Hardware-Software Co-Development for Emerging CXL Architectures

Roberto Gioiosa
Pacific Northwest National Laboratory
Richland, WA
roberto.gioiosa@pnnl.gov

Bo Fang
Pacific Northwest National Laboratory
Richland, WA
bo.fang@pnnl.gov

Lenny Guo
Pacific Northwest National Laboratory
Richland, WA
lenny.guo@pnnl.gov

Andres Marquez
Pacific Northwest National Laboratory
Richland, WA
lenny.guo@pnnl.gov

ABSTRACT

Modern scientific, Artificial Intelligence (AI), and graph analytics workloads demand substantial memory resources to accommodate their extensive datasets. While distributed systems connected via high-performance networks have traditionally been employed to address such challenges, Compute eXpress Link (CXL) technology is emerging as a compelling alternative. CXL systems offer a shared memory abstraction over physically disaggregated memory with load/store programming semantics, simplifying the development of applications that require large memory pools.

However, as CXL hardware is still under development, its internal mechanisms and the performance implications for critical applications remain largely unexplored. To address this gap, we propose a hardware-software co-development framework for future CXL systems. Our approach combines a CXL-enabled full-system emulator with a memory allocator (MemForge) backed by CXL memory devices and a high-level set of Application Programming Interface (API)s.

We demonstrate that our methodology supports the development of essential kernels across High-Performance Computing (HPC), AI, and graph analytics domains. Experimental results obtained from two CXL hardware configurations — direct-attached memory and memory accessed via a CXL switch — indicate minimal runtime overhead. Additionally, we highlight the internal introspective capabilities of our memory allocator, which facilitate profiling and debugging.

KEYWORDS

Memory, CXL, Emulation, Rust

1 INTRODUCTION

Modern workloads are becoming increasingly complex and demand vast computing and memory resources. While workloads across domains may exhibit different computation and memory access patterns, they also share common characteristics. In particular, there is a growing need for larger memory pools that can be accessed via load/store semantics within a shared-memory programming paradigm. This trend spans domains from High-Performance Computing (HPC) and scientific simulation to Artificial Intelligence (AI) and High-Performance Data Analytics (HPDA), encompassing scientific, commercial, and security applications.

In this context, Compute eXpress Link (CXL) is emerging as a promising solution to interconnect compute elements and large memory pools through fabric-attached memory [21]. CXL is an open interconnect standard designed to improve the performance and efficiency of disaggregated memory systems by enabling high-speed communication among Central Processing Unit (CPU)s, memory devices, accelerators (e.g., Graphics Processing Unit (GPU)s, Field-Programmable Gate Array (FPGA)s), and other peripherals. Based on the Peripheral Component Interconnect Express (PCIe) protocol, CXL offers lower latency and enhanced support for memory coherence, which is essential in modern computing environments. The CXL specification comprises three protocol layers: *cxl.io* manages basic discovery, configuration, and device management; *cxl.mem* allows the host CPU to directly access device-attached memory (e.g., expansion modules); and *cxl.cache* enables accelerators to cache host memory and maintain coherence. Additionally, some vendors refer to *cxl.accel* as a shorthand for *cxl.io* and *cxl.cache* (Type 1 accelerator device) and *cxl.io*, *cxl.cache*, and *cxl.mem* (Type 2 accelerator device). The protocol's flexibility and extensibility enable disaggregated memory to be presented to users as a unified, shared memory space, thereby simplifying programming and system management.

As with any emerging technology, the design and implementation of CXL devices, switches, and interconnects are still evolving [10]. Although prototypes of CXL devices and switches exist, they often support only a subset of the full protocol (typically *cxl.io* and *cxl.mem*), and the accompanying software stack (including firmware and system software) is limited primarily to testing and validation. This complicates the development of applications in anticipation of fully-featured CXL systems. This scenario is common in the lifecycle of emerging technologies: by the time hardware is ready for deployment, the corresponding software is often still under development. Consequently, software readiness delays system deployment. From a technical standpoint, this is problematic because it prevents valuable feedback from being incorporated into the hardware design, which is often finalized by that stage. Although enhancements may be deferred to future hardware revisions, this leads to a cycle in which software and hardware perpetually attempt to catch up with each other.

