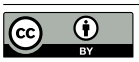
Beluga: A CXL-Based Memory Architecture for Scalable and Efficient LLM KVCache Management

XINJUN YANG, Alibaba Cloud Computing, USA
QINGDA HU*, Alibaba Cloud Computing, China
JUNRU LI*, Alibaba Cloud Computing, China
FEIFEI LI, Alibaba Cloud Computing, China
YICONG ZHU, Alibaba Cloud Computing, China
YUQI ZHOU, Alibaba Cloud Computing, China
QIURU LIN, Alibaba Cloud Computing, China
JIAN DAI, Alibaba Cloud Computing, China
YANG KONG, Alibaba Cloud Computing, China
JIAYU ZHANG, Alibaba Cloud Computing, China
GUOQIANG XU, Alibaba Cloud Computing, China
QIANG LIU, Alibaba Cloud Computing, China

The rapid increase in LLM model sizes and the growing demand for long-context inference have made memory a critical bottleneck in GPU-accelerated LLM serving. Although high-bandwidth memory (HBM) on GPUs offers fast access, its limited capacity necessitates reliance on host memory (CPU DRAM) to support large KVCache. However, the maximum DRAM capacity is constrained by the limited number of memory channels per CPU socket. To overcome this limitation, current systems often adopt RDMA-based disaggregated memory pools, which introduce significant challenges including high access latency, complex communication protocols, and synchronization overhead. Fortunately, the emerging CXL technology introduces new opportunities in KVCache design. In this paper, we propose *Beluga*, a novel memory architecture that enables GPUs and CPUs to access a shared, large-scale memory pool through CXL switches. By supporting native load/store access semantics over the CXL fabric, our design delivers near-local memory latency, while reducing programming complexity and minimizing synchronization overhead. We conduct a systematic characterization of CXL-based memory pool and propose a set of design guidelines. Based on *Beluga*, we design and implement *Beluga*-KVCache, a system tailored for managing the large-scale KVCache for LLM inference. *Beluga*-KVCache achieves an 89.6% reduction in Time-To-First-Token (TTFT) and 7.35× throughput improvement in vLLM compared to RDMA-based solutions. To the best of our knowledge, *Beluga* is the first system that enables GPUs to directly access large-scale memory pools through CXL switches (Marvell XConn XC50256), marking a significant step toward low-latency, shared access to vast memory resources by GPUs.

*Qingda Hu and Junru Li are the corresponding authors
{qingda.hqd, rusuo.ljr}@alibaba-inc.com

Authors' Contact Information: Xinjun Yang, Alibaba Cloud Computing, Sunnyvale, CA, USA; Qingda Hu, Alibaba Cloud Computing, Hangzhou, Zhejiang, China; Junru Li, Alibaba Cloud Computing, Beijing, China; Feifei Li, Alibaba Cloud Computing, Hangzhou, Zhejiang, China; Yicong Zhu, Alibaba Cloud Computing, Hangzhou, Zhejiang, China; Yuqi Zhou, Alibaba Cloud Computing, Beijing, China; Qiuru Lin, Alibaba Cloud Computing, Hangzhou, Zhejiang, China; Jian Dai, Alibaba Cloud Computing, Hangzhou, Zhejiang, China; Yang Kong, Alibaba Cloud Computing, Shanghai, China; Jiayu Zhang, Alibaba Cloud Computing, Shanghai, China; Guoqiang Xu, Alibaba Cloud Computing, Hangzhou, Zhejiang, China; Qiang Liu, Alibaba Cloud Computing, Shenzhen, Guangdong, China.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2836-6573/2026/2-ART13
https://doi.org/10.1145/3786627

CCS Concepts: • **Information systems** → **Cloud based storage**; • **Computer systems organization** → **Cloud computing**.

Additional Key Words and Phrases: Compute Express Link (CXL) Switch, Large language models (LLM), Disaggregated Shared Memory Pool, Key-Value Cache (KVCache)

ACM Reference Format:
Xinjun Yang, Qingda Hu, Junru Li, Feifei Li, Yicong Zhu, Yuqi Zhou, Qiuru Lin, Jian Dai, Yang Kong, Jiayu Zhang, Guoqiang Xu, and Qiang Liu. 2026. *Beluga*: A CXL-Based Memory Architecture for Scalable and Efficient LLM KVCache Management. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 13 (February 2026), 29 pages. https://doi.org/10.1145/3786627

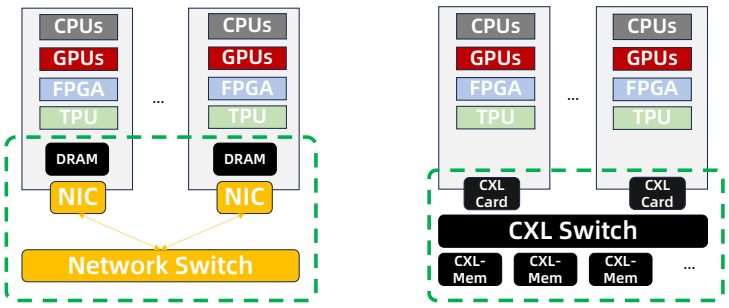
1 Introduction

Database vendors are rapidly integrating in-database (in-DB) LLM inference capabilities [23, 31, 46, 53, 73], as demonstrated by PolarDB [9, 16, 28, 62, 63], MindsDB [8], Aurora [6], and GaussDB [7]. These systems leverage GPU-CPU platforms to enable advanced applications, particularly Retrieval-Augmented Generation (RAG) based intelligent Q&A [49, 68] and natural language to SQL translation (NL2SQL) [44, 60], enabling interactive and intelligent data exploration. These applications impose significant demands on the underlying systems to efficiently process long and complex contexts. A critical optimization is reusing the model's internal representation of context—namely, the Key-Value (KV) Cache—which avoids redundant and costly recomputation of shared contexts [11, 56]. However, KVCache maintenance demands significant memory resources [26, 43]. For instance, 50M tokens in Kimi [51] requires approximately 20TB of DRAM for its KVCache to achieve maximal cache hit ratios. Given ever-increasing inference memory demands, local GPU clusters are becoming financially and technically infeasible to meet the rising requirements of LLM inference. Consequently, remote resource pooling architectures centered around KVCache [17], such as Dynamo [50] and MoonCake [51], are gaining considerable traction and real-world adoption. By offloading the KVCache to high-capacity remote memory pools via Remote Direct Memory Access (RDMA), these architectures achieve enhanced memory scalability and system efficiency, as shown in Figure 1a.

However, RDMA was fundamentally designed as a networking protocol, not a memory bus, leading to some limitations for KVCache offloading. On the data path, it requires data movement through host memory, which in turn incurs extra latency. On the control path, managing RDMA verbs involves complex programming, introduces overhead for preparing requests and polling for completions, and necessitates costly cross-component synchronization. Furthermore, the non-uniform hierarchy between local and remote memory creates a challenging scheduling problem, requiring a balance between computing resources and KVCache locality [50, 51]. These combined inefficiencies in RDMA memory pool significantly limit the potential benefits of KVCache offloading.

The emergence of the Compute Express Link (CXL) standard presents an opportunity to design high-performance memory pools. By providing a direct, low-latency, load/store memory interface, CXL allows CPU/GPUs to access remote memory with an efficiency that approaches local memory access, as shown in Figure 1b. This memory-semantic interface directly addresses the core problems of RDMA: it simplifies the data path by eliminating extra copies, and streamlines the control path by removing the need for complex network programming and explicit synchronization.

While the potential of CXL is widely discussed, prior work has focused on memory expansion with CXL 1.1 devices or FPGA-based simulations. A practical, large-scale evaluation of CXL 2.0 memory pools with multiple hosts and devices has not been possible due to the lack of commercial hardware. The recent availability of the Marvell XConn XC50256 CXL 2.0 switch [48, 57] now enables such a study. This paper presents *Beluga*, a system built on this new CXL 2.0 switch hardware. We first provide a detailed characterization of *Beluga*, measuring its performance with both CPU and



(a) RDMA-based MemPool **(b)** *Beluga*

Fig. 1. Overview of RDMA/CXL memory pools.

GPU workloads to understand its fundamental behavior and propose design guidelines. Building on these findings, we then implement *Beluga*-KVCache and integrate it into the vLLM inference engine. This demonstrates the effectiveness of our proposed optimizations for KVCache management, a critical component in LLM serving.

We summarize our main contributions as follows:

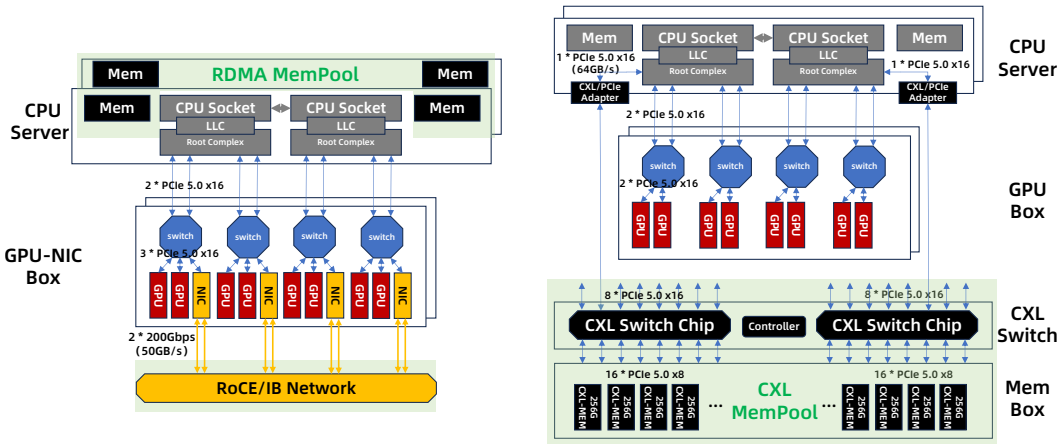
- We present *Beluga*, a memory system for GPU clusters that leverages CXL 2.0 switches to provide access to a disaggregated, shared memory pool. To the best of our knowledge, *Beluga* is the first system to integrate GPU clusters with CXL switching infrastructure, demonstrating the feasibility of extending GPU cluster memory capacity through CXL-based interconnects.
- We conduct comprehensive performance evaluations of *Beluga* on GPU clusters accessing memory via a CXL 2.0 switch, and we introduce a set of system-level optimizations and guidelines to address key performance bottlenecks. These optimizations enable *Beluga* to effectively overcome the cache coherence limitations inherent in a multi-host CXL 2.0 environment. Compared to RDMA-based disaggregated memory systems, *Beluga* achieves up to 7.0× lower latency for write operations and up to 6.3× lower latency for read operations. Additionally, *Beluga* delivers scalable memory bandwidth, making it suitable for high-performance, data-intensive GPU workloads.
- We propose *Beluga*-KVCache, a KVCache management system built on the CXL memory pool. We demonstrate its effectiveness by integrating it into vLLM [33], the open-source inference library, delivering efficient KVCache read/write, CXL-based RPCs, and a simplified scheduler. Our evaluation shows that *Beluga*-KVCache improves LLM inference throughput by up to 4.79× over MoonCake, the state-of-the-art RDMA-based solution.

This paper is organized as follows. Section 2 presents the background of this paper. Section 3 reviews RDMA-based memory pooling and presents our motivation, then Section 4 introduces the architectural design of our system *Beluga* and its inherent advantages. Section 5 details its characterization and optimizations, while Section 6 presents our KVCache management system. Section 7 then presents an end-to-end evaluation. Section 8 discusses the broader implications of our work, and Section 9 reviews the related work.

2 Background

2.1 LLM Inference and KVCache

Large Language Models (LLMs), powered by the Transformer architecture, have become a foundational workload for modern applications [10, 20, 61]. Their inference process typically consists of two phases: prefill and decode [72]. The compute-intensive prefill phase processes the input prompt in parallel to generate the initial output token. Subsequently, the decode phase generates



(a) Clusters with an RDMA-based memory pool **(b)** Clusters with a CXL-based memory pool (*Beluga*)

Fig. 2. Hardware architectures of GPU clusters with (a) RDMA / (b) CXL memory pool. The real hardware is shown in Figure 3.

the output sequence autoregressively, where each new token's generation depends on all preceding tokens and their corresponding KV activations.

