CS599 Programming Massively Parallel Multiprocessors and Heterogeneous Systems

SSD-Backed PagedAttention Extending KV Cache with GPUDirect Storage

Final Report

Team Members: Sakshi Sharma(phsakshi@bu.edu), Yuhang Song(yuhangs@bu.edu)

Github Repository: SSD offloading

1. Problem Statement and Motivation

1.1 The Challenge: GPU Memory Constraints in LLM Serving

Large Language Model (LLM) serving systems face a critical bottleneck: GPU memory scarcity. During the decoding phase of text generation, the Key-Value (KV) cache dominates memory consumption. For modern models like LLaMA-70B with long context windows (128K+ tokens), a single request can require gigabytes of KV cache storage. This creates severe limitations on:

- **Batch size**: Fewer concurrent requests can be served
- **Context length**: Longer conversations exhaust memory quickly
- **Reuse opportunities**: Valuable cached computations are discarded prematurely

1.2 Current State: PagedAttention and CPU Offloading

vLLM's PagedAttention architecture was a breakthrough in addressing memory fragmentation. It stores KV tensors in non-contiguous, fixed-size blocks ("pages"), enabling:

- Fine-grained memory allocation
- Block-level reuse across requests (prefix caching)
- Reduced internal fragmentation

However, the current system has a significant limitation: when GPU memory fills up, blocks are offloaded to CPU memory only. This presents several challenges:

Current Flow:

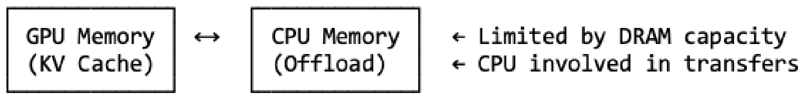

Figure 1: Current CPU Offloading flow

Limitations of CPU-only offloading:

1. **Limited capacity**: CPU RAM is finite and expensive to scale

2. **CPU overhead**: Data transfers involve CPU memory copies and bandwidth

3. **Cost**: Adding more RAM requires expensive hardware upgrades

4. **Scalability**: Each machine has fixed RAM capacity

1.3 Our Proposal: SSD Storage Tier with GPU Direct Storage

We propose adding an SSD storage tier for KV cache blocks, leveraging NVIDIA GPU Direct Storage (GDS) for efficient data movement:

Our Solution:

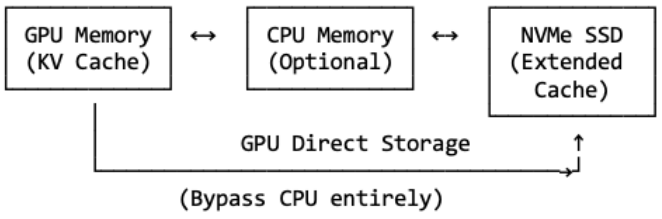

Figure 2: Proposed SSD Offloading architecture

1.4 Why SSD? Economic and Technical Rationale

Resource	Cost (per TB)	Capacity Scaling	Hardware Changes
GPU	~$2000+	Very Limited	New GPU required
CPU RAM	~$100-1000	Limited (slots)	Motherboard constraints
NVMe SSD	~$50-100	Virtually unlimited	Hot-swappable

Key benefits of SSD offloading:

1. **Cost efficiency**: SSDs provide 10-100x more storage per dollar than GPU/CPU memory
2. **Capacity**: Expand to terabytes without hardware reconfiguration
3. **No GPU changes**: Maximize existing GPU investment
4. **GDS advantage**: Direct GPU-to-SSD transfers bypass CPU, reducing latency

1.5 Why PagedAttention is Ideal for SSD Integration

PagedAttention’s block-based architecture naturally aligns with SSD I/O characteristics:

- **Block granularity**: KV blocks map directly to SSD block I/O operations
- **Sequential access patterns**: Prefill and decoding access blocks sequentially
- **Reuse tracking**: Block hashes enable efficient cache lookups

2. Overview of Our Solution

2.1 Architecture Overview

Our implementation introduces a complete SSD offloading subsystem within vLLM's v1 architecture:

