Low-overhead General-purpose Near-Data Processing in CXL Memory Expanders

Hyungkyu Ham* POSTECH
Jeongmin Hong* POSTECH
Geonwoo Park POSTECH
Yunseon Shin POSTECH
Okkyun Woo POSTECH
Wonhyuk Yang POSTECH
Jinhoon Bae POSTECH
Eunhyeok Park POSTECH
Hyojin Sung Seoul National University
Euicheol Lim SK hynix
Gwangsun Kim† POSTECH

***Abstract*—To overcome the memory capacity wall of large-scale AI and big data applications, Compute Express Link (CXL) enables cost-efficient memory expansion beyond the local DRAM of processors. While its CXL.mem protocol provides minimal latency overhead through an optimized protocol stack, frequent CXL memory accesses can result in significant slowdowns for memory-bound applications whether they are latency-sensitive or bandwidth-intensive. The near-data processing (NDP) in the CXL controller promises to overcome such limitations of passive CXL memory. However, prior work on NDP in CXL memory proposes application-specific units that are not suitable for practical CXL memory-based systems that should support various applications. On the other hand, existing CPU or GPU cores are not cost-effective for NDP because they are not optimized for memory-bound applications. In addition, the communication between the host processor and CXL controller for NDP offloading should achieve low latency, but the CXL.io (or PCIe) protocol incurs μs-scale latency and is not suitable for fine-grained NDP.**

To achieve high-performance NDP end-to-end, we propose a low-overhead general-purpose NDP architecture for CXL memory referred to as *Memory-Mapped NDP (M^2NDP)*, which comprises *memory-mapped functions (M^2func)* and *memory-mapped μthreading ($M^2\mu$thr)*. The M^2func is a CXL.mem-compatible low-overhead communication mechanism between the host processor and NDP controller in the CXL memory. The $M^2\mu$thr enables low-cost, general-purpose NDP unit design by introducing lightweight *μthreads* that support highly concurrent execution of NDP kernels with minimal resource wastage. By combining them, our M^2NDP achieves significant speedups for various applications, including in-memory OLAP, key-value store, large language model, recommendation model, and graph analytics by up to 128× (11.5× overall) and reduces energy by up to 87.9% (80.1% overall) compared to a baseline CPU or GPU host with passive CXL memory.

I. Introduction

Compute Express Link (CXL) [15] is emerging as a widely-adopted interconnect standard for communication between processors, accelerators, and memory expanders in a system. An important use case of CXL is memory expansion through the *memory-semantic* CXL.mem protocol that enables low-latency remote memory access with load/store instructions. The latency of CXL.mem is known to be significantly lower than that of PCIe and comparable to cross-socket NUMA latency, providing 150-175 ns load-to-use latency [89], [114], [125]. Thus, the host's memory capacity can be cost-effectively increased beyond the limited local DRAM. Such capability can be especially beneficial for workloads with huge memory footprints, including in-memory online analytic processing (OLAP), key-value store (KVStore), large language model (LLM) [32], deep learning recommendation models (e.g., DLRM [101]), and graph analytics [4]. The CXL memory is already supported by commercial CPUs [3], [30] and several prototypes with up to 512 GB capacity [73], [106] have been announced.

However, the CXL interconnect latency can still be significant for latency-sensitive applications that frequently access data in CXL memory [89], [97], [125]. In addition, the link bandwidth (BW) can become a bottleneck for BW-intensive applications because it is substantially lower than the internal memory BW within the CXL memory [57], [118]. Thus, compared to directly accessing CXL memory data for host-side computation, using near-data processing (NDP) in CXL memory can provide significant speedups for memory-bound workloads with low arithmetic intensity [57], [66], [70].

Unfortunately, these prior approaches implement application-specific NDP HW logic in CXL memory, limiting their target workloads. Moreover, introducing a wide variety of special-purpose HW units for different NDP targets in each CXL memory may not be a practical approach due to the high total area and NRE cost [96]. While FPGAs can be adapted to target workloads [18], they have considerable programmability challenges [29]. Existing CPU or GPU cores, when used as NDP units [31], [41], [46], [78], [107], [128], [138], do not provide sufficient performance per cost based on our evaluation, because they are not optimized for memory-bound workloads.

Furthermore, conventional MMIO-based NDP offloading using CXL.io (functionally equivalent to PCIe) in prior works [57], [66], [70], [118] can incur high latency overhead from CXL.io protocol stack as well as costly user/kernel mode switching on the host, wasting CPU cycles. While CXL.mem has low latency and can be used within user space, it supports only basic memory reads/writes. Therefore, to achieve high performance for fine-grained NDP (e.g., key-value stores), an alternative low-overhead offloading mechanism is necessary.

Thus, to realize low-overhead, general-purpose NDP in

* These authors contributed equally to this work.
† Corresponding author. Email: g.kim@postech.ac.kr

CXL memory, we propose a novel *Memory-Mapped NDP (M^2NDP)* architecture. M^2NDP is based on two key components we propose: *Memory-Mapped function (M^2func)* for low-overhead communication between the host and NDP-enabled CXL memory, and *Memory-Mapped μthreading ($M^2\mu$thr)* for efficient NDP kernel execution.

The M^2func selectively repurposes read and write packets defined in CXL.mem for efficient host-device communication beyond memory transactions. By encapsulating NDP management commands (i.e., function calls) in CXL.mem requests to pre-determined addresses, we can avoid the high latency overhead of conventional offloading using CXL.io/PCIe. A key enabler for the M^2func is a *packet filter* placed at the input port of the CXL memory. It checks if an incoming request's memory address matches the pre-allocated memory range dedicated for each host process. Then, for matching requests, different NDP management functions are triggered depending on the address. Thus, NDP management function calls (e.g., kernel registration, launch, and status poll) can be done simply by issuing memory accesses from the host. As a result, M^2func minimizes the latency of NDP offloading, especially benefiting fine-grained NDP. Additionally, we do not require any modification to the CXL.mem standard for best compatibility with host CPUs. Consequently, M^2func avoids the complexity of managing a ring buffer-based shared task queue between the host and CXL/PCIe-attached devices by providing a clean function call abstraction.

Furthermore, we propose $M^2\mu$thr for the intuitive abstraction of NDP and cost-effective kernel execution. Memory-bound workloads tend to use fewer registers than compute-bound workloads. Thus, we propose a *μthread*, which is a lightweight thread with a subset of the architectural registers, as a unit of execution. By reducing register usage, the NDP unit can concurrently execute many μthreads to hide DRAM access latency without excessive physical register file cost. In addition, memory-bound data-parallel workloads are typically implemented such that each thread is associated with specific data to be processed. In conventional programming environments such as CUDA, the association between a thread and memory location is expressed *indirectly* via code (e.g., calculating the index of the array element for a thread using threadblock ID, block dimension, and thread ID in CUDA). In contrast, with our $M^2\mu$thr, we create each μthread in *direct* association with a particular memory location – i.e., the μthreads are memory-mapped. As a result, the initial address calculation code can be avoided.

Our NDP unit's architecture is based on the RISC-V ISA with vector extension [6] to leverage SIMD units and fully utilize the DRAM BW within a CXL memory cost-effectively while supporting scalar operations to avoid redundant address calculations in SIMT-only GPUs [56]. Many memory-mapped μthreads are executed with fine-grained multithreading (FGMT) to hide memory access latency. The μthreads are also spawned individually, in contrast to threadblock spawning in GPUs, which can waste resources due to inter-warp divergence. Our NDP unit's ISA is also independent of the host ISA.

By combining M^2func and $M^2\mu$thr, our proposed M^2NDP architecture enables low-overhead, general-purpose NDP in CXL memory. We demonstrate the effectiveness of our design for various workloads, including in-memory OLAP, KVStore, LLM, DLRM, and graph analytics.

To summarize, our contributions include the following:

- We propose *M^2NDP (memory-mapped NDP)* to enable **general-purpose** NDP in CXL memory. Our architecture is based on the unmodified CXL.mem protocol and, thus, does not require any host processor hardware modifications. M^2NDP consists of *M^2func (memory-mapped function)* and *$M^2\mu$thr (memory-mapped μthreading)*.
- The M^2func supports **low-overhead NDP offloading and management** from the host processor through CXL.mem, overcoming the high overhead of CXL.io for fine-grained NDP offloading while retaining standard-compatibility. As a result, it achieves speedups of up to 3.89× (34.1% overall) compared to NDP offloading with CXL.io.
- The $M^2\mu$thr enables **efficient NDP kernel execution** by lightweight FGMT using RISC-V with vector extension while reducing redundant address calculation overhead compared to SIMT-only GPUs. Its fine-grained μthread creation also avoids the waste of resources from threadblock-granularity resource allocation.
- Our evaluation results show that M^2NDP can achieve high speedups of up to 128× (11.5× overall) for various workloads, compared to the baseline system with passive CXL memory, while reducing energy consumption by up to 87.9% (80.1% overall).

II. Background and Motivation

A. Considerations in Architecting NDP in CXL Memory

The CXL memory can cost-efficiently expand the system memory for workloads with large memory footprints [21], [42], [53], [108], [117]. Although passive CXL memory can degrade the performance of latency-sensitive [125] and BW-intensive [140] workloads, NDP in CXL memory poses a substantial opportunity to effectively address this challenge. While CXL memory with NDP capability is somewhat similar to GPU in that the host can offload computation to these devices, they are introduced with very different *primary* objectives (i.e., memory expansion vs. compute acceleration), and thus, have fundamentally different requirements in terms of memory capacity, cost, and compute throughput (Table I). In particular, the CXL memory cannot employ 100s of SMs for NDP as in high-cost GPUs [36]. In contrast to GPUs, the NDP architecture specifically targets memory-bound (either BW-bound [140] or latency-bound [125]) workloads with large

TABLE I
HIGH-LEVEL COMPARISON OF GPU AND CXL MEMORY WITH NDP.

	GPU	CXL memory with NDP
Memory capacity	Low	High
Cost (area and power)	High	Low
FLOPS per memory BW	High	Low
Key target workloads	Compute-bound	Memory-bound

memory footprints which do not fit in typical on-chip caches and have low arithmetic intensity. Compute-bound workloads or those with small working sets that fit in on-chip caches can be executed more efficiently on the host or GPUs. Consequently, these devices, with their distinctive characteristics, can complement each other in a system for different workloads.

B. Compute Express Link Interconnect

The CXL [15] uses the PHY layer of PCIe and defines three protocols: CXL.io, functionally equivalent to PCIe, is used for device management; CXL.cache allows a CXL device to access host memory using a cache coherence protocol; CXL.mem enables memory expansion through CXL. In particular, CXL.mem enables processors to access CXL memory data by simply issuing load/store instructions while providing lower latency compared to CXL.io [97], [125]. The load-to-use latency for CXL memory can be as low as ∼150 ns, which includes *round-trip* latencies through the host cache, CXL protocol stack, physical off-chip wires, and DRAM [89], [114], [115]. The round-trip latency through the CXL protocol stack and physical wires alone is ∼70 ns, as shown in Fig. 1. The CXL memory access latency through a CXL switch can approach 300 ns [89].

