Demystifying a CXL Type-2 Device: A Heterogeneous Cooperative Computing Perspective

Houxiang Ji*, Srikar Vanavasam*, Yang Zhou*, Qirong Xia*, Jinghan Huang*, Yifan Yuan†§, Ren Wang†, Pekon Gupta‡, Bhushan Chitlur‡, Ipoom Jeong¶, Nam Sung Kim*
*University of Illinois Urbana-Champaign, †Intel Labs, ‡Intel Altera, ¶Yonsei University

***Abstract*—CXL is the latest interconnect technology built on PCIe, providing three protocols to facilitate three distinct types of devices, each with unique capabilities. Among these devices, a CXL Type-2 device has become commercially available, followed by CXL Type-3 devices. Therefore, it is timely to understand capabilities and characteristics of the CXL Type-2 device, as well as explore suitable applications. In this work, first, we delve into three key features of a CXL Type-2 device: cache-coherent device accelerator to host memory, device accelerator to device memory, and host CPU to device memory accesses. Second, using microbenchmarks, we comprehensively characterize the latency and bandwidth of these memory accesses with a CXL Type-2 device, and then compare them with those of equivalent memory accesses with comparable devices, such as emulated CXL Type-2, CXL Type-3, and PCIe devices. Lastly, as applications that exploit the unique capabilities of a CXL Type-2 device, we propose two CXL-based Linux memory optimization features: compressed RAM cache for swap (`zswap`) and memory deduplication (`ksm`). Our evaluation shows that `Redis`, when running with traditional CPU-based `zswap` and `ksm`, suffers from a tail latency increase of 4.5–10.3× compared to `Redis` running alone. While PCIe-based `zswap` and `ksm` still experience a tail latency increase of up to 8.1×, CXL-based `zswap` and `ksm` practically eliminate the tail latency increase with faster and more efficient host-device communication than PCIe-based `zswap` and `ksm`.**

***Index Terms*—compute express link, heterogeneous computing**

I. Introduction

Peripheral Component Interconnect Express (PCIe) has served as the industry standard interface for connecting I/O devices such as NICs and SSDs to the CPU. Meanwhile, popular emerging data-driven applications, such as AI and ML, must process vast amounts of data, often surpassing the processing capabilities of CPUs. Such a trend has led to the growing development and adoption of domain-specific accelerators (ACCs), such as GPUs, to bridge the performance gap [3], [4], [9], [10], [17], [20], [21], [33], [34], [43], [44], [50], [58]. Albeit powerful for specific applications, such ACCs in devices still need to work with the CPU serving as a host, *i.e.*, Cooperative Heterogeneous Computing (CHC), for end-to-end execution of the applications. Such CHC demands a high-bandwidth interface for fast host-device data transfers. This has also made PCIe the most popular interface for ACCs due to its extensive hardware/software ecosystem and continuous evolution, which has doubled its bandwidth every few years.

PCIe has worked well for the contemporary throughput-oriented coarse-grained CHC applications, such as AI and ML, with occasional host-device transfers of large amounts of data (*e.g.*, a few MB to GB per transfer [61], [68]). However, PCIe also presents a few challenges for future ACCs designed for fine-grained CHC applications that demand frequent host-device transfers of small amounts of data, such as latency-sensitive datacenter applications [12], [38], [39], [67]. First, the host CPU can access the Memory-Mapped I/O (MMIO) registers and memory regions of a device ACC using its load/store instructions (`ld`/`st`) through PCIe. However, these accesses experience high latency and low bandwidth. For example, the latency and bandwidth for a 256B read access to device memory are longer than $4\mu s$ and lower than 0.3GB/s, respectively. Direct Memory Access (DMA) allows the device ACC and the host CPU to access each other's memory regions at high bandwidth through PCIe. Nevertheless, it also undergoes long latency and low bandwidth for frequent host-device transfers of small amounts of data. Second, as PCIe is not a cache-coherent interface, it requires a complex software stack and significant programming effort to manage host-device cache coherence before each transfer. As such, frequent host-device transfers of small amounts of data through PCIe are inefficient and expensive.

The industry's latest response to these challenges is Compute Express Link (CXL) [14], [15], [27], [57], an open interconnect standard built on the PCIe physical layer. CXL provides three protocols-CXL.io, CXL.cache, and CXL.mem-developed to support three types of devices. Each type of device, with a distinct composition of these protocols, offers unique capabilities. Among these devices, the CXL Type-3 device was the first to be introduced, and it has attracted considerable attention for the following three reasons. (1) It cost-effectively expands the memory capacity and bandwidth of the host CPU, providing a memory channel with 3× fewer pins than the standard DDR5 interface for the same bandwidth. (2) Since it exposes device memory to the host CPU as remote NUMA memory, the host CPU can access the device memory with `ld`/`st`. (3) It facilitates a more convenient integration of a near-memory ACC than a standard DDR5 device, with a more

This work was supported in part by a grant from PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and the Intel TSA 2030 program.

§This work was conducted while Yifan Yuan was at Intel Labs; he is currently with Meta Platforms.

flexible interface. Although it allows the host CPU to access device memory faster than a PCIe device, it shares the same two drawbacks. The device ACC cannot directly access host memory, and both the ACC and the host CPU must manage host-device cache coherence for the shared memory space through software.

Recently, a CXL Type-2 device has become commercially available. In addition to the aforementioned capability of the CXL Type-3 device, the CXL Type-2 device allows its ACC to access host memory and automatically manages host-device cache coherence through hardware. As such, for applications that can exploit these attributes, CXL Type-2 devices make programming easier and give higher performance for fine-grained CHC between the host CPU and the device ACC than CXL Type-3 and PCIe devices. While it is designed to facilitate efficient fine-grained CHC between the host CPU and device ACCs, its true capabilities and efficiencies remain to be fully understood, especially when compared to an equivalent PCIe device. In this work, we take a PCIe 5.0-based CXL Type-2 device (Intel Agilex 7 FPGA I-Series Development Kit [24]). It integrates an FPGA with ASIC-based CXL IPs. Therefore, we have the flexibility to implement any desired functions in the CXL Type-2 device. With this CXL Type-2 device, we make the following three contributions.

Contribution 1: Uncovering architecture and capabilities of a commercial CXL Type-2 device (§IV). A CXL Type-2 device can support three types of cache-coherent memory accesses: (1) device ACC to host memory (D2H), (2) device ACC to device memory (D2D), and (3) host CPU to device memory (H2D). First, we delve into D2H accesses: non-cacheable push write to host last-level cache (LLC), non-cacheable read/write, cacheable-shared read, and cacheable-owned read/write. Second, we introduce D2D accesses in two modes: host- and device-bias modes that facilitate application-specific cache coherence optimization. The host-bias mode manages host-device cache coherence through hardware, while the device-bias mode manages it through software to provide lower latency and higher bandwidth than the host-bias mode for D2D accesses. Finally, we present H2D accesses and then uncover the key differences between CXL Type-2 and -3 devices.

Contribution 2: Characterizing latency and bandwidth of the CXL Type-2 device (§V). Using microbenchmarks, we first compare the latency and bandwidth of D2H accesses by a CXL Type-2 device with those of emulated D2H accesses by a remote node in a dual-socket NUMA system[1] and PCIe devices. Second, we compare the latency and bandwidth of D2D accesses in the host-bias mode with those of D2D accesses in the device-bias mode. Lastly, we compare the latency and bandwidth of H2D accesses to a CXL Type-2 device with those of H2D accesses to CXL Type-3 and PCIe devices. From the comparisons above, we derive unique insights to assist future research using true and emulated CXL Type-2 devices.

[1]Since a CXL device is exposed as a NUMA node [15], a remote node accessing a memory region of a local node can emulate D2H accesses.

Contribution 3: Demonstrating a utility of the CXL Type-2 device's fine-grained CHC capability for datacenter applications (§VI). In Linux, compressed RAM cache for swap (`zswap`) and memory deduplication (`ksm`) are useful memory optimization features that improve the overall utilization of datacenter servers and thus the throughput of all the applications running on the servers. However, they have not been widely deployed by datacenters because they are CPU- and memory-intensive, interfere with co-running (latency-sensitive) applications, and consequently increase their tail latency. Although the latest work has proposed to offload some functions of these features to a PCIe device [32], it provides limited benefits because these features require fine-grained CHC, which PCIe devices cannot efficiently offer. To demonstrate the potential of the CXL Type-2 device for end-to-end applications, we propose offloading these memory optimization features to the CXL Type-2 device in this work. Specifically, we implement hardware to accelerate the key functions of `zswap` and `ksm` in the CXL Type-2 device and then modify the Linux kernel code of these features to use the hardware. For CXL-based `zswap`, we use CXL memory to store compressed RAM cache (`zpool`), which cannot be easily or efficiently accomplished with PCIe-based `zswap`.

II. Background

A. Peripheral Component Interconnect Express (PCIe)

PCIe has long been the industry standard for a high-speed serial interface between CPUs and I/O devices. Each lane of the latest PCIe 5.0 can deliver a transfer rate of 32GT/s (*e.g.*, ∼64GB/s with 16 lanes).

MMIO maps specific registers and memory regions of a device into memory regions of the host, also referred to as MMIO regions. Subsequently, the host can communicate with the device by having its CPU issue `ld`/`st` to the MMIO regions. However, accesses to MMIO regions through PCIe are inherently slow due to the following reasons. First, each `ld` translated to an uncacheable read access through PCIe experiences a PCIe round-trip latency ($\sim 1\mu s$ for 64B). Second, although `st` converted to an uncacheable write access incurs only a one-way PCIe trip latency, only one access can be in-flight between the host and the device due to the strict write ordering requirement of PCIe. While write-combining write accesses improves PCIe transfer efficiency by merging multiple `st`s into a single PCIe transfer of 64B, they also suffer from the ordering requirement.

