Performance Characterization of CXL Memory and Its Use Cases

Xi Wang*
University of California, Merced
Merced, CA, USA
swang166@ucmerced.edu

Jie Liu*
University of California, Merced
Merced, CA, USA
jliu279@ucmerced.edu

Jianbo Wu
University of California, Merced
Merced, CA, USA
jwu323@ucmerced.edu

Shuangyan Yang
University of California, Merced
Merced, CA, USA
syang127@ucmerced.edu

Jie Ren
William & Mary
Williamsburg, VA, USA
jren03@wm.edu

Bhanu Shankar
MemVerge, Inc
Milpitas, CA, USA
bhanu.shankar@memverge.com

Dong Li
University of California, Merced
Merced, CA, USA
dli35@ucmerced.edu

***Abstract*—Compute eXpress Link (CXL) is emerging as a promising memory interface technology. However, its performance characteristics remain largely unclear due to the limited availability of production hardware. Key questions include: What are the use cases for the CXL memory? What are the impacts of the CXL memory on application performance? How to use the CXL memory in combination with existing memory components? In this work, we study the performance of three genuine CXL memory-expansion cards from different vendors. We characterize the basic performance of the CXL memory, study how HPC applications and large language models (LLM) can benefit from the CXL memory, and study the interplay between memory tiering and page interleaving. We also propose a novel data object-level interleaving policy to match the interleaving policy with memory access patterns. Our findings reveal the challenges and opportunities of using the CXL memory.**

I. Introduction

Compute eXpress Link (CXL) is a promising memory interface technology. Based on the standard PCIe serial interface, CXL attaches memory to the CPU and appears as a CPU-less NUMA node. The CXL memory can be accessed in a cache-coherent fashion using load/store instructions. However, CXL memory introduces longer memory access latency. This longer latency comes from PCIe, CXL memory controller, and CXL home agent (HA) on the CPU. Figure 1 compares local NUMA, traditional remote NUMA, and CXL-based memory expansion in terms of memory latency.

Given the CXL performance, we face a series of questions: what are the use cases for the CXL memory? What are the impacts of real CXL memory on application performance? At the application level, how to use the CXL memory in combination with fast memory components (e.g., using uniform page-level interleaving vs. data object-level interleaving vs. memory binding)? This paper aims to discuss those questions, and explore various paths to use the CXL memory. We study three genuine CXL memory expansion cards instead of using memory simulation or emulation.

*Co-first authors.

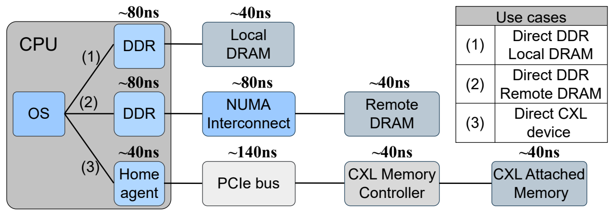

Fig. 1. Breakdown of CXL memory access latency.

The CXL memory is a unique "NUMA node". The CXL memory appears as a CPU-less NUMA node. In the three systems we evaluate, the CXL memory appears as a two-hop-away NUMA node in terms of access latency. Depending on the memory vendors, the peak bandwidth of the CXL memory varies a lot, ranging from 9.8% to 80.3% of the peak bandwidth of local DRAM. Moreover, as we increase the number of threads accessing the CXL memory, its bandwidth is quickly saturated due to the limited data transfer rate in CXL attached memory instead of PCIe bus — with saturation occurring when the number of threads reaches just four. In contrast, local DRAM (LDRAM) and remote DRAM-based NUMA nodes (RDRAM) exhibit much better scalability. This scaling difference between the CXL memory and DRAM highlights the importance of appropriately distributing memory accesses between them for high performance. Also, when the system is under heavy load, we observe that the latencies of LDRAM and RDRAM are similar to that of the CXL memory because of the contention on the memory controller (MC) or data path, which shows the potential of using CXL as LDRAM or RDRAM for latency-sensitive applications.

Using the CXL memory for large language models (LLM) faces challenges. LLM can be memory-consuming and have execution phases sensitive to memory bandwidth [27], [46], [54], hence having potentials to benefit from CXL. We study the cases that use the CPU memory as an extension to the GPU memory to enable LLM training (using ZeRO-Offload [54]) and inference (using FlexGen [63]) such that we can use less GPUs for LLM without the constraint of GPU memory capacity. This method has to frequently copy tensors between the GPU and CPU, and offloads certain computations from the CPU to the GPU to maximize GPU memory saving or reduce I/O offload. This method (named *tensor offloading* in the rest of the paper) has been commonly studied and deployed in industry.

We find that using tensor offloading based upon the CXL memory brings limited performance improvement. This is because the tensor copy between the CXL memory and GPU memory goes through a longer data path than memory accesses directly from CPU. Using CXL 1.1 on our platform, this data path is bottlenecked by the PCIe interconnect between CPU and GPU. As a result, adding CXL cannot show the bandwidth benefits for tensor offloading. In contrast, the computation offloaded to the CPU (i.e., the optimizer for LLM training and the attention computation for LLM inference), which tends to be bandwidth sensitive, can benefit from the extra bandwidth of CXL. In addition, the CXL memory increases memory capacity and allows us to use larger batch sizes for LLM inference, leading to throughput gains. Our study is unprecedented, because of its practicality of using real CXL for GPU-based LLM, different from CPU-based study [68].

CXL can be used to save fast memory without causing performance loss, and using application semantics to guide page interleaving for CXL can maximize CXL benefits. Our work goes beyond the existing work [67], [68] that focuses on applications exhibiting ms-scale latency (e.g., social network microservices) or commercial workloads to study the potential of the CXL memory. We study a spectrum of HPC workloads, covering the most common and representative "HPC dwarfs" [6]. We reveal that some HPC applications (such as CG and BT [15]) can tolerate the low bandwidth and high latency of the CXL memory under certain scales (the performance loss is less than 3.2%, compared with LDRAM), because of their compute-intensive nature.

In addition, the page interleaving policy, embraced by the industry (e.g., Micron and AMD [47], Astera Labs [30], and Samsung [48]) as an application-transparent technique to integrate CXL with the existing memory components, provides opportunities to save LDRAM for HPC applications. When interleaving CXL and RDRAM, we see minor performance difference from interleaving CXL and LDRAM for some applications. This is because the CXL memory dominates the memory performance, and the performance of other memory components has minor impact on the overall performance.

To maximize the interleaving performance, we introduce a novel data object-level interleaving policy. Different from Linux uniform page-level interleaving [67], this policy decides whether memory pages allocated to a data object should be interleaved between CXL and DRAM or allocated to LDRAM first ("LDRAM preferred"). This policy maximizes memory bandwidth (or minimizes latency) for data objects whose accesses favor high bandwidth (or low latency). This policy reduces LDRAM usage by 32% and outperforms the uniform interleaving policy (Linux default) by 65% on average.

Memory tiering solutions need to be improved. Treating the CXL memory as a memory tier, existing work [13], [32], [44], [50], [55], [73] migrates pages between the CXL memory and fast memories based on page access frequency or recency (i.e., hotness). Those solutions are seldom studied with the real CXL memory, and *how they interplay with the existing system (e.g., page interleaving) is largely unknown.*

We find that the dynamic page migration in memory tiering are not integrated well with the static page interleaving, because of invalidness of NUMA hint faults. Depending on temporal and spatial distribution of hot pages, the dynamic page migration can degrade performance compared to no migration. We also observe that the old-fashioned NUMA first touch and Tiering-0.8 [73] (the most recent Linux Patch for AutoNUMA to support memory tiering) is very effective, outperforming a set of page migration and interleaving solutions.

II. Background

A. Compute Express Link

The CXL specification defines three protocols: `CXL.io`, `CXL.cache`, and `CXL.mem`. There are three types of CXL devices. The type-3 device is related to our evaluation. Such a device supports `CXL.io` and `CXL.mem`, and is used for memory bandwidth or capacity expansion in memory tiering. The CXL specification has been going through three major versions: 1.1, 2.0, and 3.0. CXL 1.1 focuses on directly-attached CXL devices, CXL 2.0 incorporates switch-based pooling, and CXL 3.0 supports switch-less pooling and higher bandwidth.

Most of the real CXL devices nowadays are host-managed device memory with host-only coherent (HDM-H) *using CXL 1.1*. The three devices for our evaluation are among them.

B. CXL Systems for Evaluation

CXL requires compatible hardware in both the CPUs and peripheral devices. The 4th-generation Intel Xeon Scalable Processors (such as Sapphire Rapids) and the 4th-generation AMD EPYC Processors (such as Genoa) are among the first mainstream server CPUs to support the CXL 1.1. Several CXL memory devices have been developed as commercial products by leading hardware manufacturers such as Micron. We use three CXL devices from three vendors. See Table I.

The system A in Table I has two sockets (0 and 1), with a CXL device attached to Socket 1 by CXL link over PCIe 5.0. Accessing CXL memory from Socket 0 goes through HyperTransport interconnect, leading to longer latency than accessing from Socket 1. From the view of a CPU, there are three NUMA nodes: local DDR (LDRAM), remote DDR on the other socket (RDRAM), and CXL memory. The system B has the same organization as A. The system C has a different organization: the CXL device is attached to Socket 0 (not 1).

III. Basic Performance Characteristics

Evaluation methodology. We evaluate memory latency and bandwidth using Intel Memory Latency Checker (MLC) [22]. MLC disables hardware prefetcher for Intel processors (the systems B and C), but cannot do so for AMD processors (the system A). For latency tests, MLC uses typical pointer

TABLE I
THREE SYSTEMS WITH CXL DEVICES.

