CXL-Bench: Benchmarking Shared CXL Memory Access

Marcel Weisgut
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
marcel.weisgut@hpi.de

Daniel Ritter
SAP
Walldorf, Germany
daniel.ritter@sap.com

Florian Schmeller
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
florian.schmeller@hpi.de

Pınar Tözün
IT University of Copenhagen
Copenhagen, Denmark
pito@itu.dk

Tilmann Rabl
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
tilmann.rabl@hpi.de

ABSTRACT

Memory access paths between a CPU core and memory are increasingly complex. Data can be placed on local- or remote-socket memory, and on local- and remote-die memory on modern multi-die CPUs, affecting memory access performance. Cache-coherent inter-device interconnects, such as Compute Express Link (CXL), allow a CPU core to perform load and store instructions to memory of a peripheral device. Such accesses incur higher access latency than accesses to local-socket memory and increase the access path complexity. For database system developers, it is important to understand the performance implications of these complex memory architectures. In this work, we present CXL-Bench, a benchmark framework for quantifying access performance for different memory access paths. CXL-Bench provides many configuration options, such as memory access patterns, the operating system's memory abstraction, cache bypass options, and a distributed mode for setups with multiple servers accessing memory of the same device. We demonstrate the utility of CXL-Bench by quantifying memory access characteristics of two servers accessing a shared CXL 1.1 memory device. Our results show that memory accesses of one server to the device affect the access performance of another server accessing the same device. On the other hand, memory (de)allocations using CXL memory configured as a character device complete quickly, making frequent re-allocation of CXL memory feasible.

VLDB Workshop Reference Format:
Marcel Weisgut, Daniel Ritter, Florian Schmeller, Pınar Tözün, and Tilmann Rabl. CXL-Bench: Benchmarking Shared CXL Memory Access. VLDB 2025 Workshop: 16th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS25).

VLDB Workshop Artifact Availability:
The source code has been made available at https://github.com/hpides/cxlbench/tree/paper/adms25.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

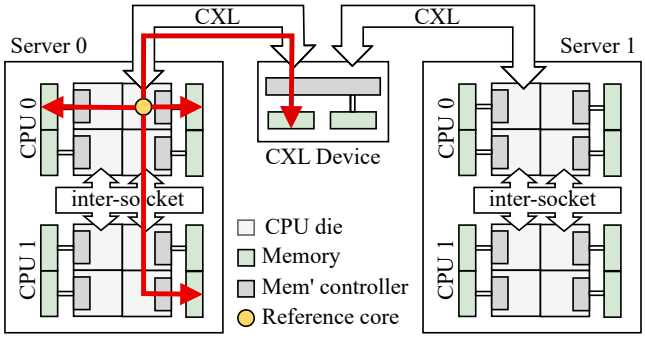

Figure 1: Heterogeneous memory access paths.

1 INTRODUCTION

CPUs of modern servers have increasingly heterogeneous and complex memory access paths as shown in Figure 1. A CPU core's memory access latency and bandwidth vary significantly, depending on whether the core accesses socket-local or socket-remote memory [26]. Memory access latency varies even within a single socket: on modern multi-die CPUs, data that a core accesses can be placed in the memory of a local die or an adjacent remote die, significantly affecting access latency [15, 33]. New cache-coherent inter-device interconnects, such as NVLink C2C [41] and Compute Express Link (CXL) [37], allow a CPU core to perform load and store instructions to the memory of a peripheral device. Such accesses to remote device memory incur higher access latency than CPU-local memory accesses [27, 38, 41] and further increase the memory access path complexity.

CXL is gaining attention in research and industry, as it enables large memory pools physically separated from CPUs and shared with multiple compute nodes [2, 3, 8, 10, 19]. The availability of the CXL interconnect has opened new design space opportunities for memory-intensive applications [10, 19, 25]. One such opportunity is building memory-centric database architectures [10]. While a CPU's load and store instructions are commonly executed to memory on the same server, CXL invalidates this assumption. With CXL, multiple servers can be connected to the same CXL device and issue memory accesses to it simultaneously.

Memory access performance depends on factors such as the CPU design, the CPU instructions used for memory access, the CPU's cache coherence protocol, the memory media type, the interconnect and memory controllers, as well as the memory abstraction of the operating system (OS). The different memory access paths and OS abstraction of heterogeneous memory types increase the data placement complexity for data processing systems. To make reasonable decisions in terms of performance, one needs to know memory access characteristics, such as access latency and bandwidth.

Contributions. In this work, we present CXL-Bench, a configurable benchmark framework for quantifying memory access performance across different interconnect and memory types. The framework contains many user-configurable options, such as memory access patterns, the OS's memory abstraction used, access operations, cache bypass options, and a distributed mode for setups with multiple servers accessing memory of the same device. Using CXL-Bench, we analyze memory access performance of two servers connected to a shared CXL memory device. We study:

- the latency and throughput of loads and stores, and the latency of atomic operations on a shared CXL memory (Section 4).
- the latency of virtual-to-physical memory mapping for CXL shared memory (Section 5).

The rest of this paper is structured as follows: In Section 2, we discuss relevant background information on the CXL interconnect standard, CXL memory configuration and allocation options, and memory access operations. Section 3 gives an overview of our benchmarking framework. We present our load and store, as well as atomic operation microbenchmark results in Section 4 and the memory mapping experiments in Section 5. We survey related work in Section 6 before concluding in Section 7.

2 BACKGROUND

We briefly introduce CXL, configuration and allocation options for CXL memory, and memory access operations.

