SYNERGYSCALE: Optimizing Offloading and Task Partitioning for Efficient Model Training

Xiaoyang Sun, *Member, IEEE*, Jie Xu, *Member, IEEE*, and Zheng Wang, *Senior Member, IEEE*

***Abstract*—Deep neural networks (DNNs) with billions of parameters power many important applications, but their training is fundamentally constrained by the limited on-chip memory of GPUs. This memory wall forces training to rely on distributed execution or memory offloading, both of which introduce substantial inefficiencies. Existing offloading techniques can scale model size but often incur severe throughput degradation, while conventional distributed training suffers from poor hardware utilization due to limited cross-device bandwidth, leaving accelerator resources underused. We present SYNERGYSCALE, a software framework that addresses the GPU memory wall by jointly optimizing memory usage and computation efficiency. SYNERGYSCALE combines dynamic hierarchical memory offloading across GPUs, CPU, and secondary storage tiers to reduce GPU memory pressure with fine-grained multi-stream task partitioning that takes advantage of the concurrent execution capabilities of modern GPUs. The framework automatically selects offloading parameters that balance data movement overhead with parallel execution efficiency. Experimental results show that SYNERGYSCALE can train a 114.41B-parameter model on a single 80 GB NVIDIA A100 GPU and a 547.77B-parameter model on eight A100 GPUs, supporting models up to 29.5× larger than those enabled by existing approaches. By co-optimizing offloading and computation, SYNERGYSCALE achieves up to 4.5× higher training throughput than offloading-only methods, without altering training semantics, making billion-scale model training both efficient and cost-effective.**

***Index Terms*—Deep learning, distributed training, offloading, multi-streaming.**

I. INTRODUCTION

THE scale of deep neural networks (DNNs) has increased at an unprecedented pace [1], [2], [3], closely following empirical scaling laws [4]. Within only a few years, model sizes have grown by more than three orders of magnitude, from approximately 100 M parameters in BERT [2], to 175B in GPT-3 [3], and further to 235B in Alibaba Qwen [5], 405B in Meta LLaMA [6], and 671B in DeepSeek [7].

In contrast, hardware memory capacity has advanced much more slowly. For example, NVIDIA's V100, A100/H100, and B100 GPUs provide 32 GB, 80 GB, and 192 GB of memory, respectively. The improvement corresponds to only about a 2.5× increase per hardware generation. This widening gap between rapidly growing model memory requirements and modest hardware improvements has created a fundamental *memory wall*. The memory wall has become a dominant bottleneck in the training of large DNNs. Many modern models can no longer fit within the memory of a single GPU, even before accounting for optimizer states, activations, and intermediate buffers. As a result, training state-of-the-art billion-scale models requires large, expensive GPU clusters, effectively limiting participation to a small number of well-resourced organizations [8]. This trend creates a serious barrier to reproducibility and broader participation in research.

To overcome memory limitations, previous work has explored memory-saving techniques and distributed training strategies [9], [10], [11], [12], [13], [14], [15]. Memory offloading methods [16], [17], [18], [19] transfer model states between GPU and CPU memory (or secondary storage), while distributed approaches partition parameters and computation across devices [13], [14], [15], [11].

However, these techniques introduce new performance bottlenecks. Offloading model states to CPU memory can severely reduce training throughput due to limited PCIe bandwidth and added transfer latency. Distributed training methods, including tensor [13], [14] and pipeline parallelism [15], [11], often suffer from low hardware utilization, as GPUs and CPU cores remain idle during synchronization, pipeline bubbles, or data stalls [20], [21]. In practice, existing approaches typically trade memory capacity for performance, or vice versa, rather than addressing both simultaneously.

A key limitation of current systems is that memory optimization and compute parallelism are typically handled in isolation. Memory offloading decisions are often made without considering their impact on parallel execution, while parallelization strategies assume a fixed memory budget. This separation leads to suboptimal outcomes: freed memory is not effectively reinvested to improve parallelism, and parallel execution frequently stalls due to poorly coordinated data movement.

We present SYNERGYSCALE, a novel software framework that addresses these challenges by jointly optimizing *memory management* and *computation parallelism*. Rather than treating offloading and parallel execution as independent concerns, SYNERGYSCALE tightly integrates a latency-hiding memory offloading mechanism with fine-grained, multi-stream task execution. This design rethinks memory management as a source of parallelism: memory freed through offloading is immediately

Received 3 February 2026; revised 2 July 2026; accepted 28 July 2026. Date of publication 3 August 2026; date of current version 25 August 2026. This work was supported in part by the U.K. Engineering and Physical Sciences Research Council (EPSRC) under Grant Agreement EP/X037304/1 and in part by A Royal Society Industry Fellowship. Recommended for acceptance by F. Zhang.*(Corresponding author: Zheng Wang.)*

The authors are with the University of Leeds, LS2 9JT Leeds, U.K. (e-mail: X.Sun4@leeds.ac.uk; J.Xu@leeds.ac.uk; Z.Wang5@leeds.ac.uk).

Digital Object Identifier 10.1109/TPDS.2026.3719351

1045-9219 © 2026 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

exploited to launch additional concurrent computation to improve device utilization. A centralized co-optimization engine explicitly coordinates memory movement and task scheduling to prevent either from becoming the dominant bottleneck. As a result, SYNERGYSCALE enables the training of larger models while simultaneously improving training throughput.

We conducted an extensive evaluation showing that SYNERGYSCALE can train models with up to 114.41B parameters on a single 80 GB GPU (and 547.77B on eight A100 GPUs), while achieving up to 4.5× higher throughput than state-of-the-art offloading-based systems [22], [8], [23].

Conference extension: This article extends StrongHold [21] published in *SC'22 (The International Conference for High Performance Computing, Networking, Storage, and Analysis)*. The extension makes the following new contributions to the conference paper:

1) We introduce SYNERGYSCALE, a software system that jointly optimizes dynamic per-layer memory offloading (extending the approach in [21]) and intra-GPU multi-stream task partitioning, simultaneously addressing GPU memory constraints and hardware underutilization.
2) We propose a fine-grained task-partitioning mechanism that enables intra-device data parallelism by executing micro-batches concurrently across multiple CUDA streams, substantially reducing idle compute time.
3) We design a co-optimization engine that explicitly models the interaction between offloading and parallel execution, allowing SYNERGYSCALE to outperform approaches that apply either technique in isolation.
4) We conduct an extended evaluation on NVIDIA A100 GPUs that demonstrates SYNERGYSCALE on modern hardware and on advanced model architectures, including models with Mixture-of-Experts (MoE) layers. The evaluation also isolates the value of the co-optimization engine, explains why SYNERGYSCALE preserves training semantics, and shows how the analytical model accounts for hardware contention through profiled effective times.

II. BACKGROUND AND MOTIVATION

DNN model training: DNN training typically requires millions of iterations over multiple epochs. Each iteration consists of three main phases: *forward propagation* (FP), *backward propagation* (BP), and *parameter update*. During FP, the model processes a batch of training samples to compute a loss value based on a predefined objective function. In BP, this loss is propagated backward through the network to compute gradients for each layer. Finally, an optimizer aggregates these gradients and updates the model parameters accordingly.

The memory footprint of DNN training is primarily determined by two categories of states: model states and residual states. Model states include the model parameters, their corresponding gradients, and optimizer-related statistics. For example, the Adam optimizer [24] maintains additional momentum and variance terms for each parameter. Residual states consist of activations, i.e., intermediate tensors saved during FP and reused in BP, as well as other temporary buffers. In large-scale

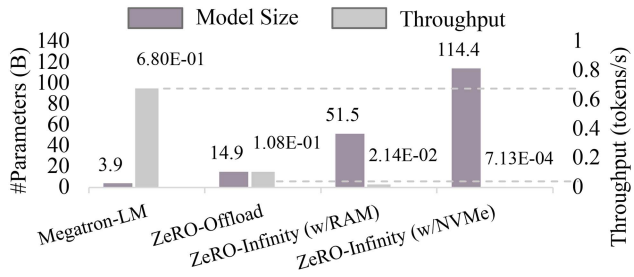


Fig. 1. Throughput and largest trainable model sizes of Megatron-LM and ZeRO-based solutions on an 80 GB A100 GPU.

DNNs, model states dominate overall memory consumption and can occupy up to 87.5% of GPU memory even when using low-precision (16-bit) training [25].

Memory wall and offloading trade-offs: The GPU memory bottleneck has motivated a range of offloading techniques that increase model scale by transferring data to CPU RAM or secondary storage [21], [26], [27]. However, this expansion often degrades performance because offloaded data travels over links with far lower bandwidth and higher latency than on-GPU memory access [28]. Fig. 1 shows this trade-off by comparing the training throughput of several offloading-based systems against a pure-GPU baseline (Megatron-LM [29]).