Sys	Component	Description
A	OS (kernel)	Ubuntu 22.04 LTS (Linux kernel v6.2.15)
	CPUs	2× AMD EPYC 9354 CPUs @3.8 GHz, 32 cores and 512 MB LLC per CPU
	PCIe	PCIe 5.0, speed 32GT/s, 16 lanes
	Memory	Socket 0: 12× DDR5-4800 channels, memory 768GB Socket 1: 12× DDR5-4800 channels, memory 768GB max bandwidth 460.8 GB/s per socket
	CXL A	Single channel DDR5-4800, memory 128 GB, max bandwidth 38.4 GB/s per channel
B	OS (kernel)	Fedora Linux 36 (Linux kernel v6.6.0-rc5)
	CPUs	2× Intel(R) Xeon(R) Platinum 8470 CPU @2.0GHz, 52 cores and 210 MB LLC per CPU (Saphire Rapids)
	PCIe	PCIe 5.0, speed 32GT/s, 16 lanes
	Memory	Socket 0: 8× DDR5-4800 channels, memory 1TB Socket 1: 8× DDR5-4800 channels, memory 1TB max bandwidth 307.2 GB/s per socket
	CXL B	Single channel DDR5-8000, memory 64 GB, max bandwidth 64.0 GB/s per channel
C	OS (kernel)	Ubuntu 22.04 (Linux kernel v6.2.15)
	CPUs	2× Intel(R) Xeon(R) Gold 6438Y+ @2.0GHz, 32 cores and 60 MB LLC per CPU
	PCIe	PCIe 5.0, speed 32GT/s, 16 lanes
	Memory	Socket 0: 8× DDR5-4800 channels, memory 512GB Socket 1: 8× DDR5-4800 channels, memory 512GB max bandwidth 307.2 GB/s per socket
	CXL C	Dual channel DDR5-6200, memory 128 GB, max bandwidth 48.4 GB/s per channel

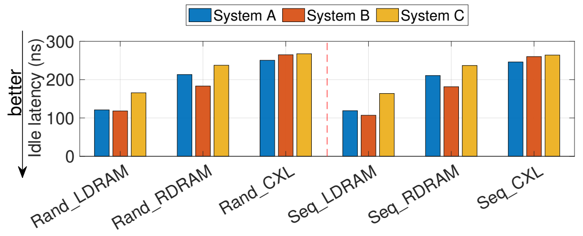

Fig. 2. Load latency with random and sequential accesses to a cache block.

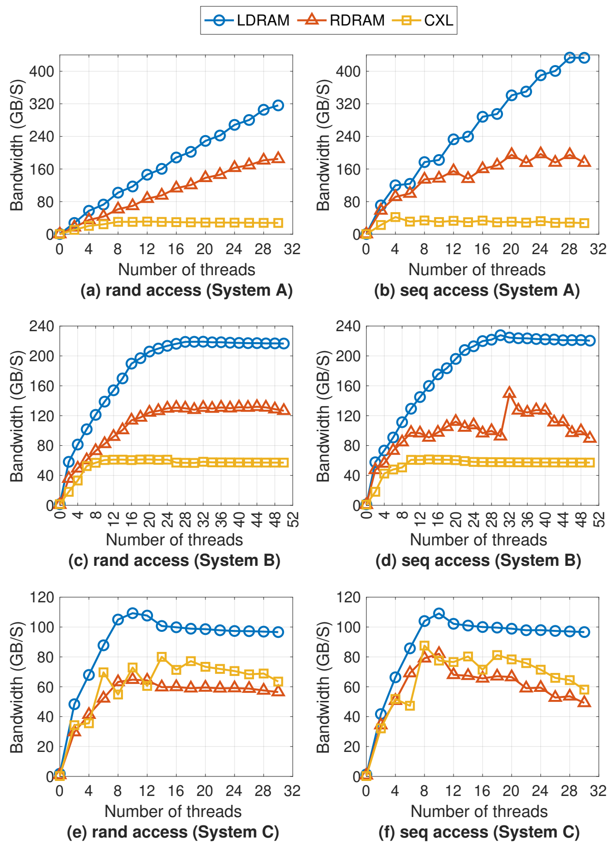

Fig. 3. Bandwidth scaling for data loading. Note that the scales of the figures for the three systems are different.

chasing. For each latency test, we repeat the test 5,000 times, and report the average value after excluding outliers (caused by operating system services and random TLB misses). For bandwidth tests, we use MLC to perform sequential and random memory accesses. The sequential accesses in combination with thread-level parallelism introduces parallel memory accesses, revealing peak memory bandwidth. For each bandwidth test, we repeat the test 2,000 times, and report the average value.

Latency results. See Figure 2. Compared with LDRAM, CXL is much slower than one-hop-away NUMA node (RDRAM). In fact, assuming that adding a hop of NUMA distance introduces a constant latency in a system, *the CXL memory is comparable to a two-hop-away NUMA node, in terms of access latency*. Figure 2 shows that the CXL memory from different vendors show quite different latency. For example, for sequential accesses, the CXL memory in the system A adds latency by 153 ns, while the CXL memory in the system B adds latency by 211 ns, compared to LDRAM. Since the two systems use the same PCIe and DRAM technologies, such a latency difference mainly comes from the difference in the CXL controller and HA on the CPU.

Bandwidth results. Figure 3 shows that bandwidth scaling of LDRAM, RDRAM, and CXL are different as we change the number of threads. The CXL memory bandwidth is saturated as the number of threads is over 8, while the saturation points for LDRAM and RDRAM are much higher than that in CXL (e.g., 28 and 20 on the system B), because the bandwidth of CXL memory is constrained by the bandwidth of a single DDR channel. The peak CXL memory bandwidth is lower than that of RDRAM (the CXL memory bandwidth is 17.1% and 46.4% of the RDRAM bandwidth on the systems A and B respectively), but can be close to the RDRAM bandwidth (see the system C).

The difference in bandwidth scaling between LDRAM, RDRAM, and CXL highlights the importance of distributing memory accesses between them. For example, in the system B, to maximize the bandwidth usage, we would assign 6, 23, and 23 threads to access CXL, LDRAM, and RDRAM respectively, because increasing thread counts beyond these points does not improve the bandwidth, shown in Figure 3(d). *Using the above thread counts can lead to a peak bandwidth of 420 GB/s, larger than any other thread assignment.*

Performance under load. We study memory latency and bandwidth under varying load. Figure 4 presents the results of how latency and bandwidth vary by gradually increasing the load on memory. For this test, we employ Intel MLC, utilizing 32 threads. In our methodology, each thread performs memory accesses to cache lines and delays for a time interval between

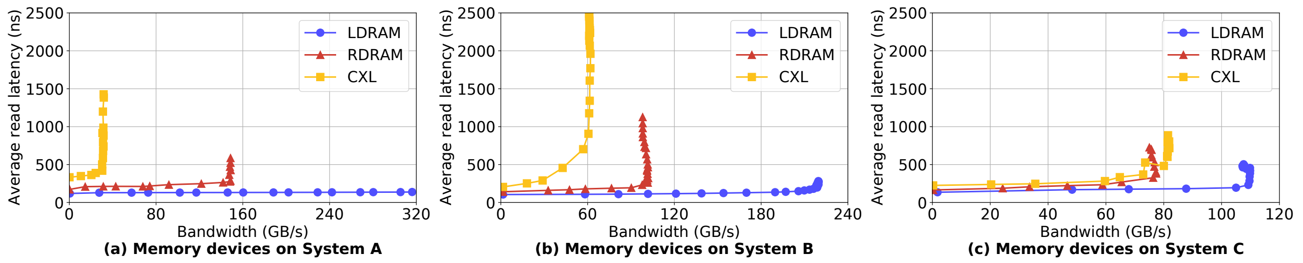

Fig. 4. Memory latency and bandwidth under varying load.

two accesses. We vary the time interval from 0 to 80μs. This setup ensures that each worker thread is engaged in repeated, sequential memory accesses, allowing for a detailed analysis of how varying load conditions impact memory performance. When the time interval is 0 (corresponding to the right side of each subfigure in Figure 4), the bandwidth is close to the maximum bandwidth and the latency skyrockets as the queuing effects in hardware dominate. When the time interval is high enough (80μs, corresponding to the left side of each subfigure in Figure 4), the latency is close to the raw, unloaded latency.

Basic observation. The latency of accessing LDRAM and RDRAM can be similar to the CXL memory when reaching their peak memory bandwidth. For example, in Figure 4(c), once the bandwidth consumption of LDRAM and RDRAM approaches the peak (110GB/s for LDRAM and 84GB/s for RDRAM), their latencies are as high as 543 ns and 600 ns respectively, which are pretty close to the CXL memory latency (400 ns - 550 ns) when approaching the peak CXL bandwidth under heavy load. *This demonstrates the potential of using CXL as LDRAM and RDRAM under heavy load.*

Takeaway: CXL memory performs as a NUMA node with latency similar to RDRAM but lower (or comparable) bandwidth. Different from the traditional NUMA node, the CXL memory is unique in terms of performance scalability and performance under heavy load.

