experimental settings

The experiments were performed on two or four servers, each equipped with eight NVIDIA H100 GPUs (80GB memory per GPU) and 1TB of system memory. The GPUs within each node were interconnected via NVLink, and each server included eight network interfaces (HCAs) connected via InfiniBand. The nodes were interconnected using InfiniBand with an observed bandwidth of 340GB/s for both allgather and reduce-scatter operations. Each server was equipped with two Intel(R) Xeon(R) Platinum 8462Y+ processors.

In our experiments, we used the Llama-3 70B and Mixtral 8x7B models (47 billion parameters)[13] as our target workloads. These models are representative examples of dense and Mixture-of-Experts (MoE) architectures[7, 15, 24], respectively. Both were run using bfloat16 with mixed precision [18], while maintaining FP32 copies of parameters, gradients, and Adam optimizer states. We also employed activation checkpointing, integrated into the model implementation at the Transformer layer level, where each layer is recomputed as a block.

We used PyTorch v2.6.0 and models from the HuggingFace Model Hub without any built-in parallelization support. The experiments were conducted with Python 3.10 and CUDA 12.4. DeepSpeed v1.6.4 and PyTorch FSDP were used as the baseline frameworks, and DeepCompile was built on top of DeepSpeed. The optimization passes and profilers in DeepCompile were implemented in Python (approximately 2.6K lines), while the custom operators injected into the computational graph were implemented in C++ (approximately 900 lines).

While some distributed training strategies support partial sharding—for example, sharding only optimizer states, or both gradients and optimizer states while keeping parameters unsharded—these configurations are not sufficient for our target models. The total parameter sizes of Llama-3 70B and Mixtral 8x7B exceed the memory capacity of a single GPU (140 GB and 94 GB, respectively), requiring all parameters, gradients, and optimizer states to be fully sharded across devices. For this reason, we use the fully sharded configurations of both DeepSpeed ZeRO-3 and PyTorch FSDP in our evaluation.

Other large-scale training frameworks, such as Megatron-LM, are not included in our comparison. Although Megatron-LM has been widely adopted for large model training, it requires a specialized implementation where parallelism strategies are tightly coupled with model code. In contrast, we focus on model-agnostic approaches that can be applied to standard implementations. Similarly, frameworks that automate parallelization, such as Alpa [19], FlexFlow [12], and Unity [29], are not included, as they remain experimental and do not currently support the models used in this evaluation.

5.2 Efficiency

Figure 7 presents the throughputs observed in our experiments. We used Llama-3-70B with 32 GPUs and Mixtral-8x7B with 16 GPUs. We evaluated DeepSpeed ZeRO-3 both without and with the PyTorch compiler enabled, labeled as `ZeRO3` and `ZeRO3 (C)`, respectively. Similarly, we include results for PyTorch FSDP, labeled as `FSDP` and `FSDP (C)` depending on whether the PyTorch compiler is enabled. For DeepCompile, we tested three configurations: enabling only proactive prefetching, enabling only selective unsharding, and enabling both in that order. These configurations are labeled as `DeepCompile (P)`, `DeepCompile (S)`, and `DeepCompile (P+S)`, respectively, in the charts. Since the models fit within the available GPU memory at these scales, adaptive offloading was not applied in these experiments.

We conducted this experiment using different batch sizes and sequence lengths. In addition, we evaluated training performance with gradient accumulation.

DeepCompile consistently outperformed all baseline methods, with the most significant gains observed when both proactive prefetching and selective unsharding were enabled (`DeepCompile (P+S)`). Under the setting with a gradient accumulation step of 1, the smallest improvement over ZeRO-3 was 1.11×, observed in the configuration with sequence length 512 and batch size 1. In this setting, computation is significantly faster relative to communication, leaving little opportunity for our optimizations to hide communication overhead through overlap. As a result, the benefit of prefetching and unsharding becomes limited. For all other conditions, DeepCompile achieved approximately 1.14× to 1.18× speedup over ZeRO-3, and 1.16× to 1.26× over FSDP in the Llama-3-70B experiments. For Mixtral-8x7B, the speedups over ZeRO-3 ranged from 1.07× to 1.35×, while those over FSDP ranged from 1.06× to 1.14×. Mixtral-8x7B is an MoE model, where only a subset of parameters is activated for each input token. Due to its relatively small computation load compared to total parameter size, it suffers more from communication bottlenecks. As a result, similar to Llama-3-70B, the improvements were slightly larger when the batch size and sequence length were increased, enabling better overlap between communication and computation.