In these systems, ZeRO-Offload [8] and ZeRO-Infinity (CPU/RAM) [23] shard and offload optimizer states and model partitions to CPU memory: ZeRO-Offload trains a ∼15B model but sustains only ∼16% of the Megatron-LM baseline throughput, and ZeRO-Infinity (CPU/RAM) scales to ∼51B at just ∼3% of the baseline, because the growing offloaded state exposes PCIe transfers and CPU-side optimization that computation cannot fully hide. A more aggressive variant, ZeRO-Infinity (NVMe) [23], increases the maximum trainable model size by approximately 29× (leveraging GPU, CPU, and secondary storage), but at a severe *practicality* penalty: throughput degrades by more than 900× relative to the GPU-only baseline. Simpler layer-serial schemes such as L2L [22], which load each layer in sequence with no overlap between computation and transfer, leave the GPU persistently idle and exemplify the same capacity-for-throughput trade-off. Overall, these results indicate that while offloading addresses the *can it run* question for very large models, it often creates a new *is it practical* challenge: current methods enable training larger models but at the cost of drastically slower training. This highlights the need for techniques that hide or reduce offloading overhead so that scaling model size does not substantially impair throughput.

Underutilization of GPU/CPU resources: Beyond the inefficiencies introduced by non-optimized offloading, a more nuanced problem arises: extensive underutilization of the underlying hardware allocated for training. Ideally, a compute-intensive task like DNN training would fully utilize both GPU and CPU resources. However, our profiling shows the opposite: during typical large-scale training, a substantial portion of GPU and CPU cycles remain idle. In a case study profiling a Transformer model [30] (Llama 4 Scout [31]) deployed in a production cluster, we measured an average SM occupancy of only ∼18%,

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

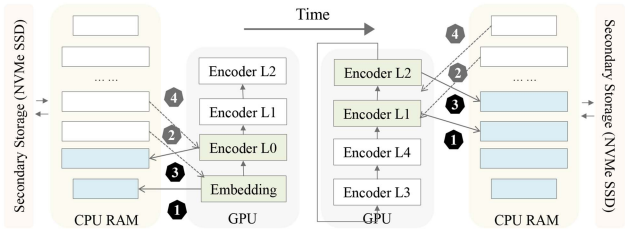


Fig. 2. Dynamic model state offloading in SYNERGYSCALE. SYNERGYSCALE keeps only a subset of DNN layers in GPU memory, moving finished layer states to CPU RAM. Actions ❶ and ❸ transfer layers already processed in FP and BP from GPU to CPU, while ❷ and ❹ concurrently prefetch upcoming layers from CPU RAM back into GPU memory.

a warp execution efficiency of ∼60%, and effectively ∼0% concurrent kernel execution, indicating that the GPU's compute units were frequently idle between operations. At the same time, host CPU utilization averaged just ∼4%.

These measurements reveal considerable unused capacity for concurrency. Contributing factors include pipeline bubbles (e.g., gaps between forward and backward passes or across distributed training stages) and the default single-stream execution model in frameworks such as PyTorch [32], which issues GPU workloads sequentially. Consequently, modern training pipelines often underuse available hardware: *both the GPU's parallel execution resources and the CPU's processing capabilities are underexploited*. Exploiting these idle periods with additional parallel work could substantially increase overall training throughput.

Opportunity: The challenges described above are fundamentally interconnected. Offloading-based approaches release GPU memory and permit larger models, but they also introduce idle periods due to data transfers and CPU-side computation. At the same time, low hardware utilization reveals unused capacity for additional parallel work. This observation motivates a key idea: *if the idle time induced by offloading can overlap with useful computation*, we can simultaneously expand model scale and enhance efficiency. SYNERGYSCALE operationalizes this idea by co-designing memory offloading and task parallelism, converting offloading overhead into productive computation, and jointly overcoming the memory and utilization walls.

III. SYSTEM ARCHITECTURE

SYNERGYSCALE is designed as an integrated framework that jointly optimizes memory management and task execution. As shown in Fig. 2, SYNERGYSCALE comprises three tightly coupled components: (1) a *dynamic offloading* module that orchestrates data movement across the GPU/CPU/storage hierarchy, (2) a *multi-streamed task partitioning* module that drives high hardware utilization, and (3) a central *co-optimization* engine that adaptively configures and coordinates these two modules.

A. Overview

Listing 1 provides a simplified example of using SYNERGYSCALE with PyTorch for DNN training. In this example, the user wraps the model and optimizer with the APIs of SYNERGYSCALE, specifying the available hardware resources

```python
import torch
import synergyscale  # API wrapper

# Setup a distributed environment
...

# Define the model and optimizer
model = TransformerModel()
optimizer = torch.optim.Adam(model.parameters()
    )

# Initialize synergyscale with resource
    constraints
resource_config = {
    'gpu_mem': '80GB',
    'cpu_mem': '512GB', # '1024GB'
    'local_storage': '1024GB',
    'storage_path': '/mnt/nvme/...'
}

# Wrap with runtime
synergy_model, synergy_optimizer = synergyscale
    .initialize(
    model, optimizer, resource_config
)

# Standard training loop
for batch in dataloader:
    inputs, labels = batch
    outputs = synergy_model(inputs)
    loss = loss_fn(outputs, labels)
    loss.backward()
    synergy_optimizer.step()
    synergy_optimizer.zero_grad()
```

Listing 1. Using SYNERGYSCALE with PyTorch.

(e.g., GPU memory, CPU memory, and storage limits). The framework then automatically manages memory offloading and parallel execution behind the scenes.

Offline processing: During the offline processing stage, SYNERGYSCALE first profiles the target DNN model to characterize its architecture and runtime behavior. A lightweight profiler executes a small number of training iterations or uses analytical estimates when appropriate (detailed in Section III-B) to collect key metrics, such as the computation time per layer and the memory usage. This process produces a layer-wise profile that captures GPU memory requirements, compute costs, and estimated data transfer overheads. These measurements provide the basis for reasoning about the trade-offs between model offloading and parallel execution.

Based on the profile results, the *Co-optimization Engine* (Section III-D) formulates a joint configuration problem. It searches for a high-throughput configuration pair (m, N), where m denotes the number of model layers retained in GPU memory at a time (i.e., the *GPU working window size* used by the offloading module), and N represents the number of concurrent execution streams employed by the task partitioning module. These parameters are tightly coupled because they compete for the same limited GPU memory. Increasing m reduces the memory available for parallel micro-batches, thereby limiting N, while decreasing m allows more streams but may increase offloading latency. Using an analytical performance model from the offloading module together with profiled layer execution times and resource

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

requirements, the engine estimates end-to-end throughput for different (m, N) configurations. It then performs a guided search to identify a configuration that maximizes predicted throughput under the GPU memory capacity constraint, yielding a high-quality, near-optimal balance between memory usage and parallelism (we detail the search and its cost in Section III-D).

Online training: With the selected configuration, SYNERGYSCALE proceeds to online training. The *Dynamic Offloading Module* (Section III-B) manages the movement of model data, including layer-wise parameters, gradients, and optimizer states, between GPU and CPU memory (and disk if necessary) according to a dynamic working-window scheme (Section III-B). In parallel, the *Task Partitioning Module* (Section III-C) schedules micro-batch execution across multiple CUDA streams based on the configured degree of parallelism N. During each training iteration, layers are prefetched and evicted from GPU memory on demand, while multiple micro-batches execute concurrently when resources permit. The SYNERGYSCALE runtime coordinates these activities so that they complement rather than interfere with each other. For example, data transfers are overlapped with computation, and alternative micro-batches are scheduled when execution would otherwise stall due to synchronization or data movement. Together, these mechanisms produce a jointly optimized execution schedule that maintains high GPU utilization while respecting memory constraints.

Overall, this architecture jointly reduces GPU memory usage and improves hardware throughput. By co-designing offloading and parallel execution, SYNERGYSCALE avoids the inefficiencies that arise when these mechanisms are optimized in isolation.

B. Dynamic Offloading

The Dynamic Offloading Module mitigates GPU memory pressure by moving model states across a heterogeneous memory hierarchy, including GPU device memory, CPU host memory, and optional secondary storage (e.g., NVMe SSDs). Building on StrongHold's offloading strategy [21], SYNERGYSCALE extends it with three key capabilities: (1) *latency-aware offloading* that maximally overlaps data movement with GPU computation, (2) *cache-aware scheduling* that exploits temporal locality across iterations, and (3) *multi-tier I/O optimization* that scales beyond host memory capacity.

Dynamic working window: At the core of SYNERGYSCALE is a *dynamic working window*. Rather than keeping all model layers in GPU memory, SYNERGYSCALE maintains only a sliding window of m layers on the GPU at any time. As training proceeds through the network in the forward and backward passes, the window slides across the model: layers entering the window are asynchronously prefetched into GPU memory, while layers leaving the window are evicted back to host memory. This design ensures that GPU memory stores only the *active* subset of layers required by the current computation, substantially reducing the peak device memory footprint.

SYNERGYSCALE can be integrated with mainstream deep learning frameworks (e.g., TensorFlow [33], PyTorch [32], and Megatron [29]) by inserting lightweight hooks at key points in the forward and backward execution of each layer.

