Accelerating Performance of GPU-based Workloads Using CXL

Moiz Arif
Rochester Institute of Technology
Rochester, NY, USA
ma3890@cs.rit.edu

Avinash Maurya
Rochester Institute of Technology
Rochester, NY, USA
am6429@cs.rit.edu

M. Mustafa Rafique
Rochester Institute of Technology
Rochester, NY, USA
mrafique@cs.rit.edu

ABSTRACT

High-performance computing (HPC) workloads such as scientific simulations and deep learning (DL) running across multi-GPU systems are memory and data-intensive, relying on the main memory to complement its limited onboard high-bandwidth memory (HBM). To facilitate faster data transfer across the slow device-to-host PCIe interconnects, these workloads typically pin memory on the host system, thereby creating a memory capacity limitation on the host memory for workloads running on peer GPUs of the same node. Compute express link (CXL) is an emerging technology that transparently extends the available system memory capacity at low latency and high throughput in a cache-coherent fashion. While this can be leveraged by workloads running across multi-GPU nodes to allocate and pin more memory, using conventional memory allocation schemes can adversely impact the data throughput due to contention on the CXL memory. To this end, we highlight the challenges related to conventional job scheduling and memory allocation on such CXL-enabled multi-GPU systems and propose an algorithm to mitigate the contention on the CXL memory, maximize throughput and reduce the overall data transfer time. Our preliminary evaluation of our proposed memory allocation approach based on simulations of a variety of job profiles and system configurations demonstrates up to 65% lower data transfer overheads as compared to the existing memory allocation approaches.

CCS CONCEPTS

• **Hardware → Emerging architectures**; **Emerging interfaces**; • **Software and its engineering → Allocation / deallocation strategies**; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*

KEYWORDS

Memory allocation; multi-GPU systems; tiered memory; Compute Express Link (CXL); pinned memory

ACM Reference Format:
Moiz Arif, Avinash Maurya, and M. Mustafa Rafique. 2023. Accelerating Performance of GPU-based Workloads Using CXL. In *Proceedings of the 13th Workshop on AI and Scientific Computing at Scale using Flexible Computing Infrastructures (FlexScience'23), June 20, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3589013.3596678

1 INTRODUCTION

The exponential amount of data to be processed by HPC systems across various disciplines of industry, research, and academics compels HPC datacenters to leverage emerging hardware technologies such as GPUs, DPUs, and FPGAs, to accelerate processing. Unsurprisingly, running HPC workloads on such accelerators shifts them from being compute-bound to becoming memory and data-bound. Such accelerator-based HPC workloads are typically characterized by their memory, data, and data-intensive nature and often involve large-scale simulations, complex computations, and massive data processing, generating and manipulating huge amounts of intermediate and output data. These workloads require fast and efficient memory and data operations to read and write data from/to memory and storage systems, as well as to exchange data between devices or servers in a distributed computing environment. Therefore, efficient data management, storage, and data operations are the key factors to achieve high performance and scalability of HPC workloads.

GPU-based HPC workloads face limitations in terms of memory capacity and memory bandwidth due to the limited onboard HBM [15]. Particularly, workloads such as adjoint computations (e.g., Reverse Time Migration and Quantum Optimal Control), DL, and other scientific simulations involve iterative read/writes of terabytes of data for storing intermediate results, supporting in-situ analytics, and collaborative processing. Given the limited HBM capacity, the GPU is forced to perform frequent data read/writes to the main memory. Furthermore, to facilitate faster transfer across the GPU HBM and host memory using direct memory access (DMA) and enable compute-transfer overlap, the GPU pins memory on the main memory. On multi-GPU setups, however, such pinning limits the amount of main memory available to other GPUs in the system, which leads to significant data transfer overheads and even job failures if the required amount of memory cannot be pinned before launching the computations.

CXL [10] is an emerging high-speed interconnect memory technology that provides an effective means to solve the challenges of limited memory capacity and bandwidth, and memory stranding. It is gaining rapid traction in both HPC and cloud computing because of its features such as byte-addressability, cache coherency, low latency, and unified access. Based on the encouraging results demonstrated by the previous and next-generation CXL protocols and prototypes [2, 13], the HPC community is actively investigating its applicability and support on multi-GPU setups for higher memory capacity and throughput. To this end, the CXL memory can mitigate, if not eliminate the challenge of limited available memory to pin multiple GPUs memory as required by the workloads.