Figure 7(a)(iii) and (b)(iii) show throughput results when varying the gradient accumulation steps from 1 to 16. As discussed in Section 4.3, this setting significantly benefits from selective unsharding. In the Llama-3-70B experiment (Fig. 7(a)(iii)), DeepCompile achieves increasingly higher throughput as the accumulation step increases, with up to 1.28× improvement over ZeRO-3 at step 16. In Mixtral-8x7B (Fig. 7(b)(iii)), the improvement is even more pronounced due to the model's heavier communication load. DeepCompile achieves up to 1.54× higher throughput compared to ZeRO-3 at step 16.

In contrast, FSDP fails to run with accumulation steps greater than 1, as it does not support accumulating gradients while keeping them partitioned. Instead, it gathers full gradients after each backward pass, which causes the total memory usage to exceed GPU capacity, leading to memory allocation failures.

5.3 Memory Utilization

The optimization passes described above allow DeepCompile to utilize GPU memory more effectively, contributing to increased throughput. We compare the memory utilization of DeepCompile against other baselines. Figure 8 presents the peak GPU memory usage for different models across various sequence lengths.

The memory footprints of ZeRO-3 and FSDP are approximately 40GB for Llama-3-70B and 30GB for Mixtral-8x7B. When selective unsharding is enabled, DeepCompile actively utilizes available memory to keep more parameters unsharded. We configure a safety margin by first reserving approximately 7GB for the CUDA driver, NCCL buffers, and other runtime components, and then applying an additional 10% margin to the remaining memory. As a result, `DeepCompile (S)` and `DeepCompile (P+S)` consistently use around 65GB in all cases. This demonstrates that the selective unsharding mechanism dynamically adapts the amount of unsharded parameters based on the available memory at runtime.

5.4 Adaptive Offloading

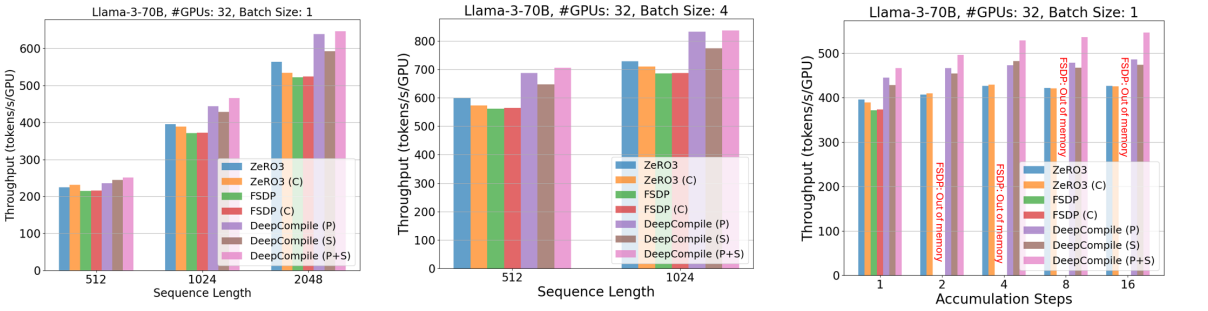
The Llama-3-70B model requires 32 or more GPUs, each with 80GB of memory, to fit entirely into GPU memory under our current settings. However, by offloading optimizer states, we are able to fit the model onto just 16 GPUs. To evaluate the benefits of adaptive offloading of optimizer states, as described in Section 4.4, we compared the throughput achieved using DeepSpeed's offloading feature with that of our adaptive offloading approach.