The CXL specification also defines three device types based on the protocols supported. In all types, CXL.io should be supported for device management. Type 1 devices are accelerators without memory (e.g., smart NIC) that use CXL.cache. Type 2 devices are cache-coherent accelerators with memory (e.g., GPU and FPGA) that use CXL.cache and CXL.mem. Type 3 devices are memory expanders that support CXL.mem.

With CXL.mem, the CXL memory is managed by the host processor, referred to as Host-managed Device Memory (HDM), and can be accessed by the host using a Host Physical Address (HPA). A type-3 CXL memory expander can use either HDM-H (host-only coherent) or HDM-DB (device coherent using back-invalidation) coherence model. The HDM-H is for passive memory expanders, which are not supposed to manipulate the memory exposed to the host [15]. In contrast, HDM-DB supports a device coherence agent (DCOH) and a snoop filter in the CXL memory to track the host's caching of HDM using a metadata field of requests. Thus, it can perform back-invalidation (BI) to host cache when needed using BI channels of CXL.mem [15]. Thus, HDM-DB is suitable for CXL memory with NDP capability and we assume this model in this work. The host can also flush HDM data from its cache using HW support in the CPUs and bound the flush time with the cache size [13], [82] or (NDP) data size [68].

The CXL 3.0 also supports direct peer-to-peer (P2P) access, allowing a CXL device to directly access the HDM of another CXL device through a CXL switch [16]. This feature can be useful for scalable NDP across multiple CXL memories.

For address translation, a CXL device can use the ATS [9] defined in PCIe to request translation from the host. However, it can incur several μs of latency due to the protocol overhead and page table walks on the host [126]. To reduce the overhead, the device can have an Address Translation Cache

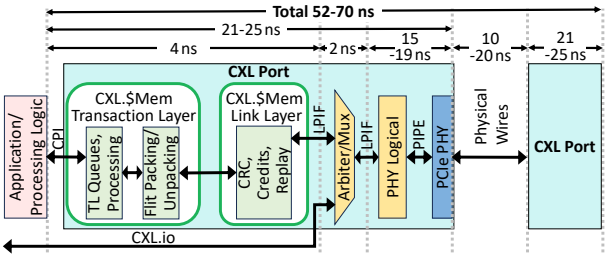


Fig. 1. CXL implementation and measured *round-trip* latencies (figure and latency numbers adapted from D. D. Sharma [114]). CXL.$Mem refers to both CXL.cache and CXL.mem.

(ATC) to keep recently used translation information. When the page table is updated, the host can invalidate the ATC on the device to prevent incorrect translations.

C. Communication Overhead with CXL.io/PCIe

Computation offloading through CXL.io/PCIe (hereafter, CXL.io) involves several SW and HW steps that can result in significant overhead in terms of latency and host processor usage, especially for fine-grained offloading. A common method used for various devices (e.g., GPU, SSD, and NIC) is based on a ring buffer shared and manipulated by both the host driver and a CXL.io device [45]. For a GPU kernel launch, the host runtime first writes the kernel launch command in the user buffer and the driver pushes a packet that points to the GPU command into the ring buffer in the kernel space. The host then updates the write (or tail) pointer for the ring buffer to notify the GPU of the new command [92], [129], which incurs additional latency through the PCIe and triggers two DMA operations from the GPU to fetch the GPU command. Overall, the complex manipulation of the ring buffer shared between the host and GPU can incur two and a half CXL.io round-trips for a kernel launch [45], resulting in high latency of ∼4.5μs [94]. To check kernel completion, polling or interrupt can be used and they consume additional host processor cycles. Polling over PCIe can incur 2-3μs [67] overhead and interrupt has similar or higher overhead [58], [60], [136]. Thus, the total latency of a kernel launch and completion check can take significantly longer than 5μs or 10,000 cycles at 2 GHz. Such latency may be acceptable for coarse-grained NDP but can be too high for latency-sensitive, fine-grained NDP kernels.

Alternatively, to avoid such overhead, a pair of device-side registers can be directly accessed through MMIO over CXL.io to send a request and check the result [43], [57], [118]. However, this approach is limited to supporting a single request at a time and cannot support multiple concurrent requests, resulting in limited performance. Thus, to enable concurrent NDP kernels, generalization of such scheme using many different addresses is required. In addition, since the memory-mapped registers are physical resources, it cannot be shared among multiple user processes safely and requires context switch to kernel space for every access. This challenge motivates us to design a low-overhead offloading mechanism based on CXL.mem that can be effective for both fine- and coarse-grained offloading while supporting high concurrency.

D. NDP Support with Unmodified CXL.mem

Whereas CXL.io can have high overhead for frequent and fine-grained communication, CXL.mem messages can be sent with lower latency [51], [97], [125] and CPU usage. The current CXL.mem protocol defines several unused bits in the packet format. Thus, one might consider using the bits to encode information required to implement special functionalities (e.g., NDP management) that are not defined in the standard.

However, to enable such customized communication, the host processor HW should be modified to support the special usage of the reserved bits. Thus, commodity processors that only support the standard protocol cannot utilize it. Furthermore, to send special packets, special instructions would need to be introduced in the host's ISA as in prior works [64], [78], [105]. Such propriety extension of the standard protocol or host's ISA would hinder widespread adoption.

III. Memory-Mapped Near-Data Processing

A. Overview

To overcome the limited flexibility and cost-efficiency of prior NDP approaches while avoiding the high latency overhead in the offloading procedure (§II-C) for NDP in CXL memory, we propose *Memory-Mapped Near-Data Processing (M^2NDP)* in CXL memory, called CXL-M^2NDP (Fig. 2). The M^2NDP comprises two mechanisms – 1) *Memory-Mapped functions (M^2func)* for low-overhead NDP management and offloading based on unmodified standard CXL.mem and 2) *Memory-Mapped μthreading ($M^2\mu$thr)* for cost-effective general-purpose NDP microarchitecture. They are combined to holistically improve end-to-end NDP performance including both offloading procedure and kernel execution. They are implemented in the CXL controller chip which also supports the basic read/write CXL.mem transactions.

B. Memory-mapped NDP Management Function (M^2func)

To exploit NDP for fine-grained computation offloading as well as coarse-grained offloading, the communication latency between the host and CXL-M^2NDP needs to be minimized. While the CXL.mem protocol provides low latency, the standard only defines packet types for normal CXL memory accesses and cannot be directly used for other communication. Extending CXL.mem to support custom packet types breaks its compatibility across different host processors and prevents widespread adoption of the NDP architecture. In contrast, CXL.io can be used for arbitrary communication, but incurs higher latency in the protocol stack and requires a context switch to the OS for privileged IO device communication, which further increases latency (§II-C).

Thus, to enable low-overhead and flexible communication with CXL-M^2NDP from the host using unmodified CXL.mem, we propose M^2func. Its basic idea is to reserve some physical memory space of the CXL memory for host communication referred to as the *M^2func region*. To distinguish between the two different usages of CXL.mem, we introduce a *packet filter* placed at CXL memory's input port to examine all packets and determine if the packet should be interpreted as normal

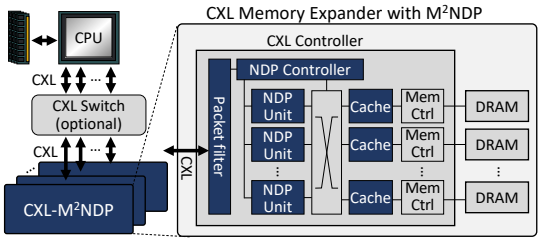


Fig. 2. Overview of the proposed system with M^2NDP-enabled CXL memory.

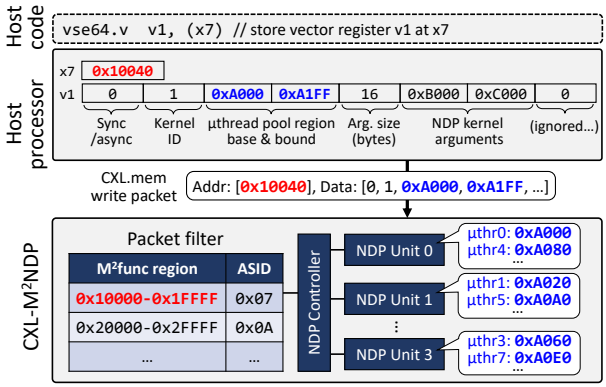


Fig. 3. Example NDP kernel launch using M^2func with VectorAdd NDP kernel that computes C=A+B. Vectors A, B, and C are placed at 0xA000, 0xB000, and 0xC000, respectively. Each μthread computes a 32B (8x4B) partial vector output. Other datapath components are not shown for brevity.

reads/writes or M^2func call based on the packet's address. M^2func calls are handled by the NDP controller (Fig. 2) implemented similarly to microcontrollers in GPUs [11]. M^2func can provide different functionalities, including NDP kernel registration, unregistration, and launch. Different functions can be called by using addresses with different offsets from the base of the M^2func region for the CXL.mem packet (Table II).

For the initialization of M^2func, each user process on the host allocates an uncacheable M^2func region in CXL memory. Through a CXL memory driver, the region's address range can be inserted into the packet filter using CXL.io. Once initialized, CXL.io is not needed anymore for NDP and CXL.mem can be used for both normal reads/writes and M^2func.

The packet filter entry requires little storage of only 18 B per host process (64-bit base, 64-bit bound, and 16-bit ASID), so a small packet filter can support many processes (e.g., 18 KB for 1024 processes). Given the small size, it can provide high throughput and be easily replicated in multi-ported (or multi-headed) CXL memory [15].

For an M^2func call, we use a write request format to include arguments in the write data portion of the request. To send it, the host executes a store instruction with a register that holds the arguments (Fig. 3). Vector registers [6], [17], [120] can be used to send multiple arguments up to the vector register's size. Because the M^2func region is uncacheable, the writes will bypass the host cache. However, the response to the write request cannot include any return value data from the NDP controller using the CXL.mem. Thus, we use a subsequent read request to the same address to access the return value of

TABLE II
PRE-DEFINED NDP MANAGEMENT FUNCTION DESCRIPTION.