While the available system memory for a multi-GPU system is extended by the CXL memory, allowing the underlying OS and/or GPU drivers to pin memory independent of other workloads may lead to suboptimal memory allocations. Such inefficient memory mappings can adversely impact the data movement of an application, leading to lower memory throughput and bandwidth utilization, and increased application execution times. In this paper, we explore the design of a CXL-enabled multi-GPU system and highlight the challenges related to pinning memory on such CXL extensions. *To the best of our knowledge, we are the first to study the impact of CXL memory and contention for workload on multi-GPU systems.* Specifically, we make the following contributions:

(1) Propose a reference architecture for enabling CXL memory extension in Nvidia's DGX-A100 system (§ 3).
(2) Highlight the performance bottlenecks of default memory allocation on CXL-enabled systems when running multiple jobs on a single multi-GPU system (§ 3.2).
(3) Propose a schedule-aware memory allocation approach that incorporates the memory requirement on each socket of a multi-GPU system and provides an efficient memory placement map to mitigate memory contention (§ 3.4).
(4) Evaluate our approach using diverse job profiles and system configurations. Our simulations show up to 65% lower data transfer times using our approach as compared to the default memory allocation approaches (§ 4).

2 RELATED WORK

Memory Management Approaches for Tiered Memory: Efforts, such as TPP [11], propose a transparent page placement mechanism for CXL-enabled memory to move pages across memory tiers based on the hotness/coldness of pages. Similarly, Radient [8] proposes a page table management technique that applies efficient page placement policies and dynamically manages the pages between main memory and NVMe. HotBox [4], which is a disaggregated memory management subsystem, maximizes the local memory hit rate with low memory management overhead. However, none of the tiered memory management approaches consider the case of pinned memory allocated on CXL-enabled multi-GPU setups.
Memory Disaggregation with CXL: Recently, CXL has been utilized to implement disaggregated memory systems [6, 7, 9] that enable accessing terabytes of memory over the CXL interface with low overhead. Several research efforts [1, 2] have utilized CXL-based memory to improve the performance of workloads by leveraging it as a memory expansion device and source of additional memory bandwidth. Similarly, CMS [13] improves the performance of memory-intensive applications by exploiting CXL interconnect to expand the memory capacity and uses a near-data processing approach to maximize the internal bandwidth. Nonetheless, the bandwidth bottleneck of CXL memory connected over PCIe lanes has not been studied yet.

3 CXL-ENABLED MULTI-GPU SYSTEM DESIGN

In this section, we discuss the high-level system architecture, constraints, and assumptions used in this paper. We propose an algorithm that leverages heuristics from the job scheduler, system configuration, and statistics to generate efficient memory placement maps for main and CXL memory on multi-GPU systems.

3.1 System Architecture

In this paper, we envision a reference CXL-enabled multi-GPU system architecture as shown in Figure 1. We extend this architecture

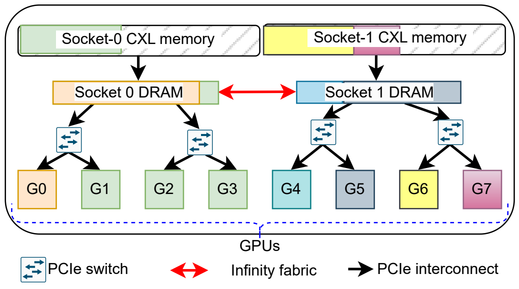


Figure 1: CXL-enabled multi-GPU system architecture.

from the Nvidia DGX-A100 system, which consists of 8 GPUs distributed evenly across the two sockets. Using PCIe switches, a pair of GPUs share the available PCIe bandwidth to connect with the main and CXL memory. All PCIe links are composed of ×16 lanes each. GPUs are interconnected to each other using a hybrid mesh-cube topology using NVLinks and NVSwitches (which we omit from this figure for simplicity). Next, we mount a CXL memory on each socket using a dedicated PCIe link (×16 lanes) to expand the capacity of the main memory. Although the processors in the DGX-A100 system can support up to 128 PCIe lanes, we map a limited number of lanes to each CXL device since most commercially available CXL expansion cards are based on PCIe ×8 configuration. Similarly, multiple CXL cards can be attached to the system PCIe interface. Therefore, when all GPUs are actively reading/writing data to/from the CXL memory of a single socket, the bandwidth of the CXL memory gets evenly distributed across all the 8 GPUs, thereby creating contention on the PCIe interconnect of the CXL memory. Implementing our design involves a combination of user and kernel-level code to ensure efficiency and maximize performance.