We argue that a more effective strategy is the co-development of hardware and software. This approach not only reduces time-to-deployment, as both hardware and software can mature concurrently, but also enables co-design opportunities. In particular, it allows software developers to provide real-time feedback that can inform hardware design choices, potentially improving overall system efficiency. However, tools to support hardware-software co-development are typically limited in scope and suitability, often restricted to simple applications or microbenchmarks. To address this gap, we introduce a co-development framework tailored for future CXL systems. Our methodology integrates a CXL full-system emulator with well-defined interfaces, a CXL-aware memory allocator implemented in Rust, and a set of Application Programming Interface (API)s for CXL system programming. This framework enables application development that leverages device-attached memory (both directly attached and accessed via CXL switches) while hardware is still under design. Consequently, applications developed with our framework are immediately ready to utilize CXL hardware once it becomes available. In essence, the full-system emulator abstracts low-level hardware components, while the memory allocator abstracts CXL protocols for end users, allowing them to reason using familiar data structures such as lists, vectors, and matrices. Furthermore, the allocator provides introspective telemetry useful for debugging and performance tuning.

While current CXL benchmarks (e.g., STREAM) are relatively simplistic, we demonstrate that our combined solution is capable of running representative HPC, AI, and HPDA kernels with minimal runtime overhead compared to conventional allocators. We also present execution traces from our allocator that illustrate memory fragmentation mitigation and caching strategies. Our evaluation includes comparisons across configurations involving device-attached memory and CXL switches, and benchmarks the ease of use relative to traditional DRAM-based allocators.

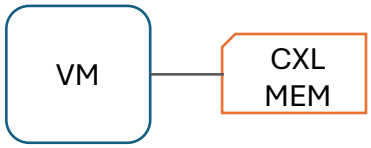
This paper makes the following contributions:

- We introduce a framework for hardware-software co-development in the context of future CXL systems.
- We present **MemForge**, a Rust-based memory allocator for CXL-attached memory devices.
- We implement and evaluate key kernels from HPC, AI, and graph analytics domains, demonstrating how MemForge simplifies the use of CXL memory.

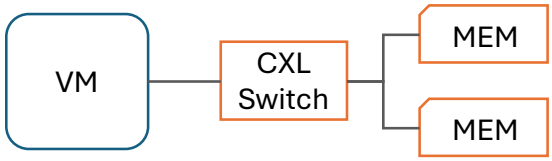
The remainder of this paper is structured as follows: Section 2 presents the CXL hardware emulator. Section 3 describes the MemForge allocator. Section 4 outlines the benchmark suite, experimental setup, and results. Section 5 discusses related work. Finally, Section 6 concludes and outlines directions for future research.

2 EMULATION SYSTEM

Our proposed emulation methodology uses QEMU [3, 19] as its core building block. QEMU provides a universal interface for running a wide range of guest operating systems on a host machine through dynamic binary translation, converting guest CPU instructions into host-compatible CPU instructions. Specifically, QEMU's support for CXL builds upon its well-established PCIe emulation infrastructure. Because CXL leverages the PCIe physical layer, QEMU extends this capability to model key components of a CXL-enabled system. This facilitates the creation of complex virtual topologies that mimic real-world hardware configurations.



(a) Memory device directly attached to host through CXL link.



(b) Two memory devices attached to the host through a CXL switch.

Figure 1: QEMU can be configured to emulate CXL memory devices in several ways. In this work, we emulate direct-attached memory devices (1a) and two CXL memory devices attached to a CXL switch (1b).

QEMU can emulate a range of CXL hardware elements, enabling the construction of complete virtualized CXL environments, including:

- **CXL Host Bridges**: These serve as the connection point between the host CPU and the CXL fabric. QEMU emulates the logic required to manage and route traffic to and from CXL devices.
- **CXL Root Ports**: Analogous to PCIe root ports, these virtual interfaces on the host bridge link to CXL devices or switches.
- **CXL Switches**: These virtual switches support complex topologies and resource pooling by connecting multiple CXL devices, allowing developers to test switching and fabric management features.
- **CXL Memory Devices (Type 3)**: These devices allow experimentation with volatile or persistent memory added to a virtual machine—a primary use case for CXL. QEMU also supports CXL Type 1 and Type 2 devices.

QEMU allows developers to build and test CXL-aware drivers, system software, and applications without needing physical CXL hardware, which is often scarce and costly. By offering a robust and accessible virtual platform, QEMU enables complete development, testing, and validation of CXL-aware software in parallel with hardware development.

In this work, we emulate two configurations (Figure 1): the first attaches a memory expander module directly to the host via a CXL link (Figure 1a), while the second connects two memory expanders to the host through a CXL switch (Figure 1b).

