Efficient Tensor Offloading Based on CXL Memory Pool For Extreme Scale Deep Learning

Yue Ma, Dongwei Xu, Peng He, Hao Fu, Jing Ying, Sheng Zhang,
Fangming Liu, *Senior Member, IEEE,* Bowen Wang, Haiyuan Wan, and Zhirun Yue

***Abstract*—The exponential growth of deep learning models imposes severe memory constraints on GPUs, significantly increasing training costs. The prevailing solution for addressing memory constraints is tensor offloading, exemplified by ZeRO-Infinity, which leverages GPUs, CPUs, and NVMe SSDs to enable large-scale model training. However, ZeRO-Infinity faces significant performance bottlenecks due to memory access imbalance, coarse-grained tensor transfers, NVMe bandwidth and latency limits, and software complexity. Compute Express Link (CXL) emerges as a promising technology for building disaggregated memory pools, yet its integration into large-scale training remains underexplored.**

This paper introduces an efficient CXL memory pool into the system for tensor offloading and leveraging CXL protocol features for hardware acceleration. The proposed design incorporates NUMA-aware memory allocation and a Dynamic Adaptive Pipelining (DAP) strategy to enhance communication–computation overlap, along with a hybrid object-based memory management scheme to mitigate fragmentation and improve allocation efficiency. Experimental evaluation on an 8-GPU system with CXL Type-3 expansion cards demonstrates up to 72.7% throughput improvement, 62.9% latency reduction, and support for training 1.14× larger models compared with SSD-based ZeRO-Infinity. This is the first work to integrate CXL with ZeRO-Infinity for large-scale training, offering practical insights for future CXL-based heterogeneous systems.

***Index Terms*—Deep learning, tensor offloading, ZeRO-Infinity, CXL, CXL memory pool, hardware acceleration**

This work was supported in part by the Major Key Project of PCL (Grant PCL2024A06 and PCL2022A05). Additionally, it was supported by the Shenzhen Science and Technology Innovation Program under Grant RCJC20231211085918010.

Yue Ma is with Tsinghua Shenzhen International Graduate School (SIGS), Tsinghua University, and with Peng Cheng Laboratory, Shenzhen 518000, China (e-mail: m-y22@mails.tsinghua.edu.cn).

Dongwei Xu is with Tsinghua Shenzhen International Graduate School (SIGS), Tsinghua University, Shenzhen 518000, China (e-mail: xdw23@mails.tsinghua.edu.cn).

Peng He is with Tsinghua Shenzhen International Graduate School (SIGS), Tsinghua University, and with Peng Cheng Laboratory, Shenzhen 518000, China (e-mail: hep23@mails.tsinghua.edu.cn).

Hao Fu is with Tsinghua Shenzhen International Graduate School (SIGS), Tsinghua University, Shenzhen 518000, China (e-mail: fuh23@mails.tsinghua.edu.cn).

Ying Jing is with Tsinghua Shenzhen International Graduate School (SIGS), Tsinghua University, Shenzhen 518000, China (e-mail: jingy23@mails.tsinghua.edu.cn).

Sheng Zhang is with Tsinghua Shenzhen International Graduate School (SIGS), Tsinghua University, Shenzhen 518000, China (e-mail: zhang sh@mail.tsinghua.edu.cn).

Fangming Liu is with Peng Cheng Laboratory, and Huazhong University of Science and Technology, China (e-mail: fangminghk@gmail.com).

Bowen Wang, Haiyuan Wan and Zhirun Yue are with Tsinghua Shenzhen International Graduate School (SIGS), Tsinghua University, Shenzhen 518000, China (e-mail: wangbw23@mails.tsinghua.edu.cn).

Corresponding authors are Zhang Sheng and Fangming Liu.

I. Introduction

DEEP learning [1] models have experienced exponential growth, with state-of-the-art models reaching hundreds of billions, or even trillions [2], [3], of parameters. This exponential growth in parameters demands substantial computational resources, resulting in memory bottlenecks and increased communication overhead. Consequently, training such models requires costly multi-GPU, multi-node clusters, often inaccessible to many researchers. The current generation of GPU memory for regular training ranges from 16 GB to 80 GB, and the GPU memory wall problem [2], [4] is extremely prevalent. Researchers have explored alternative strategies, such as memory offloading, to mitigate the GPU memory wall.

Currently, achieving linear scalability of GPU memory is impractical [5]. Augmenting memory capacity generally results in a considerable increase in cost, especially for high-bandwidth memory (e.g., HBM), where the cost increases non-linearly. Moreover, as memory capacity increases, GPU power consumption and heat generation also escalate, posing significant challenges for thermal management in hardware design. Additionally, the expansion of high-speed memory complicates hardware architecture, imposes electrical constraints, and necessitates enhanced software support. Researchers have turned to alternative strategies, such as tensor offloading [6]–[8], to mitigate the GPU memory wall.

Tensor offloading is a cost-effective technique, especially when model memory requirements exceed system limits. It involves transferring tensor data from GPU memory to slower storage tiers, such as CPU memory or SSDs, during computation. The most prominent approach to model scaling is ZeRO-Infinity [2], [3], [9]–[11], which extends model memory capacity by offloading model parameters, optimizer states, and gradients to CPU memory and NVMe SSDs. While ZeRO-Infinity has demonstrated impressive scalability, it still faces performance bottlenecks. First, while NVMe SSDs provide high-capacity storage expansion, their bandwidth and latency restrict the efficiency of model parameter transfers and overall training speed [12], [13]. Second, coarse-grained tensor transfers, which typically involve transferring large chunks of model parameters at once, can result in unbalanced memory access and high data latency. Additionally, data transfer between GPUs and SSDs often involves replication of model parameters, and frequent replication operations not only increase transfer latency but also consume computational and storage resources. Finally, ZeRO-Infinity relies on a complex software stack [14], [15] to manage data transfers

across GPUs, CPUs, and storage media (e.g., NVMe SSDs), which not only complicates development and debugging but also introduces additional communication overhead, negatively impacting overall performance. These challenges are further exacerbated in multi-GPU systems, where the effects of Non-Uniform Memory Access (NUMA) add significant complexity.

These limitations underscore the need for a next-generation interconnect that combines high bandwidth, low latency, and native memory semantics to enable efficient large-scale offloading. Compute Express Link (CXL) [16]–[20] is an open, industry-standard interconnect built on PCIe, designed to provide coherent and efficient communication among CPUs, accelerators, and memory devices. A key feature of CXL is the memory pooling [21]–[23], which enables efficient utilization and flexible expansion of memory resources in heterogeneous computing environments [24]. These characteristics make CXL an attractive candidate for overcoming the memory bottlenecks of large-scale model training. Recent studies [4], [25]–[27] have explored the integration of CXL-extended memory with tensor offloading, primarily relying on CXL's cache-coherent features. However, these efforts are largely confined to simulation or analytical modeling, and the evaluated models are typically too small to capture realistic performance behavior. Moreover, a systematic approach to integrating CXL with ZeRO-Infinity remains absent.

In this paper, we present Tensor Offloading with CXL Pools (TOCP), a novel architecture that integrates CXL memory pools into ZeRO-Infinity to achieve high-bandwidth, low-latency memory scaling. In GPU–CPU–CXL heterogeneous systems, the SSD-based management and overlap strategies of ZeRO-Infinity are no longer effective. To address this, TOCP introduces fine-grained tensor transfers and NUMA-aware load balancing tailored to CXL hardware characteristics, improving memory utilization and data transfer efficiency. We further propose a Dynamic Adaptive Pipelining (DAP) technique that adjusts the prefetch window to maximize computation–communication overlap, while leveraging CXL.mem atomic operations to reduce synchronization overhead. Unlike prior simulation-based studies, TOCP is implemented and evaluated on real CXL Type-3 cards, demonstrating its potential to advance next-generation large-scale deep learning systems.

Our contributions are as follows:

TOCP Framework. We propose TOCP (Tensor Offloading with CXL Pools), the first framework that integrates CXL Type–3 memory expansion cards into ZeRO-Infinity, enabling high-bandwidth, low-latency memory scaling for large-scale training.

NUMA-Aware Optimization. We design a NUMA-aware memory allocation and load balancing strategy that mitigates cross-node contention and improves locality, significantly reducing access latency in heterogeneous GPU–CPU–CXL systems.

Fine-Grained Tensor Scheduling and Adaptive Pipelining. We introduce fine-grained tensor partitioning and a DAP mechanism to maximize communication–computation overlap, alleviating the bandwidth and latency bottlenecks of CXL.

Hardware–Software Co-Design and Evaluation. We develop a hybrid object-based memory allocator with atomic operation support, and evaluate TOCP on a real 8-GPU system with CXL Type-3 cards, demonstrating up to 72.7% throughput improvement, 62.9% latency reduction, and 1.14× larger model capacity compared with SSD-based ZeRO-Infinity. Our study demonstrates the real-hardware integration of CXL with ZeRO-Infinity, offering practical insights into hardware–software co-design and laying a scalable foundation for future CXL-enabled heterogeneous systems.

II. Background

ZeRO-Infinity represents the leading approach in tensor offloading technology, and its integration with CXL for enhanced performance is a natural progression. Building upon this foundation, our work further optimizes heterogeneous systems, expanding model training scalability and improving overall system performance.