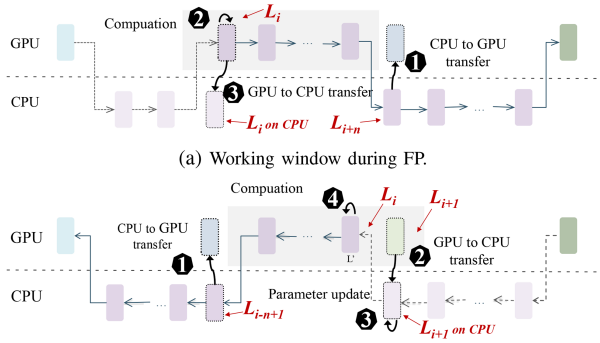


Fig. 3. Dynamic offloading in SYNERGYSCALE during forward and backward propagation, illustrated using a Transformer-based model. Here, the computation is at layer L_i. SYNERGYSCALE retains only the first and last layers (embedding and pooling) in GPU memory to minimize initialization overhead.

i) Forward propagation: Before executing layer L_i, a *pre-forward* hook asynchronously preloads the next layer's weights into GPU memory. The current layer then executes on the GPU. After the forward computation completes, a *post-forward* hook asynchronously evicts the layer's weights and any intermediate activations that are no longer needed back to CPU memory, freeing device memory for subsequent layers (Fig. 3(a)).

ii) Backward propagation: During backpropagation, a *pre-backward* hook restores the preceding layer's weights to the GPU just before they are needed for gradient computation. After the layer's backward pass completes, a *post-backward* hook transfers its gradients (and weights if required) back to the CPU (Fig. 3(b)).

This pipelined prefetch-and-evict scheme hides transfer latency by overlapping communication with computation. Intuitively, if the window size m is large enough such that computing the m layers takes longer than swapping a single layer, then the GPU can remain utilized without stalling on data transfers. By adjusting the window size and transfer schedule at runtime, SYNERGYSCALE adapts to different model sizes and system bandwidth. Conceptually, the module treats GPU memory as a cache for model layers, using predictive prefetching and eviction to mask I/O latency.

Analytical performance model: The window size m is critical for performance. A small window may not provide enough computation to hide transfer latency, leading to GPU stalls. An overly large window wastes GPU memory that could otherwise be used for activations, communication buffers, or larger batch sizes. To automate this choice, SYNERGYSCALE employs an analytical model that, after a short warm-up profiling phase, selects the smallest m that satisfies two constraints for both the forward and backward passes.

i) Latency-hiding constraint: The aggregate compute time of the m layers inside the window must be no less than the time required to transfer the next layer, so that prefetching completes before the GPU needs it. Formally, for the forward pass:

$$\sum_{i=0}^{m-1} t_i^{\mathrm{fp}} \geq t_j^{\mathrm{c2g}}, \tag{1}$$

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

and for the backward pass:

$$\sum_{i=0}^{m-1} t_i^{\text{bp}} \geq t_j^{\text{g2c}}, \tag{2}$$

where t_i^{fp} and t_i^{bp} are the computation times of layer i in forward and backward propagation, and t_j^{c2g} and t_j^{g2c} are the CPU-to-GPU and GPU-to-CPU transfer times for the next layer j.

ii) Memory-capacity constraint: The memory footprint of the $\mathtt{m}$ layers in the window, plus the buffer reserved to prefetch the next layer, must not exceed the available GPU memory S_{avail}. For the forward pass:

$$\sum_{i=0}^{m-1} S_i^{\text{fp}} + S_j^{\text{fp}} \leq S_{\text{avail}}, \tag{3}$$

and for the backward pass:

$$\sum_{i=0}^{m-1} S_i^{\text{bp}} \leq S_{\text{avail}}, \tag{4}$$

where S_i^{fp} and S_i^{bp} denote the memory footprints of layer i in the forward and backward passes (including parameters and required activations), and S_j^{fp} is the footprint of the next layer j to be prefetched.

By selecting the smallest $\mathtt{m}$ that satisfies both constraints, SYNERGYSCALE minimizes GPU memory usage while maximizing computation–communication overlap.

The per-layer times t_i^{fp}, t_i^{bp}, t_j^{c2g}, and t_j^{g2c} are treated as constants in (1) to (4), but they are *measured by profiling*, not assumed. Critically, they are profiled under the same multi-stream concurrency and overlapping-DMA conditions used at runtime, so the resource contention introduced by executing $\mathtt{N}$ concurrent micro-batches (for shared SMs, L2 cache, and HBM bandwidth) and by overlapping PCIe transfers with compute is already reflected in these effective times rather than idealized away. The engine therefore reasons about contention-aware quantities, and a runtime safeguard (Section III-C) prevents it from operating in regimes where contention would invalidate this assumption.

Multi-tier offloading and I/O concurrency: To scale beyond host memory capacity, SYNERGYSCALE supports multi-tier offloading to additional storage, such as local NVMe SSDs or a remote parallel file system (PFS). When CPU DRAM is insufficient, SYNERGYSCALE spills overflow states to secondary storage and enables *multi-path offloading*, distributing different model partitions across multiple devices in parallel. The offloading engine assigns data to tiers based on their capacity and throughput, while a concurrency control mechanism coordinates asynchronous I/O to reduce contention and improve bandwidth utilization. By combining node-local NVMe and remote PFS, SYNERGYSCALE increases aggregate offload throughput and avoids bottlenecks on any single I/O path, enabling training even when the model exceeds host memory.

Cache-aware scheduling and mixed-precision updates: The offloading module also reduces redundant transfers across iterations. In standard training, layers are processed in a fixed order, which can cause recently accessed states to be evicted and then reloaded shortly afterwards. Unlike StrongHold [21] (the early implementation of SYNERGYSCALE), SYNERGYSCALE mitigates this issue by alternating the processing order of model partitions across iterations to exploit temporal locality. For example, if a subset of layers remains resident in host memory at the end of an iteration, the next iteration can start from those layers (i.e., reversing the partition order) so that more data is reused from fast memory. In addition, SYNERGYSCALE reduces PCIe traffic in mixed-precision training via *delayed gradient conversion*, a feature not offered by StrongHold. In practice, gradients are typically produced in FP16/BF16 but are often converted to FP32 on the GPU before being offloaded for the optimizer update. Instead, SYNERGYSCALE transfers gradients to the CPU in their original low-precision format and performs the FP32 conversion on the host just before the optimizer step. This avoids transferring expanded FP32 gradients over PCIe, reducing gradient transfer volume by up to $2\times$ in practice. The conversion overhead is small on modern CPUs and is amortized over the update computation. Together, cache-aware scheduling and delayed conversion complement the latency-hiding design of SYNERGYSCALE and further reduce offloading overhead.

C. Multi-Streamed Task Partitioning

While the offloading module mitigates GPU memory pressure, the *Task Partitioning* module improves GPU underutilization. Its objective is to improve GPU compute unit occupancy and eliminate idle bubbles identified by our profiling. It does so through a novel, fine-grained form of intra-device data parallelism, which reconceptualizes data parallelism within a single GPU.

Fine-grained parallelism: We use the following terminology consistently. A *mini-batch* is the set of $\mathtt{S}$ samples that produces one optimizer update (the conventional unit of stochastic gradient descent). A *micro-batch* is a disjoint subset of a mini-batch; the $\mathtt{N}$ micro-batches of a mini-batch are processed concurrently but contribute to the *same* single update. In conventional training, a mini-batch of $\mathtt{S}$ input samples is processed as a single unit.

In SYNERGYSCALE's task-partitioning module, we instead apply an unbatching strategy: each mini-batch is divided into $\mathtt{N}$ smaller micro-batches that execute in parallel. For instance, with $\mathtt{S{=}128}$ and $\mathtt{N{=}4}$, rather than one stream handling $\mathtt{128}$ samples, $\mathtt{4}$ concurrent streams each process $\mathtt{32}$ samples. These micro-batches jointly realize a single mini-batch update: they use the same model parameters, and their gradients are summed before one optimizer step. We avoid replicating the model on the GPU for each micro-batch; all streams share a single model instance and accumulate gradients into a common buffer.

SYNERGYSCALE's runtime dispatches micro-batches to the GPU using multiple CUDA streams. Standard frameworks typically rely on a single stream, enforcing strict sequential kernel execution. Instead, we construct a custom *stream pool* and map each micro-batch to its own non-blocking stream. This enables the GPU scheduler to issue kernels from different micro-batches concurrently whenever resources are idle, allowing independent operations (e.g., matrix multiplications from separate micro-batches) to overlap on the GPU as long as sufficient SMs are

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

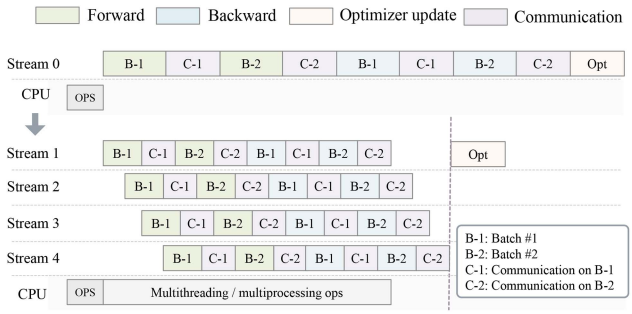


Fig. 4. The workflow of multi-streamed execution.

available. With many micro-batches in flight, some can compute while others wait on data transfers or synchronizations, so the GPU is rarely stalled across all activities.