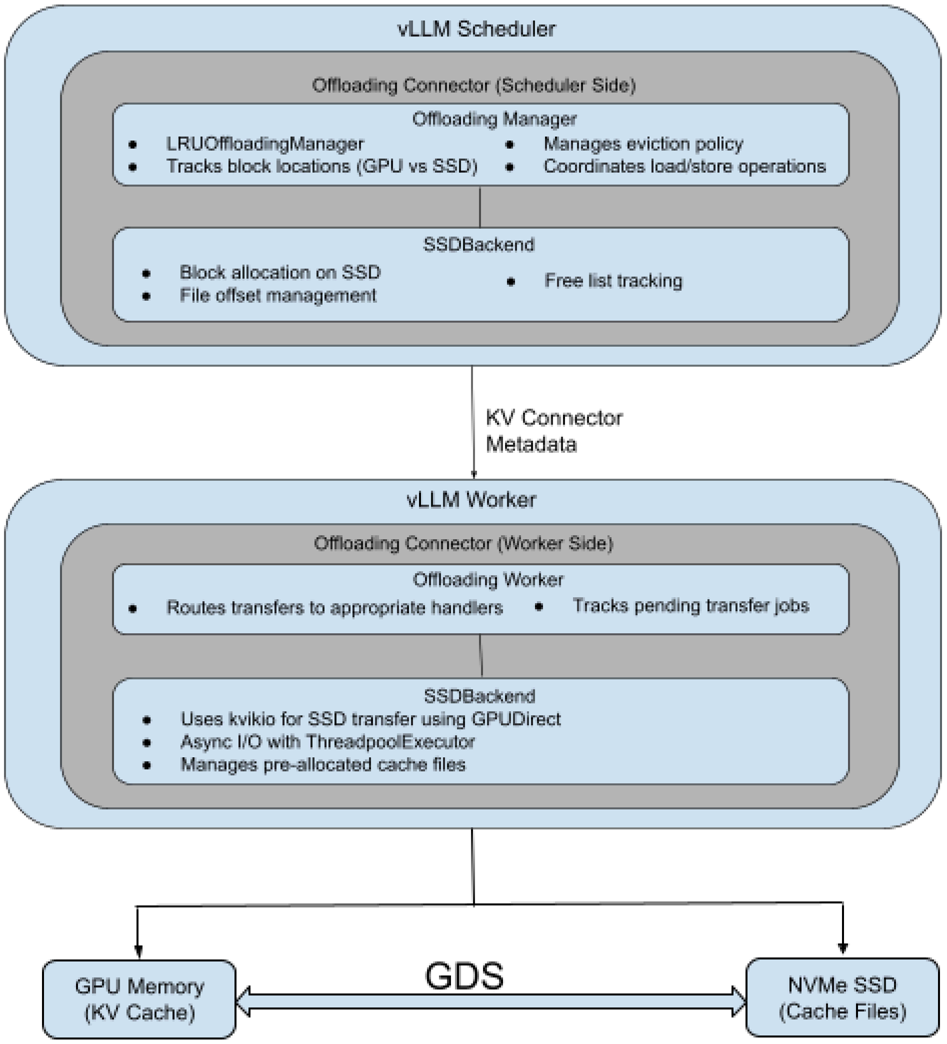
vLLM Scheduler
Offloading Connector (Scheduler Side)
Offloading Manager
LRUOffloadingManager
Tracks block locations (GPU vs SSD)
Manages eviction policy
Coordinates load/store operations
SSDBackend
Block allocation on SSD
File offset management
Free list tracking
KV Connector Metadata
vLLM Worker
Offloading Connector (Worker Side)
Offloading Worker
Routes transfers to appropriate handlers
Tracks pending transfer jobs
SSDBackend
Uses kvikio for SSD transfer using GPUDirect
Async I/O with ThreadpoolExecutor
Manages pre-allocated cache files
GPU Memory (KV Cache)
GDS
NVMe SSD (Cache Files)

Figure 3: Overall architecture of the SSD Offloading system

2.2 Data Flow

Store (GPU → SSD):

```
1. Request generates new KV cache blocks
2. Scheduler identifies blocks to offload (excess blocks)
3. OffloadingManager.prepare_store() allocates SSD blocks, evicts if needed
4. KVConnectorMetadata carries store specs to worker
5. GpuSsdOffloadingHandler.transfer_async() submits write job
6. Worker thread: kvikio.CuFile.pwrite() transfers GPU→SSD
7. Worker reports completion to scheduler
8. OffloadingManager.complete_store() marks blocks as ready
```

Load (SSD → GPU):

```
1. New request needs previously computed KV cache
2. Scheduler checks OffloadingManager.lookup() for hits
3. OffloadingManager.prepare_load() protects blocks from eviction
4. KVConnectorMetadata carries load specs to worker
5. GpuSsdOffloadingHandler.transfer_async() submits read job
6. Worker thread: kvikio.CuFile.pread() transfers SSD→GPU
7. Request scheduled after transfer completion
8. OffloadingManager.complete_load() releases eviction protection
```

