Cephalo: Harnessing Heterogeneous GPU Clusters for Training Transformer Models

Runsheng Benson Guo
University of Waterloo
Cheriton School of Computer Science
Waterloo, Canada
r9guo@uwaterloo.ca

Utkarsh Anand
University of Waterloo
Cheriton School of Computer Science
Waterloo, Canada
utkarsh.anand@uwaterloo.ca

Arthur Chen
University of Waterloo
Cheriton School of Computer Science
Waterloo, Canada
a267chen@uwaterloo.ca

Khuzaima Daudjee
University of Waterloo
Cheriton School of Computer Science
Waterloo, Canada
khuzaima.daudjee@uwaterloo.ca

Abstract

Training transformer models requires substantial GPU compute and memory resources. While training systems are typically designed for homogeneous GPU clusters, sufficiently large homogeneous clusters are difficult to acquire for most organizations due to cost and GPU scarcity. Hence, it is increasingly common to assemble heterogeneous clusters with a mix of higher and lower-end GPUs featuring differing compute power and memory capacity. Existing methods attempt to distribute the workload across heterogeneous GPUs based on compute capacity but often underutilize compute due to memory constraints. We present Cephalo, a system that holistically balances both compute and memory usage by decoupling compute distribution from training state assignment. Cephalo uses an optimizer to efficiently distribute the compute workload and storage of training state to account for GPU heterogeneity in the cluster. Additionally, it separates memory from compute requirements through an optimized gradient accumulation strategy. Compared to state-of-the-art methods, Cephalo achieves 1.2×–10.8× higher training throughput while supporting larger models and batch sizes.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3730418

CCS Concepts

• **Computing methodologies** → **Distributed computing methodologies**; **Machine learning algorithms**.

Keywords

Distributed Training, Transformers, Heterogeneous Clusters

ACM Reference Format:
Runsheng Benson Guo, Utkarsh Anand, Arthur Chen, and Khuzaima Daudjee. 2025. Cephalo: Harnessing Heterogeneous GPU Clusters for Training Transformer Models. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3730418

Transformer models [52] have demonstrated state-of-the-art performance in many domains including natural language processing (NLP), computer vision, and recommendation systems [8, 9, 49]. In particular, large language models (LLMs), which are based on the transformer architecture, have significantly advanced NLP tasks such as question-answering, translation, and summarization [5, 8, 62]. Since increasing model size can yield significant improvements in accuracy, this has led to the development of larger models that often exceed modern GPU compute and memory capabilities [37].

Consequently, many strategies have been proposed to distribute and parallelize training across multiple GPUs. *Data parallelism* replicates the model across GPUs, each training on a different subset of the inputs in parallel. *Model parallelism* splits the model across GPUs, with each GPU storing and processing only a partition of the model's parameters.

While existing parallelization strategies typically assume GPU homogeneity, ML practitioners, in reality, often do not have sufficiently large homogeneous clusters for training transformers, even at smaller scales of fewer than 32 GPUs [10, 17, 36, 47, 64]. For example, a small-scale company or

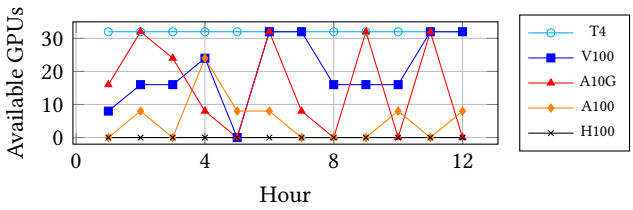

Figure 1: AWS GPU availability (12-hour period).

research lab may not have the resources to purchase an entire cluster of the latest GPUs. Instead, they are more likely to accumulate a diverse array of GPUs with varying compute and memory capacities over time [25, 51, 60]. Cloud platforms like AWS offer VMs with a variety of GPU models, but due to high demand, each model is available only in limited quantities. Figure 1 plots a trace of GPU availability on AWS over a 12-hour period in the us-west region. High-end GPUs (A100, H100) are almost always unavailable, and even mid-tier GPUs (A10G, V100, T4) are limited due to capacity and quota. Thus, it is challenging to reserve a large homogeneous cluster of GPUs.

By assembling *heterogeneous* clusters with different GPU models, users can leverage a larger pool of compute resources for training. However, existing systems are unable to utilize resources efficiently in heterogeneous clusters. Systems for homogeneous clusters divide compute and memory demands evenly among all GPUs [27, 39, 44]. In clusters with varying GPU capabilities, training is bottlenecked by the slowest GPU, leaving faster GPUs idle. Additionally, training fails if GPUs with the least memory run out, even if others have unutilized memory.

Heterogeneity-aware training methods have been proposed, which aim to balance computational load across GPUs. For instance, in data parallelism, the batch of inputs is distributed unevenly across GPUs according to their relative computational speeds [16, 20, 29]. Systems using model parallelism partition the model's layers or parameters unevenly across GPUs to balance computation [30, 64]. Recent methods integrate both data and model parallelism to further optimize compute distribution [51, 60].

These load-balancing techniques allocate memory on each GPU proportional to its computational capacity. In data parallelism, a GPU assigned a larger batch of inputs requires more memory for operations and activation storage. Similarly, in model parallelism, a GPU handling a larger model shard demands additional memory to maintain the training state. However, as shown in Figure 2, a GPU's memory capacity does not always scale with its compute speed. This mismatch can prevent effective computational load balancing due to memory limitations. For example, while the L4 offers significantly faster computation than the P40, both have the same memory capacity, meaning the L4 may lack sufficient memory to handle twice the workload of a P40.

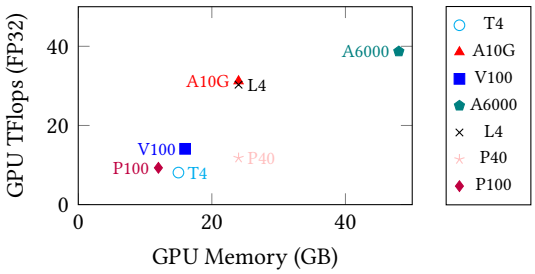

Figure 2: GPU TFlops (FP32) vs. Memory Capacity.

Thus, existing systems are susceptible to both: (i) underutilizing compute on GPUs with low memory capacity relative to compute speed, and (ii) underutilizing memory on GPUs with high memory capacity relative to compute speed.

To address these shortcomings, we designed Cephalo, a system capable of effectively utilizing the aggregate compute ***and*** memory resources in heterogeneous GPU clusters when training transformer models.

Cephalo partitions the global batch of training inputs unevenly across GPUs to control the computational workload assigned to each GPU. To control the memory utilization on each GPU, Cephalo combines the following strategies:

(i) The training state (parameters, gradients, and optimizer state) is sharded across the GPUs to balance memory utilization. Each GPU can store anywhere from none of the training state to the entire training state. Flexibly sharding the training state is implemented on top of *Fully Sharded Data Parallelism* (FSDP) [65] that evenly distributes the training state across GPUs.
(ii) Gradients can be accumulated over multiple smaller batches to replicate training on larger batch sizes while using less memory for compute operations.
(iii) Memory for storing intermediate activation and gradient values are eliminated with a combination of activation recomputation and offloading intermediate values to CPU when they are not used.

These mechanisms for controlling computational workload and memory can be applied independently. This allows Cephalo to decouple the assignment of compute and memory to each GPU and fully utilize the aggregate GPU compute and memory available within a heterogeneous cluster of GPUs in scenarios where state-of-the-art systems fall short. In this paper, we make the following contributions:

(1) We designed and implemented Cephalo, a system for training transformer models on heterogeneous GPU clusters that jointly optimizes compute and memory

distribution to maximize training throughput by efficiently utilizing resources across GPUs. Cephalo includes an optimizer to divide training data, manage training state, and configure gradient accumulation to accommodate resource heterogeneity.

(2) We integrate gradient accumulation along with activation and gradient offloading efficiently in FSDP. Our implementation of gradient accumulation minimizes the overhead of gathering training state. Our offloading strategy mitigates memory overhead from gradient accumulation and overlaps with compute to hide transfer latency.

(3) We perform an extensive evaluation of Cephalo on heterogeneous GPU clusters with up to 64 GPUs and on transformer models with up to 7 billion parameters. We show that Cephalo is able to achieve up to 10× higher training throughput than comparative state-of-the-art heterogeneous training systems while supporting training for larger models and batch sizes.

1 Background

This section describes how transformers are trained, the motivation for training on heterogeneous clusters, and how prior work parallelizes training across homogeneous versus heterogeneous clusters.