Overall, multi-stream execution proceeds in three stages. First, a full batch is split into N micro-batches. Second, the micro-batches are dispatched concurrently across N CUDA streams rather than executed sequentially. Finally, the forward and backward passes for all micro-batches run on a single model instance, their gradients are aggregated into one update, and a single synchronized optimizer step is applied after all micro-batches finish. This preserves full mini-batch training semantics (e.g., normalization statistics and a single update per mini-batch), effectively emulating data parallelism on a single GPU without replicating the model.

Benefits of overlapping: The main advantage of this method is that it overlaps work that would otherwise run sequentially, improving the utilization of both the GPU and the CPU. For example, the forward pass of one micro-batch can run concurrently with another micro-batch's all-reduce communication or CPU-side preprocessing. This fine-grained parallelism fills otherwise idle periods in the pipeline, keeping GPU cores active and masking latency. Conceptually, as shown in Fig. 4, the operation timeline becomes tightly packed: when one stream is stalled on data or synchronization, another stream can typically dispatch work to any available SMs. This yields a substantial throughput gain.

We also apply a lightweight runtime safeguard: when the GPU is already highly utilized, additional streams may add overhead without benefit. Consequently, SYNERGYSCALE tracks GPU utilization and only enables multi-stream parallelism when utilization is below a preset threshold, thus invoking parallelism opportunistically only when idle capacity is available.

D. Co-Optimization Engine

The co-optimization engine jointly configures dynamic offloading and multi-stream execution by selecting a pair (m, N), where m is the working window size (layers resident on the GPU) and N is the number of CUDA streams. It selects this pair with a *guided search* during the training warm-up stage, using a few profiling iterations. We emphasize that this search is a bounded heuristic, not a solver that certifies global optimality: it returns a high-quality configuration that is near-optimal with respect to our profiled performance model, and it can be re-run cheaply when training conditions change (e.g., batch size or multi-GPU scaling).

Parameters m and N are tightly coupled through GPU memory. Increasing m keeps more layers resident on the GPU, improving locality and overlapping more computation with PCIe transfers, but it also increases memory usage and limits the number of concurrent micro-batches, reducing the feasible N. Decreasing m frees memory for more streams, but increases per-iteration swapping and may expose transfer latency when computation is insufficient to hide communication. As a result, an overly large m reduces utilization by constraining N, while an overly small m can stall the GPU on offloaded data and reintroduce I/O bottlenecks.

During training warm-up, the engine combines the analytical model in Section III-B with lightweight profiling to estimate the throughput of candidate pairs (m, N) under the GPU memory constraint. The analytical model provides the minimum feasible m and predicts transfer overheads, while profiling captures the benefit of additional streams based on kernel behavior (e.g., memory- vs. compute-bound layers). The engine then searches for a small, guided set of configurations and selects the best-performing (m, N), so that offloading and multi-streaming reinforce rather than starve each other.

Search procedure and cost: Although m and N are discrete with bounded ranges, a full grid search of their product is wasteful for deep models. We instead exploit the *near-unimodal* structure of the objective: holding m fixed, increasing N helps until SM occupancy or memory bandwidth saturates; holding N fixed, increasing m helps until per-iteration transfer overhead dominates. The engine therefore performs a coordinate-style, expand-then-refine search. Starting from the minimum feasible m (given by the analytical model) and N=1, it expands N while the predicted throughput improves and the memory constraint holds, then expands m similarly, and finally refines locally around the best point. This visits only candidates $O(|m|+|N|)$ rather than the $O(|m| \cdot |N|)$ of an exhaustive grid, where $|m|$ and $|N|$ are the sizes of the two ranges; in our experiments, the engine evaluates on the order of ten configurations and converges within the warm-up phase, whose overhead is amortized over the full training run. The infeasible (out-of-memory) region is pruned analytically before any profiling, so that no wasted runs are launched. Because the objective is empirically near-unimodal but not provably convex, the result is a strong local optimum rather than a certified global one; Section V-E shows that it stays within a small margin of an exhaustive-search oracle while evaluating far fewer configurations.

IV. IMPLEMENTATION

We implemented SYNERGYSCALE on top of PyTorch [32], extending NVIDIA's Megatron-LM [29], which provides optimized transformer training. The prototype is implemented in Python for high-level logic and description, with performance-critical components in C++/CUDA for custom kernels, memory pinning, and related operations. Overall, we added about 8,000 lines of code to the Megatron-LM codebase. Its main mechanisms, such as offloading, streaming, and co-optimization, are

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

implemented as modular features that can be independently toggled for ablation and debugging.

Dynamic offloading: The feature was implemented by redesigning how model layers are allocated, executed, and moved across devices, integrating a working-window memory manager via custom PyTorch forward and backward hooks. GPU–CPU data transfers rely on asynchronous CUDA copies with pinned (page-locked) host memory and multiple non-blocking CUDA streams to overlap computation with prefetching and eviction. A CPU-resident Adam-style optimizer [24] was introduced so that all optimizer states and weight updates are handled on the CPU, avoiding repeated GPU–CPU transfers, while a heterogeneous communication layer coordinates these transfers efficiently.

Multi-streamed task partitioning: SYNERGYSCALE's task partitioning leverages PyTorch's CUDA stream support by dividing each batch into micro-batches, issuing them to a pool of streams, accumulating gradients in shared memory, and coordinating execution via CUDA events. We pre-allocate a fixed pool of `N` streams and, when the module is active, split each incoming batch during training into micro-batches, assigning each to a distinct stream. CUDA events enforce synchronization after all backward passes, guaranteeing that cross-stream gradient accumulation finishes before the weight update. Micro-batches execute concurrently on the GPU, while CPU tasks (e.g., data loading for upcoming batches) can proceed in parallel. A configurable GPU utilization threshold controls when to apply *unbatching:* if measured utilization is below the threshold (indicating idle capacity), we enable multi-stream execution; otherwise, we revert to single-stream for that step to prevent oversubscription. Our prototype embeds this logic directly into Megatron-LM's training loop, touching only the batch-handling code and leaving the model architecture unchanged.

Co-optimization engine: During the offline training warm-up phase, the co-optimization engine (Section III-D) profiles per-layer compute and memory behavior using Megatron-LM and measures effective PCIe bandwidth. The analytical throughput model (Section III-B) is then applied to evaluate candidate configurations and select parameters such as the working-window size, stream count, and micro-batch size. The offload engine targets heterogeneous systems by prioritizing CPU RAM and only using NVMe when necessary. During training, SYNERGYSCALE monitors GPU utilization via NVIDIA profiling APIs and enables or disables multi-stream execution accordingly.

Further optimizations. (i) Concurrent parameter update: The parameter update phase, normally executed on the GPU, is instead delegated to the host CPU. SYNERGYSCALE spawns multiple optimizer instances as parallel workers across CPU cores, enabling updates for offloaded layers to run concurrently with the GPU's backpropagation on other layers, thereby largely masking the cost of optimizer updates, as shown in Fig. 5. *(ii) Heterogeneous collective communications:* In distributed environments, SYNERGYSCALE augments low-level communication libraries (e.g., NVIDIA NCCL, Facebook Gloo) to enable concurrent collective primitives (e.g., all-reduce) over both CPU and GPU tensors, a requirement for parallel gradient synchronization across the full heterogeneous system. *(iii) Runtime memory management:* To mitigate the substantial overhead

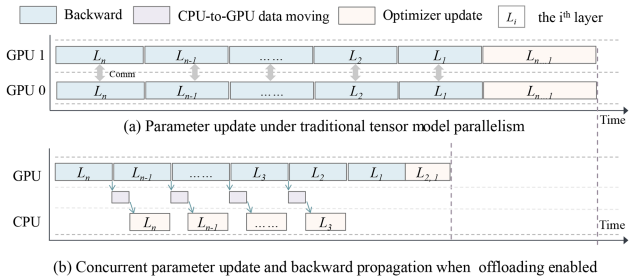


Fig. 5. Concurrent parameter update during backward propagation.

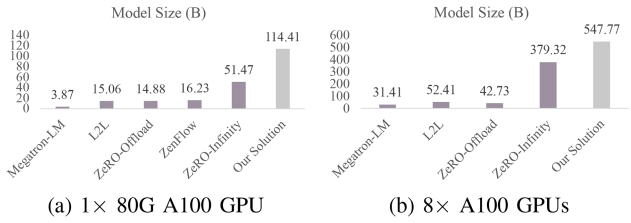


Fig. 6. The largest trainable model size with various solutions.

of repeated `cudaMalloc`/`cudaFree` operations, the module utilizes a custom user-level memory manager. It pre-allocates a compact, fixed-size pool of GPU buffers-sized only for the `m`, the working window, and relies on pinned (page-locked) host memory to enable efficient, asynchronous DMA transfers. This design markedly reduces memory management costs and lowers GPU memory usage relative to the caching strategies of conventional deep learning frameworks.

V. EVALUATION

We evaluate SYNERGYSCALE along several dimensions. First, we test how far it can scale the trainable model size beyond GPU memory and compare it to existing offloading systems (Section V-A). Second, we measure training throughput in single- and multi-GPU setups against baseline methods (Section V-B). Third, we examine how much the task partitioning module alone accelerates training when GPU memory is not the main bottleneck (Section V-C). Fourth, we run an ablation to quantify the benefit of jointly optimizing offloading and multi-streaming versus using each in isolation (Section V-D), and assess the algorithmic value of the co-optimization engine against static, greedy, manual, and exhaustive-search policies (Section V-E). We then explain why these optimizations preserve training semantics (Section V-F) and how the analytical model accounts for hardware contention (Section V-G). Finally, we evaluate throughput on MoE architectures (Section V-H).