2.3 Challenges Encountered

1. *PyTorch Inference Mode Restrictions*

Problem: PyTorch's inference mode prevents in-place tensor modifications

Solution: Use direct CUDA memcpy via ctypes to bypass restrictions

```python
def _cuda_memcpy(dst_ptr, src_ptr, size):
      """Direct CUDA memcpy to bypass inference mode"""
      cudart.cudaMemcpy(dst_ptr, src_ptr, size, cudaMemcpyDeviceToDevice)
```

2. *Variable KV Cache Layouts*

Problem: Different attention backends have different tensor layouts

Solution: Detect layout by probing backend's get_kv_cache_shape()

```python
test_shape = attn_backend.get_kv_cache_shape(num_blocks=1234, ...)
if test_shape[0] == 2:  # [2, num_blocks, ...]
      kv_dim_before_num_blocks = True
else:  # [num_blocks, ...]
      kv_dim_before_num_blocks = False
```

3. *Block Size Alignment*

   Problem: SSD I/O is more efficient with larger blocks

   Solution: Support configurable SSD block size as multiple of GPU block size

```
ssd_block_size = gpu_block_size * block_size_factor
# E.g., 4 GPU blocks (16 tokens each) = 1 SSD block (64 tokens)
```

4. *Reference Counting During Transfers*

   Problem: Blocks being transferred cannot be evicted

   Solution: ref_cnt tracking with -1 for "not ready" state

```
class BlockStatus:
      ref_cnt: int  # -1 = not ready, 0 = ready, >0 = in use
      @property
      def is_ready(self) -> bool:
      return self.ref_cnt >= 0
```

2.4 Alternatives Considered

Alternative 1: Direct GPU-to-SSD without kvikio

Considered: Implementing GDS integration from scratch

Discarded: kvikio is tested, maintained by NVIDIA/RAPIDS, handles edge cases

Alternative 2: Synchronous I/O

Considered: Blocking transfers in the main worker loop

Discarded: Would stall GPU and reduce throughput significantly

Alternative 3: Memory-mapped files

Considered: mmap() with GPU access

Discarded: Doesn't leverage GDS, still requires CPU page cache

3. Evaluation

3.1 Evaluation Setup

3.1.1 Hardware

We ran our experiments on an **NVIDIA DGX Spark** workstation equipped with a **20-core ARM CPU** and **Blackwell-generation GPU**. The system includes **128 GB of LPDDR5X coherent unified memory**, which allows both CPU and GPU to share the same memory space.

3.1.2 Model

After comparing several models, we selected the following two for our final evaluation:

1. **Tiny Model**: Llama-3.2-1B
   https://huggingface.co/meta-llama/Llama-3.2-1B
   This model is **2.47 GB**, making it lightweight and easy to handle. Benchmarking with a small model significantly reduces loading time and allows rapid iteration. This enabled us to repeatedly rerun experiments under different environment configurations and memory-limit settings to observe system behavior and stress-test our implementation. To simulate limited KV-cache availability, we **constrained memory using the --gpu-memory-utilization** parameter, modeling conditions where the **model weights occupy most of GPU memory**.
2. **Large Model**: DeepSeek-R1-Distill-Qwen-32B
   https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-32B
   This model is **65.5 GB**, and it was the largest model that could run reliably on the DGX Spark without failure. We used it to recreate a realistic scenario where model weights fully consume GPU memory, allowing us to verify whether the trends observed in the small-model setting still hold at scale.

We also evaluated Qwen3-Next-80B-A3B-Instruct-FP8 (*https://huggingface.co/Qwen/Qwen3-Next-80B-A3B-Instruct-FP8*), which is slightly larger at 82.1 GB. However, this model failed during inference because additional overhead beyond raw model size exceeded available GPU memory. As a result, we excluded it from full experimental evaluation.

3.1.3 Workload

We use the ShareGPT dataset to simulate real user prompts. Specifically, we use the sg_90k_part1.json subset from RyokoAI/ShareGPT52K on Hugging Face (*https://huggingface.co/datasets/RyokoAI/ShareGPT52K*), which contains 90,000 real-world user prompts for us to run the inference.

3.2 Overheading Test

The first goal is to verify that when the server has sufficient resources, *GPU-only*, *CPU-cache-enabled*, and *SSD-cache-enabled* configurations achieve similar performance. Although some overhead is expected for CPU/SSD offloading, the performance gap should remain small.