1.1 Training Transformers

Transformer models consists of a sequence of identical encoder and decoder layers, containing computationally expensive self-attention mechanisms and feed-forward networks [52]. These models are typically trained with stochastic gradient descent (SGD) or its variants such as Adam [21]. This process involves iteratively adjusting the model's parameters to minimize the loss, a measure of the difference between the model's predictions and the actual data. During each iteration, the forward pass computes the loss of the model, and during the backwards pass the gradients of the model are computed from the loss. These gradients are then used to update the model's parameters. Adam requires 16 bytes of memory per model parameter on the GPU [39, 46], covering not only the model parameters but also their gradients and optimizer state. Besides maintaining the training state, GPU memory is also required to run operations and store intermediate activation outputs. Even a medium-sized model like Llama 7B [50] requires more memory for training than available on high-end H100 GPUs. Given these substantial GPU memory and computational requirements, transformer training is typically parallelized.

Data Parallelism [42] replicates the model across GPUs, each computing a gradient on a local batch of data of size b. Then, the local gradients are averaged across N GPUs to compute the gradient as $\frac{1}{N}\sum_{i=1}^{N}\frac{1}{b}\sum_{j=1}^{b}\nabla_{ij}$, where ∇_{ij} is the gradient from the jth data input of the ith GPU. This "vanilla" data parallelism works only if each GPU can store the entire training state. ZeRO-3 [39], or fully sharded data parallelism (FSDP) [65] in PyTorch [23], is a variant of data parallelism that evenly shards the training state across GPUs. This allows for larger models to be trained by reducing the training state stored per GPU by a factor of N, albeit at the cost of 50% more communication [39].

Model Parallelism partitions a model across GPUs, with each GPU storing only the training state for its assigned shard, enabling the training of models larger than a single GPU's memory. Pipeline parallelism [15, 31] divides the model into stages of consecutive layers, passing activations and gradients between stages. It parallelizes compute by processing microbatches in a pipeline across these stages. Tensor parallelism [43, 44] partitions inputs, computation, and parameters across GPUs, using *AllReduce* [35] to aggregate partial outputs after each layer.

1.2 Training on Heterogeneous Clusters

Increasing Ubiquity of Heterogeneous Clusters. Many distributed training systems assume a homogeneous GPU cluster, dividing compute and memory demands equally. However, most organizations lack large homogeneous clusters due to frequent GPU release cycles, high upgrade costs, GPU shortages, and limited cloud availability [10, 25, 47, 48, 56]. As a result, organizations often assemble clusters with GPUs from different generations to achieve higher compute power in aggregate. Thus, training on heterogeneous clusters has gained attention as it allows organizations to leverage all available GPU resources for training [16, 36, 51, 60, 64].

Training Adaptations for Heterogeneous Clusters. Existing homogeneous training strategies are susceptible to underutilizing GPU resources on a heterogeneous cluster since faster GPUs will be idle while waiting to synchronize with slower GPUs. Systems like Whale [16, 29] propose to mitigate bottlenecks in data parallelism by assigning uneven batch sizes to GPUs based on their relative compute speed. However, a GPU with a high compute-to-memory ratio may run out of memory before fully utilizing its compute.

In pipeline parallelism [31, 36], balancing compute latency across stages is crucial as the slowest stage bottlenecks the pipeline. In homogeneous clusters, dividing the layers evenly across stages is effective since transformer layers are typically identical [32]. In heterogeneous clusters, compute can be balanced by partitioning layers based on relative compute speed of its GPUs. However, such a partition may not be possible due to memory constraints. The fastest GPUs may lack sufficient memory to handle the layers required to maximize their compute potential, while slower GPUs

might fully utilize their compute capacity but leave a significant portion of their memory underutilized. HAP [64] distributes workloads unevenly using data and tensor parallelism to align with GPU compute capacities, though it still assumes faster GPUs have more memory. Additionally, tensor parallelism requires high-bandwidth GPU interconnects for efficiency, which are unlikely to be available in heterogeneous clusters with lower-end GPUs. Metis [51] and HexiScale [60] integrate heterogeneous data, pipeline, and tensor (3D) parallelism, offering greater flexibility for heterogeneous training configurations but inherit limitations of each parallelism type.

In existing data and model parallelism approaches, compute and memory allocation are tightly coupled, which becomes problematic in heterogeneous clusters since a GPU's memory capacity does not always match its compute speed (Figure 2). This mismatch often prevents effective compute balancing due to memory limitations. Cephalo solves these problems by independently balancing compute and memory during training in heterogeneous clusters. Cephalo targets the training of medium sized models, such as Llama and Phi, which offer competitive performance comparable to larger models [1, 41, 63]. These models are feasible to train on moderately sized heterogeneous clusters, making them attractive options for organizations that seek high-performance models without large, high-end homogeneous clusters.

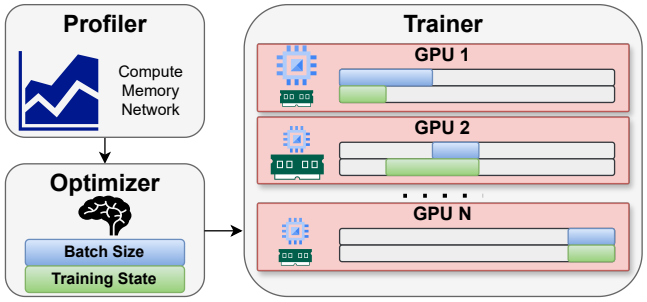

Figure 3: Architecture of Cephalo.

2 Cephalo Design

Cephalo is designed to maximize training throughput by effectively balancing computational and memory loads across heterogeneous GPUs, ensuring full utilization of the aggregate resources available in the cluster.

Cephalo is built on top of FSDP [65], which divides the training state and computation evenly across each GPU. To balance compute, Cephalo assigns a batch size to each GPU proportional to its compute speed. To balance memory utilization, Cephalo partitions the training state and decides on configurations for gradient accumulation, activation checkpointing, and activation/gradient offloading according to the relative memory capacities of each GPU. Given a model and target cluster, Cephalo profiles the model to build performance models predicting computation time, memory usage, and communication time across configurations. The optimizer then leverages these models to configure batch size, training state shard, and gradient accumulation for each GPU to maximize training throughput. Figure 3 illustrates Cephalo's architecture.

2.1 Division of Compute and Training State

A key feature of Cephalo is its ability to decouple the distribution of compute and memory loads across GPUs, essential for optimizing performance in heterogeneous clusters where GPU memory capacity does not necessarily scale with compute power. Cephalo efficiently allocates compute and training state across GPUs, leveraging the combined compute and memory resources of the cluster. We describe the mechanisms Cephalo uses for this division next.

Compute Partitioning. Given a global batch size B, Cephalo partitions the workload across GPUs by assigning each GPU i a local batch size b_i such that $\sum_i b_i = B$. To minimize iteration times, Cephalo balances b_i to reduce the maximum runtime on any GPU. To maintain equivalency with standard training, each GPU's local gradient is adjusted by $N \cdot b_i/B$, resulting in a final gradient of:

$$\nabla = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{N \cdot b_i}{B}\right)\frac{1}{b_i}\sum_{j=1}^{b_i}\nabla_{ij} = \frac{1}{B}\sum_{i=1}^{N}\sum_{j=1}^{b_i}\nabla_{ij} \quad (1)$$

Training State Partitioning. The training state – comprising model parameters, gradients, and optimizer states – consumes significant memory. FSDP partitions this state evenly across GPUs to reduce memory utilization. The model is divided into FSDP units which manage the computation and communication for their respective parameters. As in prior work [38], each unit in Cephalo contains a single transformer layer. Each unit shards its portion of the training state evenly. During the forward and backward passes for an FSDP unit, an *AllGather* collective gathers the full set of parameters on each GPU. Parameters are subsequently resharded to ensure only the current and prefetched next units are materialized in memory. After a unit's backward pass, a *ReduceScatter* collective averages gradients and distributes them to the responsible GPUs. FSDP overlaps *AllGather* and *ReduceScatter* collectives with the computation of adjacent units to hide communication latencies. In contrast to a fixed even partitioning, Cephalo assigns each GPU i a specific training state ratio r_i (where $\sum r_i = 1$), enabling fine-grained memory control independent of the compute distribution.

2.2 Managing Memory for Compute

Beyond storing training state, significant GPU memory is needed for computation and storing intermediate activations.