IV. CXL for Large Language Models

LLMs are crucial for powering various AI applications, but their substantial memory footprint presents deployment challenges. We study the performance of LLMs with *tensor offloading* techniques using the CXL memory, a promising solution to address the constraint of GPU memory capacity. Tensor offloading moves tensors out of GPU memory when they are not in use, allowing for the execution of larger models that exceed the GPU's memory capacity. *The evaluation is conducted on the system A*, featuring an NVIDIA A10 GPU with 24GB memory, connected to the host CPU via PCIe Gen 4, offering a maximum bandwidth of 32GB/s.

The CXL memory on the system A uses CXL 1.1, which does not allow the GPU to directly access the CXL memory; instead, the accesses must go through the CPU. Under CXL 1.1, the data path from the GPU to the CXL memory is "GPU - PCIe - CPU - PCIe - CXL memory", longer than the direct "CPU - PCIe - CXL memory" path. The data path in CXL 1.1 is different from the CXL devices peer-to-peer access supported in CXL 3.1 (using "GPU - PCIe - CXL memory").

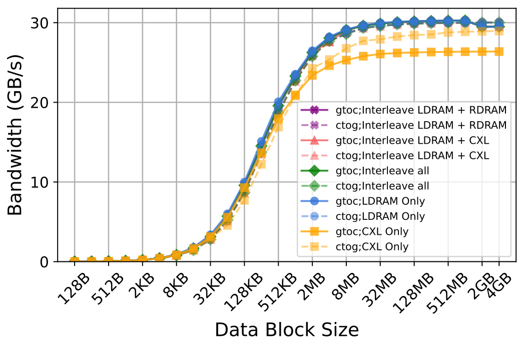

Fig. 5. Data transfer bandwidth between GPU (i.e., "g") and CPU (i.e., "c").

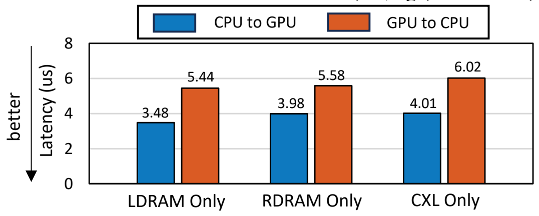

Fig. 6. 64-byte data transfer latency between the GPU and CPU.

We explore the effects of the CXL 1.1 data path on the data transfer bandwidth and latency.

Bandwidth results. We measure memory access bandwidth by repeatedly and randomly copying data blocks between the GPU memory (labeled as "g") and CPU memory (labeled as "c") using various page allocation strategies. The size of the data blocks varies from 128 byte to 4GB, illustrated in Figure 5. We use membind [5] to specify the memory device as the source or destination for data transfers.

LLM basic observation 1. The memory bandwidth for the GPU access to the memory hierarchy with CXL is constrained by the PCIe bandwidth between the CPU and GPU, due to the absence of peer-to-peer access support in CXL 1.1.

Counter-intuitively, using the CXL memory does not increase the memory bandwidth for the GPU. As shown in Figure 5, the peak memory bandwidth is similar across various memory interleaving policies (the difference is less than 3%). This lack of bandwidth increase with the CXL memory is primarily due to the PCIe interconnect between the CPU and GPU acting as a performance bottleneck.

Latency results. We develop a microbenchmark to measure the data transfer latency. The microbenchmark runs on CPU 1 (the CPU close to the CXL memory) and uses `cudaMemcpy()` for data transfer. The benchmark repeatedly transfers a 64-byte data (a cache block) between the CPU and GPU. The transfer happens 100K times, and we report the average time for one transfer. We use membind, similar to the

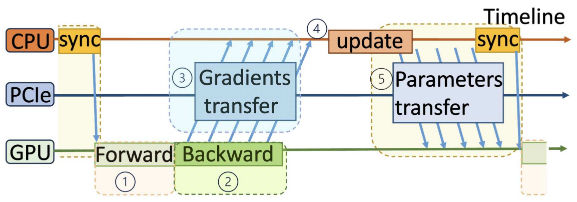

Fig. 7. Overview of ZeRO-Offload in a training step.

bandwidth tests. Figure 6 shows the results.

LLM basic observation 2. Accessing the CXL memory from the GPU can result in longer latency than expected.

The difference of data transfer latency between "GPU - CXL memory" and "GPU - CPU memory" (in Figure 6) is greater than the latency difference between "CPU - CXL memory" and "CPU - CPU memory" (in Figure 2). For example, accessing the CXL memory from the GPU is, on average, 500 ns longer than accessing the CPU memory from the GPU. In contrast, accessing the CXL memory from the CPU is only 120 ns longer than accessing the CPU memory from the CPU. The longer latency-difference from the GPU side comes from the longer data-path between the GPU and CXL memory.

We study the implication of using the CXL memory for LLM training and inference using tensor offloading on System A, as follows.

A. LLM Training

1) ZeRO-Offload Background: ZeRO-Offload [54] is commonly employed in the industry to enable larger LLM *training* using smaller GPU memory. To save the GPU memory, ZeRO-Offload efficiently offloads full-precision model parameters, gradients, and optimizer states (e.g., momentum and variance) to the CPU memory, moving them back to GPU memory when needed. Figure 7 depicts the workflow of ZeRO-Offload. Specifically, it ① performs forward and ② backward computation on GPU; and ③ offloads gradients to the CPU memory during the backward step. To reduce the overhead of tensor movement between CPU and GPU ④, ZeRO-Offload performs optimization computation (e.g., the ADAM optimizer) on the CPU; and ⑤ moves updated parameters from the CPU memory to the GPU memory before the next forward step. This strategy minimizes data movement volume between the GPU and CPU memory for each training step. As a result, ZeRO-Offload enables 10× larger model training on a single GPU with 1.4× higher throughput.

2) Using CXL Memory: We evaluate two LLMs: BERT [14] and GPT2 [49]. For BERT, we consider three configurations: 110 million (base), 340 million (medium), and 4 billion (large) parameters. For GPT2, we evaluate models with 4, 6, and 8 billion parameters. Figure 8 shows the performance with various interleaving policies and model sizes. We use the notation "bs=effective batch size@model size" to represent the batch size and number of model parameters. For a given model size, the batch size is chosen to be the maximum without causing an out-of-memory (OOM) error on the GPU. To better understand the performance, we break it down to the "optimization step" (i.e., the ADAM optimizer on the CPU, which is exposed to the critical path) and "data movement"

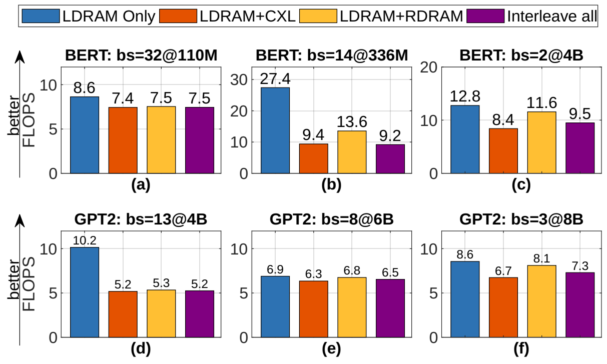

Fig. 8. Performance with various interleaving policies and model sizes for BERT and GPT2.

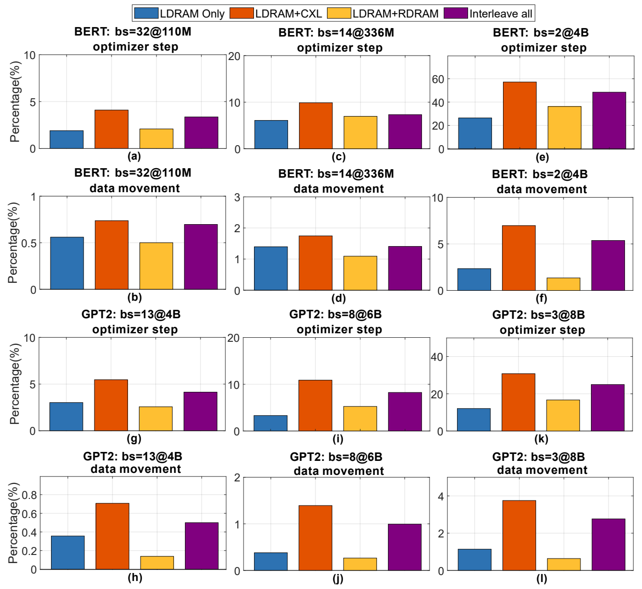

Fig. 9. Performance of the optimizer and data movement under various interleaving. The percentage numbers are in terms of total training time.

(i.e., the gradient transfer from the GPU to the CPU and the parameter transfer from the CPU to the GPU). The memory capacities for various interleaving policies (i.e., LDRAM only, LDRAM+CXL, LDRAM+RDRAM, and "interleave all") are 196GB, 324GB, 392GB, and 520GB. Figure 9 shows the results, including data movement exposed to the critical path.

Evaluation results. LLM training observation 1. Using the CXL memory brings little performance improvement or even negative performance impact to ZeRO-Offloading.

See GPT2. Figure 8 shows that the performance difference between LDRAM+RDRAM, LDRAM+CXL, and "interleave all" is less than 5% for the 4B and 6B models. For the 8B, LDRAM outperforms 'interleave all" by 14%, and LDRAM+RDRAM outperforms LDRAM+CXL by 16%. There is no performance benefit of using CXL.