A. ZeRO-Infinity and Tensor Offloading

ZeRO-Offload [3], [10], [11] extends GPU memory capacity by utilizing CPU DRAM, while ZeRO-Infinity further enhances scalability by offloading parameters, optimizer states, and gradients to NVMe SSDs. ZeRO-Infinity is one of the most advanced tensor offloading techniques. It enables the training of models with up to 32 trillion parameters on 32 NVIDIA V100 DGX-2 nodes (512 GPUs) [2], achieving unprecedented scalability. Building on ZeRO-3, it offloads parameters, optimizer states, and gradients to heterogeneous memory devices, while utilizing twin-offload and overlap-centric scheduling to reduce communication overhead. At model initialization, the parameter states, including FP16/FP32 parameters, optimizer states, and gradients, are maintained in rank distribution through the ZeRO-3 mechanism. These states are stored in the CPU/NVMe SSDs and loaded into memory only when needed for computation. At the start of each training round, the offload scheduler preloads the relevant parameters for the current training step (e.g., layers for the current batch) into the GPU, enabling forward propagation with the most up-to-date parameters. During backward propagation, gradients are generated on the GPU and stored in a gradient buffer. This buffer is periodically transferred to the CPU memory. The optimizer states (e.g., m, v in Adam) and gradients are no longer confined to GPU memory. The "`twin-offload`" mechanism allows the CPU and GPU to independently compute and update the optimizer states, loading only the relevant states for the parameters being updated. Once the update is complete, these states are offloaded, optimizing memory utilization and dynamically adapting to the training task. Complete the optimizer state update on CPU memory to generate new parameters, then transfer them from NVMe SSD to GPU memory for training. Fig. 1 shows the data flow and Twin-Offload of ZeRO-Infinity.

The ZeRO-Infinity method with SSD offload faces a significant challenge due to the long data transfer path. To address this, ZeRO-Infinity adopts an Overlap-Centric Design, which allows the GPU to perform computations (e.g., backpropagation) while asynchronously prefetching parameters for the

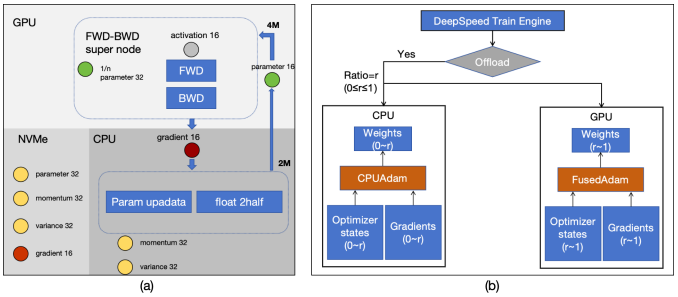

Fig. 1. (a) The dataflow of fully connected neural networks with M parameters. The specific storage locations for optimizer states and gradients can be adjusted as needed and (b) twin-offload.

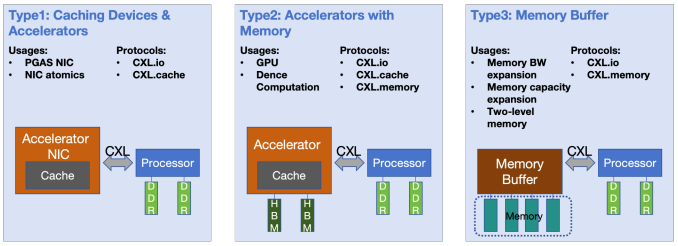

Fig. 2. Three CXL devices.

next layer and offloading computed parameters. Additionally, bandwidth-centric partitioning leverages the aggregated memory bandwidth of all parallel devices. However, the data transfer path is lengthy and constrained by SSD performance, resulting in a significant bottleneck in large-scale data transfer. Additionally, the system's complexity necessitates more meticulous software configuration and scheduling. As a result, while ZeRO-Infinity successfully breaks the GPU memory wall, its performance degrades sharply at extreme scale.

B. Compute Express Link (CXL)

CXL is an open, industry-standard interconnect protocol. CXL devices can be classified into three types [16] based on the supported sub-protocols. Fig. 2 illustrates the three CXL device types. Type 1 devices, such as smart NICs, include only a processor and no additional memory. Type 2 devices [28], which include devices like GPUs and accelerators, feature both a processor and attached memory and must support all three CXL protocols. Finally, Type 3 devices [18], such as memory expansion units, are memory-only devices and must implement the CXL.mem protocol. In this research, we utilize CXL 2.0 Type-3 devices.

CXL technology is constructed upon the PCIe physical layer [29], indicating that its bandwidth is constrained by the fundamental properties of PCIe. The performance metrics of CXL expansion cards, comparing typical commercial offerings with our customized CXL expansion cards, are presented in TABLE I. The efficacy of CXL expansion cards markedly surpasses that of NVMe SSD devices.

CXL Memory Expansion offers high bandwidth, low latency, and excellent scalability, supporting heterogeneous device connectivity. This memory expansion technology facilitates system storage expansion, unifies memory management, reduces reliance on a single high-performance memory, and lowers costs [30]. Memory connected to a CXL device can be mapped as either host-managed device memory (HDM) or private device memory (PDM) [31], [32]. HDM can be accessed from the host via standard write-back semantics, and both HDM and host memory can be mapped to a unified memory space. Additionally, CXL enables memory pooling and sharing [23], [33]–[35], allowing HDMs to be flexibly allocated and deallocated across hosts or shared consistently between them. These properties have led to increasing attention in both academia and industry for building disaggregated memory systems [36], [37].

C. Existing Research on CXL for Deep Learning

Recent efforts have begun to explore the integration of CXL in large-scale deep learning systems, focusing on disaggregated memory architectures and high-speed interconnects. Proposals include near-memory processing (NMP) [26] and memory pooling frameworks [23] to improve memory utilization and flexibility. CXLSim [27] provides a simulation platform for CXL systems, while Pond [23] demonstrates efficient pooling across CXL devices. Current research has largely focused on memory coherence [4] and hierarchical memory architectures, with limited attention to fine-grained tensor offloading and NUMA-optimized scheduling for training.

However, most prior studies have been limited to simulations or analytical models and do not evaluate the real hardware performance in large-scale deep learning training. Systematic approaches that combine tensor offloading with CXL memory pools, including NUMA-aware optimization, fine-grained tensor scheduling, and computation-communication overlap, remain largely unexplored. Our work addresses this gap by providing the first real-hardware evaluation of CXL-enabled tensor offloading, demonstrating effective scaling for extreme-scale deep learning tasks.

III. Motivation

Traditional tensor offloading with CPU DRAM and NVMe SSDs enables model scaling but is limited by inherent SSD constraints. Low bandwidth (5–7 GB/s), high latency ($\sim$80 μs), and coarse-grained tensor transfers in ZeRO-Infinity lead to severe communication delays, load imbalance, and heavy software overhead, constraining efficiency at extreme scale.

CXL, in contrast, offers tens of GB/s bandwidth, sub-microsecond latency, and atomic memory semantics. These capabilities support fine-grained tensor access and NUMA-aware allocation, making CXL better suited for large-scale training by design. However, integrating CXL introduces new challenges, including NUMA imbalance, fragmentation, and efficient communication–computation overlap.

A. Challenges of Integrating CXL Memory Pools into Heterogeneous Systems

Using CXL memory pools for tensor offloading in large deep learning models is a complex task that extends beyond

TABLE I
PERFORMANCE METRICS FOR THE CXL MEMORY EXPANSION CARD

Performance Metric	Ideal Range	Typical Market CXL Expansion Card	JTCXL-Memory Expander (Sample)
Bandwidth	32–64GB/s	20–40 GB/s	30 GB/s
Latency	200–500ns	300–800ns	380 ns
Capacity	128GB–2TB/card	128GB–2TB/card	256GB/card
Interface	PCIe5.0 + CXL 2.0	PCIe5.0 + CXL 2.0	CXL 2.0 ×16

simple hardware integration. This approach faces several key challenges when applied in practice. On the DeepSpeed platform, the memory management and communication overhead optimizations of the original ZeRO-Infinity approach fail to deliver satisfactory results.

NUMA and Memory Access Issues. CXL memory pools are typically accessed across NUMA nodes, which presents a set of challenges related to memory binding and data location management. In a heterogeneous system, each device, such as a GPU or CPU, may be connected to its local memory as well as remote memory that exists in other NUMA nodes. Improper memory binding and mismanagement of data location can lead to unbalanced memory access patterns, where GPUs might be forced to access remote memory on another NUMA node. This increases the time required to fetch or store data, leading to higher memory access latency. As a result, the performance of deep learning tasks, which are highly sensitive to memory access speed, can be significantly degraded. Moreover, data locality, which plays a crucial role in maximizing throughput, is often compromised in these setups, further hindering system performance.

Memory Fragmentation. As large models are scaled for training, memory usage patterns change dynamically, with memory being continuously allocated and freed. When there is a shortage of contiguous free memory blocks or when memory management is inefficient, fragmentation can occur within the memory pool. This fragmentation prevents the efficient allocation of memory for new or expanding tensors. The problem is further exacerbated by the fact that memory spans across multiple devices and NUMA nodes. Over time, this fragmentation reduces the effective memory capacity, leading to suboptimal memory utilization. When memory is fragmented, the system may struggle to find sufficiently large blocks of free memory, which can slow down training or even cause the system to run out of memory. Ultimately, this results in decreased system throughput, as resources are not being used as efficiently as possible.

Scheduling and Load Balancing. Incorporating CXL pools into heterogeneous GPU–CPU systems complicates scheduling. Multiple devices share the same pool, which can cause contention without careful coordination. Efficient scheduling and load balancing are thus essential to distribute workloads across resources and sustain high parallel efficiency.

B. The Communication Overhead Issue in TOCP

Bandwidth and Latency Bottlenecks. While CXL memory pools offer high bandwidth and low latency memory scaling, their performance is still significantly lower than that of GPU local memory. This issue becomes particularly evident when multiple devices access memory concurrently, leading to frequent tensor swapping. During such operations, the large bandwidth demand can cause the CXL memory pool to become a performance bottleneck.