DMA is another prevailing method for host-device data transfers. Since it uses dedicated hardware, it can offer far higher bandwidth than the host CPU issuing `ld`/`st` to MMIO regions one by one. However, when the transfer size is small, DMA may exhibit higher latency and lower bandwidth than MMIO due to the high cost of setting up DMA for every transfer, regardless of its size. Furthermore, similar to MMIO, the ordering requirement limits the bandwidth and efficiency of DMA. Lastly, a DMA-write transfer to a host memory region traditionally invalidates all the cache-lines corresponding to the DMAed addresses in the host memory region to prevent

TABLE I: CXL device types, required protocols, supported operations, and potential applications.

Device	Potocols	Description	Primary application
Type 1	io+cache	Coherent D2H accesses	ACCs, SNICs with coherent cache but no local memory
Type 2	io+cache+mem	Coherent D2H, D2D, and H2D accesses	ACCs with local memory and optional coherent cache
Type 3	io+mem	Faster H2D and D2D accesses	Memory expanders and ACCs with non-coherent access to device memory

the host CPU from accessing the cache-lines that become stale after the transfer. Such invalidation incurs a considerable performance penalty.

B. Compute Express Link (CXL)

Protocols. CXL is built on the physical layer of PCIe, and it defines three protocols: CXL.io, CXL.cache, and CXL.mem. CXL.io uses the protocol features of PCIe, such as transaction-layer packet (TLP) and data-link-layer packet (DLLP), to initialize the interface between the host and a device [14]. CXL.cache enables a device to perform cache-coherent D2H access to host memory. CXL.mem supports H2D and D2D accesses for the host CPU and device ACCs, respectively [56]. Since CXL.mem exposes device memory to the host CPU as memory in a remote node, the host CPU can access CXL memory with `ld` and `st` in the same way as it accesses memory in a remote node. Therefore, when the host CPU accesses CXL memory, it caches data from/to CXL memory in every level of its cache hierarchy. However, when the device ACCs in the CXL controller write-access CXL memory, the host CPU's cache hierarchy keeps stale data.

Device types. With a distinct composition of the three CXL protocols, CXL can offer CXL Type-1, -2, and -3 devices. Table I summarizes the required CXL protocols, supported operations, and potential applications of the three device types. The CXL Type-1 device employs CXL.io and CXL.cache to implement a local cache in the device. Nonetheless, it does not have device memory that can be accessed by the host CPU. Its envisioned target application is SmartNICs (SNICs). The CXL Type-2 device makes use of all three CXL protocols to implement both device cache and device memory. Thus, it is the most versatile and promising one for diverse ACCs, such as GPUs and FPGAs. The CXL Type-3 device is built with CXL.io and CXL.mem, and its typical target applications are a memory capacity/bandwidth expander for the host, which optionally integrate near-memory ACCs. Note that the CXL Type-2 device can be also used as a memory expander.

III. System and Devices

Table II summarizes the specifications of our system and two devices we use in this work. For the system, we use a dual-socket server comprising the two latest 5th-generation Intel Xeon Scalable Processors. For the devices, we take the latest Intel Agilex-7 I-Series development kit [24] and NVIDIA BlueField-3 SNIC, referred to as Agilex-7 and BF-3 hereafter, both of which are connected to the host CPU through the PCIe-5.0 $\times$16 lanes. Agilex-7 can serve as a CXL Type-2 device, a CXL Type-3 device, or a PCIe device, while BF-3

TABLE II: System and devices.

Component	Description		
OS (kernel)	Ubuntu 22.04.2 LTS (Linux kernel v6.5)		
CPU	2$\times$ Intel® Xeon 6538Y+ CPUs @2.2 GHz, 32 cores and 60 MB LLC per CPU, Hyper-Threading disabled		
Memory	Socket 0: 8$\times$ DDR5-4800 channels Socket 1: 8$\times$ DDR5-4800 channels		
Device	**Host I/F**	**Mem. technology**	**Max. bandwidth**
CXL Type-2 (Intel Agilex® 7 [24])	CXL 1.1 over PCIe 5.0	2$\times$ DDR4-2400	19.2 GB/s per channel
SNIC (NVIDIA BF-3 [51])	PCIe 5.0	DDR5-5200	41.6 GB/s per channel

can serve only as a PCIe device. Agilex-7 comprises multiple chiplets, including one primary FPGA chiplet and up to three 'R-Tile' chiplets, each incorporating one instance of the ASIC-based hardened CXL $\times$16 endpoint IP and four instances of the hardened PCIe IP. The FPGA chiplet is integrated with two DDR4-2400 DRAM controllers for device memory. It also comprises components like 'Soft R-Tile Wrapper' and 'Soft Support Logic,' which interface between the FPGA and the hardened CXL IP, facilitating the implementation of user-defined ACCs. BF-3 integrates a conventional 400Gbps RDMA NIC with Arm CPU cores and other subsystems, such as ACCs for various network functions, a DDR5-5200 DRAM controller connected to 32GB DRAM.

IV. Device Architecture

A CXL Type-2 device consists of one or more instances of the following components: ① Memory Controller (MC) for device memory, ② Device COHerence engine (DCOH), and ③ Coherent request ACC Functional Unit (CAFU), also referred to as device ACC (Fig. 1). DCOH plays a central role in handling D2H, D2D, and H2D requests in a cache-coherent

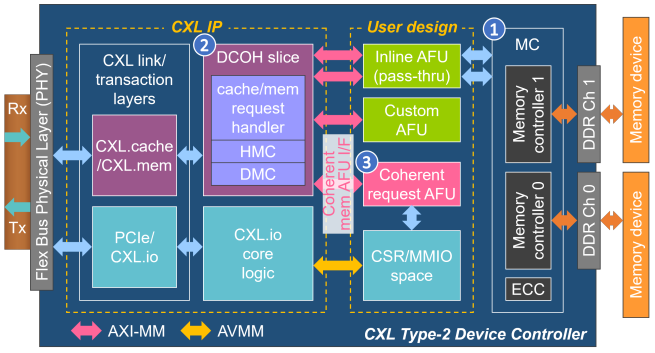


Fig. 1: CXL Type-2 device architecture.

manner through CXL.mem and CXL.cache. A device ACC is connected to a DCOH instance (or slice) through a CAFU interface[2]. It can issue D2H and D2D requests to access host memory and device memory, respectively, through DCOH. A DCOH slice comprises device cache which is divided into 'host memory cache' (HMC) and 'device memory cache' (DMC) in our CXL Type-2 device. Note that 'host memory cache' differs from 'host cache,' which collectively refers to host CPU's L1, L2, and last-level caches in this work. The CXL Type-2 device has 4-way 128 KB HMC and direct-mapped 32 KB DMC per DCOH slice. HMC and DMC store data from host memory and device memory, respectively. This division of device cache is to facilitate an asymmetric cache-coherence optimization described below.

When receiving a D2H request from a device ACC, DCOH may access HMC, host cache, or host memory to serve the memory request, depending on where it finds the latest cache-line. In contrast, when finding that an H2D request from the host CPU cannot be served by host cache, DCOH accesses only device memory to serve the memory request. That is, if DMC has a modified cache-line for the memory request, DCOH first writes back the cache-line to device memory and then accesses device memory to serve the memory request. Although DMC has a clean cache-line, DCOH still accesses device memory instead of DMC because the CXL Type-2 device is designed to allow only the device ACC to access its DMC. Meanwhile, when receiving a D2D request from the device ACC, DCOH accesses its DMC first, and then device memory in case of a DMC miss. These minimize the performance penalty of cache-coherence checks imposed on host CPU when a device ACC frequently updates its device cache and/or device memory. The remainder of this section delves into various types of D2H and D2D requests, and briefly discusses a key difference between H2D requests to a CXL Type-2 device and those to a CXL Type-3 device.

[2]A CXL Type-3 device can use only 'Inline (pass-through) AFU' and 'Custom AFU,' in Fig. 1. The former cannot issue memory requests on its own but can capture memory requests and data between the host CPU and device memory and manipulate them. The latter can issue non-cache-coherent memory requests only to device memory, in the same way as ACCs in PCIe devices do.

TABLE III: Cache coherence states after a D2H memory access.

Type	HMC cache-line			LLC cache-line		
	HMC hit	LLC hit	LLC miss	HMC hit	LLC hit	LLC miss
NC-P	Invalid			Modified		
NC-rd	No change			No change		
NC-wr	Invalid			Invalid		
CO-rd	M/E → M/E S → E	E or M‡	Exclusive	Invalid		
CO-wr	Modified			Invalid		
CS-rd	Shared			No change	I/S†	

† *Dependent on CPU implementations* ‡ *Following the original state in LLC*

A. D2H Requests

When a device ACC needs to access host memory, it issues a D2H request to DCOH with one of the desired DCOH cache states: (write-only) non-cacheable push (NC-P), non-cachable (NC), cacheable owned (CO), and (read-only) cacheable shared (CS) (Table III). The desired cache state is communicated through a cache hint of the AXI 'User Signals.' With such hints, a compiler (or programmer) strategically chooses the most appropriate type of D2H requests to optimize memory access performance for given applications.

NC-P. Compared to conventional NUMA systems, NC-P is a unique type of memory request supported by a CXL Type-2 device. When receiving NC-P from a device ACC, DCOH updates the corresponding cache-line in its HMC, writes the cache-line to a cache-line in host LLC (or simply LLC), and then invalidates the cache-line in HMC. We will present a potential improvement in performance of H2D accesses (§V-C) and a use case (§VI) in applications using NC-P.