Figure 9 reveals three reasons for the above observation. *(1)* The data movement takes a rather small portion of the training time in GPT2 (less than 5% in all cases). *(2)* Because of the bottleneck in the extra PCIe, the data movement does not get benefits from the better bandwidth offered by interleaving CXL and DDR (the CPU memory). Instead, the data movement suffers from longer access latency of the

CXL memory. As a result, compared with using LDRAM, using CXL actually increases data transfer time and has a minor impact on the training time. *(3)* The optimizer takes a larger portion of the training time, compared to the data movement. The optimizer happens on the CPU and is sensitive to memory latency. Using CXL increases its execution time (2%-18%), compared to using LDRAM. When the batch size is small, the optimizer takes a significant portion of the training time (e.g., 31% when bs=3@8B). In such cases, optimizer slowdown substantially decreases overall training throughput. For example, when bs=3@8B, using "interleave all" performs worse than LDRAM+RDRAM by 11% due to the worse performance in the optimizer by 8%. We have the similar observations for BERT.

We expect that using a larger model (such as GPT-3 with 175 billion parameters), which has larger numbers of parameters and gradients, would result in data movement taking a larger portion of training time to transfer parameters and gradients. Unfortunately, we cannot evaluate such a large model due to the limited memory capacity on our GPU. In addition, after reducing the data path between the GPU and CXL memory, the CXL memory can play a bigger role in reducing training time. Also, using the interleaving policy for the optimizer is not good for performance because of the latency sensitive nature of the optimizer. Using the first touch or "preferred policy", the optimizer performance can be improved.

B. LLM Inference

1) FlexGen Background: LLM inference is memory-consuming. FlexGen [63] is a cutting-edge framework designed for LLM *inference* with constrained GPU memory capacity. To address the memory capacity limitation, FlexGen offloads model parameters, KV cache, and activations to the host CPU memory hierarchy (including DRAM and NVMe SSD). Figure 10 shows the workflow of FlexGen. The LLM inference consists of two stages: *prefill* and *decode*.

During the *prefill stage*, which happens only once per inference batch, ① FlexGen transfers parameters from the CPU memory hierarchy to the GPU. ② FlexGen executes attention and MLP computation layer by layer on the GPU. ③ At the end of each attention layer, the generated KV cache is offloaded to the CPU memory hierarchy. The *decode stage* generates tokens and significantly influences the overall throughput of the inference process. To minimize tensor movement between the GPU and CPU, ④ FlexGen conducts attention computation directly on the CPU. ⑤ FlexGen then transfers model parameters and activations generated in the attention layer from the CPU to the GPU for MLP computation, and ⑥ transfers activations generated in the MLP layer to the CPU for the following computation.

Offloading policy and cost model. FlexGen allows tensors to be partially placed in the CPU memory hierarchy and uses a cost model to determine the optimal offloading policy for maximum inference throughput within a memory capacity constraint. The cost model considers latency and bandwidth differences between NVMe and DRAM but does not differentiate among LDRAM, RDRAM, and CXL memory.

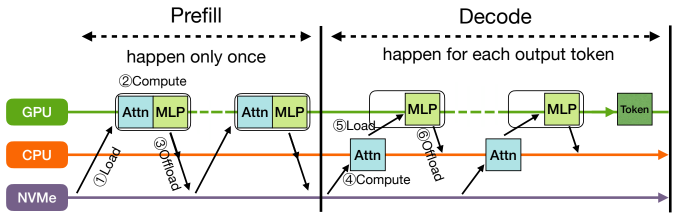

Fig. 10. Overview of LLM inference with FlexGen.

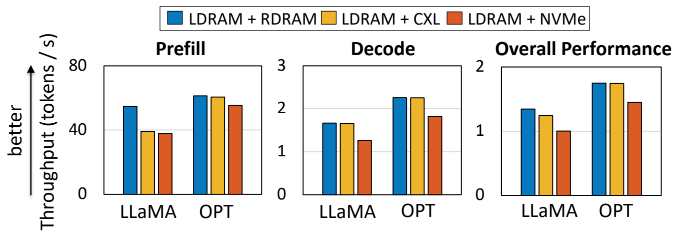

Fig. 11. Comparison LLM inference throughput across memory systems, each with 324 GB capacity.

2) Using CXL Memory: **Evaluation setup.** We evaluate LLaMA [69] with 65 billion parameters and OPT [93] with 66 billion parameters. The batch sizes for evaluation are determined by performing a linear programming-based [66] policy search to maximize inference throughput while respecting memory and hardware constraints. The lengths of the input prompts and output tokens are standardized at 2,048 and 256, respectively. The evaluation platform has 196 GB LDRAM, 196 GB RDRAM, 128 GB CXL, and 128 GB NVMe. Leveraging GRUB mmap and numactl [5], we build the host memory hierarchy with various capacities and media types.

Evaluation results. LLM inference observation (LIO) 1. The performance of the CXL memory is comparable to that of RDRAM, and surpasses NVMe when applied in LLM inference with the tensor offloading.

Figure 11 presents the results. Each evaluation uses two equal-sized memory medias with a total capacity of 324 GB. According to FlexGen's tensor offloading policy, only 8% of the KV cache resides on the GPU, and the remaining KV cache, weights, and activations reside on the CPU. Figure 11 shows that the inference throughput using LDRAM + CXL is similar to that of LDRAM + RDRAM, with the difference being less than 3%. Furthermore, LDRAM + CXL shows an improvement of 24% for LLaMA and 20% for OPT in overall throughput, compared to LDRAM + NVMe.

LIO 2. The throughput of prefill and decode stages responds differently to memory latency and bandwidth.

Figure 11 shows that during the prefill stage, the throughput difference in the prefill stage across the three interleaving policies largely reflects the trend of latency difference in the memory systems. For example, LDRAM + RDRAM outperforms LDRAM + CXL and LDRAM + NVMe by 20% and 28% on average, respectively. This is because during the prefill stage, tensors are loaded from the CPU to the GPU and such data loading is sensitive to the latency.

TABLE II
LLM INFERENCE CONFIGURATION FOR EVALUATION. "BS" AND "C" STANDS FOR "BATCH SIZE" AND "KV CACHE", RESPECTIVELY. ACCORDING TO OFFLOADING POLICY IN FLEXGEN, ALL WEIGHTS AND ACTIVATIONS ARE STORED IN CPU MEMORY.

LLM	Memory hierarchy	BS	*c* on GPU	*c* on CPU	Memory footprint
LLaMA	LDRAM Only (196 GB)	14	20%	80%	200 GB
LLaMA	LDRAM + RDRAM (392 GB)	40	4%	96%	348 GB
LLaMA	Interleave all (520 GB)	56	4%	96%	438 GB
OPT	LDRAM Only (196 GB)	9	27%	73%	168 GB
OPT	LDRAM + RDRAM (392 GB)	40	4%	96%	326 GB
OPT	Interleave all (520 GB)	64	4%	96%	448 GB

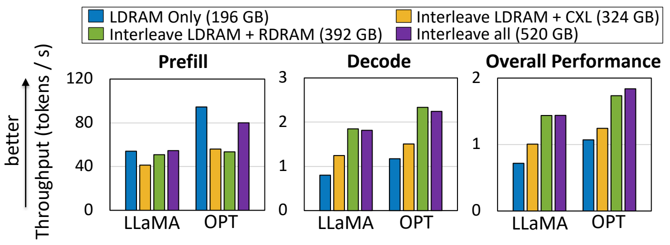

Fig. 12. Comparison of LLM inference throughput across various memory systems with different capacities.

In contrast, during the decode stage, the throughput is more sensitive to memory bandwidth (see Figure 11). The decoding throughput using LDRAM + CXL is 27% better than using LDRAM + NVMe on average. LDRAM + RDRAM and LDRAM + CXL perform similarly. This is because in the decode stage, the attention computation happens on the CPU, leading to intensive data access and sensitive to large bandwidth difference between CXL and NVMe (but not relatively small difference between CXL and RDRAM).

LIO 3. CXL increases the capacity of the memory system, thereby enhancing the throughput of LLM inference due to larger batch size.

Figure 12 illustrates the inference throughput with various capacities of the CPU memory. Table II summarizes the offloading policy search results with those capacities. The LLM batch size scales up with the increased memory system capacity. Specifically, with LDRAM + CXL, LDRAM + RDRAM, and LDRAM + RDRAM + CXL, the batch size increases by 1.14×, 1.85×, and 3× for LLaMA, and 2.11×, 3.44× and 6.11× for OPT, respectively, compared to the case of utilizing LDRAM only. The overall throughput increases by 28%, 81% and 86% on average respectively, compared to LDRAM only.

We further decompose performance into prefill and decode. During the prefill stage, the inference throughput is dominated by the latency of GPU accessing the CPU memory hierarchy, hence LDRAM only outperforms LDRAM + CXL, LDRAM + RDRAM, and LDRAM + RDRAM + CXL by 50%, 41%, and 9% on average, respectively. However, during the decode stage, as the batch size increases, the inference throughput improves by 42%, 114%, and 109% on average for LDRAM + CXL, LDRAM + RDRAM, and LDRAM + RDRAM + CXL respectively, compared with LDRAM only.

Takeaway: (1) The CXL memory as a memory capacity expander enables LLMs to train and infer with larger batch sizes, which in turn improves system throughput. However, when the GPU accesses the CXL memory, the lack of peer-to-peer access support in CXL 1.1 prevents them from leveraging the extra bandwidth and results in longer latency due to the extended data path. (2) Computation offloaded to the CPU can benefit from extra CXL bandwidth.