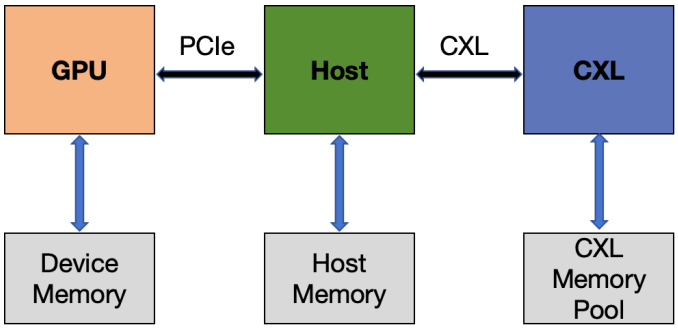

Fig. 3. The memory architecture of TOCP.

Communication and Synchronization Overheads. Despite the flexible memory scaling capabilities of CXL memory pools, cross-device synchronization and communication overheads remain significant. Data transfers between GPUs and CXL memory pools are often constrained by synchronization requirements, particularly when dealing with distributed training scenarios or models with complex interdependencies. The synchronization overhead arises from the need to maintain consistency between devices, especially during operations like backpropagation, parameter updates, and gradient calculations. These synchronization requirements often lead to waiting times as devices pause to ensure that data transfers are completed before proceeding with computations, which significantly slows down the training process. The latency introduced by cross-device communication not only delays individual training steps but also reduces the overall throughput of the system, making it increasingly difficult to maintain high performance as the model size and number of devices grow.

IV. TENSOR OFFLOADING BASED ON CXL MEMORY POOL

A. System Overview

We propose a novel tensor offloading framework that integrates CXL memory pools with the ZeRO-Infinity architecture to enable efficient scaling for large-scale deep learning, as shown in Fig. 3. In our system, CXL Type-3 memory expansion cards are employed as high-bandwidth, low-latency memory pools, which are connected to the CPU via PCIe. During training, GPU memory retains only the parameters and activations that are actively involved in computation, while the CPU stores optimizer states and caches frequently accessed parameters. The CXL memory pool holds the majority of the

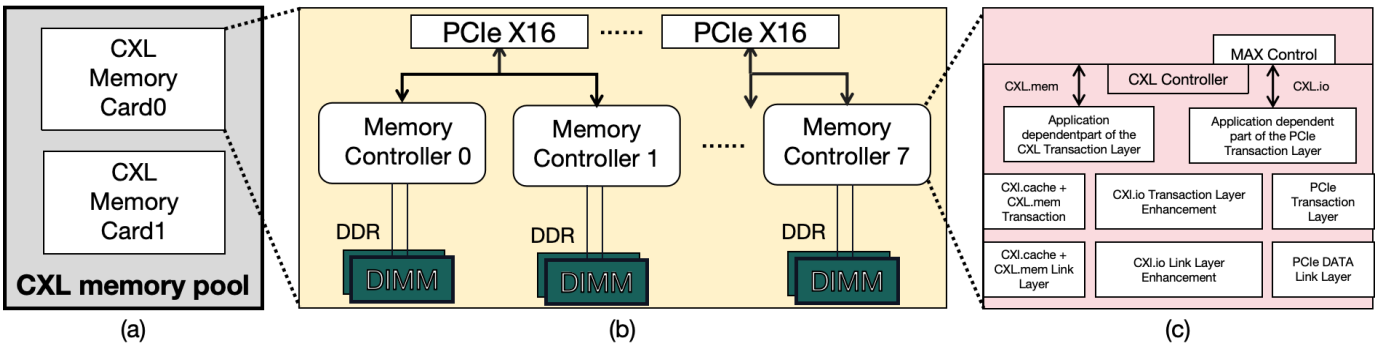

Fig. 4. (a) CXL memory pool. (b) The CXL Memory Expansion Card. (c) Memory Controller.

model weights and optimizer states for long-term storage, and supports both on-demand loading and asynchronous offloading. This allows the GPU memory capacity to be dynamically extended, addressing the GPU memory wall while ensuring low-latency access to tensors.

In the GPU-CPU-CXL heterogeneous architecture, the original SSD-based memory management and communication overlap design of ZeRO-Infinity becomes inefficient. The CXL memory pool replaces the role of SSDs by providing memory semantics (load/store), which align naturally with deep learning workloads. The system is designed to be NUMA-aware, ensuring optimal memory access patterns and reducing communication overhead across different NUMA nodes. Furthermore, the integration of fine-grained tensor scheduling and DAP maximizes the communication-computation overlap, further enhancing performance.

B. NUMA-Aware Memory Pooling

1) The Architecture of CXL Memory Pool: The CXL memory pool enables flexible system memory expansion and supports dynamic allocation based on user requirements. Fig. 4 illustrates the architecture and hardware composition of the CXL memory pool. The pool comprises multiple CXL memory expansion cards, which are Type 3 devices compliant with the CXL.io and CXL.mem protocols. These expansion cards are interconnected via CXL links using high-speed connectors (PCIe $\times 16$). Each $\times 16$ CXL high-speed link is divided into two $\times 8$ memory controllers. Each $\times 8$ memory controller supports the bandwidth of one DDR (Double Data Rate) memory channel and interfaces with two DIMMs (Dual Inline Memory Modules). The entire system is orchestrated by a unified management engine responsible for memory allocation, resource scheduling, and system health monitoring.

CXL memory pool aggregates memory across multiple devices to meet terabyte-scale capacity requirements. By centralizing memory resources, it enhances data transfer efficiency, increases bandwidth availability, and improves overall memory utilization. This approach reduces hardware redundancy and offers a cost-effective solution for large-scale memory expansion.

2) CXL Memory Management: In the CXL memory pool management system, shown in Fig. 6(a), the memory pool is connected to host CPUs via PCIe, enabling unified cross-node memory pooling without modifying the host OS. The system supports memory identification, dynamic allocation, health monitoring, access statistics, and log management, all managed by the system management module. To optimize allocation and reduce fragmentation, the CXL memory pool employs a hierarchical allocator, categorizing memory requests by object size and access pattern, with separate strategies for large blocks and slabs.

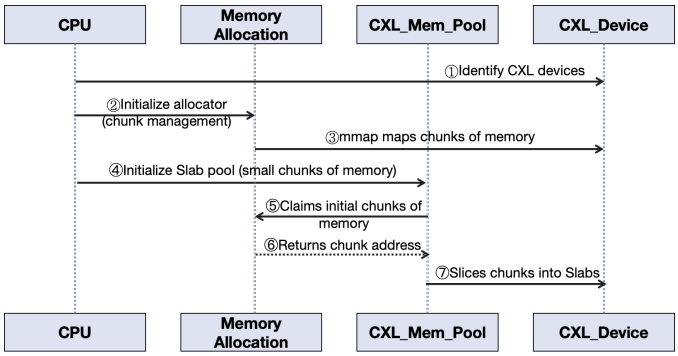

Fig. 5. The initialization of CXL memory pool.

Memory pool initialization. The CXL memory pool is initialized based on a memory management policy for object-based classification, as shown in Fig. 5 below. After identifying the CXL device, the Memory_Allocator is initialized to manage memory chunks, which are then mapped through memory mapping. The CXL_Mem_Pool (Slab pool) is then initialized, and an initial memory block is requested from Memory_Allocator, with the address of the large block returned to CXL_Mem_Pool. The large block is then sliced into multi-level Slabs within the memory pool.

Object-based Hybrid Memory Allocation Strategy. During training, different data types demonstrate distinct access patterns. Model parameters generally constitute over 70% of the total memory, characterized by prolonged lifecycles and continuous large-block access. Conversely, gradient tensors and optimizer states are smaller objects (ranging from 16 KB to 1 MB) with short lifecycles and frequent allocations and deallocations, potentially resulting in memory fragmentation. Therefore, we propose an object-based hybrid memory allocation strategy that strikes a balance between software

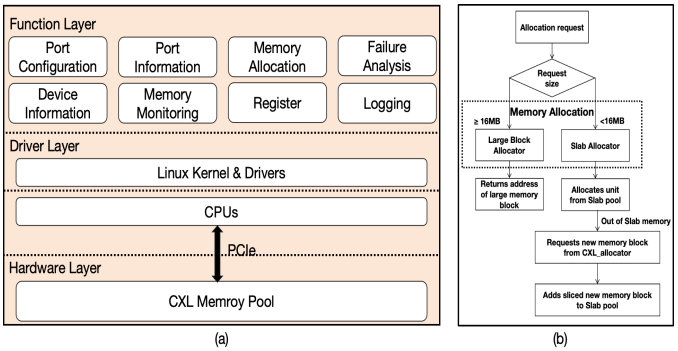

Fig. 6. (a) Memory pooling system management and (b) object-based Hybrid Memory Allocation Strategy.

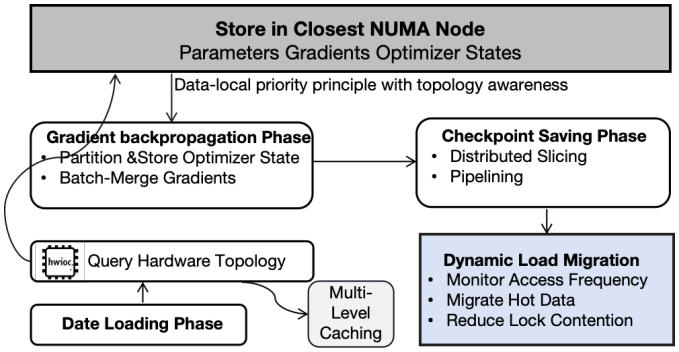

Fig. 7. NUMA load balancing strategy.

overhead and memory utilization. The procedure is illustrated in Fig. 6(b).