NC. When receiving NC-read from a device ACC (Fig. 2: ❶), DCOH accesses its HMC first. If NC-read hits HMC, DCOH

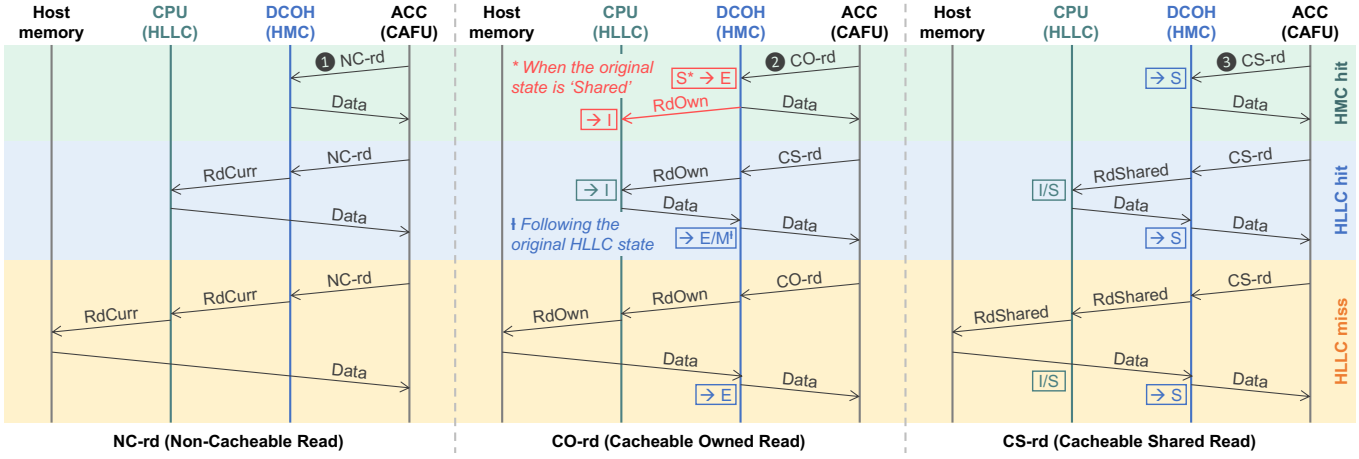


Fig. 2: Cache-coherence operations of D2H read requests from a CXL Type-2 device. RdCurr, RdOwn, and RdShared get the most current 64B word without changing the cache-coherence state, changing it to exclusive and shared states, respectively [15].

simply gets the corresponding cache-line from HMC and returns the cache-line to the device ACC without invalidating the cache-line in HMC and/or host cache. Otherwise, DCOH obtains the cache-line from LLC or host memory, but it does not cache the cache-line in HMC. For `NC-write`, DCOH checks both HMC and LLC. If DCOH locates the corresponding cache-line in HMC or LLC, it invalidates the cache-line, and then directly updates host memory, which is the key difference from `NC-P`.

CO. When DCOH receives `CO-read` (Fig. 2: ❷) or `CO-write`, it checks the state of the corresponding cache-line in its HMC. Finding that the state of the cache-line is `invalid` or `shared`, DCOH requests an exclusive ownership of the cache-line to LLC, which invalidates any corresponding cache-lines in host cache and stores an exclusive copy in HMC. `CO-write` is faster than `CO-read` because both require invalidating any corresponding cache-lines in host cache, but `CO-write` directly writes the cache-line to HMC, whereas `CO-read` needs to get the cache-line from host cache or host memory.

CS. DCOH serves `CS-read` (Fig. 2: ❸) in the same way as `NC-read`. However, when DCOH finds that the memory request cannot be served by its HMC, it gets the cache-line from LLC or host memory, and then cache the cache-line into HMC, which is the key difference from `NC-read`. Subsequently, DCOH transitions the state of the corresponding cache-line in HMC to `shared`.

B. D2D Requests

A CXL Type-2 device supports the same memory request types for both D2H and D2D accesses, but D2D requests have unique characteristics compared to D2H requests. For a given D2D request, DCOH first checks DMC and then accesses device memory if it cannot be served by DMC.

Host- and device-bias modes. To optimize the cost of managing host-device cache coherence, a CXL Type-2 device supports two modes of D2D accesses: host- and device-bias modes. In host-bias mode, DCOH always checks whether host cache has a cache-line modified by host CPU before it accesses its DMC or device memory to serve D2D memory requests. If DCOH finds that host cache has a modified cache-line and DMC also has a (stale) cache-line, it updates the cache-line in DMC and then invalidates the cache-line in host cache before serving the memory requests. The host-bias mode is useful for fine-grained CHC between the host CPU and device ACCs. In device-bias mode, also referred to as host-bypass mode, DCOH accesses its DMC and device memory without checking host cache first. This provides faster accesses to DMC and device memory for its device ACC but does not guarantee cache coherence, which demands a programmer to manage the cache coherence through software. Therefore, the device-bias mode is preferred for coarse-grained CHC between the host CPU and device ACCs. Lastly, CXL Type-2 device can define multiple device memory regions and set them to be used in different bias modes.

Dynamic switching between host- and device-bias modes. For a given device memory region, a CXL Type-2 device can dynamically switch between the two modes at runtime, but software running on host CPU must do the following preparations before switching from host bias mode to device-bias mode. Specifically, the software first determines the device memory region to which it will grant the device ACC exclusive access in device-bias mode. Next, it flushes cache-lines corresponding to the address range of the device memory region from host cache and informs the device ACC of its exclusive access the device memory region. As soon as DCOH receives an H2D request to a device memory region in device-bias mode, the memory region exits from device-bias mode and switches to host-bias mode.

Implications of bias modes to memory request types. In host-bias mode, D2D requests exhibit the same cache coherence effect (or semantics) as D2H requests. However, in device-bias mode, D2D requests do not take cache coherence into account (*i.e.*, no cache coherence state). That is, both `CO-read` and `CS-read` perform cacheable read, while `CO-write`, `NC-write`, and `NC-read` conduct cacheable write, non-cacheable write, and non-cacheable read, respectively.

C. H2D Requests

Both CXL Type-2 and -3 devices support H2D accesses, and recent work has extensively characterized the latency and bandwidth of H2D accesses to a CXL Type-3 device [60], using the same Agilex-7 device we use as a CXL Type-2 device. Yet, we reevaluate those of H2D accesses to the CXL Type-2 device, considering that it additionally features device cache compared to the CXL Type-3 device. This device cache necessitates cache-coherence checks for H2D accesses, incurring a slight performance penalty for host CPU. Consequently, H2D accesses to a CXL Type-2 device may exhibit higher latency and lower bandwidth than those to a CXL Type-3 device.

V. Device Performance Characterization

In this section, we first compare the latency and bandwidth of D2H accesses, with those of emulated D2H accesses (§V-A). Second, we compare the latency and bandwidth of D2D accesses in host-bias mode with those in device-bias mode (§V-B). Third, we compare the latency and bandwidth of H2D accesses to a CXL Type-2 device with those to a CXL Type-3 device (§V-C). Lastly, we compare the latency and bandwidth of H2D and D2H accesses with those of host-device accesses through not only DMA and RDMA over PCIe but also MMIO over PCIe (§V-D).

Microbenchmark. To measure the latency and bandwidth of H2D accesses, we take `memo` [60] and adjust it for the limited capacity of DMC. We implement a load/store unit (LSU) in a CAFU to generate N D2H and D2D requests to host memory and device memory, and record the issue time of the first memory request and the completion time of the N^{th} memory request.

Methodology. We can consider various cases where cache-lines accessed by the host CPU or a device ACC are present in device cache, device memory, host cache, and/or host memory. For host cache hits, we evaluate only the case that cache-lines are only in LLC, as cache coherence states are defined only for

LLC in the CXL standard. To guarantee that host cache has cache-lines only in LLC, we use `CLDEMOTE` [42] that pushes designated cache-lines all the way to LLC and invalidates the cache-lines in L1 and L2 caches.

For HMC and DMC hits, we first flush both caches using `CLFLUSH` and a cache-line flushing mechanism in our CXL Type-2 device, respectively. Then, we bring cache-lines to be accessed by a device ACC into those caches in `shared` state with D2H and D2D `CS-read` requests. Since D2H `CS-read` also brings cache-lines into host cache, we flush the corresponding cache-lines in host cache by allocating and initializing a separate buffer, whose size exceeds the capacity of LLC. Besides, instead of exhaustively evaluating all the possible cache coherence states in HMC, DMC, and LLC, we start to evaluate the latency and bandwidth of memory accesses after making the state of cache-lines of our interest to `shared`.

For various cache hit and miss cases, we cross-validate the presence and absence of the cache-lines in HMC, DMC, and LLC. Besides, we focus on the latency and bandwidth of only random memory accesses since we see that both sequential and random memory accesses present similar latency and bandwidth trends. For more predictable and repeatable performance, we disable hyper-threading and set the frequency of the CPU cores to 2.2GHz [2], [29]. Lastly, after repeating each experiment at least 1K times back-to-back, we take the median value.