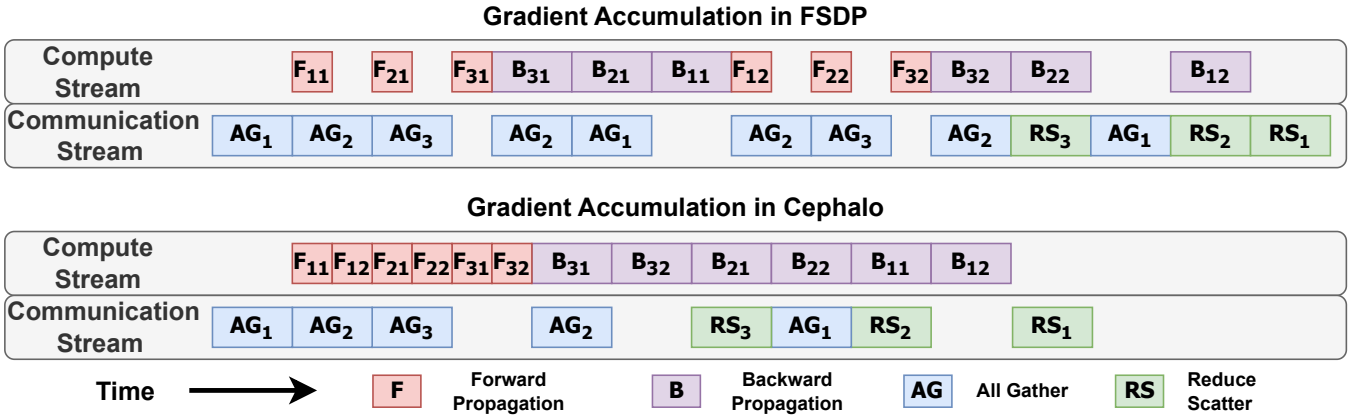

Figure 4: Gradient accumulation in FSDP (top) vs Cephalo (bottom) when training with 2 microbatches on a model consisting of 3 FSDP units. F_{ij} and B_{ij} are the forwards and backwards passes of the ith FSDP unit on the jth microbatch. AG_i and RS_i are the *AllGather* and *ReduceScatter* collectives for the ith FSDP unit.

We employ gradient accumulation [22, 31] to enable training with larger effective batch sizes while reducing memory usage. Instead of computing gradients for the full batch size b at once, we split b into smaller microbatches of size m and accumulate gradients over ℓ microbatches, where $b = \ell \cdot m$. This approach allows each GPU to process an effective batch size of b while reducing memory demands by managing smaller microbatches. In Cephalo, we develop an optimized implementation of gradient accumulation for FSDP, and configure it to control the amount of memory used for computation.

Layered Gradient Accumulation. Traditional gradient accumulation in FSDP performs the full forward and backward pass for each microbatch sequentially. This requires ℓ times more *AllGather* collectives than training without gradient accumulation because sharded parameters must be gathered for each microbatch. To reduce this overhead, we implement *layered* gradient accumulation [22], which processes *all* microbatches for a layer before moving to the next. Sequentially processing microbatches allows us to gather parameters only once per pass for all microbatches.

Figure 4 illustrates the difference between gradient accumulation in FSDP and Cephalo. Our implementation calls *AllGather* to prefetch the next FSDP unit while the current one is executing. This communication is overlapped with all executing microbatches of the current FSDP unit, effectively hiding the communication overhead even when networking is slow relative to compute. Gradient accumulation can add minor runtime overhead as smaller microbatches may not fully utilize GPU cores, introducing a tradeoff between memory savings and compute efficiency. Unlike prior systems, Cephalo automatically optimizes gradient accumulation with compute and training state partitioning (Section 2.4), balancing this tradeoff effectively. Section 3.4 details implementation optimizations that were necessary to run gradient accumulation efficiently with FSDP.

Activation Checkpointing and Offloading. While layered gradient accumulation reduces communication overhead, it introduces significant memory overhead compared to traditional gradient accumulation. This is because activations must be stored for all microbatches of a layer until the backward pass, whereas traditional gradient accumulation maintains activations for only a single microbatch. For some models, this additional activation storage can exceed the memory savings gained from smaller batch sizes.

Cephalo addresses memory overhead in layered gradient accumulation with a combination of activation checkpointing and offloading. Activation checkpointing saves activations only at layer boundaries during the forward pass [30, 44], allowing intermediate activations to be recomputed in the backward pass, which significantly reduces memory usage. However, even storing boundary activations adds overhead. To mitigate this, Cephalo offloads boundary activations to CPU memory until needed during backwards. Similarly, Cephalo offloads boundary gradients in the backwards pass. Combined, activation checkpointing and offloading eliminate memory overhead when accumulating additional microbatches. Hence, for a fixed microbatch size, Cephalo can increase the number of microbatches without using additional memory. Activations and gradients are offloaded and prefetched between the GPU and CPU asynchronously with compute, hiding transfer overheads. In the forwards pass:

(1) Microbatch activations are offloaded to the CPU while the next microbatch runs.
(2) Input activations for the next microbatch are prefetched while the current microbatch runs.

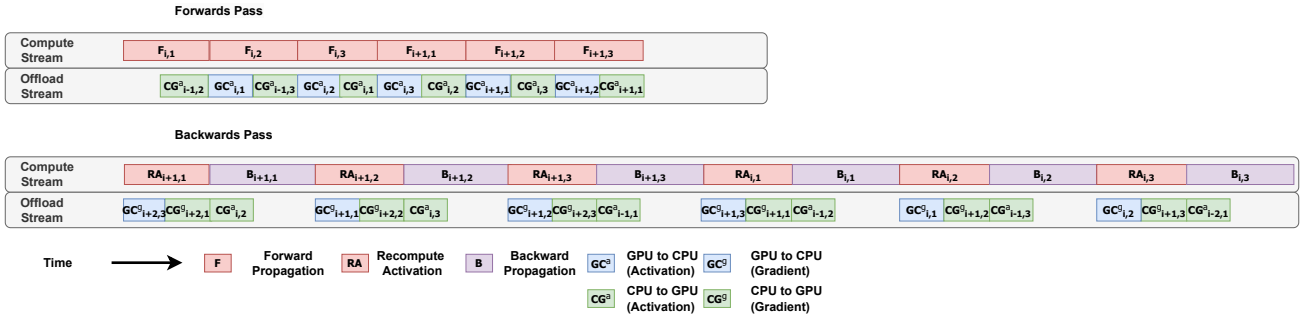

Figure 5: CPU↔GPU offloading of activations and gradients during forward and backward passes in layered gradient accumulation for layers i and $i+1$ over three microbatches. RA marks activation recomputation in the backward pass; GC^a_{ij}/CG^a_{ij} and GC^g_{ij}/CG^g_{ij} denote GPU→CPU/CPU→GPU transfers of activations and activation gradients for microbatch j.

In the backwards pass:

(1) Microbatch gradients are offloaded to the CPU while the next microbatch runs.
(2) Input activations for recomputing the next microbatch's activations are prefetched from the CPU.
(3) The subsequent layer's gradients are prefetched from the GPU to compute the current layer's gradients.

We illustrate this process for gradient accumulation with 3 microbatches in Figure 5.

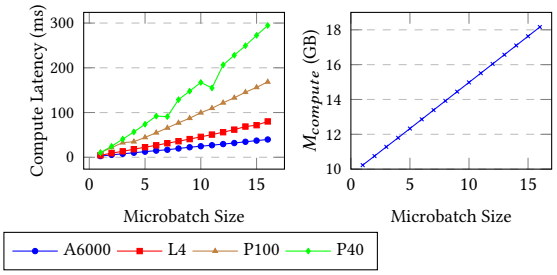

Figure 6: Training latency and memory allocated for compute as the microbatch size increases (Bert-Large).

2.3 Performance Modeling

The profiler runs training iterations on small batch sizes in the target cluster to build predictive models for compute latency and memory usage based on batch size. We use linear models, as they are simple, require minimal profiling to fit, and accurately predict both metrics. We profile communication latency for collectives with an evenly sharded training state and apply a conservative model to adjust for latency when the state is unevenly sharded. The optimizer then uses these models to find a configuration that maximizes throughput while respecting each GPU's memory capacity.

Compute Latency Model. In the left plot of Figure 6, we profile the compute latency of a single transformer layer as the batch size increases. For small batch sizes, the latency increases sublinearly as the batch size is not large enough to fully utilize the compute on the GPU. Non-linearities can also arise from certain batch sizes aligning better with GPU memory architecture, or using better kernel algorithms selected by the compiler. As the GPU compute is saturated for larger batch sizes, there is a strong linear relationship. We model latency by using the profiled data for smaller batches to capture non-linearities, then extrapolate linearly for larger batches. Profiling a single layer reduces time and resources, and since transformer layers are typically identical, we can use this to extrapolate the entire model's latency.

Let $T_f(m)$ and $T_b(m)$ be the latency models for forwards and backwards compute as a function of the microbatch size m. We linearly scale the latency of a single microbatch by the number of microbatches ℓ to derive the total forwards $T_f(m,\ell)$ and backwards $T_b(m,\ell)$ compute latencies.