3 MEMFORGE

Our memory allocator, MemForge, is developed in Rust—a modern programming language that emphasizes performance, safety, and concurrency. The choice of Rust is deliberate: at scale (e.g., hundreds

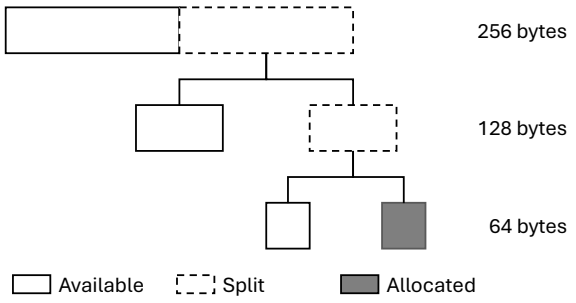


Figure 2: MemForge's buddy system algorithm recursively split larger blocks of continuous memory to satisfy memory requests. In this case, the original 256-byte memory block is split first in two 128-byte blocks, then of the block is split into two 64-byte blocks. One 64-byte block is allocated to store the requested memory object.

of thousands of compute nodes connected to terabytes of disaggregated memory), achieving hardware-enforced consistency is challenging. While some degree of localized hardware consistency may be feasible, system-wide consistency will largely rely on software. Software-level consistency, however, is notoriously difficult to enforce and often introduces runtime overhead. By leveraging Rust's ownership model and compile-time borrow checker, MemForge shifts a portion of these consistency checks from runtime to compile time, reducing associated overhead.

From an allocation perspective, MemForge implements the buddy system algorithm [12]. This approach is well-suited for managing contiguous blocks of memory while minimizing fragmentation. MemForge maintains large contiguous blocks to satisfy substantial memory requests and recursively splits them into smaller chunks for smaller allocations. Figure 2 illustrates this process: an initial 256-byte block is recursively split to fulfill a 64-byte request. Upon memory deallocation, MemForge attempts to merge adjacent free blocks to recreate larger blocks, improving future allocation efficiency. Internally, MemForge maintains a hierarchical list of free blocks indexed by size and starting address. This list is protected by a mutex, enabling safe use in multi-threaded applications.

MemForge supports multiple memory backing strategies. It can map a regular file into memory, which is useful for development on non-CXL systems such as laptops. Alternatively, it can use a device file that represents a CXL-attached memory module. Applications do not need to change between environments: the memory backing source is controlled via the environment variable `MEMFORGE_PATH`.

MemForge implements the `allocate()` and `deallocate()` methods from Rust's `Allocator` trait, allowing it to be used with standard library data structures such as vectors. Given an application already developed, switching to MemForge requires only replacing `vec::with_capacity()` with its internal version `vec::with_capacity_in()`, which takes an allocator reference as an additional argument.[1]

Tracing and logging in MemForge are user-configurable and do not require application or allocator recompilation. Logging is built on Rust's standard `log` subsystem [7] and can be enabled via the `RUST_LOG` environment variable (e.g., `info`, `debug`, `trace`). Instrumentation is provided using the `tracing` crate [5]. Key functions such as `allocate()` and `deallocate()`, as well as significant runtime events (e.g., block splits and merges), are instrumented for observability. Trace data can be directed either to a default subscriber (console) or to the OpenTelemetry subscriber provided by the `tracing-opentelemetry` crate [6]. The latter enables trace collection in multi-threaded and distributed environments, with visualization support in tools such as Perfetto and Chrome Tracing.

Finally, MemForge is simple to integrate into Rust applications. Thanks to the `Cargo` build system, importing MemForge automatically resolves all dependencies with proper versioning.

[1] `vec::with_capacity(size)` is essentially equivalent to `vec::with_capacity_in(size, Global)`

4 EVALUATION

This Section describes our benchmarks, experimental setup, and experimental results.

