CACHEFLOW: FAST AND MEMORY EFFICIENT LLM INFERENCE VIA IN-NETWORK PREFIX CACHE POOL

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic systems consist of multiple copies of the same agent, defined by a common prompt prefix, that are hosted on multiple physical machines. Each of the machines is required to process the same common prefixes separately, leading to redundant computation across and increased latency. We recognize that all of the physical machines hosting the agents are connected by a computer network, making the network the trivial choice when storing the shared data needed across all machines. Leveraging this idea, this paper presents CacheFlow, a design and an implementation of a prefix cache pool that stores KV cache on a network device and delivers it to the GPU via GPUDirect RDMA bypassing the host CPU and DRAM. Our evaluations show that CacheFlow improves the overall prefix cache efficiency compared to vanilla vLLM by reducing the average Time-to-First-Token (TTFT) by up to **82%**, increasing throughput by **39%**, reducing the host memory storage requirement by **50%**. In addition, CacheFlow shows up to **3X** improvement in I/O throughput compared to host DRAM based cache systems.

1 INTRODUCTION

Large Language Model (LLM)-based agentic systems have recently gained wide traction for their complex problem-solving abilities, achieved through the coordination of multiple agents, each with a role defined by role-specific prompt prefixes Talebirad & Nadiri (2023); Han et al. (2024); Li et al. (2024); He et al. (2025); Yu et al. (2024); Tran et al. (2025); Li et al. (2023); Wu et al. (2024); Chen et al. (2023). Interaction with an agent is a set of inference requests—incorporating these agent-specific prefixes—sent to a set of LLMs hosted on physical machines equipped with AI accelerators (e.g., GPUs, TPUs, ASICs) Achiam et al. (2023); Anil et al. (2023); Shen et al. (2023). However, hosting the same type of agent on multiple machines inherently introduces high redundancy, resulting in high inference latency, due to the need for each machine to ingest and process the same set of prefixes repeatedly Kwon et al. (2023); Patel et al. (2025); Zheng et al. (2024).

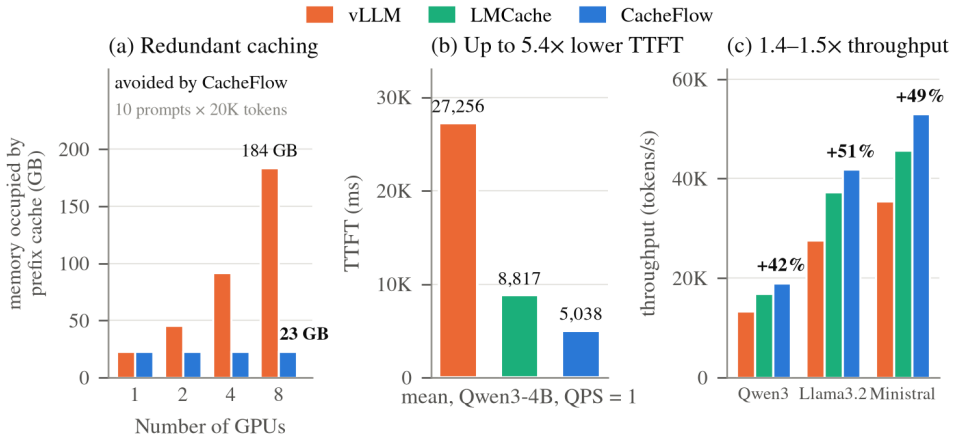


Figure 1: (a) Estimated prefix cache memory for 10 prompts × 20K tokens (Llama-3.2-3B): per-GPU caching replicates 23 GB on every GPU, while CacheFlow keeps one copy in the network. (b) Mean TTFT, Qwen3-4B, QPS=1. (c) Mean throughput over QPS 1–10.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

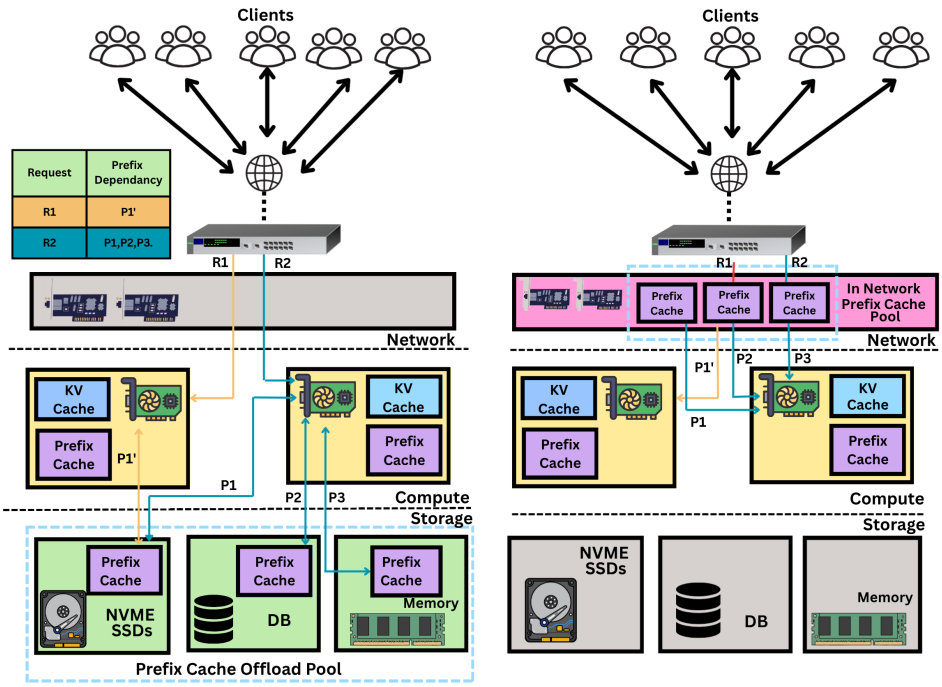


Figure 2: CacheFlow (right) is more efficient than current solutions (left) since cache operations are performed on the path of the query. This effect compounds as more prefix caches are retrieved.

Although solutions such as prefix caching Juravsky et al. (2024) reduce such computational redundancy by reusing the key-value (KV) values produced from static prompt prefix, such solutions are focused on reducing computation on a single machine and do not apply for agents distributed across multiple machines. There are also solutions to store prefix caches in a remote machine Cheng et al. (2025), but these solutions inherently require multiple hops to a remote machine to retrieve KV values and often transfer data across slower network interfaces and protocols.

In modern data centers specialized for running artificial intelligence (AI) and machine learning (ML) inference workloads, machines with AI accelerators that host the agents are often collocated within a rack Chatterjee & V (2023) connected by high-speed, often programmable, network interconnects (e.g., NVLink Foley & Danskin (2017), NVSwitch Li et al. (2020), ICI Jouppi et al. (2023)) responsible for both north-south traffic (e.g., agent queries and typical web traffic) and east-west traffic (e.g., object cache and databases for RAG Lewis et al. (2020)). A trivial fact that is universal to any modern computer system is that the network is the first entry point and the last exit point for any shared data, making the network the perfect place to store multiple types of data that are shared across the machine. Coupled with the increased programmability of network devices Bosshart et al. (2014); Kfoury et al. (2024) and the large overhead needed to recompute KV values across each machine, it makes increasingly more sense to utilize the network to assist prefix caching.

Motivation. Previous work on optimizing prefix caching focused on improving the cache hit rate within a single machine or using a set of remote machines to store the secondary cache layer to be shared by multiple machines. However, we observe that such a design results in inherent inefficiencies when there are multiple copies of the same agent in a distributed setting, in terms of both the amount of total memory used and the increased latency. Keeping prefix cache on every machine inherently increases cache, thus memory, redundancy as the set of agents are repeated across multiple machines. Using a remote machine as a secondary cache layer requires multiple round-trips across the network to retrieve the cache, resulting in lower throughput, higher network overhead and latency (e.g., time-to-first-token (TTFT)). Given such shortcomings, the main motivation for this work is to utilize the network as the prefix cache pool, i.e., the logic and the storage layer to store and share prefix caches (Figure 2 provides a high-level overview of this approach compared to existing solutions). Some benefits of this approach compared to other existing solutions are: 1) Given

the incoming queries already pass through network devices, performing cache lookup and retrieval along the way can cut redundant hops on the query path. 2) Instead of allocating prefix caches in each GPU memory or in another host memory, both of which are quite expensive, you take advantage of underutilized memory inside network devices, freeing up the more expensive host memory to be used for other tasks. 3) Modern network devices have specialized ASICs, protocols, and dedicated communication links to accelerate data transfer between the network and to AI accelerators.

Challenges. The design and implementation of prefix cache pool within network devices is not trivial, and there are few challenges that must be overcome to achieve efficiency and effectiveness simultaneously. First, there are multiple types of network devices that exist within AI data centers, and one must first identify the correct type of network device to use that gives the highest benefit in terms of performance and cost. For example, given the sizes of prefix cache values that often exceed tens of gigabytes, switches may not be the best choice due to their limited memory size. Second, efficiently implementing the entire prefix cache logic, which includes hash calculation, efficient cache copies and eviction, and token matching, on a network device requires specialized knowledge of network programming, device drivers, and network protocols. Finally, developing the efficient and secure algorithms for performing cache offload and eviction requires deep systems knowledge of how the memory operations work, e.g., which memory operations are blocking versus non-blocking, which operations ensure isolation, and which operations ensure high throughput and low latency.

Contributions. The main contribution of this work is a design and implementation a unique cache offload system named **CacheFlow**. CacheFlow addresses the above challenges by enabling network offload for prefix-cache using a programmable network interface card, a.k.a. SmartNICs Kfoury et al. (2024). One of the main reasons for using SmartNICs is that they are capable of hosting tens of gigabytes of memory with specialized interfaces for memory transfer to GPUs, such as GPUDirect RDMA Potluri et al. (2013), and contain sufficient processing power to perform complex logic. In addition, SmartNICs are located in multiple locations in the data center network topology, allowing for a flexible deployment of the prefix cache depending on the service-level objectives and level of cache sharing. (Section A.3 discusses the current state of SmartNICs in modern AI data centers). CacheFlow is implemented as ***a simple drop-in plug-in to the popular vLLM framework*** Kwon et al. (2023), a widely used AI/ML inference framework in both academia and industry, and is available as an open source project [1]

In summary, the contributions are:

- Design of a prefix cache pool system that utilizes programmable network devices realized on a SmartNIC and extensible to a multi node deployment (Section 3).
- An efficient implementation of the design that is available as an open source, easy to install, plug-in (Section 3).
- An evaluation of CacheFlow against vanilla vLLM and LMCache (a host DRAM offload system) covering latency, throughput and memory utilization (Section 4).