Memory Utilization Model. During training, GPU memory utilization includes memory for the training state, M_{state}, and computation, $M_{compute}$, resulting in a total memory usage of $M = M_{state} + M_{compute}$. M_{state} is derived from the parameters in a GPU's model shard $|P|$. We assume standard full-precision training with the Adam optimizer, where each parameter requires 4 bytes for the parameter, 4 bytes for its gradient, and 8 bytes for the first and second gradient moments. Thus, the total memory needed for the training state is $M_{state} = 16 \cdot |P|$. $M_{compute}$ encompasses memory for materializing layer parameters, executing GPU kernels, storing activations, and other framework state. In the right plot of Figure 6, we plot $M_{compute}$ against batch size by subtracting M_{state} from the total memory usage, showing a strong linear relationship. We profile $M_{compute}$ for small batch sizes to fit a linear model based on microbatch size. The linear increase is

due to the need to run kernels and store activations for larger batch sizes. Notably, $M_{compute}$ is unaffected by the number of microbatches, as activations are checkpointed and offloaded after computation in Cephalo.

Communication Latency. FSDP uses NCCL [35] for inter-GPU communication, using *AllGather* to collect parameters and *ReduceScatter* to average gradients. With even training state sharding, inputs to NCCL collectives are equal in size; however, uneven sharding introduces variable input sizes. Cephalo employs generalized collective implementations which handle uneven inputs but incur overhead from extra GPU memory copies [65]. We observed empirically that the overhead of using uneven sharding remained within 25% of even sharding latency regardless of the input skew, defined to be the ratio of the size of the largest shard to the total input size. Thus, we profile collective latency under even input sizes and conservatively assume a 25% overhead for uneven sharding. Since transformer layers are identical, profiling a single layer suffices.

2.4 Optimizer

Given a model, cluster of N GPUs, and a target batch size B to train with, the optimizer decides how to divide the computation, training state, and configure gradient accumulation to maximize training throughput. Next, we describe how the optimizer formulates this as an optimization problem and solves it with dynamic programming.

Optimization Formulation. We maximize training throughput by minimizing the latency for one iteration of training. Under the typical assumption that transformer layers are identical, this problem is equivalent to minimizing the forwards and backwards pass for a single transformer layer. We wrap each transformer layer as an FSDP unit [38], which efficiently overlaps communication during the forwards and backwards pass. The forwards pass runs in

$$T_f = \max(\max_i(T_f^{g_i}(m_i, \ell_i)), AG), \tag{2}$$

Symbol	Description
N, B	Number of GPUs and global batch size
m_i, ℓ_i, g_i	Microbatch size and number of microbatches for ith GPU g_i
$M(m, \ell)$	Compute memory for ℓ microbatches of size m
$M_{cap}^{g_i}$	Memory capacity of g_i
$T_f^{g_i}(m, \ell)$	Forwards latency of g_i for ℓ microbatches of size m
$T_b^{g_i}(m, \ell)$	Backwards latency of g_i for ℓ microbatches of size m
AG, RG	*AllGather* and *ReduceScatter* latency
M_{state}^{es}	Memory required to store an even training state share

Table 1: Notation and Definitions

where variables are defined in Table 1. The forwards pass waits on the slowest GPU to finish its computation, as well as the *AllGather* that is running concurrently to fetch the next FSDP unit. Similarly, with activation recomputation, the backwards pass runs in

$$T_b = \max(\max_i(T_f^{g_i}(m_i, \ell_i) + T_b^{g_i}(m_i, \ell_i)), RS + AG), \tag{3}$$

where a *ReduceScatter* is required to average the gradient. The training state must be unevenly sharded if, for any GPU, its combined compute memory and memory holding the evenly distributed training state exceeds its memory capacity. Then, the goal is to minimize the layer latency $T_f + T_b$ subject to the constraints: (I) Batch size: $B = \sum_i b_i = m_i \cdot \ell_i, \ell_i \in \mathbb{Z}_{>0}$ (II) Individual memory: $M(m_i) \leq M_{cap}^{g_i}, \forall i$ (III) Aggregate memory: $M_{state} + \sum_i M(m_i) \leq \sum_i M_{cap}^{g_i}$.

The second constraint specifies that the memory used for compute cannot exceed the memory capacity of the GPU. The last constraint specifies that the aggregate GPU memory in the cluster is at least as much as the sum of the memory required to store the complete training state and perform computation on each GPU. Under these conditions, Cephalo is able to train the model without running out of memory.

Algorithm 1 Throughput Maximization using DP

Input: # of GPUs N, Batch Size B
Output: Max achievable throughput, training configuration
Initialize $DP[0 \ldots N][0 \ldots B][0 \ldots B]$ with ∞
$DP[0][0][0] \leftarrow 0$
for $i \leftarrow 1$ **to** N, $j \leftarrow 1$ **to** B, $k \leftarrow 1$ **to** j,
  $m \leftarrow 1$ **to** k, $\ell \leftarrow 1$ **to** $\lfloor j/m \rfloor$ **do**
  if $M(m, \ell) > M_{cap}^{g_i}$ **then**
    continue with the next m
  end if
  $AG' = AG$, $RS' = RS$
  if $M(m, \ell) + M_{state}^{es} > M_{cap}^{g_i}$ **then**
    $AG' = AG_{uneven}$, $RS' = RS_{uneven}$
  end if
  $T_{i,\ell,m} \leftarrow \max(T_f^{g_i}(m, \ell), AG') + \max(T_b^{g_i}(m, \ell), AG' + RS')$
  $R \leftarrow \max(DP[i-1][j - \ell \cdot m][k-m], T_{i,\ell,m})$
  $DP[i][j][k] \leftarrow \min(DP[i][j][k], R)$
end for
$optimalThroughput \leftarrow 0, solution \leftarrow None$
for $k \leftarrow 1$ to B **do**
  if $B/DP[N][B][k] > optimalThroughput$ **then**
    $optimalThroughput \leftarrow B/DP[N][B][k]$
    $solution \leftarrow Backtrack(DP[N][B][k])$
  end if
end for
return $optimalThroughput, solution$

Dynamic Programming Solution. We solve the optimization problem using dynamic programming. Let $D(i, j, k)$ be

the minimum achievable runtime for the first i GPUs to process a total batch size of j and total microbatch size of k. That is, the sum of the batch sizes on the first i GPUs is j, and the sum of their microbatch sizes is k. Suppose that the optimal solution assigns ℓ microbatches of size m (batch size of $\ell \cdot m$) to the ith GPU. Then the optimal solution can be constructed by combining this assignment with the solution to $D(i-1, j-\ell \cdot m, k-m)$. Thus, by this optimal subproblem property, we can compute $D(i, j, k)$ as

$$D(i,j,k) = \min_{m,\ell} \max(D(i-1, j-\ell \cdot m, k-m), T_{i,\ell,m}) \ , \quad (4)$$

where $\ell \cdot m \leq j, m \leq k, M(m) \leq M_{cap}^{g_i}$ and $T_{i,\ell,m}$ is the runtime of forwards and backwards for ℓ microbatches of size m on the ith GPU using Eqs. 2 and 3.

From our memory model, we can compute the aggregate memory utilization using the sum of the microbatch sizes, k. Hence, the last dimension in the recurrence represents the aggregate memory utilization. This dimension is needed in the recurrence to ensure constraint (III) is satisfied. The minimum latency is $\min_k D(N, B, k)$ over all k meeting the memory constraint. We then backtrack to find the batch and microbatch sizes that achieve this latency (Pseudocode is provided in Algorithm 1).

Partitioning Training State. After determining the compute partitioning, the optimizer allocates training state to minimize the maximum memory utilization across GPUs, balancing each GPU's memory usage relative to its capacity. This avoids out-of-memory issues and reduces memory allocation overheads when utilization approaches capacity. This allocation is determined with a greedy algorithm:

(1) Sort: Sort the GPUs in ascending order based on their total memory utilization, which includes memory required for computation and memory allocated for the training state by this algorithm.
(2) Assign: Allocate parameters to the GPU with the lowest utilization until its memory matches the GPU with the second-lowest utilization.
(3) Iterate: Repeat this for $k = 2, \ldots, N$. In the kth iteration, evenly distribute parameters to the GPUs with the kth lowest memory utilization until their utilization matches the GPU with the $(k+1)$st lowest utilization.

This process ensures that the parameters are distributed to the GPUs with the lowest memory utilization first in order to minimize the maximum memory utilization across GPUs.

Complexity Analysis. The optimizer runtime is dominated by the dynamic programming algorithm which runs in $O(N \cdot B^3 \cdot \log B)$, where N is the GPU count and B the global batch size. This arises from $O(N \cdot B^2)$ states, each requiring $O(B \cdot \log B)$ to compute. The greedy algorithm for training state partitioning runs for N iterations, where each iteration compares and assigns parameters in $O(N)$ time. Thus, the greedy algorithm runs in $O(N^2)$.

3 Implementation

This section details Cephalo's implementation and optimizations. Cephalo is implemented on top of PyTorch FSDP and consists of a profiler, optimizer and model trainer (Figure 3).