V. HPC WORKLOADS

We analyze HPC workload performance, focusing on seven workloads from NPB [15] and XSBench benchmark [70], which require processing a large working memory set using multiple threads, are summarized in Table III. They collectively cover the most common and representative HPC applications [6]. We use various memory allocation policies, including preferred and interleaving. The preferred policy for a specific memory node indicates that the memory is allocated in that memory node first; when that memory node runs out of space, the page allocation goes to another memory node closet to the CPU according to the NUMA distance. The interleaving policy indicates that pages are allocated among memory nodes in a round robin fashion. We present evaluation results on the system A, because we have limited accesses to B and C.

A. Evaluation Results

We have the following observations unseen in the existing CXL evaluation [67], [68].

HPC observation 1. When the interleaving involves the CXL memory, we can save LDRAM by using RDRAM.

Figure 13 shows the results with various interleaving policies, and the benchmarks run on CPU 0. We can see that the performance difference between the interleaving RDRAM+CXL and interleaving LDRAM+CXL is less than 9.2% for all benchmark. In general, we can save LDRAM while achieving similar performance by interleaving RDRAM+CXL instead of interleaving LDRAM+CXL.

The above results are because of large performance gap between the CXL memory and DDR (e.g., 2.1× and 1.2× longer in memory access latency, compared to LDRAM and RDRAM respectively.): because of data dependency and limited hardware resources (e.g., Miss Status Handling Registers (MSHR) and the queues in MC), the performance is highly impacted by the slow CXL memory and irrelevant whether the LDRAM or RDRAM is utilized.

HPC observation 2. Bandwidth-sensitive and latency-sensi -tive applications respond differently to the bandwidth increase offered by CXL.

Figure 14 shows the results for MG (bandwidth-sensitive) and CG (latency-sensitive). For MG, we see that when the number of threads increases from 4 to 32, the performance of "interleave all" (i.e., interleaving between LDRAM, RDRAM and CXL, which achieves the highest bandwidth) is consistently better than that of CXL preferred by 10%-85%. Also, for CG, the performance of "interleave all" performs worse than that of CXL preferred by up to 1.6×, because using CXL preferred, consecutive memory accesses tend to fall into

TABLE III
HPC WORKLOADS FOR EVALUATION. "BW" STANDS FOR "BANDWIDTH".

	Type	Workload Characterization	Input Problem	Mem. footprint	BW-hungry Objects
BT	Dense linear algebra	Unit-strided memory accesses from dense matrices	Class E	166 GB	u(39.6G), rsh(39.6G), forcing(39.6G)
LU	Sparse linear algebra	Indexed loads and stores from compressed matrices	Class E	134 GB	u(39.6G), rsd(39.6G)
CG	Sparse linear algebra	Irregular memory accesses based on indirect indexing	Class E	134 GB	a(48.9G)
MG	Structured grids	Dynamic updates based on subdivided regular grids	Class E	210 GB	v(64.2G), r(73.4G)
SP	Structured grids	Intense floating-point computations for linear equations	Class E	174 GB	u(39.6G), rsh(39.6G), forcing(39.6G)
FT	Spectral method	Bandwidth-consuming matrix transpose	Class D	80 GB	u0(32.0G), u1(32.0G)
XSBench	Monte Carlo	Computation based on repeated random trials	Extra large	116 GB	nuclide_grids

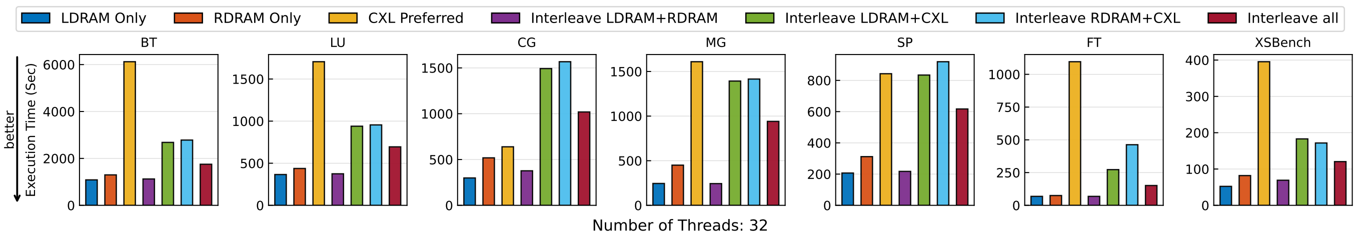

Fig. 13. The performance of various interleaving policies for HPC applications.

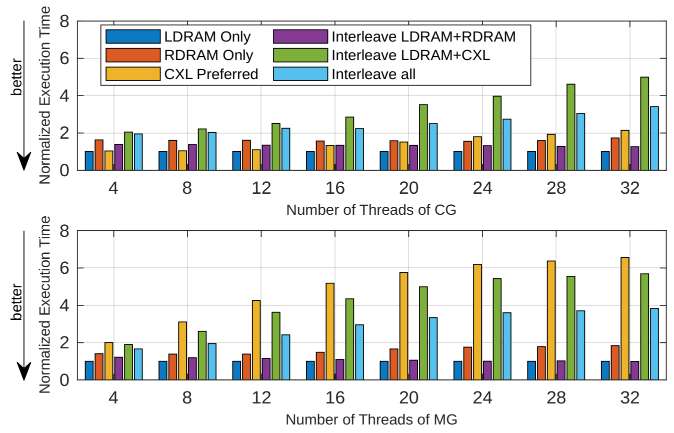

Fig. 14. The scalability study for CG and MG. The performance is normalized by that of using LDRAM only.

the same memory node, which is favored by CG for high performance (see more discussions for HPC observation 3).

HPC observation 3. The CXL memory can show unexpected high performance for the latency-sensitive applications with random memory accesses.

The CG results in Figures 13 and 14 support the above observation. CG is a latency-sensitive application because of its indirect indexing-based memory accesses. In Figure 14, CXL preferred performs better than using RDRAM-only by 10.9%-57.2% when the number of threads is 4-20. This seems to be counter-intuitive, because the CXL memory has higher latency than RDRAM. This superior performance comes from the optimization in the CXL device or customized caching policy in the processor for expensive, CPU-less memory accesses [67], and this optimization is especially effective for CG-style workloads. When the number of threads is larger than 20, the CXL memory accesses become more intensive, and CXL inferior performance becomes more obvious.

In Figure 13, we also observe that CXL preferred outperforms any other CXL-related interleaving policies in CG, despite other polices provide higher bandwidth and shorter average access-latency. This indicates that for a latency-sensitive workload with a random, scattered memory accesses, gathering accesses in one memory node instead of spreading to multiple memory nodes benefits performance because of reduction of row buffer misses in memory devices.

B. Object-Level Interleaving

Instead of generally interleaving pages across the entire application (named *uniform interleaving*), we propose a method to interleave pages at the data object level. This enables fine-grained control over how pages are interleaved. Since different data objects have different access patterns, using the fine-grained control allows the bandwidth-sensitive object to be accessed with high bandwidth, while the latency-sensitive object is allocated locally for high performance.

Interleaving method. To implement object-level interleaving, we employ `numa_alloc_interleaved_subset()` [5] in Linux, which allows for the allocation of a data object in an interleaved manner among specific NUMA nodes. We use two criteria to select data objects to use the interleaving policy.

- The object must have a large memory footprint, which means taking at least 10% of total memory consumption.
- Memory accesses to the object must be intensive; among the data objects that meet the first criterion, we select data objects with the largest number of memory accesses. Multiple data objects may be selected.

With high thread-level parallelism, memory accesses to the above data objects are more sensitive to memory bandwidth. Besides the above data objects, the memory allocation for other objects uses the "preferred" policy. The last column in Table III summarizes those bandwidth-hungry data objects selected for interleaving.

We evaluate the effectiveness of our object-level interleaving using the following approach. In each test, we run the workload on CPU 0 using both LDRAM (memory node 0) and CXL memory. LDRAM is limited to either 64GB or 128GB by using GRUB mmap, and the memory consumption of all applications exceeding 64GB, which allows us to assess the cases with both sufficient and insufficient LDRAM. The CXL memory is consistently 128GB. We make the following object-level interleaving observations (abbreviated as OLI observations).

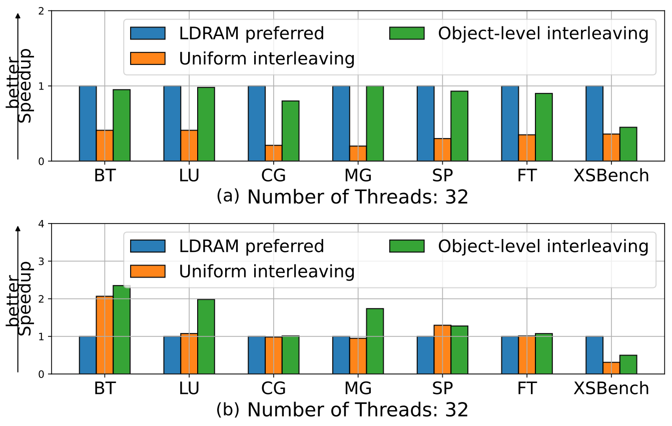

Fig. 15. (a) Performance speedup of various memory allocation strategies compared to "LDRAM preferred", using sufficient LDRAM (128GB). (b) Performance speedup of various memory allocation strategies, compared to "LDRAM preferred", using insufficient LDRAM (64GB).

<u>OLI observation 1.</u> When the local DRAM is sufficient (with 128 GB), the object-level interleaving performance is similar to that with LDRAM preferred, and consistently outperforms the uniform interleaving in all scenarios.