2 Related Works & Preliminaries

Recent research in efficient inference systems has identified prefix caching as a pivotal technique that offsets the cost of computational and memory overhead of redundant prompt prefixes by sharing state across concurrent queries. Efficiently managing this shared prefix has motivated a diverse range of strategies spanning from granular memory level optimizations to large scale disaggregated serving infrastructure.

KV cache offload techniques. The linear growth of the memory utilized by the KV cache with increasing sequence length presents a fundamental scalability challenge, as the growing context window rapidly saturates the limited memory on GPUs. Hierarchical storage systems such as InfiniteLLM Lin et al. (2024) and FlexGen Sheng et al. (2023) offload KV tensors to the host CPU's memory and NVMe SSDs to support virtually infinite context lengths. While effective for expanding the context length, these approaches suffer from substantial performance overheads due to dependence on the host CPU. To mitigate this latency overhead, InfiniGen Lee et al. (2024) proposes speculative prefetching mechanisms that attempt to hide data transfer latency by predicting future token

[1]The entire repository information will be available upon acceptance of this work and anonymized code is provided as part of the supplemental material.

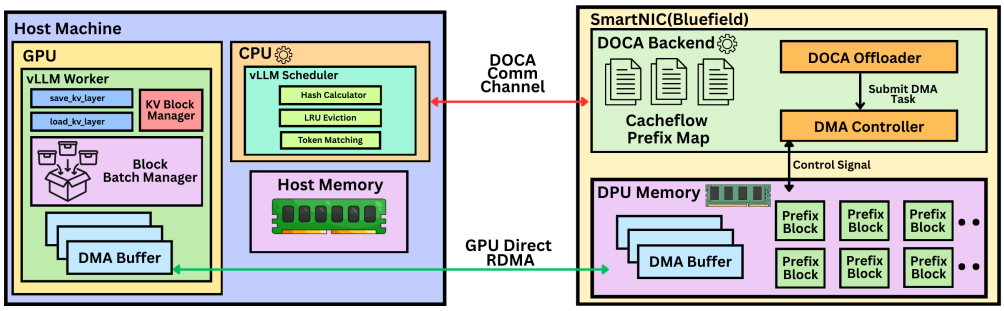


Figure 3: Architectural overview of CacheFlow.

access patterns and overlapping computation with memory retrieval. CacheFlow improves this by completely bypassing the host CPU.

In-Network Computing and Cache The idea of a network-level cache is not new. For example, works like NetCache Jin et al. (2017) and IncBricks Liu et al. (2017) propose offloading key-value databases to programmable network devices. However, these systems mainly focus on small key-value pairs rather than the massive, static tensors required for LLM inference. CacheFlow is the first to leverage the SmartNIC's architecture to create a dedicated, disaggregated prefix cache pool.

3 CacheFlow Overview

By transforming the network infrastructure into an active caching tier, CacheFlow establishes a high performance prefix cache pool that significantly mitigates the redundant recomputations and memory overhead inherent in distributed LLM serving. The overview of CacheFlow is in Figure 3.

3.1 Architecture

CacheFlow extends vLLM's core inference engine interface by implementing the KV Connector v1 abstraction introducing a disaggregated architecture which can be broadly divided into 2 parts: the vLLM integrated Host Environment and the DOCA based DPU Backend. The inference engine's scheduler, which primarily runs as a host CPU process, consists of a Hash Calculator that generates unique identifiers for KV prefixes to generate unique 16 character identifiers for KV prefixes enabling efficient matching of reusable cache blocks across requests, an LRU based cache eviction engine manages memory constraints within the connector's metadata layer tracking block allocations and determining prefix blocks to be evicted from the DPU's on board memory when capacity is exceeded and a token matching logic that leverages computed prefix hashes to perform constant time lookup of previously cached prefix blocks allowing the scheduler to identify opportunities for KV prefix reuse without redundant computation. In addition, a dedicated GPU worker thread is responsible for the physical transfer of KV cache data through a zero copy layer batching strategy and save_kv_layer, load_kv_layer kernels that facilitate efficient bidirectional data movement. The DPU Backend consists of a CacheFlow prefix map which tracks which prefixes are currently present in the DPU memory, a DOCA offloader which orchestrates the DMA transfers according to the control signals received over the DOCA Comm channel communication interface, the DMA controller which manages the hardware DMA engine. The interaction between the host and the DPU is bifurcated into 2 specialized paths:

- *Control Plane (DOCA Comm Channel)*: A low latency channel used for exchanging control messages such as prefix lookup and metadata synchronization.
- *Data Plane (DOCA DMA Transfer)*: A high throughput path dedicated to the asynchronous movement of actual prefix tensors between the host DMA buffers and the DPU memory.

3.2 CacheFlow Pipeline

CacheFlow's pipeline is designed to decouple the heavy data movement allowing the host CPU to orchestrate transfers while the SmartNIC and the GPU handle the high bandwidth data path directly. The lifecycle of a request flows through the following stages:

- *Prefix Lookup and Hash Calculation*: When a prompt arrives at the port an inference engine exposes, CacheFlow's scheduler attempts to identify the reusable KV blocks. The `get_num_matched` computes a hash of the prompt's prefix tokens and queries the CacheFlow Prefix Map residing on the SmartNIC via the DOCA Comm Channel. This choice was made specifically keeping in mind the ultra low latency offered by the control plane that enables CacheFlow to determine whether a KV block already exists on the SmartNIC further preventing the GPU from wasting cycles on the expensive prefill phase knowing that the tensors already exist in the network tier.

- *Metadata Construction and State Update*: Once cache hits are identified, the scheduler calls `build_meta` to prepare the transfer specifications. This step retrieves the memory addresses of the DPU Memory blocks and maps them directly to the GPU's Memory (HBM) addresses.

- *Direct Data Transfer*: CacheFlow worker initiates the fetch operation using `start_load_kv`, which triggers the DOCA DMA Controller to begin a direct RDMA transfer to the GPU.

- *Layer-wise Synchronization*: As the GPU prepares for the prefill phase, it calls `wait_layer_load` ensuring that the KV tensors required by the current layer have fully been populated by the DMA engine, allowing fine grained overlap of compute and communication. The GPU can begin processing the first token, while the DMA engine populates the subsequent tokens effectively masking the transfer latency.

- *Cache Miss Handling*: For new prefixes generated during the request, the worker invokes `save_kv_layer` which triggers a reverse RDMA operation pulling the new tensors from the GPU memory directly into the SmartNIC's internal memory. Finally, the pipeline calls `get_finished` to update LRU eviction policies and clean up the request trackers.

3.3 Performance Optimizations

To ensure that the network offloaded prefix caching does not introduce additional latency CacheFlow employs three critical performance optimization techniques:

- *DOCA Buffer pooling and batching*: Frequent memory allocation and registration operations are expensive in high performance networking contexts. CacheFlow amortizes this cost by implementing a custom DOCA Buffer Pool on the SmartNIC. We pre allocate and pin a fixed pool of memory buffers at initialization time. Furthermore, rather than issuing a DMA transaction for every individual KV block we batch contiguous block requests into larger transfer units. This batching strategy minimizes the overhead of PCIe transaction headers and ensures that we consistently utilize the full bandwidth of the PCIe Gen x16 interface.

- *Layer-wise Asynchronous Pipelining*: Retrieving the entire KV Cache for a long context window (e.g 128k tokens) can incur non trivial network latency. To address this issue CacheFlow initiates asynchronous DMA transfers for all the layers simultaneously synchronizing them individually using CUDA Streams instead of blocking the execution until the full prefix is loaded. This allows the GPU to begin computing attention for Layer i while the KV data for Layer $i + 1$ is still being transferred over the PCIe bus masking the transfer latency for deeper neural networks.

- *Zero Copy GPUDirect RDMA*: Traditional offload systems rely on an intermediate buffer based approach where data is first copied from the storage to the host's RAM and subsequently to the device requesting it. This mechanism consumes double the PCIe bandwidth and suffers from significant kernel and context switching overhead. CacheFlow avoids this allowing the SmartNIC to push the KV tensors directly to the GPU's memory bypassing the kernel and the host CPU.

3.4 Theoretical Efficiency Analysis

To theoretically understand and be able to estimate the benefits of CacheFlow's network offloaded architecture, we adopt a theoretical framework inspired by Pliops' Efficiency model Hillel (2024) for disaggregated KV caching. We define the theoretical speedup as a function of the ratio between the GPU's compute capability and the SmartNIC's I/O retrieval bandwidth. We hypothesize that network offloaded prefix caching is superior to the recomputation or even the retrieval from existing host based offloading systems when the time to retrieve a KV block from the DPU (T_{DPU}) is less than the time required to recompute the attention states of the same block (T_{compute}). We introduce

an acceleration factor (e) defined as the ratio of compute time to retrieval time:

$$e = \frac{T_{\text{compute}}}{T_{\text{DPU}}} \tag{1}$$

We define the time taken to compute T_{compute} as a function of input tokens (N_{in}), model size ($|M|$) and GPU FLOPS (F_{GPU}):

$$T_{\text{compute}} \approx \frac{2 \cdot |M| \cdot N_{\text{in}}}{F_{\text{GPU}}} = N_{\text{in}} \cdot \underbrace{\left(\frac{2 \cdot |M|}{F_{\text{GPU}}}\right)}_{T_{\text{token}}} \tag{2}$$

and T_{DPU} is the time to transfer the equivalent KV cache of size (S_{KV}) over the PCIe bus with a bandwidth (BW_{IO})

$$T_{\text{DPU}} = \frac{S_{\text{KV}}}{BW_{\text{IO}}} \tag{3}$$

Let α represent the prefix cache hit rate which is a fraction of the total context length that is retrieved from a DPU based prefix cache pool ($0 \leq \alpha \leq 1$). The effective Time to First Token ($TTFT_{\text{effective}}$) for a request with N_{in} tokens is modeled as the maximum of the retrieval time for the cached portion and the compute time for the non cached portion.