3.1 Profiler

The profiler conducts lightweight profiling to model compute latency, memory usage, and communication latency. It profiles a few training iterations for each batch size from 1 to B, fitting linear models for compute latency and memory usage. In practice, $B = 8$ suffices for accuracy. Additionally, the profiler measures *AllGather* and *ReduceScatter* latencies by averaging profiled latencies over 50 runs to obtain a representative sample of network performance.

3.2 Optimizer

The optimizer uses models built by the profiler to configure Cephalo for maximum training throughput (Section 2.4). It determines each GPU's microbatch size, number of microbatches, and assigned portion of the global batch size and training state. To avoid memory allocation bottlenecks as usage nears capacity, the optimizer caps GPU memory usage at 80%. It completes within 20 minutes for all workloads — a meager overhead to generate optimized, time-saving, training configurations compared to the GPU-years required to train these models [50].

In practice, we further reduce the optimizer runtime by observing that scaling the number of microbatches (ℓ_i) by some constant C on each GPU increases the global batch size (B) without affecting computation or memory distribution. With Cephalo's offloading optimizations (Section 2.2), memory utilization remains constant for a microbatch regardless of the number of microbatches. Moreover, constant scaling maintains a proportional compute load across GPUs. Consequently, we can optimize for a batch size of B by running the optimizer on a smaller batch size B/C and later adjusting the number of microbatches to $\ell_i' = C \cdot \ell_i$. This approach reduces optimization time for larger batch sizes.

3.3 Trainer

Compute and Training State Division. The *trainer* trains the model using the batch size and training state assignments set by the optimizer. Each process's data loader is configured to load its assigned batch size. Implementing uneven parameter sharding required modifying FSDP's *shard* and *unshard* operations to follow the training state divisions configured

by the optimizer. The gradient reduction logic in the backward pass was also modified to average gradients according to this training state division.

When the training state is unevenly sharded, Cephalo uses generalized *AllGather* and *ReduceScatter* implementations to handle uneven input sizes. We observed uneven sharding incurs up to a 25% runtime overhead, but does not have a strong correlation with the input skew in shard sizes. Therefore, we apply a greedy strategy to minimize uneven sharding across FSDP units. For instance, if two identical FSDP units are split across two GPUs in a 3:1 ratio, we would shard one unit evenly (1:1) and the other as 1:0, thereby incurring uneven sharding overhead for only one unit.

3.4 Layered Gradient Accumulation

The trainer implements a training loop for layered gradient accumulation (LGA), splitting each batch into microbatches and processing them one at a time through each FSDP unit before moving to the next. This order differs from FSDP's assumed sequential unit-by-unit execution, which it uses to overlap communication with computation. Consequently, several changes were needed in FSDP to avoid unnecessary communication and support communication-computation overlap with this new order of execution.

FSDP reshards parameters after each forward pass, assuming the next unit runs next and the current is no longer needed. However, in LGA, the same unit runs all microbatches before moving to the next. We modified FSDP to reshard parameters only after all microbatches are processed, avoiding unnecessary *AllGather*'s.

Communication-Computation Overlap. To maintain efficient overlap, we updated the prefetching logic to align with the layered gradient accumulation order and scheduled parameter unsharding logic on its own GPU stream to avoid communication from blocking backwards computation. Furthermore, we adjusted post-backward logic to accumulate gradients across microbatches and reset the execution state only after all microbatches are processed. We configured the FSDP prefetcher (with the *limit_all_gathers* flag [7]) to retain only the current and subsequent layers' parameters in memory, avoiding memory overhead from excessive prefetching.

Compute Synchronization. We observed severe memory fragmentation from PyTorch scheduling multiple microbatches simultaneously, leading to out-of-memory errors even below 50% memory usage. We avoid this fragmentation by synchronizing the GPU's compute stream to process one microbatch at a time.

Offloading. PyTorch's default activation offloading incurred large runtime overheads due to synchronous CPU-GPU transfer, which blocked GPU computation. Consequently, we implemented our own offloading and prefetching logic on a dedicated GPU stream, enabling GPU-CPU memory transfer to run in parallel with computation.

In Section 4.4, we show that our optimizations for LGA are essential for achieving high performance.

Task	Model	Layers	Embd. Size	Attn. Heads	Parameters
IC	ViT-G [61]	48	1664	16	1.8B
IC	ViT-e [6]	56	1792	16	3.9B
TC	BERT-Large [8]	24	1024	16	0.4B
TC	BERT-XLarge [8]	36	1536	24	1.2B
TG	GPT 2.7B [5]	32	2560	80	2.7B
TG	GPT 6.7B [5]	32	4096	128	6.7B
TG	Tiny Llama [63]	22	2048	32	1.1B
TG	Llama 3B [12]	26	3200	32	3.5B
TG	Llama 7B [50]	32	4096	32	6.7B

Table 2: Model Statistics.

Cluster	GPUs	Generation	Memory	TFlops	Total Memory	Total TFlops
A	3×P40	Pascal	24 GB	11.8	192 GB	153.3
	2×P100	Pascal	12 GB	9.3		
	1×A6000	Ampere	48 GB	38.7		
	2×L4	Ada	24 GB	30.3		
B	16×V100	Volta	16 GB	14.1	1120 GB	984.0
	32×T4	Turing	15 GB	8.1		
	16×A10G	Ampere	24 GB	31.2		

Table 3: GPU Specifications (FP32 TFlops).

4 Performance Evaluation

We evaluate the performance of Cephalo compared to state-of-the-art training methods on 9 popular transformer models across 2 heterogeneous GPU clusters. End-to-end results are presented in Section 4.2, and larger-scale experiments in Section 4.3. Sections 4.4 and 4.5 analyze how Cephalo's design components impact performance. Section 4.6 evaluates the accuracy of Cephalo's performance model and Section 4.7 presents training configurations generated by Cephalo.

4.1 Experimental Setup

We evaluate popular transformer models used for text classification (TC), text generation (TG), and image classification (IC) following the training setup from [38]. Activations are checkpointed after each transformer layer and models are trained in full precision with the Adam optimizer, using a sequence length of 512 for language modeling (unless otherwise specified). Table 2 provides further details on the models.

Clusters. We evaluated Cephalo on environments representative of typical heterogeneous GPU clusters used by ML practitioners. Cluster A was assembled with four types of GPUs acquired over several years. Cluster B is a mix of higher- and lower-end GPU VMs on AWS, selected to reflect the typical quantities available for reservation.

Table 4: Throughput comparison of different models/batch sizes on 8-GPU Cluster A. *OOM* denotes Out-of-Memory.

System	ViT-G		ViT-e		Bert-Large		Bert-XLarge		GPT 1.3B		GPT 2.7B		Tiny Llama		Llama 3B	
	128	256	128	256	128	256	128	256	128	256	128	256	128	256	128	256
Megatron-Het	3.41	0.79	*OOM*	*OOM*	19.77	20.57	6.40	6.80	4.18	4.35	1.82	1.82	7.93	8.63	*OOM*	*OOM*
HexiScale	2.88	2.97	1.38	1.4	25.64	28.90	8.63	9.06	5.81	5.83	2.79	2.83	8.67	8.75	1.91	1.83
Cephalo	**6.38**	**6.41**	**3.02**	**3.23**	**33.56**	**33.69**	**11.47**	**11.72**	**6.83**	**7.09**	**4.57**	**4.67**	**12.58**	**12.91**	**4.51**	**4.85**

- Cluster A: 2 machines (8 GPUs), connected via a 50 Gbps link. One contains 2×L4, 1×A6000, and 1×P40; the other contains 2×P40 and 2×P100.
- Cluster B: 8 VMs (64 GPUs), equipped with 100 Gbps bandwidth. 2×g5.48xlarge (8×A10G), 2×p3.16xlarge (8×V100-16GB), and 4×g4dn.metal (8×T4) VMs.

A summary of GPU specifications appear in Table 3.

Baselines. We compare against representative state-of-the-art techniques for training on heterogeneous GPU clusters:

- Megatron-Het [34]: Pipeline parallelism across nodes and data/tensor parallelism within nodes. We adapted it for heterogeneous training by partitioning the model proportionally to each node's compute capacity.
- HexiScale [60]: Combines ZeRO-2 data [39] (optimizer state and gradient sharding), tensor, and pipeline parallelism. An optimizer balances memory and compute.

4.2 Training Throughput

We evaluated Cephalo's end-to-end training throughput against baselines, measuring throughput as samples processed per second (images for image classification models, sequences for language models). Experiments on Cluster A included models of up to 3.9 billion parameters with global batch sizes of 128 and 256. Cluster A is highly heterogeneous, with four GPU types varying substantially in compute and memory. Baselines do not auto-configure pipeline parallelism, so we tested various microbatch sizes (powers of 2), with the best results reported in Table 4. Cephalo consistently achieved significantly higher throughput without out-of-memory (*OOM*) errors across all models and batch sizes.

