CCom: A Compiler Framework for Linear Algebra on CXL-Attached Memory

Zhen Peng
Pacific Northwest National Laboratory
Richland, USA
zhen.peng@pnnl.gov

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

Gokcen Kestor
Barcelona Supercomputing Center
Barcelona, Spain
gokcen.kestor@bsc.es

Andres Marquez
Pacific Northwest National Laboratory
Richland, USA
Andres.Marquez@pnnl.gov

Abstract

Memory disaggregation via Compute Express Link (CXL) offers a promising path to address the growing memory demands of data-intensive applications. However, effectively utilizing CXL-attached memory remains challenging due to its higher access latency compared to local DRAM. We propose CCom, a compiler framework that enables efficient utilization of CXL memory for linear algebra computations. CCom provides a domain-specific language (DSL) with an `allocator` annotation, allowing programmers to specify memory placement while the compiler automatically allocates memory objects to specific memory pools and generates optimized code. Built on MLIR, CCom employs tiling and double buffering with auxiliary threads to improve cache performance and hide CXL memory latency. We evaluate CCom on matrix multiplication using a CXL prototype system. Results show that while naïve CXL memory access incurs a 3.3× slowdown compared to DRAM, CCom's optimizations achieve a 2.50× speedup over naïve DRAM, with 12.6× higher instructions per cycle than naïve CXL memory code.

CCS Concepts

• **Software and its engineering** → **Compilers**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**; **Interconnection architectures**.

Keywords

Compiler, compute express link, fabric-attached memory, domain-specific language, linear algebra

ACM Reference Format:
Zhen Peng, Roberto Gioiosa, Gokcen Kestor, and Andres Marquez. 2026. CCom: A Compiler Framework for Linear Algebra on CXL-Attached Memory. In *Proceedings of the 5th Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS 2026)*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

HCDS 2026, Pittsburgh, PA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

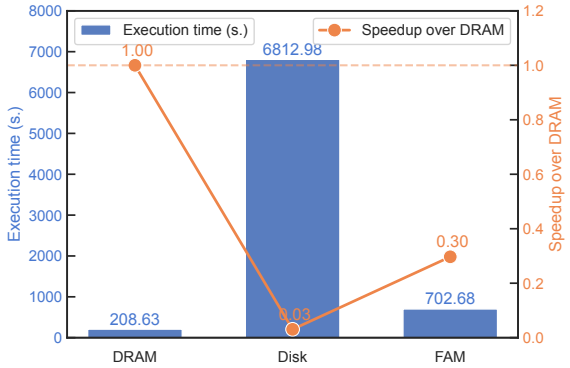

Figure 1: Performance of matrix multiplication on DRAM, Disk, and FAM. Matrix size: 16384 × 16384**, FP64, 28 threads. To simulate the case where data cannot fit in DRAM and must be frequently accessed from disk, the disk version uses `msync` to enforce synchronization from the `mmap` memory to its underlying disk file.**

1 Introduction

The growing memory demands of data-intensive applications, such as scientific simulations, machine learning, and graph analytics, are pushing the limits of traditional server memory architectures. Practically, going beyond the limit of workstation main memory requires the use of storage technologies, such as hard drives or non-volatile memory (NVMe) devices, which add orders of magnitude higher latencies compared to DRAM. Fabric-attached memory (FAM) has emerged as a trade-off between the low latency, low capacity, and high cost of DRAM and the high capacity, high latency, and low cost of storage devices. Figure 1 shows a clear example: in this experiment, the inputs and output of a square matrix-matrix multiplication are stored in DRAM, disk, and FAM. The results show that storing data on disk increases execution time by orders of magnitude, while FAM provides performance that is between DRAM and disk storage. As opposed to storage memory, which is interconnected through the I/O bus and provides block-level access,[1] FAM is attached to computing elements through specific interconnection buses and provides word-level access (like DRAM), thus enabling users to control their data at the object level.

[1]Because of the complexity and latency of moving data, the OS generally maintains a cache in memory, which means an entire page (4KB, 2MB, or even 8MB) of data is moved from disk to DRAM when a new block is accessed.