Offset (stride=2^5)	Description	Privileged	M^2func Arguments	Return Value
$0 \ll 5$	NDP kernel registration	No	CodeLoc, ScratchpadMemSize, NumIntRegs, NumFloatRegs, NumVectorRegs	New kernel ID or -1 (error)
$1 \ll 5$	NDP kernel unregistration	No	NDPKernelID	0 (success) or -1 (error)
$2 \ll 5$	NDP kernel launch	No	Synchronicity, NDPKernelID μthreadPoolRegion (base, bound), KernelArgSize, KernelArguments	Kernel instance ID or -1 (error)
$3 \ll 5$	NDP kernel status poll	No	NDPKernelInstanceID	0 (finished) or 1 (running)
$4 \ll 5$	TLB entry shootdown	Yes	ASID, VirtualPageNumber	0 (success) or -1 (error)

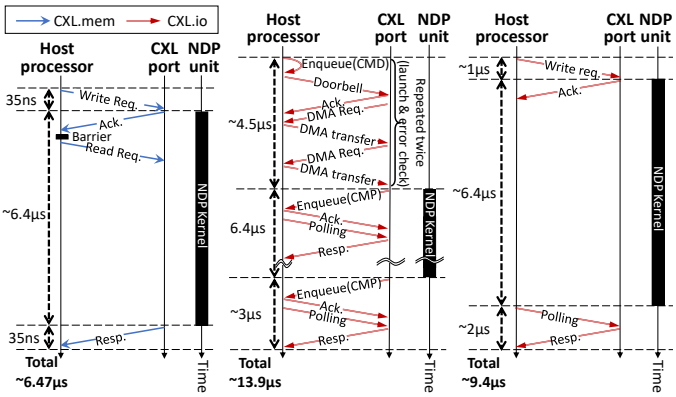
the latest call of the function by the current process. Because the return value will be accessed with normal memory access, the NDP controller can simply store the function's return value at the corresponding memory address and serve the read request as normal access. For proper ordering, the host process code should have a fence instruction between the requests.

Table II lists the NDP management functions for different address offsets from the base of the M^2func region. To support sufficient sizes for function arguments and return values, the offsets can be strided (by 32 B in this example). Thus, multiple arguments and return values can be communicated. For example, to register (unregister) an NDP kernel, assuming the base address is `0x00FF0000`, a write request to `0x00FF0000` (`0x00FF0020` or `0x00FF0000+(1≪5)`) can be used. Since different kernels can require varying amounts of register and scratchpad memory (§III-G), they are given as arguments for registration. In addition, the kernel argument size should be specified such that the arguments can be properly extracted from a kernel launch packet. The metadata of registered kernels are stored in the M^2func region for the current host process, beginning at a pre-determined location beyond the offsets used in Table II for ease of accesses by the host. As the M^2func region is allocated by each process, it is protected from other processes by the host.

C. NDP Kernel Launch

The M^2func enables NDP kernel launch with minimal overhead (Fig. 4a). NDP kernel launch can be done by calling the M^2func at offset $2 \ll 5$ (Table II) by sending a write request with kernel launch arguments. Note the difference between M^2func arguments for kernel launch function (which determines how a kernel is launched) and NDP *kernel* arguments (which will be directly used in the NDP kernel code). Large kernel inputs (e.g., arrays) can be stored in a separate memory location in CXL memory and its pointer can be passed as an argument. Each kernel instance is associated with a virtual memory region for an input or output data array called *μthread pool region* provided in a kernel launch call for our $M^2\mu$thr mechanism (§III-D). After a kernel launch, the NDP controller sends back an acknowledgment packet immediately.

Afterward, the host can have a memory fence and a load instruction to fetch the return value for the kernel launch function at the same M^2func offset $2 \ll 5$. The difference is



(a) M^2func (b) CXL.io (ring buffer) (c) CXL.io (direct)

Fig. 4. Example timelines with different NDP offloading schemes, assuming a synchronous launch and 6.4 μs NDP kernel runtime from `DLRM(SLS)-B32` (§IV-C). We assume ~4.5 μs for ring buffer latency [94], 2 μs latency for round-trip CXL.io/PCIe and kernel overhead [67], and 70 ns round-trip CXL.mem protocol latency (from 150 ns load-to-use latency for CXL memory [89], [114]). For the ring buffer, CMD and CMP refers to command and completion messages enqueued into the ring buffers, respectively. Two pairs of CMD and CMP are needed for kernel launch and error checks [20].

that this time, a read request will be sent. Its response with the return value can be sent back differently based on the `Synchronicity` argument given for kernel launch: it will be returned immediately for an asynchronous launch, whereas it will be returned after kernel termination for a synchronous launch. The asynchronous launch enables overlapping host-side computation with an NDP kernel. The host can then later use the kernel status poll function to check its completion.

When the NDP unit's available resource is insufficient for a kernel launch due to other kernels currently running, the kernel launch request will be buffered and served after prior kernels are completed. If the buffer is full, the kernel launch will return -1 to signify an error.

Comparison with traditional approaches. With the traditional ring buffer scheme used by various PCIe/CXL.io devices, an NDP kernel launch can require multiple link round-trips to update the write pointer, and transfer the pointer to the command from the ring buffer and then the command itself to the device similar to GPU kernel launches [92], [129] (Fig. 4b). Subsequently, to check if the launch is done without an error, the procedure should be repeated [20]. This approach incurs high latency but allows concurrent execution of multiple NDP kernels. On the other hand, a simpler approach of directly manipulating dedicated device registers through MMIO [43] takes a shorter latency (Fig. 4c) but can execute only one kernel at a time as the registers should not be overwritten.

In contrast to these approaches, M^2func reduces the kernel launch latency by requiring fewer round-trips compared to the ring buffer scheme while exploiting CXL.mem and avoiding kernel mode transition to further reduce latency. In addition, M^2func also supports concurrent execution of multiple kernels. As a result, we reduce the end-to-end latency of NDP by 31.2-53.5% compared to the traditional schemes for a fine-grained example NDP kernel from DLRM as shown in Fig. 4.

Note that while we focus on supporting NDP offloading

with CXL.mem to minimize overhead, we do not preclude the use of CXL.io for NDP management. For long kernels, CXL.io overhead can be well-amortized over the runtime.

D. Memory-mapped μthreading (M^2μthr)

To maximize the NDP kernel's memory bandwidth utilization, a large number of memory accesses need to be done concurrently to hide memory latency. While out-of-order cores can perform multiple memory accesses simultaneously, it is not suitable for cost-efficient NDP due to high control logic overhead. Fine-grained multithreading (FGMT), especially with a large number of threads as in GPUs, can efficiently provide high concurrency. However, GPU SM's SIMT-only execution can be inefficient when its threads perform redundant computation within a warp due to a lack of scalar operations (e.g., loop variable management, and address calculation) [56].

Thus, to efficiently support both scalar and SIMD operations, we adopt RISC-V ISA with vector extension and modify it to support highly concurrent FGMT-based M^2μthr (Table III). Particularly, for CPUs, the OS creates and manages threads, but the overhead can be tremendous for a large number of threads, especially if they are short-lived [134], due to μs-scale delay per thread [7], [88]. In addition, a CPU thread requires the entire ISA-defined register set, so the register file grows linearly with the HW thread count. However, memory-bound workloads tend to use fewer registers than compute-bound workloads due to lower arithmetic intensity. Thus, we use GPU-style HW-managed threads without the conventional OS for CPUs and provision the number of registers for each thread as specified by SW (i.e., compiler) during kernel registration (Table II) to reduce register file cost. For example, if 5 integer and 3 vector registers are needed, only registers x0-x4 and v0-v2 are used in the kernel. We refer to this type of thread as *μthread* due to its low resource usage. Creating a μthread can be done quickly as in GPUs. The μthreads can also use on-chip scratchpad memory for communication.

Despite similarities, our μthreads differ from GPU threads in several ways besides the difference in ISA (i.e., SIMT-only GPU ISA vs. SISD+SIMD by RISC-V ISA with vector extension for μthreads). First, whereas a GPU thread is identified by multidimensional threadblock and thread indices, μthreads are identified by the address it is mapped to in a μthread pool region. By using one of the input data arrays as a μthread pool region (Fig. 3), the μthread can begin data access without redundant address calculation done across threads in a GPU warp, which can account for a substantial ∼30% of dynamic instruction counts [56]. The mapped address is given as a (base, offset) pair (§III-E), so the offset can also be used to access other data with different bases.

Second, whereas GPU threads are created in a coarse threadblock granularity, μthreads are created in fine, individual thread granularity. The coarse-grained thread creation can result in resource fragmentation and underutilization due to inter-warp divergence – i.e., resource unused by finished warps of a threadblock will remain unused until the entire threadblock they belong to is finished and its resource is released for the next threadblock [135]. For example, Fig. 5 shows that, on a GPU SM used for NDP, the ratio of active warps varies between 0.5 and 1.0 over time with different threadblock sizes. In contrast, with M^2μthr, resources for a finished μthread are released immediately for the next μthread, improving resource utilization and performance/cost. While reducing the GPU's threadblock size can improve resource utilization in some cases, it can make it more difficult to effectively use the CUDA shared memory because different threadblocks cannot share data through shared memory. As a result, global memory traffic can be increased. While NVIDIA's Hopper GPU [19] introduces distributed shared memory that allows different threadblocks within a thread block cluster to share data in the on-chip shared memory, it requires that the threadblock be scheduled in the even coarser cluster granularity and can aggravate the SM resource underutilization issue (Fig. 5). By removing the threadblock hierarchy, M^2μthr also eliminates the need for optimizing the threadblock dimension, which can significantly affect performance [98].

The size of the data associated with a μthread equals the memory access granularity of the DRAM (e.g., 64 B for DDR5 and 32 B for LPDDR5). In contrast, GPU tends to process a larger amount of data in a warp (e.g., 128 B per warp with 32 threads processing FP32 data). As a result, for irregular workloads, there can be significant intra-warp control and memory divergence, lowering compute resource utilization. To load-balance NDP units, μthreads are mapped to NDP units in an interleaved manner with the memory-access granularity. The μthreads are concurrently executed in the bulk synchronous parallel model without any ordering guarantee as with CUDA threads in a GPU kernel. Thus, the NDP kernel should be written accordingly.

TABLE III
ARCHITECTURAL DIFFERENCES BETWEEN THE CPU, GPU, AND M^2NDP.

	CPU	GPU	M^2NDP
Thread creation granularity	**Each thread (fine-grained)**	Threadblock (corase-grained)	**Each μthread (fine-grained)**
Flynn's taxonomy	**SISD + SIMD**	SIMD (SIMT) only	**SISD + SIMD**
Per-thread registers	Fixed by ISA	**By usage**	**By usage**
Thread creation	By OS	**By HW**	**By HW**
Thread scheduling	ST/SMT/ FGMT/CGMT	**FGMT**	**FGMT**
Out-of-order exec.	Yes or No	**No**	**No**
Scratchpad memory scope	N/A	Threadblock	**All μthreads run on an NDP unit**
Thread Identification	Process ID	(Threadblock ID, thread ID)	**mapped μthread pool address**

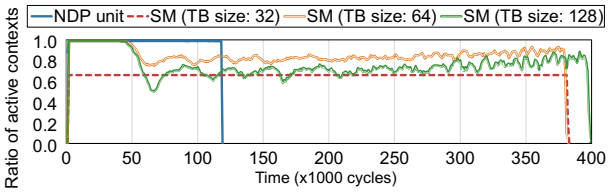


Fig. 5. Ratio of active contexts (i.e., warps for GPU SMs and μthreads for M^2μthr) executed on an SM or NDP unit over time for a main kernel of PGRANK benchmark [34] with configuration in §IV-A. Maximum threadblock count per SM limits the active warp ratio for the threadblock (TB) size of 32.