Comparison to Megatron-Het. Megatron uses four pipelines of two GPUs each across the two nodes. However, each pipeline must be partitioned identically, despite the mixed GPU types on each node. This results in different GPUs being assigned the same stage across pipelines, causing compute bottlenecks due to the slower P40 GPUs, which underutilizes faster L4 and A6000 GPUs, reducing throughput. For larger models (GPT 2.7B and Llama 3B), Megatron applies tensor parallelism within each node, further decreasing throughput due to high communication overhead. Megatron is optimized for clusters with fast interconnects like NVSwitch, which Cluster A and most AWS VMs do not have (except mostly unavailable A100 and H100 VMs).

Table 5: Throughput comparison on 64-GPU Cluster B.

System	ViT-e		GPT 6.7B		Llama 7B	
	512	1024	512	1024	512	1024
Megatron-Het	12.06	12.12	3.59	1.71	5.53	1.65
HexiScale	12.84	13.37	4.78	4.99	5.42	5.47
Cephalo	**20.37**	**26.08**	**11.62**	**17.04**	**13.12**	**17.74**

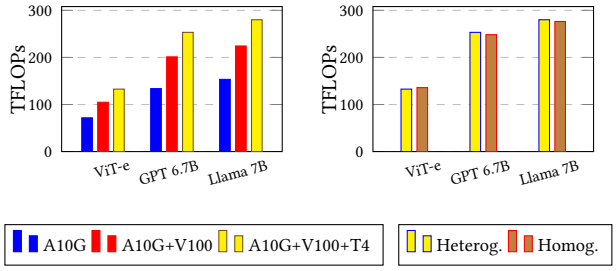

Figure 7: Left: Throughput (TFLOPs) with different heterogeneous cluster configurations. Right: Throughput on Cluster B vs. a 32×A10G homogeneous cluster.

Comparison to HexiScale. Like Cephalo, HexiScale trains larger batch sizes with a reduced memory footprint by using smaller microbatches and gradient accumulation. However, smaller microbatches may not fully utilize GPU compute, and frequent gradient accumulation reduces pipeline parallelism efficiency. Cephalo automatically optimizes the microbatch size and gradient accumulation configuration, whereas HexiScale requires manual tuning. Additionally, Cephalo's layered gradient accumulation implementation does not incur extra communication overhead. HexiScale, like Megatron-Het, relies on communication-heavy tensor parallelism for larger models. These factors enable Cephalo to achieve significantly higher throughput across all configurations.

4.3 Larger Cluster Experiments

We evaluated Cephalo's scalability on the larger Cluster B featuring 64 GPUs (16 V100s, 16 A10Gs, and 32 T4s) using ViT-e, GPT-6.7B, and Llama-7B models with batch sizes of 512 and 1024. Cephalo consistently delivered 2–10× higher throughput than other systems.

At a batch size of 512, Megatron uses ZeRO-2 data parallelism within each node. Since it does not shard the model parameters like Cephalo, Megatron needs to configure pipeline

parallelism with a smaller microbatch size and a suboptimal model partitioning to avoid running out of memory. It is unable to fully utilize compute on the V100 GPUs since it has similar memory to the T4 despite being significantly faster. At a batch size of 1024, Megatron uses tensor parallelism to manage memory. However, this reduces throughput for GPT 6.7B and Llama 7B, as V100 GPUs' NVLink lacks all-to-all connectivity and is not fast enough to offset the communication overhead of tensor parallelism.

HexiScale is able to more flexibly parallelize training, supporting a different degree of tensor parallelism per pipeline stage and a different number of GPUs for each pipeline. This enables faster training at a batch size of 1024 when memory pressure is larger. However, it still relies on tensor parallelism (albeit less than Megatron) and partitions layers into pipeline stages according to memory, rather than compute, to avoid running out of memory. This partitioning assigns the T4s a similar workload as the V100s, despite being slower, resulting in a performance bottleneck.

In contrast, Cephalo leverages FSDP to shard training state, reducing memory requirements and enabling training at a batch size of 1024 without tensor parallelism. Additionally, independent partitioning of training state from compute allows Cephalo to fully utilize each GPU by assigning batch sizes proportional to its compute capacity.

Scaling Heterogeneous GPUs. In the left plot of Figure 7, we compare the training throughput (in TFLOPs) of Cephalo as we scale from using only the fastest A10G GPUs in Cluster B, to using the A10G and V100 GPUs, to finally using all GPUs. The training throughput almost doubles when comparing using only A10G to utilizing all the heterogeneous GPUs in the cluster. Cephalo is able to achieve a significant improvement in training throughput by utilizing all of the (heterogeneous) GPUs available on the cluster.

Comparison to Homogeneous Training. In the right plot of Figure 7, we compare Cephalo's training TFLOPs on Cluster B to a homogeneous cluster of 32×A10Gs with similar peak TFLOPs (984 vs. 998). Despite Cluster B's mix of lower-memory lower-compute GPUs, Cephalo is able to achieve comparable TFLOPs to the homogeneous cluster, demonstrating effective utilization of heterogeneous GPUs.

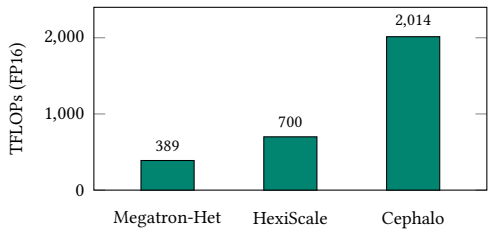

Figure 8: Mixed Precision Training on Cluster B.

Mixed Precision Training. Our evaluation focuses on full precision training since it preserves model convergence guarantees and older GPUs in cluster A were not optimized for lower precision. However, Cephalo's design is also effective for lower-precision training, where non-proportional scaling of GPU memory and compute still exists. For example, the V100 and T4 have similar memory capacity but the V100 has 2× higher FP16 TFLOPs. We validate Cephalo's effectiveness at lower precisions by showing it outperforms other systems when training Llama 7B with FP16 mixed precision [28], a batch size of 1024, and a sequence length of 2048 on Cluster B (Figure 8). Megatron-Het and HexiScale use more communication intensive configurations due to memory constraints and obtain lower throughput due to communication overhead. Generally, hiding latencies through communication-computation overlap at lower precisions is more difficult as both computation and communication latencies decrease, but computation latency at a greater rate. Despite this, Cephalo manages to effectively hide communication latencies with its memory and communication efficient implementation of FSDP with layered gradient accumulation.

4.4 Ablation Study

We conducted an ablation study to assess the individual and joint contributions of compute and memory balancing to Cephalo's performance. We compared Cephalo's throughput with two variants: compute balancing only (Cephalo-CB) and memory balancing only (Cephalo-MB), alongside baseline FSDP. Experiments were run on Cluster A with ViT-e, GPT-2.7B, and Llama-3B, scaling batch sizes to 256, as shown in Figure 9. Cephalo-CB improves throughput over FSDP by balancing compute but encounters out-of-memory (OOM) issues beyond a batch size of 100 for all models, with throughput declining as it nears max memory capacity. Cephalo-MB prevents OOM by balancing memory with uneven training state partitioning and using gradient accumulation with a microbatch size of 1. However, its throughput is lower than FSDP's, as gradient accumulation with such a small microbatch size fails to fully utilize GPU compute, underscoring the need for prudently configuring gradient accumulation. Cephalo overcomes Cephalo-CB and Cephalo-MB limitations by jointly balancing compute, memory, and gradient accumulation, essential for high throughput on heterogeneous clusters. It achieves the highest training throughput across all batch sizes and sustains high throughput up to a batch size of 256 without running OOM.

4.5 Gradient Accumulation Optimizations

In Figure 10, we investigate the throughput and memory improvements obtained from Cephalo's gradient accumulation optimizations. Starting from the existing gradient accumulation in FSDP (FSDP-GA), we introduce layered gradient

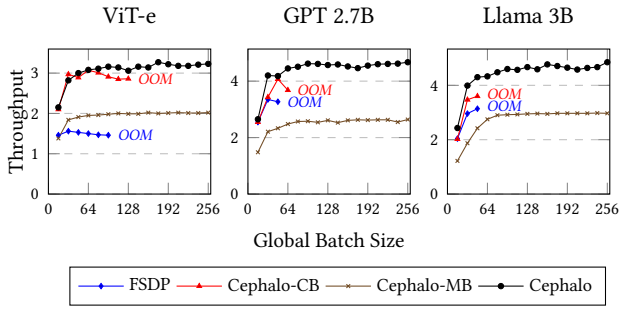

Figure 9: Throughput at different batch sizes with, and without, compute and memory balancing.