4.1 Benchmarks

GEMM. GEMM (General Matrix Multiply) is a fundamental operation in linear algebra and widely used across domains such as machine learning and scientific computing. It computes the product of two matrices, $A(N, K)$ and $B(K, M)$, yielding a third matrix $C(N, M)$. The operation follows the expression $C = \alpha AB + \beta C$, where α and β are scalars. In our experiments, we set $\beta = 0$. Given the significance of GEMM operations, they are heavily optimized across various hardware platforms, including CPU, GPU, and FPGA. Our implementation leverages the `rayon` crate [17] to parallelize the outer loop using a parallel iterator. While more efficient versions are feasible, our implementation prioritizes simplicity and clarity, making it easier to explain. Listing 1 presents our main kernel implementation.

```
fn matrix_multiply(a: &[Vec<f32, &MemForge>], b: &[Vec<f32, &MemForge>]) ->
↪  Vec<Vec> {
let n = a.len();
let m = b[0].len();
let k = b.len();

(0..n).into_par_iter()
    .map(|i| {
        (0..m)
            .map(|j| (0..k).map(|x| a[i][x] * b[x][j]).sum())
            .collect()
    })
    .collect()

}

```

Listing 1: Parallel implementation of GEMM using Rayon. We adopt a functional programming style to highlight Rust's and MemForge's flexibility.

SpMM. SpMM (Sparse Matrix-Dense Matrix Multiplication) operations are prevalent in graph analytics and Graph Neural Network (GNN). They serve as critical building blocks in algebraic graph

```rust
pub struct CSRMatrix<'a> {
values: Vec<f64, &'a MemForge>,
col_indices: Vec<usize, &'a MemForge>,
row_pointers: Vec<usize, &'a MemForge>,
num_rows: usize,
num_cols: usize,
nnz: usize,
format: MatrixFormat,
field_type: MatrixType,
sym: MatrixSymmetry,
}

pub fn par_spmm(&self, dense_matrix: &[Vec]) -> Vec<Vec> {
let n = self.num_rows;
let k = dense_matrix[0].len();
let mut result = vec![vec![0.0; k]; n];

result
    .par_iter_mut()
    .enumerate()
    .for_each(|(row, result_row)| {
        for idx in self.row_pointers[row]..self.row_pointers[row + 1] {
            let col = self.col_indices[idx];
            let val = self.values[idx];
            for dense_col in 0..k {
                result_row[dense_col] += val * dense_matrix[col][dense_col];
            }
        }
    });

result

}
```

Listing 2: Parallel SpMM implementation with CSR format. The method belongs to `CSRMatrix`; for simplicity, the result is stored in dense format.

Table 1: Matrix properties.

Name	Rows	Cols	NNZ	Type	Domain
b1_ss	7	7	15	Real	Chemistry
E40r5000	17281	17281	553956	Real	2D/3D
fs_183_1	183	183	1069	Real	2D/3D
Jgl009	9	9	50	Pattern	Graph
lp_afiro	27	51	102	Real	Linear Prog.

algorithms and are frequently optimized. Our implementation supports both COOrdinate (COO) and Compressed Row (CSR) sparse matrix formats.[2] As with GEMM, the outer loop is parallelized using the `rayon` crate. Listing 2 illustrates the CSR-based implementation. Input matrices are drawn from the SuiteSparse Matrix Collection, as summarized in Table 1.

Transformer Kernel. This benchmark simulates key operations in transformer models such as attention mechanisms and feedforward layers, used in models like BERT and GPT-3. The implementation includes multi-head attention, feedforward layers, and layer normalization. Multi-head attention computes multiple sets of queries, keys, and values in parallel, enabling diverse focus across input tokens. The outputs are concatenated while preserving the model dimension (`dim_model`). Feedforward layers apply transformations with intermediate dimensionality (`dim_ff`), typically larger than the model dimension, and utilize ReLU activation. Layer normalization ensures stable training. As dimensionality and head count increase, the workload intensifies, allowing us to evaluate performance in compute-intensive conditions. The core computation is a parallel GEMM kernel, previously shown in Listing 1.

4.2 Environmental Setup

Our experiments use QEMU version 9.2.90 with CXL extensions.[3] Each virtual machine is configured with 8 x86 64-bit virtual CPUs, 8GB DRAM, and a virtual NIC. In the first configuration, a 4GB CXL memory expander is directly attached to the host and backed by a regular file on the host.[4] In the second configuration, two 2 GB memory devices connect to the host through a CXL switch. Both configurations operate in Direct Access (DAX) mode to bypass the I/O stack.

We use Rust version 1.88.0-nightly due to required allocator features.

4.3 Experimental Results

4.3.1 Benchmarks. QEMU is not cycle-accurate and lacks a detailed timing model. Thus, absolute timing data must be interpreted cautiously. Nevertheless, relative trends are meaningful. Figure 3 compares runtime performance of GEMM, SpMM, and LLM Transformer benchmarks using MemForge in both QEMU configurations against the Rust standard allocator. In all plots, *global* indicates use of the standard Rust allocator, *dax* refers to the configuration with direct-attached CXL memory, and *switch* refers to the configuration using a CXL switch.