2.1 Compute Express Link

Compute Express Link (CXL) [37] is an open standard for interconnects between machines based on PCIe 5.0 and 6.0. CXL allows a machine to access a different machine's memory in a cache-coherent way. The specification differentiates between *hosts* and *devices*. A machine managing the cache coherence of attached memory locations is referred to as the *host*, while every other machine in the CXL topology is a *device* [5]. CXL includes the three protocols `CXL.io`, `CXL.cache`, and `CXL.mem`. `CXL.io` is the base protocol containing PCIe transactions. `CXL.cache` allows a device to access and cache data stored in host memory. `CXL.mem` allows a host to access and cache data stored in CXL device memory with load and store semantics. A key use case is memory expansion. A memory expansion device that supports `CXL.io` and `CXL.mem` is called a Type 3 device [11], which we use in our evaluation.

Three major revisions of the CXL standard exist [37]. Each higher revision introduces additional features. CXL 1.x specifies the three protocols, which serve as the foundation for memory expansion. CXL 2.0 supports resource pooling, which allows dynamic allocation and deallocation of the same resource to different hosts [37]. CXL 3.x adds the dynamic capacity feature of CXL memory devices [12]. This allows for dynamically changing memory capacity without resetting the device [12]. Another key feature of CXL 3.x is the support of hardware-based memory sharing across host boundaries [37]. The CXL device used in our evaluation supports CXL 1.1 connectivity. This means that memory accesses to the shared CXL device memory do not trigger coherence management of affected cache lines on the other server.

2.2 CXL Memory Configuration & Allocation

CXL device memory can be configured as a non-uniform memory access (NUMA) node in system-RAM mode or as a character device in device direct access (DAX) mode (*devdax*) [4, 20]. Configured as NUMA node, an application can use CXL device memory by first allocating an anonymous memory region (via the `mmap` system call) and binding the region's pages to the device's NUMA node (via the `mbind` system call). Configured as character device, an application first needs to get a file descriptor by opening the device. The application then creates a virtual memory mapping to the CXL device's existing physical memory via `mmap` with the file descriptor as parameter. Listing 1 shows a corresponding code example.

Configured as NUMA node, CXL device memory is zero-initialized [4]. For a shared scenario where multiple servers access the memory of the same CXL device, configuring the memory as NUMA node is not suitable [4]. In our evaluation, we configure the CXL device memory as character device.

Listing 1: Character device memory mapping example.

```
size_t region_size = 17'179'869'184; // 16 GiB
size_t offset = 0;
int file_descriptor = open("/dev/dax2.0", O_RDWR | O_SYNC);
void* address = mmap(nullptr, region_size, PROT_READ |
      ↪ PROT_WRITE, MAP_SHARED, file_descriptor, offset);
```

2.3 Memory Access Operations

Scalar load and store instructions are a common way to access memory, each accessing a single data element (e.g., 4 B or 8 B integers). Using vector instructions allows leveraging data-level parallelism by performing the same operation on multiple data elements [34]. Numerous database systems utilize vector intrinsics to increase query processing speed significantly [6, 31, 42, 43]. Vector loads and store instructions read and write to and from memory in vector-register-sized amounts of sequential data. On recent Intel CPUs, vector load and store instructions perform memory transactions on up to 512 bits of data (i.e., a complete cache line) using AVX-512 [21]. A streaming (also non-temporal) vector instruction bypasses the cache hierarchy of the CPU and directly interacts with the memory sub-system [13]. Streaming stores come with relaxed memory ordering rules, requiring explicit memory barriers by the programmer [13]. Streaming instructions are useful when an algorithm will not access the same data in the near future, avoiding cache pollution [35].

Besides reads and writes, atomic CPU hardware operations, such as compare-and-swap (CAS) and fetch-and-add (FAA), are essential primitives for building efficient, lock-free algorithms [24, 29, 36]. Such atomic operations modify data elements (e.g., integer values). If the CPU cache does not contain the element to be modified, the

CPU needs to fetch the corresponding cache line from memory. The memory access path and the resulting latency to fetch the required cache line influence the latency of an atomic operation. In our evaluation, we quantify the latency of atomic operations with the corresponding cache line stored in CXL memory.

3 BENCHMARK FRAMEWORK: CXL-BENCH

We introduce CXL-Bench, a highly configurable open-source framework for benchmarking access to heterogeneous memory tiers. The framework description in this section is an extended version of the description in our recent work [40]. CXL-Bench supports benchmarking memory access performance for any kind of memory configured as NUMA node or character device. This includes memory locally attached to a CPU via DDR, remote socket memory attached via an inter-socket interconnect and DDR, high-bandwidth memory, and device memory, e.g., attached via CXL or NVLink-C2C. For scenarios in which multiple servers are connected to the same shared memory, e.g., in a CXL shared memory setup, CXL-Bench supports distributed execution. This allows running multiple memory access workloads on multiple servers simultaneously.

3.1 Benchmark Features & Configuration

CXL-Bench supports measuring read and write operations, atomic operations, and the mapping and unmapping of memory regions. The framework provides a variety of parameters to configure the memory access operations to be analyzed. Table 1 shows an excerpt of the parameters and their options.

Reads and writes include scalar, vector, and streaming vector instructions. Cache instructions like cache line write back (`clwb`), flush cache line (`clflush`), and flush cache line optimized (`clflushopt`) can be specified for reads and writes. For reads, the execution of a cache instruction is used as a preparation of the cache line to be accessed. This preparation is not part of the latency measurement. For writes, the cache instruction is executed after the write to ensure that a modified cache line is written back to memory. The latency measurement includes the execution of the cache instruction.

Atomic operations include compare-and-swap and fetch-and-add. CXL-Bench further supports latency measurements for mapping and unmapping memory regions using the system calls `mmap` and `unmap`. This is especially relevant for shared CXL memory, where connected servers may dynamically allocate and release memory regions. The memory mapping benchmarks quantify the duration of allocating such memory regions. A user needs to specify the benchmark parameters in configuration (YAML) files as illustrated in Listing 2. Users can configure multiple options per parameter as a matrix parameter. For matrix parameters, CXL-Bench generates the Cartesian product and creates one benchmark configuration per combination. Users can run CXL-Bench with multiple configuration files. The framework derives benchmark configurations from each file and runs one benchmark for each generated benchmark configuration.