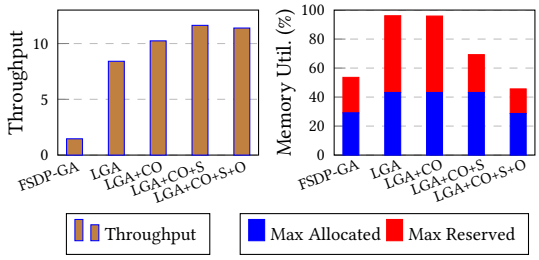

Figure 10: Speedup and memory reduction from our gradient accumulation optimizations (LGA+CO+S+O).

accumulation (LGA), then add communication overlap with computation (CO), compute synchronization (S), and offloading (O). We train the GPT 6.7B model with a batch size of 256 (16 microbatches of size 1 per GPU). A homogeneous cluster of 16×V100 GPUs is used to isolate from the effects of heterogeneous GPUs.

While FSDP-GA encounters communication bottlenecks, LGA achieves a 6× speedup by minimizing communication overhead and increases throughput by 22% through full communication overlap with gradient accumulation. Additionally, compute synchronization and offloading eliminate memory overhead and fragmentation, boosting throughput by an extra 11%. The final implementation with all optimizations (LGA+CO+S+O) delivers a 7.8× speedup over FSDP-GA while reducing memory usage.

4.6 Performance Model Accuracy

Cephalo's optimizer uses a performance model to predict runtime across training configurations, which is essential for efficiently navigating the large search space and optimizing configurations. Figure 11 shows the absolute relative error (ARE) between predicted and actual latencies on Cluster A. Across all models and batch sizes, errors remained within 10%, with a mean absolute relative error of 2.9%. Notably,

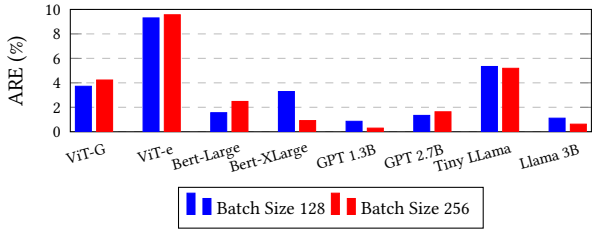

Figure 11: Performance model absolute relative error.

error rates did not increase for larger models or batch sizes, demonstrating the performance model's robustness.

4.7 Optimized Training Configurations

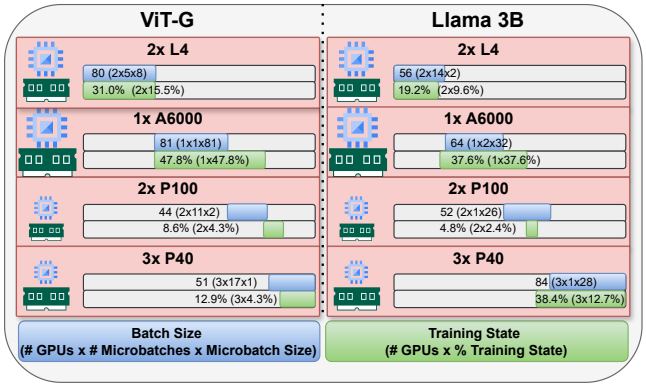

Figure 12: Training configurations (ViT-G & Llama 3B).

In Figure 12, we show Cephalo's optimized configurations for ViT-G and Llama 3B on Cluster A with batch size 256. The A6000 GPU, being faster and having more memory than the L4s, P100s, and P40s, is assigned the largest portion of the training state and compute. The L4s, with about half the compute and memory of the A6000, receive roughly half the batch size and training state. P100s and P40s are assigned smaller batch sizes, with the P40 handling a larger training state due to its greater memory capacity.

5 Discussion

Cephalo is designed to address compute and memory imbalances in heterogeneous clusters. This section discusses Cephalo's adaptability to various networking environments.

5.1 Adapting to Slower Networks

Network communication can become a bottleneck during training when network speed is slow compared to computation capabilities. The layered gradient accumulation optimizations in Cephalo allow both communication overhead and GPU memory usage to remain constant while increasing the number of microbatches. Consequently, when network

latency significantly limits training performance, users are able to increase the global batch size to scale relative computation time, enabling full overlapping and masking of communication overhead. Although extremely large batch sizes could potentially impact model convergence, organizations training at this scale typically possess very large, dedicated, homogeneous GPU clusters [11, 18], eliminating the need for heterogeneous training solutions. Cephalo specifically targets mid-sized clusters where heterogeneous compute is still frequently required [17, 26, 36, 64] and batch sizes remain well within the established limits at which modern transformers can achieve efficient convergence [11].

5.2 Adapting to Heterogeneous Networks

Heterogeneous networks with varying interconnect speeds can bottleneck *AllGather* and *ReduceScatter* collectives in FSDP on slower links. Although increasing batch size can reduce networking overheads, modifying the parallelism strategy can also be effective. Hierarchically-sharded FSDP variants replicate parameters within GPU subgroups connected by faster interconnects [53, 59, 65], minimizing communication across slower links during parameter gathering.

Integrating pipeline parallelism with FSDP offers an orthogonal optimization. By partitioning the model into stages, FSDP can be applied within each stage among GPUs with faster interconnects, while GPUs connected with slower links are assigned to different stages. This approach leverages communication-efficient pipeline parallelism over slower links and utilizes FSDP across faster links. In stages with very fast interconnects, tensor parallelism can be combined with FSDP to scale computation without increasing batch size. Optimizations in compute-communication overlap for tensor parallelism can help maintain high training efficiency [54, 55]. We leave efficient integration of other parallelism strategies in Cephalo for future work.

6 Related Work

Distributed training strategies falls into two main categories: data and model parallelism. Data parallelism assigns each GPU a distinct data batch to process on the complete model [42]. ZeRO-3 [39] and FSDP [65] minimize memory usage by sharding training states across GPUs rather than replicating them. Model parallelism divides the model among GPUs, with pipeline parallelism distributing model layers across GPUs [15, 30] and tensor parallelism splitting individual layer computations across multiple GPUs [43, 44].

Several systems combine data and model parallelism for training [13, 18, 34, 45, 66]. Megatron-LM [34] uses heuristics to select parallelization strategies: tensor parallelism is used within nodes with fast interconnects, pipeline parallelism spans nodes with slower connections, and data parallelism is employed for further scaling. Alpa [66] utilizes a dynamic programming optimizer to find optimal parallelism configurations. Although Cephalo focuses solely on data parallelism, it has demonstrated high effectiveness in evaluations on its targeted workload and can be integrated with model-parallel techniques for broader applicability (Section 5.2).

Given the cost and limited availability of GPUs, heterogeneous clusters are commonly assembled to aggregate compute resources [3, 17, 47] for deep learning workloads. Prior work has explored the optimization of data parallelism [16, 20, 29], model parallelism [64], and hybrid approaches [51, 60] for heterogeneous clusters. However, these methods distribute the workload solely based on compute speed and overlook differences in memory capacity. In constrast, Cephalo balances both the computational and memory workload across GPUs, thereby maximizing cluster utilization and enabling the training of larger models and batch sizes without sacrificing efficiency. Like prior work [16, 20, 51, 60, 64], our evaluation utilizes NVIDIA GPUs due to their widespread availability and common usage in training [24]. However, Cephalo can be easily adapted for any accelerator compatible with PyTorch, including AMD GPUs [4] and TPUs [14, 19].

Other systems optimize scheduling for multiple training jobs in heterogeneous clusters but restrict each job to homogeneous GPU resources, precluding heterogeneous training for a single job [2, 33, 48, 57]. Cephalo could be integrated with cluster schedulers to improve utilization in shared heterogeneous clusters. Activation checkpointing [30, 32] and offloading [40, 58] are techniques for reducing memory during training. However, Cephalo is the first system to optimize these techniques for layered gradient accumulation.

7 Conclusion

Cephalo is the first system that jointly resolves imbalances in compute and memory across GPUs when training on a heterogeneous cluster. It decouples compute and memory requirements for each GPU through uneven compute division, parameter sharding, and gradient accumulation. Cephalo models compute, memory, and communication holistically and uses an optimizer to optimally allocate training state, batch size, and gradient accumulation across GPUs. Evaluations on multiple clusters show that Cephalo achieves significantly higher training throughput while supporting larger models and batch sizes than existing systems.

Acknowledgments

We are thankful to the anonymous reviewers for providing valuable feedback and Lori Paniak for hardware support used in this work. This work was supported by funding from the Natural Sciences and Engineering Research Council of Canada and credits from Amazon Web Services.

References

[1] Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219* (2024).

[2] Nawras Alnaasan, Bharath Ramesh, Jinghan Yao, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda. 2024. HyperSack: Distributed Hyperparameter Optimization for Deep Learning using Resource-Aware Scheduling on Heterogeneous GPU Systems. In *2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 100–110. https://doi.org/10.1109/HiPC62374.2024.00019

[3] Amazon Web Services, Inc. 2024. AWS Cloud Computing. https://aws.amazon.com/ Accessed: 2024-02-06.