$$TTFT_{\text{effective}} = \max\left(\underbrace{\frac{\alpha \cdot N_{\text{in}} \cdot T_{\text{token}}}{e}}_{\text{Retrieval (Cache Hit)}}, \underbrace{(1-\alpha) \cdot N_{\text{in}} \cdot T_{\text{token}}}_{\text{Compute (Cache Miss)}}\right) \tag{4}$$

where T_{token} is the compute time per token. The *Theoretical Speedup* (S) compared to a standard GPU only baseline (where $TTFT_{\text{baseline}} = N_{\text{in}} \cdot T_{\text{token}}$) is derived as:

$$S = \frac{TTFT_{\text{baseline}}}{TTFT_{\text{effective}}} = \min\left(\frac{e}{\alpha}, \frac{1}{1-\alpha}\right) \tag{5}$$

Equation (5) also reveals a hardware inflection point: depending on the cache hit rate α, the system shifts from compute-bound (bottlenecked by GPU FLOPS) to I/O-bound (bottlenecked by PCIe/network bandwidth). We define the transition point α^* as the cache hit rate at which the time to retrieve the cached prefix from the DPU equals the time to compute the remaining uncached tokens:

$$\alpha^* = \frac{e}{1+e} = \frac{T_{\text{compute}}}{T_{\text{DPU}} + T_{\text{compute}}} \tag{6}$$

(see Section C.2 for the full derivation). This threshold dictates the optimal hardware provisioning for CacheFlow:

- *When $\alpha < \alpha^*$ (Compute-Bound)*: The prefill phase is dominated by the GPU computing the uncached suffix. In this regime, the system gains no additional benefit from faster I/O speeds (e.g., upgrading to PCIe Gen 5), because the retrieval finishes before the computation.
- *When $\alpha > \alpha^*$ (I/O-Bound)*: The prefill phase is dominated by the time it takes the SmartNIC to push the cached tensors to the GPU. Here, maximum gains are achieved *only* if the network and PCIe bandwidth (BW_{IO}) are scaled up.

Under continuous batching, a lower isolated $TTFT(1)$ also shrinks the prefill interruption penalty imposed on concurrent decodes, letting CacheFlow sustain larger batch sizes without violating TPOT SLOs (see Appendix C.1 for the full model).

Our experimental results (Table 1) are consistent with this efficiency model as at a query rate of QPS=1, CacheFlow yields measured TTFT speedups of 1.75× (Llama-3.2-3B), 1.71× (Ministral-3B), and 5.41× (Qwen3-4B). However, exact speedups vary with each model's compute to retrieval ratio (e).

Model	System	Metric (Avg.)	QPS= 1	QPS= 2	QPS= 3	QPS= 4	QPS= 5	QPS= 6	QPS= 7	QPS= 8	QPS= 9	QPS= 10
Qwen3-4B	vLLM	TTFT (ms)	27,256	48,046	56,270	60,364	62,798	64,417	65,580	66,544	67,155	67,702
		TPOT (ms)	65.41	63.55	63.59	63.57	63.57	63.58	63.57	64.74	63.59	63.59
		Throughput (t/s)	12,931	13,371	13,378	13,385	13,390	13,395	13,398	13,364	13,397	13,398
	LMCache	TTFT (ms)	8,817	32,385	40,542	44,706	47,134	48,840	49,897	50,787	51,559	52,034
		TPOT (ms)	49.92	48.85	49.11	48.88	49.63	48.92	48.85	48.84	48.87	48.88
		Throughput (t/s)	16,652	16,869	16,925	16,889	16,902	16,893	16,918	16,919	16,905	16,912
	CacheFlow	TTFT (ms)	**5,038** (-82%)	**26,283** (-45%)	**38,199** (-32%)	**35,202** (-42%)	**39,765** (-37%)	**42,754** (-34%)	**43,891** (-33%)	**48,610** (-27%)	**42,048** (-37%)	**44,786** (-34%)
		TPOT (ms)	**44.20** (-32%)	**45.69** (-28%)	**49.44** (-22%)	**43.17** (-32%)	**47.27** (-26%)	**48.04** (-24%)	**48.03** (-24%)	**47.56** (-27%)	**43.14** (-32%)	**46.09** (-28%)
		Throughput (t/s)	**17,945** (+39%)	**18,770** (+40%)	**17,723** (+32%)	**20,004** (+49%)	**19,007** (+42%)	**18,729** (+40%)	**18,143** (+35%)	**20,004** (+50%)	**20,324** (+52%)	**19,374** (+45%)
Llama3.2-3B	vLLM	TTFT (ms)	1,242	9,747	16,277	20,015	22,432	24,242	25,474	26,354	27,060	27,632
		TPOT (ms)	40.33	61.42	61.53	63.93	63.76	63.39	63.49	63.49	63.51	63.50
		Throughput (t/s)	19,398	27,697	28,566	28,762	28,773	28,710	28,683	28,692	28,689	28,691
	LMCache	TTFT (ms)	846	2,505	8,512	12,056	14,410	15,525	16,870	17,564	19,529	20,429
		TPOT (ms)	32.75	48.19	50.42	48.75	49.55	49.11	51.98	51.27	51.20	51.61
		Throughput (t/s)	19,575	37,311	38,928	39,369	39,596	40,208	39,776	40,108	39,088	38,739
	CacheFlow	TTFT (ms)	**708** (-43%)	**1,370** (-86%)	**4,809** (-70%)	**7,135** (-64%)	**10,087** (-55%)	**13,115** (-46%)	**15,880** (-38%)	**11,186** (-58%)	**18,168** (-33%)	**13,847** (-50%)
		TPOT (ms)	**26.95** (-33%)	**37.55** (-39%)	**45.32** (-26%)	**44.41** (-31%)	**47.48** (-26%)	**47.69** (-25%)	**46.36** (-27%)	**43.16** (-32%)	**48.13** (-24%)	**43.60** (-31%)
		Throughput (t/s)	**19,651** (+1%)	**36,959** (+33%)	**44,961** (+57%)	**48,083** (+67%)	**44,676** (+55%)	**42,792** (+49%)	**41,660** (+45%)	**49,462** (+72%)	**42,187** (+47%)	**47,927** (+67%)
Ministral	vLLM	TTFT (ms)	653	2,136	8,420	12,504	14,806	16,317	17,579	18,202	18,875	19,367
		TPOT (ms)	23.16	65.10	66.07	72.00	72.75	72.32	72.56	72.11	71.97	71.87
		Throughput (t/s)	19,813	35,990	37,321	37,191	37,229	37,366	37,245	37,671	37,479	37,541
	LMCache	TTFT (ms)	521	833	2,639	5,730	9,214	11,845	13,773	14,245	14,849	13,644
		TPOT (ms)	25.56	33.15	42.98	55.77	58.45	56.50	58.89	59.18	53.05	57.53
		Throughput (t/s)	19,922	37,850	46,120	49,540	48,320	51,400	50,850	51,500	50,200	51,500
	CacheFlow	TTFT (ms)	**381** (-42%)	**399** (-81%)	**712** (-92%)	**3,470** (-72%)	**7,350** (-50%)	**7,280** (-55%)	**10,371** (-41%)	**11,460** (-37%)	**9,505** (-50%)	**11,736** (-39%)
		TPOT (ms)	**16.63** (-28%)	**23.03** (-65%)	**35.28** (-47%)	**49.85** (-31%)	**53.35** (-27%)	**50.80** (-30%)	**53.99** (-26%)	**53.92** (-25%)	**47.94** (-33%)	**52.75** (-27%)
		Throughput (t/s)	**19,922** (+1%)	**39,313** (+9%)	**56,947** (+53%)	**61,632** (+66%)	**56,745** (+52%)	**61,428** (+64%)	**56,786** (+52%)	**56,538** (+50%)	**63,663** (+70%)	**56,843** (+51%)

Table 1: Performance comparison of CacheFlow versus vLLM, and LMCache. We evaluate Time to First Token (TTFT), Time Per Output Token (TPOT), and Throughput across Qwen3, Llama3.2, and Ministral models under varying load (QPS 1–10). Values in parentheses and green text denote the percentage improvement of CacheFlow over the baselines.

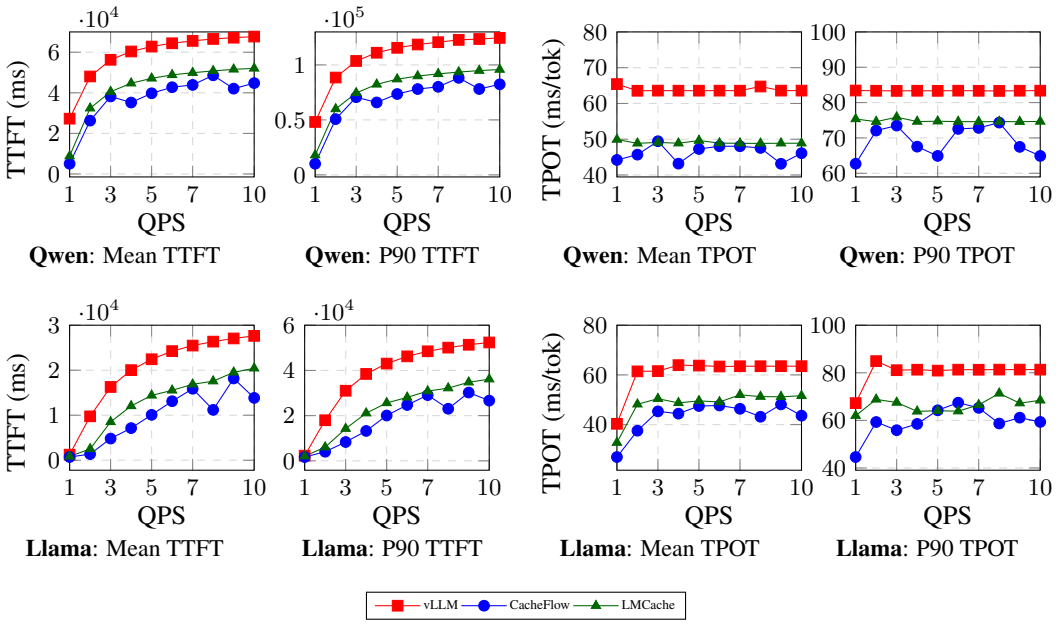
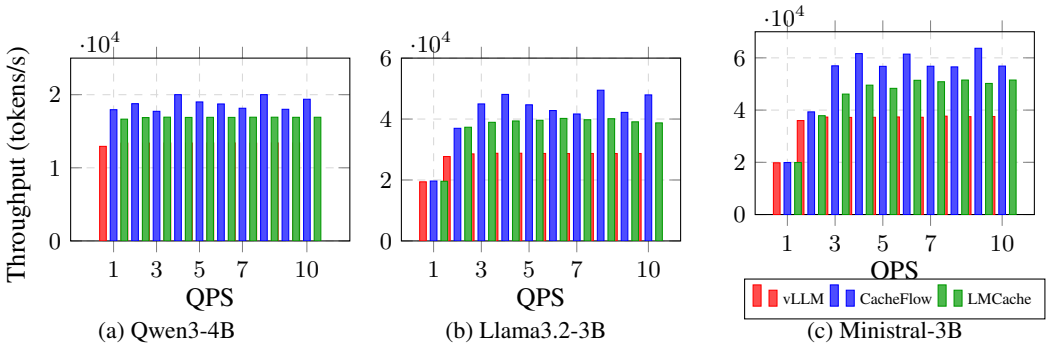


Figure 4: System overall latency (Mean TTFT, P90 TTFT, Mean TPOT, P90 TPOT, lower is better). CacheFlow improves latency by serving prefix KV from SmartNIC memory via DMA.