Across all benchmarks, MemForge introduces no notable runtime overhead. This is especially evident in small input sizes where allocation cost is relatively high. As input sizes increase, and execution becomes compute-dominated, the *global* and *dax* configurations exhibit similar performance. The *switch* configuration incurs slightly higher overhead due to increased latency from the CXL switch.

Overall, these findings support the viability of co-developing hardware and software using our approach. While the QEMU latency model is not reflective of real hardware, relative trends provide useful insight into expected behaviors.

4.3.2 Tracing. MemForge includes built-in tracing support to analyze correctness (e.g., memory leaks) and performance. With `RUST_LOG=trace`, MemForge emits trace data in JSON format, compatible with tools like Chrome Tracing and Perfetto [18].

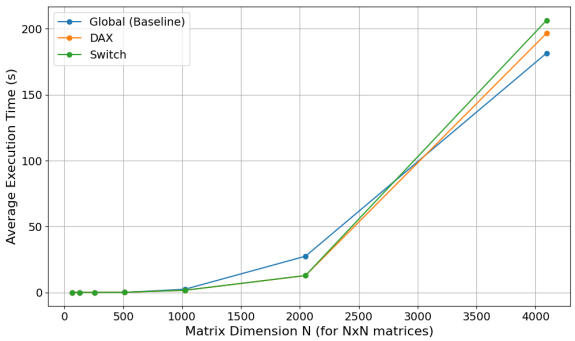
Figure 4 shows a trace from the SpMM benchmark (Listing 2) visualized with Perfetto. Red indicates `allocate()` calls; green indicates `deallocate()` calls. Two waves of calls appear: the first corresponds to allocating a `COOMatrix` from the parsed Matrix Market input; the second arises from constructing the `CSRMatrix`. The `COOMatrix` is deallocated after conversion, and the `CSRMatrix` is deallocated at the benchmark's end.

The trace reveals fine-grained timing and thread activity, aiding performance diagnosis. Additional introspective metrics (e.g., block

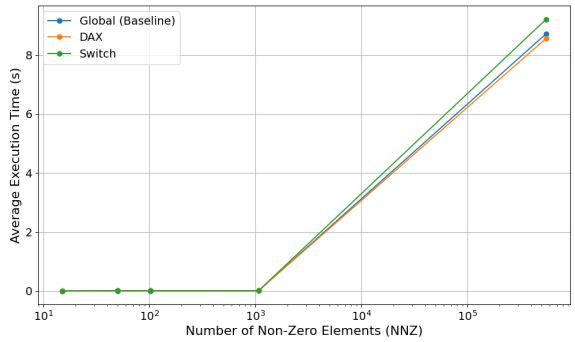
[2]We use CSR in our evaluation due to its slightly better performance.

[3]https://gitlab.com/jic23/qemu, branch cxl-2025-03-20.

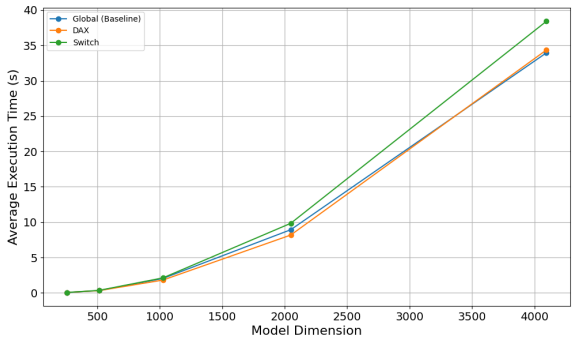
[4]The emulated memory is persistent, although persistence is not utilized here.



(a) Average execution time of GEMM operator on square input matrices of dimensions $N \times N$.



(b) Average execution time of SpMM operator for various sparse matrices from the SuiteSparse Matrix Collection. Log scale used on x-axis.



(c) Average execution time for LLM Transformer kernel across model dimensions. The number of heads and layer dimensions are adjusted to maintain compatibility.

Figure 3: Average execution time for tested benchmarks. Results reflect mean of 10 runs for each input. Experiments cover the Rust standard allocator (*global*) and two QEMU configurations (*dax* and *switch*).

splits/merges) are available but omitted here for brevity. We refer readers to Perfetto's documentation for a complete overview.