Figure 9 shows the iteration times for each method. `ZeRO3 (Offload optimizer)` and `ZeRO3 (Offload optimizer) + Compile` represent the results of using DeepSpeed ZeRO-3 with and without the PyTorch compiler, respectively. Notably, DeepSpeed not only offloads optimizer states to host memory but also performs parameter updates on the CPU. While this approach saves more memory than DeepCompile's offloading, it significantly slows down parameter updates. Since this behavior makes it difficult to compare DeepSpeed's offloading directly with our method—which keeps parameter updates on the GPU—we also included results for `DeepCompile (Offload optimizer all + sync)`, which offloads all optimizer states at the beginning of the forward pass and reloads them synchronously at the end. Our adaptive offloading method is shown as `DeepCompile (Offload selective + async)`, which minimizes the amount of data offloaded and overlaps offloading and reloading asynchronously with computation.

The experimental results in Figure 9 indicate that adaptive offloading (`DeepCompile (Offload selective + async)`) achieves up to 7.0× and 6.9× higher throughput than `ZeRO3 (Offload optimizer)` and `ZeRO3 (Offload optimizer) + Compile`, respectively, when the batch size is 1 and the sequence length is 1024. It also demonstrates a 2.6× improvement over `DeepCompile (Sync)`.

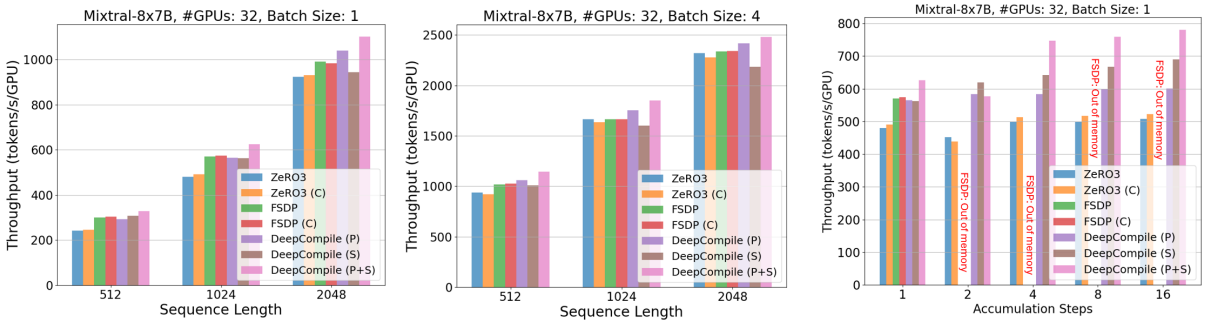
5.5 Compilation time

As shown in Fig.3, DeepCompile applies a sequence of optimization passes. In our experiments, we first executed the



(i) Batch size: 1, gradient accumulation step: 1 (ii) Batch size: 4, gradient accumulation step: 1 (iii) Batch size: 1, gradient accumulation step: 1-16, sequence length: 1024

(a) Llama-3-70B (#GPUs: 32)



(i) Batch size: 1, gradient accumulation step: 1 (ii) Batch size: 4, gradient accumulation step: 1 (iii) Batch size: 1, gradient accumulation step: 1-16, sequence length: 1024

(b) Mixtral 8x7B (#GPUs: 32)