E. NDP Unit Microarchitecture

The NDP unit is designed at low cost while supporting general-purpose computation (Fig. 6). When an NDP kernel is launched, the NDP controller commands the *µthread generator* to spawn µthreads by allocating *µthread slots* and register file resources across the sub-cores of the NDP unit. Having multiple sub-cores instead of a monolithic core simplifies the dispatch unit. A µthread slots consist of a PC (program counter), CSR (configure and status register) of RISC-V, opcode and register IDs of the current instruction decoded, and base IDs for INT/FP/vector registers. The base register IDs are determined when each µthread is created and allocated the required registers for a kernel. Logical registers are renamed to physical registers simply by adding a logical ID to the base ID. In addition, the first two non-zero-valued scalar registers (i.e., x1 and x2) are initialized with the address and offset within the µthread pool associated with the µthread. After a µthread is allocated a slot, its PC is initialized with the kernel code location to begin execution.

An on-chip scratchpad memory is also available in each NDP unit for data sharing among all µthreads executed on the same NDP unit. The kernel arguments are also placed in the scratchpad memory after the launch. The scratchpad memory is mapped to the unused region in the virtual memory layout [49] and can be accessed using normal loads/stores.

A load/store unit for the scratchpad memory with atomic operations capability [8] is also provided to manipulate shared data in an NDP unit (e.g., for reduction by multiple µthreads). Global memory atomics are done at the memory-side L2 cache to avoid coherence issues (§III-F). Address translation is done using the on-chip TLBs, DRAM-TLB, and ATS (§III-H). The NDP unit can access any memory location in CXL memories in the system through on-chip and off-chip interconnects.

Instructions from a µthread are executed serially while different µthreads independently issue instructions with FGMT. Thus, complex dependency checks between instructions or data forwarding logic are not needed, minimizing control logic overhead. With sufficient µthread slots (e.g., 64 per NDP unit), the CXL memory bandwidth can be highly utilized. The width of the vector functional units is also matched with the DRAM access granularity (e.g., 32 B for LPDDR5) for high efficiency. When a µthread is finished, another µthread in the µthread pool is spawned in the idle slot.

F. Caches Hierarchy

To avoid the complexity of cache coherence, we adopt the cache hierarchy of the GPU [127], using write-through policy for L1 data cache of NDP units and placing the L2 cache in front of the memory controller (Fig. 2). L1 data cache's capacity is also configurable between normal L1 data cache and scratchpad memory. The L2 cache supports global memory atomic operations for data from DRAM. The NDP unit employs a small instruction cache because data-parallel, memory-bound workloads have relatively smaller instruction footprint than compute-bound workloads. To prevent access to stale code, the instruction caches are flushed when an NDP

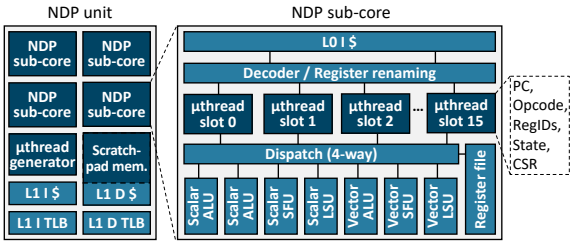


Fig. 6. Proposed NDP unit microarchitecture.

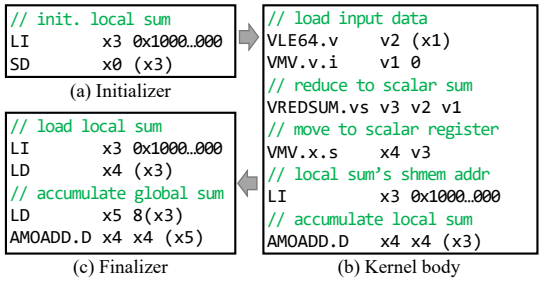


Fig. 7. NDP kernel example for reduction of a large data. It is assumed that the scratchpad memory is mapped to 0x10000000 and the final result will be stored at the location given in scratchpad memory at 0x10000008. `AMOADD` instruction performs atomic memory operation.

kernel is unregistered (§III-B). However, it would be done infrequently and have negligible performance impact.

G. NDP Kernel Structure

To support various use cases, an NDP kernel consists of an *initializer*, *kernel body*, and *finalizer*. The initializer (Fig. 7a) is executed only once when an NDP kernel is launched for initialization of scratchpad memory (if needed) and any required pre-computation before the main computation. For the initializer, one µthread is spawned in each µthread slot with a unique ID in the x2 (or offset) register (§III-E). When they are finished, the µthread generator starts spawning µthreads from the µthread pool region to execute the kernel body (Fig. 7b). There can be multiple kernel bodies such that when a kernel body is finished for all µthreads, all µthreads are generated again for the next kernel body. After all kernel bodies finish, the finalizer (Fig. 7c) is executed, similar to the initializer, but for post-processing and storing the result to DRAM if needed.

H. Virtual Memory Support

Our M^{2}NDP can efficiently support virtual memory. Because the host uses host physical addresses for normal CXL.mem requests, address translation is not needed for them. However, the µthread pool region is given with virtual address and NDP kernels also use virtual addresses for memory accesses. Our NDP unit employs TLBs (Fig. 6), but on-chip TLBs may not be sufficient for kernels that process large data in CXL memory. Although ATS is supported by CXL, its latency can be high (§II-B). Thus, we adopt DRAM-TLB [69], [109] to cost-effectively improve the TLB reach of NDP units and minimize the miss penalty of on-chip TLBs.

Each DRAM-TLB entry uses 16 bytes to store the ASID, tag, physical page number, and other attributes (e.g., permis-

sion bits). The location of a DRAM-TLB entry is computed based on the hash of the virtual page number and ASID as well as base address per CXL memory, ensuring that all NDP units within the same CXL memory can share them.

The DRAM-TLB can be implemented with low overhead. Even with the smallest 4 KB page size, the overhead of storing a DRAM-TLB entry is only 16 B/4 KB=0.4%, and for 2 MB pages, the overhead is negligible. If the DRAM-TLB region is sized such that its TLB reach is similar to the memory capacity of CXL memory, there will be few DRAM-TLB misses with the hashed location calculation, after DRAM-TLB warms up.

The on-chip and DRAM TLBs of CXL-M^{2}NDP can also keep translations for addresses in other CXL memories if they exist. A TLB shootdown needs to be done for all CXL-M^{2}NDPs if a page's mapping changes, but it can rarely occur for in-memory data we assume (i.e., no swapping to disks).

I. Scaling with Multiple Memory Expanders

Using direct P2P access between CXL devices through a CXL switch (§II-B), NDP kernels can access data from other CXL-M^{2}NDPs to process huge data. However, the CXL interface bandwidth can become a bottleneck for frequent P2P accesses, so localizing data across multiple CXL memories needs to be done carefully. Because different workloads exhibit varying memory access patterns, data partitioning schemes are typically specialized for target workloads [116]. For best performance, current multi-GPU systems also require the user-level SW to partition the data across GPUs and launch separate kernels. Thus, we similarly assume that the data are placed by SW across CXL memories and an NDP kernel is launched in each CXL-M^{2}NDP for multi-device scaling, and leave the exploration of automatic scaling for future work. However, the data localization does not have to be perfect since NDP units can directly access other CXL memories for reads and atomic operations similar to GPUs. We assume page-granularity data placement across them by the user for localization opportunity.

J. Concurrent NDP Kernel Execution

With M^{2}NDP, multiple NDP kernels from one or multiple users can be concurrently executed on the same or different NDP units, similar to the multi-process service (MPS) of GPUs [102]. However, resource sharing in any system introduces fundamental trade-offs between performance isolation, security, and resource utilization. The sharing of resources (e.g., DRAM) in the CXL memory can inevitably result in performance interference and security concerns between different NDP kernels and host processes that share them. GPU's MPS is also reported to have similar issues [27], [100]. Static partitioning of the resources, including the caches and memory channels can provide performance isolation and better security but may result in lower resource utilization, similar to multi-instance GPU (MIG) [27]. We leave an in-depth investigation of such trade-offs in NDP for future work, but as in cloud services in general, the system can provide different options to meet the requirements of different users [26].

IV. Evaluation

A. Methodology

We faithfully modeled the functional and timing aspects of CXL-M^{2}NDP with an in-house cycle-level simulator based on Ramulator [81]. Baseline CPU and GPU with passive CXL memory are modeled using modified ZSim [111] and Accel-Sim [77]; while CPUs are typically used as hosts, for data-parallel GPU workloads, we assume GPU as the host processor because GPUs integrated with CPU cores [74] can function as a host. Table IV gives the simulator configurations. In addition, we provide comparison with high-end CPU [12] and GPU cores [14] used for NDP within CXL memory, referred to as `CPU-NDP` and `GPU-NDP`, respectively. They represent prior approaches for general-purpose NDP.

For `CPU-NDP` evaluation for `OLAP` workload, we measure the performance on a dual-socket system with high-end AMD EPYC 75F3 CPUs (2.3 GHz) [12] that has the same total memory bandwidth as the CXL memory that we model (i.e., 409.6 GB/s). The evaluation was done using multiple copies of Apache Arrow processes and memory allocation was done locally to avoid the NUMA effect. We use 32 CPU cores in total (i.e., 16 cores per socket) to match the 32 NDP units

TABLE IV
SIMULATOR CONFIGURATION. WHEN MULTIPLE VALUES ARE GIVEN, THE DEFAULT IS INDICATED WITH BOLDFACE.