5 RELATED WORK

CXL Emulation: Efforts to emulate CXL hardware and system functionality have emerged as critical tools for evaluating CXL designs prior to the availability of physical hardware. CXL-DMSim [26] extends the gem5 simulator to create a software-based full-system simulation framework. By integrating support for CXL.mem and CXL.io protocols, it models CXL memory expanders and enables diverse emulation scenarios, ranging from memory allocation to OS-level management through NUMA-compatible device drivers. Although CXL-DMSim has been silicon-validated against real FPGA- and ASIC-based designs, it is limited to x86 systems and excludes support for CXL.cache protocols.

QEMU [3, 19], an open-source emulator, provides fundamental building blocks for emulating CXL.io configurations via host bridges, root ports, and mailbox functionality mapped to PCIe address spaces. Its Type-3 device emulation supports Host-Managed Device Memory (HDM), allowing memory endpoints configuration using either RAM or file backends. However, accelerator memory integration (HDM-D) and cache coherency—critical features of CXL.cache—are currently unsupported in QEMU. CXLSim [11] bridges gem5 and QEMU to introduce cache hit/miss tracking, enabling projections of application performance under simulated memory expander scenarios. Similarly, CXLMemSim [30] employs a software-level approach using eBPF traces to inject memory delays, providing rudimentary CXL.mem emulation for experimental purposes. However, due to the absence of accurate timing frameworks and latency variation support, CXLMemSim fails to capture the nuanced characteristics of the CXL microarchitecture.

CXL Performance Optimization: Recent advances in optimizing Compute Express Link (CXL) performance span performance characterization, modeling, and application-specific optimizations. Several studies have focused on characterizing performance on real CXL platforms. For instance, [23, 24] leverage genuine CXL-ready systems to examine synchronization challenges and the broader implications of memory expansion, while [28] explores emulated NUMA-based layouts to replicate CXL memory pooling scenarios on systems with varying processor-memory topologies. CXL performance has also been studied in serverless computing environments [4, 13, 14, 31]. These studies demonstrated CXL memory pooling approaches using latency-insensitive virtual machines (VMs) where entire memory allocations come from disaggregated CXL pools, while latency-sensitive VMs dynamically split their allocations between local DRAM and remotely pooled CXL memory based on memory predictions for untouched pages. However, these frameworks are restricted by their limited support for CXL protocols and lack multi-host emulation capabilities.

In addition to characterization, performance optimization techniques for CXL systems have been proposed in [15, 16, 25]. These efforts model CXL latency across hierarchical levels of the system to mitigate memory-related bottlenecks. For instance, [15] addresses slowdowns caused by page migration, proposing strategies such as adaptive throttling and optimization of interleaving ratios, while [25] devises object-level interleaving mechanisms combined with thread assignment tuned for scalability. Together, these studies provide foundational insights into overcoming CXL performance limitations.