Figure 7: Throughputs resulting from Llama-3 70B and Mixtral 8x7B models

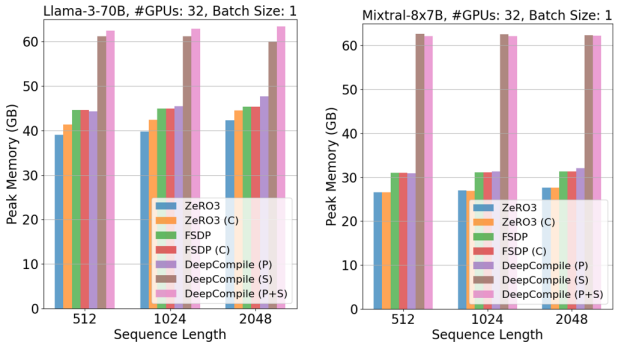


Figure 8: Memory utilization from Llama-3 70B and Mixtral 8x7B models

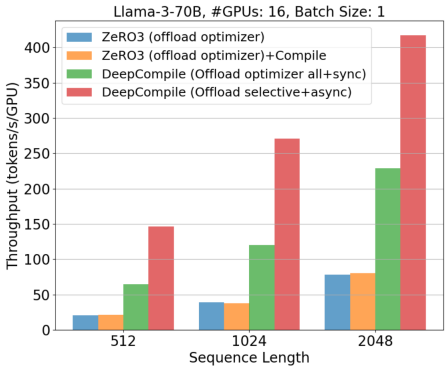


Figure 9: Throughput results for adaptive offloading

pass that inserts allgather and release operations. After a short warmup period (five training iterations), we applied additional passes including proactive prefetching and selective unsharding.

These passes involve graph analysis and profiling, resulting in several minutes of overhead for our target models. Table 2 reports the time required for each configuration, measured with batch size 1 and sequence length 512.

The variation in compile time across optimization configurations for the same model is less than 10%, indicating

Table 2: Time for compilation

	Proactive prefetching	Selective unsharding	Both
Llama-3 70B	254.8 s	248.6 s	266.9 s
Mixtral 8x7B	437.0 s	416.5 s	407.8 s

that the additional cost of enabling proactive prefetching or selective unsharding is relatively minor compared to the base analysis and scheduling pass. On the other hand, compile times vary more significantly across models, as seen in the longer durations for Mixtral-8x7B compared to Llama-3-70B. This difference stems from model-specific factors such as graph size and layer composition.

It is important to note that this compilation overhead occurs only once before training begins. During actual training, the compiled graph is reused without incurring any additional overhead. Given that model training typically consists of hundreds to thousands of iterations, the one-time compilation cost is negligible in practice.

Furthermore, the results of the optimization passes can be cached and reused across training runs as long as key settings such as batch size and sequence length remain unchanged. This reuse is particularly valuable when sweeping hyperparameters such as learning rates or dropout ratios, enabling users to avoid repeating the initial compilation and further reducing end-to-end overhead.

5.6 Correctness

To verify the correctness of the optimization passes in DeepCompile, we compared the resulting loss values with those of DeepSpeed ZeRO-3. We initialized Llama-3 70B model with random weights but the same random seed in both settings. We used the AG News corpus [35] as training examples, with the sequence length and micro batch size set to 512 and 1, respectively. The learning rate was set to 1.5e-5. Figure 10 shows the training losses for both settings. Although some operators are non-deterministic and introduce subtle differences, the loss curves were closely aligned.

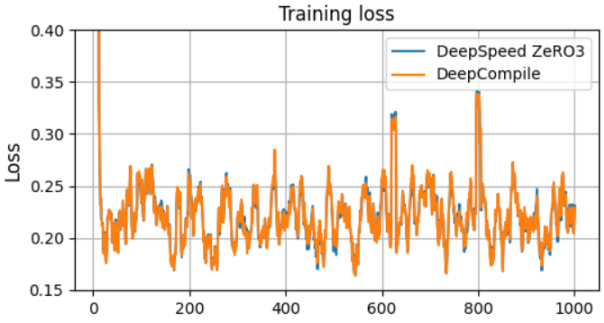


Figure 10: Verification of loss values

6 Related Work

6.1 Distributed Training Framework

As discussed in Section 2, various distributed training strategies have been proposed to scale deep learning models beyond the capacity of a single GPU. For instance, tensor parallelism, as introduced in Megatron-LM [25], splits large matrix operations across devices, while pipeline parallelism, as implemented in GPipe [11], assigns consecutive layers to different GPUs. While effective, both approaches require the user to manually modify the model to specify how tensors or layers are partitioned. This makes it challenging to adapt these strategies to diverse or rapidly evolving model architectures.

In contrast, our work focuses on the fully sharded approach, as implemented in ZeRO [22] and FSDP [27], which requires no model code changes and is applicable to a wide range of architectures. DeepCompile builds on this fully sharded approach and introduces compiler-level optimizations to improve its efficiency.