Unless specified otherwise, we adopt the implementation of Megatron-LM transformer as the baseline in GPU training and compare it with the following offloading systems: L2L [22], ZeRO-Offload [8], ZeRO-Infinity [23] and ZenFlow [34]. We also compare SYNERGYSCALE with its predecessor, StrongHold [21], demonstrating the benefits of our new contributions to the conference paper (Section V-B).

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

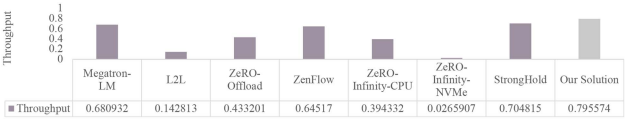


Fig. 7. The performance of SYNERGYSCALE.



(a) 1× 80G A100 GPU (b) 8× 80GB A100 GPUs

Fig. 8. Training performance for the largest trainable model in each solution.

The additional baseline, *ZenFlow* [34], is a stall-free offloading engine for LLM fine-tuning that extends ZeRO-Offload [8]: FP16 forward and backward passes run on the GPU while the full FP32 optimizer state resides in CPU DRAM. Unlike prior offloading systems, which apply a synchronous full-model optimizer step on the CPU each iteration and leave the GPU idle, ZenFlow decouples updates by importance, device, and time. Using a lightweight per-column gradient-norm proxy—motivated by the observation that the top 1% of gradients account for over 90% of the total gradient norm—it updates important gradients in-place on the GPU every step, while the remainder are asynchronously offloaded, accumulated on the CPU over a bounded number of steps, and applied intermittently. This bounded-asynchronous, double-buffered pipeline overlaps CPU updates and PCIe transfers with subsequent GPU computation. ZenFlow targets the *optimizer-stall regime* rather than maximum model size, making it complementary to SYNERGYSCALE.

A. Trainable Model Size

SYNERGYSCALE is designed to maximize the trainable model size within the available system memory. We quantify this as the largest parameter count that can be trained on a given setup before running out of system memory (CPU + GPU). On a single A100 GPU, vanilla Megatron-LM (without offloading) can handle transformers of up to 3.87B parameters. Offloading approaches expand this limit: L2L, ZeRO-Offload, and ZenFlow reach about 15∼16B parameters (∼ 3.8× over the baseline) by shifting parts of the model to CPU memory, and ZeRO-Infinity (using CPU RAM, not NVMe) scales further to ∼51.47B parameters (∼ 13× over the baseline). With SYNERGYSCALE, we train a 114.41B model on the same single GPU, saturating both GPU and CPU memory. This represents a ∼ 7.6× increase over L2L, ZeRO-Offload, and ZenFlow, and approximately 2.2× over the best ZeRO-Infinity (offloading to CPU RAM) result, shown in Fig. 6(a). Although ZeRO-Infinity can theoretically support an infinite number of parameters on a single GPU by offloading to NVMe, training becomes prohibitively slow, so we omit such configurations.

In a distributed setup, the benefits remain. Using 8 NVIDIA A100 GPUs in a DGX node, SYNERGYSCALE scales the trainable model size to 547.77B, surpassing the baseline offloading systems. Under the same conditions, ZeRO-Infinity (with CPU memory) reached about 379.32B parameters, and ZeRO-Offload reached about 42.73B before hitting memory or performance bottlenecks, as shown in Fig. 6(b). Thanks to its efficient working-window design, SYNERGYSCALE's offloading module greatly reduces per-GPU memory usage and utilizes nearly all aggregate GPU and CPU memory. This enables the training of models that would normally require far more GPUs or larger-memory GPUs. In general, SYNERGYSCALE extends the limit of trainable model size on constrained hardware by roughly 1.4× −12.81× over existing offloading methods on a single node, with comparable gains in multi-GPU settings.

B. Training Throughput

Increasing the size of the trainable model is advantageous only if the training performance remains within an acceptable range. To assess this, we evaluated the *training performance* for each system, using both its largest trainable model and a common reference model.

Throughput at the largest trainable model sizes: Even when each system was run with the largest model it could handle, SYNERGYSCALE consistently achieved higher training throughput. On a single A100 GPU, SYNERGYSCALE reached about 1.88× - 3.6× the throughput of competing offloading methods, although those methods trained smaller maximum models. Concretely, SYNERGYSCALE was about 1.88× faster than ZeRO-Infinity (offloading to CPU RAM) and roughly 3.6× faster than L2L, with comparable throughput to ZenFlow, since ZenFlow optimizes overall offloaded data volume by sacrificing unimportant gradients. Fig. 8 shows that our approach not only scales to larger models but does so efficiently: the combination of offloading and parallelism offsets the typical offload-induced slowdowns. A multi-GPU experiment on 8×A100 showed the same pattern: at matched model sizes, SYNERGYSCALE sustained up to 4.5× the throughput of ZeRO-Infinity and about 2.0× that of L2L (Fig. 8(b)), while also training a larger 547.77B model than any baseline (e.g., ZeRO-Infinity at 379.32B and ZeRO-Offload at 42.73B), owing to more effective overlap of communication and computation.

Throughput on equal-size models: To compare raw efficiency, we also evaluated all systems in a shared configuration that fits entirely on each device, as shown in Fig. 7: a 3.87B transformer trained on a single A100. In this setting, SYNERGYSCALE matches the GPU-only baseline (Megatron-LM without offloading) and even slightly exceeds it, reaching about 1.17× its throughput because multi-stream execution reclaims otherwise-idle GPU cycles. By contrast, ZeRO-Offload and L2L reach only about 63% and 20% of GPU-only throughput, respectively, while ZenFlow attains around 94%. For this model, ZeRO-Infinity with CPU offloading reaches roughly ∼57% of the baseline, and with NVMe it is orders of magnitude slower. This result is crucial: it shows that the overheads of SYNERGYSCALE

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

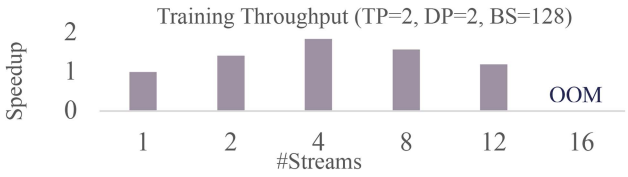


Fig. 9. The performance across different stream settings.

TABLE I
THE PERFORMANCE BREAKDOWN OF SYNERGYSCALE

Version	Throughput
w/o multi-streaming	1.94 tokens/s
w/o offloading	3.01 tokens/s
full (w/ joint optimization)	3.52 tokens/s

are so effectively overlapped that, at moderate scale, offloading imposes essentially no performance penalty, whereas previous approaches substantially degrade training speed even when they can handle the model size. By integrating the co-optimization engine, the proposed approach achieves an additional 12.8% improvement in performance over StrongHold [21]. Our analytically guided window sizing, together with asynchronous transfers, masks most of the PCIe latency, while multi-stream execution keeps the GPU highly utilized, preserving throughput. In effect, SYNERGYSCALE transforms the offload from a high-cost necessity into a low-overhead mechanism that supports much larger models with only a small loss in training speed.

C. Impact of Task Partitioning

To specifically assess the contribution of the multi-stream task partitioning module of SYNERGYSCALE, we conducted experiments with disabled memory offload, isolating the impact of intra-GPU parallelism. We selected models and batch sizes that fit entirely in GPU memory and compared training throughput with and without multi-stream execution (integrated into Megatron-LM). The experiments were carried out on a server training a 3B Transformer model [6] across various parallelism strategies and batch sizes. Throughout this section we use *data parallelism* (DP), which replicates the model and partitions the input batch across devices, and *tensor model parallelism* (TP), which partitions the layers/tensors of the model across devices; the notation `DP=`d (`TP=`p) denotes a degree-d (degree-p) configuration of the respective strategy.

The results in Fig. 10 show that the task-partitioning module alone can markedly increase hardware utilization and speed up training. In a data-parallel setup over four GPUs, configuring `N=2` streams per GPU achieved up to a 1.7× speedup at large batch sizes relative to the conventional single-stream baseline. Under certain settings, particularly with tensor model parallelism, which often leaves GPU cores stalled on communication, we observed even greater improvements, reaching up to 2.18× faster training iterations with multi-streaming enabled on each GPU.

As can be seen from Fig. 9, increasing the number of streams from 1 to 2 and 4 substantially reduces the iteration time (almost linearly at first) when the workload contains idle gaps. Beyond a threshold (`#streams`$\geq$`4` in our experiments), the benefits decrease as overhead and contention increase; however, even a modest increase to 2–4 streams effectively exploits otherwise-idle execution capacity. The effect is stronger at larger batch sizes, which expose more internal parallelism. Furthermore, across different GPU interconnect bandwidths, shown in Fig. 11, including throttled scenarios, multi-stream execution alleviates performance degradation by overlapping communication with computation. Overall, these results demonstrate that fine-grained micro-batch parallelism is highly effective for improving utilization and substantially accelerating training, independent of any offloading mechanisms.