Figure 15(a) shows that OLI performance is 65% better than the uniform interleaving on average, demonstrating the effectiveness of OLI to meet the diverse requirements of latency-sensitive and bandwidth-sensitive objects. Additionally, *we observe that by using OLI, we can effectively reduce the fast memory size by an average of 32% (and up to 40% for FT), while still achieving similar performance as LDRAM preferred.* The performance difference between OLI and LDRAM preferred is less than 1% on average for all HPC workloads, except XSBench, when LDRAM is sufficient.

<u>OLI observation 2.</u> When the local DRAM is insufficient (64 GB), the OLI outperforms any other cases.

Figure 15(b) shows that on average, OLI performs 1.42× better than LDRAM preferred (and up to 2.35× for BT). It also outperforms the uniform interleaving by 1.32× on average (and up to 1.84× for LU). See the following reasons. (1) The performance of LDRAM preferred is highly related to when the data objects are allocated. If LDRAM is full, latency-sensitive objects might end up in the slower CXL memory. (2) The bandwidth-sensitive data objects do not have opportunities to be allocated on both LDRAM and CXL for high bandwidth.

In XSBench, LDRAM preferred performs better than both the uniform and object-level interleaving. This is because most of the memory accesses in XSBench are concentrated in a small, latency-sensitive memory set.

Takeaway: The CXL memory offers additional bandwidth, but HPC workloads may not benefit from it directly. The uniform interleaving can undermine performance, while the data object-level interleaving delivers performance comparable to or better than the default LDRAM-centric page allocation while saving fast memory size.

VI. Memory Tiering based on Page Migration

Memory tiering is an application-transparent solution to integrate the CXL memory into the existing memory systems. Treating the CXL memory as a memory tier, existing memory tiering solutions [13], [32], [44], [50], [55], [73] rely on memory profiling to count memory accesses at the page level. Then, those solutions move frequently accessed (hot) pages to the fast memory tier, and demote less frequently access (cold) pages to the slow memory tier. In contrast to the interleaving studied in Sections IV- V, which are static page placement solutions, the memory tiering solutions are dynamic.

We use the system A. We limit the capacity of the fast memory (i.e., LDRAM), while the capacity of the slow memory (i.e., CXL) remains the same.

We evaluate three state-of-the-art memory tiering solutions as follows. We also evaluate `No Balance` (between NUMA nodes) representing the static page placement without migration. *Our study is featured with analyzing the interplay between page migration and interleaving.*

- `AutoNUMA` [13] is the default NUMA-balancing policy in Linux. `AutoNUMA` counts memory accesses by manipulating an access bit in PTE. When a page is accessed, a NUMA hint fault is triggered. By tracking these faults, `AutoNUMA` determines the origin of memory accesses and relocates pages to minimize their distance from the computing processes. `AutoNUMA` is enabled by setting `numa_balancing` to 1 in `/proc/sys/kernel`.
- `Tiering-0.8` [73] is a recent Linux patch which uses the similar hint fault-based profiling as `AutoNUMA`. Unlike AutoNUMA, it considers the recency of page accesses based on the re-fault interval to identify hot pages. Additionally, Tiering-0.8 dynamically adjusts the page promotion criteria to throttle migration traffic and save memory bandwidth. Tiering-0.8 is enabled by setting `numa_balancing` to 2.
- `TPP` [44] uses hint faults to determine page migration. Upon encountering hint faults, TPP decides to promote a page based on its presence on the LRU list in Linux.

A. Page Migration with Uniform interleaving

Evaluation setup. To study the impact of page migration, we evaluate four memory-intensive applications, including BTree [2], an in-memory index lookup; PageRank [7] and Graph500 [45], both graph processing applications; and Silo [71], an in-memory database engine. We run them with 64 threads. We configure the memory consumption of each application to be around 130GB. This configuration ensures a fair comparison between static page placement solutions (i.e., the NUMA first touch and interleaving): LDRAM (i.e., 50GB) is set to less than half of each application's memory consumption, hence both solutions can fully utilize LDRAM.

Metrics. We collect the execution time and page migration statistics, including the number of hint faults and migrated pages. Such statistics is collected by periodically reading the Linux counters from `/proc/vmstat`. Those counters capture page accesses from the entire system, but primarily influenced by page migrations because of the application.

Evaluation results. <u>Page migration observation (PMO) 1.</u> Different applications perform differently with different page migration and static page placement solutions. No single

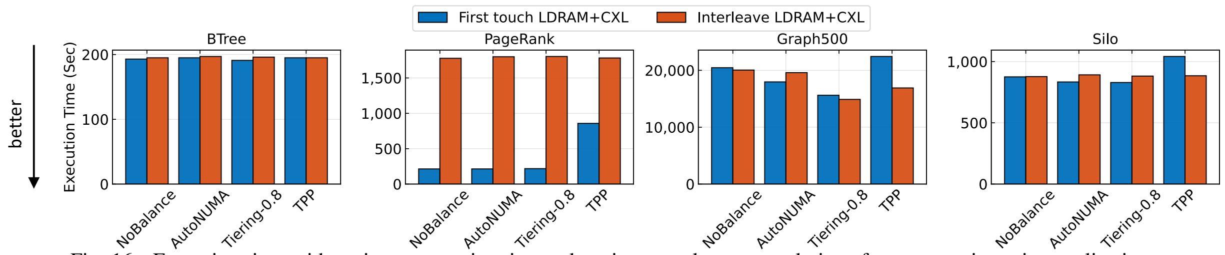

Fig. 16. Execution time with various page migration and static page placement solutions for memory-intensive applications.

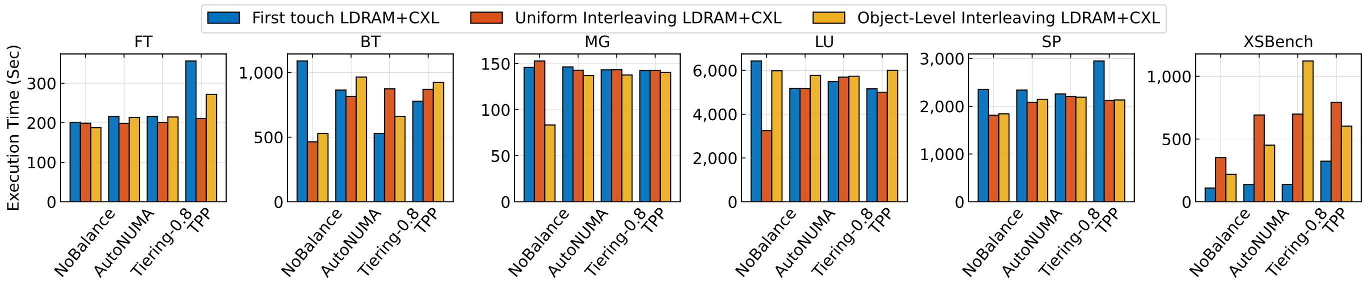

Fig. 17. Execution time with various page management solutions for HPC applications.

TABLE IV
COMPARISON OF PERFORMANCE STUDIES ON GENUINE CXL

Paper	HPC Study	LLM Study		Memory Tiering		Performance Optimization
		Training	Inference	Solutions Evaluated	Identify Limitations	
MICRO23 [67]	No	N/A	N/A	TPP	N/A	BW-aware page allocation
EuroSys24 [68]	No	N/A	CPU-based	AutoNUMA	N/A	N/A
SupMario [41]	Yes	N/A	CPU-based	TPP	Yes	Predictive interleaving & regulated migration
This work	**Yes**	**GPU-based**	**GPU-based**	**TPP, AutoNUMA, Tiering-0.8**	**Yes**	**Object-level interleaving**

solution can get the best performance for all applications. The effectiveness of a solution to an application depends on the distribution of hot pages in the working set (e.g., scattered or concentrated), and variance and size of the hot page set.

Figure 16 shows execution time. We observe that BTree is not sensitive to any solution (the performance variance is less than 3%), because of its irregular memory access patterns. PageRank achieves the best performance with the first touch without page migration, which is 88% better than any page migration solution plus interleaving, because PageRank has a small and stable set of hot pages. Graph500 achieves the best performance with Tiering-0.8 plus interleaving, outperforming other solutions by up to 33%, because Graph500's hot pages are scattered across memory tiers, and interleaving is helpful for improving data locality. Silo achieves the best performance with Tiering-0.8 plus the first touch, outperforming other solutions by up to 20%. Silo implements a B-tree-like data structure gathering hot data into fewer pages, making the first touch more effective than interleaving.

PMO 2. When using first touch, Tiering-0.8 outperforms TPP and AutoNUMA, because of its smaller profiling overhead and dynamic adjustment of the page promotion threshold.

Figure 16 illustrates that using the first touch, Tiering-0.8 outperforms `NO Balance`, AutoNUMA, and TPP by 7%, 3%, and 31%. We quantify page hints faults caused by different memory tiering strategies and observe that Tiering-0.8 has 59 $\times$ fewer hint faults. Memory profiling in TPP incurs a large overhead, and the reduced number of hint faults in tiering-0.8 contributes to performance improvement. Compared to AutoNUMA, Tiering-0.8 has a comparable number of hint faults but exhibits large variance in the number of migrated pages. For example, in PageRank, Tiering-0.8 results in 7.6 million more migrations. This variance is due to adaptiveness of page promotion threshold in Tiering-0.8.

PMO 3. The page migration based on hint faults is not integrated well with the uniform interleaving.

We observe that using page migration plus the application level interleaving, the number of hint faults is 72,721 $\times$ less than that of page migration plus the first touch on average. Our investigation reveals that when the application-level interleaving is employed, the application's pages are placed in unmigratable regions, preventing the pages to trigger hint faults. Hence, page migration cannot be effective.