A. D2H Accesses

To measure the latency and bandwidth of true and emulated D2H accesses, we first choose `NC-read`, `CS-read`, `NC-write`, and `CO-write` corresponding to non-temporal `ld` (`nt-ld`), `ld`, non-temporal `st` (`nt-st`), and `st`, respectively. `CO-read` and `NC-P` present latency and bandwidth similar to `CS-read` and `CO-write`, respectively. Second, we consider D2H accesses only to LLC or host memory as we focus on the latency and bandwidth of accesses to remote memory from the device's perspective. We see that those of D2H accesses to HMC is similar to those of D2D accesses to DMC in host-bias mode (§V-B). Lastly, the LSU consecutively issues 16 64B D2H requests to random addresses. We choose 16 D2H accesses because our focus is to show the advantage of the CXL Type-2 device for frequent host-device transfers of small amounts of data (§I). Fig. 3 presents the measured latency and bandwidth of true and emulated D2H accesses.

Latency. Whether hitting or missing LLC, D2H accesses present higher latency than emulated D2H accesses. Specifically, when hitting LLC ('LLC-1'), `NC-read`, `CS-read`, `NC-write`, and `CO-write` give 38%, 96%, 71%, and 56% higher latency than `nt-ld`, `ld`, `nt-st`, and `st`, respectively. When missing LLC and accessing host memory ('LLC-0'), they provide 2%, 18%, 67%, and 57% higher latency. This trend aligns with recent work [60] with the same device showing that H2D accesses through a CXL Type-3 IP also gives higher latency than emulated H2D accesses. This is because the cache-coherence mechanism of a CXL Type-2 device is more generic and/or less mature than that of the Intel UPI connecting CPUs in NUMA systems.

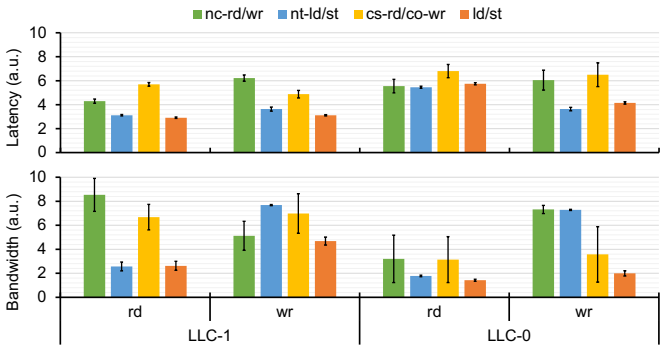


Fig. 3: Latency (top) and bandwidth (bottom) of true and emulated D2H accesses. The error bars represent the standard deviation of measured latency and bandwidth values.

Bandwidth. Whether hitting or missing LLC, D2H accesses provide higher bandwidth than emulated D2H accesses, when they exhibit latency similar to emulated D2H accesses. For instance, `CS-read` and `NC-read` for LLC-0 present 76–120% and 80–125% higher bandwidth than `nt-ld` and `ld`, respectively. This is because the CXL interconnect over PCIe 5.0 with 16 lanes (32Gbps/lane) provides 40% higher bandwidth than the UPI with 18 lanes (20Gbps/lane). However, `NC-write` for both hitting and missing LLC and `CO-write` for missing LLC present lower bandwidth than `nt-st` because it exhibits 67–79% higher latency, which affects the bandwidth for a small number of D2H accesses. For example, as we increase the number of D2H accesses over 16, we see that `CO-write` for missing LLC offers higher bandwidth than `st`.

The lower bandwidth of read accesses (`NC-read`, `nt-ld`, `CS-read`, and `ld`) than write accesses (`NC-write`, `nt-st`, `CO-write`, and `st`) is because 16 D2H write accesses (1KB) is small enough to fit into the write queues of the 8 memory controllers, each with 32 64B entries (16KB). A CPU core and the CXL Type-2 device LSU recognize that D2H write accesses are completed as soon as they enter the write queues [7]. That is, unlike D2H read accesses, D2H write accesses do not actually access (off-chip) memory which provides lower bandwidth than the (on-chip) write queues. As we increase the number of D2H write accesses to exceed the capacity of the write queues, we observe that the bandwidth of `NC-write`, `nt-st`, `CO-write`, and `st` decreases. Lastly, we use a single LSU in the CXL Type-2 device and a single CPU core in the remote node to measure the bandwidth. Note that the FPGA-based LSU in the CXL Type-2 device can issue 64B memory requests at the rate of 400MHz constrained by the maximum frequency of the FPGA. That is, it can accomplish the maximum bandwidth of 25.6GB/s. However, an ASIC-based LSU can provide higher maximum bandwidth. As we employ more and/or faster LSUs and more CPU cores, the bandwidth will approach ∼90% of the maximum bandwidth of both the CXL interconnect and the UPI, assuming that both interconnects share the same characteristics.

Insight 1: An emulated CXL Type-2 device with a remote NUMA node can present misleading performance, depending on the specific characteristics (*e.g.*, latency- and bandwidth-sensitivity) of a given fine-grained CHC application.

B. D2D Memory Accesses

To measure the latency and bandwidth of D2D accesses, we use the same memory request types as D2H accesses, *i.e.*, `NC-read`, `CS-read`, `NC-write`, and `CO-write`. Fig. 4 plots the measured latency and bandwidth of D2D accesses in both host- and device-bias modes (§ IV-B), where the lighter colors represent the latency in host-bias mode and the bandwidth in device-bias mode, respectively. While showing the bandwidth and latency of both true and emulated D2D accesses, we focus on comparing those in host-bias mode with those in device-bias mode to analyze the impact of cache-coherence checks on the bandwidth and latency in this work.

Latency. `NC-write` and `CO-write`, when hitting DMC ('DMC-1') in device-bias mode, D2D accesses in device-bias mode offer 60% lower latency than those in host-bias mode. `NC-read` and `CS-read` hitting DMC in device-bias mode do not offer notably lower latency than those in host-bias mode. This is because both `NC-read` and `CS-read` access DMC cache-lines in `shared`, eschewing the cache coherence check with LLC. In contrast, since `NC-write` and `CO-write` accompany the invalidation of shared cache-lines in LLC, they experience longer latency in host-bias mode. When missing DMC ('DMC-0'), `NC-read` and `CS-read` need to check whether LLC has any shared cache-lines before they access device memory, undergoing longer latency in host-bias mode. In general, the device-bias mode exhibits lower latency than the host-bias mode. This is because the device-bias mode does not incur the performance penalty of managing the cache coherence through hardware when the CXL Type-2 device accesses its local device memory (§IV-B).

For emulated D2D accesses hitting DMC, we assume that a CPU core hits its L1 equivalent to DMC since the CXL Type-2 device has a single-level of cache. Since the host CPU frequency is 5.5× higher than the (FPGA-based) CXL Type-2 device frequency, D2D accesses hitting DMC in host-bias mode always gives higher latency than emulated D2D accesses. If the FPGA-based CXL Type-2 device is replaced with an ASIC CXL Type-2 device, the latency of D2D memory accesses in host-bias mode is expected to be comparable to that of emulated D2D accesses.

Bandwidth. `NC-read` and `CS-read` in device-bias mode do not offer notably higher bandwidth than those in host-bias mode because of the same reason that they do not yield lower latency above. `NC-write` and `CO-write` in device-bias mode provide 8–12% and 10–13% higher bandwidth than those in host-bias mode.

Insight 2: The device-bias mode can potentially provide a higher performance for memory-intensive applications, such as near-memory processing, than the host-bias mode but at the cost of a more programming effort to manage host-device cache coherence through software.

C. H2D Accesses

To measure the latency and bandwidth of H2D accesses, we use a CPU core issuing `nt-ld`, `nt-st`, `ld`, and `nt-ld`, respectively. Fig. 5 plots the latency and bandwidth of H2D accesses to CXL Type-2 and -3 devices, perceived by the host CPU. In this section, we focus only on a key difference in H2D accesses between CXL Type-2 and -3 devices from the same Agilex-7 device, since recent work [60] has analyzed the difference in H2D accesses between true and emulated CXL Type-3 devices with the same Agilex-7 device. Since the CXL Type-3 device does not have DMC, we present the latency and bandwidth of H2D accesses to the CXL Type-3 device only for missing DMC.

When missing DMC, the CXL Type-2 device offers the latency and bandwidth comparable to the CXL Type-3 device. Specifically, `ld`, `nt-ld`, `st`, and `nt-st` to the CXL Type-2 device present 5%, 4%, 5%, and 2% higher latency and

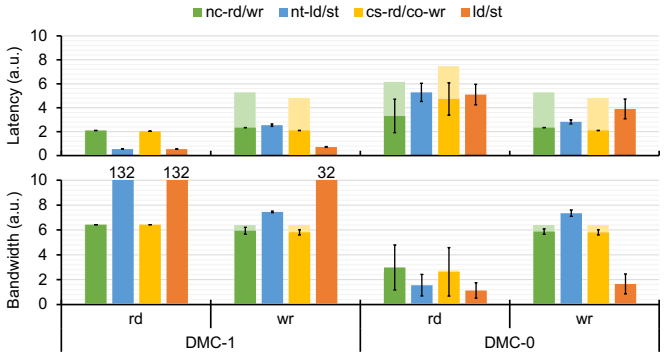


Fig. 4: Latency (top) and bandwidth (bottom) of D2D accesses. The error bars representing the standard deviation of measured values are applied to latency in device-bias mode and bandwidth in host-bias mode.

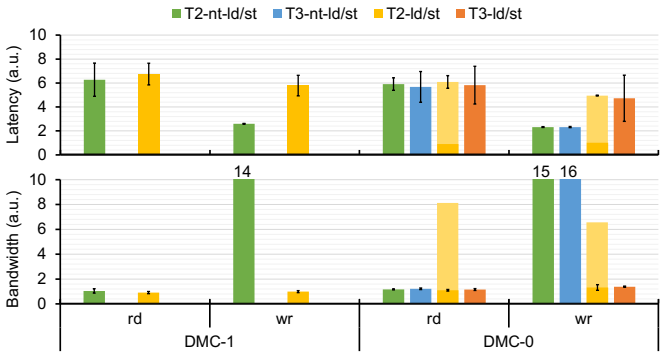


Fig. 5: Latency (top) and bandwidth (bottom) of H2D accesses. T2 and T3 denote CXL Type-2 and -3 devices, respectively. The error bars represent the standard deviation of measured latency and bandwidth values.