We benchmark using the following command:

```
VLLM_LOG_STATS_INTERVAL=5 vllm bench kv-offload \
  --max-model-len 4096 \
  --model meta-llama/Llama-3.2-1B  \
  --dataset-path ./sg_90k_part1.json \
  --dataset-name sharegpt  \
  --num-prompts 4096 \
  --gpu-memory-utilization 0.5 \
  --cpu-num-blocks 65536  \
  --ssd-num-blocks 131072 \
  --ssd-cache-dir ./vllm_kv_cache
```

The model size is 2.6 GB, and we allocate 64 GB of total memory for vLLM inference. With GPU memory sufficiently large, we expect all three configurations (GPU-only, CPU-cache, SSD-cache) to show comparable performance.

The first figure shows **Remaining Requests vs. Time**, sampled every 2 seconds. All three configurations exhibit nearly identical progress curves.

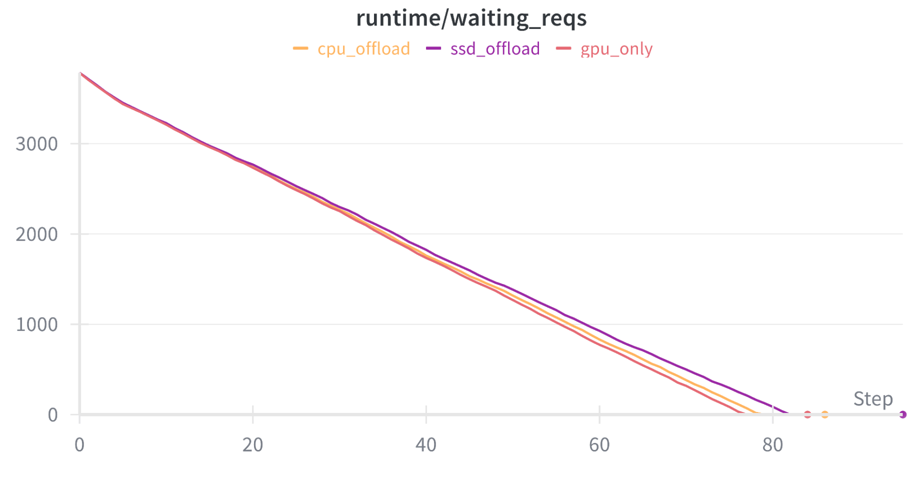

Figure 4: Remaining Requests Vs Time

The next figure displays **average generated tokens per second**, again showing that the three implementations perform similarly.

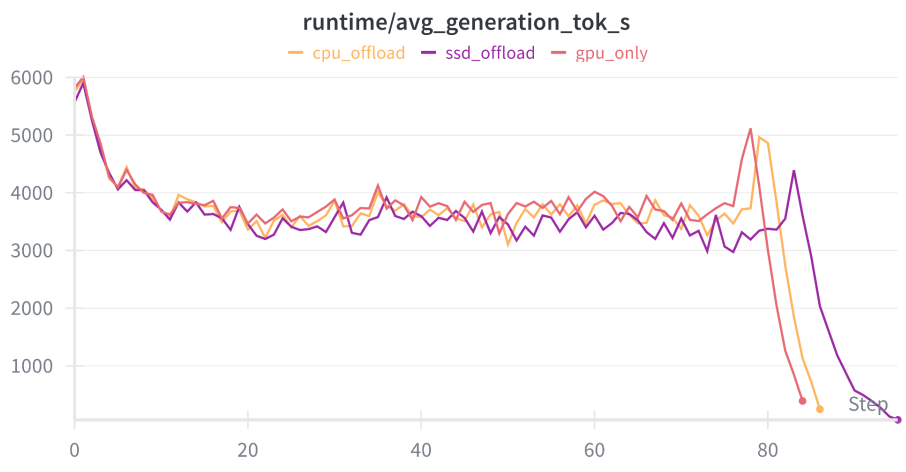

Figure 5: Average generated tokens/second

Finally, the **KV-cache hit-rate** plot confirms our expectation: with ample GPU cache capacity, all three configurations reach similar hit rates.