[4] AMD. 2025. *PyTorch on ROCm*. https://rocm.docs.amd.com/projects/install-on-linux/en/latest/install/3rd-party/pytorch-install.html Accessed: 2025-02-24.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[6] Xi Chen and Xiao Wang. 2022. PaLI: Scaling Language-Image Learning in 100+ Languages. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[7] PyTorch Contributors. 2025. *torch.distributed.fsdp.FullyShardedDataParallel*. PyTorch. https://pytorch.org/docs/stable/fsdp.html#torch.distributed.fsdp.FullyShardedDataParallel

[8] 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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations*. https://openreview.net/forum?id=YicbFdNTTy

[10] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. 2024. Parcae: proactive, liveput-optimized DNN training on preemptible instances. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 62, 19 pages.

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

[12] Xinyang Geng and Hao Liu. 2023. *OpenLLaMA: An Open Reproduction of LLaMA*. https://github.com/openlm-research/open_llama

[13] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. 2022. Hydrozoa: Dynamic hybrid-parallel dnn training on serverless containers. *Proceedings of Machine Learning and Systems* 4 (2022), 779–794.

[14] Ronghang Hu, Vaibhav Singh, Jack Cao, Milad Mohammadi, Yeounoh Chung, Shauheen Zahirazami, and Ross Girshick. 2022. Scaling PyTorch Models on Cloud TPUs with FSDP. https://pytorch.org/blog/scaling-pytorch-models-on-cloud-tpus-with-fsdp/ Accessed: 2025-02-24.

[15] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[16] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. 2022. Whale: Efficient Giant Model Training over Heterogeneous GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 673–688. https://www.usenix.org/conference/atc22/presentation/jia-xianyan

[17] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Guoliang He, Xupeng Miao, Ana Klimovic, Bin Cui, Binhang Yuan, and Eiko Yoneki. 2025. Demystifying Cost-Efficiency in LLM Serving over Heterogeneous GPUs. arXiv:2502.00722 [cs.DC] https://arxiv.org/abs/2502.00722

[18] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: scaling large language model training to more than 10,000 GPUs. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 41, 16 pages.

[19] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 82, 14 pages. https://doi.org/10.1145/3579371.3589350

[20] Kyeonglok Kim, Hyeonsu Lee, Seungmin Oh, and Euiseong Seo. 2022. Scale-Train: A Scalable DNN Training Framework for a Heterogeneous GPU Cloud. *IEEE Access* 10 (2022), 68468–68481.

[21] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[22] Joel Lamy-Poirier. 2021. Layered gradient accumulation and modular pipeline parallelism: fast and efficient training of large language models. *arXiv preprint arXiv:2106.02679* (2021).

[23] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3005–3018. https://doi.org/10.14778/3415478.3415530

[24] MarketsandMarkets. 2023. Nvidia's Dominance in the AI Chip Market. https://www.marketsandmarkets.com/blog/SE/nvidia-dominance-in-the-ai-chip-market

[25] Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, and Bin Cui. 2021. Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2262–2270. https://doi.org/10.1145/3448016.3452773

[26] Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, and Zhihao Jia. 2023. Sdpipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2354–2363.

[27] 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. *Proc. VLDB Endow.* 16, 3 (nov 2022), 470–479. https://doi.org/10.14778/3570690.3570697

[28] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *International Conference on Learning Representations*. https://openreview.net/forum?id=r1gs9JgRZ

[29] Sergio Moreno-Alvarez, Juan M Haut, Mercedes E Paoletti, Juan A Rico-Gallego, Juan C Diaz-Martin, and Javier Plaza. 2020. Training deep neural networks: a static load balancing approach. *The Journal of Supercomputing* 76 (2020), 9739–9754.

[30] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[31] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[32] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*. PMLR, 7937–7947.

[33] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 481–498. https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak

[34] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[35] NVIDIA. 2024. NCCL: NVIDIA Collective Communications Library. https://developer.nvidia.com/nccl.

[36] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. {HetPipe}: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 307–321.

[37] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D. Sinclair. 2023. Tale of Two Cs: Computation vs. Communication Scaling for Future Transformers on Future Hardware. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*. 140–153. https://doi.org/10.1109/IISWC59245.2023.00026

[38] PyTorch. 2023. Training a 1 Trillion Parameter Model with PyTorch Fully Sharded Data Parallel on AWS. https://shorturl.at/6Y4LT. Accessed: 2024-01-30.

[39] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[40] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 551–564. https://www.usenix.org/conference/atc21/presentation/ren-jie

[41] Timo Schick and Hinrich Schütze. 2020. It's not just size that matters: Small language models are also few-shot learners. *arXiv preprint arXiv:2009.07118* (2020).

[42] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[43] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems* 31 (2018).

[44] 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. *CoRR* abs/1909.08053 (2019). http://arxiv.org/abs/1909.08053

[45] Siddharth Singh, Olatunji Ruwase, Ammar Ahmad Awan, Samyam Rajbhandari, Yuxiong He, and Abhinav Bhatele. 2023. A Hybrid Tensor-Expert-Data Parallelism Approach to Optimize Mixture-of-Experts Training. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 203–214. https://doi.org/10.1145/3577193.3593704

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

[47] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. 2024. ML Training with Cloud GPU Shortages: Is Cross-Region the Answer?. In *Proceedings of the 4th Workshop on Machine Learning and Systems* (Athens, Greece) *(EuroMLSys '24)*. Association for Computing Machinery, New York, NY, USA, 107–116. https://doi.org/10.1145/3642970.3655843

[48] Suhas Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 642–657.

[49] Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. 2019. BERT4Rec: Sequential recommendation with bidirectional encoder representations from transformer. In *Proceedings of the 28th ACM international conference on information and knowledge management*. 1441–1450.

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

[51] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. 2024. Metis: Fast Automatic Distributed Training on Heterogeneous {GPUs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 563–578.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[53] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Xiaoxia Wu, Connor Holmes, Zhewei Yao, Samyam Rajbhandari, Olatunji Ruwase, Feng Yang, Lei Yang, and Yuxiong He. 2024. ZeRO++: Extremely Efficient Collective Communication for Large Model Training. In *ICLR 2024*. https://www.microsoft.com/en-us/research/publication/zero-extremely-efficient-collective-communication-for-large-model-training/

[54] Guanhua Wang, Chengming Zhang, Zheyu Shen, Ang Li, and Olatunji Ruwase. 2024. *Domino: Eliminating Communication in LLM Training via Generic Tensor Slicing and Overlapping*. Technical Report MSR-TR-2024-40. Microsoft. https://www.microsoft.com/en-us/research/publication/domino-eliminating-communication-in-llm-training-via-generic-tensor-slicing-and-overlapping/

[55] Yifu Wang, Horace He, Less Wright, Luca Wehrstedt, Tianyu Liu, and Wanchao Liang. 2024. Distributed w/ TorchTitan: Introducing Async Tensor Parallelism in PyTorch. https://discuss.pytorch.org/t/distributed-w-torchtitan-introducing-async-tensor-parallelism-in-pytorch/209487 Accessed: 2025-02-24.

[56] Alex Woodie. 2023. How AWS Plans to Cope with GenaAI's Insatiable Desire for Compute. *Datanami* (11 Dec 2023). https://shorturl.at/Gx69T Accessed: 2024-02-06.

[57] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[58] Dong Xu, Yuan Feng, Kwangsik Shin, Daewoo Kim, Hyeran Jeon, and Dong Li. 2024. Efficient Tensor Offloading for Large Deep-Learning Model Training based on Compute Express Link. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 94, 18 pages. https://doi.org/10.1109/SC41406.2024.00100

[59] Lang Xu, Quentin Anthony, Jacob Hatef, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda. 2024. Scaling Large Language Model Training on Frontier with Low-Bandwidth Partitioning. In *Proceedings of the 2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC)*. Bengaluru, India, 57–67. https://doi.org/10.1109/HiPC62374.2024.00015

[60] Ran Yan, Youhe Jiang, Xiaonan Nie, Fangcheng Fu, Bin Cui, and Binhang Yuan. 2025. HexiScale: Accommodating Large Language Model Training over Heterogeneous Environment. arXiv:2409.01143 [cs.DC] https://arxiv.org/abs/2409.01143

[61] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. 2022. Scaling vision transformers. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 12104–12113.

[62] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. 2020. PEGASUS: pre-training with extracted gap-sentences for abstractive summarization. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 1051, 12 pages.

[63] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. TinyLlama: An Open-Source Small Language Model. arXiv:2401.02385 [cs.CL]

[64] Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. 2024. HAP: SPMD DNN Training on Heterogeneous GPU Clusters with Automated Program Synthesis. In *Proceedings of the European Conference on Computer Systems (EuroSys '24)* (Athens, Greece, April 22–25). ACM, New York, NY, USA, 18. https://doi.org/10.1145/3627703.3629580

[65] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3848–3860. https://doi.org/10.14778/3611540.3611569

[66] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.