5%, 4%, 4%, and 4% lower bandwidth than those to a CXL Type-3 device (*i.e.*, the same device without CXL.cache), respectively. `ld`, `nt-ld`, `st`, and `nt-st` hitting DMC (with corresponding cache-lines in `owned`) exhibit 11%, 6%, 17%, and 10% higher latency and 18%, 12%, 26%, and 8% lower bandwidth than those missing DMC. This is counter-intuitive but can be explained with the following reasons.

DMC can be accessed only by its device ACC and it does not serve H2D requests from host CPU (§IV). Hence, when receiving an H2D request, DCOH always first accesses DMC to check and update the cache coherence state of the corresponding cache-line in DMC. Subsequently, DCOH accesses device memory to serve the H2D request. This makes H2D requests to the CXL Type-2 device slower than those to the CXL Type-3 device that directly accesses device memory.

To demonstrate this point, we change the coherence state of the cache-lines from `owned` to `shared` after making a device ACC issue `CS-read` to DMC, and then measure the latency of H2D accesses hitting DMC. Comparing the latency of `ld` for hitting DMC with that for missing DMC, we see a negligible difference between the two because of the following reasons. If the cache coherence state is `owned`, DCOH has to change the cache coherence state to `shared` since the cache-line will be also cached in host cache. However, if the cache coherence state is `modified`, DCOH has to not only change the cache coherence state but also write back the cache-line to device memory. For instance, `ld` and `st` hitting DMC with cache-lines in `modified` gives 36–40% higher latency than `ld` and `st` missing DMC. `nt-st` gives 12.2, 13.2, and 10.7× higher bandwidth than `nt-ld`, `ld`, and `st`, respectively, because the CPU core perceives that `nt-st` is completed as soon as it arrives at the CXL controller.

Insight 3: When a CXL Type-2 device is used as a memory expander with a near-memory processing capability, cache-lines in DMC should be in `shared` or flushed not to hurt the performance of H2D accesses from host CPU.

Lastly, we evaluate the benefit of `NC-P` (§IV-A) for temporal H2D accesses (*i.e.*, `ld` and `st`). The darker and lighter colors for DMC-0 represent the latency and bandwidth of the H2D accesses, respectively, when we use `NC-P` to push 64B words, which the host CPU is going to access, to host LLC in advance. H2D accesses to host LLC offers 82–87% lower latency and 4.1–6.7× higher bandwidth than H2D accesses missing LLC and then accessing device memory.

Insight 4: When `NC-P` is intelligently used, we can eliminate the performance penalty of H2D accesses incurred by long latency of accessing device memory through CXL.

D. Transfer Efficiency: CXL vs. PCIe

To demonstrate the efficiency of CXL over PCIe, we measure the latency and bandwidth of D2H and H2D accesses using both CXL Type-2 device and PCIe devices. For the PCIe devices, we use (1) Agilex-7, which is the same device used for CXL Type-2 device and (2) BF-3. Agilex-7 has 16 PCIe 5.0 lanes whereas BF-3 has 32 PCIe 5.0 lanes, providing up to 2× higher bandwidth. Specifically, for H2D accesses, the host CPU issues memory requests to a device memory region through MMIO over PCIe (PCIe-MMIO), DMA [30] over PCIe (PCIe-DMA), RDMA over PCIe (PCIe-RDMA), Data Center On-a-Chip (DOCA) DMA [52] over PCIe (PCIe-DOCA-DMA), `ld`/`st` over CXL (CXL-LD/ST), and DSA over CXL (CXL-DSA). DSA denotes Data Streaming Accelerator [28], which supports DMA between two host memory regions, and CXL memory is also recognized as part of host memory. For D2H accesses, the device issues memory requests to a host memory region through PCIe-MMIO supported by BF-3, PCIe-DMA, PCIe-RDMA, PCIe-DOCA-DMA, and CXL-LD/ST. We do not evaluate PCIe-DMA for D2H DMA because Agilex-7 requires a proprietary IP to perform D2H DMA. Fig. 6 plots the latency and bandwidth of these H2D (top) and D2H (bottom) accesses.

H2D access. CXL-LD/ST gives the lowest H2D-access latency for the transfer size of up to 1KB. For example, CXL-ST

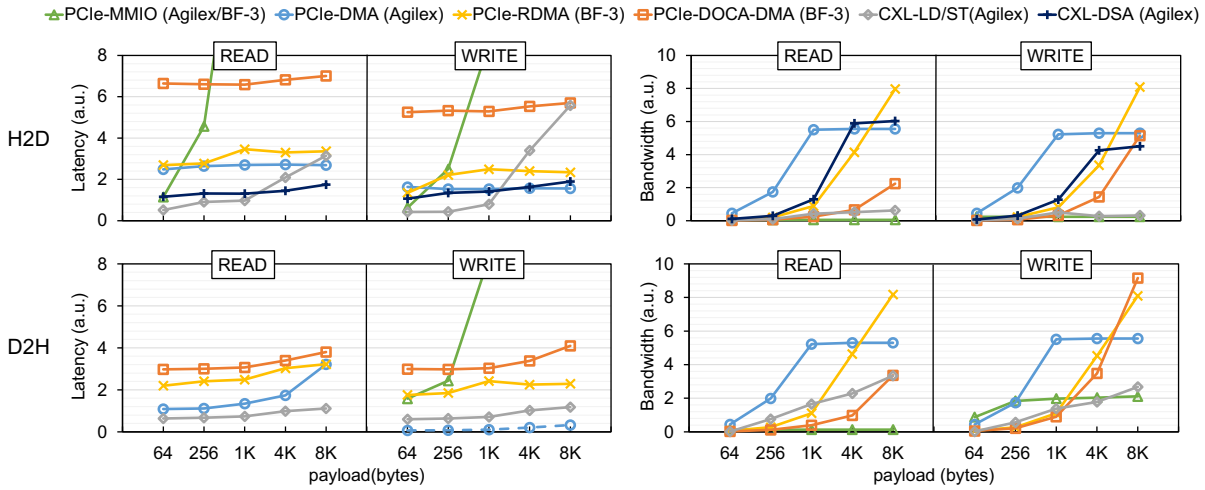


Fig. 6: Comparison of `ld`/`st` and DMA over CXL with MMIO, DMA, and RDMA over PCIe.

offers 83%, 72%, 81%, and 92% lower H2D-access latency than PCIe-MMIO, PCIe-DMA, PCIe-RDMA and PCIe-DOCA-DMA, respectively, for the transfer size of 256B. The higher H2D-access latency for the transfer size over 1KB (through PCIe-MMIO and CXL-LD/ST) is contributed by the fact that the host CPU becomes a bottleneck with the limited size of its LD/ST queues. We address it with CXL-DSA that offers the lowest H2D-access latency comparable to PCIe-DMA for the transfer size over 1KB. The H2D-access bandwidth of PCIe-DMA and CXL-DSA saturates at ∼30GB/s, while that of PCIe-RDMA offers up to 40GB/s, partly because BF3 provides 2× more PCIe 5.0 lanes than Agilex-7. Both PCIe-DOCA-DMA and PCIe-RDMA use BF-3, but PCIe-RDMA is more performant than PCIe-DOCA-DMA, providing higher latency and lower bandwidth [63].

D2H access. CXL-LD/ST (with `CS-read` and `NC-P`) gives considerably lower D2H-access latency than PCIe-MMIO, PCIe-DOCA-DMA and PCIe-RDMA. Note that we use `NC-P` for CXL-ST as DMA and RDMA directly write to host LLC with Intel DDIO instead of host memory [26]. For instance, CXL-LD gives ∼3× lower D2H-access latency than PCIe-RDMA across all the transfer sizes we evaluated. While D2H CXL-LD and H2D CXL-ST accesses are equivalent, D2H CXL-LD presents shorter latency as the D2H CXL-LD accesses go through a shorter logic/interconnect path than the H2D CXL-ST accesses. The D2H-access bandwidth of CXL-LD/ST linearly increases with the transfer size, currently limited by the ability to issue D2H requests (§V-A). Lastly, D2H PCIe-DMA presents the seemingly lowest latency among D2H write accesses because the DMA IP considers the submission of a DMA descriptor as the completion of a given transfer without considering the transfer time.

Insight 5: The CXL Type-2 device gives notably lower latency than the PCIe devices especially for a small transfer size for both H2D and D2H accesses. However, when a choice is given, D2H accesses should be used to accomplish even lower latency than H2D accesses.

VI. Device-based Kernel Feature Acceleration

We have demonstrated that a CXL Type-2 device offers fast cache-coherent D2H and H2D accesses in a unified memory space, which a traditional PCIe device cannot easily and efficiently achieve. The CXL Type-2 device with such a capability gives us unprecedented opportunities for more efficient fine-grained CHC with a broader range of applications than the PCIe device. Prior work has exploited other cache-coherent interconnect technologies to accelerate AI/ML and database applications through fine-grained CHC. Yet, it has used a device like CXL Type-1 devices (*i.e.*, no device memory), relying on a traditional NUMA system, FPGA-based prototypes, or an experimental industry device [6], [12], [38], [54], [67].