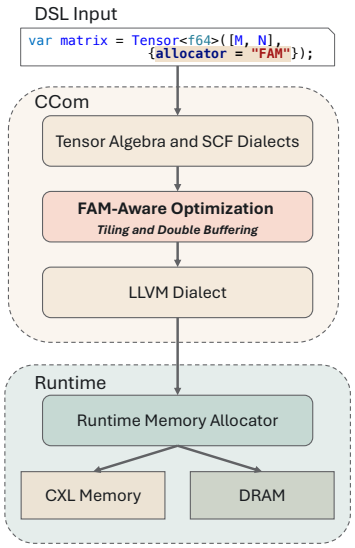

Figure 2: Overview of the CCom workflow. Users annotate tensor declarations with memory placement attributes. The compiler generates optimized code targeting heterogeneous memory systems with CXL-attached memory.

Among various standards, Compute Express Link (CXL) [28] is an industry-standard interconnect that enables access to fabric-attached memory with significantly lower latency than prior network-attached solutions such as remote direct memory access (RDMA) or disk storage. CXL-attached memory provides a byte-addressable memory pool accessible via standard load/store instructions, making it an attractive target for memory-intensive workloads. However, CXL memory exhibits 2–5× higher access latency compared to local DRAM [13], presenting both opportunities and challenges for system software and applications. As CXL-based systems start to appear, in the form of prototypes and production systems, it is important to understand how to leverage fabric-attached memory and prepare the software stack and applications [27].

Existing approaches to mitigating FAM latency fall into three categories: *Kernel-based* solutions [6, 9, 13] modify the operating system to transparently migrate pages between local and remote memory, but suffer from page fault overhead and lack application-level control; *Library-based* approaches [21] provide explicit APIs for remote memory access, offering fine-grained control but requiring significant application modifications; *Compiler-based* solutions [8, 25] offer a middle ground: they allow programmers to express intent through high-level annotations while automatically generating optimized code for heterogeneous memory systems.

In this paper, we propose CCom, a compiler-based solution to leverage CXL FAM while mitigating the effects of higher memory latency. CCom allows users to write linear algebra computations using a domain-specific language (DSL) with simple annotations specifying object memory placement. Based on the multi-level intermediate representation (MLIR) framework, our compiler automatically generates optimized code that leverages a large CXL-attached memory pool while hiding its latency through techniques such as tiling and double buffering. Double buffering prefetches data into DRAM buffers and overlaps prefetching with computation (as

```
def main() {
  # Tensor declarations with allocator attribute
  var A = Tensor<f64>([M, K], {allocator = "FAM"});
  var B = Tensor<f64>([K, N], {allocator = "FAM"});
  var C = Tensor<f64>([M, N], {allocator = "DRAM"});

  # Matrix multiplication
  C[i, j] = A[i, k] * B[k, j];
}
```

Figure 3: Matrix-matrix multiplication implemented in the CCom DSL. Users can annotate each tensor with an `allocator` attribute. The compiler automatically associates each memory reference with its requested allocator.

detailed in Section 2.3). These optimizations increase cache performance and proactively move data from FAM to DRAM based on application semantics, thereby enabling applications to exploit both larger memory pools and DRAM-like latency.

We demonstrate CCom's capabilities using a CXL prototype system and matrix-matrix multiplication kernels, which are the basis of many scientific and engineering applications. While the demonstration is for a specific CXL prototype, in our design the compiler effectively decouples the user-facing DSL from the low-level allocator library. Our results show that the code optimized by CCom achieves a 2.50× speedup over the unoptimized DRAM version on 16384 × 16384 matrices using 28 threads. In summary, we make the following contributions:

- A DSL for linear algebra computation where users can explicitly annotate memory objects (tensors) with an `allocator` attribute. The DSL provides explicit control over data placement in heterogeneous memory systems.
- A multi-level IR compilation pipeline built on MLIR that propagates memory allocation decisions through progressive lowering stages.
- Optimization strategies including tiling and double buffering with auxiliary threads to mitigate CXL memory latency.
- A case study demonstrating CCom on matrix multiplication, with analysis of tile size selection and performance trade-offs.

2 Design and Implementation

CCom is designed to simplify the development of scientific applications that leverage FAM to extend memory capacity beyond the DRAM available on computing workstations.

CCom is implemented within the MLIR framework, which allows us to reuse well-established optimization and code lowering passes and focus on FAM-specific concerns: memory allocation, data movement between DRAM and FAM, and latency-hiding optimizations.

As illustrated in Figure 2, users write linear algebra programs in a tensor-based DSL, annotating tensors with their desired memory placement using the `allocator` attribute. The compiler processes these programs through a series of lowering passes, generating executable code that allocates data in the appropriate memory domain and applies optimizations to hide CXL memory latency. Our solution consists of three integrated components: a DSL, a multi-level IR compilation pipeline, and optimization strategies for proactive data movement.