The KVCache, which stores intermediate KV activations from previous tokens, serves as an essential space-for-time optimization [33, 70]. Eliminating redundant computation is essential for achieving low-latency token generation. This technique is widely used in various scenarios: preserving history in multi-turn dialogues, enabling inter-request sharing for fixed contexts (e.g., system prompts), and amortizing the cost of processing long documents in RAG [21]. These scenarios reveal a new direction of data management: designing efficient storage systems to offload, reuse, and share KVCache [27, 34, 43, 51, 64]. Such systems have the following four fundamental requirements:

Scalable capacity. The memory footprint of the KVCache increases linearly with context length, frequently consuming gigabytes of memory per request. The disaggregated system must therefore provide a large, elastic memory pool capable of scaling beyond the limitations of a single node's on-package memory.

Efficient sharing. Optimizing resource utilization and throughput in GPU clusters necessitates a disaggregated architecture, in which all servers access a unified KVCache memory pool.

Low-latency access. The primary benefit of employing a cached prefix is to eliminate the overhead of recomputation, which directly enhances metrics like Time-To-First-Token (TTFT). Consequently, the end-to-end latency for KVCache retrieval from the memory pool must be substantially lower than its recomputation latency.

High aggregate throughput. The inherent parallelism within a multi-GPU server necessitates a high-throughput requirement for the memory system. Insufficient aggregate bandwidth will lead to GPU stalling, directly constraining system throughput and potentially rendering recomputation a more efficient alternative.

2.2 Remote Direct Memory Access (RDMA)

RDMA enables a Network Interface Controller (NIC) to directly access memory on a remote host, bypassing the remote CPU and the OS kernel [29, 42, 58, 67, 75]. By minimizing network stack overhead, RDMA achieves high bandwidth and low CPU utilization, making it a widely adopted

interconnect in modern datacenters. Most RDMA memory pools rely on one-sided communication primitives, which facilitate direct memory-to-memory data transfers between nodes. This architectural design eliminates traditional overheads, such as kernel context switches and data copies. Key primitives include RDMA Read and RDMA Write for direct data retrieval and placement, and RDMA Atomic Operations (e.g., CAS, FAA) for executing atomic read-modify-write on remote memory. RDMA's one-sided primitives provide an ultra-low latency, high-throughput interconnect, forming the foundation for building high-performance memory pools.

2.3 Compute Express Link (CXL)

CXL is an emerging interconnect protocol designed to facilitate efficient memory sharing between host CPUs and accelerators. It defines three distinct protocols: CXL.IO, dedicated to CXL device management and DMA data transfers; CXL.cache, for coherent device access to host memory; and CXL.mem, enabling host load/store access to device-attached memory for expansion and pooling. The standard's rapid evolution has progressively enhanced system capabilities: transitioning from single-device attachment (CXL 1.1), to switch-based memory pooling (CXL 2.0), and most recently to multi-tier switching with cache coherency support (CXL 3.0), thereby enabling more flexible and powerful heterogeneous systems.

This paper focuses primarily on memory expansion solutions leveraging the CXL.mem interface. Currently, the design and production of hardware supporting CXL.mem remains in its early stages. Most existing hardware only supports CXL 1.1, providing memory expansion capabilities for single servers. Although several research efforts have developed FPGA-based prototypes supporting CXL 2.0, their capabilities are constrained in both device count and memory capacity [24, 35, 40, 59, 71]. Marvell XConn [48, 57, 63] has produced the first commercial CXL 2.0 switch, providing CXL.mem interfaces with support for *256 lanes* and concurrent access from multiple hosts, delivering minimal 64-byte I/O latency of *~750ns* and maximum switching capacity of *2TB/s*. The introduction of CXL 2.0 switches fundamentally elevates the protocol beyond the single-node memory expansion of CXL 1.1, thereby enabling the creation of a large-scale shared memory pool.

3 RDMA-Based MemPool

The conventional approach for KVCache management, shown in Figure 2a, uses an RDMA-based memory pool to aggregate DRAM across the cluster for storing the offloaded KVCache. The RDMA architecture offers two main benefits: scalable memory capacity and data sharing across servers. However, this design suffers from fundamental limitations, including performance bottlenecks from extra data copies, expensive control-path synchronization and high architectural complexity (§3).

3.1 A Simple Review of RDMA-Based MemPool

Figure 2a shows a multi-GPU cluster interconnected by an RDMA network. This architecture is common in modern data centers [38]. Each server in the cluster contains a host CPU, multiple GPUs, and RDMA NICs. These components are interconnected by a fabric of PCIe switches. A typical 8-GPU server, for example, uses four such switches. Each switch provides five PCIe x16 links. Three links connect to two GPUs and an RDMA NIC. The remaining two serve as uplinks to a host CPU socket. The dedicated RDMA NICs on each server provide up to 1.6 Tbps of inter-server bandwidth. This high-speed interconnect allows each server to contribute local DRAM (e.g., 2 TB), forming a distributed memory pool shared across the entire cluster. To leverage this shared memory pool, LLM inference frameworks access the distributed memory pool using two main approaches: a CPU-driven approach or a GPU-driven one. The key distinction is which processor orchestrates the data transfers.

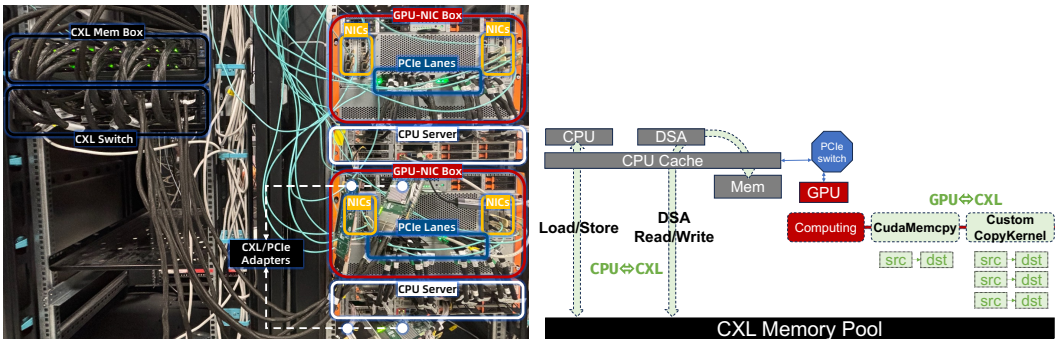
CPU-driven RDMA memory pool access. This approach is adopted by prominent frameworks such as vLLM [33], MoonCake [51], and LMCache [43]. In this model, the host CPU issues RDMA commands directly to the NIC to fetch remote data. The process of storing the KVCache involves two steps. First, data is copied from the GPU to a "bounce buffer" in host memory. The CPU then initiates an RDMA transfer to move data from this buffer to the remote pool. Loading data follows the reverse path: from the remote pool, through the bounce buffer, and finally back to the GPU.
GPU-driven RDMA memory pool access. Alternatively, a GPU-driven approach uses GPUDirect RDMA (GDR) [5]. This technology allows the GPU to issue commands directly to the NIC, completely bypassing the host CPU. In this model, a dedicated GPU kernel manages all network communication. To store the KVCache, this kernel issues an RDMA Write and polls for its completion. To read the cache, it issues an RDMA Read and polls for a work completion (WC) signal. Once the data arrives, the polling kernel must synchronize with the main computation stream to proceed with the transformer computation. Note that GDR is a feature exclusive to expensive datacenter-class GPUs like the NVIDIA A100/H100. It is not available on widely-used consumer GPUs such as the RTX 4090 [2], restricting its practical use.

3.2 The Inefficiencies in RDMA-Based MemPool

While RDMA offers a high-throughput interconnect, it is not a silver bullet for memory disaggregation. Relying on RDMA forces fundamental architectural trade-offs that manifest as critical performance inefficiencies and prohibitive system complexity.
Performance inefficiencies stem from three main sources:

- *Indirect host-staged data path.* The CPU-driven model subverts the "direct" nature of RDMA by forcing all data through a "bounce buffer" in host memory. This creates a costly data path (GPU → Host DRAM → Remote Memory for writes, and the reverse for reads). This mandatory data staging introduces significant latency for data transfers between GPU and memory pool.
- *Complex and inefficient control path.* A fundamental limitation of RDMA-based offloading is the costly synchronization required between different components. The CPU-driven model, for example, requires CPU-GPU coordination. Similarly, the GPU-driven model needs synchronization between its polling and compute Streaming Multiprocessors (SMs). In either case, this cross-component coordination imposes a significant latency penalty. This stands in sharp contrast to the seamless execution within a single, unified GPU stream [52]. Our microbenchmark on an H20 GPU reveals the severity of this overhead. A 16 KB transfer takes 10.55 µs in total, but the actual data movement accounts for only 2.68 µs. The remaining 8 µs (nearly 75% of the total latency) is attributable to synchronization overhead. This cost stems from launching the kernel or waiting for its completion. In fact, this synchronization penalty alone is almost 3x the duration of the data transfer itself. Futhermore, unlike simple bulk transfers, KVCache reads and writes are complex due to differing data layouts between the GPU and the memory pool. For example, a single KVCache block in Qwen-32B (GQA) requires 128 non-contiguous 20KB data transfers. This introduces extra overheads in the control path.
- *Inefficient resource utilization.* RDMA's polling-based control path inherently leads to resource waste. In the CPU-driven model, a host thread consumes entire CPU cores simply to poll for network completions. This problem is significantly exacerbated in the GPU-driven model. Here, a dedicated polling kernel occupies valuable SMs. These SMs are the GPU's most critical and limited resource. This occupation creates direct contention with inference tasks and also complicates stream synchronization [38, 52].

System complexity. Beyond performance issues, RDMA-based approaches also suffer from significant system complexity, including:



(a) The actual deployment of *Beluga* **(b)** Data access interfaces of *Beluga*

Fig. 3. The actual deployment of *Beluga* and the data access interfaces of *Beluga*.

- *Non-trivial development complexity*. RDMA itself presents prohibitive programming complexity. Instead of simple memory operations, developers are burdened with low-level network management. The GPU-driven model further forces developers to manage the RDMA stack within CUDA and orchestrate synchronization between polling and computation streams.
- *Optimization complexity for tiered memory*. The overheads of RDMA-based memory access lead to substantial latency penalties for remote KVCache operations. This latency may overshadow the computational savings from cache reuse, ultimately negating the entire benefit of offloading the KVCache. To mitigate this performance penalty, developers must manually implement critical optimizations like request batching [18, 32] and ensure data consistency for one-sided RDMA through careful management of QP ordering semantics and software checks [19, 75]. This further forces systems to adopt complex, cache-aware scheduling policies that tightly couple scheduling decisions to KVCache locality. This locality-driven approach creates significant drawbacks, including high scheduling complexity, poor load balancing, and increased maintenance overhead [50, 51, 76].

4 *Beluga*: A CXL-Based Memory Architecture

To overcome RDMA's limitations, we propose *Beluga*, an architecture that leverages the industry's first production-ready CXL switch (Marvell XConn XC50256 [48, 57]) to build a scalable, shared memory pool. This allows GPUs to access pooled memory with simple load/store operations, solving the performance bottlenecks and programming complexity of the RDMA approach. In this sections, we first give an overview of *Beluga*'s architecture and then show its inhenrent advantages.

4.1 Overview

As shown in Figure 2b, this architecture replaces the four dedicated RDMA NICs with two PCIe/CXL adapters, and the hardware deployment of *Beluga* is shown in Figure 3. Each server has two CPU sockets (NUMA architecture), and each socket connect to the CXL switch via a PCIe 5.0 x16 PCIe/CXL adapter. The CXL memory pool itself comprises a switch node and a separate memory box. At its core, the switch is equipped with two chips (Marvell XConn XC50256). Each chip provides 2 TB/s of forwarding capacity over 256 PCIe 5.0 lanes. These lanes are typically split evenly between CXL memory devices and compute servers. The underlying CXL switch connects up to 16 servers to an 8 TB memory pool with 1 TB/s of total bandwidth. This connectivity enables *Beluga* to support concurrent multi-host access through its internal address mapping and forwarding logic.