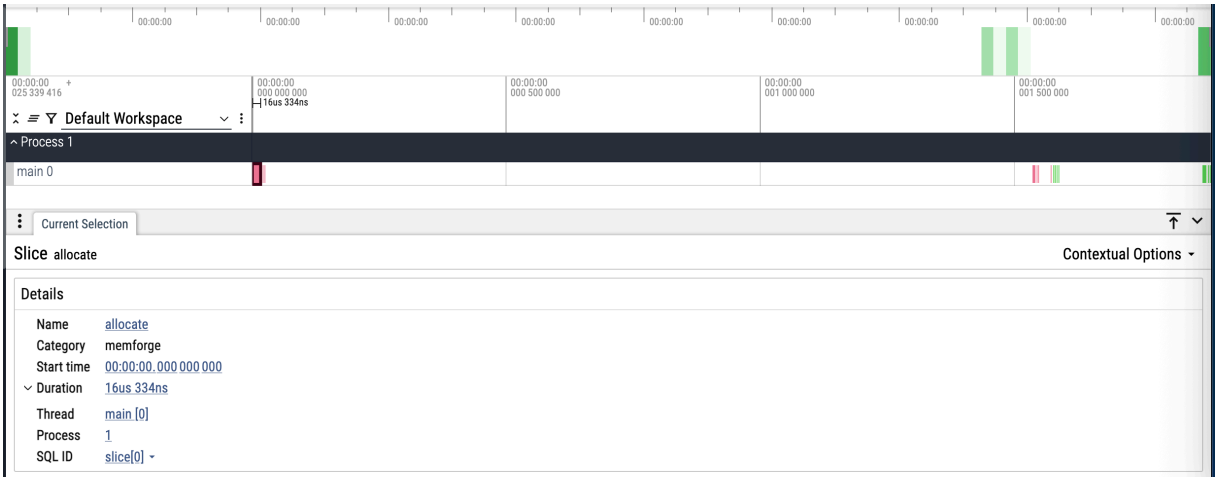


Figure 4: Execution trace of the SpMM benchmark using `b1_ss` as input and CSR format. Red and green blocks denote `allocate()` and `deallocate()` calls in MemForge, respectively.

CXL Applications: The versatility of CXL makes it applicable across diverse domains, enabling performance gains in memory-intensive workloads. In approximate nearest neighbor search, [9] demonstrates how software-hardware collaborative memory disaggregation can accelerate billion-scale search workflows. Likewise, CXL has been successfully employed in machine learning and database systems. For distributed deep learning [2] and large-scale training workloads [29], CXL facilitates efficient memory access, tensor offloading, and scalability. In K-nearest neighbor search [22], CXL-powered memory solutions address computational bottlenecks by leveraging pooled disaggregated memory. The technology also enhances database systems, with studies like [1, 8] exploring opportunities and challenges in memory expansion to support transaction processing [27] and emerging paradigms such as Retrieval Augmented Generation [20].

6 CONCLUSIONS AND FUTURE WORK

This work addresses the challenge of hardware-software co-development for future CXL systems. We emphasize the importance of developing software in tandem with hardware to foster bi-directional feedback between development teams. To this end, we introduced a co-development framework comprising a full-system CXL emulator and a novel memory allocator, MemForge.

Our results demonstrate that MemForge offers a simple and intuitive API, enabling the implementation of key kernels across various domains, including HPC, AI, and HPDA. We evaluated two hardware configurations—direct-attached and CXL switch-based memory—and showed that MemForge introduces negligible runtime overhead. Moreover, we highlighted MemForge's introspective capabilities for tracing and debugging.

Our experience shows that this framework provides a practical and effective environment for developing applications and testing CXL hardware prototypes. In future work, we plan to enhance timing accuracy by integrating a detailed timing model for CXL hardware. This will involve extending QEMU with support for either cycle-accurate simulators or analytical timing models to better assess performance characteristics of CXL-enabled systems. We also plan to support *cxl.cache*, which our current setup does not allow, and to bring under our hardware-software co-development system accelerator devices.

REFERENCES

[1] Ahn, M., Chang, A., Lee, D., Gim, J., Kim, J., Jung, J., Rebholz, O., Pham, V., Malladi, K., and Ki, Y. S. Enabling cxl memory expansion for in-memory database management systems. In *Proceedings of the 18th International Workshop on Data Management on New Hardware* (2022), pp. 1–5.

[2] Arif, M., Assogba, K., Rafique, M. M., and Vazhkudai, S. Exploiting cxl-based memory for distributed deep learning. In *Proceedings of the 51st International Conference on Parallel Processing* (2022), pp. 1–11.

[3] Bellard, F. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track* (2005), vol. 41, California, USA, pp. 10–5555.

[4] Berger, D. S., Ernst, D., Li, H., Zardoshti, P., Shah, M., Rajadnya, S., Lee, S., Hsu, L., Agarwal, I., Hill, M. D., et al. Design tradeoffs in cxl-based memory pools for public cloud platforms. *IEEE Micro 43*, 2 (2023), 30–38.

[5] Developers, T. P. tracing - application-level tracing for rust. https://crates.io/crates/tracing, 2024. Version 0.1.40.

[6] Developers, T. P. tracing-opentelemetry - opentelemetry integration for tracing. https://crates.io/crates/tracing-opentelemetry, 2024. Version 0.22.0.

[7] Developers, T. R. P. log - a lightweight logging facade for rust. https://crates.io/crates/log, 2024. Version 0.4.21.

[8] Guo, Y., and Li, G. A cxl-powered database system: Opportunities and challenges. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)* (2024), IEEE, pp. 5593–5604.

[9] Jang, J., Choi, H., Bae, H., Lee, S., Kwon, M., and Jung, M. {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)* (2023), pp. 585–600.