In this work, on the contrary, we exploit the capability of a commercial CXL Type-2 device to accelerate applications in a unique application domain (*i.e.*, Linux kernel features), compared to the prior work. Such kernel features have proven to better utilize resources of given systems, but they considerably contend host CPU cycles and cache space with co-running (latency-sensitive) applications when invoked. Tackling such a problem, recent work proposed to make PCIe SNIC execute CPU- and memory-intensive functions of the kernel features on behalf of the host CPU to prevent the execution of those functions from consuming host CPU cycles and polluting host cache [32]. Nonetheless, it has shown limited success and application due to long latency and inefficiency of host-device transfers of small amounts of data through PCIe. This work aims to demonstrate the superior performance of CXL-based acceleration compared to PCIe-based acceleration for such an application domain.

For a full-system demonstration, we choose `zswap` and `ksm` and then have a CXL Type-2 device execute their key CPU- and memory-intensive functions on behalf of the host CPU, *i.e.*, CXL Type-2 device-based `zswap` and `ksm`, and compare them with PCIe-based `zswap` and `ksm`, respectively. Note that, while `ksm` exploits only the D2H access capability of a CXL Type-2 device, `zswap` uses the memory capacity expansion capability of the CXL Type-2 device to store compressed pages in device memory unlike PCIe-based implementations.

A. Compressed Cache for Swap

`zswap` serves as a compression backend for the Linux swap daemon (`kswapd`) which includes synchronous direct and asynchronous background paths.

Workflow. `kswapd` takes the synchronous direct path when the memory allocator fails to allocate pages due to a lack of free memory space. This requires `kswapd` to immediately swap out the least recently used (LRU) pages to the backing swap device. `kswapd` takes the asynchronous background path when the amount of free memory space drops below the `page_low` watermark. This makes `kswapd` begin to swap out pages from the inactive page list, continuing until the amount of free memory space exceeds the `page_high` watermark.

When deployed, `zswap` intercepts the pages from both the paths above, compresses them, and places them in a dynamically allocated memory pool in DRAM (*i.e.*, `zpool`). Meanwhile, when the size of `zpool` reaches the `max_pool_percent` threshold, `zswap` wakes up and takes the LRU page from `zpool`, decompresses and relocates it to the backing swap device, and frees the compressed page from `zpool`. To serve a page fault, `zswap` first checks `zpool` to find whether the page is evicted to the backing swap device. If the page is found in `zpool`, it is simply decompressed and returned by `zswap`. Otherwise, the system follows the standard process for swapping in a page from the backing swap device.

Implementation. In CXL-based `zswap` (`cxl-zswap`), we take two data-plane functions that compress LRU pages and decompress compressed pages from `zpool`. We implement `cxl-zswap` within a device ACC (*i.e.*, CAFU in Fig. 1) operating at 400MHz. The implementation only utilizes 32% of FPGA LUTs.

When `kswapd` needs to swap out an LRU page (Fig. 7), ❶ it simply passes the starting addresses of the LRU page and

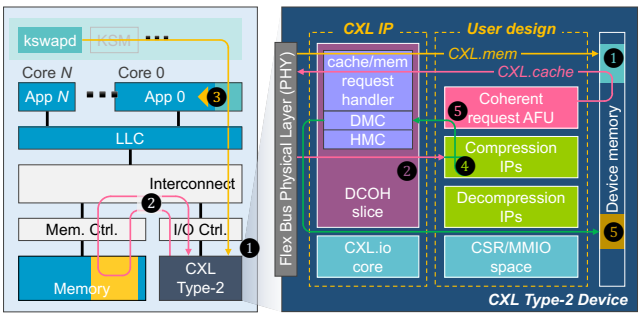


Fig. 7: Workflow of CXL-based `zswap`.

a memory region in `zpool`, as the source and destination. to the shared memory region between the host and the device in device memory, using `nt-st` accesses to eschew the pollution of host cache. The device keeps polling the shared memory region until receiving the addresses through D2D `CS-read` accesses. We choose `CS-read` over `NC-read` because it reads DMC faster than `NC-read`, especially when device memory is unchanged. ❷ `cxl-zswap` starts the D2H transfer of the page from the host to the device using `NC-read` that gives the lowest latency among all the D2H accesses for 4KB (Fig. 6). Note that `CS-read` still pollutes host cache, whereas `CO-read` does not. However, `CO-read` gives longer latency and lower bandwidth than `NC-read`. After passing the addresses, ❸ `kswapd` immediately yields the host CPU core to a co-running application process and sleeps for a conservatively determined period based on the data transfer and compression time (*i.e.*, $\sim 10\mu$s); `cxl-zswap` can be implemented with ❹ a hardware-based compression IP that offers 1.8–2.8× faster compression speed than the host CPU for a 4KB page. After ❺ the device ACC completes the compression of a given page, it issues D2D `NC-write` to store the compressed page to `zpool` allocated in device memory. Given the limited size of DMC (32KB), there is no benefit to cache compressed pages as they are likely to be evicted before decompression. Additionally, `NC-write` provides the lowest latency among D2D write accesses. We pipeline ❷, ❹, and ❺ (only the part that stores the page) because the compression IP operates in a streaming manner and CXL D2H/D2D accesses are performed at the cache-line granularity. The size of the compressed page is then returned to the shared memory region through a D2D `NC-write` access. After ❺ the device ACC completes the compression of the page and returns the size of the compressed page to the shared memory region, `kswapd` wakes up and resumes and completes the remaining control-plan operations.

When `kswapd` needs to decompress a compressed page, ❶ it passes the starting addresses of the compressed page in `zpool` and a host memory region where the decompressed page will be written, as the source and destination, to the shared memory region. Then, it yields the host CPU core as it does in compression. ❷ Then the device begins to D2D `CS-read` the compressed page from `zpool` to DMC and ❹ hardware-based IP decompresses the page. ❺ Once it completes the decompression, it starts to perform D2H `NC-P` the decompressed page to host LLC, which facilitates much faster H2D access (*cf*. Insight 4). Again, we pipeline the ❷, ❹ and ❺ as we do for the compression.

B. Memory Deduplication

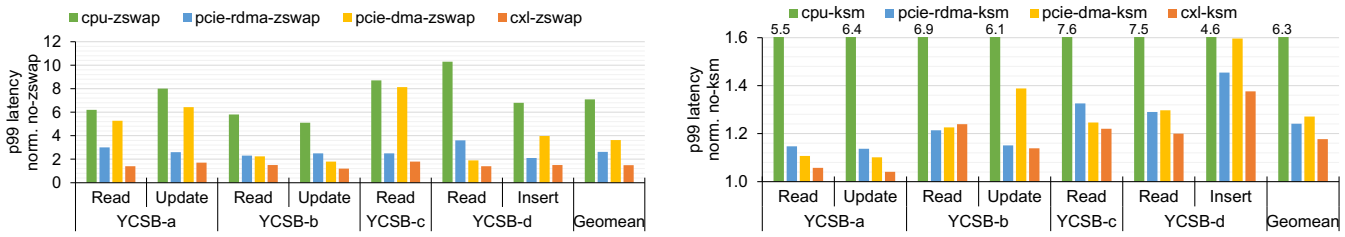
`ksm` is a Linux kernel feature that deduplicates pages with the same content among multiple Virtual Machines (VMs), *e.g.*, pages storing code for the OS and common libraries, and is used with Kernel-based VM (KVM) to deploy more VMs within a given physical memory capacity [46].

Workflow. `ksm` periodically and incrementally scans the pages of two or more running processes to identify those with the same contents. While scanning the pages, it computes a 32-bit hash value for each scanned page, serving as a hint as to whether the page has been altered since the last scan. When it finds that two pages have the same hash value, it takes the two pages and sequentially compares each byte of the two pages with the same address offset until the two bytes differ. This determines whether the two pages can be merged and their relative order in the unstable and stable trees [31]. If it determines that the two pages are identical, it merges them into a single physical copy, updates associated page table entries with a copy-on-write (CoW) attribute, and reclaims the memory space previously occupied by the pages. Nonetheless, both the hash computation and the page comparison are CPU- and memory-intensive [32].

Implementation. In CXL-based `ksm` (`cxl-ksm`), we also take two CPU- and memory-intensive data-plane functions that compute the checksum of a scanned page with `xxhash` [13] and perform the byte-by-byte comparison of two pages. We implement the `xxhash` and byte-by-byte comparison hardware in a device ACC. Due to the simplicity of these functions, the implementation only utilizes 23% of FPGA LUTs. ❶ The device ACC receives the starting addresses of two pages to be compared, following the same method as `cxl-zswap`. We pipeline ❷ the D2H data transfer with ❹ the comparison to accelerate the process, while the checksum requires the entire page transfer to be completed before starting the calculation. ❺ Upon completion of the checksum calculation or comparison, the device ACC `NC-P`s the results directly back to host LLC. Other than these, `cxl-ksm` works in the same way as `cxl-zswap`.

VII. Comparison between CXL- and PCIe-based Linux Kernel Features

Benchmark. To evaluate the impact of running `zswap` and `ksm` on the tail latency of co-running applications, we choose `Redis` [55] with `YCSB` [16]. `Redis` is a popular high-performance in-memory Key-Value Store (KVS) and `YCSB` serves as workloads for `Redis`. `YCSB` comprises four workloads: (a) `update heavy`, (b) `read heavy`, (c) `read only`, and (d) `read latest` that consist of (a) 50% read and 50% update, (b) 95% read and 5% update, (c) 100% read, and (d) 95% read and 5% insert, respectively. We use a uniform distribution for key values.