4.2 The Inherent Advantages of *Beluga*

By altering the memory access paradigm from a network protocol (RDMA) to a memory-semantic interface (CXL), *Beluga* offers several straightforward advantages:

Improved performance. As shown in 3b, *Beluga* provides standard data access interfaces for both CPUs and GPUs. For the CPU, *Beluga* supports (1) direct memory access via `load/store` instructions and (2) hardware-accelerated transfers using the Intel Data Streaming Accelerator (DSA), a DMA engine available in Intel's Sapphire Rapids CPUs. For the GPU, it supports (1) direct peer-to-peer (P2P) transfers via the `cudaMemcpy` API and (2) fine-grained, non-contiguous access through custom CUDA kernels. These methods provide a straightforward performance benefit compared with RDMA, in both data and control paths.

- *In the data path*, *Beluga* allows the GPU to access the global memory pool directly, eliminating the multi-step data path and bounce buffers required by CPU-driven RDMA, which significantly reduces latency.
- *In the control path*, the data transfer kernels in *Beluga* integrate seamlessly into the GPU's native CUDA stream. This unification of control eliminates the expensive cross-component synchronization inherent in both CPU-driven and GPU-driven RDMA, thereby removing the overheads associated with external CPU coordination or internal GPU polling.

System simplification. Beyond performance gains, *Beluga* also introduces fundamental system-level simplifications. These advantages manifest in a more accessible programming model, streamlined memory management, and reduced hardware cost.

- *Programming model.* The data access interfaces are analogous to those used for local DRAM. Consequently, developers are freed from managing the low-level network stack, complex work-request preparations, and performance optimizations required by RDMA-based memory pools. Furthermore, the elimination of cross-component synchronization overhead for GPUs simplifies the coordination between data transfers and computation.
- *Memory management. Beluga* provides a unified address space with simplified space control. During startup, the BIOS on each host detects the attached CXL devices and reserves a contiguous physical address space for them. *Beluga* then leverages this foundation by having each host manage the CXL memory pool in Direct Access (DAX) mode. In this mode, the CXL memory is exposed as a block device, allowing user-space processes to use the `mmap()` to map the entire memory region directly into their virtual address spaces. This direct-mapping mechanism is the key to both resource partitioning and data sharing. Applications built on *Beluga* can either assign different memory offsets to different hosts to achieve logical resource partitioning, or have multiple hosts map the same region to enable efficient data sharing. Crucially, this approach provides a unified memory view across all servers, facilitating simple and efficient memory management.
- *Hardware cost.* Further, high-speed networking hardware, such as 400Gbps NIC, is often over-provisioned for the typical bandwidth demands of LLM inference [51]. Replacing expensive RDMA NICs with more cost-effective CXL components can therefore significantly reduce the total cost of ownership. As shown in Table 1, the CXL-based approach reduces the cost of host interface cards and switches at the device level [1]. For memory devices, unlike traditional DIMM memory in the host, CXL memory devices can use lower-density chips, reducing the cost per GB. At the system level, CXL memory pools separate memory from CPU resources, allowing multiple servers to share memory and improve utilization in cloud environments.

In summary, for a single-rack memory pool, CXL demonstrates clear advantages over RDMA by offering an optimal balance among performance, cost-effectiveness, and resource utilization.

[1] Marvell XConn CXL Switch: B1 sample price, and final price subject to change.

Table 1. Hardware Cost Analysis.

	RDMA-based	**CXL-based**
Interface on Host	CX-7 (2×200Gbps)	PCIe/CXL Adapter
PCIe Lanes	x16	x16
Price	$1,745	$210
Switch	Mellanox RoCE Switch	Marvell XConn CXL Switch
Ports	40×200Gbps	32×PCIe5.0 x16
Price	$16,000	$5,800
$ / (64GB/s)	$800	$218.75

5 Characterization and Optimization of *Beluga*

While CXL provides a simple memory-like interface, the poorly understood performance of its CXL 2.0 hardware, particularly for direct GPU-to-memory access, hinders the development of software that can unlock its full potential. To address this gap, this section presents a comprehensive performance analysis of all data paths in *Beluga*, yielding a set of practical optimizations summarized in Table 3. The analysis proceeds in three parts: First, we design and evaluate three software-based coherence methods, providing clear optimization guidelines for developers (§5.1). Second, we analyze different I/O transfer methods and identify the optimal approach for latency-sensitive workloads (§5.2). Third, we evaluate and identify the bandwidth bottleneck in *Beluga* and propose two effective solutions to scale performance for bandwidth-intensive workloads (§5.3). All experiments are validated on a platform using commercially available hardware, with detailed specifications in Table 2.

Table 2. Experimental setup.

OS	Ubuntu 22.04, kernel 6.2.0-1015
CPU	2 × Intel(R) Xeon(R) Platinum 8575C
L3 Cache	640 MiB (320 MiB per CPU)
DRAM	2 TB (32 × DDR5 4800 MT/s 64 GB)
PCIe	4 × PCIe Switch, 5.0
GPU	8 × H20 (96 GB)
NICs / Server	4 × ConnectX-7 (Dual-port 200Gbps)
RDMA MemPool	4 TB (2 × GPU Servers)
PCIe/CXL Adapter / Server	2 × PCIe 5.0 x16
CXL MemPool	8 TB (32 × DDR5, 4800 MT/s, 256 GB)

5.1 Data Sharing over Non-Coherent CXL

Goals and methodology. While CXL.cache and CXL.mem in CXL 2.0 support coherent memory sharing between host CPUs and devices like accelerators or memory expanders, they do not support

Table 3. Performance optimizations in *Beluga*.

Aspect	**Optimizations**	**Applicable To**
Cache Coherency (§5.1)	**O1.** For CPU access, use `ntstore` for writes and invalidate CPU cache before reads.	CPU `load/store`
	O2. For CPU DSA, set memory as Uncachable.	CPU DSA
	O3. For GPU access, set memory as Uncachable and disable DDIO.	GPU
Latency (§5.2)	**O4.** Use direct `load/store` for small I/Os (< 4 KB) and use DSA for larger transfers.	CPU
	O5. Launch kernels asynchronously using CUDA streams to hide the launch latency.	GPU
	O6. For GPU transfers (< 24 KB) on Uncachable memory, use customized copy kernel.	GPU
Bandwidth (§5.3)	**O7.** Support direct GPU-to-CXL switch connections in future architecture.	GPU
	O8. Use more PCIe/CXL adapters for scalable bandwidth.	-
	O9. Interleave data across multiple CXL memory devices.	-

cache coherence across multiple host CPUs. While CXL switches present a logically unified memory address space to host processors, each computing node maintains an independent cache hierarchy (L1/L2/L3) that operates in isolation from the other nodes. As a result, a write by one host remains confined to its local cache and is not automatically propagated to remote hosts. In the KVCache, this deficiency leads to inaccurate, inconsistent, or incoherent output. This occurs when one GPU reads a stale cache entry that is being concurrently updated by another GPU. Without hardware-supported multi-host coherence, the burden of maintaining data integrity shifts to software. This necessitates explicit cross-node synchronization protocols to coordinate all cache state transitions.

Our system focuses on optimizing KVCache sharing mechanisms, enabling a single writer to insert a KVCache block while allowing multiple readers to access it, thereby eliminating redundant computations. Targeting this scenario, we propose three software-managed methods to enforce consistency for the writer-side and two methods for the reader-side. We then conduct a series of experiments to measure the latency and to select the optimal method for different access operations.

Writer: ensuring data reaches CXL memory. To ensure a host's writes are visible to other hosts, the data must be flushed from its private cache hierarchy to the CXL memory. There are three methods to achieve this:

- *Static uncacheable configuration.* A straightforward approach for enforcing memory coherence in CXL-based systems is to configure the CXL memory regions as uncacheable, by setting the corresponding memory type in the Memory Type Range Registers (MTRRs) [4]. Consequently, any CPU-initiated write, whether via a standard `store` instruction or a DSA transfer, bypasses the host's cache hierarchy and is sent directly to the *Beluga* CXL memory pool. Symmetrically, for Device-to-Host (D2H) memcpy from GPU, we disable Data Direct I/O (DDIO) [3]. By default, DDIO directs inbound traffic from devices like GPUs and NICs into the CPU's Last-Level Cache (LLC) for high performance. Disabling DDIO alters this behavior, forcing D2H memcpy traffic to bypass the LLC and write directly into the CXL memory pool.
- *Fine-grained cache flushing after write.* This approach caches data but relies on software to explicitly flush modified cache lines to memory using CPU instructions like `CLFLUSH`, `CLFLUSHOPT` and `CLWB` [1]. However, these methods impose high instruction overhead, which scales linearly with the data size, as it uses per-cache-line operations.
- *Fine-grained bypassing-cache write without cache flushing.* To eliminate the explicit flushing, we can leverage some bypassing-cache instructions. CPU-originated writes can use non-temporal store instructions (`ntstore`),which bypasses the cache hierarchy and writes data into the CXL memory pool directly. Similarly, Intel DSA provides a cache bypass flag, enabling DMA operations to write their payloads directly to memory without populating the CPU Cache.

Reader: ensuring fresh data from CXL memory. Similarly, to prevent reading stale data in local cache, the system should either configure the memory region as uncacheable or explicitly invalidate the target cache lines before the read operation.

- *Static uncacheable configuration.* For CPU reads, the uncacheable attribute ensures correctness by forcing every load to access the remote CXL memory pool. Correct GPU access to CXL memory requires the uncacheable attribute, even for device-initiated transfers like `cudaMemcpy`. Although the GPU operation itself does not populate the CPU cache, concurrent or prior CPU activity can create cached copies of the data. Without this attribute, the GPU might read stale data from the CPU cache, leading to data inconsistency.
- *Fine-grained flushing cache before read.* This approach ensures read correctness by proactively invalidating stale cache lines with `CLFLUSH` instructions before the read operation. Note that the `CLWB` does not work for this purpose. These flush instructions suffer from high overhead as they operate on a per-cache-line granularity.

Table 4. Latency of cache coherency methods (μs). Bold entries highlight the optimal mechanism for each operations. The result of custom kernel includes the time of kernel launch. (**Exp #1**)

Write Direction	**CPU ⇒ CXL**		**GPU ⇒ CXL**
Operation (16 KB)	Store	DSA Write	Custom Kernel
UC Mem / Disable DDIO	281.56	**1.69**	**9.14**
CLFLUSH after Write	8.50	3.64	11.06
Bypassing-Cache Write	**2.41**	**1.76**	-
Read Direction	**CPU ⇐ CXL**		**GPU ⇐ CXL**
Operation (16 KB)	Load	DSA Read	Custom Kernel
UC Mem	166.49	**2.12**	**10.55**
CLFLUSH before Read	**5.98**	4.84	16.81

The above designs in writer or reader are simple: setting up cache states and disabling DDIO are one-time steps, and cache flushing is as simple as read/write. In contrast, achieving cache coherence in RDMA requires complex management of asynchronous work requests and completions ordering.
(Exp #1) Performance characterization. We evaluate the latency of 16 KB read/write and the results are shown in Table 4. For write operations, the results reveal a clear performance hierarchy. For CPU-initiated store, non-temporal stores (`ntstore`) are the most efficient method, achieving a latency of 2.41 μs. They bypass the cache and also avoids the explicit flush overhead. In contrast, standard writes followed by CLFLUSH-family instructions are slower (8.50 μs). Using uncacheable memory is prohibitively slow (281.56 μs), as each `store` instruction stalls the CPU pipeline while waiting for the entire CXL access to complete. For the DSA, both using uncacheable memory and the cache-bypassing flag yield nearly identical, top-tier performance, as the DSA is not hindered by CPU pipeline stalls associated with uncacheable memory access. For GPU D2H transfers, disabling DDIO to bypass the cache is more effective (9.14μs) than relying on a subsequent, slow CPU flush (11.06μs). For read operation, we observe that CPU `load`s from uncacheable memory are prohibitively slow (166.49 μs). Consequently, the only viable method for CPU `load` is to flush the cache before the read, which achieves a much lower latency of 5.98 μs. Similar to its write performance, the CPU DSA read and GPU H2D copy perform better when reading directly from uncacheable memory.
Optimizations. Current CXL 2.0 switches do not support host-to-host hardware cache coherency, a limitation that requires software-managed cache coherency. Optimal performance for data sharing is achieved by applying specific cache management strategies based on the operation's initiator. There are three design hints: for CPU store/load, non-temporal stores should be used for writes, and a CLFLUSH should precede loads **(O1)**. For the Intel DSA engine, operating on uncacheable memory is the best practice for both reads and writes **(O2)**. Finally, for GPU memcpy operations, disabling DDIO while utilizing uncacheable memory offers the highest efficiency **(O3)**. Looking ahead, transparent cross-host cache coherence via CXL 3.0 switches could remove the need for software-based synchronization. But CXL 3.0 coherence is still evolving; its region size and guarantees are unclear and may be limited due to cost and complexity. Our software-based designs are therefore necessary for non-coherent regions. When hardware coherence is available for small regions, we can use it to further simplify and accelerate our software protocol.