3.2 Bandwidth Contention on CXL Memory

Figure 1 shows each job with different color codes mapped to respective GPUs. We consider each of the GPUs mapped to 6 different jobs, such that the job to GPU mapping looks as follows: $J1 : \langle G0 \rangle; J2 : \langle G1, G2, G3 \rangle; J3 : \langle G4 \rangle; J4 : \langle G5 \rangle; J5 : \langle G6 \rangle; J6 : \langle G7 \rangle$. Since the CXL device is used for memory expansion, the operating system considers it a logical extension of the local memory and extends the physical memory address space to append the CXL memory addresses at the end of the main memory address. With such a design, the initial memory allocations are made from the main memory, and after completely exhausting the main memory, further allocations are mapped to the CXL memory. We exemplify this in Figure 1, where we observe that the jobs which started sooner ($J1 : \langle G0 \rangle$: color-coded orange) allocate and pins the required memory from the main memory (consumed 90%), compelling the later jobs ($J2 : \langle G1, G2, G3 \rangle$) to allocate memory from the CXL memory. In such scenarios, although we see an added memory capacity with the CXL device, the data read/write throughput for the job $J2$ running on $G1$, $G2$, and $G3$ gets negatively impacted by the contention on the ×8 lane PCIe interface of the CXL device. Assuming that all GPUs ($G0 - G3$) need to transfer data in parallel to the host, such memory allocation would allow $G0$ to utilize all the DRAM bandwidth, while $\langle G1, G2, G3 \rangle$ compete to access their respective data on $socket - 0$ CXL memory.

3.3 Design Goals and System Constraints

Our goal is to design an efficient schedule for memory allocation across the main and CXL memory tiers on a CXL-enabled multi-GPU setup, such that the overall time spent in data movement between GPU and host memory is minimized. The performance and efficiency of production HPC jobs are impacted by several constraints, including memory, compute resources, time, data, and software. Addressing these constraints requires both hardware and software-level optimizations, such as adjustments to the scheduler and other system components to ensure job constraints are satisfied. The design of our system is subject to the following constraints:

- **Jobs will be scheduled based on the availability of compute resources [5]**: The scheduled jobs are sent for execution as compute resources (GPUs and/or CPUs) become available regardless of the memory availability and contention on the memory and CXL interfaces.
- **Uncoordinated data movement [14]**: All running jobs transfer data with the assumption that the entire transfer bandwidth is available without performing any contention control. Similarly, the Linux kernel and the memory controller will utilize basic contention control mechanisms, e.g., fair-share and first-come-first-serve, to ensure fairness.
- **CXL device for memory expansion [2]**: The CXL memory provides additional byte-addressable memory and bandwidth to cache-coherently support the main memory.

Several HPC systems support specifying job requirements to ensure proper execution and optimal performance. Additionally, since HPC clusters are typically heterogeneous in nature [3], it is crucial to meet job requirements to ensure deterministic performance. To incorporate these requirements of real-world applications, we make the following assumptions in this paper:

- The job queue always contains a batch of jobs such that they can consume all available GPUs on the system.
- The memory required by each job is pinned during application initialization to achieve higher transfer rates using DMA and cannot be resized until the job completes.
- All jobs specify unique memory requirements and the memory footprint remains constant throughout the execution [16].
- All jobs are memory bound and each job performs continuous reads/writes to/from the data residing on the main memory and CXL memory tiers.
- The CXL device has a large enough capacity to support the memory requirements of active and scheduled jobs.

3.4 Schedule Aware Data Allocation Approach

We propose a memory placement approach that leverages the tiered memory and the optimal memory source to maximize the data transfer rate and reduces the total execution time. Our proposed approach is shown in Algorithm 1. We consider a series of batch jobs J enqueued on the scheduler ready for execution. The job configuration enlists the number of GPUs required and the total memory footprint which is either known in advance or can be estimated using predictors [12]. Additionally, the system-level statistics, such as the amount of available memory per tier and data movement bandwidth, are provided to the scheduler using resource monitoring tools, micro-benchmarks, and node specifications.