Large Block Allocation. For parameter partitions larger than 16 MB, a large-block allocator is used to map memory directly into the virtual address space, eliminating metadata overhead. Key features include direct mapping, zero-copy streaming, and NUMA binding. Parameter updates are executed directly on CXL memory via CXL.mem atomic operations, bypassing CPU cache involvement.

Slab Allocation. For small, high-frequency objects (gradients and optimizer states < 16 MB), a multi-granularity slab allocator (16 KB, 1 MB, 16 MB) is employed. Independent pools and lock-free queues reduce contention and improve allocation efficiency under frequent operations.

Memory recycling and release. The design of memory recycling in the CXL pool is aligned with the object-based allocation strategy, enabling efficient lifecycle management through hardware–software co-design. During training, memory is released after backpropagation. Large partitions (≥ 16 MB) are freed via the Memory_Allocator, which marks the block as free and adds it to the global free list. Smaller objects (< 16 MB) are managed by CXL_Mem_Pool, which updates the corresponding slab metadata and bitmap. Defragmentation and Merging.

Defragmentation and Merging. A background thread periodically merges adjacent free blocks using the Memory_Allocator free list. To minimize NUMA overhead, merging is restricted to blocks within the same node [38]–[40], triggered upon allocation failure or when free blocks exceed a threshold. An address-aligned merging strategy prioritizes 2 MB page boundaries to improve subsequent allocations. When all slab units are freed, the entire block is returned to Memory_Allocator to avoid fine-grained stagnation.

Cross-Level Recycling. When memory utilization falls below a predefined threshold (e.g., 20%), low-frequency data in DRAM is migrated back to CXL memory, and redundant slab blocks are released to reduce the memory pool size. Dynamic hierarchical waterline control is implemented, with high and low watermarks (e.g., 80%/20%) monitored in real-time using atomic counters. When idle units exceed the high watermark, blocks are returned to Memory_Allocator. Conversely, when occupancy falls below the low watermark, additional blocks are allocated to increase memory capacity. Atomic bit operations update the slab unit occupancy status, eliminating lock contention and minimizing the impact on system throughput.

3) NUMA-Aware Adaptation for Heterogeneous Systems: To mitigate cross-NUMA access imbalance and PCIe bandwidth contention, we introduce a NUMA-aware memory allocation strategy. The key idea is to co-locate frequently accessed tensors with the nearest NUMA node, thereby reducing remote memory traffic. Compared with naive allocation policies, this approach dynamically adapts to workload characteristics, significantly improving locality.

During data loading, hwloc (v2.9.0) is employed to query the hardware topology, and parameters, gradients, and optimizer states are placed in the NUMA node closest to each GPU according to PCIe locality. A multi-level caching strategy further prefetches frequently accessed parameters into local NUMA nodes, thereby reducing cross-node latency. In the backpropagation phase, optimizer states are partitioned into the CXL memory of local NUMA nodes, while gradients are batch-merged into larger blocks before being written locally, effectively lowering transaction overhead and alleviating cross-node bandwidth contention. During checkpointing, distributed slicing and pipelining enable each NUMA node to handle its local slice independently, avoiding transmission bottlenecks. A dynamic migration mechanism continuously monitors access frequency and relocates hot data to low-load nodes, while CXL.mem atomic operations (e.g., atomic_add) are leveraged to minimize lock contention and reduce cross-node synchronization. Overall, this NUMA-aware strategy optimizes resource utilization, enhances scalability, and mitigates explicit GPU memory demands. The pseudo-code for the proposed algorithm is provided in Algorithm 1.

C. Fine-Grained Tensor Offloading

A key innovation of our framework is the fine-grained tensor offloading strategy. Unlike traditional offloading techniques that rely on coarse-grained tensor transfers, TOCP splits large tensors into smaller sub-blocks, which are then dynamically allocated to different CXL memory pools and NUMA nodes, as shown in Fig. 8. These sub-blocks are dynamically scheduled on demand, guided by the computational graph's dependency order and runtime access patterns. To improve data availability and reduce latency, the system leverages an asynchronous prefetching mechanism to load upcoming tensor slices before

Algorithm 1 NUMA Load Balancing and Data Migration.

```
function NUMA_LOAD_BALANCING(training_tasks)
    # 1. Initialization Phase: Data Localization Distribution
    for each gpu in training_tasks.gpus do
        numa_node = get_nearest_numa_node(gpu.id)
        assign_parameters_to_local_numa(gpu, numa_node)
        bind_thread_to_cpu_core(gpu, numa_node)
    end for

    # 2. Dynamic Load Migration Loop
    while training_ongoing do
        execute_training_step()
        if time_to_balance() then
            hot_nodes = detect_hot_numa_nodes()
            for each node in hot_nodes do
                cold_node = select_underutilized_node()
                migrate_hot_data(node, cold_node)
            end for
        end if

        # 3. Hardware Acceleration Optimization
        use_cxl_atomic_ops()
        pipeline_data_transfer()
    end while
end function

function MIGRATE_HOT_DATA(src_node, dst_node)
    # Helper Function: Topology-Aware Data Migration
    data = fetch_remote_data(src_node)
    lock_free_copy(data, dst_node)
    update_metadata_locality(data)
end function
```

each layer is executed, while simultaneously offloading unused slices back to the CXL memory pool. This allows the system to optimize memory resource utilization and reduce latency by allowing parallel access to different parts of the tensor. The dynamic scheduling of these sub-blocks is informed by the tensor access patterns, ensuring that frequently accessed data is stored in the closest memory pool.

This fine-grained approach is particularly well-suited for scaling up the training of extremely large models, where memory pressure and data movement overhead are key bottlenecks. By integrating NUMA topology awareness, bandwidth utilization monitoring, and CXL-supported atomic operations, the system further optimizes data transfer paths and memory access efficiency. These enhancements significantly improve training throughput and resource utilization, effectively mitigating the bandwidth and latency gap between CXL memory and GPU on-chip memory. With continuous monitoring of tensor usage, the system dynamically adjusts tensor sub-block sizes and locations, enabling adaptive memory management based on the current workload.

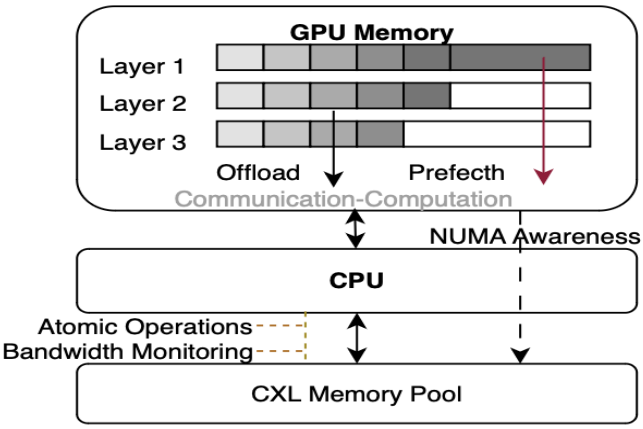

Fig. 8. Fine-Grained Tensor Offloading with CXL Memory Pools

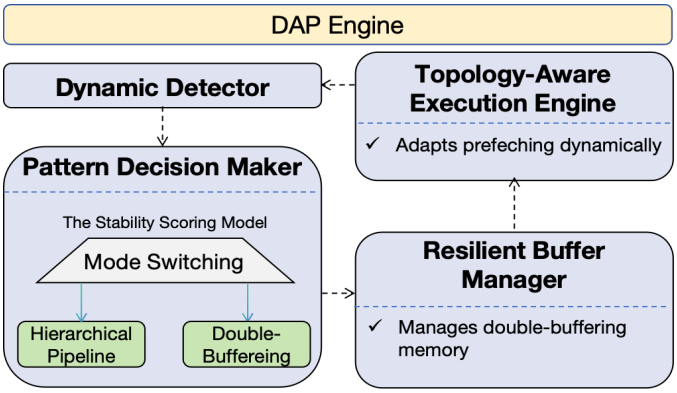

Fig. 9. The DAP engine.

D. Dynamic Adaptive Pipelining (DAP)

GPUs access the CXL 2.0 Type-3 expansion card indirectly through the CPU, which introduces additional data transfer latency. The PCIe bandwidth between CPU and CXL can further become a bottleneck when multiple GPUs concurrently offload data. To coordinate GPU-CPU and CPU-CXL transfers, CUDA Streams [41] and multithreading are employed, but this increases pipeline synchronization complexity. Although conventional double-buffering and static pipelining are widely used to overlap computation and communication, they generally assume homogeneous hardware and fixed transfer patterns. In contrast, heterogeneous GPU-CPU-CXL systems exhibit highly variable transfer latency due to NUMA topology, PCIe bandwidth limits, and CXL device characteristics. As a result, static pipelines often cause idle GPU cycles and yield suboptimal overlap efficiency.

To address this, we propose Dynamic Adaptive Pipelining (DAP). DAP continuously monitors the status of tensor offloading and adjusts the prefetch window size in real-time to ensure that data is preloaded into memory before it is needed by the GPU. By dynamically adjusting the prefetch window according to tensor access patterns and GPU memory utilization, DAP reduces idle time and enables seamless overlap between computation and communication.