3.2 Benchmark Workflow

For each benchmark task, a number of threads perform memory accesses on a dedicated memory region based on the user-defined configuration. For each benchmark run, CXL-Bench prepares the memory regions to be accessed by the tasks, prepares the memory addresses to be accessed, performs the access operations, verifies memory page locations to ensure that no pages were moved during the access execution, and generates the results containing throughput or latency metrics. When preparing a benchmark task's memory region, CXL-Bench differentiates between NUMA node memory and character device memory.

Table 1: CXL-Bench configuration parameter excerpt.

Parameter	Options
Read & write operations	load, store (scalar, vector, and streaming vector)
Cache instructions	`clwb`, `clflush`, `clflushopt`
Atomic operations	compare-and-swap, fetch-and-add
Memory mapping ops	`mmap`, `unmap`
Access pattern	sequential, random
Random distribution	uniform, random
Access size	4 B to 64 KiB (powers of two)
OS memory abstraction	NUMA (system RAM), DAX device (`devdax`)
Memory NUMA nodes	list of NUMA node IDs
Device path	string
Memory region offset	integer ≥ 0
Data placement mode	interleaved (NUMA), partitioned (NUMA), linear (DAX device)
Memory region size	multiple of the access size
Thread pinning	thread to core mapping, thread to NUMA node mapping (one core per thread), thread to NUMA node mapping (multiple cores per thread)
Number of operations	integer > 0
Latency sampling interval	number of operations $>$ integer > 0
Run time [seconds]	integer > 0

Listing 2: Example configuartion YAML file.

```yaml
streaming_writes:
  matrix:
    number_threads: [ 1, 4, 16, 24, 48 ]
    exec_mode: [sequential, random]
  args:
    memory_region_size: 16G
    access_size: 64
    cache_instruction: none
    operation: stream-write
    numa_task_nodes: [ 1 ]
    device_path: "/dev/dax2.0"
    run_time: 10
    region_offset: 0
```

3.2.1 NUMA Memory Preparation. The memory preparation step for a benchmark task includes allocating virtual memory, binding virtual memory regions to user-defined NUMA nodes, and backing a region's pages by physical memory via pre-faulting. Pages can be allocated on any kind of memory that is configured as a NUMA node, including CXL device memory if configured as such.

CXL-Bench pins pages of a NUMA memory region either in a *interleaved* or a *partitioned* mode. The interleaved mode uses the entire memory region and pins its pages to the user-defined NUMA nodes via `mbind` in a round-robin fashion with Linux's interleaved allocation policy. The partitioned mode allows users to split the memory region into two partitions with different user-defined sizes relative to the region's total size. It then pins the partitions to different NUMA nodes. In both modes, pages of a memory (sub-)region are pinned to the corresponding NUMA nodes via the `mbind` system call with Linux's interleaved allocation policy.

The partitioned mode allows the evaluation of the memory access performance in a tiered memory scenario where the partitions of a memory region are located on different memory types, e.g., CPU-local and CXL device memory. If the corresponding task's memory operations include read operations, the memory region is filled with data in advance. CXL-Bench checks if the pages are located on the correct NUMA node at the end of the memory preparation.

3.2.2 Character Device Memory Preparation. When using memory configured as a character device, CXL-Bench first creates a file descriptor for the device. It then creates a memory mapping with that file descriptor as shown in Listing 1. A user-defined offset parameter determines the start position of the memory region in the device's address space. Configuring the offset allows users to run workloads simultaneously on disjoint memory regions of a shared CXL memory device.

3.2.3 Thread Pinning. CXL-Bench pins a thread to a set of cores using the GNU C library function `pthread_setaffinity_np`. Users can specify to which cores each task's thread pool is pinned. The user can either list the core identifiers or NUMA nodes. In the latter case, the cores associated with the NUMA nodes will be used for pinning the threads.

3.2.4 Latency Measurements. The framework differentiates between throughput and latency measurements. For latency measurements, a single thread performs the respective operation n times where n is a user-defined parameter (*number of operations* in Table 1). CXL-Bench measures the latency with a user-defined sampling interval, e.g., every 1 000th executed operation. Before each operation measurement, CXL-Bench performs a memory fence (`mfence`) to ensure that previous loads and stores are completed. Optional, configuration-dependent preparation steps are executed before measuring the actual operation. These include pre-loading a cache line when measuring only write latency or evicting a cache line to ensure that the target cache line is fetched from memory.

3.2.5 Throughput Measurements. CXL-Bench supports throughput measurements for reads and writes. It generates a user-defined number of worker threads executing the memory access operations. For this set of threads, CXL-Bench creates batches of access operations. Worker threads continuously fetch memory access batches from a shared queue and execute the corresponding memory access operations. This represents a common execution model where workers operate on small work packages [7]. By default, the total number of accessed bytes per batch is 64 MiB. Such batches are short-running and, thus, avoid the skew of large, long-running batches [7].

3.2.6 Page Verification & Result Generation. After executing the benchmark tasks, CXL-Bench verifies that the memory regions' pages are still located on the target NUMA nodes. Finally, CXL-Bench generates the overall benchmark results by writing all latency samples or collecting the measurements from all threads and calculating throughput values.

3.3 Distributed Execution

Users can execute CXL-Bench in a distributed manner. The distributed execution downloads CXL-Bench's source code, builds the binary, and runs the user-configured workload on user-configured servers. To do this, the user needs to configure parameters, such as the hostnames and login methods for the target servers, and the workload per server. The distributed execution ensures that benchmarks start at the same time. We use the distributed execution in our evaluation to run workloads on two servers, simultaneously accessing memory of a shared CXL device.