(a) zswap (b) ksm

Fig. 8: 99th-percentile (p99) latency of `Redis` with `YCSB` workloads running on `cpu-zswap/ksm`, `pcie-rdma-zswap/ksm`, `pcie-dma-zswap/ksm`, and `cxl-zswap/ksm`, normalized to `no-zswap/ksm`.

Methodology. We compare `cxl-zswap` and `cxl-ksm` with host CPU-, PCIe-RDMA, and PCIe-DMA-based `zswap` and `ksm` (`cpu-*`, `pcie-rdma-*`, and `pcie-dma-*`), respectively. After taking `pcie-rdma-*` from prior work [32], we re-implement them with BF-3. Since Agilex-7 does not provide a proper DMA library and IP for D2H DMA as BF-3 RDMA does, we emulate `pcie-dma-*` with `cxl-*` after making the host-device transfer time of CXL-LT/ST for relevant transfer sizes to closely match that of PCIe-DMA and PCIe-DOCA-DMA, as measured earlier (Fig. 6). `pcie-rdma-*` and `pcie-dma-*` perform the data-plane functions with BF-3 CPU and Agilex-7 FPGA, respectively.

We use the same evaluation methodology as prior work evaluating `pcie-rdma-*` [32]. However, since we use the latest host CPU with 2× more cores and memory channels, we turn on a sub-NUMA clustering mode [25] to use only half of the host CPU resources (*i.e.*, 16 host CPU cores and 4 memory channels) and make our system comparable with the system used by the prior work. To evaluate `*-zswap`, we run 2 `Redis` servers and 6 clients on 8 host CPU cores, and an antagonist workload, which allocates and frees memory space periodically, on the remaining 8 host CPU cores. To evaluate `*-ksm`, we set up 16 VMs (*i.e.*, 12 VMs for `Redis` clients and 4 VMs for `Redis` servers), pin each VM to a host CPU core,

Tail latency. We choose the 99th-percentile (p99) latency as a performance metric for user-serving datacenter applications, such as `Redis` [49], [53]. Fig. 8 plots the p99 latency of `Redis` running with `cpu-*`, `cxl-*` and `pcie-*`, normalized to those of `Redis` running alone (`no-*`). `cpu-zswap` and `cpu-ksm` increase the p99 latency of `Redis` with `YCSB` by 5.1–10.3× and 4.5–7.6×, compared to `no-zswap` and `no-ksm`, respectively. These values fall within the ranges demonstrated in prior work [32]. `pcie-rdma-zswap` and `pcie-rdma-ksm` decrease the p99 latency increase to 29–49% and 17–32%, respectively. `pcie-dma-zswap` and `pcie-dma-ksm` decrease the p99 latency increase to 18–93% and 16–35%, respectively. Lastly, `cxl-zswap` and `cxl-ksm` further decrease the p99 latency increase to 14–26% and 16–30%, respectively.

`cxl-*` offers lower p99 latency than `pcie-*` because it consumes considerably fewer host CPU cycles, *i.e.*, less interference with `Redis`, for H2D and D2H accesses. Specifically, for ❶ and ❺ (Fig. 7), `cxl-*` simply uses `nt-st` and `nt-ld`, whereas `pcie-*` must perform an expensive RDMA or DMA initialization before starting a data transfer to the device ACC and the host CPU, respectively. Additionally, `pcie-*` requires the host CPU to handle interrupts from the device for ❺. Nonetheless, `pcie-rdma-*` offers moderately lower p99 latency than `pcie-dma-*` on average, although PCIe-RDMA offers slightly higher latency and lower bandwidth than PCIe-DMA for the host-device transfer of data equal to or smaller than 4KB (Fig. 6). This because the software stack of PCIe-DMA we use is less efficient than that of PCIe-RDMA, which consumes more host CPU cycles and thus impacts the p99 latency of co-running `Redis` more. Subsequently, we will provide more detailed analyses of the consumption of host CPU cycles, the rate of LLC misses, and the latency of offloading functions to devices, as well as the coding complexity.

Host CPU cycle and LLC miss rate. Both `cxl-*` and `pcie-*` reduce host CPU cycle consumption and cache pollution since they have the device ACC execute CPU- and memory-intensive functions. On average, `pcie-rdma-zswap` and `pcie-rdma-ksm` reduce the host CPU cycles consumed by `zswap` and `ksm` from 25% to 16% and 21% to 7%, respectively, while `pcie-dma-zswap` and `pcie-dma-ksm` achieve reductions from 25% to 19% and 21% to 9%, respectively. `cxl-zswap` and `cxl-ksm` further reduce the host CPU cycles consumed by `zswap` and `ksm` to 11% and 5%, respectively. Lastly, we observe that `pcie-rdma-*`, `pcie-dma-*`, and `cxl-*` reduce the cache pollution (*i.e.*, LLC miss rates) to a similar degree. This is because they also work in a similar fashion for data transfers, with the only difference being whether they use DMA, RDMA, or the CXL Type-2 device's capability of directly accessing host cache and memory.

Offloading latency. Table IV shows the breakdown of the average latency of offloading the compression function of `zswap` to the PCIe-RDMA, PCIe-DMA, and CXL Type-2 devices. The compression function of `cxl-zswap` achieves 64% and 37% lower latency than that of `pcie-rdma-zswap` and `pcie-dma-zswap`, respectively, due to the following reasons. The FPGA-based compression IP used for `cxl-zswap` (and `pcie-dma-zswap`) is faster than the Arm CPU performing compression for `pcie-rdma-zswap`. Besides, unlike PCIe devices, CXL Type-2 devices transparently and unrestrictedly expose device memory to the host CPU, which allows `cxl-zswap` to allocate `zpool` to device memory and store compressed pages directly into `zpool`. Therefore,

TABLE IV: Breakdown of the offloading latency values of compression function in `zswap`. We only report the total latency of `cxl-zswap` as it pipelines three steps.

Latency (a.u.)	❷	❹	❺	Total
`pcie-rdma-zswap`	3.1	5.5	2.3	10.9
`pcie-dma-zswap`	1.7	2.9	1.6	6.2
`cxl-zswap`				3.9

`cxl-zswap` eschews an additional data transfer of compressed pages to `zpool` in host memory through PCIe.

Note that the prior work has BF-2 execute the compression and decompression functions only for the asynchronous background path in `zswap` because of the following reasons. The synchronous direct path is in the performance critical path handling page faults. However, the latency of executing the decompression function with BF-2 is higher than that with the host CPU. This is because of long latency of both page transfers over PCIe and decompression using either the slow Arm CPU or a high-throughput-but-long-latency off-path ACC in BF-2. This increases the time to handle a page fault by 31%, proportionally degrading the performance of applications causing page faults. In contrast, the CXL Type-2 device boasts 2.1× and 1.6× lower latency than BF-2 and the host CPU, respectively, for delivering a decompressed 4KB page to the host CPU.

Coding complexity. With cache-coherent H2D and D2H accesses using `ld`/`st`, a CXL Type-2 device can make fine-grained CHC notably simpler than a PCIe device. `cxl-ksm` and `cxl-zswap` require ∼20 and 50 lines of code (LoC) modification in `ksm` and `zswap`, respectively. Meanwhile, `pcie-rdma-*` [32] demands ∼1,300 LoC modification since many kernel-space RDMA data structures and verbs are used to manage RDMA connections and transfer pages between the host CPU and BF-3.

VIII. Related Work

Cache-coherent interconnect for accelerators. Recently, the cache-coherent interconnect for NUMA systems has been extended to host-device communication, *e.g.*, Intel UPI-based CPU-FPGA platform [11], [23], followed by CAPI [59], CCIX [8], Enzian [12], and Gen-Z [22]. CXL has emerged as the next-generation cache-coherent interconnect, attracting the industry to build an active ecosystem. This work takes the first step to help the community understand the design, implementation, interface, and characteristics of a CXL Type-2 device, paving the way for wider adaption and application of CXL.

Unified memory for accelerators. For decades, there have been continuous efforts to unify the memory space between the host CPU and device ACCs. Commonly, a unified memory space is explicitly achieved and managed by the device driver running on the host CPU. A notable example is Unified Virtual Memory (UVM) for GPUs [35], [37], [41], [47], [62]. Such unified memory, however, is inefficient due to high software complexity and performance costs. With a cache-coherent interconnect such as CXL, a device ACC is able to access unified memory space at the hardware level. As such, host-device communication, especially with small transfer sizes, can become cheaper and more flexible, unlocking the potential for device ACCs to offload/accelerate more fine-grained functions from the host CPU. This has been proven by a recent body of work [6], [12], [38], [54], [67], and we extend this to the kernel memory optimization features in this work.

Offloading OS functions/stacks to accelerators. Recently, there has been increasing interest in offloading expensive OS functions and stacks to various accelerators. For instance, PCIe-based SNIC has been leveraged to offload distributed file systems [36], networking stacks and functions [18], [48], [69], and memory optimization features [32]. FPGAs are also popular for accelerating kernel functions [1]. In this work, we show that CXL Type-2 device, with the unique advantages of unified memory space and fast host-device communications, has a great potential for (1) accelerating the same workloads with higher performance, (2) extending the spectrum of offloading to both memory-intensive and fine-grained μs-scale functions, and (3) simplifying the accelerator hardware/software design.

CXL Type-3 device. Since the inception, CXL has been actively discussed and invested by the industry. Meta envisioned using a CXL Type-3 device for memory tiering and swapping [45], [64]. Microsoft emulated a CXL Type-3 device with a NUMA system to explore memory disaggregation [5], [40]. There are also efforts in building software-based CXL simulators [65], [66]. For example, Gouk *et al.* built a CXL memory prototype with FPGA [19]. The latest work extensively characterized commercial CXL Type-3 devices [60].