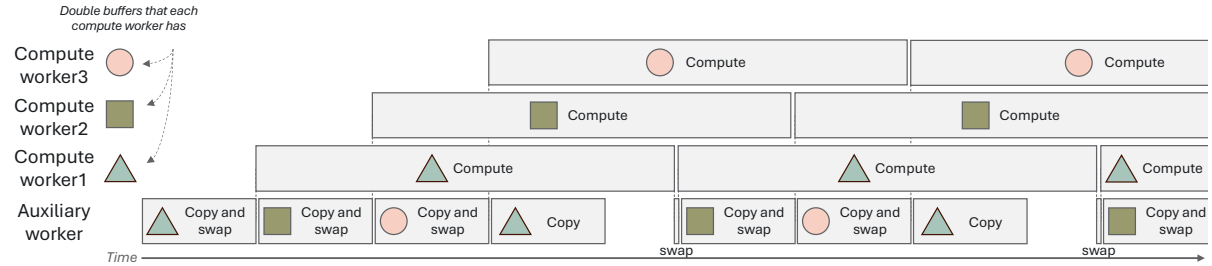

Figure 4: Double buffering timeline example. Three compute workers process data in their buffers, while one auxiliary worker prefetches (copies) data from the original FAM location to dedicated DRAM buffers and swaps buffers to enable simultaneous copying and computing. The buffer shapes are illustrative and do not represent actual sizes.

2.1 Domain-Specific Language

To simplify the task of writing scientific applications and storing memory objects in the appropriate memory space, we designed a domain-specific language (DSL) for tensor algebra applications. With our DSL, users express linear algebra computations using high-level tensor operations instead of low-level nested loops. Figure 3 shows an example of a matrix-matrix multiplication kernel developed using the CCom DSL. In this program, the user defines three tensors (`A`, `B`, and `C`): `A` and `B` are the program inputs while `C` contains the result of the matrix multiplication. The CCom DSL provides the `allocator` keyword to specify the memory domain for tensor storage, either `DRAM` or `FAM`.[2] In the example in Figure 3, the two input tensors `A` and `B` are stored in FAM, while the output tensor `C` is stored in DRAM. Beyond the annotation, the rest of the code is agnostic of memory placement—the compiler generates the correct allocation calls and, as described in the following sections, automatically applies optimizations to hide FAM latency.

2.2 Compilation Pipeline

CCom leverages a multi-level IR compilation pipeline built on COMET and MLIR [11, 16]. The key advantage of this multi-level approach is that it enables *progressive lowering*: each level of abstraction exposes different optimization opportunities, and information from higher levels (such as tensor semantics and memory placement decisions) can guide transformations at lower levels. This is particularly important for CXL memory optimization, where we need both high-level knowledge about data access patterns and low-level control over memory allocation.

At the top of the pipeline, CCom utilizes the Tensor Algebra (TA) dialect and Index Tree dialect [16, 18]. The TA dialect captures the semantics of algebraic operations—including multi-dimensional tensors, index labels, and contractions—using Einstein notation. This high-level representation is essential because it preserves the mathematical structure of computations, enabling algebraic optimizations such as operation reordering and fusion. Critically, the TA dialect is where the `allocator` attribute is first attached to tensor declarations, marking which objects should reside in FAM versus DRAM. The Index Tree dialect extends support for sparse tensors, semirings, and masking operations. Together, these dialects provide a rich, domain-specific representation that would be lost if we started directly with low-level loops.

[2] `DRAM` is inferred by default if no allocator is specified.

The middle layers leverage MLIR's built-in MemRef and SCF (Structured Control Flow) dialects. The MemRef dialect [5] converts abstract tensors into concrete memory references with explicit shapes, strides, and memory spaces. This is where the `allocator` attribute is translated into memory space annotations that will eventually determine which allocation function to call. The SCF dialect [4] represents the loop nests that implement tensor operations. Having explicit loop structures at this level is crucial for our optimizations: we can apply *tiling* to partition computations into cache-friendly blocks, and we can identify loop levels where *double buffering* should be enabled. These transformations would be difficult to express at the tensor level (too abstract) or at LLVM IR (too low-level, loop structure is obscured).