4 Evaluation

As part of our evaluation, our aim is to answer the following research questions:

- *Q1*: How does CacheFlow affect query serving latency compared to existing inference engines and prefix cache offload solutions?
- *Q2*: How does CacheFlow improve throughput?



(a) Qwen3-4B (b) Llama3.2-3B (c) Ministral-3B

Figure 5: System throughput (tokens/second, higher is better) across varying QPS for three model architectures. CacheFlow significantly improves throughput by reducing redundant recomputation through network offloaded prefix caching.

- *Q3*: How does CacheFlow improve memory storage requirements, I/O transfer rates?

The solutions to the above questions are discussed in Section 4.1, Section 4.2, and Section 4.3 as *S1, S2, S3*, respectively, based on the experimental data presented in Table 1. All the experiments were conducted on 2 testbed setups detailed in Appendix A.1 & A.5.

4.1 Latency Comparison *(S1)*

Latency results comparing baseline systems and CacheFlow using the experimental setup described in Section A.1 are presented in Figure 4. We mainly compare Time to First token (TTFT) and Time per Output Token (TPOT), as TTFT represents the latency that is perceived by the user for streaming queries and TPOT represents the latency mainly associated with the decoding step. More detailed experimental results can be found in Section C.3. **(i)** CacheFlow achieves substantial TTFT reductions across all models by offloading precomputed prefix cache blocks to the DPU and reloading them via DMA on cache hits, avoiding redundant prefill computation. **(ii)** For Llama 3.2-3B, CacheFlow reduces mean TTFT by 43% compared to vanilla vLLM and by 16% compared to LMCache at a request rate of 1. The improvement is most pronounced at QPS=2, where CacheFlow's DMA-accelerated cache hits bypass the prefill bottleneck that dominates both baselines. At higher request rates, the advantage narrows as queuing delays begin to dominate. **(iii)** Although all systems experience increasing TTFT as the request rate increases, CacheFlow's rate of degradation is significantly slower. This stems from the fact that rising request rates result in queuing, and every cache hit request bypasses the GPU prefill pipeline entirely, loading prefix blocks via DMA from the DPU instead. With increasing request rate, the baseline systems queue more prefill jobs on the GPU, and each one blocks every subsequent prefill and active decode. CacheFlow avoids this contention by decoupling the prefix cache loading, allowing the GPU proportionally more time for decoding. **(iv)** While prefix caching primarily targets prefill latency affecting TTFT, CacheFlow demonstrates a consistent reduction in Time Per Output Token (TPOT) compared to both baselines. For the Qwen3-4B model, CacheFlow reduces mean TPOT by 32% at a request rate of 1. The TPOT improvement suggests that by bypassing the GPU-bound prefill phase for shared prefixes, CacheFlow significantly reduces contention for GPU execution units. While in vLLM the scheduler interleaves compute-bound prefill jobs with memory-bound decode jobs, CacheFlow offloads the prefix via DMA, reading and writing directly to GPU memory and freeing the GPU of this contention. Unlike LMCache, which swaps KV blocks through the CPU and host DRAM, CacheFlow's direct-to-DPU DMA minimizes PCIe bus noise and host CPU interruptions.

4.2 Throughput Comparison *(S2)*

Figure 5 and Table 1 illustrate the throughput advantage gained by CacheFlow over existing solutions. Across all three model architectures, CacheFlow maintains a higher token generation rate that scales with increasing request rate. Specifically, for the Ministral-3B model at QPS=9, CacheFlow achieves a peak throughput of 63,663 t/s, representing a 70% improvement over vLLM and a 26% lead over LMCache. This trend is mirrored in the Llama3.2-3B results, where CacheFlow reaches 49,462 t/s at QPS=8 which is a 72% increase over the vanilla vLLM baseline. These gains are primarily driven by CacheFlow's ability to eliminate redundant prefill computations.

Table 2: **Multi-node experiments where multiple GPU servers share single SmartNIC prefix cache (Qwen3-4B) (Percentages results are relative to vLLM baseline).**

	QPS = 1 per server		QPS = 2 per server		QPS = 4 per server	
System	TTFT (ms)	Throughput (t/s)	TTFT (ms)	Throughput (t/s)	TTFT (ms)	Throughput (t/s)
vLLM	97,447	7,055	126,430	6,853	133,024	6,979
CacheFlow	55,274 **[-43%]**	10,525 **[+49%]**	79,582 **[-37%]**	10,633 **[+55%]**	94,301 **[-29%]**	10,487 **[+50%]**

4.3 Memory Requirement & I/O Throughput *(S3)*

CacheFlow significantly reduces the reliance on costly, limited GPU memory by offloading prefix KV caches to the SmartNIC's internal memory, effectively decoupling context storage from physical GPU constraints. By liberating VRAM typically consumed by prefixes CacheFlow increases compute elasticity, allowing the GPU to allocate more resources to active decoding, model weights and higher batch sizes. For the Llama-3.2-3B (using GQA with a hidden dimension of 3072, 32 heads, and 8 key-value heads) a 20,000 token prefix consumes approximately 1.2 GB of memory per unique prefix in half precision making the memory requirement 12GB for 10 such prefixes. By transferring this 12 GB of the available 24GB GPU RAM to the DPU CacheFlow freed up around 50% of required GPU RAM.

4.4 Multi node prefix sharing

We evaluate CacheFlow using a multi node testbed where multiple GPU servers concurrently share a centralized prefix cache hosted on a single SmartNIC(see Appendix A.5 for details) configured for concurrent prefix sharing. CacheFlow consistently outperforms the baseline vLLM deployment as shown in Table 2 by eliminating cross node redundant prefill computations. Each server retrieves precomputed KV from DPU memory over RoCEv2 eliminating the redundant prefix copies that would otherwise need to be maintained on each GPU's memory.

5 Future Works

While CacheFlow one of the first that implements prefix cache pool on network hardware, several potential research directions to further maximize its efficiency:

Hierarchical Cache Orchestration Scaling CacheFlow cluster wide requires managing KV state across distributed switches and SmartNICs, while avoiding stale lookups and redundant prefills. Future iterations can couple CacheFlow with specialized distributed engines like Mooncake Store to maintain fast, low overhead coherence across multiple network nodes (see Appendix C for full architectural considerations).

Cache compression To expand on board DPU capacity one can offload KV quantization and sparsity aware pruning directly onto the SmartNIC's embedded compute engines. Compressing blocks before PCIe transfer substantially amplifies effective network memory and in effect helps speed the transfer of KV blocks reducing latency further(detailed compression and sparsity strategies elaborated in Appendix C).

6 Conclusion

In this work, we present CacheFlow, a prefix cache pool system that uses programmable network devices to achieve significantly lower latency and higher throughput with a smaller overall memory footprint than existing solutions. CacheFlow is implemented as an easy plug-in for vLLM, allowing simple installation on existing deployments. In addition to the implementation, the design of CacheFlow is generalizable and can be applied to other types of network hardware with similar effects. Finally, as AI accelerators and memory become ever more expensive, CacheFlow enables faster LLM inference without additional AI accelerators or RAM and is easily deployable. Thus, we argue that CacheFlow or a variant of CacheFlow will be a necessity in future AI data centers serving exponentially increasing AI and agentic workloads.

AI USE STATEMENT

We used AI tools to proofread and format the paper for readability. We did not use them to form hypotheses, design experiments, or interpret results. All experimental methodology and conclusions were determined by the authors. Other required disclosure tasks are not applicable. The authors take full responsibility for the content of this work.

ETHICS STATEMENT

This work improves the efficiency of LLM inference serving and it does not involve human subjects, new data collection or model training. All experiments use public models and datasets under their original licenses. KV cache blocks are derived from user prompts and CacheFlow inherits the prefix sharing and isolation semantics of vLLM without changing what is cached or who may reuse it. Multi tenant deployments should partition the cache per tenant to avoid privacy breach in a shared cloud environment.

REPRODUCIBILITY STATEMENT

The design of CacheFlow and its vLLM integration are described in Section 3. The hardware, software, models, and baselines are listed in Appendix. Source code and scripts to reproduce all results are available at `https://anonymous.4open.science/r/cacheflow2-C4D0`.

REFERENCES

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

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

Mooncake Authors. Mooncake store. `https://kvcache-ai.github.io/Mooncake/design/mooncake-store.html`, 2024. Accessed: 2026-05-23.

Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. ISSN 0146-4833. doi: 10.1145/2656877.2656890. URL `https://doi.org/10.1145/2656877.2656890`.

Idan Burstein. Nvidia data center processing unit (dpu) architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pp. 1–20, 2021. doi: 10.1109/HCS52781.2021.9567066.

Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, 2016. doi: 10.1109/MICRO.2016.7783710.

Aninda Chatterjee and Vivek V. Designing data centers for ai clusters. White paper, Juniper Networks, November 2023. URL `https://www.juniper.net/documentation/us/en/software/nce/ai-clusters-data-center-design/ai-clusters-data-center-design.pdf`.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*, 2023.

Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025.

Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pp. 51–66, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/nsdi18/presentation/firestone.

Denis Foley and John Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17, 2017. doi: 10.1109/MM.2017.37.

Google Cloud Team. Five techniques to reach the efficient frontier of llm inference. https://cloud.google.com/blog/topics/developers-practitioners/five-techniques-to-reach-the-efficient-frontier-of-llm-inference, 2026. Google Cloud Blog. Accessed: 2026-05-18.

Shanshan Han, Qifan Zhang, Yuhang Yao, Weizhao Jin, and Zhaozhuo Xu. Llm multi-agent systems: Challenges and open problems. *arXiv preprint arXiv:2402.03578*, 2024.