IX. Conclusion

The CXL Type-2 device has emerged and been commercially available with appealing features like cache coherence and unified memory space between the host CPU and device accelerators. In this work, we first introduced various capabilities of a commercial CXL Type-2 device. Second, we performed an extensive characterization of the CXL Type-2 device to help the community understand the capabilities and characteristics of CXL Type-2 device, as well as differences among CXL Type-2, CXL Type-3, and PCIe devices. Lastly, we motivated the adaption of a CXL Type-2 device with a compelling use case, *i.e.*, acceleration of Linux kernel memory optimization features. They provided significant reduction in tail latency of a latency-sensitive application and consumption of host CPU cycles, compared to the host CPU- and PCIe-based kernel features.

References

[1] P. T. Abeyrathne, S. D. Dewasurendra, and D. Elkaduwa, "Offloading specific performance-related kernel functions into an FPGA," in *Proceedings of the 30th IEEE International Symposium on Industrial Electronics (ISIE'21)*, 2021.

[2] B. Acun, P. Miller, and L. V. Kale, "Variation Among Processors Under Turbo Boost in HPC Systems," in *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*, 2016.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*, 2015.

[4] E. Baek, H. Lee, Y. Kim, and J. Kim, "FlexLearn: Fast and highly efficient brain simulations using flexible on-chip learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*, 2019.

[5] D. S. Berger, D. Ernst, H. Li, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, L. Hsu, I. Agarwal, M. D. Hill, and R. Bianchini, "Design tradeoffs in CXL-based memory pools for public cloud platforms," *IEEE Micro*, 2023.

[6] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.

[7] S. Carroll and W.-M. Lin, "Latency-aware write buffer resource control in multi-threaded cores," *Int. J. Distrib. Parallel Syst.(IJDPS)*, 2016.

[8] CCIX Consortium, "CCIX: Cache Coherent Interconnect for Accelerators," https://www.ccixconsortium.com, accessed in 2021.

[9] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, 2014.

[10] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*, 2016.

[11] Y.-K. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "In-depth analysis on microarchitectures of modern heterogeneous CPU-FPGA platforms," *ACM Transactions on Reconfigurable Technology Systems*, 2019.

[12] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, R. Achermann, G. Alonso, and T. Roscoe, "Enzian: An open, general, CPU/FPGA platform for systems software research," in *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, 2022.

[13] Y. Collet, "xxHash: Extremely fast hash algorithm," https://github.com/Cyan4973/xxHash, accessed in 2023.

[14] Compute Express Link Consortium, "Compute Express Link (CXL)," https://www.computeexpresslink.org, accessed in 2023.

[15] Compute Express Link Consortium, "Compute Express Link Specification - Revision 3.0, Version 1.0," https://computeexpresslink.org/cxl-specification, accessed in 2023.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, 2010.

[17] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*, 2019.

[18] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, L. Jack, L. Norman, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, M. Silva, Ganriel nd Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, 2018.

[19] D. Gouk, M. Kwon, H. Bae, S. Lee, and M. Jung, "Memory Pooling with CXL," *IEEE Micro*, 2023.

[20] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, 2016.

[21] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "ExTensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*, 2019.

[22] S. Hong, S.-W. Sok, W.-O. Kwon, and M.-H. Oh, "Implementation and analysis of a memory-semantic interconnect based on Gen-Z protocol," in *Proceedigs of the 2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia'20)*, 2020.

[23] Intel Corporation, "Intel Xeon Gold 6138P Processor," https://ark.intel.com/content/www/us/en/ark/products/139940/intel-xeon-gold-6138p-processor-27-5m-cache-2-00-ghz.html, accessed in 2021.

[24] Intel Corporation, "Agilex™ 7 FPGA I-Series Development Kit," https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/agi027.html, accessed in 2023.

[25] Intel Corporation, "Intel Xeon Processor Scalable Family Technical Overview," https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html, accessed in 2023.

[26] Intel Corporation, "Intel® Data Direct I/O (DDIO)," https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html, accessed in 2023.

[27] Intel, Corporation, "Intel® FPGA Compute Express Link (CXL) IP," https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/cxl-ip.html, accessed in 2023.

[28] Intel Corporation, "Introducing the Intel®Data Streaming Accelerator (Intel®DSA)," https://01.org/blogs/2019/introducing-intel-data-streaming-accelerator, accessed in 2023.

[29] Intel Corporation, "Optimizing Computer Applications for Latency: Part 1: Configuring the Hardware," https://www.intel.com/content/www/us/en/developer/articles/technical/optimizing-computer-applications-for-latency-part-1-configuring-the-hardware.html, accessed in 2023.

[30] Intel Corporation, "PCIe Multi-Channel DMA IP," https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/multichannel-dma-mcdma.html, accessed in 2023.

[31] E. Izik and D. Hugh, "Kernel Samepage Merging," https://docs.kernel.org/next/admin-guide/mm/ksm.html, accessed in 2023.

[32] H. Ji, M. Mansi, Y. Sun, Y. Yuan, J. Huang, R. Kuper, M. M. Swift, and N. S. Kim, "STYX: Exploiting SmartNIC capability to reduce datacenter memory tax," in *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC'23)*, 2023.

[33] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*, 2017.

[34] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, "A hardware accelerator for protocol buffers," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21)*, 2021.

[35] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-aware unified memory management in GPUs for irregular workloads," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[36] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel, "LineFS: Efficient smartnic offload of a distributed file system with pipeline parallelism," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, 2021.

[37] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *Proceedigs of the 2014 IEEE High Performance Extreme Computing Conference (HPEC'14)*, 2014.

[38] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: Efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs," in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.

[39] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-performance in-memory key-value store with programmable NIC," in *Proceedings of the ACM SIGOPS 26th Symposium on Operating Systems Principles (SOSP'17)*, 2017.

[40] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "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 (ASPLOS'23)*, 2023.

[41] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on nvidia GPUs," in *Proceedigs of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'15)*, 2015.

[42] Linux Kernel Documentation, "CLDEMOTE: Cache Line Demote — felixcloutier.com," https://www.felixcloutier.com/x86/cldemote, accessed in 2024.

[43] A. Lottarini, J. a. P. Cerqueira, T. J. Repetti, S. A. Edwards, K. A. Ross, M. Seok, and M. A. Kim, "Master of none acceleration: A comparison of accelerator architectures for analytical query processing," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*, 2019.

[44] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "TABLA: A unified template-based framework for accelerating statistical machine learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*, 2016.

[45] 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 *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, 2023.

[46] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "XLH: More effective memory deduplication scanners through cross-layer hints," in *Proceedigs of 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, 2013.

[47] A. Mishra, L. Li, M. Kong, H. Finkel, and B. Chapman, "Benchmarking and evaluating unified memory for OpenMP GPU offloading," in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC'17)*, 2017.

[48] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "AccelTCP: Accelerating network applications with stateful TCP offloading," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, 2020.

[49] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proceedigs of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013.

[50] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," *SIGARCH Computer Architecture News*, 2017.

[51] NVIDIA Corporation, "NVIDIA BlueField-3 DPU," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, accessed in 2023.

[52] NVIDIA Corporation, "DOCA Documentation," https://docs.nvidia.com/doca/sdk/doca+dma/index.html, accessed in 2024.

[53] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads." in *Proceedigs of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, 2019.

[54] M. Owaida, D. Sidler, K. Kara, and G. Alonso, "Centaur: A framework for hybrid CPU-FPGA databases," in *Proceedings of the 25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*, 2017.

[55] Redis Labs, "Redis: The real-time data platform," https://redis.io, accessed in 2023.

[56] Robert Blankenship, "Compute Express Link (CXL): Memory and Cache Protocols," https://snia.org/sites/default/files/SDC/2020/130-Blankenship-CXL-1.1-Protocol-Extensions.pdf, accessed in 2020.

[57] D. D. Sharma, "Compute Express Link (CXL): Enabling Heterogeneous Data-Centric Computing With Heterogeneous Memory Hierarchy," *IEEE Micro*, 2022.

[58] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, 2016.

[59] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "CAPI: A coherent accelerator processor interface," *IBM Journal of Research and Development*, 2015.

[60] 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 *Proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, 2023.

[61] Y. Tatsugi and A. Nukada, "Accelerating data transfer between host and device using idle GPU," in *Proceedings of the 14th Workshop on General Purpose Processing Using GPU*, 2022.

[62] P. Wang, J. Wang, C. Li, J. Wang, H. Zhu, and M. Guo, "Grus: Toward unified-memory-efficient high-performance graph processing on GPU," *ACM Trans. Archit. Code Optim.*, 2021.

[63] X. Wei, R. Cheng, Y. Yang, R. Chen, and H. Chen, "Characterizing Off-path SmartNIC for Accelerating Distributed Systems," in *Proceedigs of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, 2023.

[64] 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," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, 2022.

[65] Xu Zhang, "Gem5-CXL," https://github.com/zxhero/gem5-CXL, accessed in 2023.

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

[67] Y. Yuan, J. Huang, Y. Sun, T. Wang, J. Nelson, D. R. Ports, Y. Wang, R. Wang, C. Tai, and N. S. Kim, "RAMBDA: RDMA-driven acceleration framework for memory-intensive μs-scale datacenter applications," in *Proceedings of the 2023 IEEE International Symposium on High Performance Computer Architecture (HPCA'23)*, 2023.

[68] S. Zhang, Z. Qin, Y. Yang, L. Shen, and Z. Wang, "Transparent partial page migration between CPU and GPU," *Frontiers of Computer Science*, 2020.

[69] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100gbps intrusion prevention on a single server," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.