Note that DeepCompile is not inherently limited to the fully sharded approach. Its design is compatible with other parallelization strategies. For example, RaNNC [26] automatically partitions the computation graph to enable pipeline parallelism, and Alpa [19] selects distributed execution rules based on the partitioned state of tensors. Similarly, DeepCompile could support automated parallelization by replacing computation operators in the graph and integrating this with other optimization passes within a unified framework.

6.2 Optimizing Parallelization Strategies

To improve training efficiency for large-scale models, various strategies such as data, tensor, and pipeline parallelism have been studied. Recent work has focused on automatically discovering optimal combinations of these strategies, including Alpa [19], FlexFlow [12], Unity [29], and Galvatron [17]. Dynamic switching techniques have also been proposed, such as HotSpa [8], which aims to adaptively reconfigure parallelization strategies during training.

However, these approaches generally assume that the model is already implemented in a way that supports the desired forms of parallelism. In practice, tensor and pipeline parallelism often require model-specific partitioning logic and integration with low-level APIs. Supporting multiple strategies and enabling dynamic switching can involve considerable engineering effort, particularly for complex or evolving model architectures. As a result, despite their promise, such

automated parallelization systems have seen limited adoption in current practice.

6.3 Computation-Communication Overlapping

As discussed earlier, widely used frameworks such as ZeRO-3 and FSDP incorporate prefetching techniques to overlap communication with computation and reduce latency from all-gather operations. Prefetching has become a standard optimization for improving efficiency in fully sharded training.

More recently, techniques for overlapping communication at a finer granularity, such as intra-layer parallelism, have also been proposed [31, 32]. These approaches aim to eliminate or hide communication costs by restructuring computation within layers. In contrast, our work focuses on profiling-guided, graph-level optimization of communication overlap.

Like DeepCompile, SimpleFSDP [33] targets compiler-level optimization of the fully sharded approach. It performs memory-aware fusion of all-gather operations to reduce communication overhead, but does not take into account dynamic changes in memory usage when deciding how early each all-gather can be issued without exceeding memory constraints. Moreover, SimpleFSDP focuses solely on scheduling communication for FSDP. In contrast, DeepCompile is designed to support a broader range of optimizations and enables techniques such as selective unsharding and adaptive offloading.

7 Conclusion

We introduced DeepCompile, a compiler-based framework for optimizing distributed training of large-scale deep learning models. DeepCompile starts from a standard single-GPU computation graph and incrementally inserts distributed operators such as all-gather and release, enabling graph-level analysis and holistic optimization of distributed training. Unlike existing frameworks like ZeRO-3 and FSDP, which rely on heuristics and fixed module boundaries, DeepCompile leverages runtime profiling to flexibly reorder and coordinate operations based on actual memory usage and data dependencies.

Built on this foundation, DeepCompile implements a fully sharded training approach along with three key optimizations: proactive prefetching to improve communication–computation overlap, selective unsharding to reduce redundant communication, and adaptive offloading to dynamically manage memory under tight resource constraints. These optimization passes are applied in sequence, informed by profiling feedback, allowing DeepCompile to coordinate them effectively in a unified compilation framework.

Our experiments demonstrate that DeepCompile consistently outperforms ZeRO-3 and FSDP baselines across a range of configurations. It achieves up to 1.28× and 1.54× higher throughput for Llama-3 70B and Mixtral 8×7B models, respectively. Moreover, in scenarios with limited GPU memory, DeepCompile's adaptive offloading achieves up to a 7.01× throughput improvement by automatically overlapping communication and reducing memory pressure.

DeepCompile represents a step forward in compiler-driven distributed training. Its graph transformation capabilities and runtime-aware optimization passes offer a scalable and automated path toward efficient parallelization. In future work, we aim to extend DeepCompile to support broader classes of optimizations, such as fully automated parallelization planning and enhanced memory scheduling.

References

[1] Marah Abdin, Jyoti Aneja, Hany Awadalla, et al. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. *arXiv preprint arXiv:2404.14219* (2024).

[2] Jason Ansel, Edward Yang, Horace He, et al. 2024. 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 (ASPLOS '24)*. 929–947.