Junda He, Christoph Treude, and David Lo. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–30, 2025.

Eshcar Hillel. Disaggregated KV storage: A new tier for effecient scalable LLM Inference. https://www.youtube.com/watch?v=zJ1OfODySF8, 2024. Pliops Technology Lecture.

Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pp. 121–136, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132764. URL https://doi.org/10.1145/3132747.3132764.

Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589350. URL https://doi.org/10.1145/3579371.3589350.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput llm inference with shared prefixes. *arXiv preprint arXiv:2402.05099*, 2024.

Elie F Kfoury, Samia Choueiri, Ali Mazloum, Ali AlSabeh, Jose Gomez, and Jorge Crichigno. A comprehensive survey on smartnics: Architectures, development models, applications, and research directions. *IEEE Access*, 12:107297–107336, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 155–172, 2024.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 9459–9474. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2020. doi: 10.1109/TPDS.2019.2928289.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large scale language model society. 2023.

Haoyang Li, Yiming Li, Anxin Tian, Tianhao Tang, Zhanchao Xu, Xuejia Chen, Nicole Hu, Wei Dong, Qing Li, and Lei Chen. A survey on large language model acceleration based on kv cache management. *arXiv preprint arXiv:2412.19442*, 2024.

Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, Shen Li, Zhigang Ji, Tao Xie, Yong Li, and Wei Lin. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache, 2024. URL https://arxiv.org/abs/2401.02669.

Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. *SIGARCH Comput. Archit. News*, 45(1):795–809, April 2017. ISSN 0163-5964. doi: 10.1145/3093337.3037731. URL https://doi.org/10.1145/3093337.3037731.

Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pp. 318–333, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359566. doi: 10.1145/3341302.3342079. URL https://doi.org/10.1145/3341302.3342079.

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. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, pp. 38–56, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706141. doi: 10.1145/3651890.3672274. URL https://doi.org/10.1145/3651890.3672274.

Meta AI. Llama 3.2 model card. https://huggingface.co/meta-llama/Llama-3.2-3B, September 2024. Accessed: 2026-08-13.

Mistral AI. Ministral 3 3b instruct 2512 model card. https://huggingface.co/mistralai/Ministral-3-3B-Instruct-2512, December 2025. Accessed: 2026-08-13.

Itay Ozery and Scott Ciccone. Transform the data center for the ai era with nvidia dpus and nvidia doca. NVIDIA Technical Blog, March 2023. URL https://developer.nvidia.com/blog/transform-the-data-center-for-the-ai-era-with-nvidia-dpus-and-nvidia-doca/.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture*, ISCA '24, pp. 118–132. IEEE Press, 2025. ISBN 9798350326581. doi: 10.1109/ISCA59077.2024.00019. URL https://doi.org/10.1109/ISCA59077.2024.00019.

Yajuan Peng, Haoran Wei, Xiaolong Zhong, Junkai Huang, Haohan Xu, Zicheng Wang, Yang Bai, Zhuo Jiang, Jianxi Ye, Xiaoliang Wang, et al. Barre: Empowering simplified and versatile programmable congestion control in {High-Speed}{AI} clusters. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pp. 343–363, 2025.

Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus. In *2013 42nd International Conference on Parallel Processing*, pp. 80–89, 2013. doi: 10.1109/ICPP.2013.17.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, et al. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *ACM Transactions on Storage*, 2024.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180, 2023.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pp. 31094–31116. PMLR, 2023.

Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pp. 173–191, 2024.

Yashar Talebirad and Amirhossein Nadiri. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *arXiv preprint arXiv:2306.03314*, 2023.

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D Nguyen. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2501.06322*, 2025.

Dezhan Tu, Danylo Vashchilenko, Yuzhe Lu, and Panpan Xu. Vl-cache: Sparsity and modality-aware kv cache compression for vision-language model inference acceleration, 2024. URL https://arxiv.org/abs/2410.23317.

vLLM Team. vllm: benchmark_prefix_caching.py. https://github.com/vllm-project/vllm/blob/main/benchmarks/benchmark_prefix_caching.py, 2024.

vLLM Team. vllm documentation: vllm bench serve, 2025. URL https://docs.vllm.ai/en/latest/cli/bench/serve/. Accessed: 2026-02-08.

Chengkun Wei, Xing Li, Ye Yang, Xiaochong Jiang, Tianyu Xu, Bowen Yang, Taotao Wu, Chao Xu, Yilong Lv, Haifeng Gao, et al. Achelous: Enabling programmability, elasticity, and reliability in hyperscale cloud networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pp. 769–782, 2023.

Guanlong Wu, Zheng Zhang, Yao Zhang, Weili Wang, Jianyu Niu, Ye Wu, and Yinqian Zhang. I know what you asked: Prompt leakage via kv-cache sharing in multi-tenant llm serving. In *NDSS*, 2025.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Takeshi Yoshimura, Valentijn Dymphnus van de Beek, and Tatsuhiro Chiba. Accuracy is speed: Towards long-context-aware routing for distributed llm serving, 2026. URL https://arxiv.org/abs/2604.15732.

Yangyang Yu, Zhiyuan Yao, Haohang Li, Zhiyang Deng, Yuechen Jiang, Yupeng Cao, Zhi Chen, Jordan Suchow, Zhenyu Cui, Rong Liu, et al. Fincon: A synthesized llm multi-agent system with conceptual verbal reinforcement for enhanced financial decision making. *Advances in Neural Information Processing Systems*, 37:137010–137045, 2024.

Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab Mirrokni. Turboquant: Online vector quantization with near-optimal distortion rate. *arXiv preprint arXiv:2504.19874*, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

A Appendix

A.1 Experimental Setup

Testbed. Our testbed is equipped with an AMD Ryzen 9 5900 CPU (12 cores, 24 threads at 3.0 GHz), 32 GB of DDR4 memory, and an NVIDIA RTX 4090 GPU with 24 GB of GDDR6X memory connected via PCIe Gen 4.0. The software environment consists of Ubuntu 24.04.3 LTS with CUDA 13.1 and Driver 590.48.01 and PyTorch 2.11.0. For the baseline inference engines, we employ vLLM (v0.1.dev13303) and LMCache (v0.3.13) ensuring that our evaluations leverage the latest optimizations in distributed KV cache management. Specifically for CacheFlow, we utilized an NVIDIA BlueField-3 DPU SmartNIC, which features 16 Arm cortex A78 cores with 32GB of DDR5 RAM and 128GB of storage with Nvidia DOCA 3.1 installed.

Cache System Configurations. To demonstrate the performance gains of CacheFlow we compare it against vLLM with automatic prefix caching enabled, which relies on the GPUs internal memory to cache prefixes, and LMCache Cheng et al. (2025). For LMCache, the configuration used was maximum local cpu size was set to 20 and local cpu being set to true with the chunk_size being 256 using the *LMCacheConnectorV1* whereas vanilla vLLM relies solely on the GPU's built in automatic prefix caching with no external KV offloading. For CacheFlow, we reserve 512MB blocks on the SmartNIC RAM with up to 35 blocks available (≈17.5 GB) with 2,048 tokens per block, asynchronous DMA transfers with a pool of 4 copy streams and DOCA buffer pooling with 16 staging buffers. For all systems, the inference engine was configured with the following common parameters: maximum sequence length of 32000 tokens and GPU memory utilization of 95%.

Model Configuration. To evaluate the versatility of CacheFlow across different architectures and parameter scales, we select three state-of-the-art small language models (SLMs): **Qwen3-4B** Yang et al. (2025), **Llama-3.2-3B** Meta AI (2024), and **Ministral-3-3B-Instruct-** Mistral AI (2025). These models were chosen because their memory footprints allow for high throughput serving on a single NVIDIA RTX 4090, while their varying attention mechanisms and hidden dimensions provide a diverse testbed for KV cache management. All models were deployed using their official Hugging Face weights in BFloat16 precision to maintain numerical stability during the KV cache offloading and reloading process.

Workload Configuration. We evaluate all three systems using vLLM's built in serving benchmark (`vllm bench serve` vLLM Team (2025)) with the *prefix_repetition* dataset vLLM Team (2024), which is designed to stress test prefix cache under workloads with varying prefix sizes and load. For our evaluation, we configured *prefix_repetition* workload to generate a pool of $P = 10$ unique prefixes, each comprising $L_p = 20000$ tokens. A total of $N = 100$ requests are constructed by randomly assigning each request one of the 10 prefixes, yielding approximately 10 requests per unique prefix. Each request consists solely of the shared prefix ($L_s = 0$ suffix tokens) and is configured to generate $L_o = 100$ output tokens. This results in a total of $2{,}000{,}000$ input tokens and 10000 generated tokens per benchmark run. The workload is representative of production scenarios including multi turn conversations, system prompts, and retrieval augmented generation (RAG) documents. These parameters were chosen to ensure that the total KV cache size can be bigger than the GPU RAM, allowing for prefix cache offload to be effective. Requests arrive according to a Poisson process (burstiness factor 1.0) at a configurable target rate λ. We configure λ from 1 to 10 queries per second (QPS) to evaluate system behavior on increasing load levels. All runs use a fixed

seed and set `PYTHONHASHSEED=0` to ensure reproducible request ordering and prefix assignment across the 3 systems.

A.2 Installation of CacheFlow vLLM Plug-in

CacheFlow is designed as a lightweight easy to use plug-in which unlike complex infrastructure overhauls that require custom kernels or modified PyTorch distributions operates as a modular extension to the existing vLLM Inference engine. The deployment workflow needs installation of NVIDIA CUDA Toolkit and DOCA drivers along with the DOCA Host package. No software installation is required on the BlueField SmartNIC as it comes with the firmware and the OS image installed out of the box. After successfully verifying these package installations the user needs to install PyTorch and vLLM. Post setting up the platform requirements Cacheflow can be built from source on the host and used as a vLLM instance through regular vLLM serving commands by passing the **–kv-transfer-config** flag with the arguments set as "kv_connector": "CacheFlowConnectorV1","kv_role": "kv_both","dpu_pci_addr":"your DPU PCI BDF","prefix_map_path":"Location of the prefix map on the DPU". Additional configuration can be passed in through the same object allowing the admin to configure variety of parameters depending on the model type and workloads being run. Besides this, the DOCA Backend is written in C, which makes it easy to compile using any of the popular compilers and directly run it and start the DOCA Offloader.

A.3 Are SmartNICs Popular in AI Data Centers?