GPU	
Parameter	**Value**
SM count and freq.	82 SMs @ 1695 MHz
SM organization	Max. 32 threadblocks, Max. 1536 threads, 256 KB reg. file, 4 SP units, 4 DP units, 4 SFU units, 4 INT units, 4 INT units, 4 TC (tensor core) units
L1 D-cache	128 KB per SM, 128 B line, 32 B sector @ 1695 MHz
L2 cache	6 MB per GPU, 128 B line, 32 B sector @ 1695 MHz
NoC	82x48 crossbar (32B flit)
DRAM (GDDR6) organization and timing param. in clk	24 channels, 4 bankgroups/channel, 4 banks/bankgroup, tRC=78, tRCD=24, tCL=24, tRP=24, tCCDs=4, tCCDl=6, Freq=3500 MHz
CPU	
Parameter	**Value**
Cores	64 OoO cores @ 3.2 GHz
Caches	64 KB L1 (8-way, 4-cycle; 64 B line, LRU), 1 MB L2 (8-way, 12-cycle, 64 B line, LRU), 96 MB L3 (16-way, 74-cycle, 64 B line, LRU)
DRAM (timing parameters in clk)	DDR5-6400 with 409.6 GB/s (8 channels) tRC=149, tRCD=46, tCL=46, tRP=46
CXL Memory Expander	
Parameter	**Value**
CXL	64 GB/s (in each dir.) from CXL 3.0 (PCIe 6.0) x8, 256 B flit Load-to-use latency: **150 ns**, 300 ns, 600 ns
NoC	Four 32x32 crossbars (32B flit)
Memory-side L2 cache	4 MB (128 KB per memory channel, 16-way, 7-cycle, 128 B line, 32 B sector, LRU)
DRAM (timing parameters in clk)	LPDDR5 with 409.6 GB/s BW (32 channels similar to [132]), tRC=48, tRCD=15, tCL=20, tRP=15
NDP in CXL Memory	
Type	**Configuration**
M^{2}NDP (SC: sub-core)	32 NDP units @ 2 GHz, 4 SCs per NDP unit, 48 KB register file, 512 B L0 I-cache per SC, 2 KB L1 I-cache, 128 KB scratchpad/L1D cache, (16-way, 4-cycle, 128 B line, 32 B sector), 256-entry I-TLB, 256-entry D-TLB (8-way), Scalar units: 2 ALUs, 1 SFU, and 1 LSU per SC, 256-bit vector units: 1 vALU, 1 vSFU, and 1 vLSU per SC 16 μthread slots per SC, Max. concurrent kernels: 48
GPU-NDP	EqPerf(8SMs), 4×Perf(32SMs), 16×Perf(128SMs) @2 GHz, SM organization: same as the above GPU SM without TC,

we assume for M^{2}NDP. Note that M^{2}NDP has substantially lower cost than this CPU with OoO pipeline and large caches.

The `GPU-NDP(EqFLOPS)` uses eight Ampere GA102 SMs that provide equivalent peak FLOPS as the 32 NDP units in CXL-M^{2}NDP. `GPU-NDP(4×FLOPS)` and `GPU-NDP(16×FLOPS)` are also evaluated to show the impact of 4x and 16x higher SM counts (i.e., 32 and 128 SMs). All configurations except for `M2NDP` use CXL.io for kernel launch. The direct MMIO scheme (Fig. 4b), which uses dedicated device registers with a 3 μs latency overhead, is the default for CXL.io and is indicated with the `DR` suffix. The `RB` suffix indicates the ring buffer scheme with a 7.5 μs latency overhead (Fig. 4a). The `M2NDP` configuration uses CXL.mem-based M^{2}func for kernel launches with CXL.mem latency according to Table IV.

In the CXL memory, we assume fine-grained 256 B-granularity hashed interleaving across memory channels. For multiple CXL memories, we assume each page (2 MB) is mapped to a single CXL memory as in current NUMA or multi-GPU systems [110]. We assume the DRAM-TLB is warmed up for the CXL memory-resident data.

The CPU energy is modeled with McPAT [90] and for GPU and NDP units, we use AccelWattch [72], CACTI 6.5 [2], [99] (SRAM), and DSENT [122] (NoC). During NDP, we do include the energy of the idle host. We assumed an off-chip link energy of 8 pJ/bit [38].

B. Workloads

We use workloads from important domains – such as in-memory OLAP, No SQL, graph analytics, and deep learning – in Table V that require large memory capacity and exhibits little cache locality. We assume that the NDP kernel registration is done when the data are loaded in CXL memory. Because accelerators (e.g., GPUs) are not cache-coherent with most hosts (e.g., x86), we assumed that the host does not have dirty cachelines for the NDP kernel data unless otherwise mentioned, but show dirty host cache's impact in §IV-D.

In-memory OLAP. Filtering operations are commonly used in OLAP, but executing them from the host processor can make the CXL link a bottleneck. Thus, using NDP, we offload the `Evaluate` phase of the filtering operation, which sweeps column data to check the filtering condition and generates a boolean mask in the CXL memory because this phase is memory-intensive. For baseline, we use Polars [5], a high-performance columnar in-memory query engine based on Apache Arrow [1]. A subsequent `Filter` phase (creating a resulting filtered column) and other parts of query execution (e.g., query planning) can be efficiently executed on the host due to small memory footprints. We select queries from TPC-H [10] and SSB (Star Schema Benchmark) [103] that spend non-negligible time on filtering operations. To filter multiple columns, multiple NDP kernels are launched. The address range of the column data is used as the μthread pool region.

KVStore. For large KVStores, the CXL memory can store hash tables and key-value pairs [33], [44], [125]. Serving a KVStore request in such systems can require memory access through the CXL link for hash table lookup, key comparison, and linked list traversal (for hash collisions). Thus, the tail response latency can be increased for the baseline, but NDP can minimize data movement over CXL by offloading hash table lookup, reducing tail latency. We model a simplified Redis and offload GET/SET operations with NDP after compute-intensive hash function on the host. Request traces are obtained using YCSB [37] and have 10K requests for varying GET:SET ratios (G50:S50 for `KVS_A` and G95:S5 for `KVS_B`).

TABLE V
WORKLOADS USED FOR EVALUATION.

Baseline	Workload	Input problem
CPU	`OLAP` [10], [103]	TPC-H (Q6, Q14), SSB (Q1.1, Q1.2, Q1.3)
	`KVStore` [33]	24B key, 64B value, 10M KV items
GPU	`SPMV` [55]	28924 nodes, 1036208 edges
	`PGRANK` [34]	299067 nodes, 1955352 edges
	`SSSP` [34]	264346 nodes, 733846 edges
	`DLRM(SLS)` [101]	1000000 256-dimension vectors, 256 req.
	`OPT` [80] (Generation phase)	OPT-30B, OPT-2.7B with context length 1024

Graph analytics. Large graph analytics require high memory capacity [4] and can exploit CXL memory. As for the μthread pool region, we use the address range of the row pointers from the graph's CSR format. Each NDP kernel corresponds to a kernel in CUDA benchmarks [35], [55], [121].

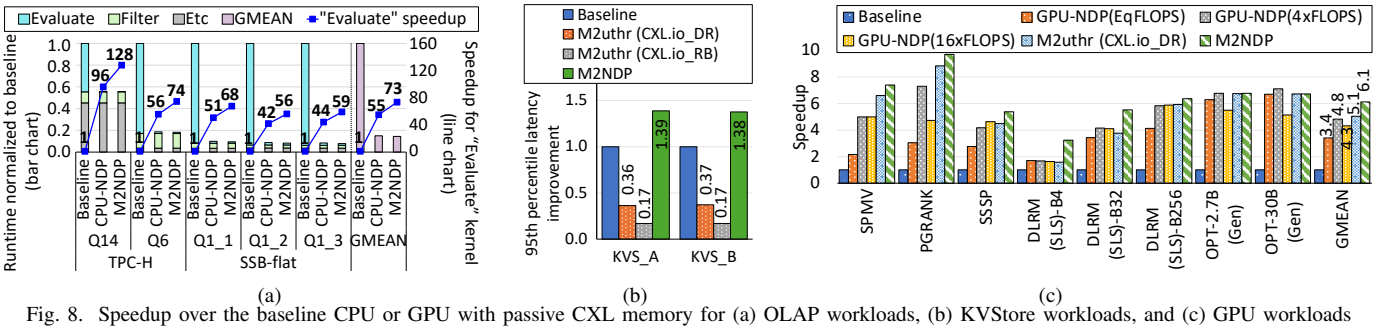
DLRM. Recommendation models can account for over 79% of inference cycles in datacenters [54]. The CXL memory can be used to cost-effectively store their TB-scale embedding tables [133]. However, the CXL link can be a bottleneck when the host accesses the embedding tables for the Sparse Length Sum (SLS) operations, which can account for up to 80% of runtime [101]. Thus, we offload it with NDP, using the output vector of SLS as μthread pool region. We use Criteo Dataset [40] for input with 80 embedding lookup operations per request [75] and use batch sizes of 4, 32, and 128.

LLM inference. Generative LLMs require large memory capacity from weight matrices and the key-value cache that grows linearly with the context length during the generation phase [104]. In addition, as GPUs are not efficiently utilized during the long generation phase [71], recent work proposed running this phase separately on GPUs with lower cost [104]. Thus, we evaluate NDP for a token generation with Meta's OPT-2.7B and OPT-30B models [139] assuming a batch size of 1 and KV cache of 1024 tokens. For the GPU baseline, we use the highly optimized inference kernels from vLLM [83] and NDP kernels are implemented similarly.

C. Performance

CPU workloads. Compared to the CPU baseline, our M^{2}NDP achieved significant speedups (Fig. 8a-b). For `evaluate` phase of `OLAP`, M^{2}NDP achieved significant speedups of up to 128× (73.4× on average) with a high 90.7% CXL memory's internal DRAM BW utilization on average (Fig. 8a). Even compared to M^{2}NDP even reached performance of an `Ideal NDP` with 100% DRAM BW utilization within 10.3%. Our NDP units also outperformed the `CPU-NDP` equipped with 32 aggressive out-of-order cores with large caches [12] by 33.7% on average.

For the KVStore, M^2μthr without M^{2}func exhibits slowdowns as hash table access kernels can be very short with



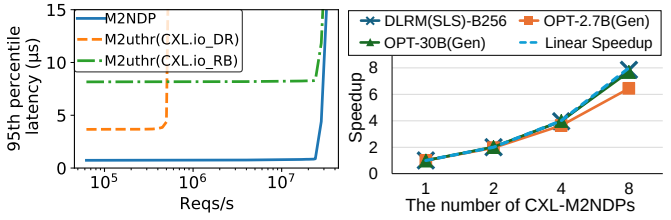
(a) (b) (c)

Fig. 8. Speedup over the baseline CPU or GPU with passive CXL memory for (a) OLAP workloads, (b) KVStore workloads, and (c) GPU workloads

95th percentile (p95) latency of 0.77μs, considering the long CXL.io latency of $\sim$2μs. In contrast, M^{2}func effectively addresses the communication overhead and reduces the p95 latency of requests by 38.2% on average (Fig. 8b).