Finally, the code is lowered to LLVM IR, which provides a portable representation across architectures. At this stage, memory allocations marked for FAM are lowered to calls to the CXL memory allocator, while DRAM allocations use the standard `malloc()`. The LLVM backend then generates optimized machine code, benefiting from standard compiler optimizations such as register allocation and instruction scheduling. Thanks to LLVM's multi-target support, CCom can potentially compile the same DSL code for different hardware platforms.[3]

The placement of CCom's CXL-specific optimizations—between the SCF dialect and LLVM IR—is deliberate. At this level, we have access to both (1) high-level information about tensor access patterns from the TA dialect, which tells us *what* data will be accessed, and (2) explicit loop structures from SCF, which tell us *when* data will be accessed. This combination enables effective tiling (to improve cache utilization) and double buffering (to hide FAM latency). Performing these optimizations earlier (at the TA level) would be premature, as we lack concrete loop structures. Performing them later (at LLVM IR) would be difficult, as loop structures are obscured by control flow graphs and the connection to tensor semantics is lost.

2.3 Optimization Strategies

To mitigate the higher latency of CXL memory compared to local DRAM, CCom implements two key optimizations: tiling and double buffering. The key idea is that CCom exploits the semantics of high-level operators in the DSL, such as matrix-matrix multiplication, to automatically apply optimizations that would otherwise be difficult or require complex code analysis. Because CCom knows exactly

[3] Currently, CCom's optimizations are tested on x86 CPUs. Support for GPUs and other accelerators is future work.

what computation the user wants to perform, it can directly apply these optimizations without runtime overhead.

Tiling is a well-known technique that divides input tensors into smaller, cache-friendly chunks to improve data locality. Tile sizes are selected so that tiles fit in processor cache. While loop tiling is effective on many architectures and widely employed in scientific and engineering applications, in this context it provides an additional opportunity: hiding the latency of FAM. The compiler knows exactly the order in which tiles will be accessed and can prefetch the next tiles from FAM into DRAM. In this way, CCom effectively prefetches the next tile from CXL memory while computing on the current tile, maximizing data reuse in DRAM.

Exploiting the predictable access patterns of tiling, CCom applies double buffering to overlap computation with data movement. The optimization proceeds as follows: First, CCom analyzes the code's computation pattern and gathers information such as input size and loop range. Then, CCom tiles the loop structure to divide input tensors into cache-friendly chunks. Finally, CCom generates double buffering code to enable efficient prefetching. Specifically, in addition to the existing parallel execution, it creates an *auxiliary worker* process, while the existing processes are called *compute workers*. There can be multiple auxiliary workers, each serving multiple compute workers.

For double buffering, CCom also generates auxiliary data structures, including a double buffer and an atomic token for each compute worker. The double buffer contains two tiles: one for computing and one for prefetching. While a compute worker operates on data in one tile, the auxiliary worker prefetches the next tile into the other tile. Once both prefetching and computation complete, the auxiliary worker swaps the two buffers. This allows computation and prefetching to overlap, with the atomic token providing synchronization between auxiliary and compute workers.

Double buffering improves performance for two reasons. First, the tiled buffer structure is cache-friendly: tiles are sized to fit in cache, and once allocated in DRAM, compute workers can leverage data reuse to hide FAM access latency. Second, overlapping prefetching with computation hides data movement overhead and improves compute efficiency. Figure 4 illustrates three compute workers operating with one auxiliary worker. While buffer swapping and computing cannot occur simultaneously, prefetching and computation are effectively overlapped.

3 Evaluation

We evaluate CCom using dense matrix multiplication (GEMM) as a case study. Our evaluation addresses the following questions:

- What speedup does CCom achieve over baseline FAM and DRAM execution? (Section 3.1)
- How do CCom's optimizations affect low-level performance metrics such as cache miss rates and IPC? (Section 3.2)
- How sensitive is performance to tile size selection? (Section 3.3)

Hardware Platform. We conduct experiments on a CXL memory prototype system. The testbed consists of a compute node connected to an FPGA-based memory accelerator via a CXL 1.x to 2.x protocol conversion interface over PCIe 5.0. The compute node has a dual-socket Intel Xeon Gold 5420+ processor (28 cores per socket, 4.1 GHz). Each core has a private 48 KB L1d cache, a private 2 MB L2 cache, and access to a shared 52.5 MB L3 cache. The FAM device provides 32 GB of CXL-attached memory, managed by a memory server daemon running on the compute node.