Algorithm 1: Our proposed memory allocation approach.

```
Input  : $N$: # sockets per node, $J$: list of jobs containing tuples
         $\langle j\_id, total\_mem, n\_gpus \rangle$, $G$: list of vacant GPUs IDs,
         $D$: List of DRAM memory available per socket, $C$: List of
         CXL memory available per socket, $BW_D$: DRAM
         bandwidth per socket, $BW_P$: PCIe bandwidth, $BW_C$: CXL
         bandwidth
Output : $S$: Amount of main and CXL memory to be allocated
begin
    $S \leftarrow [j \in J$ if $j[n\_gpus] < avail\_gpus]$
    for $j \in S$ do
        $j[gpus] \leftarrow allocate\_gpus(G, j[n\_gpus])$
        $j[mpg] \leftarrow j[total\_mem]/j[n\_gpus]$  // req_mem/GPU
    for $j \in S$ do
        for $g \in j[gpus]$ do
            $spill \leftarrow calc\_spill(j, N, socket(g), D, C)$
            $cxl\_pull \leftarrow calc\_cxl(spill, C, BW_D, BW_P, BW_C)$
            $on\_cxl \leftarrow min(cxl\_pull, j[mpg])$
            $on\_dram \leftarrow min(j[mpg] - on\_cxl, D[socket(g)])$
            $j[`dram`][socket(g)] += on\_dram$
            $j[`cxl`][socket(g)] += on\_cxl$
            $D[socket(g)] -= on\_dram$
            $C[socket(g)] -= on\_cxl$
    return $S$
```

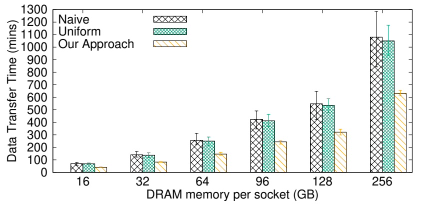
Our proposed algorithm, listed in Algorithm 1 works as follows: select a list of jobs S for execution on the available GPU resources (Lines 2-5). Next, the scheduler determines the excess amount of memory required by each GPU, referred to as *spill* based on the scheduled jobs and available memory on the CXL and DRAM cache tiers (Line 8). The `calc_spill` function computes the fraction of DRAM memory requested by the GPU g of job j which exceeds the DRAM capacity when all the scheduled jobs on $socket(g)$ are allocated fair proportions of the DRAM memory. Based on the *spill*, CXL memory available on that socket, and bandwidth of DRAM, PCIe, and CXL, respectively, the routine `calc_cxl` computes the amount of memory that can be efficiently allocated on the CXL device, such that none of the jobs scheduled on the peer-GPUs face DRAM starvation (Line 9-10). Once the efficient memory allocations are computed, they are mapped to the job j, and deducted from the available DRAM (D) and CXL (C) memory for the next set of jobs (Line 12-15). Finally, the algorithm outputs an efficient multi-tier memory allocation plan for the scheduled S jobs.

4 PRELIMINARY EVALUATIONS

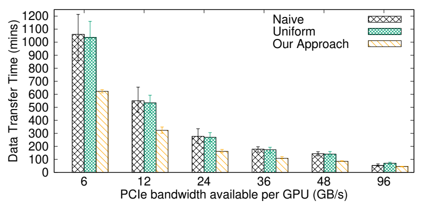
In this section, we describe our evaluation methodology, performance metrics, and performance results of our proposed approach.

4.1 Evaluation Methodology

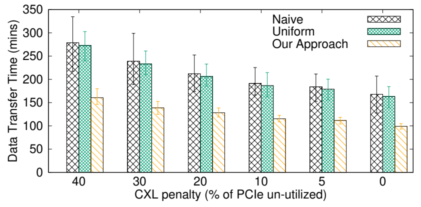
4.1.1 Simulation and Traces. We evaluate our proposed schedule-aware CXL memory placement approach for different hardware and workload profiles. To mimic the workload scheduling similar to HPC data centers, we develop a simulation model, written in Python with about 400 lines of code, which enables us to evaluate our proposed approach with various configurations. In addition to running our proposed memory allocation approach, the simulation



(a) Increasing memory capacity



(b) Variable PCIe bandwidth



(c) Varying degrees of CXL penalty

Figure 2: Data transfer time for varying memory capacity, PCIe bandwidth per GPU, and CXL penalties.

also generates synthetic job traces to run across 100 CXL-enabled multi-GPU nodes. The simulation is performed on Ubuntu 22.04 LTS server operating system with two 2.40 GHz Intel Xeon Gold 6240R processors, with 192 GB main memory. Each job consumes 1 to 8 GPUs and runs either on a single socket or both sockets depending on the requested GPUs which are allocated evenly across sockets.