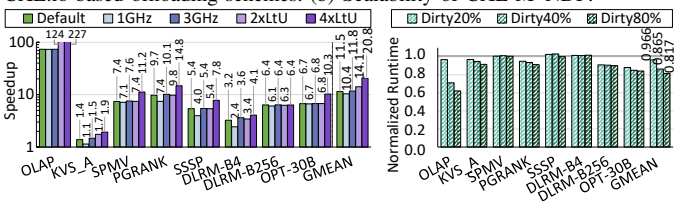
GPU workloads. M^{2}NDP achieved significant speedups of up to 9.71× (6.14× on average) compared to the baseline GPU by avoiding the CXL link BW bottleneck (Fig. 8c). By combining different techniques, our 32 NDP units (M^{2}NDP) even outperformed 128-SM `GPU-NDP(16×FLOPS)` by 1.42× by better utilizing the resources and reducing host communication overhead. The M^2μthr alone outperformed 128-SM by 16.6%. through efficient resource utilization. In addition, our NDP units (M^2μthr) significantly outperformed `GPU-NDP(EqFLOPS)` by up to 2.91× and 1.48× on average. The relative performance of graph workloads depended on the characteristics of the graph data. While our NDP unit uses four separate 256-bit SIMD units, a GPU SM issues instructions in 32-thread warp granularity, which is equivalent to 1024-bit SIMD width for 32-bit data. Thus, for the irregular graph workloads, the SMs suffer more from memory divergence depending on the graph data structure. `DLRM` requires frequent memory address calculations for different indices to the embedding table, resulting in a higher number of integer instructions across threads in a warp for the SIMT-only SMs, while M^2μthr is less affected by effectively using low-cost scalar units. For smaller batches with shorter kernel runtime, M^{2}NDP achieves a 91.5% performance improvement over `GPU-NDP(EqFLOPS)` by reducing kernel launch overhead. For the generation phase of `OPT`s, `GEMV` operations dominates the kernel runtime making them sensitive to the memory bandwidth. The speedup for `OPT`s approaches the ratio between the CXL's internal DRAM bandwidth and CXL link bandwidth ($\sim$ 6.4×) for all `GPU-NDP`s, M^2μthr, and M^{2}NDP except for `GPU-NDP(16×FLOPS)`; having more SMs can reduce performance as excessive memory accesses from more threadblocks reduce DRAM row buffer locality.

Impact of M^{2}func. By using low-overhead M^{2}func for host communication, M^{2}NDP achieved an additional speedup of up to 3.9× (34% overall) for CPU and GPU workloads compared to M^2μthr that uses CXL.io. It was particularly effective for fine-grained NDP kernels. In addition, compared to the device register-based offloading (§II-C) that cannot support concurrent NDP kernels, M^{2}func increased the throughput by 38.1× when requests were concurrently served (Fig. 9a).



(a) (b)

Fig. 9. (a) p95 latency of `KVS_A` with varying request rates using M^{2}func and CXL.io-based offloading schemes. (b) Scalability of CXL-M^{2}NDP.



(a) (b)

Fig. 10. (a) Speedup over the baseline by CXL-M^{2}NDP across different NDP unit frequencies and Load-to-Use (LtU) CXL memory latencies (`2xLtU`=300 ns, `4xLtU`=600 ns). (b) Normalized runtime with dirty cacheline ratios over clean host cache. `OLAP(Eval)` is the average from all queries' `Evaluate` part. For `KVStore`, we show p95 latency improvement.

D. Scalability and Sensitivity Study

Scalability. To evaluate the scalability of M^{2}NDP for `OPT` and `DLRM`, we partition the weight matrix or embedding table across different CXL-M^{2}NDPs using model parallelism [116]. As shown in Fig. 9b, we achieved near-linear speedups of 7.84× (7.69×) for `DLRM` (`OPT-30B`) with eight CXL-M^{2}NDPs. `OPT-2.7B` scaled less well with 6.45× speedup for 8 devices because the all-reduce took longer portion for smaller models.

Sensitivity study. Reducing the frequency of NDP units to 1 GHz degraded performance by 9.6% overall compared to the default 2 GHz (Fig. 10a). However, increasing the frequency to 3 GHz resulted in additional speedup of only 2.5% as the default frequency already utilized the memory BW well.

When load-to-use latency for CXL memory (from the host) was increased by 2-4× (`2xLtU` and `4xLtU`), the speedups by M^{2}NDP further increased to 14.1× and 20.8× on average, respectively, because the baseline suffered even more from the longer latency whereas M^{2}NDP kernels do not use the CXL link during execution and are unaffected by its latency.

In addition, when it is assumed that the host cache has a significant amount of dirty cachelines (between 20-80%) for the data accessed by NDP kernels, M^{2}NDP still provided good performance (Fig. 10b). Note that for our target workloads that require CXL memory expansion, the data size significantly

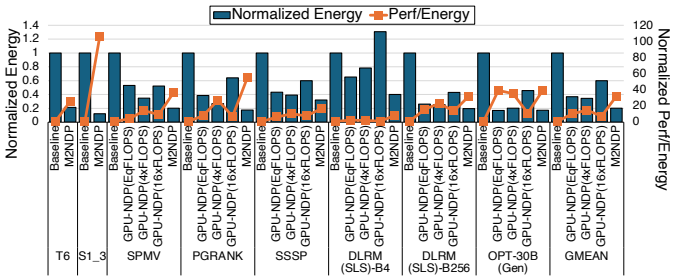


Fig. 11. Energy and performance per energy normalized to baseline CPU and GPU for OLAP and GPU workloads respectively. `T6` and `S1_3` represent TPC-H Q6, and SSB Q1.3 respectively. GMEAN is calculated using all OLAP and GPU workload benchmarks.

exceeds the host's cache size and these scenarios are very unlikely given that the kernel data (e.g., LLM weights and DLRM embedding table during inference) are not supposed to be modified during the kernel, but we present the results as a limit study. The performance impact was not significant, since even when BI is done by a μthread during NDP, other μthreads can continue execution and hide the latency. In addition, when the CXL memory BW is saturated, fetching some data from the host's cache through the CXL port can provide additional BW for moderate dirty cacheline ratios, countering the BI latency impact. For 20-80% dirty cacheline ratios, the NDP kernel runtime was affected by only 3.4-18.3% overall.

E. Energy

Compared to the baselines, M^{2}NDP significantly improved the performance per energy up to 106× and 32.0× on average (Fig. 11). For `OLAP`, M^{2}NDP substantially reduced energy consumption by up to 87.9% (83.9% on average) compared to the CPU baseline without NDP by reducing data movement over the CXL link and static/constant energy with lower runtime. Similarly, for GPU workloads, M^{2}NDP also significantly reduced energy compared to the baseline 77.6% on average. Compared to the `GPU-NDP(EqFLOPS)`, we reduced energy by up to 61.9% (37.4% on average).

F. Hardware Cost

We estimated the areas of caches and TLBs in the NDP unit using CACTI 6.5 and scaled them to 7 nm by using the node-scaling factor from [61] and obtained 0.47 mm^2 per NDP unit. The area of register files (integer, float, and vector) is estimated to be 0.11 mm^2. Each NDP unit has a unified L1 and scratchpad memory of 0.45 mm^2. With each μthread slot occupying 0.002 mm^2, a single NDP unit with compute units from [95] occupies 0.61 mm^2. Thus, the 32 NDP units that we assumed in the evaluation are estimated to incur an area overhead of only 19.4 mm^2.

V. Related Work

A. CXL Memory Expander

Several works studied the performance impact of CXL memory on cloud workloads and proposed memory placement schemes [79], [97], [125] as well as memory pooling [50], [89]. DirectCXL [51] also demonstrated the performance benefits of CXL.mem over RDMA. D. D. Sharma [113], [114] analyzed the CXL architecture and its performance.

B. Near-Data Processing and Processing-In-Memory

NDP in memory expanders. NDP logic in a memory expander. Several recent works proposed application-specific NDP in a memory expander or disaggregated memory for genome analysis [66], recommendation model [57], [84], [85], nearest neighbor [70], [118], and DNN parameter server [131]. In contrast, we propose a general-purpose NDP architecture for CXL memory to overcome their limited flexibility.

PIM. Recent DRAM-PIM designs implemented PIM units in DRAM to exploit the high DRAM-internal BW across all banks, targeting DNNs [59], [86], [87] or data-parallel workloads in general [39]. They have different trade-offs, including memory-bandwidth available, flexibility (e.g., instructions supported), communication between PIM units, and virtual memory support within PIM kernel. However, PIM reduces memory capacity [59] and is not suitable for workloads with huge memory footprints [4], [10], [32], [133]. PIM can also be combined with NDP in the same CXL memory for computation that cannot be localized in a single DRAM chip.

NDP in SSD. Several works explored NDP in SSD using CPU cores [52], [130], [137] or FPGA [91], [119], [124], [130] to exploit the high bandwidth and low latency available internally. However, there are significant gaps between DRAM and flash in terms of BW (e.g., 10 GB/s within SSD vs. 100s GB/s in CXL memory) and latency (10s of μs for flash vs. 10s of ns for DRAM). Still, for workloads with low BW demand (e.g., cold KV stores), NDP in SSD can be useful. Since our NDP units are memory device-agnostic and can saturate DRAM BW while being more cost-effective than CPU or GPU cores, they can be employed in the SSD for efficient general-purpose NDP. If CXL is used for the SSD's interface, our M^{2}func can also enable low-overhead kernel offloading. The speedup by NDP in SSD would be largely determined by its internal BW.

Other NDP approaches. Application-specific NDP in HMCs has been proposed for DNNs [48], [63], [93], linked-lists [62], [65], and graph workloads [22]. For programmable NDP, FPGA/CGRA has been proposed [28], [43], [47], [75], [76], [112], but they pose programmability challenges of mapping application algorithms to HW logic. Several works proposed placing simple NDP logic for very fine-grained NDP [23], [64], [78], but they do not support coarse-grained NDP and are not suitable for data-intensive NDP because the large number of offload command packets required can create a link BW bottleneck. Furthermore, they require modifying the memory protocol. These approaches also cannot work independently of the host CPU/GPU and are tightly coupled with the thread on the host – e.g., they require the host to send input data for each NDP thread. Thus, they are not suitable for scalable NDP in CXL memories. Some prior works introduced CPU or GPU cores in HMCs [41], [93], [107], [138], but our proposed M$^2\mu$thr can achieve higher efficiency with lightweight μthreads and flexible utilization resources (§III-D and IV-C). Several works explored offloading NDP operations to buffer chips of DIMMs [24], [25], [123], [141]. They are orthogonal to M^{2}NDP and can be used in the DIMMs of CXL memory.

VI. Conclusion

In this work, we propose memory-mapped NDP (M^2NDP) which enables a cost-effective, general-purpose NDP in CXL memory expanders by combining memory-mapped function (M^2func) and memory-mapped μthreading ($M^2\mu$thr). M^2func leverages the unmodified CXL.mem protocol for lightweight communication between the host and CXL device for NDP kernel launch and management, avoiding the high overhead of traditional PCIe/CXL.io-based schemes. $M^2\mu$thr introduces μthread, a lightweight thread with minimal register allocation, allowing a sufficient number of μthreads to be concurrently executed on a low-cost NDP unit. Allocation/deallocation of NDP unit's resources including μthread slots are also done more flexibly compared to GPU SMs, achieving higher resource utilization. Directly mapping μthreads to memory and providing scalar units also address the overhead of SIMT-only GPU warps. Compared to the baseline host processor with a passive CXL memory expander, M^2NDP can achieve significant speedups (up to 128×) for various applications that require large memory capacity, including in-memory OLAP, KVStore, LLM, DLRM, and graph analytics.