SmartNICs have become a critical component in modern AI data centers, driven by the need to maximize the efficiency of high-value computing resources. In distributed deep learning training, network protocol processing can consume up to 30% of the host CPU cycles, creating a bottleneck that starves GPUs of data. By offloading network virtualization, security, and storage functions to SmartNICs, providers such as AWS (via the Nitro System) and Azure have effectively standardized these devices to ensure bare-metal performance for AI workloads Firestone et al. (2018); Caulfield et al. (2016). Furthermore, modern SmartNICs accelerate key-value stores and data aggregation within the network fabric, completely bypassing host memory latency Liu et al. (2019). Finally, GPU vendors such as Nvidia are beginning to sell complete AI data center solutions that include a tight coupling between SmartNICs and GPUs Ozery & Ciccone (2023). Therefore, they are no longer optional optimizations but architectural prerequisites for hyperscale AI infrastructure.

A.4 KV Cache Size

Large Language model inference consists of two distinct phases: Prefill and Decode. Each phase exhibits fundamentally different computational characteristics that define the system best suited for processing tokens in each phase. The prefill phase which involves computing the initial attention scores and populating the KV cache for the prompt is compute bound as it process all the input tokens in parallel as a large matrix multiplication operation and the latency associated with it being referred to as the Time to First Token (TTFT). The decode phase which computes the subsequent tokens is memory bandwidth bound with it's latency referred to as Time per output token (TPOT). The size of the KV cache generated can be approximated by the following equation-

$$M_{\text{KV}} \approx 2 \cdot b \cdot s \cdot L \cdot n_{\text{KV}} \cdot d_{\text{head}} \cdot P_{\text{bytes}} \tag{7}$$

where:
M_{KV}: Total KV cache size in bytes.
b: Batch size.
s: Sequence length.
L: Number of layers
d_{head}: Head dimension.
P_{bytes}: Precision in bytes.
n_{KV}: KV heads ($= n_{\text{heads}}$ for MHA, $= 1$ for MQA).

A.5 Multi node prefix sharing testbed setup

Multinode Experiment Testbed. The multi-node experiments (Section 4) use two NVIDIA DGX Spark servers, each with a GB10 Grace Blackwell superchip, 128 GB of unified LPDDR5X memory and a ConnectX-7 NIC, and one NVIDIA BlueField-3 DPU (16 Arm Cortex-A78 cores, 32 GB DDR5) hosted in a separate x86 server and operated in DPU mode. Each server is connected directly to one BlueField-3 port over RoCE v2. Both servers run Ubuntu 24.04, CUDA 13.0 (driver 580.159.03), PyTorch 2.11.0 and vLLM 0.20.1; the BlueField-3 runs DOCA 3.2.1.

B PRELIMINARIES

In this section, we provide the high-level background on KV cache, prefix caching, cache offload, and the architecture of SmartNICs that enables network-side offloading.

B.1 PREFIX CACHING AND CACHE OFFLOAD

KV Cache. The generation process of Decoder-only Transformer models consists of two distinct phases: *prefill* and *decode*. In the prefill phase, the model processes the input prompt in parallel to compute the initial hidden states. In the decode phase, the model generates tokens autoregressively, one at a time. Both phases rely on the self-attention mechanism, which is defined as $Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$. Representing this with respect to a single query vector q_t at step t (or token t), the model generates a query vector q_t which must attend to the keys (k) and values (v) of all current and preceding tokens. The output o_t is computed as a weighted sum of value vectors:

$$o_t = \sum_{j=1}^{t} \alpha_{tj} v_j, \quad \text{where} \quad \alpha_{tj} = \text{softmax}\left(\frac{q_t \cdot k_j^T}{\sqrt{d_k}}\right)$$

Here, α_{tj} represents the attention weight (relevance) of the token at position j to the current query at t.

To avoid recomputing values of k_j and v_j for all previous positions ($j < t$) at every step of generation, modern inference engines maintain a ***Key-Value (KV) Cache*** that stores these tensors in memory, allowing the computation for q_t to simply retrieve $K_{<t}$ and $V_{<t}$ rather than re-calculating them.

Prefix Caching. In multi-agent systems and multi-turn conversations, requests often share a significant portion of their input tokens—referred to as the *prefix* (e.g., system prompts or shared context). Without optimization, the inference engine treats every request as independent, redundantly performing the compute-intensive prefill phase to generate k and v vectors for these identical prefixes. Prefix caching addresses this by persisting the KV cache of the prefix tokens. Mathematically, K is decomposed into a static prefix component and a dynamic suffix component:

$$K = [K_{\text{prefix}}; K_{\text{new}}]$$

When a new request arrives with a matching prefix of length L_p, the system bypasses the computation of the indices $1 \dots L_p$. The query q_t (where $t > L_p$) simply attends to the pre-computed K_{prefix} retrieved from the KV cache pool.

Cache Offload. To alleviate GPU memory pressure, *cache offload* strategies move evicted or less frequently used KV blocks to a secondary storage hierarchy, typically the host CPU DRAM Sheng et al. (2023); Lin et al. (2024). While this expands capacity, retrieving data from host RAM introduces latency bottlenecks due to PCIe bandwidth constraints and CPU orchestration overhead, often negating the performance benefits for latency-sensitive applications.

B.2 SMARTNIC AND NETWORK OFFLOAD

SmartNIC Architecture. Modern data centers increasingly deploy Smart Network Interface Cards (SmartNICs) or Data Processing Units (DPUs), such as the NVIDIA BlueField Kfoury et al. (2024). Unlike traditional NICs, which are fixed-function packet movers, SmartNICs are equipped with programmable general-purpose compute cores (e.g., ARM processors), on-board DRAM, and specialized hardware accelerators. A critical feature for our work is the SmartNIC's Direct Memory Access (DMA) engine. Using frameworks like NVIDIA Data Center on a Chip Architecture (DOCA) Burstein (2021), the SmartNIC can initiate direct read/write operations to the host's or GPU's memory (via GPUDirect RDMA Potluri et al. (2013)) without involving the host CPU.

Network Offload. Network offload leverages the SmartNIC's unique position as the gateway to the server. By migrating infrastructure tasks—such as storage management, security isolation, network offload techniques achieves two primary benefits:

- *Zero-Host Overhead:* Computation and Data movement is handled by the SmartNIC's DMA engines, freeing host CPU cycles and host RAM for other tasks.

- *Architectural Isolation:* The SmartNIC provides a separate, dedicated memory pool (on-board DRAM) that is closer to the network fabric than the host CPU, enabling efficient "in-network" storage and retrieval of shared state across distributed systems.

C Discussion and Future Works

Availability of SmartNICs in AI Data Centers A common concern regarding network offloaded caching has always been whether it is feasible to deploy specialized hardware across massive computing clusters. However SmartNICs have already become a critical and standard component in modern AI data centers Kfoury et al. (2024). In fact, the latest hyperscale AI clusters which support tens of thousands of GPUs already rely heavily on programmable high speed NICs operating at 400 Gbps and beyond to manage complex, high throughput tasks like dynamic congestion control Peng et al. (2025). As major cloud providers and hardware vendors already build their platforms around this technology to guarantee performance at scale, deploying CacheFlow does not require a disruptive or costly hardware upgrade. It simply takes advantage of the existing hardware backbone of today's AI infrastructure.

Hierarchical Cache Orchestration. CacheFlow leverages a distributed network cache to address the increasing memory demands of agentic workloads, effectively extending KV cache capacity beyond local GPUs and establishing a cluster wide shared prefix cache. While this architecture offers significant scalability advantages, offloading state to multiple SmartNICs raises concerns around consistency. Overcoming these coherence and consensus hurdles is critical to preventing stale reads and redundant prefix recomputation across the cluster and so we back CacheFlow with Mooncake Store Authors (2024); Qin et al. (2024). Compared to generalized in-memory stores like Redis or Memcached, Mooncake Store is designed specifically as a distributed KV storage engine, providing the robust multi replica support and strong consistency guarantees necessary for our architecture. In traditional caching systems, the lookup key is typically derived from the value through hashing, meaning the value is immutable after inserting whereas Mooncake Store provides low level object storage which includes configurable caching and eviction strategies offering high memory efficiency designed to accelerate LLM inference performance. For CacheFlow this architectural choice is highly advantageous as it guarantees consistency while allowing the network hardware to fetch and serve cached prefixes with minimal latency, maximizing the efficiency of our offloaded architecture.

Scalability and Semantic Routing As massive multi agent systems scale in complexity and the volume of requests they serve relying on traditional round robin or simple hash based load balancers could hamper inference performance at scale. If a user's prompt is blindly routed to a compute node that lacks the relevant agent's prefix cache it triggers an expensive avoidable cache miss. This forces the system to either fetch massive KV tensors across the network or recompute them entirely on the GPU, negating the advantages of CacheFlow. To maintain the low Time-to-First-Token (TTFT) and high throughput at scale, request steering must become intrinsically prefix aware Google Cloud Team (2026), Yoshimura et al. (2026). In future iterations of CacheFlow we plan to integrate semantic routing directly into the programmable SmartNIC cache layer via ingress controller that is accelerated by the DPU. Instead of treating all network destinations equally an intelligent semantic router can analyze an incoming prompt's intent or required agent's role and dispatch the query specifically to the server pod already containing those exact KV caches. By logically partitioning the distributed cache pool based on task semantics rather than random distribution, CacheFlow could drastically minimize cache duplication and avoids unnecessary cross rack memory fetches. As workload heterogeneity increases this intent driven clustering ensures the distributed SmartNIC memory pool is maximally utilized allowing the system to scale cluster wide without experiencing cache thrashing Yoshimura et al. (2026).

Multi tenant Security Public cloud deployments routinely run concurrent agents from different organizations on the same shared physical infrastructure. Future iterations of CacheFlow could leverage the DPU's hardware enforced memory domains such as SR-IOV and ARM TrustZone to guarantee the rigid isolation of proprietary prompt caches Wei et al. (2023). Without such partitioning shared LLM infrastructure could remain vulnerable to cross tenant side channel attacks and prompt leakage rendering network level caching not feasible for sensitive production workloads Wu et al. (2025). CacheFlow's modular design and integration with vLLM means it could perform these security verifications on the SmartNIC itself, ensuring the gains from prefix cache offloading do not get bottlenecked by host level CPU overhead.