4 MEMORY ACCESS MICROBENCHMARKS

Load and store instructions are commonly limited to a single server. A CXL memory expansion device connected to multiple servers allows the CPU cores of all connected servers to perform loads and stores to shared CXL device memory simultaneously. In this section, we investigate the memory access performance to a CXL 1.1 memory device connected to two servers using CXL-Bench.

4.1 Setup

Hardware Setup. Table 2 details the two servers used for our experimental evaluation. Figure 2 shows the hardware setup, including the CPU memory population and the CXL device setup. Both CPUs support AVX-512 vector instructions, which are used in this work. The CXL memory device is an FPGA-based *Seagate Composable Memory Appliance (CMA) Blade* prototype [14, 30] with four DDR4 memory channels and two DIMMs per channel (DPC). Data stored on the device is interleaved across all eight DIMMs. The CMA supports the PCIe Gen5 x16 CXL 1.1 specification connectivity. One CMA has two ports. Each port can be connected to a separate server for a shared memory setup. We refer to a single blade as CXL device in the remainder of this work.

Table 2: Specifications of the evaluation servers.

Identifier	EMR	GNR
Server	Supermicro SYS-741GE-TNRT	Avenue City Platform
CPUs	2× Intel Xeon Gold 6542Y	1× Intel Xeon 6 Engineer' Sample
Cores	24 per CPU	96 per CPU
Caches	L1i: 32 KiB, L1d: 48 KiB L2: 2 MiB, L3: 60 MiB	L1i: 64 KiB, L1d: 48 KiB L2: 2 MiB, L3: 504 MiB
DDR5 DIMMs	8× 32 GB (1 DPC) with 4800 MT/s	12× 16 GB (1 DPC) with 5600 MT/s
OS	Ubuntu 24.04, Kernel 6.13.0	Ubuntu 24.04, Kernel 6.13.0

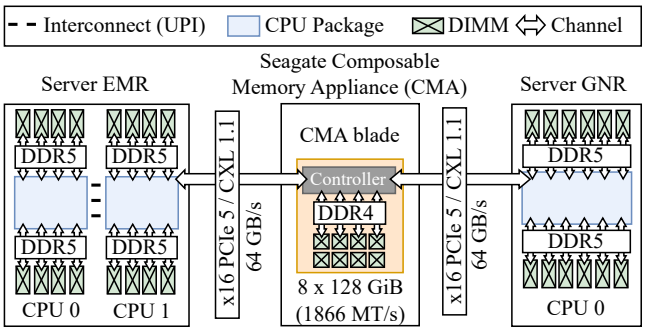

Figure 2: Hardware setup.

The CXL device is configured as character device in devdax mode. Its physical memory is mapped into the virtual memory address space using the `mmap` system call, as shown in Listing 1. In this example code, the CXL device path is `/dev/dax2.0`.

Workload Execution. In the experiments, we differentiate between the main workload and the background workload. The main workload performs memory accesses to the CXL device, for which we quantify either the latency or bandwidth, depending on the experiment. A separate, simultaneous background workload generates load on the CXL device's memory. This allows us to study how an additional workload accessing the shared CXL device affects the other workload's performance. The main workload runs on server EMR while the background workload runs on GNR. The background workload continuously performs sequential vector reads to a memory region of the CXL device. If not mentioned otherwise, the memory region of both the main and background workload has a size of 16 GiB and starts at offset 0 of the device's memory address space.

4.2 Latency

Memory access latency is a key performance metric for transaction processing [26], as well as, join and aggregate operations [32]. We investigate the memory access latency with and without running the background workload.

Setup. We measure the latency of 64 B (temporal) vector and streaming (i.e., non-temporal) vector loads and stores with a uniform random access pattern. We measure loads followed by a memory fence (`mfence`). For vector stores, we first load the target cache line into the cache. We then measure the latency of a store instruction followed by a cache line write back (`clwb`). The `clwb` ensures that the corresponding cache line is written back to memory. For streaming stores, we ensure that the cache line is written to memory with a memory fence after the streaming store. We measure the latency of accesses to the CXL device memory on the EMR server. We measure the latency with and without a background workload running on GNR, generating load on the CXL device. We perform each operation 10 M times and sample the latency of every 1 000th operation, resulting in 10 000 latency measurements per operation.

Results. Figure 3 shows the results. For individual cache line loads and stores, the latency behaves very similarly for vector instructions and streaming vector instructions. Without background load, the median latencies for vector loads and streaming vector loads are 480 ns and 490 ns. The 99th percentile latencies are about 75% to 80% higher with 860 ns. With background load, the median latencies increase to 820 ns and 830 ns. The 99th percentile latencies are between 50% and 55% higher with 1270 ns and 1260 ns. Compared to the latencies without background load, the median latencies increase by about 70%.

For vector stores and vector streaming stores, the corresponding latencies with and without background load are almost identical. Vector stores show median latencies of 390 ns with and without load. 99th percentile latencies are between 33% and 36% higher (i.e., between 520 ns and 530 ns) with load. Streaming vector stores show median latencies of 360 ns with and without load. 99th percentile latencies are between 500 ns and 520 ns.

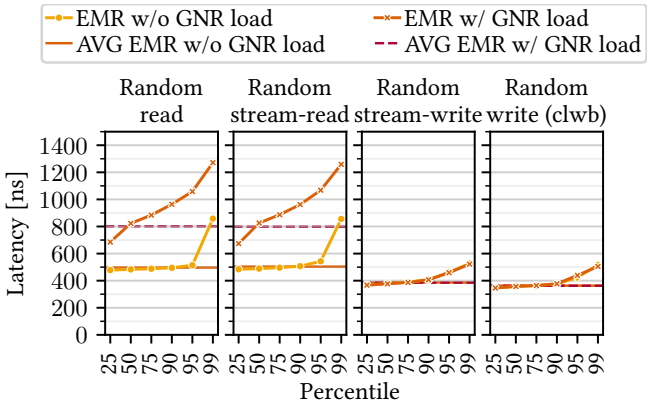

Figure 3: Random load and store latency on EMR with and without load generated by GNR.