References

[1] "Apache Arrow." [Online]. Available: https://arrow.apache.org/docs/

[2] "Cacti: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model." [Online]. Available: https://www.hpl.hp.com/research/cacti/

[3] "Genoa - cores - amd," WikiChip. [Online]. Available: https://en.wikichip.org/wiki/amd/cores/genoa

[4] "Graph500 Benchmark specification." [Online]. Available: https://graph500.org/?page_id=12

[5] "Polars: Lightning-fast DataFrame library for Rust and Python." [Online]. Available: https://www.pola.rs/

[6] "RISC-V "V" Vector Extension." [Online]. Available: https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc

[7] "Thread management," Threading Programming Guide, Apple. [Online]. Available: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/CreatingThreads/CreatingThreads.html

[8] "Vector amo extension," RISC-V Vector extension specification, May. [Online]. Available: https://github.com/riscv/riscv-v-spec/blob/master/v-amo.adoc

[9] "Address translation services revision 1.1." Peripheral Component Interconnect Special Interest Group (PCI-SIG)., 2009. [Online]. Available: https://www.pcisig.com/specifications/iov/ats/

[10] "TPC BENCHMARK™ H (Decision Support) Standard Specification Revision 2.17.1," Transaction Processing Performance Council (TPC), November 2014. [Online]. Available: https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf

[11] "RISC-V in NVIDIA," 6th RISC-V Workshop, May 2017. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/Tue1345pm-NVIDIA-Sijstermans.pdf

[12] "AMD EPYC™ 75F3," March 2021. [Online]. Available: https://www.amd.com/ko/products/cpu/amd-epyc-75f3

[13] "Hardware-based cache flush engine," Arm CoreLink™ CI-700 Coherent Interconnect Technical Reference Manual, May 2021. [Online]. Available: https://developer.arm.com/documentation/101569/0300/SLC-memory-system/SLC-memory-system-components-and-configuration/Hardware-based-cache-flush-engine?lang=en

[14] "NVIDIA AMPERE GA102 GPU ARCHITECTURE," 2021. [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[15] "Compute Express Link Specification 3.1," CXL Consortium, August 2023.

[16] "Compute Express Link Specification 3.1," CXL Consortium, August 2023, section 3.3.2.1 "Direct P2P CXL.mem for Accelerators".

[17] "Intel® Advanced Vector Extensions 10 Architecture Specification," July 2023. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/784267

[18] "Intel® fpga compute express link (cxl) ip," May 2023. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/cxl-ip.html

[19] "NVIDIA H100 Tensor Core GPU Architecture," 2023. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

[20] "CUDA C++ Best Practices Guide," March 2024, section 1.4. "Recommendations and Best Practices". [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#recommendations-and-best-practices

[21] M. Adnan, Y. E. Maboud, D. Mahajan, and P. J. Nair, "Accelerating recommendation system training by leveraging popular choices," *Proc. VLDB Endow.*, vol. 15, no. 1, p. 127–140, sep 2021. [Online]. Available: https://doi.org/10.14778/3485450.3485462

[22] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," p. 105–117, 2015. [Online]. Available: https://doi.org/10.1145/2749469.2750386

[23] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, New York, NY, USA, 2015, p. 336–348.

[24] M. Alian and N. S. Kim, "Netdimm: Low-latency near-memory network interface architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 699–711. [Online]. Available: https://doi.org/10.1145/3352460.3358278

[25] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O'Halloran, D. Chen, J. Xiong, D. Kim, W.-m. Hwu, and N. S. Kim, "Application-transparent near-memory processing architecture with memory channel network," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 802–814. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00070

[26] Amazon. Configure instance tenancy with a launch configuration. [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-dedicated-instances.html

[27] R. Armstrong, "S41793 - optimizing gpu utilization: Understanding mig and mps," NVIDIA GTC 2022. [Online]. Available: https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41793/

[28] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783753

[29] D. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses: The programmability of fpgas must improve if they are to be part of mainstream computing." *Queue*, vol. 11, no. 2, p. 40–52, feb 2013. [Online]. Available: https://doi.org/10.1145/2436696.2443836

[30] A. Biswas, "Sapphire rapids," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–22.

[31] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, p. 316–331.

[32] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[33] J. Carlson, *Redis in action*. Simon and Schuster, 2013.

[34] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC 2013, Portland, OR, USA, September 22-*

24, 2013. IEEE Computer Society, 2013, pp. 185–195. [Online]. Available: https://doi.org/10.1109/IISWC.2013.6704684

[35] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[36] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "Nvidia a100 tensor core gpu: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, 2021.

[37] 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*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[38] B. Dally, "Gtc china 2020 keynote," https://investor.nvidia.com/events-and-presentations/events-and-presentations/event-details/2020/GTC-China-2020-Keynote-Bill-Dally/default.aspx, 2020, [Online; accessed 18-February-2022].

[39] F. Devaux, "The true processing in memory accelerator," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1–24.

[40] Diemert Eustache, Meynet Julien, P. Galland, and D. Lefortier, "Attribution modeling increases efficiency of bidding in display advertising," in *Proceedings of the AdKDD and TargetAd Workshop, KDD, Halifax, NS, Canada, August, 14, 2017*. ACM, 2017, p. To appear.

[41] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The mondrian data engine," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 639–651. [Online]. Available: https://doi.org/10.1145/3079856.3080233

[42] N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. W. Yu, O. Firat, B. Zoph, L. Fedus, M. P. Bosma, Z. Zhou, T. Wang, E. Wang, K. Webster, M. Pellat, K. Robinson, K. Meier-Hellstern, T. Duke, L. Dixon, K. Zhang, Q. Le, Y. Wu, Z. Chen, and C. Cui, "GLaM: Efficient scaling of language models with mixture-of-experts," in *Proceedings of the 39th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., vol. 162. PMLR, 17–23 Jul 2022, pp. 5547–5569. [Online]. Available: https://proceedings.mlr.press/v162/du22c.html

[43] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 283–295.

[44] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, p. 5, aug 2004.

[45] M. Flajslik and M. Rosenblum, "Network interface design for low latency Request-Response protocols," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, Jun. 2013, pp. 333–346. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/flajslik

[46] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 113–124.

[47] M. Gao and C. Kozyrakis, "Hrl: Efficient and flexible reconfigurable logic for near-data processing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 126–137. [Online]. Available: https://doi.org/10.1109/HPCA.2016.7446059

[48] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 751–764. [Online]. Available: https://doi.org/10.1145/3037697.3037702

[49] A. Ghiti, "Virtual memory layout on risc-v linux," February 2021. [Online]. Available: https://docs.kernel.org/riscv/vm-layout.html

[50] D. Gouk, M. Kwon, H. Bae, S. Lee, and M. Jung, "Memory pooling with cxl," *IEEE Micro*, vol. 43, no. 2, pp. 48–57, 2023. [Online]. Available: https://doi.org/10.1109/MM.2023.3237491

[51] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, High-Performance memory disaggregation with DirectCXL," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 287–294. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/gouk

[52] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A framework for near-data processing of big data workloads," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 153–165. [Online]. Available: https://doi.org/10.1109/ISCA.2016.23

[53] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. Kim, "Bigdebug: Debugging primitives for interactive big data processing in spark," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 784–795. [Online]. Available: https://doi.org/10.1145/2884781.2884813

[54] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The architectural implications of facebook's dnn-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 488–501.

[55] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system," *IEEE Access*, vol. 10, pp. 52 565–52 608, 2022.

[56] D. Ha, Y. Oh, and W. W. Ro, "R2d2: Removing redundancy utilizing linearity of address generation in gpus," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589039

[57] M. Ha, J. Sim, D. Moon, M. Rhee, J. Choi, B. Koh, E. Lim, and K. Park, "Cms: A computational memory solution for high-performance and power-efficient recommendation system," in *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2022, pp. 491–494. [Online]. Available: https://doi.org/10.1109/AICAS54282.2022.9869851

[58] B. Harris and N. Altiparmak, "When poll is more energy efficient than interrupt," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 59–64. [Online]. Available: https://doi.org/10.1145/3538643.3539747

[59] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. N. Vijaykumar, "Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. [Online]. Available: https://doi.org/10.1109/MICRO50266.2020.00040

[60] B. Herzog, L. Gerhorst, B. Heinloth, S. Reif, T. Hönig, and W. Schröder-Preikschat, "Intspect: Interrupt latencies in the linux kernel," in *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2018, pp. 83–90. [Online]. Available: https://doi.org/10.1109/SBESC.2018.00021

[61] M. Hibben, "Tsmc, not intel, has the lead in semiconductor processes," https://seekingalpha.com/article/4151376-tsmc-not-intel-lead-in-semiconductor-processes, 2018.

[62] B. Hong, G. Kim, J. H. Ahn, Y. Kwon, H. Kim, and J. Kim, "Accelerating linked-list traversal through near-data processing," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016, pp. 113–124. [Online]. Available: https://doi.org/10.1145/2967938.2967958

[63] B. Hong, Y. Ro, and J. Kim, "Multi-dimensional parallel training of winograd layer on memory-centric architecture," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 682–695. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00061

[64] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016. [Online]. Available: https://doi.org/10.1109/ISCA.2016.27

[65] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation," in *2016 IEEE 34th*

International Conference on Computer Design (ICCD), 2016, pp. 25–32. [Online]. Available: https://doi.org/10.1109/ICCD.2016.7753257

[66] W. Huangfu, K. T. Malladi, A. Chang, and Y. Xie, "Beacon: Scalable near-data-processing accelerators for genome analysis near memory pool with the cxl support," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 727–743. [Online]. Available: https://doi.org/10.1109/MICRO56248.2022.00057

[67] C. Hwang, K. Park, R. Shu, X. Qu, P. Cheng, and Y. Xiong, "ARK: GPU-driven code execution for distributed deep learning," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 87–101. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/hwang

[68] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, May 2023, chapter 9.4.7 "CLFLUSHOPT Instruction".

[69] A. Jaleel, E. Ebrahimi, and S. Duncan, "Ducati: High-performance address translation by extending tlb reach of gpu-accelerated systems," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 1, mar 2019. [Online]. Available: https://doi.org/10.1145/3309710

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

[71] Y. Jin, C.-F. Wu, D. Brooks, and G.-Y. Wei, "S^3: Increasing gpu utilization during generative inference for higher throughput," 2023.

[72] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T. G. Rogers, T. M. Aamodt, and N. Hardavellas, "Accelwattch: A power modeling framework for modern gpus," in *54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. [Online]. Available: https://doi.org/10.1145/3466752.3480063

[73] U. Kang, "Adding new value to memory subsystems through cxl," Flash Memory Summit, August 2022. [Online]. Available: https://memverge.com/wp-content/uploads/2022/08/CXL-Forum_SKhynix.pdf

[74] Karl Freund, "Will AMD's MI300 Beat NVIDIA In AI?" [Online]. Available: https://www.forbes.com/sites/karlfreund/2023/01/09/will-amds-mi300-beat-nvidia-in-ai/

[75] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20, 2020, p. 790–803.