5.2 Latency Optimization

Goals and methodology. The performance of CXL memory access for both CPUs and GPUs depends heavily on the specific operation and access size. We therefore begin by characterizing the latency of four fundamental operations in *Beluga* to identify their optimal use cases. Furthermore, we compare *Beluga* against local memory and RDMA-based memory pools as baselines to highlight the scenarios where it offers the most significant performance benefits.

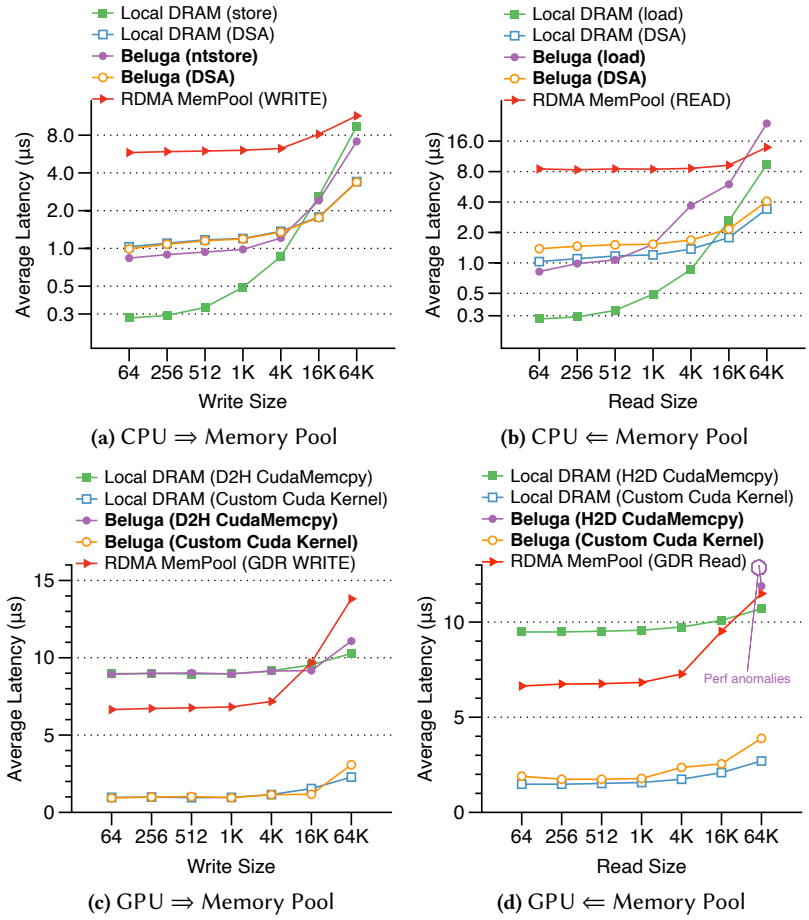


Fig. 4. Latency between CPU/GPU ⇔ remote memory pool (**Exp #2**). Local memory and RDMA memory are included as baselines. This experiment not only demonstrates the performance differences among various CXL access methods but also reveals that CXL memory pooling achieves latency characteristics comparable to local memory pools, significantly outperforming RDMA-based memory pooling

(Exp #2) Performance characterization. Figure 4 presents the I/O operations between CPU/GPU and memory pool. Unless otherwise specified, all latency benchmarks use a queue depth of one (QD=1) to measure single-request latency. Our key observations are as follows:

First, for GPU access, direct CXL-to-GPU data transfers are highly competitive with traditional CPU-to-GPU paths (Figure.4c and 4d), establishing the CXL pool as a viable data source for the GPU. We find that at a 64 KB transfer size, the latency of a CXL-to-GPU copy is 11.73 µs, which is remarkably close to the 10.32 µs latency of a conventional CPU-to-GPU copy. This demonstrates that CXL memory can function as a primary data source for GPUs in the same manner as CPU memory.

Second, a clear performance trade-off exists between CPU-based copies and DMA-based transfers for moving data to CXL (Figure.4a and 4b). For I/O sizes up to 4 KB, direct CPU load/store operations exhibit lower latency than offloading the transfer to the DMA engine. The benefit of DMA parallelism

begins to outweigh its setup overhead for transfers larger than 4 KB. The performance crossover occurs at 16 KB, where the Intel DMA engine DSA outperforms the CPU-based copy.

Third, our analysis shows that the poor latency in CUDA Memcpy Kernel is dominated by software overhead, rather than the physical data movement time (Figure.4c and 4d). This overhead stems primarily from the kernel launch latency within the GPU driver stack. For instance, while the CUDA Memcpy Kernel for a 16 KB H2D (Host to Device or CPU memory to GPU memory) transfer is 10.55 µs, profiling with NVIDIA Nsight Systems (nsys) reveals that the actual data transfer on the GPU completes in only 2.68μs. Therefore, we recommend using custom data copy kernels to combine multiple copy tasks in a single kernel. This method avoids kernel launch and CPU synchronization overhead. As a result, it achieves better latency compared to both CUDA Memcpy Kernel and GDR.

Finally, as shown in Figure.4d, we observe that the standard cudaMemcpy for H2D transfers: when the source memory is Uncachable, performance degrades dramatically for transfers smaller than 24 KB, taking approximately 1.23 ms (values omitted from the Figure.4d for clarity of scale). We hypothesize that the CUDA runtime leverages CPU-based instructions to optimize small transfers, a strategy that is ill-suited for Uncachable memory. Consequently, to achieve efficient H2D data movement for transfers under 24 KB from Uncachable CXL memory, it is necessary to use a custom CUDA memcpy kernel.

Optimizations. *Beluga* delivers latency that is competitive with local DRAM. This key characteristic enables the creation of a flat, unified memory pool, with the potential to eliminate the need for complex, multi-tiered memory hierarchies. Furthermore, we propose the following three optimizations to leverage this low latency. First, the CPU should use direct load/store for small I/Os (< 4 KB) and DSA for larger transfers **(O4)**. Second, when moving data between the GPU and CXL memory pool, tasks should be combined into a single kernel to hide the kernel launch latency **(O5)**. Third, for data transfers up to 24 KB from CXL memory to the GPU, a custom-implemented CUDA memcpy kernel is recommended **(O6)**.

5.3 Bandwidth Optimization

Goals and methodology. We observe two performance anomalies in *Beluga* when we use a single PCIe/CXL Adapter (16 lanes). First, it exhibits asymmetric read/write performance. The read bandwidth from CPU to CXL memory pool reaches the expected 46.2 GB/s, which is comparable to that of RDMA. However, the write throughput is limited to 33 GB/s. Second, we observe a significant performance degradation during GPU access to the CXL memory pool, with throughput dropping to 26 GB/s. This is substantially lower than the bandwidth of CXL memory controllers and the GPU's own PCIe bandwidth (55.4 GB/s). The data path from a CPU/GPU to the CXL memory pool is complex, traversing components such as the PCIe Switch, Root Complex, and CXL Switch. Any of these components can become a performance bottleneck, limiting the end-to-end bandwidth. Our approach is to first perform a bottleneck analysis of this path and then present targeted optimizations.

Performance characterization. Through the systematic empirical analysis, we identify the CPU's Root Complex (RC) as the primary performance bottleneck in CXL memory pool architectures. To verify this conclusion, we construct a dedicated micro-benchmark (Figure 5). It measures the bandwidth between a GPU and a NIC that are attached to different PCIe switches, forcing the data path to go through the CPU's Root Complex. The results of this experiment precisely match the performance with CXL memory pool: the PCIe P2P write (i.e., NIC⇒GPU) bandwidth was 33 GB/s, and the P2P read (i.e., GPU⇒NIC) bandwidth was limited to 23 GB/s. This indicates that the bottleneck may not be in the CXL memory devices or the CXL links themselves. Instead, performance is limited by the underlying peer-to-peer capabilities of the Root Complex. To further

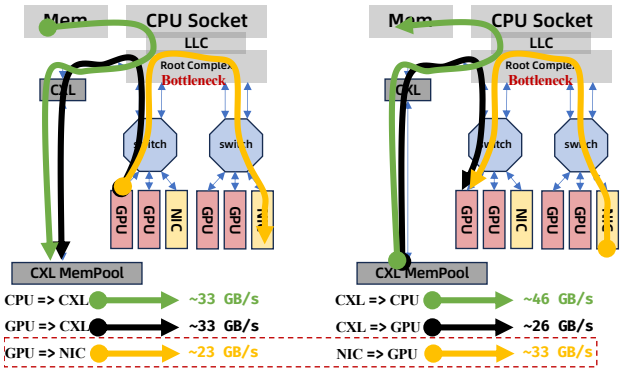


Fig. 5. Bandwidth for different CXL access paths.

understand this bottleneck, we conduct additional experiments with multiple GPU-NIC pairs. The aggregate bandwidth scales linearly with the number of pairs: two pairs achieve 46 GB/s, approximately double the throughput of a single pair. It suggests that the limitation lies in per-lane or per-link resources within the Root Complex rather than an overall throughput cap.

Furthermore, we observe that the memory devices themselves can be another bottleneck. Each device supports a bandwidth of 22.5 GB/s. If all workloads from a server are directed to a single device, the bandwidth will be limited by that device. Therefore, software running on *Beluga* should distribute data across different memory devices to avoid the single-device bottleneck. Our current implementation relies on software-based memory interleaving at a 2MB granularity. The upcoming Intel Granite Rapids (6th-generation Xeon) processors provide hardware support for CXL device interleaving, offering flexible configurations from 256B chunks up to eight-way interleaving.

Optimizations. Based on our observations, we further propose three optimization strategies for bandwidth-sensitive scenarios. First, future architecture should support direct GPU-to-CXL switch connections. As such, *Beluga* is able to bypass the RC, avoiding the major bottleneck between the GPU and CXL memory **(O7)**. Second, bandwidth of a single PCIe/CXL adapter is limited to PCIe5.0 x16. And, for higher bandwidth, the number of PCIe/CXL adapters should be scale with application requirements **(O8)**. Third, *Beluga* interleaves data across multiple CXL memory devices. This strategy parallelizes access and helps maximize the aggregated throughput **(O9)**.

5.4 Performance under Complex Workloads

To evaluate *Beluga* performance in complex scenarios, we conduct additional benchmarks focusing on concurrent and conflicting memory access patterns.

(Exp #3) Performance under skewed memory access. We measure bandwidth, median latency, and p99 tail latency during concurrent memory access, and the results and configurations are shown in Figure 6. For local DRAM and CXL access, we use ntstore for write operations and clflush before read operations. This experiment also include the performance with 2 MB interleaving size and without memory interleaving, which demonstrates the necessity of enabling memory interleaving in skewed workloads. We have the following two observations. First, CXL demonstrates superior performance over RDMA. For median latency, CXL shows only 10.2%~13.3% of RDMA latency in 64B operations, and 39.5%~56.2% in 16KB operations. Notably, CXL achieves comparable write latency to local DRAM in 16KB write operation. Second, CXL without memory interleaving exhibits lower bandwidth and higher latency under 16KB skewed workloads. This occurs because the first memory device in the CXL memory box becomes a bottleneck, introducing queuing latency during concurrent access.