[10] Kim, K., Kim, H., So, J., Lee, W., Im, J., Park, S., Cho, J., and Song, H. Smt: Software-defined memory tiering for heterogeneous computing systems with cxl memory expander. *IEEE Micro 43*, 2 (2023), 20–29.

[11] Kim, S., Kang, J., Kim, K., Lee, S., and Nam, B. Cxlsim: A simulator for cxl memory expander. In *2025 IEEE International Conference on Big Data and Smart Computing (BigComp)* (2025), IEEE, pp. 156–159.

[12] Knuth, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Addison-Wesley, 1997. See Section 2.5: Dynamic Storage Allocation.

[13] Li, H., Berger, D. S., Hsu, L., Ernst, D., Zardoshti, P., Novakovic, S., Shah, M., Rajadnya, S., Lee, S., Agarwal, I., et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (2023), pp. 574–587.

[14] Liu, J., Hadian, H., Wang, Y., Berger, D. S., Nguyen, M., Jian, X., Noh, S. H., and Li, H. Systematic cxl memory characterization and performance analysis at

scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (2025), pp. 1203–1217.

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

[16] Maruf, H. A., Wang, H., Dhanotia, A., Weiner, J., Agarwal, N., Bhattacharya, P., Petersen, C., Chowdhury, M., Kanaujia, S., and Chauhan, P. 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* (2023), pp. 742–755.

[17] Matsakis, N., and the Rayon Developers. rayon - data parallelism in rust. https://crates.io/crates/rayon, 2024. Version 1.10.0.

[18] Perfetto Development Team. Perfetto documentation. https://perfetto.dev/docs/, 2025. Accessed: 2025-06-20.

[19] QEMU Project. Qemu emulator. https://github.com/qemu/qemu, 2024.

[20] Quinn, D., Patel, N., and Alian, M. Compute-enabled cxl memory expansion for efficient retrieval augmented generation. *IEEE Micro* (2025).

[21] Sharma, D. D. Compute express link (cxl): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy. *IEEE Micro 43*, 2 (2022), 99–109.

[22] Sim, J., Ahn, S., Ahn, T., Lee, S., Rhee, M., Kim, J., Shin, K., Moon, D., Kim, E., and Park, K. Computational cxl-memory solution for accelerating memory-intensive applications. *IEEE Computer Architecture Letters 22*, 1 (2022), 5–8.

[23] Suetterlein, J., Manzano, J., and Marquez, A. Synchronization for cxl based memory. In *Proceedings of the International Symposium on Memory Systems* (2024), pp. 178–185.

[24] Sun, Y., Yuan, Y., Yu, Z., Kuper, R., Song, C., Huang, J., Ji, H., Agarwal, S., Lou, J., Jeong, I., et al. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (2023), pp. 105–121.

[25] Wang, X., Liu, J., Wu, J., Yang, S., Ren, J., Shankar, B., and Li, D. Performance characterization of cxl memory and its use cases. In *International Parallel and Distributed Processing Symposium* (2025).

[26] Wang, Y., Wu, L., Hong, W., Ou, Y., Wang, Z., Gao, S., Zhang, J., Ma, S., Dong, D., Qi, X., et al. A comprehensive simulation framework for cxl disaggregated memory. *arXiv preprint arXiv:2411.02282* (2024).

[27] Wang, Z., Chen, Y., Li, C., Guan, Y., Niu, D., Guan, T., Du, Z., Wei, X., and Sun, G. Ctxnl: A software-hardware co-designed solution for efficient cxl-based transaction processing. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (2025), pp. 192–209.

[28] Wu, J., Liu, J., Kestor, G., Gioiosa, R., Li, D., and Marquez, A. Performance study of cxl memory topology. In *Proceedings of the International Symposium on Memory Systems* (2024), pp. 172–177.

[29] Xu, D., Feng, Y., Shin, K., Kim, D., Jeon, H., and Li, D. Efficient tensor offloading for large deep-learning model training based on compute express link. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis* (2024), IEEE, pp. 1–18.

[30] Yang, Y., Safayenikoo, P., Ma, J., Khan, T. A., and Quinn, A. Cxlmemsim: A pure software simulated cxl. mem for performance characterization. *arXiv preprint arXiv:2303.06153* (2023).

[31] Zhong, Y., Berger, D. S., Waldspurger, C., Wee, R., Agarwal, I., Agarwal, R., Hady, F., Kumar, K., Hill, M. D., Chowdhury, M., et al. Managing memory tiers with {CXL} in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (2024), pp. 37–56.