In our experiments, multi-stream execution delivered up to a 2.18× increase in training throughput for a fixed model on a single node, compared to conventional single-stream execution. Benefits are largest at higher batch sizes and in scenarios such as multi-GPU data-parallel training, where GPUs would otherwise stall on communication, as our approach turns these idle periods into productive work on other micro-batches.

D. Joint Optimization Vs. Individual Techniques

Finally, we quantify the advantage of SYNERGYSCALE's co-optimization strategy by comparing it against strategies that employ only offloading or only multi-stream parallelism. We use a setup where both memory and computation are limiting factors (training a 3.87B-parameter model on a single 80 GB A100 GPU), and measure throughput for three configurations: (i) offloading only (with no multi-streaming), (ii) multi-streaming only (with no offloading, using an in-memory model), and (iii) SYNERGYSCALE (both offloading and multi-streaming enabled, with our joint optimization). The offloading-only configuration was tuned with the best measured window `m`, and the multi-stream-only configuration was run with an equivalent total batch size divided into micro-batches.

The results show that SYNERGYSCALE clearly outperforms single-technique variants. For this 3.87B model, offloading-only achieved 1.94 tokens/s, multi-streaming-only reached 3.01 tokens/s, while SYNERGYSCALE reached 3.52 tokens/s. In other words, SYNERGYSCALE delivered about a 1.81× speedup over the offloading-only baseline and about a 1.16× speedup over multi-streaming alone. This demonstrates that the benefits of offloading and parallelism are complementary and, when properly combined, yield greater gains than either in isolation. Offloading extends model capacity but would incur stalls without parallelism; multi-streaming boosts utilization but cannot train the model at all without offloading due to memory limits. SYNERGYSCALE's co-optimization strategy unlocks the full potential by simultaneously addressing both constraints. These findings reinforce our design motivation that coordination of memory management with computation scheduling is essential for maximum training performance in large models.

E. Impact of the Co-Optimization Engine

Table I shows that combining offloading with multi-streaming outperforms using either alone. In this experiment, we quantify the benefits provided by the co-optimization engine (Section III-D). Table II compares the configuration chosen by

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

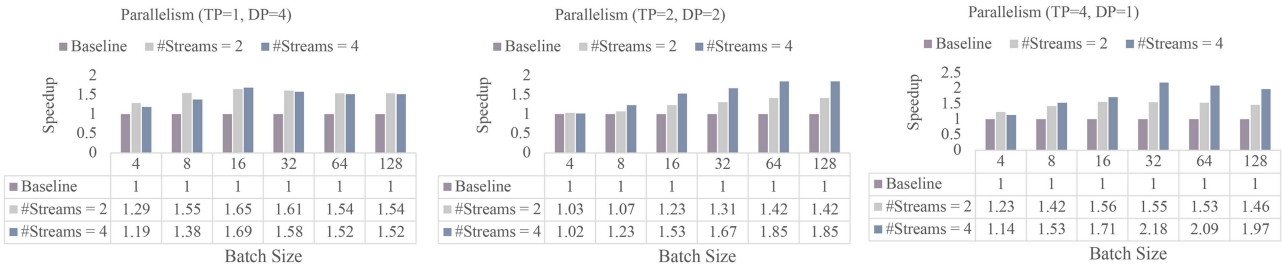


Fig. 10. Performance of multi-stream task partitioning with data and tensor model parallelism. (DP=2 means the degree of data parallelism is 2; TP=4 means the degree of tensor model parallelism is 4.).

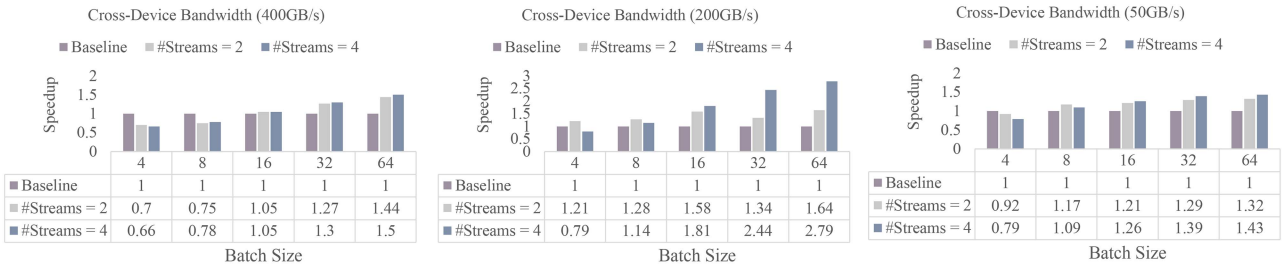


Fig. 11. Multi-stream task partitioning performance under varying cross-device bandwidth.

TABLE II
THE CO-OPTIMIZATION ENGINE ON A SINGLE A100: THROUGHPUT OF THE ENGINE-CHOSEN (m,N) COMPARED WITH STATIC, GREEDY, MANUAL, AND EXHAUSTIVE-SEARCH STRATEGIES. *#CFG* DENOTES HOW MANY (m,N) CONFIGURATIONS EACH STRATEGY EVALUATES.

Policy	(m, N)	#Cfg	Tput (tok/s)	% oracle
Static-default (N=1)	(8, 1)	1	1.94	54%
Greedy-streams (min m)	(2, 8)	1	2.61	73%
Manual (expert-tuned)	(6, 2)	4	3.19	89%
SYNERGYSCALE engine	(8, 2)	11	**3.52**	98%
Exhaustive oracle	(8, 3)	30	3.58	100%

the engine to four reference policies on a single-A100 setup: a *static-default* policy (a fixed window with single-stream execution), a *greedy-streams* policy (minimal window to maximize stream count), an expert-designed *manual* configuration, and an *exhaustive search oracle*. The static-default and engine entries match the *w/o multi-streaming* and *full* points in Table I; for the other policies, we fix (m,N) per their heuristic and report the resulting throughput on the same model.

The configuration given by the co-optimization engine outperforms the baseline schemes. The static baseline, offloaded with a static window and a single execution stream (N=1), delivers a throughput of 1.94 tokens/s, while the engine's selected (m,N) reaches 3.52 tokens/s. This corresponds to a 1.81× improvement in throughput, achieved by selecting a better operating point within the GPU's existing memory budget. The engine also surpasses the greedy and manual policies, by dynamically reducing the window (incurring offload stalls) and increasing it (depriving the streams of work).

F. Training Convergence and Numerical Fidelity

A primary design goal of SYNERGYSCALE is to accelerate training *without* changing the optimizer semantics. With SYNERGYSCALE, multi-stream task partitioning divides each mini-batch into N micro-batches that share a *single* set of model weights. Their gradients are accumulated in a common buffer, followed by a *single*, synchronized optimizer step (Section III-C). The resulting gradient is equivalent to that of single-stream mini-batch training, except for differences in the order in which the per-micro-batch contributions are summed. Such differences arise from floating-point non-associativity and are common in data-parallel training. Similarly, *delayed gradient conversion* (Section III-B) only changes where and when the exact FP16/BF16→FP32 cast is performed. Instead of converting the gradients on the GPU before transfer, SYNERGYSCALE performs the conversion on the host immediately before the optimizer update. No gradient value is quantized, approximated, or discarded. Unlike asynchronous-update schemes such as ZenFlow [34], SYNERGYSCALE introduces neither gradient staleness nor importance-based truncation. Every parameter is updated at each training step using the complete accumulated gradient. This behavior is also consistent with StrongHold [21], the offloading foundation on which SYNERGYSCALE builds, which has been shown to avoid stale updates and accuracy loss when training models with hundreds of billions of parameters in production. Because both new mechanisms preserve the original update rule by construction, training under SYNERGYSCALE follows the loss trajectory of unmodified Megatron-LM, up to the usual floating-point variation caused by parallel gradient accumulation.

G. Analytical Model Under Contention

The latency-hiding and memory-capacity constraints in (1) to (4) take the per-layer computation and transfer times as inputs. These values are obtained through *profiling*, rather than assumed. Specifically, they are measured under the same multi-stream concurrency and overlapping-DMA conditions

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

used at runtime. The measured effective times therefore capture contention for shared SMs, the L2 cache, and HBM bandwidth, as well as interference between PCIe transfers and GPU computation, rather than idealizing these effects away. The runtime utilization safeguard (Section III-C) further prevents the system from operating in regimes where additional streams would make contention the dominant factor. Consequently, (1) to (4) are used only in operating regions where they remain predictive. This profiling-based overlap model is supported by prior evidence from real hardware. The detailed computation and offloading traces reported by StrongHold [21] show that CPU–GPU transfers can be hidden by computation once the latency-hiding and memory-capacity constraints are satisfied, consistent with the predictions of (1) to (4).

H. Support for Mixture-of-Experts Models

Beyond dense Transformers, SYNERGYSCALE also supports Mixture-of-Experts (MoE) architectures, which place particularly high demands on memory offloading. An MoE layer replaces the single feed-forward network of a dense Transformer with a set of E expert networks. Because feed-forward parameters account for a large fraction of each layer, this design substantially increases the total model size.