B. *Page Migration with Object-Level Interleaving*

Evaluation setup. To enable fair comparison between the uniform and object-level interleaving, the capacity of LDRAM is set to 40GB for FT, 100GB for MG, and 50GB for others, such that all static page placement solutions can fully utilize LDRAM for each application. The CXL memory does not have a capacity constraint, because it is the slowest memory tier. We use 32 threads on the socket 1.

Evaluation results. PMO 4. Using the page migration plus the object-level interleaving results in minor performance improvement on HPC workloads, compared to the object-level interleaving without page migration.

Figure 17 shows that *without page migration*, the object-level interleaving outperforms the first-touch and uniform interleaving by up to 45%. However, page migration negatively impacts effectiveness of the object-level interleaving: when using AutoNUMA, Tiering-0.8, and TPP, the performance degrades by 46%, 88%, and 63% on average, compared to using the object-level interleaving without page migration. This occurs because the object-level interleaving utilizes the additional CXL bandwidth for bandwidth-hungry objects. However, the page migration undermines this benefit.

PMO 5. The page migration can improve performance for some applications but lose performance for others.

Figure 17 shows that different HPC workloads exhibit different preferences for page migration. For example, the page migration degrades performance for FT, SP and XSBench, and yields almost no performance difference in MG, regardless of using the interleaving or first touch, because these workloads have uniformly accessed working set or highly skewed and scattered hot memory region, which make hotness detection challenging. In contrast, the page migration improves BT and LU performance by up to 51% and 20%, respectively, since hot pages in BT and LU have good locality to be detected.

Takeaway: (1) There is significant potential to improve the performance of page migration in the memory tiering solutions. (2) Dynamic page migration and static page interleaving are not well-integrated. (3) Dynamic page migration may degrade performance; a better static page placement strategy *without page migration*, such as object-level interleaving, can lead to better performance.

VII. Related Work

CXL. In 2019, Intel introduced Compute Express Link (CXL [61], [72]), an open industry-standard interconnect between processors and devices such as accelerators, memory buffers, and smart network interfaces. Since its introduction, the CXL has garnered significant attention and investment from both researchers and industry practitioners [4], [11], [18], [23], [33]–[36], [44], [60], [62], [64], [65], [67], [74], [85], [88], [92]. For example, Google has explored the potential of CXL memory in memory tiering and swapping in its cloud computing infrastructure [34], while Microsoft has developed CXL-based memory pools for public cloud platforms [8], [35], and Meta has designed the CXL-based tiered memory system in hyperscale datacenters [44].

Emulation-based CXL study. Most of the explorations of using the CXL memory from both industry and academics leverage NUMA servers to emulate CXL memory performance [18], [23], [79], [88], [91]. For example, Yang et al. [88] overcome the memory wall with emulated CXL-enabled SSDs, Jang et al. [23] apply emulated CXL memory to the billion-scale approximate nearest neighbor search using a software-hardware co-design solution.

Performance study on genuine CXL. Recent studies analyzed the performance of actual CXL hardware [41], [67], [68]. Sun et al. [67] represent the first comprehensive effort to analyze CXL memory performance using genuine CXL-ready systems and devices. Nonetheless, a wide range of applications from some fields (such as HPC, AI inference and training) are still not extensively studied. Tang et al. [68] investigate the CXL performance in various datacenter scenarios and propose an Abstract Cost Model to estimate the cost-benefit of using CXL memory. However, their work does not address the limitations of existing memory tiering solutions. Liu et al. [41] analyze the root causes of CXL latency's impact on system performance. They further propose a performance model designed to predict CXL-induced slowdowns, guide the uniform interleaving ratio, and regulate page migration throttling in memory tiering. Nonetheless, their performance evaluation is restricted to CPU-centric scenarios. Our work differs in multiple perspectives, as shown in Table IV: (1) workload diversity, covering both LLMs and HPC; (2) practicality of the evaluation platform, which utilizes GPUs instead of CPUs for LLM; (3) deep insights into memory tiering; and (4) performance optimization techniques, such as object-level interleaving (Sec. VI-B) and thread assignments based on bandwidth scaling (Sec. III).

Tiered memory systems. Different memory components [1], [9], [10], [16], [19], [61], [72], [77], [83] can show different memory latency, bandwidth, capacity, and monetary cost. Using multiple memory components, tiered memory systems can save cost and improve memory capacity [3], [12], [28], [31], [56], [58], [87]. Using the CXL memory, it is natural to build a tiered memory where local DDR is the fast tier and the CXL memory is a slow tier. Many solutions [17], [20], [21], [24], [25], [29], [37]–[40], [42], [43], [50]–[53], [55], [57], [75], [76], [78], [80]–[82], [84], [86], [89], [90] have been proposed to explore and leverage the tiered memory systems to improve application performance. However, the performance of those solutions on CXL-ready systems (especially their interplay with page interleaving) is not clear [26], [59], [64].

VIII. Conclusions

In this paper, we characterize the performance of the real CXL memory and explore its use cases. We explore the use of CXL for LLM training and inference, study the performance of the CXL memory with a spectrum of HPC applications, and investigate how application-transparent solutions (page interleaving and memory tiering) perform with the real CXL. We also create a data object-level interleaving method that switches between the interleaving and NUMA node-preferred polices at the data object level, and demonstrate its superior performance. We hope that our study can shed some lights on how the future HPC systems can leverage CXL memory expansion.

Acknowledgment

This work was partially supported by U.S. National Science Foundation (2104116, 2316202 and 2348350) and the Chameleon Cloud. We thank the anonymous reviewers, as well as our shepherd, for their feedback on the paper.

References

[1] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu, "Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 971–985.

[2] R. Achermann and A. Panwar., "Mitosis workload BTree." 2019, https://github.com/mitosis-project/mitosis-workload-btree.

[3] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:8753499

[4] G. Alonso, A. Klimovic, T. Kuchler, and M. Wawrzoniak, "Rethinking serverless computing: from the programming model to the platform design," 2023.

[5] Andi Kleen (SUSE Labs), "NUMA Support for Linux," https://github.com/numactl/numactl.

[6] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," University of California at Berkeley, Tech. Rep. UCB/EECS-2006-18, 2006.

[7] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[8] D. S. Berger, D. Ernst, H. Li, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, L. Hsu, I. Agarwal, M. D. Hill *et al.*, "Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms," *IEEE Micro*, vol. 43, no. 2, pp. 30–38, 2023.

[9] S. Bergman, P. Faldu, B. Grot, L. Vilanova, and M. Silberstein, "Reconsidering os memory optimizations in the presence of disaggregated memory," in *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management*, 2022, pp. 1–14.

[10] G. Chen, B. Wu, D. Li, and X. Shen, "PORPLE: An Extensible Optimizer for Portable Data Placement on GPU," in *IEEE/ACM International Symposium on Microarchitecture*, 2014.

[11] A. Cho, A. Saxena, M. Qureshi, and A. Daglis, "A Case for CXL-Centric Server Processors," *arXiv preprint arXiv:2305.05033*, 2023.

[12] J. Choi, S. Blagodurov, and H.-W. Tseng, "Dancing in the dark: Profiling for tiered memory," *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 13–22, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:232134240

[13] J. Corbet, "AutoNUMA: the Other Approach to NUMA Scheduling," http://lwn.net/Articles/488709.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[15] N. A. S. N. Division, "NAS Parallel Benchmarks." [Online]. Available: https://www.nas.nasa.gov/software/npb.html

[16] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "{FaRM}: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[17] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. R. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," *Proceedings of the Eleventh European Conference on Computer Systems*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:8681081

[18] 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.

[19] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 649–667.

[20] M. Hildebrand, J. A. Khan, S. N. Trika, J. Lowe-Power, and V. Akella, "Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming," *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:212641763

[21] Y. Huang and D. Li, "Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.

[22] Intel Corporation, "Intel memory latency checker v3.5," 2019. [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[23] J. Jang, H. Choi, H. Bae, S. Lee, M. Kwon, and M. Jung, "CXL-ANNS: Software-Hardware collaborative memory disaggregation and computation for Billion-Scale approximate nearest neighbor search," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 585–600. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/jang

[24] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "Heteroos — os design for heterogeneous memory management in datacenter," *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 521–534, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:19189083

[25] J. Kim, W. Choe, and J. Ahn, "Exploring the design space of page management for multi-tiered memory systems," in *USENIX Annual Technical Conference*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:236992513

[26] K. D. Kim, H. Kim, J. So, W. Lee, J.-H. Im, S.-R. Y.-C. Park, J. Cho, and H. U. Song, "SMT: Software-Defined Memory Tiering for Heterogeneous Computing Systems With CXL Memory Expander," *IEEE Micro*, vol. 43, pp. 20–29, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:256491961

[27] S. Kim, C. Hooper, A. Gholami, Z. Dong, X. Li, S. Shen, M. W. Mahoney, and K. Keutzer, "Squeezellm: Dense-and-sparse quantization," *arXiv preprint arXiv:2306.07629*, 2023.

[28] V. R. Kommareddy, C. Hughes, S. D. Hammond, and A. Awad, "Deact: Architecture-aware virtual memory support for fabric attached memory systems," *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[29] S. Kumar, A. Prasad, S. R. Sarangi, and S. Subramoney, "Radiant: efficient page table management for tiered memory systems," *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235463147

[30] A. Labs, "Breaking Through the Memory Wall," https://www.asteralabs.com/general/breaking-through-the-memory-wall/.