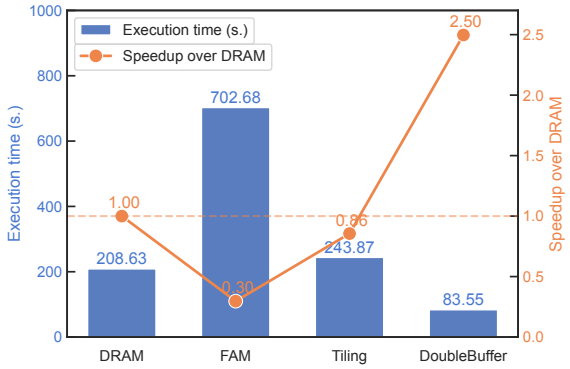

Figure 6: Performance of matrix multiplication for DRAM, FAM, Tiling, and DoubleBuffer configurations. Matrix size: 16384×16384**. FP64. 28 threads.**

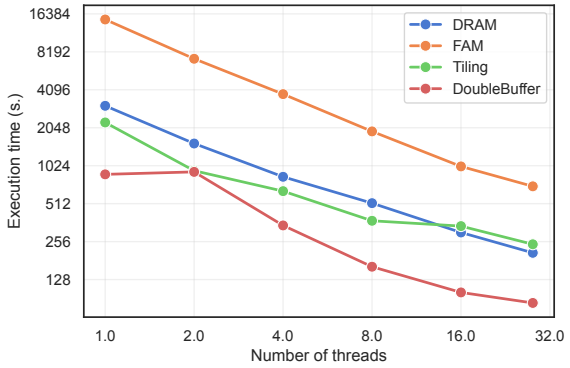

Figure 7: Performance scalability of matrix multiplication for DRAM, FAM, Tiling, and DoubleBuffer from 1 to 28 threads. Matrix size: 16384×16384**, FP64.**

Workloads and Configurations. We evaluate GEMM ($C = A \times B$) with square matrices of size $N \times N$ using double-precision (FP64) values. We compare four configurations:

- *DRAM*: All matrices allocated in local DRAM (baseline).
- *FAM*: All matrices allocated in CXL-attached FAM.
- *Tiling*: Matrices in FAM with tiled loop execution.
- *DoubleBuffer*: Matrices in FAM with double buffering; tiles are prefetched into DRAM buffers.

```
for (i = 0; i < N; i++)
  for (k = 0; k < N; k++)
    for (j = 0; j < N; j++)
      C[i,j] += A[i,k] * B[k,j];
```

Figure 5: Unoptimized GEMM kernel.

The *DRAM* and *FAM* both use the unoptimized kernel shown in Figure 5. For *DoubleBuffer*, one thread serves as the auxiliary worker while the remaining threads are compute workers. In single-threaded mode, the same thread performs both prefetching and computation sequentially. The best tile sizes for Tiling and DoubleBuffer are determined by tuning. We use up to 28 threads (single socket) to avoid NUMA effects.

3.1 Main Performance Results

Figure 6 compares the four configurations on 16384×16384 matrices using 28 threads. Using DRAM as the baseline, naïve FAM

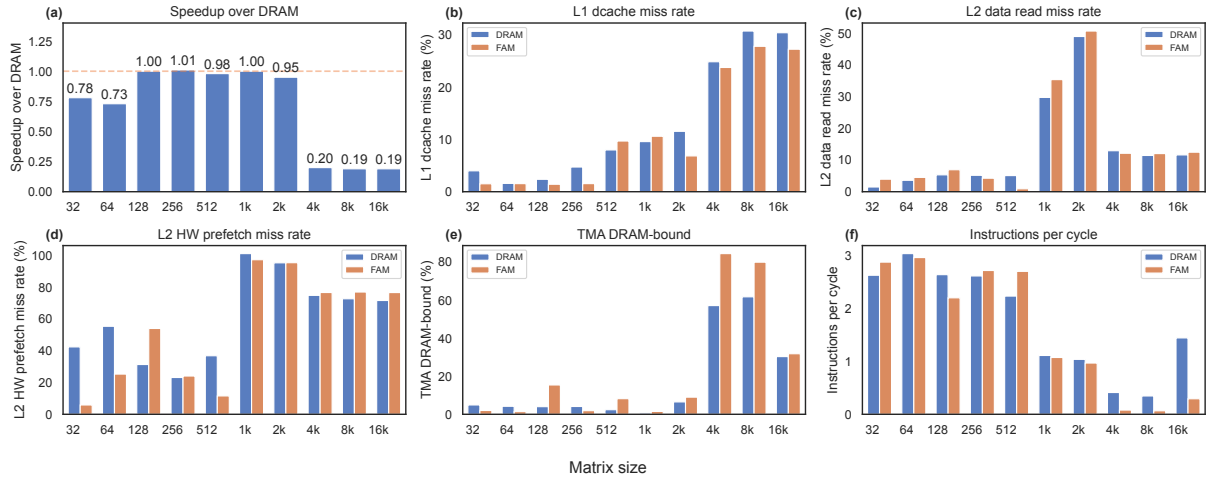

Figure 8: Performance comparison of matrix multiplication on DRAM and FAM. Matrix side length: 32 to 16384, FP64, single thread.

execution achieves only 0.30× the performance of DRAM—a 3.3× slowdown due to FAM's higher access latency. Tiling alone improves FAM performance to 0.86× of DRAM by improving cache locality, narrowing the gap significantly but still falling short of DRAM performance.