4.1.2 Simulation Testbed. We simulate a series of different testbed profiles using the aforementioned simulation. We vary the profiles of the testbed starting from the default configuration of the Nvidia DGX-A100 machine, with the exception of considering 64 GB memory available per socket instead of the default 512 GB. Multiple GPUs are connected to the host system using PCIe Gen 4.0 as per the topology shown in Figure 1. The observed idle memory access latency is approximately 130 *ns*.

4.1.3 Compared Approaches. We compare three approaches for pinned memory allocation on CXL-enabled multi-GPU devices:

- **Naive**: This is the default approach adopted for memory allocation where the system starts allocating memory from the main memory followed by the CXL memory tier. In this approach, jobs that get scheduled first end up consuming all the available main memory, forcing the later jobs to allocate memory from the CXL device.
- **Uniform**: In this approach, the scheduler uniformly distributes the available main memory across all GPUs. This approach ensures equal main memory allocation to all jobs.
- **Our Approach**: This approach is detailed in § 3.4.

4.2 Performance Results

We evaluate the performance of various compared approaches by measuring the total amount of time taken by the job to perform data transfer across the main and CXL memory allocations. In our evaluations, GPUs access data concurrently to the host memory tiers, as observed in GPU-bound HPC and DL applications. We measure the data transfer time for an increasing amount of main memory available per socket, varying PCIe bandwidth available, and varying degrees of CXL penalty.

4.2.1 Increasing Available Main Memory per Socket. Our first set of experiments evaluates the data transfer times for an increasing main memory capacity. As observed in Figure 2a, our approach yields faster data transfer times with increasing capacity. This is because with increased main memory capacity our approach can perform better memory placement and load distribution across both main and CXL memory. For varying job profiles, our approach demonstrates a reduction in data transfer overheads from 15.4% to 61.2% as compared to the naive memory allocation approach.

4.2.2 Varying PCIe Bandwidth. Our next set of experiments measures the data transfer overheads of varying amounts of PCIe bandwidth available for both the GPUs and the CXL memory. This experiment studies the impact of various PCIe generations (starting from PCIe 3.0). As shown in Figure 2b, our approach performs 65.35% and 21.3% better on average as compared to the naive and uniform allocation-based allocation approaches, respectively. We note that the bandwidth reported on the x-axis is the actual share of PCIe bandwidth available to each GPU when two GPUs share a single PCIe bus using the PCIe switch. In real-world testbeds, we achieve only ~75% of the theoretical transfer throughput from the GPU to the host memory. We use this to estimate the PCIe bandwidth of the next-generation PCIe protocols.

4.2.3 Varying Degrees of CXL Penalty. As specified in the CXL 3.0 specification, the CXL protocol is currently capable of achieving only 60%-90% of actual PCIe bandwidth, which we refer to as the CXL penalty. Therefore, in our last set of experiments, we evaluate the data transfer times for the compared memory allocation approaches for different degrees of CXL penalties. As observed in Figure 2c, our approach demonstrates 17.7% to 67% lower data transfer overheads as compared to the naive and uniform memory allocation policies.

Our evaluations show that while CXL has promising benefits in terms of main memory expansion, increased data transfer throughput, and low latency, the limited PCIe bandwidth connecting these CXL devices can become a bottleneck when the memory allocations on multi-GPU systems are done using the default schedulers.

5 CONCLUSION

GPU-based HPC workloads are data intensive and process large amounts of data during execution. The performance of such workloads is often limited by the amount of onboard system memory and the contention at shared memory resources, e.g., main memory, and interconnects, e.g., PCIe. The CXL memory provides additional memory capacity to workloads, however, the default memory allocation approach is suboptimal for GPU-bound HPC workloads. In this paper, we propose an efficient memory allocation approach that leverages job schedules and additional memory tiers to mitigate contention at the CXL memory tier and maximize the performance of HPC workloads. Our preliminary evaluations show up to 65% lower data transfer overheads as compared to the default memory allocation approach. In the future, we plan to improve our memory allocation to include dynamic memory resizing and intelligent data movement between various memory tiers.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (NSF) under Awards No. 2106634/2106635.