Fault Tolerance and Disaster Recovery For production grade LLM serving the architecture must also have robust fault tolerance against hardware issues such as unexpected NIC resets or localized power fluctuations. The failure of a NIC hosted cache means the GPU must recompute the lost KV vectors, abruptly shifting the service from being memory bound back to being compute bound. This sudden penalty can delay concurrent queries leading to cascading tail latency spikes that easily violate strict Service Level Agreements (SLAs) Sun et al. (2024). To protect against such occurrences depending on the environment and targeted SLOs a straightforward disaster recovery extension could introduce proactive lightweight cache replication which is facilitated by Mooncake store. By utilizing background RDMA bandwidth to mirror and store hot KV blocks on secondary nodes, CacheFlow could ensure high availability without interrupting the primary serving path.

Cache compression Pairing CacheFlow with KV cache quantization and compression techniques Liu et al. (2024); Zandieh et al. (2025) could drastically multiply the effective storage capacity of the network layer. As agentic workloads grow increasingly multi-modal, CacheFlow can leverage the inherent attention sparsity they present. Recent studies Tu et al. (2024) demonstrate that vision workloads allow up to 90% of the KV cache to be safely pruned without any degradation in accuracy by assigning a sparsity aware layer wise KV budget and a token scoring policy that evicts less critical tokens. Furthermore, executing this quantization and compression directly on the SmartNICs before transmission could significantly reduce the host to NIC I/O latency using techniques provided by the DOCA SDK and the compression/decompression hardware accelerators present on these devices. Ultimately, this combined approach ensures that massive context windows are stored compactly in the network fabric without saturating the host's PCIe bandwidth or bottlenecking the GPU during the prefill phase.

C.1 Continuous batching

Under continuous batching the decode phase of active requests is frequently interrupted by the newly arriving requests. Assuming a target batch size of B concurrent requests and an average output sequence length of N_{out}, we can model the expected prefill latency $\mathbb{E}[TTFT(B)]$ under load. If requests arrive uniformly the expected queueing delay added by interleaving a new prefill into a batch of size B scales the effective TTFT multiplicatively:

$$\mathbb{E}[TTFT(B)] \approx TTFT(1) \cdot \left(1 + \frac{B-1}{N_{\text{out}}}\right) \tag{8}$$

Furthermore, the Time Per Output Token ($TPOT$) degrades linearly as the batch size increases. This degradation is driven by two factors: memory bus contention from fetching the KV cache of other concurrent requests (where R_{KV} is the cache size and BW_{HBM} is memory bandwidth), and the compute interruptions caused by new prefill jobs. The expected decoding time under load is modeled as:

$$\mathbb{E}[TPOT(B)] \approx TPOT(1) + (B-1) \cdot \left(\frac{TTFT(1)}{N_{\text{out}}} + \frac{R_{\text{KV}}}{BW_{\text{HBM}}}\right) \tag{9}$$

These equations formally justify the end to end performance gains of CacheFlow. By offloading prefix caching to the SmartNIC the system drastically reduces the base isolated latency $TTFT(1)$. Equation 9 dictates that minimizing $TTFT(1)$ directly shrinks the prefill interruption penalty ($\frac{TTFT(1)}{N_{\text{out}}}$) imposed on the entire decode batch. This mathematical relationship demonstrates that resolving the prefill bottleneck at the network layer inherently protects the GPU's streaming multiprocessors from compute starvation allowing CacheFlow to sustain higher concurrent batch sizes (B) without violating strict TPOT service level objectives.

C.2 Derivation of the Transition Point

The transition point α^* is the cache hit rate at which the time taken by the DPU to retrieve the cached prefix over the PCIe bus equals the time taken by the GPU to compute the remaining uncached tokens. Equating the two terms inside the maximum of Equation (4):

$$\frac{\alpha^* \cdot N_{\text{in}} \cdot T_{\text{token}}}{e} = (1-\alpha^*) \cdot N_{\text{in}} \cdot T_{\text{token}} \tag{10}$$

Simplifying this relation yields the transition threshold in terms of the acceleration factor e:

$$\frac{\alpha^*}{e} = 1 - \alpha^* \implies \alpha^* = \frac{e}{1+e} \tag{11}$$

Substituting the definition of the acceleration factor expresses α^* directly in terms of the hardware latencies:

$$\alpha^* = \frac{T_{\text{compute}}/T_{\text{DPU}}}{1 + T_{\text{compute}}/T_{\text{DPU}}} = \frac{T_{\text{compute}}}{T_{\text{DPU}} + T_{\text{compute}}} \tag{12}$$

This is the result stated in Equation (6).

Table 3: KV Cache Memory Requirements (GB) across Architectures

Model	**Size**	**L**	n_{KV}	d_h	**Batch**	**8K**	**32K**	**128K**
Llama 3.1	70B	80	8	128	1	0.25	1.00	4.00
					32	8.00	32.00	128.00
					64	16.00	64.00	256.00
	8B	32	8	128	1	0.10	0.40	1.60
					32	3.20	12.80	51.20
					64	6.40	25.60	102.40
Qwen 2.5	72B	80	8	128	1	0.25	1.00	4.00
					32	8.00	32.00	128.00
					64	16.00	64.00	256.00
	7B	28	4	128	1	0.04	0.17	0.70
					32	1.40	5.60	22.40
					64	2.80	11.20	44.80
Gemma 2	27B	46	16	128	1	0.29	1.15	4.60
					32	9.20	36.80	147.20
					64	18.40	73.60	294.40
	9B	42	8	128	1	0.13	0.52	2.10
					32	4.20	16.80	67.20
					64	8.40	33.60	134.40

C.3 Full Experimental Details

Below, we add the full experimental results that include inter-token latency (ITL), P50 and P99 results for all 3 models (Qwen, Llama, and Ministral).

Table 4: Qwen/Qwen3-4B Comprehensive Prefix Repetition Benchmark Results for CacheFlow, LMCache, and vLLM

System	QPS	1	2	3	4	5	6	7	8	9	10
CacheFlow	Mean TTFT	5038.08	26283.58	38199.02	35202.11	39765.80	42754.97	43891.90	48610.41	42048.27	44786.79
	P50 TTFT	4226.52	21823.35	34640.33	31907.40	36536.76	39909.54	41142.67	46086.33	39453.36	42202.67
	P90 TTFT	10115.53	50732.47	70643.52	65871.80	73524.41	78044.52	80095.21	88211.31	78090.57	82358.19
	P99 TTFT	11971.77	55284.52	76656.79	72053.97	82215.27	87107.78	89433.57	94841.21	85917.39	90299.57
	Mean TPOT	44.20	45.69	49.44	43.17	47.27	48.04	48.03	47.56	43.14	46.09
	P50 TPOT	44.39	42.40	45.39	40.23	44.40	44.73	44.72	42.43	40.24	44.43
	P90 TPOT	62.69	72.08	73.49	67.53	64.91	72.55	72.83	74.34	67.47	64.91
	P99 TPOT	77.69	87.98	96.03	76.33	74.87	74.91	74.88	88.20	76.31	74.98
	Mean ITL	44.20	45.69	49.51	43.17	47.27	48.04	48.03	47.56	43.14	46.10
	P50 ITL	25.01	27.02	27.51	27.81	28.01	28.03	28.01	27.30	27.86	28.00
	P90 ITL	133.02	81.57	169.70	32.69	148.32	153.25	155.19	153.58	32.61	80.75
	P99 ITL	270.79	269.24	271.35	266.34	272.26	272.17	272.31	270.62	265.81	271.00
LMCache	Mean TTFT	8817.86	32385.70	40542.78	44706.08	47134.57	48840.35	49897.47	50787.93	51559.26	52034.20
	P50 TTFT	6150.43	29199.27	38145.48	42596.98	45275.02	47153.66	48307.71	49288.26	50126.70	50648.02
	P90 TTFT	17948.48	59909.12	74504.44	82316.42	87035.65	89863.75	91861.76	93486.73	94833.38	95788.06
	P99 TTFT	20008.10	67457.71	83729.06	92314.82	97219.91	100621.26	102826.77	104605.39	106089.49	107150.75
	Mean TPOT	49.92	48.85	49.11	48.88	49.63	48.92	48.85	48.84	48.87	48.88
	P50 TPOT	47.90	48.66	48.52	48.68	48.59	48.61	48.63	48.61	48.86	48.66
	P90 TPOT	75.36	74.60	75.87	74.64	74.72	74.62	74.59	74.62	74.60	74.66
	P99 TPOT	86.67	86.58	86.50	86.57	86.55	86.52	86.54	86.51	86.51	86.63
	Mean ITL	49.92	48.85	49.06	48.88	49.68	48.92	48.85	48.84	48.87	48.88
	P50 ITL	27.83	27.93	28.01	27.95	26.84	27.93	27.94	27.93	27.93	27.92
	P90 ITL	182.99	42.81	43.75	43.15	134.84	46.91	46.79	42.98	46.90	42.98
	P99 ITL	308.36	302.32	301.30	302.71	305.98	302.96	301.93	301.72	302.04	302.97
vLLM	Mean TTFT	27256.02	48046.34	56270.72	60364.47	62798.95	64417.26	65580.69	66544.93	67155.50	67702.93
	P50 TTFT	24563.81	45765.13	54593.88	59097.11	61769.30	63553.22	64830.48	65806.00	66560.65	67161.54
	P90 TTFT	48144.74	88519.76	103586.48	111073.84	115561.71	118543.56	120672.34	122761.58	123560.01	124560.23
	P99 TTFT	52720.53	96731.87	113291.48	121521.65	126458.19	129733.07	132072.12	134273.54	135250.41	136348.27
	Mean TPOT	65.41	63.55	63.59	63.57	63.57	63.58	63.57	64.74	63.59	63.59
	P50 TPOT	65.51	64.55	64.55	64.56	64.54	64.55	64.53	64.40	64.55	64.54
	P90 TPOT	83.42	83.35	83.32	83.35	83.33	83.37	83.34	83.31	83.36	83.37
	P99 TPOT	96.81	96.98	97.03	96.96	96.98	97.00	97.01	97.00	96.95	97.00
	Mean ITL	65.41	63.55	63.59	63.57	63.57	63.58	63.57	64.74	63.59	63.59
	P50 ITL	28.09	28.07	28.08	28.08	28.07	28.08	28.07	28.08	28.08	28.07
	P90 ITL	222.54	218.30	218.38	218.76	218.25	218.54	218.29	220.58	218.73	218.48
	P99 ITL	278.08	277.98	278.08	278.04	278.39	277.91	278.21	280.10	277.72	278.41