Notably, *DoubleBuffer* achieves a 2.50× speedup over the DRAM baseline. This counterintuitive result—FAM-based execution outperforming DRAM—is explained by the combination of two effects. First, double buffering prefetches tiles from FAM into DRAM buffers and overlaps prefetching with computation, so compute workers operate entirely on DRAM-resident data, effectively eliminating FAM latency from the critical path. Second, the tiled access pattern ensures that the working set fits in L2 cache (a 512×512 FP64 tile is 2 MB), resulting in significantly better cache utilization than the naïve DRAM baseline, which suffers from cache thrashing on the full 16384×16384 matrix. In other words, DoubleBuffer does not merely hide FAM latency—it restructures the computation to be more cache-friendly than unoptimized DRAM execution.

Figure 7 shows thread scalability from 1 to 28 threads on the same matrix size. DoubleBuffer consistently outperforms the other configurations across all thread counts. Both DRAM and FAM scale well with increasing thread counts but maintain a constant performance gap between them. Tiling narrows this gap at all thread counts by improving cache behavior. The exception is DoubleBuffer at 2 threads, where it shows minimal improvement over single-threaded execution. This occurs because with only one compute thread and one auxiliary thread, the auxiliary thread cannot prefetch fast enough to keep up, and synchronization overhead offsets the benefit of overlapping. Beyond 2 threads, DoubleBuffer scales effectively as the ratio of compute workers to auxiliary workers increases. Overall, DoubleBuffer achieves a 2.64× geometric mean speedup over DRAM across all thread counts.

3.2 Profiling Results

To understand the performance differences, we use `perf` to collect hardware performance counters.

Impact of Matrix Size. Figure 8 profiles DRAM and FAM (no tiling or double buffering) across matrix sizes (side length from 32 to 16384) using a single thread (to isolate memory effects from inter-thread cache interference). The results in Figure 8-(a) reveal three distinct regimes: (1) For small matrices ($N \leq 64$), FAM is slower because the working set fits in cache but each initial load incurs FAM's higher latency. (2) For medium matrices ($128 \leq N \leq 2048$), FAM achieves near-DRAM performance because the working set fits in L3 cache, allowing data reuse to amortize the initial FAM access cost. (3) For large matrices ($N \geq 4096$), FAM performance degrades significantly as the working set exceeds cache capacity, exposing FAM's latency on every cache miss. The profiling data confirms this interpretation: L1 cache miss rates increase with matrix size (Figure 8-(b)), and IPC drops sharply for large matrices (Figure 8-(f)), indicating memory-bound execution. In addition, Figure 8-(c) and -(d) show that L2 miss rates and hardware prefetch miss rates increase when the matrix exceeds the L2 cache size (2 MB for a 512×512 FP64 matrix). Figure 8-(e) shows the TMA DRAM Bound metric, which estimates how often the CPU stalls on DRAM accesses. When the matrix side length reaches 4096 or larger, DRAM-related stalls increase significantly as the working set exceeds L3 cache capacity. We use TMA DRAM Bound in place of LLC miss rates, as the latter could not be reliably measured on our prototype system.

DoubleBuffer's Effect. Table 1 compares performance counters for FAM and DoubleBuffer on 16384×16384 matrices with 28 threads. DoubleBuffer achieves 12.6× higher IPC (4.30 vs. 0.34) due to several factors: (1) Lower L2 miss rate (1.98% vs. 4.42%) because the tiled working set is sized to fit within the 2 MB L2 cache, reducing capacity misses compared to naïve FAM execution that traverses the full matrix with poor locality. (2) Lower hardware prefetch miss rate (41.19% vs. 52.13%) because the tiled, sequential access pattern is more regular and predictable for the hardware prefetcher. (3) Higher DRAM Bound metric (6.40% vs. 2.10%), indicating that DoubleBuffer's remaining stalls come from DRAM access rather than FAM access—exactly the intended effect of prefetching data from FAM into DRAM buffers.