Running the same experiment with a larger memory region of 64 GiB yields similar results without significant differences. Changing the start offset of the background workload's memory region (to 512 GiB) so that the regions of the main and background workload are disjoint yields the same latencies.

Discussion. The measurements show that memory access of a separate server to the used shared CXL device significantly affect memory access latency. Assigning disjoint memory regions of the device to workloads on separate compute nodes still shows the same performance impact. This is the case as, independent of a memory region's position in the device's memory address space, data stored on the CXL device is interleaved across all DIMMs. When using the device for data management, simply monitoring and balancing memory accesses of a single server is insufficient for optimizing memory access performance. A memory-centric database system [10] could consist of multiple compute nodes connected to the shared CXL device. One should be aware of the potential load that all connected servers can generate on the device's CXL and memory controllers. With the current hardware trend of an increasing number of cores per CPU package [9], hundreds of CPU cores can be connected to a shared CXL memory pool (cf. [8]). The actual number depends on the number of ports of a CXL device and the total number of cores across the servers. To avoid oversubscription of CXL and memory controllers, one needs to carefully distribute data across the different memory tiers accessible to a CPU.

4.3 Throughput

Memory throughput is another key performance metric when utilizing memory. It is relevant for bandwidth-bound database operations, such as selections, projections, and ungrouped aggregations [32]. We quantify the memory throughput achievable with vector and streaming vector load and store instructions, with and without running the background load.

Setup. We measure the throughput with 64 B vector and streaming vector loads and stores with different numbers of threads. In the latency experiment, we execute memory fences and explicit cache line writebacks as we measure the latency of individual accesses. In this experiment, we focus on the maximum throughput. We do

not execute memory fences and cache instructions (e.g., `clwb`) as they incur additional overhead and prevent instruction reordering, both of which can be disadvantageous for maximising memory throughput. We perform the respective operation for 10 seconds and report the throughput.

Results. Figure 4 shows the results. Loads with sequential accesses achieve higher throughput than loads with random accesses. While vector loads and streaming loads do not show significant differences for sequential access, regular vector loads perform better for random access than streaming vector loads. With sequential read load on the CXL device generated by the GNR server, the achieved throughput is reduced by approximately 50%.

The latency study shows no differences for stores with and without additional load on the device. Without memory fences and cache instructions in this experiment, the throughput achieved without load is significantly higher compared to the throughput with additional load. The throughput differs significantly between vector stores and streaming vector stores for both sequential and random accesses. Vector stores achieve a maximum of 16.5 GB/s. The throughput of streaming stores is by 2.4× higher, reaching a plateau of approximately 40 GB/s. One possible reason for the higher throughput is reduced cache coherence management effort: Temporal stores write to a cache line in exclusive cache line state. If a store causes a store miss, the core issues a read-for-ownership (RFO) transaction to fetch the target cache line in exclusive state. This is an additional load to the target memory. Streaming stores write the data to a write combining buffer, where writes may be delayed and combined [21]. This avoids the additional RFO loads, reducing additional load on the controller so that more write requests can be processed faster. Similar to the latency experiment, running the experiment with either a larger memory region of 64 GiB or the background workload accessing a disjoint memory region (with an offset of 512 GiB) shows similar results.

Discussion. The results confirm the findings of our latency study: loads to the shared CXL memory device by one server significantly impact the memory access performance of another server to the device. The results further show that streaming vector stores can achieve a significantly higher throughput than regular vector stores. In our experiment, the throughput with load on the system for streaming stores approximately matches the throughput of vector stores without load. Database operations involving writes, where the written data is not promptly accessed again, significantly benefit from streaming vector stores over regular vector stores.

As CXL 1.x/2.0 does not support hardware-based memory sharing across multiple hosts [37], using regular vector stores only affects the cache coherence domain of the server that issued the memory access. When writing large amounts of data to CXL 1.x/2.0 shared memory, streaming stores are better suited due to their higher throughput.

4.4 Atomic Operations

Atomic operations are essential for building concurrent algorithms and data structures. We quantify the latency of atomic operations with values to be stored in CXL shared memory, with and without load generated by the background workload.

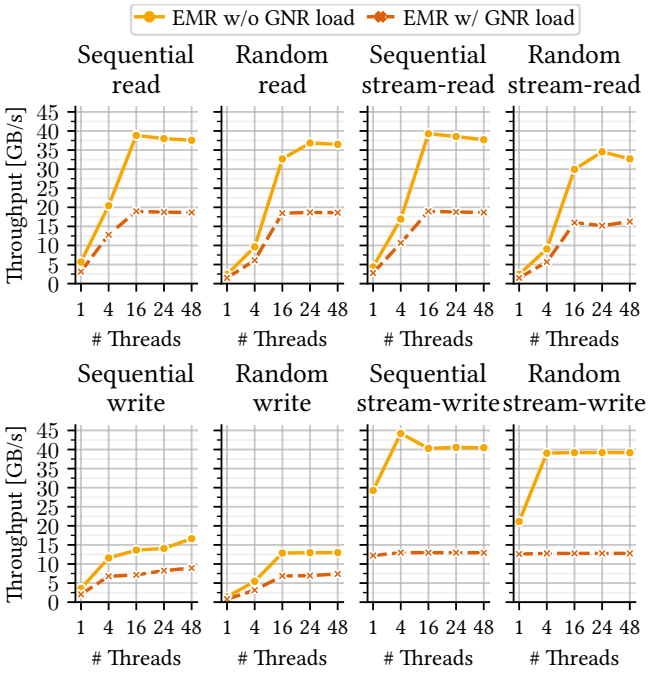

Figure 4: Load and store throughput on EMR with and without load generated by GNR.