During *training*, routing decisions vary across tokens and iterations. With large mini-batches and load-balanced routing, many, and potentially all, experts may receive tokens and gradients in a single iteration. As a result, the system must provision and update the full expert set, rather than only the small subset used by an individual inference request. At the same time, expert parameters must be brought into the GPU working window whenever tokens are routed to them. This makes MoE training a demanding test of hierarchical offloading.

SYNERGYSCALE supports MoE models by treating each expert as an independently offloadable unit. It prefetches upcoming experts while the current experts are being evaluated and overlaps expert transfers with attention and router computation through multi-stream execution. As a result, the complete expert set can reside in host memory while only the experts needed by the current computation occupy the GPU working window. The working-window scheme in Section III-B depends on the parameter and activation footprints of each computational unit rather than on whether the parameters belong to one dense matrix or several experts. Consequently, an MoE model is subject to the same total-parameter memory budget as a dense Transformer of comparable size. The capacity results in Section V-A extend naturally to MoE training: SYNERGYSCALE can train an MoE model whose total parameter count reaches the single-A100 memory limit, closely matching the 114.41B maximum observed for dense models.

Table III reports two representative Megatron-LM MoE configurations. On a single A100, SYNERGYSCALE trains MoE-A, a 47.5B-parameter model with 8 experts, at 0.29 tokens/s. It also trains MoE-B, a 115.7B-parameter model with 16 experts, at 0.12 tokens/s. Because all experts participate in routing, gradient computation, and optimizer updates over the course of training, these configurations require movement and management of a model state comparable to that of similarly sized dense Transformers. The working-window offloading and multi-stream execution mechanisms overlap this movement with computation, as for dense layers. These results show that MoE training is a natural use case for SYNERGYSCALE. The expert set can increase the total model size far beyond GPU memory, while the computation performed per token remains limited to the selected experts. Hierarchical offloading enables SYNERGYSCALE to exploit sparse computation without requiring the entire expert set to reside on the GPU.

TABLE III
SYNERGYSCALE'S CAPACITY AND THROUGHPUT FOR MIXTURE-OF-EXPERTS MODELS ON A SINGLE 80,GB A100, WITH A HIDDEN SIZE OF 4096, AN FFN SIZE OF 14336, AND EITHER 8 (MOE-A) OR 16 (MOE-B) EXPERTS. TRAINING MUST PROVISION THE FULL EXPERT SET FOR ROUTING AND UPDATES; THEREFORE, THE TOTAL PARAMETER COUNT DETERMINES THE MEMORY REQUIREMENT.

Model	Layers	Experts	Total Params (B)	Throughput (tok/s)
MoE-A	32	8	47.5	0.29
MoE-B	40	16	115.7	0.12

Overall, SYNERGYSCALE enables substantially larger models to be trained on the same hardware while improving training efficiency. On a single GPU, it increases model capacity by up to 7.6× and throughput by up to 3.6× over other offloading systems. For models that fit within its supported GPU-only regime, it achieves approximately 1.17× the throughput of GPU-only training and substantially outperforms alternative offloading methods. The gains also extend to multi-GPU settings, where SYNERGYSCALE achieves up to 4.5× the throughput of ZeRO-Infinity at the same model sizes (Section V-B). In addition, its co-optimized design scales MoE training to approximately 115B parameters on a single A100. The results also demonstrate the importance of joint optimization. Neither offloading nor multi-stream execution alone matches their combined performance in SYNERGYSCALE: multi-stream execution provides an additional speedup of approximately 1.8× over offloading alone. Moreover, the co-optimization engine contributes a substantial part of the improvement beyond simply enabling both mechanisms (Section V-E). These gains are achieved without changing the optimizer semantics of the baseline training system.

VI. RELATED WORK

SYNERGYSCALE combines and extends multiple techniques, offering a holistic integration of techniques that were previously studied primarily in isolation.

Parallelism strategies: A range of parallelism strategies have been proposed for distributed DNN training [35], [36], [37]. These include data parallelism [9], [38], tensor parallelism [13], [39], and pipeline parallelism [11], [12], [40], [41], [42], [43], [44]. In data parallelism, an identical model replica is placed on each GPU; the training data set is partitioned between devices, and the gradients are synchronized at each iteration. Tensor parallelism instead decomposes the model (e.g., by layers or tensor shards) across GPUs so that each device stores only a subset of the parameters. Pipeline parallelism, used in systems such as GPipe [11] and PipeDream [12], segments the model into

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

stages mapped to different GPUs and processes micro-batches in a pipeline to keep all stages occupied. Although these schemes enable scaling to large models, they introduce nontrivial communication costs and remain constrained by per-device memory capacity.

Our approach is both orthogonal and complementary: SYNERGYSCALE's offloading can be integrated into data- and model-parallel setups to increase effective memory capacity, while its intra-device parallelism effectively introduces pipeline-like parallelism within a single GPU (across pipeline micro-batches). Previous work has explored using *multiple CUDA streams* and intra-GPU concurrency for other purposes, such as overlapping preprocessing with training or leveraging CUDA Graphs to run independent kernels. However, common frameworks do not automatically parallelize training workloads on a single GPU beyond the default hardware scheduling of a single stream. In contrast, SYNERGYSCALE's task-partitioning module explicitly *enables intra-GPU parallel execution* of training tasks, targeting a largely underexplored parallelism dimension in DNN training. Functionally, this behaves like data parallelism confined to one GPU, but without weight replication (preserving memory) and with communication–compute overlap similar to pipeline methods.

Memory-saving techniques: A substantial amount of previous research focuses on mitigating the GPU memory bottleneck. Activation checkpointing [45] exchanges additional computation for reduced memory usage by discarding intermediate activations in the forward pass and recomputing them in the backward pass (as in BERT [2] and similar models [6]), effectively saving memory at the expense of extra FLOPs. Compression and quantization-based methods reduce memory usage by storing parameters or gradients with lower precision [46], sometimes with a minor accuracy loss. The most relevant to our work are model offloading systems [47], [8], [23], [48]. ZeRO-Offload [8] transfers optimizer states (and occasionally gradients) to CPU memory to free GPU capacity, while ZeRO-Infinity [23] generalizes this to distribute all model states across GPU, CPU, and even NVMe, supporting extremely large models but incurring I/O latency. L2L (layer-to-layer) [22] targets Transformers specifically, offloading whole layers in sequence during training. MLP-Offload [28] extends DeepSpeed ZeRO-3's NVMe offloading [25] with a unified, bandwidth-proportional *multi-path* storage tier (striping offloaded optimizer state across node-local NVMe and a remote parallel filesystem) and optimizer-update I/O optimizations such as cache-friendly subgroup reordering and on-the-fly FP32 gradient conversion, reporting up to 2.5× faster iterations than ZeRO-3. However, its gains largely target the offloading *I/O* path and depend on HPC cluster storage (a fast parallel filesystem) for roughly half of the speedup. In contrast, SYNERGYSCALE introduces a dynamic latency-sensitive working window together with analytically tuned offload schedules *and* an intra-GPU multi-stream executor; the two directions are largely orthogonal and could in principle be composed. Whereas earlier offloading systems often incur substantial slowdowns (see Section V), SYNERGYSCALE's co-design with parallel execution effectively hides offloading costs. By more thoroughly overlapping communication with computation, our method delivers significantly higher large-scale throughput, rendering offloading delays nearly invisible.

Co-optimization of computation and memory: Several recent studies [49], [50] have examined the interaction between memory usage and computation scheduling. Systems, such as AdaScale [51], adapt batch sizes or recomputation rates at runtime to trade off throughput against memory limits. ZenFlow [34] eliminates GPU stalls caused by slow CPU optimizer updates by splitting gradients by importance: the most important gradients are updated immediately on the GPU, while the remaining gradients are accumulated on the CPU and applied *asynchronously* over a bounded multi-step window, overlapping the CPU optimizer step with GPU forward/backward computation. This trades update exactness for speed (introducing bounded staleness and gradient truncation), whereas SYNERGYSCALE overlaps computation and transfer *without* altering update semantics, and extends overlapping to the parallel execution of multiple compute tasks rather than simply hiding a single optimizer step. In particular, ZenFlow itself uses StrongHold [21] as a baseline and argues that its CPU optimizer-update phase is the limiting stage; ZenFlow and StrongHold-derived SYNERGYSCALE thus lie on the same lineage but attack different bottlenecks, i.e., optimizer-update stalls versus offload scheduling and intra-GPU micro-batch parallelism, and are largely complementary.

A parallel line of work specializes in offloading to particular hardware. SuperOffload [27] re-architects offloading around tightly coupled Grace-Hopper *superchips*, exploiting the 900 GB/s NVLink-C2C link and ARM CPU to speculatively overlap CPU optimizer steps with GPU backward computation; its core techniques are tuned to that interconnect and do not target commodity PCIe-attached GPUs. Aqua [26] accelerates LLM *inference* by offloading state into spare peer-GPU HBM over an NVLink/NVSwitch scale-up domain. Both are orthogonal to SYNERGYSCALE, which targets PCIe-attached A100 GPUs and a GPU/CPU/NVMe hierarchy; a like-for-like quantitative comparison is not meaningful because their headline results are inseparable from specialized interconnects or a different (inference) task.