Summary. The profiling results in Figure 8 show that when the working set fits in cache, FAM achieves comparable performance to DRAM. Tiling exploits this insight by partitioning the computation

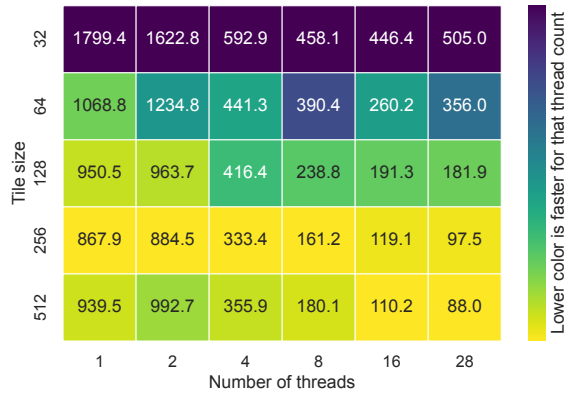

Figure 9: DoubleBuffer performance sensitivity to tile size and thread count. Cell values show execution time in seconds. Matrix size: 16384 × 16384**, FP64.**

into cache-friendly chunks, which is why it outperforms naïve FAM. DoubleBuffer combines tiling's cache performance benefits with prefetching data from FAM into DRAM (Table 1) and overlapping prefetching with computation, further reducing latency. This is why DoubleBuffer surpasses even naïve DRAM execution in Figure 6.

3.3 Sensitivity Results

Figure 9 shows DoubleBuffer's sensitivity to tile size across different thread counts on 16384 × 16384 matrices. The optimal tile size consistently falls in the range of 256–512, regardless of thread count. This range corresponds to tiles that fit within the 2 MB L2 cache (a 512 × 512 FP64 tile is 2 MB), enabling effective data reuse while the auxiliary thread prefetches the next tile. Smaller tiles incur higher prefetching overhead relative to computation, while larger tiles exceed L2 capacity, increasing cache misses. The consistency of the optimal range across thread counts suggests that tile size selection can be determined statically based on cache hierarchy parameters, simplifying auto-tuning.

4 Related Work

CXL and Disaggregated Memory Systems. Early disaggregation work such as Infiniswap [7], LegoOS [23], and Google's far memory system [9] explored RDMA-based approaches, while AIFM [21] and Semeru [26] provided application-level APIs. With CXL, Pond [12] studied memory pooling, TPP [13] proposed kernel-level page placement, and DirectCXL [6] demonstrated direct CXL memory access. Unlike these kernel-based approaches, CCom takes a compiler-based approach with explicit data placement annotations.

Heterogeneous Memory and Data Tiering. Data tiering research has explored how to place data across memory tiers with different performance characteristics. Dulloor et al. [3] presented foundational work on DRAM–NVM tiering, HeMem [20] introduced scalable tiered memory management, and online guidance approaches [17] use profiling to steer allocation. These systems operate at page or object granularity with runtime profiling, whereas CCom specifies placement at the tensor level through compile-time annotations.

Metric	FAM	DoubleBuffer
L1 d-cache miss rate	5.84%	6.32%
L2 data read miss rate	4.42%	1.98%
L2 hardware prefetch miss rate	52.13%	41.19%
Instructions per cycle	0.34	4.30
TMA DRAM Bound	2.10%	6.40%

Table 1: Hardware performance counter comparison between FAM and DoubleBuffer. Matrix size: 16384 × 16384**, FP64, 28 threads.**

Tensor Compilers and DSLs. Domain-specific compilers for tensor computations have gained significant traction. Halide [19] pioneered separating algorithm from schedule, TVM [2] extended this with auto-tuning, MLIR [11] provides multi-level IR infrastructure, and Tiramisu [1] leverages polyhedral compilation. CCom builds on COMET [16, 18], extending it with memory placement annotations and CXL-specific optimizations.

Loop Tiling, Prefetching, and Double Buffering. Loop tiling is a well-established technique for improving cache locality. Lam et al. [10] established foundational principles for tile size selection and cache performance of blocked algorithms. Software-controlled prefetching [14, 15] hides memory latency by issuing prefetch instructions ahead of data use. Double buffering maintains two buffers to overlap memory access with computation; Sancho and Kerbyson [22] analyzed optimal buffer sizing on multicore architectures, and Zhang et al. [24] proposed a non-stop double buffering mechanism for dataflow architectures. CCom adapts these techniques to the CXL memory context, where higher latency necessitates smaller tile sizes and auxiliary threads for data movement.