Setup. We measure the latency of CAS and FAA atomic operations with the value to be modified stored in CXL shared memory. We first write a 64-bit unsigned integer to each cache line of the CXL memory region. During the benchmark workload, we randomly pick one of these 64-bit unsigned integers and perform the respective atomic operation (using `atomic_ref`'s `compare_exchange_weak` and `fetch_add` functions of the `C++` standard library). We perform each operation 10 M times and sample the latency of every 1 000th operation, resulting in 10 000 latency measurements per operation.

Results. Figure 5 shows the results. Similar to the load latencies in Section 4.2, the latency is significantly affected by additional background load. The latency for both operations is between 500 ns

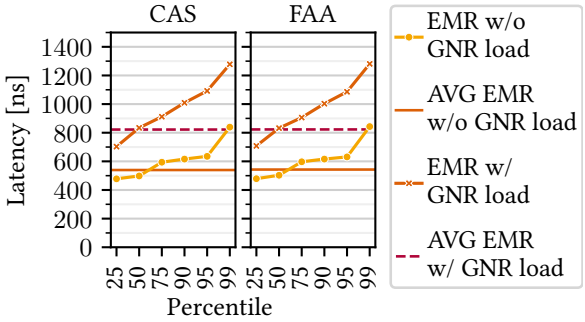

Figure 5: Compare-and-swap (CAS) and fetch-and-add (FAA) atomic operation latency on EMR with and without load generated by GNR.

and 550 ns without load and between 800 ns and 850 ns with load. These numbers are similar but slightly higher than the load latencies in Section 4.2. In a variation of this experiment, we first load the cache line in which the target value is placed into the cache and then perform the respective atomic operation. This experiment shows a 90th percentile latency of under 50 ns for both operations.

Discussion. When performing atomic operations on values that are not cached, the atomic operation is dominated by the memory access latency. Atomic operations on values stored in CXL 1.x/2.0 shared memory should be used carefully, as an atomic operation neither ensures an actual write back of the affected value to CXL memory, nor does it trigger coherence management of the affected cache line on the other servers connected to the same device.

5 MAPPING SHARED CXL DEVICE MEMORY

With shared CXL memory, a large memory pool can be split into smaller memory regions, each accessible by one or more attached servers [25]. Before an application running on an attached server can access a CXL memory region, it needs to map the CXL device's memory into the application's virtual memory address space. In this section, we study the latency of mapping CXL memory regions with different sizes into an application's virtual address space.

Setup. We measure the latency of mapping and unmapping memory regions using the `mmap` system call. We vary the latency for memory regions with power-of-two sizes, ranging from 16 GiB to 128 GiB. With a character device, we can only define a memory-mapped region as shared. For comparison, we measure the latency of mapping anonymous shared and private memory regions. We prefault all pages of the anonymous memory regions (using the `mmap` option `MAP_POPULATE`). We measure each mapping five times and report the average latency.

Results. Figure 6 shows the latency of mapping and unmapping memory regions on the EMR server. Mapping and unmapping character device memory has a significantly lower latency than mapping and unmapping anonymous memory regions. For all options, the latency increases linearly with the region size. For 128 GiB, mapping the character device takes only one second while allocating anonymous memory regions takes 40 seconds and 26 seconds in shared and private mode. Unmapping the memory region happens promptly for the character device in less than a tenth of a second. In contrast, unmapping the shared and private anonymous regions requires eight and three seconds. For anonymous regions, we see that memory regions declared as private exhibit lower latency for mapping and unmapping than regions declared as shared.

Discussion. Memory mapping and unmapping on CXL memory configured as a character device is fast compared to anonymous memory region allocations. It does not involve expensive heap allocations by the kernel as with allocating anonymous memory regions. The prompt completion of character device mappings and unmappings indicates that frequent re-mapping is feasible for data processing systems. Unmapping and mapping the character device do not clear the physical pages. Most recently written data still exists in the pages. On the one hand, this allows simple resizing of memory regions: In case a previously allocated memory region is too small, it can simply be unmapped and mapped with a larger region size, starting at the same offset. The data from the previous region can still be used. On the other hand, if an application requires an erased (e.g., zeroed) memory region, the application needs to perform manual erasure. This can be costly as erasing each page requires many memory accesses and their associated latency.

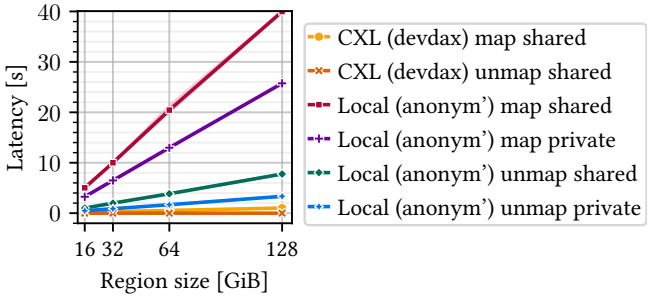

Figure 6: Memory map and unmap latency.

6 RELATED WORK

Our proposed benchmark framework is highly inspired by *PerMA-Bench*, a benchmark framework for persistent memory access [7]. CXL-Bench significantly extends PerMA-Bench by supporting memory configured as NUMA nodes, several thread pinning modes, memory allocation policies, latency measurements of individual operations, and additional operations, e.g., atomic operations and memory mapping and unmapping. Especially the NUMA configuration adds support for several types of memory, such as memory of local and remote CPU sockets, individual CPU dies, and memory of peripheral devices (if configured as NUMA nodes). While PerMA-Bench runs on a single server, CXL-Bench supports a distributed mode. This allows benchmarking memory access scenarios in which separate servers access a shared memory device.