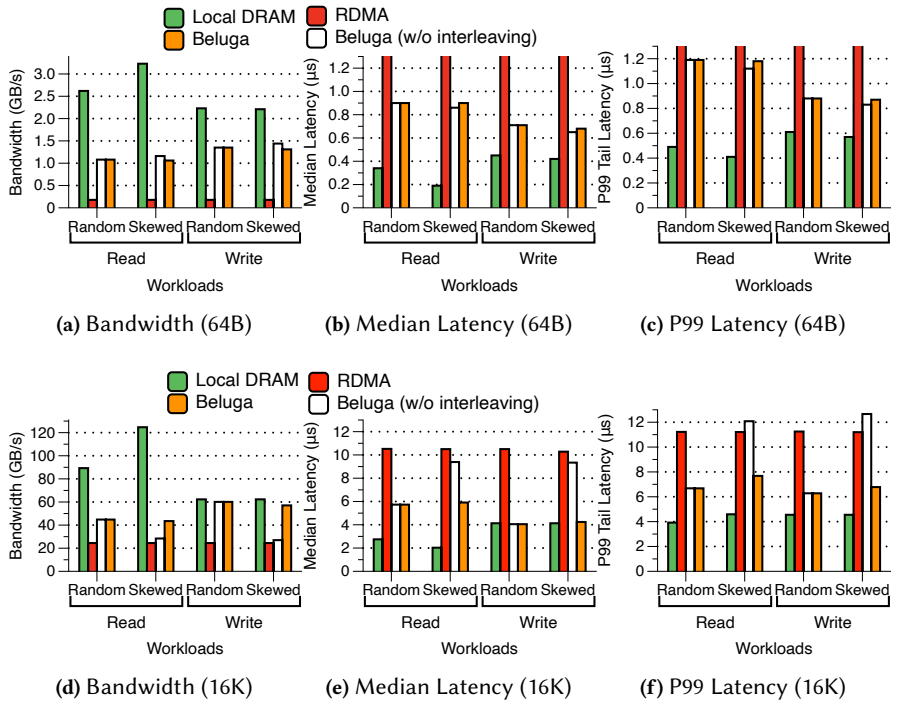


Fig. 6. Performance under concurrent 64B/16KB accesses (**Exp #3**). The setup consists of 64GB memory space with 16 threads (one thread per CPU core) performing synchronized memory access. Memory addresses are selected using zipf distribution, where the highly skewed workloads use parameter 0.99.

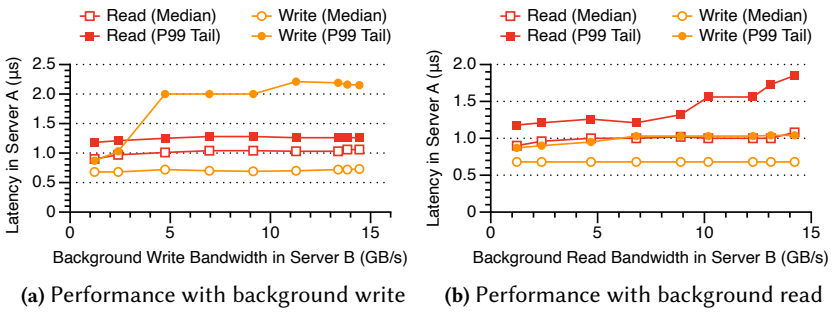


Fig. 7. Performance with concurrent background workloads (**Exp #4**). It measures the 64B read/write latency while varying the background bandwidth from 0 to 15 GB/s on the same device. The RDMA latencies in (b) and (c) exceeded 8μs and are omitted for better visualization.

(Exp #4) Impact of background workloads. In this experiment, a background server generates sustained bandwidth pressure on a single memory device, while a foreground server concurrently performs 64B read/write operations on the same device. Figure 7 shows the read/write latency under varying levels of background pressure. Results show that while median latency remains stable regardless of background bandwidth, p99 latency increases when background bandwidth pressure operates in the same direction.

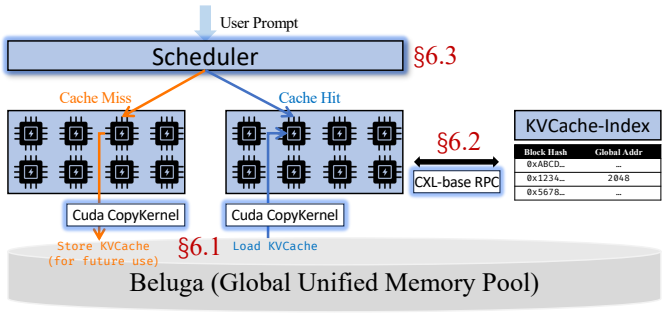


Fig. 8. KVCache management with *Beluga*.

6 Managing LLM KVCache with *Beluga*

In this section, we elaborate how to integrate *Beluga* with a popular LLM inference framework (e.g., vLLM) and efficiently manage the corresponding KVCache. As shown in Figure 8, an LLM inference system typically consists of three key components: a large, shared memory pool to store the KVCache; a global index to map token blocks to their physical addresses in the pool; and a centralized scheduler to dispatch requests to different LLM instances.

Beluga-KVCache integrates with each of these components. First, it introduces *Beluga* into the KVCache management component, where the direct memory access interface provided by CXL greatly simplifies the KVCache access process (§6.1). Second, *Beluga*-KVCache replaces the network communication between LLM instances and the indexing service via a CXL-based RPC (§6.2). Lastly, with the flatter memory hierarchy in *Beluga*-KVCache accelerates, the scheduler no longer needs to manage KVCache locality and focuses on computing resource allocation (§6.3).

6.1 Data Transfer Operations for KVCache

In transformer-based LLM inference systems, the KVCache for each token is typically stored in a highly fragmented memory layout. As shown in Figure 9, a token's key and value tensors are typically stored non-contiguously across different attention layers. Furthermore, within a single layer, the KV tensors from different tokens are also stored non-contiguously. However, an LLM inference system usually has to serialize the KVCache data (i.e. the key and value tensors), into contiguous blocks in a memory pool for effective use. This results in a significant KVCache transfer overhead.

KVCache transferring has two access patterns. *Gather write (KVCache write)*: Data from many non-contiguous GPU locations is gathered into one contiguous block in the remote pool. *Scatter read (KVCache read)*: Data from one contiguous block in the remote pool is scattered to many non-contiguous GPU locations.

RDMA relies on scatter-gather lists (sglists) to handle the above transfers. Sglists allow combining multiple GPU memory chunks into a single RDMA request. However, for KVCache, sglists are inefficient. For a Qwen-32B model with GQA (i.e., n_{heads}=8), a single token's KVCache may split into 128 non-contiguous chunks (64 layers x 2 for key/value). However, the hardware constraints (e.g., ConnectX-7 NIC limits sglists to 30 entries), resulting in complex logic to split the operation into multiple RDMA requests.

Further, the data transfer problem becomes even more challenging when we exploit KVCache sparsity. Recent studies [37, 69] show that not all tokens contribute equally to generation. This shifts the "Scatter Read" access pattern into a more difficult non-contiguous-to-non-contiguous read. For a Qwen-32B model with GQA(n_{heads}=8), exploiting sparsity results in 1024 small, 160-byte chunks

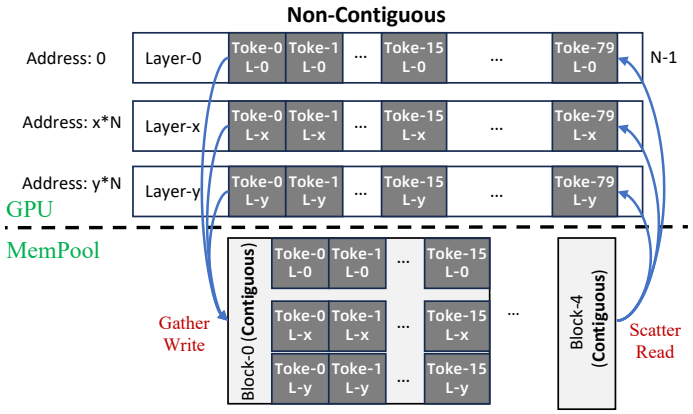


Fig. 9. KVCache Layout in GPU and Memory Pool. A single KVCache block (16 tokens) in Qwen-32B (GQA) requires 128 non-contiguous 20KB data transfers.

for a single token: n_{chunks}=$n_{\text{layers}} \times n_{\text{heads}} \times 2$, where n_{layers}=64, n_{heads}=8. Accordingly, thousands of small requests are issued, which may trigger IOPS bottlenecks. In this case, RDMA is also inefficient.

In contrast, *Beluga* leverages the fine-grained custom copy kernel to handle unlimited gather writes (from multiple GPU regions to single CXL block) or scatter reads (from single CXL block to multiple GPU regions), eliminating the tedious request management in RDMA. Therefore, by shifting KVCache transfer duty from RDMA to CXL memory pool, *Beluga* fundamentally resolves the KVCache migration bottleneck in an LLM inference system.

6.2 CXL-Based RPC

For LLM inference systems with shared KVCache storage, efficient metadata management is critical. Most systems rely on centralized metadata services to track KVCache block locations, incurring frequent remote procedure calls (RPCs). These RPCs are conventionally realized by RDMA or TCP/IP protocols that incur high latency due to network stack overhead or kernel-user space transitions.

Different from the conventional RPC realization, *Beluga* realizes the RPC via shared memory, fully utilizing the load/store semantics offered by CXL. We reserve a small memory space in *Beluga* to achieve cross-server communication. *Beluga* establishes the communication between a metadata server and clients in a producer-consumer manner. Initially, clients and the server pre-allocate fixed-size memory slots in CXL memory pool for request and response buffering. To issue a request, a client(i.e., producer) writes an idle slot and updates a status flag (i.e., REQ_READY). To consume a request, the server (i.e., consumer) continuously polls status flags using its internal spin-wait loops. After retrieving and processing a request, the server writes results to a dedicated reply slot and sets a RESP_READY flag. The client obtains the response upon detecting the flag update.

We also apply the following four optimizations to achieve high-performance. First, non-temporal store (`ntstore`) instruction is adopted by the clients, avoiding CPU cache pollution. Second, the metadata server executes a `CLFLUSH` command before reading to ensure the visibility of the latest client-written data. Third, we also batch mfence operations for concurrent RPC requests, and ensure cache line alignment for the data in RPC. Further, during the whole producing-consuming process, *Beluga* eliminates kernel-mode transitions and context switches by keeping all operations in user space. Putting these techniques together, *Beluga* is able to achieve more efficient RPCs compared with RDMA or TCP/IP.

6.3 Scheduling without KVCache Hierarchy

Traditional LLM inference systems using RDMA-based memory pools, such as NVIDIA's Dynamo [50] and MoonCake [51], suffer from a significant performance gap between local and remote memory access. To mitigate this gap, these systems must employ well-designed, cache-aware scheduling policies. These policies enforce KVCache locality by routing requests to nodes that already host the required data blocks, thereby avoiding expensive remote fetches. In practice, these policies lead to some obvious drawbacks, including the scheduling complexity, load imbalance, and maintenance overhead [76]. For example, such a policy may lead to skewed KVCache distribution, and the corresponding routers are difficult to recover from the skewed KVCache distribution.

Beluga largely eliminates the above constraints by leveraging CXL memory pools with near-local access latency. The entire CXL memory pool is abstracted as a unified and symmetric address space, where remote CXL memory access latency is comparable to local buffer access latency as demonstrated in §5.2, eliminating the performance discrepancy incurred by RDMA. We compared the performance of offloading KVCache to local memory versus *Beluga*. The results show that for cache-hit requests, the TTFT of *Beluga* is highly competitive with that of local memory. Therefore, the design in *Beluga* enables cache-oblivious scheduling. The incoming requests can be easily distributed using standard load-balancing techniques, ignoring cache locality. Furthermore, LLM inference nodes can be added/removed without re-balancing KVCache partitions, as access overhead remains uniform. By decoupling compute scheduling from KVCache locality, *Beluga* significantly reduces scheduling overhead, and is able to maintain high throughput under skewed workloads. This paradigm shift enables LLM inference systems to prioritize computational efficiency over memory hierarchy optimization.