DAP comprises four essential components: the Dynamic

Detector, Pattern Decision Maker, Resilient Buffer Manager, and Topology-Aware Execution Engine Module, as shown in Fig. 9. The Dynamic Detector continuously tracks the operational characteristics of the computational graph, including operation types, execution times, and data dependencies. To optimize the analysis of computational graphs, we employ a lightweight dynamic graph analysis approach, focusing on the features (type, execution time, and dependencies) of the nearest N operations. This reduces analysis overhead. The Stability Scoring Model is defined as:

$$\text{StabilityScore} = \alpha \cdot \frac{\text{consecutive same op count}}{N} + \beta \cdot \left(1 - \frac{\text{execution time variance}}{\text{mean}}\right) \quad (1)$$

where α and β are weight coefficients from offline training, α reflects the relative importance of consecutive operations within the computational graph, and β represents the weight of execution time variance in the stability score. The coefficients were derived using a set of representative Transformer-based training traces collected from models of 1.3B–130B parameters. The optimization objective was to maximize pipeline stability by minimizing execution stalls and variance in operator latency, thereby ensuring that the computed StabilityScore correlates with actual overlap efficiency during runtime.

The Mode Decision Maker switches between hierarchical pipeline and double-buffering prefetch modes using lightweight graph analysis. Resilient Buffer Manager allocates and releases double-buffering memory on demand, allowing DRAM and CXL memory co-allocation. The Topology-Aware Execution Engine binds the data transfer path to the optimal NUMA node, dynamically adjusts the prefetch step size based on real-time PCIe bandwidth and computational latency, and fine-tunes the prefetch window for optimal performance.

The hierarchical pipeline architecture for computing and communication overlap divides jobs into computation, GPU-CPU transfer, and CPU-CXL transfer. To maximize throughput, tasks are distributed to separate CUDA Streams and overlap via inter-stream asynchronous parallelism. The data management method uses CUDA Stream management and a single buffer with asynchronous copying. This design supports NUMA binding naturally, enabling high generality and low overhead.

Pipeline prefetching uses computational graph dependencies to predict future data needs and start prefetching in advance for double-buffering switching and no-wait computing. Data management uses double-buffering (Buffer A/B alternating) to physically isolate processing and data transport. However, this approach requires additional memory for the double-buffering, resulting in high system transformation costs and limited scalability, as it is focused on achieving maximum performance.

DAP is tightly coupled with ZeRO-3, CUDA streams, and the CXL memory topology, making it well-suited for heterogeneous clusters based on NVLink and CXL. However, it lacks portability to non-CXL architectures such as pure NVMe offloading or InfiniBand with GPU Direct RDMA, where direct

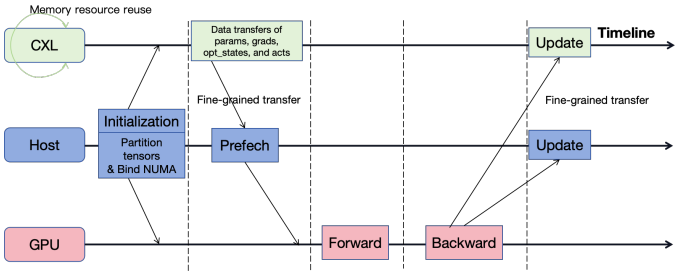

Fig. 10. Overview of TOCP.

adaptation is not feasible.It differentiates itself from conventional pipelining approaches by leveraging dynamic execution graph analysis, topology-aware data transfer scheduling, and a dual-mode prefetching mechanism. Although DAP incurs additional memory overhead due to buffer separation and relies on accurate runtime telemetry, it substantially enhances system throughput and mitigates latency under conditions of fluctuating system load, dynamic interconnect delays, and large-scale model memory footprints.

E. Atomic Operations for Hardware Acceleration

To reduce the software overhead associated with tensor offloading, TOCP leverages atomic operations provided by the CXL.mem sub-protocol. These atomic operations ensure efficient synchronization between the GPU, CPU, and CXL memory pools, particularly during complex tensor updates. Traditional memory synchronization methods rely heavily on software-based locking mechanisms, which introduce significant overhead due to frequent context switching, increased latency, and contention on shared memory resources. By offloading synchronization tasks to hardware through CXL's native atomic operations, we eliminate the need for these software-based locks, thereby significantly reducing overall latency and improving memory access efficiency.

This hardware acceleration capability is particularly useful in distributed training environments, where tensor updates and memory synchronization can become a bottleneck. By relying on CXL's native atomic operations, we ensure that memory access between devices is as efficient as possible, which is crucial for scaling deep learning models across multiple GPUs.

F. Applying TOCP

To enhance ZeRO-Infinity, we apply the TOCP framework, which utilizes a heterogeneous memory hierarchy composed of CXL memory pools and CPU DRAM as an extended memory substrate. As illustrated in Fig. 10, the training workflow is initialized on the host CPU, where model parameters are partitioned into tensor slices and assigned to NUMA-aware memory domains. A full replica of both model parameters and gradients is pre-stored in the CXL memory pool, while CPU DRAM primarily serves as a cache and transient staging buffer. Data placement can be further customized based on training configurations.

During runtime, we adopt the TOCP to enable fine-grained tensor-level transfers, mitigating transmission stalls and reducing end-to-end latency. TOCP also dynamically adjusts

TABLE II
SYSTEM CONFIGURATIONS

Configuration of GPU-CPU-CXL System	
GPU	8 NVIDIA Tesla V100 Tensor Core GPUs
GPU Memory	32GB HBM2 on each GPU
CPU	2 AMD EPYC Genoa 9274 24-Core @4.1GHz
CPU Memory	576GB DDR4
CPU cache	L1, L2, and L3 are 3M, 48M, and 512M, respectively
CXL Memory	4 CXL Memory Expander Cards, 1TB DDR5
PCIe	Bidirectional 32 GB/s PCIe
SSD	4TB PCIe 4.0

TABLE III
EXPERIMENT CONFIGURATIONS

#params	hidden dim	layers	batch/GPU	mp	fp16 param	Opt State
1.3B	2K	24	16	1	CXL Memory	CPU
13B	5K	40	12	1	CPU	CXL Memory
50B	8K	62	26	1	CPU	CXL Memory
130B	10K	90	24	1	CXL Memory	CXL Memory

the prefetch window size to maximize data transfer efficiency under variable bandwidth and latency constraints.

In the forward pass, GPU-resident parameters are directly utilized. Intermediate activation values, optimizer states, and other required data for the backward phase are loaded on-demand following a latency-aware scheduling policy. The scheduler employs the DAP algorithm to overlap computation and communication, thereby concealing long transmission paths and reducing communication overhead.

Write-backs of gradients and updated optimizer states are also managed with fine-grained control. In particular, CXL-based atomic operations are leveraged to directly update optimizer states within the CXL memory pool, effectively eliminating unnecessary host-memory round trips and alleviating bandwidth saturation.

To further improve memory efficiency, TOCP incorporates tensor reuse and fragmentation-aware memory management, enabling high utilization of the CXL memory pool. This reduces the risk of offload delays and fallback allocations to CPU DRAM due to pool exhaustion or excessive fragmentation.

V. EVALUATION

A. Experimental Setup

Testbed. We conducted experiments on a real system with CXL expansion cards. The system configuration is summarized in TABLE II. The software stack includes PyTorch with DeepSpeed-ZeRO-Infinity, extended to support our proposed CXL memory pool framework. The CXL expansion card is equipped with 16 DDR5–4800 DIMMs, and we utilized the Montage Technology CXL intelligent memory controller.

Workloads. For performance evaluation, we utilize a Transformer-based model similar to GPT. We fixed the sequence length at 1024 and varied the number of hidden dimensions and layers to generate models with different parameter counts. The TABLE III details the specific configurations of the model.

Baseline. For model training, we adopt state-of-the-art methods in DeepSpeed, including ZeRO-Offload and ZeRO-Infinity. To assess the effectiveness of CXL-extended memory and the performance of TOCP, we evaluate the following configurations:

ZeRO-Infinity (SSD): the baseline system that performs offloading to NVMe SSDs using the CPU as an intermediate buffer. ZeRO-Offload (CPU DRAM): a configuration in which tensors are offloaded exclusively to host DRAM. Naive CXL Offloading: a direct integration of CXL memory without NUMA-aware scheduling or dynamic adaptive pipelining (DAP). TOCP (Proposed): our full system that incorporates CXL memory pools with NUMA-aware allocation, fine-grained tensor offloading, and DAP to maximize communication–computation overlap.

B. Overall Performance For Extreme Scale Deep Learning Offloading

1) System Throughput: Fig. 11(a) compares the throughput across different model scales (1.3B, 13B, 50B, 100B). SSD-based ZeRO-Infinity exhibits significant degradation due to SSD latency. ZeRO-Offload (CPU DRAM) achieves moderate throughput but lacks scalability beyond 40B models due to limited capacity. Naive CXL Offloading improves over SSD but suffers from cross-NUMA imbalance and high communication overhead. TOCP achieves up to 72.7% higher throughput than SSD-based ZeRO-Infinity, narrowing the bandwidth gap with GPU HBM. As a result, scientists can effortlessly scale their models. If future CXL expansion cards support higher protocols and GPUs integrate with the CXL ecosystem, significant performance improvements can be anticipated.

2) Impact of System Features on Model Scale Sensitivity: We demonstrate the effects of various device placement strategies utilizing 8 GPUs on model size and the maximum trainable model size.

Maximum model size. Fig. 12(a) illustrates the impact of different offloading strategies on the maximum trainable model size. In ZeRO-Infinity, parameter states are partitioned and offloaded. Under the same settings, TOCP achieves a maximum model size 1.14× larger than ZeRO-Infinity (SSD), due to its higher bandwidth and superior communication strategies. For fairness, the SSD baseline and the CXL extension are both constrained to 1 TB in our setup, although SSDs typically provide larger capacity. Ideally, CXL-extended memory could scale to the petabyte (PB) level. Based on the model scaling law in ZeRO-Infinity with SSD, it is estimated that a single DGX-2 node, equipped with 20 TB of CXL-extended memory, would be capable of training very large models exceeding 1000 billion parameters (1000B).