[31] T. Lee and Y. I. Eom, "Optimizing the page hotness measurement with re-fault latency for tiered memory systems," *2022 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pp. 275–279, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247618508

[32] T. Lee, S. K. Monga, C. Min, and Y. I. Eom, "MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[33] A. Lerner and G. Alonso, "CXL and the Return of Scale-Up Database Engines," *arXiv preprint arXiv:2401.01150*, 2024.

[34] P. Levis, K. Lin, and A. Tai, "A Case Against CXL Memory Pooling," in *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, 2023, pp. 18–24.

[35] 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.

[36] Y. Li and S. Yao, "Understanding and Optimizing Serverless Workloads in CXL-Enabled Tiered Memory," *arXiv preprint arXiv:2309.01736*, 2023.

[37] Z. Li and M. Wu, "Transparent and lightweight object placement for managed workloads atop hybrid memories," *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247108266

[38] J. Liu, D. Li, and J. Li, "Athena: High-Performance Sparse Tensor Contraction Sequence on Heterogeneous Memory," in *International Conference on Supercomputing (ICS)*, 2021.

[39] J. Liu, J. Ren, R. Gioiosa, D. Li, and J. Li, "Sparta: High-Performance, Element-Wise Sparse Tensor Contraction on Heterogeneous Memory," in *Principles and Practice of Parallel Programming*, 2021.

[40] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-Memory for Energy-Efficient Neural Network Training: A Heteroge-

neous Approach," in *IEEE/ACM International Symposium on Microarchitecture*, 2018.

[41] J. Liu, H. Hadian, H. Xu, D. S. Berger, and H. Li, "Dissecting cxl memory performance at scale: Analysis, modeling, and optimization," *arXiv preprint arXiv:2409.14317*, 2024.

[42] J. Luo, L. Guo, J. Ren, K. Wu, and D. Li, "Enabling Faster NGS Analysis on Optane-based Heterogeneous Memory," in *Proceedings of Supercomputing '20 (Posters)*, Nov. 2020.

[43] A. Maruf, A. Ghosh, J. Bhimani, D. Campello, A. Rudoff, and R. Rangaswami, "Multi-clock: Dynamic tiering for hybrid memory systems," *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 925–937, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:248865268

[44] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[45] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, no. 45-74, p. 22, 2010.

[46] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 118–132.

[47] V. Petrucci, E. Mirakhur, N. Agarwal, S. W. Lim, V. Tanna, R. Gupta, and M. Wagh, "CXL Memory Expansion: A Closer Look on Actual Platform," https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-memory-expansion-a-close-look-on-actual-platform.pdf, 2024.

[48] J. Prout, "Expanding Beyond Limits With CXL-based Memory," https://memverge.com/wp-content/uploads/2022/08/CXL-Forum_Samsung.pdf.

[49] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[50] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM," *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:239029009

[51] J. Ren, J. Luo, I. Peng, K. Wu, and D. Li, "Optimizing Large-Scale Plasma Simulations on Persistent Memory-based Heterogeneous Memory with Effective Data Placement Across Memory Hierarchy," in *International Conference on Supercomputing (ICS)*, 2021.

[52] J. Ren, J. Luo, K. Wu, M. Zhang, and H. Jeon, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 598–611, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:231620477

[53] J. Ren, B. Ma, S. Yang, B. Francis, E. K. Ardestani, M. Si, and D. Li, "Machine Learning-Guided Memory Optimization for Industry-Scale Recommendation Model Inference on Tiered Memory," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2025.

[54] 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 *USENIX Annual Technical Conference*, 2021.

[55] J. Ren, D. Xu, J. Ryu, K. Shin, D. Kim, and D. Li, "MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory Systems," in *European Conference on Computer Systems*, 2024.

[56] J. Ren, D. Xu, S. Yang, J. Zhao, Z. Li, C. Navasca, C. Wang, H. Xu, and D. Li, "Enabling large dynamic neural network training with learning-based memory management," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2024.

[57] J. Ren, M. Zhang, and D. Li, "HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory," in *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[58] J. H. Ryoo, L. K. John, and A. Basu, "A case for granularity aware page migration," *Proceedings of the 2018 International Conference on Supercomputing*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52277753

[59] S. Ryu, S. Kim, J. Jun, D. Moon, K. Lee, J. Choi, S. Kim, H. Kim, L. Kim, W. H. Choi, M. Nam, D. Hwang, H. Roh, and Y.-P. Joo, "System Optimization of Data Analytics Platforms using Compute Express Link (CXL) Memory," *2023 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pp. 9–12, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257644666

[60] Y. Shan, W. Lin, Z. Guo, and Y. Zhang, "Towards a fully disaggregated and programmable data center," in *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2022, pp. 18–28.

[61] D. D. Sharma, "Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing," in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2022, pp. 5–12.

[62] D. D. Sharma, R. Blankenship, and D. S. Berger, "An Introduction to the Compute Express Link (CXL) Interconnect," 2023.

[63] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU," in *Proceedings of the 40th International Conference on Machine Learning*, ser. ICML'23. JMLR.org, 2023.

[64] J. Sim, S. Ahn, T. Ahn, S. Lee, M. Rhee, J. Kim, K. Shin, D. Moon, E. Kim, and K. Park, "Computational CXL-Memory Solution for Accelerating Memory-Intensive Applications," *IEEE Computer Architecture Letters*, vol. 22, pp. 5–8, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:254301130

[65] K. Song, J. Yang, S. Liu, and G. Pekhimenko, "Lightweight Frequency-Based Tiering for CXL Memory Systems," *arXiv preprint arXiv:2312.04789*, 2023.

[66] W. A. Spivey, "Linear programming: It is one of several mathematical approaches to problems of optimal choice under constraint." *Science*, vol. 135, no. 3497, pp. 23–27, 1962.

[67] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim, "Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices," in *IEEE/ACM International Symposium on Microarchitecture*, 2023.

[68] Y. Tang, P. Zhou, W. Zhang, H. Hu, Q. Yang, H. Xiang, T. Liu, J. Shan, R. Huang, C. Zhao, C. Chen, H. Zhang, F. Liu, S. Zhang, X. Ding, and J. Chen, "Exploring Performance and Cost Optimization with ASIC-Based CXL Memory," in *Proceedings of the European Conference on Computer Systems*, 2024.

[69] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[70] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench-the Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[71] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 18–32.

[72] S. Van Doren, "Hoti 2019: Compute express link," in *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2019, pp. 18–18.

[73] V. Verma., "Tiering-0.8." 2022, https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/log/?h=tiering-0.8.

[74] J. Wahlgren, M. Gokhale, and I. B. Peng, "Evaluating Emerging CXL-enabled Memory Pooling for HPC Systems," in *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2022, pp. 11–20.

[75] J. Wahlgren, G. Schieffer, M. Gokhale, and I. Peng, "A quantitative approach for adopting disaggregated memory in hpc systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–14.

[76] C. Wang, H. Cui, T. Cao, J. N. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu, "Panthera: holistic memory management for big data processing over hybrid memories," *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:150372592

[77] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 496–508.

[78] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos, "TMO: transparent memory offloading in datacenters," *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247026540

[79] J. Wu, J. Liu, G. Kestor, R. Gioiosa, D. Li, and A. Marquez, "Performance Study of CXL Memory Topology," in *Proceedings of the International Symposium on Memory Systems*, 2024.

[80] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[81] K. Wu, J. Ren, and D. Li, "Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Memory for Task Parallel Programs," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018.

[82] P. Wu, D. Li, Z. Chen, J. Vetter, and S. Mittal, "Algorithm-directed data placement in explicitly managed non-volatile memory," in *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2016.

[83] L. Xiang, X. Zhao, J. Rao, S. Jiang, and H. Jiang, "Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at its On-DIMM Buffering," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 488–505.

[84] Z. Xie, J. Liu, J. Li, and D. Li, "Merchandiser: Data Placement on Heterogeneous Memory for Task-Parallel HPC Applications with Load-Balance Awareness," in *Proceedings of the Symposium on Principles and Practices of Parallel Programming (PPoPP)*, 2023.

[85] 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 *36th ACM/IEEE International Conference for High Performance Computing, Performance Measurement, Modeling and Tools*, 2024.

[86] D. Xu, J. Ryu, J. Baek, K. Shin, P. Su, and D. Li, "FlexMem: Adaptive Page Profiling and Migration for Tiered Memory," in *30th USENIX Annual Technical Conference (ATC)*, 2024.

[87] Z. Yan, D. Lustig, D. W. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:102348046

[88] S.-P. Yang, M. Kim, S. Nam, J. Park, J. yong Choi, E. H. Nam, E. Lee, S. Lee, and B. S. Kim, "Overcoming the memory wall with CXL-Enabled SSDs," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 601–617. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/yang-shao-peng

[89] S. Yang, M. Zhang, W. Dong, and D. Li, "Betty: Enabling Large-Scale GNN Training with Batch-Level Graph Partitioning," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[90] S. Yang, M. Zhang, and D. Li, "Buffalo: Enabling Large-Scale GNN Training via Memory-Efficient Bucketization," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2025.

[91] Y. Yang, P. Safayenikoo, J. Ma, T. A. Khan, and A. Quinn, "CXLMemSim: A pure software simulated CXL. mem for performance characterization," *arXiv preprint arXiv:2303.06153*, 2023.

[92] M. Zhang, T. Ma, J. Hua, Z. Liu, K. Chen, N. Ding, F. Du, J. Jiang, T. Ma, and Y. Wu, "Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 658–674.

[93] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.