[3] Tom Brown, Benjamin Mann, Nick Ryder, et al. 2020. Language models are few-shot learners. *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS '20)* 33, Article 159 (2020), 1877–1901 pages.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*. 579–594.

[5] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, et al. 2022. PaLM: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4171–4186.

[7] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research (JMLR)* 23, 120 (2022), 1–39.

[8] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. 2024. Enabling Parallelism Hot Switching for Efficient Training of Large Language Models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. 178–194.

[9] Google. 2023. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla.

[10] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).

[11] Yanping Huang, Yanan Cheng, Ankur Bapna, et al. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Proceedings of the 33rd International Conference on Neural Information*

Processing Systems, Vol. 32. 103–112.

[12] Zhihao Jia, Matei Zaharia Lin, and Alex Aiken. 2018. Beyond data and model parallelism for deep neural networks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 485–500.

[13] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, et al. 2024. Mixtral of Experts. *arXiv preprint arXiv:2401.04088* (2024).

[14] Diederik P Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980* (2014).

[15] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, et al. 2021. GShard: Scaling giant models with conditional computation and automatic sharding. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[16] Aixin Liu, Bei Feng, et al. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. *arXiv preprint arXiv:2405.04434* (2024).

[17] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2022. Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism. 16, 3 (2022), 470–479.

[18] Paulius Micikevicius, Sharan Narang, Jonah Alben, et al. 2017. Mixed Precision Training. http://arxiv.org/abs/1710.03740

[19] Zhihao Mo, Zhuang Zhao, Yuze Zhu, Ajay Jain, Tushar Yu, Shaoduo Shi, Yida Yang, Hang Zhang, Qimin Ho, Tianqi Chen, et al. 2022. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *Proceedings of 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.

[20] Sharan Narang, Aakanksha Chowdhery, Jacob Devlin, Gaurav Mishra, and Adam Roberts. 2021. Do Transformer Modifications Transfer Across Implementations and Applications?. In *Findings of the Association for Computational Linguistics (ACL-IJCNLP 2021)*. 2051–2061.

[21] Adam Paszke, Sam Gross, Francisco Massa, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. 8024–8035.

[22] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. 1–16.

[23] Noam Shazeer. 2019. Fast Transformer Decoding: One Write-Head is All You Need. In *Proceedings of the NeurIPS Workshop on Machine Learning and the Physical Sciences*.

[24] Noam Shazeer, Azalia Mirhoseini, Piotr Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations (ICLR)*.

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

[26] Masahiro Tanaka, Kenjiro Taura, Toshihiro Hanawa, and Kentaro Torisawa. 2021. Automatic Graph Partitioning for Very Large-scale Deep Learning. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1004–1013.

[27] PyTorch Team. 2023. Fully Sharded Data Parallel (FSDP). https://pytorch.org/docs/stable/fsdp.html.

[28] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Naman Goyal, Siddhartha Batra, Piotr Bojanowski, Armand Joulin, Edouard Grave, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288* (2023).

[29] Tobias Unger, Hang Zhang, Tushar Yu, Zhuang Zhao, Zhihao Mo, Yujing He, Chengcheng Peng, Alex Aiken, Matei Zaharia, and Zhihao Jia. 2022. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 579–596.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. *Advances in Neural Information Processing Systems* 30 (2017), 5998–6008.

[31] Guanhua Wang, Chengming Zhang, Zheyu Shen, Ang Li, and Olatunji Ruwase. 2024. Domino: Eliminating Communication in LLM Training via Generic Tensor Slicing and Overlapping. arXiv:2409.15241

[32] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, et al. 2022. Overlap Communication with Dependent Computation via Decomposition in Large Deep Learning Models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023)*. 93–106.

[33] Ruisi Zhang, Tianyu Liu, Will Feng, Andrew Gu, Sanket Purandare, Wanchao Liang, and Francisco Massa. 2024. SimpleFSDP: Simpler Fully Sharded Data Parallel with torch.compile. arXiv:2411.00284 [cs.DC] https://arxiv.org/abs/2411.00284

[34] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Seung Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[35] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2016. Character-level Convolutional Networks for Text Classification. arXiv:1509.01626