7 Evaluation

In this section, we seek to answer the following questions:

- **Overall performance:** How does *Beluga* improve the end-to-end performance of LLM inference compared with a state-of-the-art RDMA-based system? (§7.1)
- **Sensitivity analysis:** How do workload characteristics and software/hardware configurations affect the performance benefits of *Beluga*-KVCache? (§7.2)
- **Performance breakdown:** How does each component affect *Beluga*-KVCache's overall performance? (§7.3)

Experimental setup. The evaluation is conducted on a cluster of two servers, each equipped with eight H20 (96 GB) GPUs, for a total of 16 concurrent vLLM instances managed by a centralized scheduler. The vLLM version is V1 (v0.8.5), and we enable its prefix caching feature and leverage GPU HBM to store KVCache. Other hardware configurations are shown in Table 2. To evaluate the end-to-end impact of our design, we compare our *Beluga*-KVCache against MoonCake (v3.2) [51], a highly-optimized RDMA-based baseline built upon the vLLM and LMCache framework (v0.3.1) [33, 43], and Dynamo (v0.4.1), a RDMA-based baseline from Nvidia. For fairness, we limit the cache capacity in memory pool to 2 TB for both RDMA-based solutions and *Beluga*.

Model and workloads. We used the unquantized Qwen-32B model by default. Our primary workload was **LV-Eval** [66], which features long-context QA traces with all input sequences exceeding 15K tokens. To analyze the sensitivity of our system to context length, we created three variants of the LV-Eval workload by limiting the input context to 2K, 4K, and 8K tokens, named LV-Eval-2K, LV-Eval-4K, and LV-Eval-8K, respectively. We evaluate three metrics in these workloads: Time-to-First-Token (TTFT), Time-Per-Output-Token (TPOT), and Queries-Per-Second (QPS).

7.1 End-to-End Performance Evaluation

To isolate the performance impact of initial cache population versus subsequent cache reuse, we designed two distinct experimental scenarios:

- *Cache-populate (first run):* This scenario measures the performance during the initial cache population, as vLLM computes and stores the KVCache into *Beluga*.
- *Cache-hit (second run):* All KVCache is pre-populated in the shared memory pool. This scenario measures the performance when the prefill stage is accelerated entirely by cache reuse.

Table 5. Inference performance on LV-Eval [66] (**Exp #5**).

Metric	Dynamo	vLLM	vLLM +MoonCake	vLLM +*Beluga*
First Run (Cache-populate)				
Avg TTFT	17.96 s	18.76 s	19.66 s	17.22s
P99 TTFT	54.53 s	40.47 s	41.65 s	44.6 s
Average TPOT	1.55 s	2.580 s	1.97 s	1.54 s
P99 TPOT	10.99 s	20.17 s	20.84 s	16.14 s
QPS (req/s)	1.15	0.96	1.02	1.24
Second Run (Cache-hit)				
Average TTFT	15.69 s	18.23 s	13.00 s	1.36 s
P99 TTFT	40.97 s	39.25 s	39.91 s	5.02 s
Average TPOT	1.38 s	2.82 s	1.10 s	0.15 s
P99 TPOT	11.01 s	19.74 s	10.58 s	1.34 s
QPS (req/s)	1.32	0.96	1.54	11.32

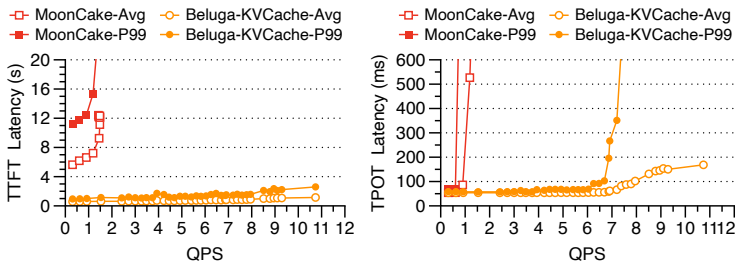
(Exp #5) Results. The evaluation uses a closed-loop client model to measure the peak throughput. As shown in Table 5, *Beluga*-KVCache consistently outperforms the RDMA-based baseline across all metrics. In the cache-populate scenario, where the workload has a 30% cache hit ratio, both MoonCake and *Beluga*-KVCache outperform the original vLLM. Notably, *Beluga*-KVCache further improves upon MoonCake, reducing the average TTFT by 12.4% and increasing QPS by 21.5%. The performance advantage is most significant in the cache-hit scenario. *Beluga*-KVCache reduces the average TTFT by 89.6% and achieves a 7.35× improvement in QPS. MoonCake's lower performance can be attributed to two main factors. First, there exists an inherent performance gap between CXL and RDMA technologies. Second, its implementation includes additional overhead in the critical path, such as extra memory copies and allocations, which are being continuously addressed by the community[2].

Cache Hit Ratio in GPU. In our experimental setup, the model occupies 60.0 GB, we allocate 92% of the total memory space, therefore there are 28.3 GB for KVCache storage. Our performance analysis shows that the cache hit ratio in GPU HBM varies dynamically throughout the execution, reaching a peak performance of 14.6% under this memory configuration.

7.2 Sensitivity Analysis

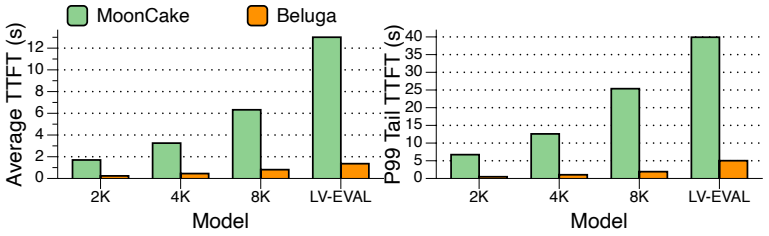
In this section we specifically analyze the sensitivity of our systems across three critical dimensions: the workload characteristics, software configurations, and hardware configurations.

[2]Our evaluation is based on the versions available at the time of paper submission. After paper submission, we also compared against the latest version of MoonCake (v3.6), which achieves 4.10 QPS—2.66× higher than MoonCake-v3.2, but still only 36% of *Beluga*-KVCache. We found that v3.2 incurred expensive memory allocations and copies in the critical path, which v3.6 subsequently optimized. This evolution underscores a key point of our paper: designing an efficient zero-copy RDMA solution for KVCache is inherently difficult (v3.6 was released approximately eighteen months after the original MoonCake paper/code), whereas CXL fundamentally simplifies the programming complexity.



(a) TTFT with different Request Rates **(b)** TPOT with different Request Rates

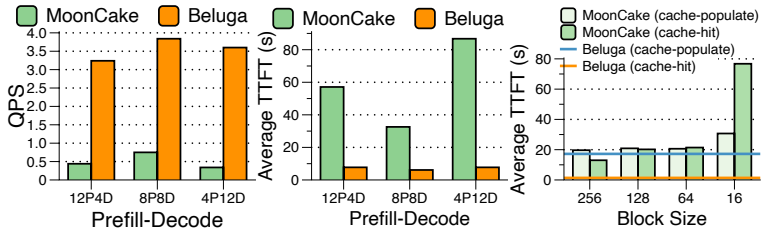
Fig. 10. Sensitivity to request arrival rates (**Exp #6**).



(a) Average TTFT **(b)** P99 TTFT

Fig. 11. Sensitivity to input context lengths (**Exp #7**).

Performance under different workload characteristics. To assess *Beluga* performance under realistic scenarios, we design two test workloads that vary in request arrival rates and input lengths.
(**Exp #6**) Request arrival rates. Our evaluation methodology consists of two phases. First, we execute the LV-EVAL workload to initialize the KVCache. Then, we conduct performance measurements with request rates ranging from 0.3 to 9.0 QPS during a second execution. Figure 10 shows the results of TTFT and TPOT. We observe that *Beluga*-KVCache consistently outperforms MoonCake, achieving lower latencies for both metrics. This is because, on the second run, all requests result in a cache hit, making the KVCache read time the primary system bottleneck. In this cache-bound scenario, the CXL-based data access in *Beluga* is significantly more efficient than the RDMA-based access used by MoonCake.
(**Exp #7**) Input context lengths. We evaluate the systems using workloads with varying input context lengths: 2K, 4K, and 8K, via limiting the input length of the original LV-EVAL. Figure 11 shows the average and P99 TTFT for both *Beluga*-KVCache and MoonCake. We observe that as the number of input tokens increases, the performance improvement of *Beluga* becomes more significant. This is because the KVCache write/read time accounts for a larger portion of the end-to-end latency in longer-context scenarios.
(Exp #8) Performance with different software configurations. We evaluate *Beluga* performance under different inference framework configurations, focusing on two aspects: Prefill-Decode deployment and KVCache block size configurations.
Prefill-decode disaggregated. In the prefill-decode disaggregated architecture, KVCache is first stored in the memory pool after the prefill phase, then loaded by decode nodes. As demonstrated in Fig.12a and Fig.12b, *Beluga*'s optimized KVCache load/store path achieves 3.41×~9.47× higher QPS compared to MoonCake.
Block size of KVCache access. Our experiments reveal significant differences in block size requirements between RDMA-based solutions and *Beluga*. RDMA-based approaches require operation



(a) P-D Ratio (QPS) **(b)** P-D Ratio (TTFT) **(c)** Block sizes (TTFT)

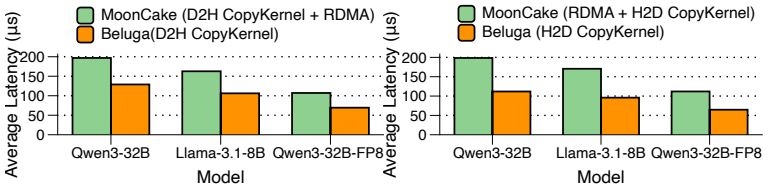
Fig. 12. Sensitivity to software configurations (**Exp #8**).

batching to amortize control overhead, combining multiple vLLM KVCache blocks into larger super blocks. While vLLM uses a 16-token block size for efficient GPU HBM space management, this granularity is inefficient for RDMA transfers. Consequently, LMCache defaults to a larger block size of 256 tokens for KVCache indexing and transfer. As shown in Fig.12c, MoonCake achieves 13.0s TTFT for cache hits with the larger block size (256 tokens). However, when using 16-token block size, TTFT increases to 76.8s, exceeding even the recomputation latency of the first run. In contrast, *Beluga* operates efficiently with vLLM's native block size.

Performance with and without memory interleaving. We conduct an experiment to demonstrate the effectiveness of software memory interleaving in *Beluga*. With software interleaving across two CXL/PCIe adapters and 32 memory devices, the system achieves a QPS of 11.32 during rerun (cache hit), a 33.2% improvement compared to 8.49 without interleaving. The results confirm that software interleaving effectively distributes memory access load and reduces access contention, leading to better overall system performance.

7.3 The Breakdown of Performance Improvement

In this section we analyze the breakdown of performance improvement of our systems, including KVCache transfer performance and RPC performance.



(a) Write Latency **(b)** Read Latency

Fig. 13. Data transfers for dense KVCache (**Exp #9**).

(Exp #9) Performance of data transfers for KVCache (Dense). To demonstrate the advantages of CXL over RDMA in KVCache transfers, we conducte benchmarks focusing on scatter-gather operations - the crucial operations in dense KVCache transfers. Our evaluation encompasses three different models: Qwen3-32B, Llama-3.1-8B, and Qwen3-32B-FP8. Each model exhibits distinct KVCache block layouts. For instance, in the Qwen-32B model (using Grouped Query Attention, or GQA), a KVCache block is distributed across 128 non-contiguous sub-blocks, while Llama-3.1-8B utilizes 64 sub-blocks. Across all models, each KVCache block maintains 16 tokens, which is vLLM's default configuration. This token count remains constant and does not influence the number of non-contiguous sub-blocks in the models. As shown in Figure 13, compared to MoonCake, which

uses a CPU-centric method with an inefficient two-step data path (GPU → Host → MemPool), *Beluga* eliminates the host memory bounce buffer entirely. This direct data path reduces KVCache write and read latencies by 36.2% and 38.7%, respectively, showcasing its superior efficiency and simpler data management.