To ensure fairness, we clarify that the improvement in maximum trainable model size is not due to raw capacity differences. In our setup, both SSD and CXL are limited to 1 TB for consistency, although SSDs can provide larger capacity in practice. The advantage of TOCP over SSD-based ZeRO-Infinity stems from higher bandwidth, lower latency, and finer-grained memory semantics of CXL, rather than capacity. Thus, the 1.14× gain reflects effective utilization and communication efficiency, not storage size.

Scalability Analysis. While our evaluation is conducted on a single node with 8 NVIDIA V100 GPUs and CXL 2.0

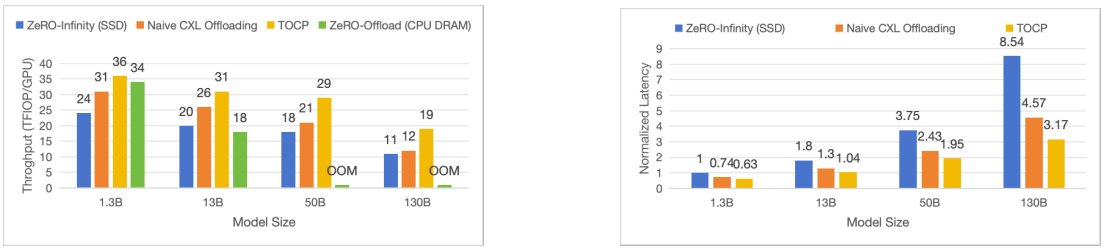

Fig. 11. (a) Training throughput comparison of ZeRO-Infinity (SSD), ZeRO-Offload (CPU DRAM), Naive CXL offloading, and TOCP on a single node with 8 V100 GPUs. (b) The forward and backward propagation (FWD+BWD) times for training models of various sizes, normalized against the FWD+BWD time of a 13B model trained on ZeRO-Infinity (SSD).

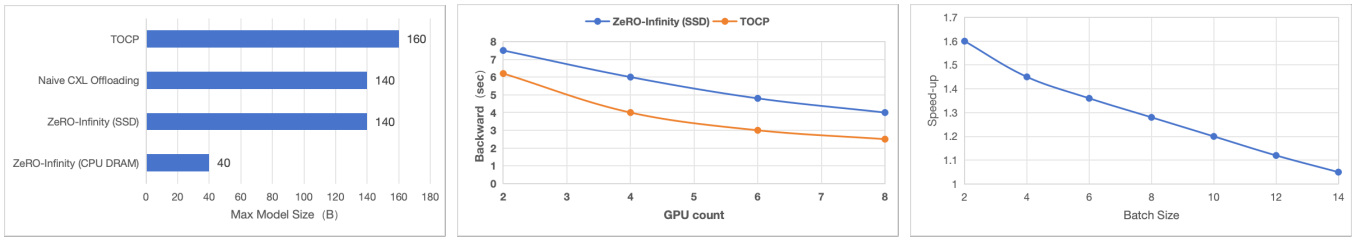

Fig. 12. (a) Comparison of the maximum trainable models under different ZeRO-based tensor offloading approaches. (b) Backpropagation time of an 8B-parameter model: ZeRO-Infinity (SSD) vs. TOCP. (c) Speedup from communication overlap under different batch sizes.

Type-3 expansion cards, we further provide a simulation-based extrapolation to assess scalability in multi-node settings. Following the scaling behavior reported in ZeRO-Infinity [2], TOCP is expected to maintain its relative performance advantages when extended to multi-node GPU-CXL clusters. For example, in a 4-node configuration with aggregated CXL memory pools, TOCP could support models in the order of hundreds of billions of parameters, while sustaining overlap efficiency comparable to that of single-node experiments. This analysis indicates that TOCP has the potential to scale toward trillion-parameter training in future large-scale deployments.

3) Impact of System Features on Performance Model Scale: We evaluate the impact of memory pooling, NUMA load balancing, and communication overlapping designs on training speed. Fig. 11(b) illustrates the forward propagation and backpropagation times for different offloading configurations across various model sizes. TOCP significantly alleviates the communication bottleneck by mitigating PCIe bandwidth contention through NUMA load balancing, in addition to leveraging the high bandwidth advantages of CXL. This results in enhanced computation and communication overlap, reducing latency by 11–16.4% compared to Naive CXL Offloading and by 37–62.9% compared to ZeRO-Infinity with SSD.

TOCP vs. ZeRO-Infinity with SSD. Fig. 12(b) illustrates the impact of gradient offloading to CPU memory, comparing ZeRO-Infinity with SSD and TOCP on backpropagation time for an 8B parameter model. TOCP achieves a performance boost of nearly 1.58× with 8 GPUs, outperforming the SSD-limited bandwidth.

Prefetching and Overlapping. Fig. 12(c) illustrates the relative throughput difference for the 8B parameter model over 8 GPUs, comparing communication overlapping and prefetching with and without activation. Results indicate that prefetching and overlapping are essential for attaining optimal performance with small batch sizes per GPU, while their significance decreases as batch size increases.

C. Performance of CXL Memory Pool

We evaluate the efficacy of CXL memory pools in comparison with conventional memory allocation methods, emphasizing memory management efficiency and latency. Furthermore, we investigate the impact of NUMA load balancing schemes on access efficiency.

1) Efficiency of Memory Allocation and Release in CXL Memory Pools: We conducted memory allocation and release tests using various request sizes (e.g., 16MB, 1MB, 16KB) to evaluate the latency of memory allocation in three configurations: ZeRO-Infinity with SSD, Naive CXL Offloading with standard memory mapping, and TOCP utilizing object-based hybrid memory allocation. The memory allocation latency is presented in Fig. 13. The SSD offloading scheme demonstrated the highest latency, primarily due to the overhead of multiple software layers, including file system metadata management. In contrast, Naive CXL Offloading, which uses standard memory allocation, reduces the latency for 16KB, 1MB, and 16MB allocations to 10.0% through hardware-level access, with latency times of 10.7μs, 18.3μs, and 126.6μs, respectively. However, it is still constrained by the limitations of the standard memory allocator. The TOCP, employing object-based hybrid memory allocation, significantly outperforms the standard memory allocator. For 16KB allocations, the latency is reduced by 80.4% to 2.1μs; for 1MB allocations, the latency

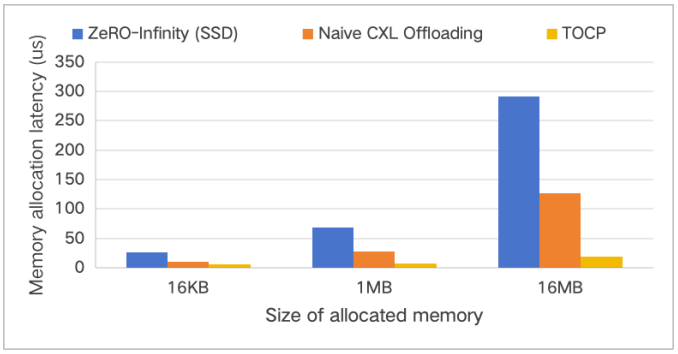

Fig. 13. Memory allocation latency for different allocation policies and sizes.

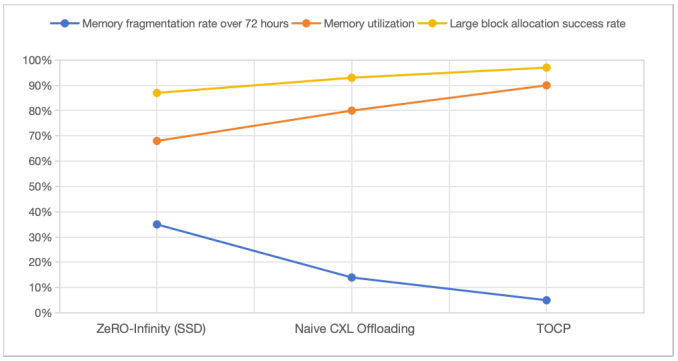

Fig. 14. Memory fragmentation rate over 72 hours, memory utilization, and large block allocation success rate.

decreases by 77.4% to 6.4μs; and for 16MB allocations, latency is reduced by 78.8% to 26.8μs.

2) Memory Utilization and Fragmentation: The test results are presented in Fig. 14. The SSD offload solution exhibits only 68% memory utilization due to overhead from file system metadata and external fragmentation. After 72 hours, fragmentation increases to 35%, reducing the allocation success rate for $\geq$16MB chunks to 87%. In contrast, the CXL memory allocation scheme with direct mapping improves memory utilization to 80%, but still experiences 14% fragmentation due to standard page-based management, resulting in a 93% chunk allocation success rate. The optimized memory pooling scheme, employing a hybrid object classification management strategy, achieves 90% memory utilization, with a fragmentation rate of just 5% over 72 hours and a 97% chunk allocation success rate. This improvement is attributed to zero-copy direct mapping of chunk memory requests, which eliminates metadata overhead, and the Slab pre-allocation mechanism, which limits internal fragmentation to 5%. The results demonstrate that customized memory management effectively mitigates fragmentation issues in large-scale training scenarios, providing stable memory guarantees for very large models.