5 Conclusion, Limitations, and Future Work

We presented CCom, a compiler framework that enables efficient utilization of CXL-attached memory for linear algebra computations. CCom allows programmers to annotate tensor declarations with an `allocator` attribute to specify memory placement, while the compiler automatically generates optimized code through tiling and double buffering with auxiliary threads. Our evaluation on matrix multiplication demonstrates that while naïve FAM access incurs a 3.3× slowdown compared to DRAM, CCom's double buffering optimization achieves a 2.50× speedup over naïve DRAM execution. Profiling results confirm that double buffering improves IPC by 12.6× by effectively prefetching tiles from FAM into DRAM buffers.

The major limitations of CCom are that tile sizes must be tuned based on cache sizes, the optimized computation kernel is limited, and memory placement decisions rely on user annotations. Looking forward, we plan to extend CCom with automatic tile size selection based on cache hierarchy analysis, support for additional linear algebra operations, and exploration of automatic hybrid allocation and placement strategies that dynamically balance data placement between CXL memory and DRAM based on access patterns.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments and helpful suggestions. This work is supported by the US DOE Office of Science project "Advanced Memory to Support Artificial Intelligence for Science" at PNNL. PNNL is operated by Battelle Memorial Institute under Contract DEAC06-76RL01830.

References

[1] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. (2019), 193–205.

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[3] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[4] Multi-Level IR Compiler Framework. 2020. SCF Dialect. https://mlir.llvm.org/docs/Dialects/SCFDialect/. Accessed: 2026-02-24.

[5] Multi-Level IR Compiler Framework. 2021. MemRef Dialect. https://mlir.llvm.org/docs/Dialects/MemRef/. Accessed: 2026-02-24.

[6] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.

[7] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[8] Zhiyuan Guo, Zijian He, and Yiying Zhang. 2023. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 692–708.

[9] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 317–330.

[10] Monica D Lam, Edward E Rothberg, and Michael E Wolf. 1991. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review* 25, Special Issue, 63–74.

[11] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

[12] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.

[13] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 742–755.

[14] Todd Carl Mowry. 1994. *Tolerating latency through software-controlled data prefetching*. Ph. D. Dissertation.

[15] Todd C Mowry, Monica S Lam, and Anoop Gupta. 1992. Design and evaluation of a compiler algorithm for prefetching. *ACM Sigplan Notices* 27, 9 (1992), 62–73.

[16] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2020. Comet: A domain-specific compilation of high-performance computational chemistry. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 87–103.

[17] M Ben Olson, Brandon Kammerdiener, Michael R Jantz, Kshitij A Doshi, and Terry Jones. 2022. Online application guidance for heterogeneous memory systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 3 (2022), 1–27.

[18] Zhen Peng, Rizwan A Ashraf, Luanzheng Guo, Ruiqin Tian, and Gokcen Kestor. 2023. Automatic Code Generation for High-Performance Graph Algorithms. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 14–26.

[19] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[20] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 392–407.

[21] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332.

[22] José Carlos Sancho and Darren J Kerbyson. 2008. Analysis of double buffering on two different multicore architectures: Quad-core Opteron and the Cell-BE. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–12.

[23] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 69–87.

[24] Xu Tan, Xiao-Wei Shen, Xiao-Chun Ye, Da Wang, Dong-Rui Fan, Lunkai Zhang, Wen-Ming Li, Zhi-Min Zhang, and Zhi-Min Tang. 2018. A non-stop double buffering mechanism for dataflow architecture. *Journal of Computer Science and Technology* 33, 1 (2018), 145–157.

[25] Brian R Tauro, Brian Suchy, Simone Campanoni, Peter Dinda, and Kyle C Hale. 2024. TrackFM: Far-out compiler support for a far memory world. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS*. 401–419.

[26] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A {Memory-Disaggregated} managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 261–280.

[27] Xi Wang, Bin Ma, Jongryool Kim, Byungil Koh, Hoshik Kim, and Dong Li. 2025. cMPI: Using CXL Memory Sharing for MPI One-Sided and Two-Sided Inter-Node Communications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2216–2232.

[28] Jianbo Wu, Jie Liu, Gokcen Kestor, Roberto Gioiosa, Dong Li, and Andres Marquez. 2024. Performance study of cxl memory topology. In *Proceedings of the International Symposium on Memory Systems*. 172–177.