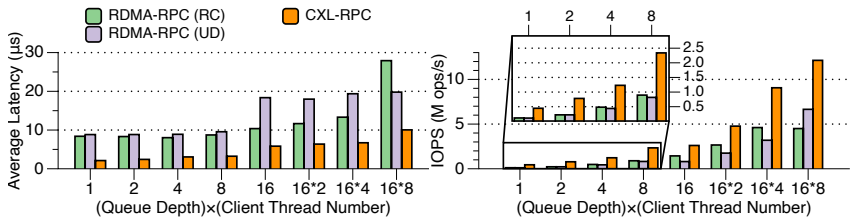
Table 6. Sparsity analysis and the read latency (**Exp #10**).

		Llama-3-8B	**Qwen3-32B**
Top 256 tokens (per head and per layer)	Non-contig	131330	782774
	Contiguous	130814	265802
Loading 16 tokens (μs)	RDMA	2670	5260
	CXL	**97**	**211**

(Exp #10) Data transfers for KVCache (Sparse). To further demonstrate CXL's efficiency in KVCache transfer, we extend our evaluation to models with sparse KVCache implementations.

We first analyze the KVCache sparsity patterns in both Qwen-32B and Llama-3-8B models using an attention score-based sparsification method [69]. This analysis helps us understand the unique characteristics of sparse workloads in large language models. Table 6 shows the results of selecting the top 256 tokens per head and per layer for a sequence with 7942 tokens. Our analysis reveals that over 74% of the selected high-importance tokens in Qwen-32B are non-contiguous, confirming that the memory access pattern for sparse KVCache is overwhelmingly discrete.

Based on these findings, we evaluate the performance of reading KVCache for 16 sparse tokens. In the Qwen-32B model, *Beluga* achieves a remarkable 95.9% latency reduction compared to RDMA. The reason is fundamental: the RDMA solution is bottlenecked by issuing numerous high-overhead requests for non-contiguous data, while *Beluga* allows a single CUDA kernel to manage the entire fine-grained transfer efficiently through CXL's directly mapped memory.



(a) Latency **(b)** Throughput

Fig. 14. Performance of CXL/RDMA RPC (**Exp #11**).

(Exp #11) Performance of RPC. In this evaluation, we measure how much faster RPC performs when using CXL instead of RDMA. We compare our CXL-RPC against two functionally equivalent RDMA implementations: RDMA-RC (Reliable Connection) and RDMA-UD (Unreliable Datagram). The RDMA baseline also uses busy-polling on its completion queue (CQ), just similar to the thread-model of our CXL-RPC. We use a ping-pong benchmark between two servers, with a single-threaded RPC service and multi-threaded clients. Each RPC request and reply is 64 bytes.

The results are presented in Figure 14. At low concurrency (QD=1), CXL-RPC achieves a round-trip latency of 2.11 μs, which involves four memory operations (two reads and two writes) and queuing overhead It achieves a 4× improvement over both RDMA-RC (8.39 μs) and RDMA-UD (8.83 μs). This advantage stems from CXL's direct load-store access model, which bypasses the extensive software and hardware overhead inherent in RDMA protocols. Under high concurrency (QD=128), CXL-RPC's throughput (single thread) reaches 12.13 Mops, significantly outperforming both RDMA-RC (4.5 Mops) and RDMA-UD (6.65 Mops) by 2.7× and 1.8×, respectively. This

demonstrates how CXL leverages the CPU's cache hierarchy to buffer and burst writes, effectively amortizing protocol overhead under heavy load. It is important to note that CXL-RPC provides lower reliability guarantees compared to RDMA transport protocols. Our implementation relies on upper-layer mechanisms for reliability assurance. While strengthening these guarantees remains a topic for future research, the current CXL-RPC design prioritizes performance and cost efficiency within rack-scale deployments, rather than serving as a complete RDMA replacement.

8 Future Work on CXL Switches

Using one of the first commercial CXL 2.0 switches, this paper presents a foundational analysis of a CXL-switched memory pool. This paper establishes key performance baselines for CXL 2.0 systems. It also demonstrates CXL's effectiveness for complex tasks like KVCache offloading. These results open up a vast design space for future CXL-native systems. Based on our analysis, we outline several promising directions for future research.

Future work on hardware architecture. The emergence of CXL 3.1 suggests a path toward a large-scale, fully disaggregated architecture. We envision a future GPU cluster built on a CXL fabric, as shown in Figure 15. In this architecture, a symmetric fabric would connect pools of compute, memory, and storage, providing unified, low-latency access for all attached devices. By connecting GPUs directly to memory box, the latency overhead of traversing the host's PCIe switch and root complex is eliminated, providing lower-latency memory access compared to *Beluga*.

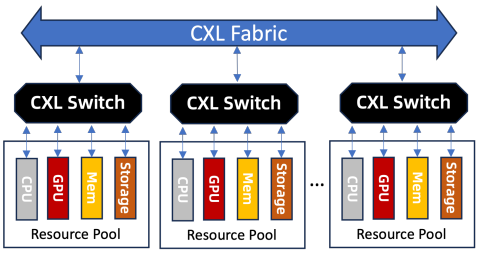


Fig. 15. Fully disaggregated architecture.

Future work on software design. This new hardware architecture requires a corresponding software stack to manage resource pooling and coherent data sharing at scale. First, it needs to develop resource schedulers that can dynamically manage GPU/CPU resources across multiple CXL switches. This requires a co-design approach that balances application needs with the specific performance characteristics of the hardware. Second, a critical research area is the design of more efficient coherence protocols for CXL memory pools. Potential directions include: (1) leveraging application-level semantics to relax coherency, (2) implementing scalable, directory-based coherence using in-switch resources, and (3) developing hybrid models where a small, hardware-coherent region manages coherence metadata for a significantly larger address space.

Future work on database design. CXL memory pooling enables new opportunities for memory-intensive data management systems that require large-scale random access capabilities, particularly for vector databases and graph databases. For instance, graph-based algorithms like HNSW, which traditionally demand full in-memory storage, can leverage CXL memory pools to break through memory capacity limitations.

9 Related Work

Data management in LLM inference. KVCache represents the primary memory bottleneck in LLM inference [22, 37, 56], motivating extensive research efforts aimed at reducing its footprint. At

the algorithmic level, researchers have explored various optimizations: MQA [55] and GQA [13] reduce KVCache computation complexity through architectural innovations, while H2O [69], FastGen [22], and SnapKV [37] leverage attention score sparsity to selectively compress or evict less important KVCache entries. At the system level, PagedAttention [33] introduced a virtual memory-inspired mechanism for efficient KVCache management, while MoonCake [51] and Dynamo [50] leveraged remote resource pooling to enable large-scale request scheduling. Building upon these insights, *Beluga* proposes a new hardware architecture for efficient KVCache management that reduces data movement overhead at the system level while enabling flexible random access to support the current novel algorithmic optimizations, such as the sparse KVCache.

Disaggregated memory architectures for data management. Disaggregated memory as a resource extension has been widely adopted in database systems. RDMA-based memory disaggregation [15, 29, 30, 36, 58, 65, 74] has been applied in various scenarios, including transaction processing and storage optimization [36, 45, 47]. Recent systems like MoonCake [51] and Dynamo [50] leverage RDMA to offload KVCache management to memory pools. While RDMA provides one-sided memory access, it operates as a network protocol rather than a true memory interface, introducing additional complexity and latency overhead. CXL addresses these limitations by offering native memory-like interfaces and characteristics. There also exist specialized solutions like CloudMatrix [76], a large-scale supernode built on Unified Bus (UB) in Huawei. While it establishes a CXL-like fabric for uniform peer-to-peer communication, its vendor-specific design limits widespread adoption.

CXL-based memory architectures. The introduction of CXL has triggered extensive research into its performance characteristics and system-level potential. Early characterization studies [24, 35, 40, 59, 71] have leveraged FPGA-based prototypes or initial CXL 1.1 hardware to provide the first quantitative analysis of this emerging interconnect. Subsequently, a range of studies have demonstrated the practicality of CXL across diverse real-world scenarios, encompassing in-memory databases [12, 25, 63], graph processing [54], and deep learning systems [14, 39], among others. Moreover, several works [41, 59] have conducted careful comparative analyses of CXL and RDMA. For example, [41] demonstrates the performance superiority of CXL over RDMA, and [59] reports on leveraging CXL to enhance the performance of RDMA-based systems. Prior work has explored CXL for CPU memory expansion. This paper extends CXL's capabilities to GPU-centric KVCache management using commercial CXL 2.0 switches.

10 Conclusion

In this paper, we have designed, evaluated, and optimized Beluga, a novel CXL-switch-based memory system for GPU clusters. Through a detailed characterization of commercial CXL hardware, we identify key performance characteristics and propose a set of optimizations for building high-performance applications. To demonstrate the effectiveness of Beluga, we design and implement Beluga-KVCache, a system tailored for managing the large-scale KVCache in LLM inference. This system includes optimized and simplified data transfers, a lightweight RPC, and a simplified scheduler based on Beluga. Our evaluation demonstrates that Beluga-KVCache significantly outperforms the state-of-the-art RDMA-based solution in MoonCake, highlighting CXL's potential as a foundation for future disaggregated memory systems.

Acknowledgements

We sincerely thank anonymous reviewers for their valuable feedback, which significantly improved this paper. We thank the PolarDB Infrastructure Team and Alibaba Infrastructure Service (AIS) Group for their support and contributions to this work.

References

[1] [n. d.]. CLFLUSH. [Online]. https://www.felixcloutier.com/x86/clflush.

[2] [n. d.]. GeForce RTX 4090 Graphics Cards for Gaming. [Online]. https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/.

[3] [n. d.]. Intel® Data Direct I/O Technology. [Online]. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[4] [n. d.]. MTRR (Memory Type Range Register) control. [Online]. https://docs.kernel.org/arch/x86/mtrr.html.

[5] [n. d.]. NVIDIA GPUDirect. [Online]. https://developer.nvidia.com/gpudirect.

[6] 2025. Aurora. https://aws.amazon.com/rds/aurora.

[7] 2025. GaussDB. https://www.huaweicloud.com/intl/en-us/product/gaussdb.html.

[8] 2025. MindsDB. https://mindsdb.com.

[9] 2025. PolarDB. https://www.alibabacloud.com/en/product/polardb.

[10] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[11] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant J. Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: KV Cache reduction through key tokens selection for Efficient Generative Inference. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*, Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2024/hash/48fecef47b19fe501d27d338b6d52582-Abstract-Conference.html

[12] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna T. Malladi, and Yang-Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*, Spyros Blanas and Norman May (Eds.). ACM, 8:1–8:5. doi:10.1145/3533737.3535090

[13] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 4895–4901. doi:10.18653/V1/2023.EMNLP-MAIN.298

[14] Moiz Arif, Kevin Assogba, M. Mustafa Rafique, and Sudharshan Vazhkudai. 2022. Exploiting CXL-based Memory for Distributed Deep Learning. In *Proceedings of the 51st International Conference on Parallel Processing, ICPP 2022, Bordeaux, France, 29 August 2022 - 1 September 2022*. ACM, 19:1–19:11. doi:10.1145/3545008.3545054

[15] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.* 11, 11 (2018), 1604–1617. doi:10.14778/3236187.3236209

[16] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1849–1862. doi:10.14778/3229863.3229872

[17] Shiyang Chen, Rain Jiang, Dezhi Yu, Jinlai Xu, Mengyuan Chao, Fanlong Meng, Chenyu Jiang, Wei Xu, and Hang Liu. 2025. KVDirect: Distributed Disaggregated LLM Inference. *CoRR* abs/2501.14743 (2025). arXiv:2501.14743 doi:10.48550/ARXIV.2501.14743

[18] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–14.