3) NUMA Performance Test: We evaluate the impact of NUMA topology optimization on the CXL memory pool and measure memory access latency under NUMA load optimization, and the test results are shown in Fig. 15. The CPU latency for accessing local DRAM is 123.3ns. When the CPU is not bound to the CXL extended memory, the remote access latency to CXL memory is 471.8ns. This latency is reduced to

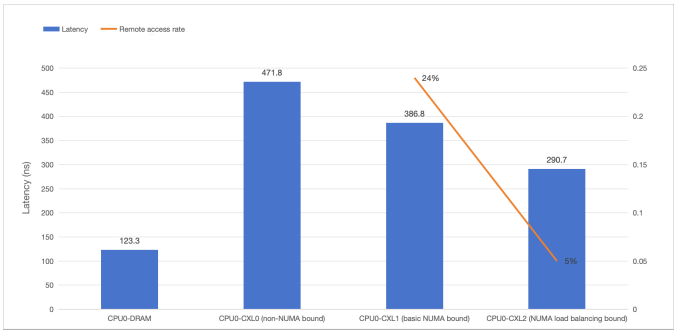

Fig. 15. Access atency and remote access rate under different NUMA binding policies.

386.8ns with basic CPU binding to the CXL extended memory using numactl, and further reduced to 290.7ns when employing a NUMA load balancing strategy to dynamically bind the CPU to the CXL extended memory. The dynamic binding with NUMA load balancing achieves a 24.8% reduction in latency compared to basic binding. Additionally, the NUMA load balancing strategy improves memory affinity, and the topology-optimized CXL memory pool architecture reduces the remote access rate from 24% (under basic binding) to 5%, significantly minimizing cross-node accesses and enhancing access efficiency.

VI. Discussion

Strengths. Our results confirm that integrating CXL memory pools into ZeRO-Infinity effectively mitigates the performance bottlenecks of SSD-based offloading. The proposed framework, which incorporates NUMA-aware allocation, fine-grained tensor scheduling, and DAP, achieves substantial improvements in both throughput and memory utilization. Moreover, leveraging CXL.mem atomic operations reduces synchronization overhead and improves cross-device memory access efficiency.

Limitations. Despite these benefits, several limitations remain. Current evaluations are restricted to a single node, and cross-node scalability under limited CXL bandwidth is not yet fully validated. Based on the scaling trends reported in ZeRO-Infinity [2], we extrapolate that TOCP can sustain its relative advantages when extended to multi-node deployments. For instance, under a 16-node cluster with CXL-extended memory, the projected capacity could support models in the hundreds of billions of parameters, with overlap efficiency comparable to that observed in single-node experiments. In addition, CXL 2.0 Type-3 devices only support CXL.mem and CXL.io, requiring GPU access via the CPU and introducing extra latency. The integration into frameworks like DeepSpeed also increases the complexity of the software stack.

Implications. This work demonstrates the potential of CXL memory pools to bridge the gap between GPU on-chip memory and secondary storage. By enabling flexible, scalable, and cost-effective memory expansion, CXL reduces reliance on expensive HBM, offering a practical path to training models with hundreds of billions of parameters using commodity hardware.

Future Directions. Looking ahead, CXL 3.0 and fabric-level switching will enable true multi-node memory pooling, enhancing scalability. Research on NUMA load balancing, adaptive scheduling, and fault-tolerant memory pooling will further enhance robustness. Extending these techniques to other accelerators beyond GPUs offers additional opportunities for heterogeneous system optimization.

VII. Related Work

NUMA and Memory Tiering with CXL. Recent efforts have investigated CXL-based tiering and NUMA-aware memory management. Zhong *et al.* demonstrate CXL-enabled tiering in virtualized systems with near-DRAM performance [35], while Zhou *et al.* propose NeoMem, which pushes profiling into CXL controllers for low-overhead hot-page detection [30]. These works focus on general-purpose and cloud workloads, but do not address tensor-level scheduling, NUMA imbalance, or computation–communication overlap critical for large-scale training.

CXL for AI Acceleration and GPU Memory Expansion. Several studies have used CXL to accelerate AI tasks and expand GPU memory. Park *et al.* introduce CXL-PNM with LPDDR-based near-memory acceleration for transformer inference [24], while Yun *et al.* propose CLAY, a scalable NDP architecture for embedding layers [42]. Gouk *et al.* demonstrate sub-10 ns latency CXL controllers for GPU memory expansion [33], and Tang *et al.* design Yggdrasil, a CXL-based distributed shared memory framework to reduce network I/O overhead [25]. While these studies highlight CXL's potential in AI acceleration, they do not address fine-grained tensor offloading or NUMA-aware scheduling for training large models.

Tensor Offloading for Large Model Training. ZeRO-Offload [3] and ZeRO-Infinity [2] extend GPU capacity using CPU and SSD storage, but are limited by bandwidth, latency, and coarse-grained transfers. Recent work confirms that offloading granularity and bandwidth mismatches critically affect throughput [4], [43]. Building on these insights, we use CXL.mem load/store semantics and atomic operations to redesign tensor scheduling for large-scale training.

Simulation and Tooling for CXL Systems. Simulation tools like CXLSim [27], DRackSim [44], and CXLMemSim model CXL architectures at the system level, enabling design space exploration. However, these tools are simulation-based. In contrast, our work evaluates CXL-based tensor offloading in ZeRO-Infinity on real hardware, bridging the gap between simulation and practical deployment.

VIII. Conclusion

In this paper, we proposed TOCP, a novel tensor offloading framework that integrates CXL memory pools into ZeRO-Infinity to overcome the bandwidth and latency limitations of SSD-based offloading. Our design combines NUMA-aware allocation, fine-grained tensor scheduling, and dynamic adaptive pipelining (DAP), while leveraging CXL.mem atomic operations to reduce synchronization overhead. Evaluation on a GPU-CPU-CXL heterogeneous system shows that TOCP achieves up to 72.7% higher throughput, 62.9% lower latency, and supports 1.14× larger model sizes compared with ZeRO-Infinity (SSD). As CXL continues to evolve, we foresee its broad application in deep learning, AI, and scientific computing, driving advancements in performance and scalability.

Acknowledgments

This work was supported in part by the Major Key Project of PCL (Grant PCL2024A06 and PCL2022A05). Additionally, it was supported by the Shenzhen Science and Technology Innovation Program under Grant RCJC20231211085918010.

References

[1] N. Li, A. Iosifidis, and Q. Zhang, "Dynamic semantic compression for cnn inference in multi-access edge computing: A graph reinforcement learning-based autoencoder," *IEEE Transactions on Wireless Communications*, vol. 24, no. 3, pp. 2157–2172, 2025.

[2] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[3] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "{Zero-offload}: Democratizing {billion-scale} model training," in *2021 USENIX Annual Technical Conference (ATC 21)*, 2021, pp. 551–564.

[4] D. Xu, Y. Feng, K. Shin, D. Kim, H. Jeon, and D. Li, "Efficient tensor offloading for large deep-learning model training based on compute express link," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024, pp. 1–18.

[5] H. Zhang, Y. E. Zhou, Y. Xue, Y. Liu, and J. Huang, "G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations," in *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023, pp. 395–410.

[6] Z. Zong, L. Lin, L. Lin, L. Wen, and Y. Sun, "Str: Hybrid tensor re-generation to break memory wall for dnn training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 8, pp. 2403–2418, 2023.

[7] Y. Hu, X. Liu, G. Yang, L. Li, K. Zeng, Z. Zhao, S. Chen, L. Zhao, W. Li, and K. Li, "Tightllm: Maximizing throughput for llm inference via adaptive offloading policy," *IEEE Transactions on Computers*, vol. 74, no. 7, pp. 2195–2209, 2025.

[8] N. Li, A. Iosifidis, and Q. Zhang, "Attention-based feature compression for cnn inference offloading in edge computing," in *ICC 2023 - IEEE International Conference on Communications*, 2023, pp. 967–972.

[9] A. A. Damana and P. Hitesh, "Innovative adapter-based fine-tuning and streamlined training strategies for text summarization," in *2024 International Conference on Electrical and Computer Engineering Researches (ICECER)*, 2024, pp. 1–6.

[10] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–16.

[11] Q. Chen, Q. Hu, G. Wang, Y. Xiong, T. Huang, X. Chen, Y. Gao, H. Yan, Y. Wen, T. Zhang, and P. Sun, "Lins: Reducing communication overhead of zero for efficient llm training," in *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*, 2024, pp. 1–10.

[12] J. Wu, L. Cai, Z. Cai, F. Zhang, and J. Liao, "Improving i/o performance and fairness in nvme ssds with pooling portions of cache partitions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2025.

[13] G. Xu, "Non-volatile memory express (nvme) resource pooling mechanism based on pcie," in *2024 4th International Conference on Electronic Information Engineering and Computer Communication (EIECC)*, 2024, pp. 1018–1022.

[14] S. Li, K. Lu, Z. Lai, W. Liu, K. Ge, and D. Li, "A multidimensional communication scheduling method for hybrid parallel dnn training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 8, pp. 1415–1428, 2024.

[15] Y. Xia, Z. Zhang, D. Yang, C. Hu, X. Zhou, H. Chen, Q. Sang, and D. Cheng, "Redundancy-free and load-balanced tgnn training with hierarchical pipeline parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 11, pp. 1904–1919, 2024.

[16] D. Das Sharma, R. Blankenship, and D. Berger, "An introduction to the compute express link (cxl) interconnect," *ACM Computing Surveys*, vol. 56, no. 11, pp. 1–37, 2024.

[17] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong *et al.*, "Demystifying cxl memory with genuine cxl-ready systems and devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 105–121.

[18] J. Park, W. Lee, T. Kim, Y. Lee, and S. Hong, "Performance characterization of cxl memory expander: Impact on read and write latencies," in *2024 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, 2024, pp. 1–5.