In the remainder of this section, we survey existing evaluation studies on CXL memory access performance. Ahn et al. [1] and Lee et al. [23] evaluate the performance impact of placing different parts of data of the SAP HANA database system on FPGA-based CXL memory using the TPC-C, TPC-DS, and TPC-H benchmarks. Sun et al. [38] evaluated performance on FPGA-based and two ASIC-based CXL memory devices using Intel Sapphire Rapids CPUs. The authors evaluate latency and throughput with microbenchmarks and application workloads. While the microbenchmarks are executed on all devices, the application workloads are performed only on the best-performing CXL memory device. The authors show that CXL memory can expand a system's total memory bandwidth when a deep-learning recommendation model utilizes only two memory DIMMs and one CXL memory compared to only using the two memory DIMMs. Gouk et al. [18] evaluate performance characteristics for customized FPGA-based CXL 2.0 memory cards. The authors built and utilized a custom RISC-V ISA CPU with CXL support and performed random memory requests with different access sizes and several real-world workloads. Geyer et al. [17] experimentally evaluate the access performance on local memory and remote memory attached via either Ultra Path Interconnect (UPI), CXL, or remote direct memory access (RDMA) on a system with Intel Sapphire Rapids CPUs. They evaluate the access performance with aggregate, filter, and hash join database operations. Fridman

et al. [16] evaluate CXL's potential to provide persistent memory to a server system on an FPGA-based CXL device. They perform an experimental evaluation with an FPGA-based CXL device using the STREAM benchmark and provide absolute throughput numbers. Tang et al. [39] experimentally evaluate the access performance of an ASIC-based CXL memory expansion device with Intel Sapphire Rapids CPUs for basic and application workloads, including Redis. Kim et al. [22] present performance metrics for Samsung's CXL memory expander *CXL MXP*. Besides basic memory access patterns, they evaluate the performance of applications workloads, including Memcached and Redis. Liu et al. [28] perform an experimental evaluation of basic access patterns as well as high-performance computing (HPC) workloads. The authors find that HPC workloads may not use additional theoretical bandwidth with CXL memory.

7 CONCLUSION

We introduce CXL-Bench, a benchmark framework for benchmarking heterogeneous memory access paths. We demonstrate CXL-Bench by quantifying memory access characteristics of two servers accessing a shared CXL 1.1 memory device. For the used CXL device, we show that a workload running on one server accessing the device significantly reduces the performance of another workload running on a different server. We further show that the latency of atomic operations is mainly determined by memory access latency. In terms of memory management, we show that memory re-allocations with CXL memory configured as a character device are fast, which make frequent reallocation of CXL memory feasible. Memory access characteristics influence the quality of memory integration concepts for database systems. We encourage database researchers and practitioners to use CXL-Bench to quantify memory access characteristics for upcoming CXL memory devices.

ACKNOWLEDGMENTS

We thank Seagate Technology LLC for their support and the anonymous reviewers for their feedback. This work was partially funded by SAP, the German Research Foundation (ref. 414984028), the European Union's Horizon 2020 research and innovation programme (ref. 957407), and the Independent Research Fund Denmark's Inge Lehmann program (grant agreement number 0171-00062B).

REFERENCES

[1] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*. 1–5.

[2] Minseon Ahn, Thomas Willhalm, Norman May, Donghun Lee, Suprasad Mutalik Desai, Daniel Booss, Jungmin Kim, Navneet Singh, Daniel Ritter, and Oliver Rebholz. 2024. An Examination of CXL Memory Use Cases for In-Memory Database Management Systems using SAP HANA. *Proceedings of the VLDB Endowment (PVLDB)* 17, 12 (2024), 3827–3840.

[3] Anastasia Ailamaki, Samuel Madden, Daniel Abadi, Gustavo Alonso, Sihem Amer-Yahia, Magdalena Balazinska, Philip A. Bernstein, Peter A. Boncz, Michael J. Cafarella, Surajit Chaudhuri, Susan B. Davidson, David J. DeWitt, Yanlei Diao, Xin Luna Dong, Michael J. Franklin, Juliana Freire, Johannes Gehrke, Alon Y. Halevy, Joseph M. Hellerstein, Mark D. Hill, Stratos Idreos, Yannis E. Ioannidis, Christoph Koch, Donald Kossmann, Tim Kraska, Arun Kumar, Guoliang Li, Volker Markl, Renée J. Miller, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Aditya G. Parameswaran, Ippokratis Pandis, Jignesh M. Patel, Andrew Pavlo, Danica Porobic, Viktor Sanca, Michael Stonebraker, Julia Stoyanovich, Dan Suciu, Wang-Chiew Tan, Shivaram Venkataraman, Matei Zaharia, and Stanley B. Zdonik. 2025. The Cambridge Report on Database Research. *CoRR* abs/2504.11259 (2025).

[4] Ramesh Aravind and Groves John. 2024. Study of CXL Memory Sharing with FamFS and its Use cases. In *Proceedings of the International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*. 73–77.

[5] Gal Assa, Michal Friedman, and Ori Lahav. 2024. A Programming Model for Disaggregated Memory over CXL. *CoRR* abs/2407.16300 (2024).

[6] Lawrence Benson, Richard Ebeling, and Tilmann Rabl. 2023. Evaluating SIMD Compiler-Intrinsics for Database Systems. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*.

[7] Lawrence Benson, Leon Papke, and Tilmann Rabl. 2022. PerMA-bench: benchmarking persistent memory access. *Proceedings of the VLDB Endowment (PVLDB)* 15, 11 (2022), 2463–2476.

[8] Daniel S. Berger, Yuhong Zhong, Pantea Zardoshti, Shuwei Teng, Fiodar Kazhamiaka, and Rodrigo Fonseca. 2025. Octopus: Scalable Low-Cost CXL Memory Pooling. *CoRR* abs/2501.09020 (2025).