REFERENCES

[1] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN'22)*. Association for Computing Machinery, New York, NY, USA, Article 8, 5 pages. https://doi.org/10.1145/3533737.3535090

[2] Moiz Arif, Kevin Assogba, M. Mustafa Rafique, and Sudharshan Vazhkudai. 2023. Exploiting CXL-Based Memory for Distributed Deep Learning. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 19, 11 pages. https://doi.org/10.1145/3545008.3545054

[3] Moiz Arif, M. Mustafa Rafique, Seung-Hwan Lim, and Zaki Malik. 2020. Infrastructure-Aware TensorFlow for Heterogeneous Datacenters. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–8. https://doi.org/10.1109/MASCOTS50786.2020.9285969

[4] Shai Bergman, Priyank Faldu, Boris Grot, Lluís Vilanova, and Mark Silberstein. 2022. Reconsidering OS Memory Optimizations in the Presence of Disaggregated Memory. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management* (San Diego, CA, USA) *(ISMM 2022)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3520263.3534650

[5] Yuping Fan, Zhiling Lan, Paul Rich, William E. Allcock, Michael E. Papka, Brian Austin, and David Paul. 2019. Scheduling Beyond CPUs for HPC. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/3307681.3325401

[6] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory Pooling With CXL. *IEEE Micro* 43, 2 (2023), 48–57. https://doi.org/10.1109/MM.2023.3237491

[7] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 287–294. https://www.usenix.org/conference/atc22/presentation/gouk

[8] Sandeep Kumar, Aravinda Prasad, Smruti R. Sarangi, and Sreenivas Subramoney. 2021. Radiant: Efficient Page Table Management for Tiered Memory Systems. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management* (Virtual, Canada) *(ISMM 2021)*. Association for Computing Machinery, New York, NY, USA, 66–79. https://doi.org/10.1145/3459898.3463907

[9] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. https://doi.org/10.1145/3575693.3578835

[10] Compute Express Link. 2023. The Breakthrough CPU-to-Device Interconnect CXL. Retrieved May 6, 2023 from https://www.computeexpresslink.org/

[11] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. https://doi.org/10.1145/3582016.3582063

[12] Md Nahid Newaz and Md Atiqul Mollah. 2023. Memory Usage Prediction of HPC Workloads Using Feature Engineering and Machine Learning. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region* (Singapore, Singapore) *(HPC Asia '23)*. Association for Computing Machinery, New York, NY, USA, 64–74. https://doi.org/10.1145/3578178.3578241

[13] Joonseop Sim, Soohong Ahn, Taeyoung Ahn, Seungyong Lee, Myunghyun Rhee, Jooyoung Kim, Kwangsik Shin, Donguk Moon, Euiseok Kim, and Kyoung Park. 2023. Computational CXL-Memory Solution for Accelerating Memory-Intensive Applications. *IEEE Computer Architecture Letters* 22, 1 (2023), 5–8. https://doi.org/10.1109/LCA.2022.3226482

[14] Abel Souza, Mohamad Rezaei, Erwin Laure, and Johan Tordsson. 2019. Hybrid Resource Management for HPC and Data Intensive Workloads. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 399–409. https://doi.org/10.1109/CCGRID.2019.00054

[15] Maohua Zhu, Youwei Zhuo, Chao Wang, Wenguang Chen, and Yuan Xie. 2018. Performance Evaluation and Optimization of HBM-Enabled GPU for Data-Intensive Applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 5 (2018), 831–840. https://doi.org/10.1109/TVLSI.2018.2791442

[16] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Hyunsung Shin, Jongpil Son, Sally A. Mckee, Paul M. Carpenter, Petar Radojković, and Eduard Ayguadé. 2017. Main Memory in HPC: Do We Need More or Could We Live with Less? *ACM Trans. Archit. Code Optim.* 14, 1, Article 3 (mar 2017), 26 pages. https://doi.org/10.1145/3023362