[76] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, K. Kim, J. Jung, I. Yun, S. J. Park, H. Park, J. Song, J. Cho, K. Sohn, N. S. Kim, and H.-H. S. Lee, "Near-memory processing in action: Accelerating personalized recommendation with axdimm," *IEEE Micro*, pp. 1–1, 2021. [Online]. Available: https://doi.org/10.1109/MM.2021.3097700

[77] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00047

[78] G. Kim, N. Chatterjee, M. O'Connor, and K. Hsieh, "Toward standardized near-data processing with unrestricted data placement for gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. [Online]. Available: https://doi.org/10.1145/3126908.3126965

[79] K. Kim, H. Kim, J. So, W. Lee, J. Im, S. Park, J. Cho, and H. Song, "Smt: Software-defined memory tiering for heterogeneous computing systems with cxl memory expander," *IEEE Micro*, vol. 43, no. 2, pp. 20–29, 2023. [Online]. Available: https://doi.org/10.1109/MM.2023.3240774

[80] S. Kim, C. Hooper, T. Wattanawong, M. Kang, R. Yan, H. Genc, G. Dinh, Q. Huang, K. Keutzer, M. W. Mahoney, Y. S. Shao, and A. Gholami, "Full stack optimization of transformer inference: a survey," 2023.

[81] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[82] R. Kuper, I. Jeong, Y. Yuan, J. Hu, R. Wang, N. Ranganathan, and N. S. Kim, "A quantitative analysis and guideline of data streaming accelerator in intel 4th gen xeon scalable processors," *CoRR*, vol. abs/2305.02480, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.02480

[83] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[84] Y. Kwon, Y. Lee, and M. Rhu, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, p. 740–753.

[85] Y. Kwon, Y. Lee, and M. Rhu, "Tensor casting: Co-designing algorithm-architecture for personalized recommendation training," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 235–248. [Online]. Available: https://doi.org/10.1109/HPCA51647.2021.00029

[86] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, K. Vladimir, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, J. Lee, D. Ko, Y. Jun, K. Cho, I. Kim, C. Song, C. Jeong, D. Kwon, J. Jang, I. Park, J. Chun, and J. Cho, "A 1ynm 1.25v 8gb, 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep-learning applications," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 1–3. [Online]. Available: https://doi.org/10.1109/ISSCC42614.2022.9731711

[87] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware architecture and software stack for pim based on commercial dram technology : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 43–56. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00013

[88] D. Lemire, "Cost of a thread in c++ under linux." [Online]. Available: https://lemire.me/blog/2020/01/30/cost-of-a-thread-in-c-under-linux/

[89] 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, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 574–587. [Online]. Available: https://doi.org/10.1145/3575693.3578835

[90] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480.

[91] S. Liang, Y. Wang, C. Liu, H. Li, and X. Li, "Ins-dla: An in-ssd deep learning accelerator for near-data processing," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 173–179.

[92] M. LILJESON. GPU submission strategies. AMD. [Online]. Available: https://gpuopen.com/presentations/2022/gpuopen-gpu_submission-reboot_blue_2022.pdf

[93] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 655–668. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00059

[94] D. Lustig and M. Martonosi, "Reducing gpu offload latency via fine-grained cpu-gpu synchronization," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 354–365. [Online]. Available: https://doi.org/10.1109/HPCA.2013.6522332

[95] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, "Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 04, pp. 774–787, apr 2021.

[96] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, "Asic clouds: Specializing the datacenter," in *2016 ACM/IEEE 43rd Annual*

International Symposium on Computer Architecture (ISCA), 2016, pp. 178–190. [Online]. Available: https://doi.org/10.1109/ISCA.2016.25

[97] 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, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 742–755. [Online]. Available: https://doi.org/10.1145/3582016.3582063

[98] P. Micikevicius, "Performance optimization: Programming guidelines and gpu architecture reasons behind them," NVIDIA GPU Technology Conference, 2013. [Online]. Available: https://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf

[99] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, April 2009.

[100] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered insecure: Gpu side channel attacks are practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2139–2153. [Online]. Available: https://doi.org/10.1145/3243734.3243831

[101] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," 2019.

[102] NVIDIA. Multi-process service. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[103] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "The star schema benchmark and augmented fact table indexing," in *Performance Evaluation and Benchmarking*, R. Nambiar and M. Poess, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 237–252.

[104] P. Patel, E. Choukse, C. Zhang, Íñigo Goiri, A. Shah, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," 2023.

[105] A. Pattnaik, X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Opportunistic computing in gpu architectures," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, 2019, p. 210–223.

[106] J. Prout, "Expanding Beyond Limits With CXL™-based Memory," Flash Memory Summit, August 2022. [Online]. Available: https://memverge.com/wp-content/uploads/2022/08/CXL-Forum_Samsung.pdf

[107] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 190–200. [Online]. Available: https://doi.org/10.1109/ISPASS.2014.6844483

[108] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "Hemem: Scalable tiered memory management for big data applications and real nvm," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 392–407. [Online]. Available: https://doi.org/10.1145/3477132.3483550

[109] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 469–480. [Online]. Available: https://doi.org/10.1145/3079856.3080210

[110] N. Sakharnykh, "Everything you need to know about unified memory," NVIDIA GPU Technology Conference, 2018.

[111] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, p. 475–486, jun 2013. [Online]. Available: https://doi.org/10.1145/2508148.2485963

[112] B. C. Schwedock, P. Yoovidhya, J. Seibert, and N. Beckmann, "Täkō: A polymorphic cache hierarchy for general-purpose optimization of data movement," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2022, p. 42–58.

[113] D. D. Sharma, "Compute express link (cxl): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy," *IEEE Micro*, vol. 43, no. 2, pp. 99–109, 2023.

[114] D. D. Sharma, "Novel composable and scaleout architectures using compute express link," *IEEE Micro*, vol. 43, no. 2, pp. 9–19, 2023.

[115] D. D. Sharma, R. Blankenship, and D. S. Berger, "An introduction to the compute express link (CXL) interconnect," *CoRR*, vol. abs/2306.11227, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.11227

[116] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020.

[117] A. Shrivastava, V. Lakshman, T. Medini, N. Meisburger, J. Engels, D. Torres Ramos, B. Geordie, P. Pranav, S. Gupta, Y. Adunukota, and S. Jain, "From research to production: Towards scalable and sustainable neural recommendation models on commodity cpu hardware," in *Proceedings of the 17th ACM Conference on Recommender Systems*, ser. RecSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1071–1074. [Online]. Available: https://doi.org/10.1145/3604915.3610249

[118] J. Sim, S. Ahn, T. Ahn, S. Lee, M. Rhee, J. Kim, K. Shin, D. Moon, E. Kim, and K. Park, "Computational cxl-memory solution for accelerating memory-intensive applications," *IEEE Computer Architecture Letters*, vol. 22, no. 1, pp. 5–8, 2023. [Online]. Available: https://doi.org/110.1109/LCA.2022.3226482

[119] M. Soltaniyeh, V. L. Moutinho Dos Reis, M. Bryson, R. Martin, and S. Nagarakatte, "Near-storage acceleration of database query processing with smartssds," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 265–265. [Online]. Available: https://doi.org/10.1109/FCCM51124.2021.00052

[120] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017. [Online]. Available: https://doi.org/10.1109/MM.2017.35

[121] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[122] C. Sun, C.-H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "Dsent - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *Proceedings of the 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, ser. NOCS '12. USA: IEEE Computer Society, 2012, p. 201–210.

[123] W. Sun, Z. Li, S. Yin, S. Wei, and L. Liu, "Abc-dimm: Alleviating the bottleneck of communication in dimm-based near-memory processing with inter-dimm broadcast," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE Press, 2021, p. 237–250. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00027

[124] X. Sun, H. Wan, Q. Li, C.-L. Yang, T.-W. Kuo, and C. J. Xue, "Rm-ssd: In-storage computing for large-scale recommendation inference," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 1056–1070.

[125] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. J. 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 *MICRO-56: 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23, 2023.

[126] S. Tamimi, F. Stock, A. Koch, A. Bernhardt, and I. Petrov, "An evaluation of using ccix for cache-coherent host-fpga interfacing," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022, pp. 1–9. [Online]. Available: https://doi.org/10.1109/FCCM53951.2022.9786103

[127] G. Thomas-Collignon and V. Mehta, "Optimizing cuda applications for nvidia a100 gpu," NVIDIA GTC, 2020. [Online]. Available: https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21819-optimizing-applications-for-nvidia-ampere-gpu-architecture.pdf

[128] B. Tian, Q. Chen, and M. Gao, "Abndp: Co-optimizing data access and load balance in near-data processing," in *Proceedings of the 28th ACM*

International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 3–17. [Online]. Available: https://doi.org/10.1145/3582016.3582026

[129] K. Tian, Y. Dong, and D. Cowperthwaite, "A full GPU virtualization solution with mediated Pass-Through," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 121–132. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/tian

[130] T. Vinçon, L. Weber, A. Bernhardt, A. Koch, I. Petrov, C. Knödler, S. Hardock, S. Tamimi, and C. Riegger, "nkv in action: Accelerating kv-stores on nativecomputational storage with near-data processing," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 2981–2984, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p2981-vincon.pdf

[131] Z. Wang, J. Sim, E. Lim, and J. Zhao, "Enabling efficient large-scale deep learning training with cache coherent disaggregated memory systems," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[132] Wikipedia. Apple m3. [Online]. Available: https://en.wikipedia.org/wiki/Apple_M3

[133] M. Wilkening, U. Gupta, S. Hsia, C. Trippel, C.-J. Wu, D. Brooks, and G.-Y. Wei, "Recssd: Near data processing for solid state drive based recommendation inference," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021, p. 717–729.

[134] H. Wu and M. Becchi, "Evaluating thread coarsening and low-cost synchronization on intel xeon phi," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 1018–1029. [Online]. Available: https://doi.org/10.1109/IPDPS47924.2020.0010

[135] P. Xiang, Y. Yang, and H. Zhou, "Warp-level divergence in gpus: Characterization, impact, and mitigation," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 284–295.

[136] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12. USA: USENIX Association, 2012, p. 3.

[137] Z. Yang, Y. Lu, X. Liao, Y. Chen, J. Li, S. He, and J. Shu, "λ-IO: A unified IO stack for computational storage," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*. Santa Clara, CA: USENIX Association, Feb. 2023, pp. 347–362. [Online]. Available: https://www.usenix.org/conference/fast23/presentation/yang-zhe

[138] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "Top-pim: Throughput-oriented programmable processing in memory," in *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '14, 2014, p. 85–98.

[139] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," 2022.

[140] W. Zhang, Q. Chen, K. Fu, N. Zheng, Z. Huang, J. Leng, and M. Guo, "Astraea: Towards qos-aware and resource-efficient multi-stage gpu services," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 570–582. [Online]. Available: https://doi.org/10.1145/3503222.3507721

[141] Z. Zhou, C. Li, F. Yang, and G. Sun, "Dimm-link: Enabling efficient inter-dimm communication for near-memory processing," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 302–316. [Online]. Available: https://doi.org/10.1109/HPCA56546.2023.10071005