[9] Thomas Bodner, Martin Boissier, Tilmann Rabl, Ricardo Salazar-Díaz, Florian Schmeller, Nils Strassenburg, Ilin Tolovski, Marcel Weisgut, and Wang Yue. 2025. A Case for Ecological Efficiency in Database Server Lifecycles. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[10] Yannis Chronis, Anastasia Ailamaki, Lawrence Benson, Helena Caminal, Jana Gičeva, Dave Patterson, Eric Sedlar, and Lisa Wu Wills. 2025. Databases in the Era of Memory-Centric Computing. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[11] CXL Consortium. 2023. Compute Express Link Specification - Revision 3.1.

[12] CXL Consortium. 2024. Compute Express Link Specification - Revision 3.2.

[13] Ulrich Drepper. 2007. *What every programmer should know about memory*. Technical Report. Red Hat, Inc.

[14] Mohamad El-Batal and Hongjian Fan. 2024. Seagate Composable Memory Appliance (Presentation at Open Compute Project). https://www.youtube.com/watch?v=RjKP1mg7bu8&t=1660s Last access: 2025-07-08.

[15] Alessandro Fogli, Bo Zhao, Peter R. Pietzuch, Maximilian Bandle, and Jana Giceva. 2024. OLAP on Modern Chiplet-Based Processors. *Proceedings of the VLDB Endowment (PVLDB)* 17, 11 (2024), 3428–3441.

[16] Yehonatan Fridman, Suprasad Mutalik Desai, Navneet Singh, Thomas Willhalm, and Gal Oren. 2023. CXL Memory as Persistent Memory for Disaggregated HPC: A Practical Approach. In *Proceedings of the Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W)*. 983–994.

[17] Andreas Geyer, Johannes Pietrzyk, Alexander Krause, Dirk Habich, Wolfgang Lehner, Christian Färber, and Thomas Willhalm. 2023. Near to Far: An Evaluation of Disaggregated Memory for In-Memory Data Processing. In *Proceedings of the Workshop on Disruptive Memory Systems (DIMES)*. 16–22.

[18] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 287–294.

[19] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. 2025. Pasha: An Efficient, Scalable Database Architecture for CXL Pods. *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[20] Intel Corporation. 2024. CXL* Type 3 Memory Device Software Guide. Revision 1.1.

[21] Intel Corporation. 2025. Intel® 64 and IA-32 Architectures Software Developer's Manual. Order Number: 325462-087US. March 2025.

[22] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. 2023. SMT: Software-Defined Memory Tiering for Heterogeneous Computing Systems With CXL Memory Expander. *IEEE Micro* 43, 2 (2023), 20–29.

[23] Donghun Lee, Thomas Willhalm, Minseon Ahn, Suprasad Mutalik Desai, Daniel Booss, Navneet Singh, Daniel Ritter, Jungmin Kim, and Oliver Rebholz. 2023. Elastic Use of Far Memory for In-Memory Database Management Systems. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*. 35–43.

[24] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proceedings of the VLDB Endowment (PVLDB)* 17, 12 (2024), 4536–4545.

[25] Alberto Lerner and Gustavo Alonso. 2024. CXL and the Return of Scale-Up Database Engines. *Proceedings of the VLDB Endowment (PVLDB)* 17, 10 (2024), 2568–2575.

[26] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[27] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. 2025. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1203–1217.

[28] Jie Liu, Xi Wang, Jianbo Wu, Shuangyan Yang, Jie Ren, Bhanu Shankar, and Dong Li. 2024. Exploring and Evaluating Real-world CXL: Use Cases and System

Adoption. *CoRR* abs/2405.14209 (2024).

[29] Darko Makreshanski, Justin J. Levandoski, and Ryan Stutsman. 2015. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing. *Proceedings of the VLDB Endowment (PVLDB)* 8, 11 (2015), 1298–1309.

[30] Seagate Technology Team Member(s). 2024. Composable Memory Appliance (CMA) Base Specification V1.1. Open Compute Project.

[31] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1493–1508.

[32] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2023. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Computing Surveys (CSUR)* 55, 2 (2023), 11:1–11:38.

[33] Viktor Sanca and Anastasia Ailamaki. 2023. Post-Moore's Law Fusion: High-Bandwidth Memory, Accelerators, and Native Half-Precision Processing for CPU-Local Analytics. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*.

[34] Lennart Schmidt, Johannes Pietrzyk, Juliana Hildebrandt, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2025. Rethinking MIMD-SIMD Interplay for Analytical Query Processing in In-Memory Database Engines. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[35] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. 2015. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning. *Proceedings of the VLDB Endowment (PVLDB)* 8, 9 (2015), 934–937.

[36] Ori Shalev and Nir Shavit. 2006. Split-ordered lists: Lock-free extensible hash tables. *J. ACM* 53, 3 (2006), 379–405.

[37] Debendra Das Sharma, Robert Blankenship, and Daniel S. Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Computing Surveys (CSUR)* 56, 11 (2024), 290:1–290:37.

[38] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 105–121.

[39] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 818–833.

[40] Marcel Weisgut, Daniel Ritter, Pınar Tözün, Lawrence Benson, and Tilmann Rabl. 2025. CXL Memory Performance for In-Memory Data Processing. *Proceedings of the VLDB Endowment* 18, 9 (2025), 3119–3133.

[41] Felix Werner, Marcel Weisgut, and Tilmann Rabl. 2025. Towards Memory Disaggregation via NVLink C2C: Benchmarking CPU-Requested GPU Memory Access. In *Proceedings of the Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS)*. 8–14.

[42] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *Proceedings of the VLDB Endowment (PVLDB)* 2, 1 (2009), 385–394.

[43] Jingren Zhou and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 145–156.