[19] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Rethinking stateful stream processing with rdma. In *Proceedings of the 2022 International Conference on Management of Data*. 1078–1092.

[20] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv e-prints* (2024), arXiv–2407.

[21] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2, 1 (2023).

[22] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2024. Model Tells You What to Discard: Adaptive KV Cache Compression for LLMs. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=uNrFpDPMyo

[23] Victor Giannakouris and Immanuel Trummer. 2025. λ-Tune: Harnessing Large Language Models for Automated Database System Tuning. *Proc. ACM Manag. Data* 3, 1 (2025), 2:1–2:26. doi:10.1145/3709652

[24] 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.

[25] Yunyan Guo and Guoliang Li. 2024. A CXL- Powered Database System: Opportunities and Challenges. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 5593–5604. doi:10.1109/ICDE60146.2024.00447

[26] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/028fcbcf85435d39a40c4d61b42c99a4-Abstract-Conference.html

[27] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, et al. 2024. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565* (2024).

[28] Qingda Hu, Xinjun Yang, Feifei Li, Junru Li, Ya Lin, Yuqi Zhou, Yicong Zhu, Junwei Zhang, Rongbiao Xie, Ling Zhou, et al. 2026. PolarStore: High-Performance Data Compression for Large-Scale Cloud-Native Databases. In *24th USENIX Conference on File and Storage Technologies*. USENIX Association, Santa Clara, CA, USA.

[29] Matthias Jasny, Lasse Thostrup, Sajjad Tamimi, Andreas Koch, Zsolt István, and Carsten Binnig. 2024. Zero-sided RDMA: Network-driven Data Shuffling for Disaggregated Heterogeneous Cloud DBMSs. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–28.

[30] Matthias Jasny, Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2025. Synchronizing Disaggregated Data Structures with One-Sided RDMA: Pitfalls, Experiments and Design Guidelines. *ACM Transactions on Database Systems* 50, 1 (2025), 1–40.

[31] Saehan Jo and Immanuel Trummer. 2025. SpareLLM: Automatically Selecting Task-Specific Minimum-Cost Large Language Models under Equivalence Constraint. *Proc. ACM Manag. Data* 3, 3, Article 219 (June 2025), 26 pages. doi:10.1145/3725356

[32] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance {RDMA} systems. In *2016 USENIX annual technical conference (USENIX ATC 16)*. 437–450.

[33] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 611–626. doi:10.1145/3600006.3613165

[34] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 155–172. https://www.usenix.org/conference/osdi24/presentation/lee

[35] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835

[36] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, Ashvin Goel and Dalit Naor (Eds.). USENIX Association, 99–114. https://www.usenix.org/conference/fast23/presentation/li-pengfei

[37] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. SnapKV: LLM Knows What You are Looking for Before Generation. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/28ab418242603e0f7323e54185d19bde-Abstract-Conference.html

[38] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[39] Haifeng Liu, Long Zheng, Yu Huang, Jingyi Zhou, Chaoqiang Liu, Runze Wang, Xiaofei Liao, Hai Jin, and Jingling Xue. 2024. Enabling Efficient Large Recommendation Model Training with Near CXL Memory Processing. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*. IEEE, 382–395. doi:10.1109/ISCA59077.2024.00036

[40] 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 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Katai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 1203–1217. doi:10.1145/3676641.3715987

[41] 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). arXiv:2405.14209 doi:10.48550/ARXIV.2405.14209

[42] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. 2003. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th Annual International Conference on Supercomputing* (San Francisco, CA, USA) *(ICS '03)*. Association for Computing Machinery, New York, NY, USA, 295–304. doi:10.1145/782814.782855

[43] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 38–56. doi:10.1145/3651890.3672274

[44] Lin Long, Xijun Gu, Xinjie Sun, Wentao Ye, Haobo Wang, Sai Wu, Gang Chen, and Junbo Zhao. 2025. Bridging the Semantic Gap Between Text and Table: A Case Study on NL2SQL. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. https://openreview.net/forum?id=qmsX2R19p9

[45] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. 2024. DEX: Scalable Range Indexing on Disaggregated Memory. *Proc. VLDB Endow.* 17, 10 (2024), 2603–2616. doi:10.14778/3675034.3675050

[46] Kuan Lu, Zhihui Yang, Sai Wu, Ruichen Xia, Dongxiang Zhang, and Gang Chen. 2025. Adda: Towards Efficient in-Database Feature Generation via LLM-based Agents. *Proc. ACM Manag. Data* 3, 3, Article 125 (June 2025), 27 pages. doi:10.1145/3725262

[47] Xuchuan Luo, Jiacheng Shen, Pengfei Zuo, Xin Wang, Michael R. Lyu, and Yangfan Zhou. 2024. CHIME: A Cache-Efficient and High-Performance Hybrid Index on Disaggregated Memory. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton (Eds.). ACM, 110–126. doi:10.1145/3694715.3695959

[48] Marvell and XConn Technologies. [n. d.]. Marvell to Acquire XConn Technologies, Expanding Leadership in AI Data Center Connectivity. https://investor.marvell.com/news-events/press-releases/detail/1004/marvell-to-acquire-xconn-technologies-expanding-leadership-in-ai-data-center-connectivity.

[49] Jixuan Nie, Xia Hou, Wenfeng Song, Xuan Wang, Xingliang Jin, Xinyu Zhang, ShuoZhe Zhang, and Jiaqi Shi. 2024. Knowledge Graph Efficient Construction: Embedding Chain-of-Thought into LLMs. In *Proceedings of Workshops at the 50th International Conference on Very Large Data Bases, VLDB 2024, Guangzhou, China, August 26-30, 2024*. VLDB.org. https://vldb.org/workshops/2024/proceedings/LLM+KG/LLM+KG-4.pdf

[50] NVIDIA Corporation. 2025. NVIDIA Dynamo Open-Source Library Accelerates and Scales AI Reasoning Models. https://nvidianews.nvidia.com/news/nvidia-dynamo-open-source-library-accelerates-and-scales-ai-reasoningmodels. Accessed: 2025-04-23.

[51] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 155–170. https://www.usenix.org/conference/fast25/presentation/qin

[52] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, Chris J Newburn, Dmitri Vainbrand, I-Hsin Chung, et al. 2023. GPU-initiated on-demand high-throughput storage access in the BaM system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 325–339.

[53] Ricardo Salazar-Díaz, Boris Glavic, and Tilmann Rabl. 2024. InferDB: In-Database Machine Learning Inference Using Indexes. *Proc. VLDB Endow.* 17, 8 (2024), 1830–1842. doi:10.14778/3659437.3659441

[54] Shintaro Sano, Yosuke Bando, Kazuhiro Hiwada, Hirotsugu Kajihara, Tomoya Suzuki, Yu Nakanishi, Daisuke Taki, Akiyuki Kaneko, and Tatsuo Shiozawa. 2023. GPU Graph Processing on CXL-Based Microsecond-Latency External Memory. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, Denver, CO, USA, November 12-17, 2023*. ACM, 961–972. doi:10.1145/3624062.3624173

[55] Noam Shazeer. 2019. Fast Transformer Decoding: One Write-Head is All You Need. *CoRR* abs/1911.02150 (2019). arXiv:1911.02150 http://arxiv.org/abs/1911.02150

[56] Luohe Shi, Hongyi Zhang, Yao Yao, Zuchao Li, and Hai Zhao. 2024. Keep the Cost Down: A Review on Methods to Optimize LLM' s KV-Cache Consumption. *CoRR* abs/2407.18003 (2024). arXiv:2407.18003 doi:10.48550/ARXIV.2407.

18003
[57] XConn Technologies. [n. d.]. World's first CXL 2.0 and PCIe Gen5 Switch IC. https://www.xconn-tech.com/product.
[58] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. 2023. Distributed GPU joins on fast RDMA-capable networks. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[59] Zhonghua Wang, Yixing Guo, Kai Lu, Jiguang Wan, Daohui Wang, Ting Yao, and Huatao Wu. 2024. Rcmp: Reconstructing RDMA-Based Memory Disaggregation via CXL. *ACM Trans. Archit. Code Optim.* 21, 1, Article 15 (Jan. 2024), 26 pages. doi:10.1145/3634916
[60] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. OpenSearch-SQL: Enhancing Text-to-SQL with Dynamic Few-shot and Consistency Alignment. *Proc. ACM Manag. Data* 3, 3, Article 194 (June 2025), 24 pages. doi:10.1145/3725331
[61] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).
[62] Xinjun Yang, Feifei Li, Yingqiang Zhang, Hao Chen, Qingda Hu, Panfeng Zhou, Qiang Zhang, Shuai Li, Zongzhi Chen, Zheyu Miao, et al. 2025. From Scale-Up to Scale-Out: PolarDB's Journey to Achieving 2 Billion tpmC. *Proceedings of the VLDB Endowment* 18, 12 (2025), 5059–5072.
[63] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, Wenpu Hu, Jim Kao, and Jianping Jiang. 2025. Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 689–702. doi:10.1145/3722212.3724460
[64] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*. 94–109.
[65] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu
[66] Tao Yuan, Xuefei Ning, Dong Zhou, Zhijie Yang, Shiyao Li, Minghui Zhuang, Zheyue Tan, Zhuyu Yao, Dahua Lin, Boxun Li, et al. 2024. Lv-eval: A balanced long-context benchmark with 5 length levels up to 256k. *arXiv preprint arXiv:2402.05136* (2024).
[67] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking database high availability with RDMA networks. *Proc. VLDB Endow.* 12, 11 (July 2019), 1637–1650. doi:10.14778/3342263.3342639
[68] Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M. Patel. 2024. ReAcTable: Enhancing ReAct for Table Question Answering. *Proc. VLDB Endow.* 17, 8 (2024), 1981–1994. doi:10.14778/3659437.3659452
[69] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark W. Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/6ceefa7b15572587b78ecfcebb2827f8-Abstract-Conference.html
[70] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html
[71] Yuhong Zhong, Daniel S. Berger, Pantea Zardoshti, Enrique Suarez, Jacob Nelson, Antonis Psistakis, Joshua Fried, and Asaf Cidon. 2025. My CXL Pool Obviates Your PCIe Switch. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems* (Banff, AB, Canada) *(HotOS '25)*. Association for Computing Machinery, New York, NY, USA, 58–66. doi:10.1145/3713082.3730393
[72] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[73] Lixi Zhou, Jiaqing Chen, Amitabh Das, Hong Min, Lei Yu, Ming Zhao, and Jia Zou. 2022. Serving Deep Learning Models with Deduplication from Relational Databases. *Proc. VLDB Endow.* 15, 10 (2022), 2230–2243. doi:10.14778/3547305.3547325

[74] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A fast and cost-efficient storage engine using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 International Conference on Management of Data*. 685–699.

[75] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design guidelines for correct, efficient, and scalable synchronization using one-sided RDMA. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[76] Pengfei Zuo, Huimin Lin, Junbo Deng, Nan Zou, Xingkun Yang, Yingyu Diao, Weifeng Gao, Ke Xu, Zhangyu Chen, Shirui Lu, Zhao Qiu, Peiyang Li, Xianyu Chang, Zhengzhong Yu, Fangzheng Miao, Jia Zheng, Ying Li, Yuan Feng, Bei Wang, Zaijian Zong, Mosong Zhou, Wenli Zhou, Houjiang Chen, Xingyu Liao, Yipeng Li, Wenxiao Zhang, Ping Zhu, Yinggang Wang, Chuanjie Xiao, Depeng Liang, Dong Cao, Juncheng Liu, Yongqiang Yang, Xiaolong Bai, Yi Li, Huaguo Xie, Huatao Wu, Zhibin Yu, Lv Chen, Hu Liu, Yujun Ding, Haipei Zhu, Jing Xia, Yi Xiong, Zhou Yu, and Heng Liao. 2025. Serving Large Language Models on Huawei CloudMatrix384. arXiv:2506.12708 [cs.DC]

Received July 2025; revised October 2025; accepted November 2025