[19] C. Wang, K. He, R. Fan, X. Wang, W. Wang, and Q. Hao, "Cxl over ethernet: A novel fpga-based memory disaggregation design in data centers," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2023, pp. 75–82.

[20] K. Kim, H. Kim, J. So, W. Lee, J. Im, S. Park, J. Cho, and H. Song, "Smt: Software-defined memory tiering for heterogeneous computing systems with cxl memory expander," *IEEE Micro*, vol. 43, no. 2, pp. 20–29, 2023.

[21] X. Gao, X. Chen, and Y. Wang, "Investigating the memory extension and disaggregated memory pooling system," in *2024 4th International Conference on Electronic Information Engineering and Computer Science (EIECS)*, 2024, pp. 667–671.

[22] D. Gouk, M. Kwon, H. Bae, S. Lee, and M. Jung, "Memory pooling with cxl," *IEEE Micro*, vol. 43, no. 2, pp. 48–57, 2023.

[23] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 574–587.

[24] S.-S. Park, K. Kim, J. So, J. Jung, J. Lee, K. Woo, N. Kim, Y. Lee, H. Kim, Y. Kwon *et al.*, "An lpddr-based cxl-pnm platform for tco-efficient inference of transformer-based large language models," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 970–982.

[25] W. Tang, Y. Han, T. Ai, G. Li, B. Yu, and X. Yang, "Yggdrasil: Reducing network i/o tax with (cxl-based) distributed shared memory," in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024, pp. 597–606.

[26] H. Liu, L. Zheng, Y. Huang, J. Zhou, C. Liu, R. Wang, X. Liaot, H. Jinf, and J. Xue, "Enabling efficient large recommendation model training with near cxl memory processing," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 382–395.

[27] S. Kim, J. Kang, K. Kim, S. Lee, and B. Nam, "Cxlsim: A simulator for cxl memory expander," in *2025 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2025, pp. 156–159.

[28] H. Ji, S. Vanavasam, Y. Zhou, Q. Xia, J. Huang, Y. Yuan, R. Wang, P. Gupta, B. Chitlur, I. Jeong, and N. S. Kim, "Demystifying a cxl type-2 device: A heterogeneous cooperative computing perspective," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 1504–1517.

[29] R. Abdullah, H. Lee, H. Zhou, and A. Awad, "Salus: Efficient security support for cxl-expanded gpu memory," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 1–15.

[30] Z. Zhou, Y. Chen, T. Zhang, Y. Wang, R. Shu, S. Xu, P. Cheng, L. Qu, Y. Xiong, J. Zhang, and G. Sun, "Neomem: Hardware/software co-design for cxl-native memory tiering," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 1518–1531.

[31] D. S. Berger, D. Ernst, H. Li, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, L. Hsu, I. Agarwal, M. D. Hill, and R. Bianchini, "Design tradeoffs in cxl-based memory pools for public cloud platforms," *IEEE Micro*, vol. 43, no. 2, pp. 30–38, 2023.

[32] C. Chen, X. Zhao, G. Cheng, Y. Xu, S. Deng, and J. Yin, "Next-gen computing systems with compute express link: a comprehensive survey," *arXiv preprint arXiv:2412.20249*, 2024. [Online]. Available: https://arxiv.org/abs/2412.20249

[33] D. Gouk, S. Kang, H. Bae, E. Ryu, S. Lee, D. Kim, J. Jang, and M. Jung, "Breaking barriers: Expanding gpu memory with sub-two digit nanosecond latency cxl controller," in *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, 2024, pp. 108–115.

[34] M. Ha, J. Ryu, J. Choi, K. Ko, S. Kim, S. Hyun, D. Moon, B. Koh, H. Lee, M. Kim *et al.*, "Dynamic capacity service for improving cxl pooled memory efficiency," *IEEE Micro*, vol. 43, no. 2, pp. 39–47, 2023.

[35] Y. Zhong, D. S. Berger, C. Waldspurger, R. Wee, I. Agarwal, R. Agarwal, F. Hady, K. Kumar, M. D. Hill, M. Chowdhury *et al.*, "Managing memory tiers with {CXL} in virtualized environments," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 37–56.

[36] J. Hermes, J. Minor, M. Wu, A. Patil, and E. Van Hensbergen, "Udon: A case for offloading to general purpose compute on cxl memory," *arXiv preprint arXiv:2404.02868*, 2024.

[37] Q. Yang, R. Jin, B. Davis, D. Inupakutika, and M. Zhao, "Performance evaluation on cxl-enabled hybrid memory pool," in *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 2022, pp. 1–5.

[38] D. He, L. Chen, J. Liang, C. Kang, and J. Bai, "Numa-aware contention scheduling on multicore systems," in *2021 16th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*, 2021, pp. 452–457.

[39] Z. Chu, P. Jin, Y. Luo, X. Wang, and S. Wan, "Nobtree: A numa-optimized tree index for nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3840–3851, 2024.

[40] S. Sameer and V. R. Chintapalli, "Numa-aware parallelized service function chain deployment in multi-core servers," in *2024 16th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, 2024, pp. 445–447.

[41] V. Geraeinejad, Q. Qian, and M. Ebrahimi, "Investigating register cache behavior: Implications for cuda and tensor core workloads on gpus," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 14, no. 3, pp. 469–482, 2024.

[42] S. Yun, H. Nam, K. Kyung, J. Park, B. Kim, Y. Kwon, E. Lee, and J. H. Ahn, "Clay: Cxl-based scalable ndp architecture accelerating embedding layers," in *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024, pp. 338–351.

[43] B. Hanindhito, B. Patel, and L. K. John, "Bandwidth characterization of deepspeed on distributed large language model training," in *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2024, pp. 241–256.

[44] A. Puri, J. Jose, T. Venkatesh, and V. Narayanan, "Dracksim: Simulator for rack-scale memory disaggregation," *arXiv preprint arXiv:2305.09977*, 2023.

Yue Ma is currently a Ph.D. student at Tsinghua Shenzhen International Graduate School. Her research interests include storage circuit design, computer architecture, high-speed interconnect protocols and distributed computing. Her current work focuses on key technologies for storage optimization in heterogeneous computing environments, including memory pooling, disaggregated memory architecture, and storage server systems. She has also participated in several projects related to high-performance computing.

Dongwei Xu is currently pursuing a master's degree at Tsinghua Shenzhen International Graduate School. He obtained his bachelor's degree in Microelectronics Science and Engineering from Zhejiang University in 2023. His research focuses on training optimization for recommendation models, also with high interest in computer architecture, and distributed computing.

Peng He is currfently a Ph.D. student jointly trained by Tsinghua University and Pengcheng Laboratory. He received his Master of Engineering degree from the Institute of Aerospace Information Research Institute, University of Chinese Academy of Sciences. His current research interests include disaggregated large language model (LLM) inference and memory expansion technologies based on Compute Express Link (CXL). During his master's studies, he published three first-author papers in ICMMT, CSR-SWTC, and IEEE IVEC.

Hao Fu is currently pursuing a master's degree at Tsinghua Shenzhen International Graduate School. He obtained his bachelor's degree in Electronic Engineering from Tsinghua University in 2023. His research focuses on optimizing CXL memory expansion for memory-intensive workloads. His interests also include computer architecture, large language model (LLM) inference, and distributed computing.

Ying Jing is currently pursuing a master's degree at Tsinghua Shenzhen International Graduate School. She obtained her bachelor's degree in Electronic Engineering from Harbin Institute of Technology in 2023. Her research focuses on disaggregated memory systems, with a particular emphasis on memory pooling based on CXL and low-latency memory access optimization.

Sheng Zhang received the Ph.D. degree from Tsinghua University, China, in 2004. He is currently an Associate Professor with the Graduate School at Shenzhen, Tsinghua University. His research interests include distributed storage systems, computer architecture, and large language model (LLM) inference.

Fangming Liu (S'08, M'11, SM'16) received the B.Eng. degree from the Tsinghua University, Beijing, and the Ph.D. degree from the Hong Kong University of Science and Technology, Hong Kong. He is currently a Full Professor with the Huazhong University of Science and Technology, Wuhan, China. His research interests include cloud computing and edge computing, datacenter and green computing, SDN/NFV/5G and applied ML/AI. He received the National Natural Science Fund (NSFC) for Excellent Young Scholars, and the National Program Special Support for Top-Notch Young Professionals. He is a recipient of the Best Paper Award of IEEE/ACM IWQoS 2019, ACM e-Energy 2018 and IEEE GLOBECOM 2011, the First Class Prize of Natural Science of Ministry of Education in China, as well as the Second Class Prize of National Natural Science Award in China.

Bowen Wang is currently a Ph.D. student at Tsinghua Shenzhen International Graduate School, supervised by Professor Sheng Zhang. He received his bachelor's degree from Department of Automation at Tsinghua University in 2023. His recent research interests focus on continual learning in LLM, distributed intelligence, and multi-agent collaboration. Currently, he is a visiting student at Pengcheng Lab, where he is the co-lead author of a paper on knowledge fusion and catastrophic forgetting in multi-agent systems.

Haiyuan Wan is currently a Ph.D. student at Tsinghua Shenzhen International Graduate School. He received his bachelor's degree in Communication Engineering from East China Normal University in 2024. His research interests include large language models, continual learning, multi-agent reasoning, and system-level optimization for efficient inference.

Zhirun Yue is currently a Engineering Ph.D. Candidate at Tsinghua Shenzhen International Graduate School. He received his bachelor's degree in Electronic and Information Engineering from Xidian University in 2023. His research interests lie in artificial intelligence, LLM architecture, and distributed computing. Currently, his research focuses on the optimization of LLM inference for memory-intensive tasks, aiming to enhance system performance and efficiency.