Conceptually, SYNERGYSCALE performs a joint *optimization of scheduling and placement:* it simultaneously orchestrates multiple computation streams and decides which data to keep or fetch in GPU versus CPU memory. This integrated design underlies the high utilization and throughput that we achieve. To our knowledge, SYNERGYSCALE is the first system to explicitly *co-design a memory offloader with an intra-GPU parallel executor* and to demonstrate that such tight integration can outperform memory- or compute-focused techniques in isolation.

In summary, earlier work has proposed key components: memory offloading, distributed parallelism, and various scheduling techniques, but each solves only a subset of the challenge. The main contribution of SYNERGYSCALE is to integrate these elements into a unified framework that *simultaneously addresses memory scalability and utilization efficiency*, advancing practical and efficient training of billion-scale models on comparatively modest hardware.

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

VII. CONCLUSION AND FUTURE WORK

Training large-scale DNN models has two challenges: limited GPU memory and low hardware utilization. We presented SYNERGYSCALE, a software system that addresses both challenges through a synergistic design. Its dynamic *working window* enables training models far larger than GPU memory by proactively offloading layers between GPU and CPU memory. At the same time, the task partitioning module exploits the freed GPU memory and execution time to run multiple micro-batches concurrently, filling pipeline bubbles that constrain conventional training. These components reinforce each other: offloading creates capacity for parallelism, while parallel execution hides offloading latency.

Our evaluation shows that SYNERGYSCALE can support the training of a 114.41B-parameter Transformer on a single 80 GB A100 GPU and scale to 547.77B parameters on an 8-GPU cluster, exceeding the limits of existing systems, and that its design extends to MoE architectures. Our results show that jointly optimizing offloading and task partitioning achieves up to a 4.5× speedup over state-of-the-art offloading-only baselines while preserving the baseline optimizer semantics. By treating memory management as a source of parallelism, SYNERGYSCALE enables efficient, cost-effective training of large models.

REFERENCES

[1] Z. Liu et al., "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2021, pp. 10012–10022.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguist.: Hum. Lang. Technol.*, vol. 1, 2019, pp. 4171–4186.

[3] T. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 1877–1901.

[4] J. Kaplan et al., "Scaling laws for neural language models," 2020, *arXiv:2001.08361*.

[5] A. Yang et al., "Qwen3 technical report," 2025, *arXiv:2505.09388*.

[6] H. Touvron et al., "Llama: Open and efficient foundation language models," 2023, *arXiv:2302.13971*.

[7] A. Liu et al., "DeepSeek-V3 Technical Report," 2024, *arXiv:2412.19437*.

[8] J. Ren et al., "Zero-offload: Democratizing billion-scale model training," in *Proc. USENIX Annu. Tech. Conf.*, I. Calciu and G. Kuenning, Eds., 2021, pp. 551–564.

[9] S. Li et al., "Pytorch distributed: Experiences on accelerating data parallel training," 2020, *arXiv:2006.15704*.

[10] J. Dean et al., "Large scale distributed deep networks," *NeurIPS*, vol. 25, 2012.

[11] Y. Huang et al., "GPIPE: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 103–112.

[12] D. Narayanan et al., "Pipedream: Generalized pipeline parallelism for dnn training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 1–15.

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

[14] M. Wang, C.-C. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–17.

[15] A. Harlap et al., "Pipedream: Fast and efficient pipeline parallel DNN training," 2018, *arXiv:1806.03377*.

[16] H. Jin et al., "Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, pp. 37:1–37:26, 2018.

[17] M. Hildebrandet al., "AutoTM: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *Proc. 25th Int. Conf. Architect. Support Programm. Lang. Operating Syst.*, 2020, pp. 875–890.

[18] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the GPU memory limit via smart swapping," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, New York, NY, USA: Association for Computing Machinery, 2020, pp. 1341–1355, doi: 10.1145/3373376.3378530.

[19] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *Proc. IEEE Int. Symp. High- Perform. Comput. Archit.*, Seoul, South Korea, 2021, pp. 598–611.

[20] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?," in *Proc. ACM Conf. Fairness, Accountability, Transparency*, 2021, pp. 610–623.

[21] X. Sun et al., "Stronghold: Fast and affordable billion-scale deep learning model training," in *Proc. SC22: Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2022, pp. 1–17.

[22] B. Pudipeddiet al., "Training large neural networks with constant memory using a new execution algorithm," 2020, *arXiv:2002.05645*.

[23] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the GPU memory wall for extreme scale deep learning," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–14.

[24] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. ICLR*, 2015.

[25] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *Proc. SC20, Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–16.

[26] A. Vijaya Kumar, G. Antichi, and R. Singh, "Aqua: Network-accelerated memory offloading for LLMs in scale-up GPU domains," in *Proc. 30th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2025, vol. 2, pp. 48–62.

[27] X. Lian, M. Tanaka, O. Ruwase, and M. Zhang, "Superoffload: Unleashing the power of large-scale llm training on superchips," in *Proc. 31st ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2026, vol. 1, pp. 249–264.

[28] A. K. Maurya, M. M. Rafique, F. Cappello, and B. Nicolae, "Mlp-offload: Multi-level, multi-path offloading for LLM pre-training to break the gpu memory wall," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2025, pp. 1381–1394.

[29] Megatron-LM," 2021. [Online]. Available: https://github.com/NVIDIA/Megatron-LM

[30] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 6000–6010.

[31] Meta, "The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation," 2025. [Online]. Available: https://ai.meta.com/blog/llama-4-multimodal-intelligence/

[32] PyTorch, 2021. [Online]. Available: https://pytorch.org

[33] TensorFlow, 2021. [Online]. Available: https://www.tensorflow.org

[34] T. Lan et al., "Zenflow: Enabling stall-free offloading training via asynchronous updates," 2025, *arXiv:2505.12242*.

[35] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *CSUR*, vol. 52, no. 4, pp. 1–43, 2019.

[36] DeepSpeed, (n.d.). [Online]. Available: https://www.deepspeed.ai

[37] L. Lin et al., "AIACC-training: Optimizing distributed deep learning training through multi-streamed and concurrent gradient communications," in *Proc. 42nd IEEE Int. Conf. Distrib. Comput. Syst.*, 2022, pp. 853–863.

[38] Q. Xu and Y. You, "An efficient 2D method for training super-large deep learning models," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*), 2023, pp. 222–232.

[39] D. Narayanan et al., "Efficient large-scale language model training on GPU clusters using megatron-lm," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–15.

[40] S. Fanet al., "Dapple: A pipelined data parallel approach for training large models," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 431–445.

[41] A. Kosson, V. Chiley, A. Venigalla, J. Hestness, and U. Koster, "Pipelined backpropagation at scale: Training large models without batches," *Proc. Mach. Learn. Syst.*, vol. 3, pp. 479–501, 2021.

[42] C. He, S. Li, M. Soltanolkotabi, and S. Avestimehr, "PipeTransformer: Automated elastic pipelining for distributed training of transformers," 2021, *arXiv:2102.03161*.

[43] Z. Li et al., "TeraPipe: Token-level pipeline parallelism for training large-scale language models," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 6543–6552.

[44] D. Narayananet al., "Memory-efficient pipeline-parallel DNN training," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 7937–7947.

[45] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," 2016, *arXiv:1604.06174*.

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.

[46] P. Micikeviciuset al., "Mixed precision training," in *Proc. Int. Conf. Learn. Representations*, 2018.
[47] J. Linet al., "M6-10t: A Sharing-delinking paradigm for efficient multi-trillion parameter pretraining," 2021, *arXiv:2110.03888*.
[48] S. Singh and A. Bhatele, "AxoNN: An asynchronous, message-driven parallel framework for extreme-scale deep learning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2022, pp. 606–616.
[49] L. Wang et al., "Superneurons: Dynamic GPU memory management for training deep neural networks," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*2018, pp. 41–53.
[50] J. Thorpeet al., "Dorylus: Affordable, scalable, and accurate GNN training with distributed cpu servers and serverless threads," in *Proc. 15th USENIX Symp. Operating Syst. Des. Implementation*, 2021, pp. 495–514.
[51] T. Johnson, P. Agrawal, H. Gu, and C. Guestrin, "Adascale SGD: A user-friendly algorithm for distributed training," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 4911–4920.



Xiaoyang Sun (Member, IEEE) is currently a research fellow with the School of Computer Science, the University of Leeds, U.K., supported by funding from the Engineering and Physical Sciences Research Council. His research interests include memory safety, security, system software optimization, and cloud computing.



Jie Xu (Member, IEEE is currently a chair professor of the Distributed Systems and Services (DSS) Theme, with the School of Computer Science, the University of Leeds. He has worked in the field of distributed computing systems for more than thirty-five years. He has been the leader for a Research Peak of Excellence with Leeds, the Director of the EPSRC-funded White Rose Grid e-Science Centre.



Zheng Wang (Senior Member, IEEE) is currently a professor of intelligent software technology with the School of Computer Science, the University of Leeds. He is a Royal Society Industry fellow. His research interests include code optimization, parallelization, and systems security.

Authorized licensed use limited to: University of Leeds. Downloaded on September 14,2026 at 11:45:04 UTC from IEEE Xplore. Restrictions apply.