Table 5: Meta-llama/Llama-3.2-3B Comprehensive Prefix Repetition Benchmark Results for CacheFlow, LMCache, and vLLM

System	QPS	1	2	3	4	5	6	7	8	9	10
CacheFlow	Mean TTFT	708.08	1370.75	4809.45	7135.75	10087.67	13115.72	15880.52	11186.91	18168.93	13847.95
	P50 TTFT	150.54	158.25	4834.39	8040.09	8878.55	13162.51	14688.69	9622.79	19141.66	12530.68
	P90 TTFT	1690.74	4036.92	8334.67	13230.84	20063.20	24710.55	29081.13	23039.25	30217.12	26647.24
	P99 TTFT	2319.53	8529.11	9856.50	15016.89	22948.71	28771.71	32427.60	26552.29	34844.59	30354.61
	Mean TPOT	26.95	37.55	45.32	44.41	47.48	47.69	46.36	43.16	48.13	43.60
	P50 TPOT	22.05	32.88	45.31	45.40	45.78	45.68	44.78	44.46	47.01	44.42
	P90 TPOT	44.6	59.31	55.88	58.52	64.24	67.44	65.26	58.68	61.15	59.35
	P99 TPOT	68.96	83.42	64.41	66.85	74.54	74.84	80.63	73.00	70.92	70.56
	Mean ITL	26.95	37.59	45.28	44.41	47.47	47.69	46.36	43.16	48.13	43.60
	P50 ITL	17.96	28.87	33.24	32.99	33.96	33.64	33.16	33.33	34.97	33.32
	P90 ITL	31.5	34.51	48.02	50.43	50.43	50.50	50.46	50.39	50.94	50.35
	P99 ITL	176.66	197.22	199.65	196.20	195.27	212.47	200.19	182.11	195.49	182.98
LMCache	Mean TTFT	846.3	2505.47	8512.96	12056.72	14410.90	15525.43	16870.94	17564.64	19529.76	20429.57
	P50 TTFT	311.68	2197.55	9741.79	13042.53	15434.13	16136.12	17846.73	18692.53	20579.85	21559.92
	P90 TTFT	2084.40	6043.74	14252.73	21086.45	25611.70	28077.66	30849.95	32175.55	34768.42	36122.24
	P99 TTFT	2933.91	8123.36	16542.32	24327.50	28979.06	31590.03	34487.98	35790.42	38566.15	40139.85
	Mean TPOT	32.75	48.19	50.42	48.75	49.55	49.11	51.98	51.27	51.2	51.61
	P50 TPOT	24.82	47.51	48.6	47.51	48.69	48.5	48.77	48.67	48.28	47.96
	P90 TPOT	61.99	68.83	67.52	63.8	64.05	63.87	66.51	71.35	67.24	68.44
	P99 TPOT	80.21	83.62	83.08	78.52	80.67	77.12	82.19	76.07	74.89	76.67
	Mean ITL	32.75	48.19	50.42	48.75	49.55	49.06	51.98	51.27	51.2	51.61
	P50 ITL	20.52	32.81	35.06	32.81	32.96	32.91	33.04	33.11	33.08	33.11
	P90 ITL	31.81	48.33	47.95	47.98	47.93	47.86	47.93	50.5	50.45	50.42
	P99 ITL	231.43	288.52	289.29	291.18	284.22	265.11	303.34	296.65	290.56	338.17
vLLM	Mean TTFT	1242.34	9747.65	16277.82	20015.74	22432.81	24242.09	25474.76	26354.56	27060.85	27632.74
	P50 TTFT	1304.01	10035.32	16222.45	19816.20	22260.41	24159.62	25337.11	26194.91	26879.76	27438.74
	P90 TTFT	2281.53	17963.01	31004.97	38446.50	43016.86	46221.36	48466.26	50076.14	51352.57	52367.81
	P99 TTFT	6034.28	21048.75	35505.85	43359.75	48334.60	51814.96	54263.43	56030.41	57425.56	58532.95
	Mean TPOT	40.33	61.42	61.53	63.93	63.76	63.39	63.49	63.49	63.51	63.50
	P50 TPOT	37.38	58.30	58.74	58.66	58.70	58.36	58.65	58.74	58.73	58.72
	P90 TPOT	67.26	84.90	81.09	81.30	80.92	81.32	81.28	81.34	81.35	81.33
	P99 TPOT	69.77	103.03	97.36	95.60	91.32	91.45	91.33	91.35	91.42	91.40
	Mean ITL	40.33	61.42	61.69	63.93	63.76	63.54	63.49	63.49	63.51	63.50
	P50 ITL	22.72	35.18	35.20	35.16	35.30	35.21	35.20	35.32	35.28	35.31
	P90 ITL	134.98	164.45	164.61	168.70	167.15	165.22	165.22	165.28	165.26	165.48
	P99 ITL	188.39	204.24	202.92	205.00	204.63	204.96	207.61	207.66	205.78	207.57

Table 6: Ministral-3-3B-Instruct Comprehensive Prefix Repetition Benchmark Results for CacheFlow, LMCache, and vLLM

System	QPS	1	2	3	4	5	6	7	8	9	10
CacheFlow	Mean TTFT	381.08	399.52	712.21	3470.33	7350.75	7280.76	10371.86	11460.60	9505.72	11736.19
	P50 TTFT	73.24	86.04	292.04	3551.01	7631.93	7613.62	11236.74	12386.60	10197.75	12484.47
	P90 TTFT	1276.49	1266.24	2182.99	6195.00	13269.95	12939.09	17452.16	18772.78	15756.99	20393.67
	P99 TTFT	1814.74	1609.86	2864.86	6918.67	14431.69	14860.01	19953.24	21614.53	18939.38	23524.83
	Mean TPOT	16.63	23.03	35.28	49.85	53.35	50.80	53.99	53.92	47.94	52.75
	P50 TPOT	12.87	20.14	37.49	53.73	56.68	51.13	56.87	56.15	50.91	52.03
	P90 TPOT	32.43	45.23	49.11	57.65	65.68	60.41	67.45	64.12	53.50	62.72
	P99 TPOT	47.04	46.90	53.59	67.28	72.39	63.88	69.91	67.26	59.30	68.95
	Mean ITL	16.63	23.03	35.28	49.85	53.35	50.80	54.26	53.92	47.94	52.79
	P50 ITL	12.94	18.80	29.06	49.55	47.54	45.94	47.18	46.73	48.11	47.72
	P90 ITL	23.04	33.42	51.72	56.27	58.45	55.94	56.36	56.44	56.18	56.34
	P99 ITL	155.12	171.10	164.40	179.86	232.80	222.68	232.68	232.91	170.32	222.99
LMCache	Mean TTFT	521.62	833.65	2639.18	5730.40	9214.22	11845.00	13773.00	14245.00	14849.00	13644.00
	P50 TTFT	67.22	543.80	2420.65	6310.15	9603.20	12623.15	14945.71	15486.27	16091.31	14685.20
	P90 TTFT	1917.59	2022.85	5609.50	10263.90	16544.10	21013.92	23588.66	23970.65	25241.25	23839.40
	P99 TTFT	3415.36	2581.40	6791.50	11985.30	18837.20	24733.92	27613.68	28114.39	30504.64	28037.60
	Mean TPOT	25.56	33.15	42.98	55.77	58.45	56.50	58.89	59.18	53.05	57.53
	P50 TPOT	12.64	31.20	45.56	59.14	61.39	57.77	61.54	60.27	55.82	57.71
	P90 TPOT	68.15	56.95	59.28	65.88	71.93	73.01	79.41	78.14	78.47	85.31
	P99 TPOT	102.85	58.81	65.20	74.70	80.12	78.16	83.98	84.66	90.40	95.39
	Mean ITL	25.56	33.15	42.98	55.77	58.45	56.50	58.89	59.18	53.05	57.53
	P50 ITL	12.88	25.96	34.50	49.60	48.07	46.82	47.76	47.40	48.55	48.78
	P90 ITL	31.04	67.90	80.45	86.84	88.59	86.91	86.53	86.85	86.66	86.85
	P99 ITL	219.25	186.20	180.49	193.44	246.33	233.56	244.11	244.82	185.79	236.65
vLLM	Mean TTFT	653.48	2136.05	8420.13	12504.72	14806.89	16317.28	17579.54	18202.79	18875.81	19367.38
	P50 TTFT	269.74	1917.22	8806.58	14584.45	16511.97	17840.38	19142.88	20019.43	20735.47	21290.67
	P90 TTFT	1562.48	4292.70	15889.21	22470.58	26368.27	28877.56	31030.09	32175.44	33307.86	34195.29
	P99 TTFT	2376.60	5498.47	18567.68	27089.77	32034.88	35164.47	37721.28	38898.68	40562.88	41576.99
	Mean TPOT	23.16	65.10	66.07	72.00	72.75	72.32	72.56	72.11	71.97	71.87
	P50 TPOT	16.01	71.96	69.80	74.67	75.69	74.80	75.64	73.04	75.64	74.84
	P90 TPOT	47.99	92.02	89.85	90.57	90.67	88.39	88.40	87.93	88.07	87.33
	P99 TPOT	59.82	94.61	100.47	103.98	102.41	99.22	96.05	93.11	90.79	90.16
	Mean ITL	23.16	65.10	66.07	72.00	72.75	72.31	72.56	71.38	71.97	71.85
	P50 ITL	13.63	47.47	50.83	49.66	49.62	49.48	49.51	49.25	49.41	50.35
	P90 ITL	29.23	171.44	166.62	178.90	179.83	179.37	179.64	178.01	178.90	177.75
	P99 ITL	177.31	231.06	231.77	234.23	233.39	234.16	232.54	233.42	233.29	233.38

Function	Pivotal Role in CacheFlow
Scheduler-Side Methods	
`get_num_matched`	Hashes prompts to detect KV cache hits within the DPU memory mapping.
`update_state`	Records vLLM's physical block allocations to track GPU memory addresses.
`build_meta`	Orchestrates data movement by creating `Load` and `Save` specifications.
Worker-Side Methods	
`start_load_kv`	Initiates asynchronous DOCA DMA transfers from DPU to GPU staging buffers.
`wait_layer_load`	Synchronizes the DMA engine and injects KV tensors into the model's physical layers.
`save_kv_layer`	Executes zero-copy offloading by writing layer data directly to batched DMA buffers.
`get_finished`	Cleans up request trackers and updates the prefix LRU after transfers complete.

Table 7: CacheFlow Connector function descriptors.