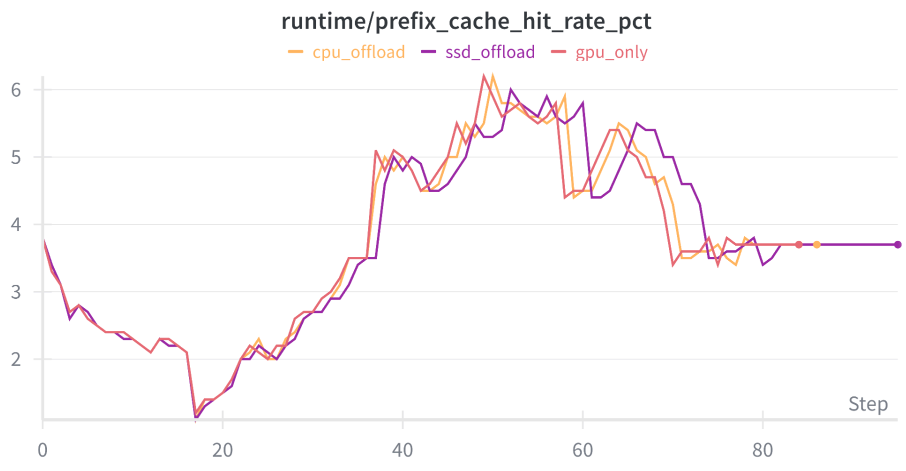

Figure 6: KV cache hit rate

3.3 Testing Under Extremely Low GPU Memory

Next, we evaluate performance when GPU memory is extremely limited. We set `--gpu-memory-utilization=0.04`, leaving **less than 1 GB** for GPU KV-cache — just enough to load the model. This simulates running a model on a personal computer where GPU memory is fully consumed by the model itself.

We also *reduce CPU cache capacity to the minimum*, mimicking unified-memory systems (e.g., Apple Silicon, DGX Spark with UMA), where additional CPU RAM is unavailable for KV caching once GPU memory is exhausted.

Surprisingly, performance remains strong. The **Remaining Requests vs. Time** plot shows that **SSD-offload performs nearly as well as GPU-only**, while CPU-offload is the slowest due to intentionally restricted CPU cache capacity, which provides no benefit and adds overhead.

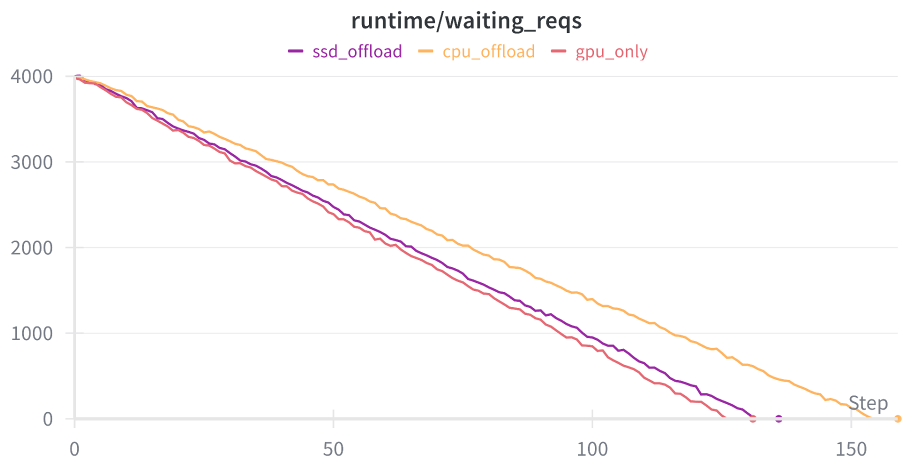

Figure 7: Remaining requests Vs Time

As expected, the **GPU cache becomes fully occupied at the start** of the run.

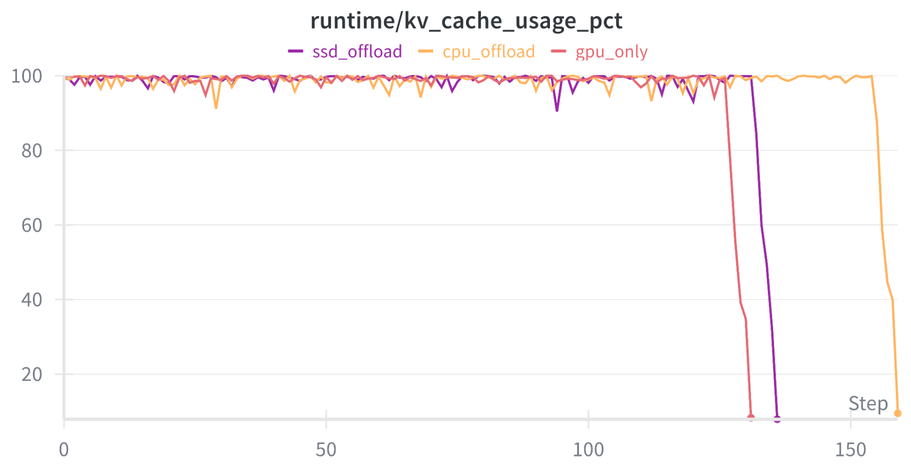

Figure 8: KV Cache usage

For SSD-offload, we observe the **KV hit rate gradually increasing**, peaking around **~6%**. Because CPU cache capacity was minimized, CPU-offload produces nearly **0% hit rate**, as expected.

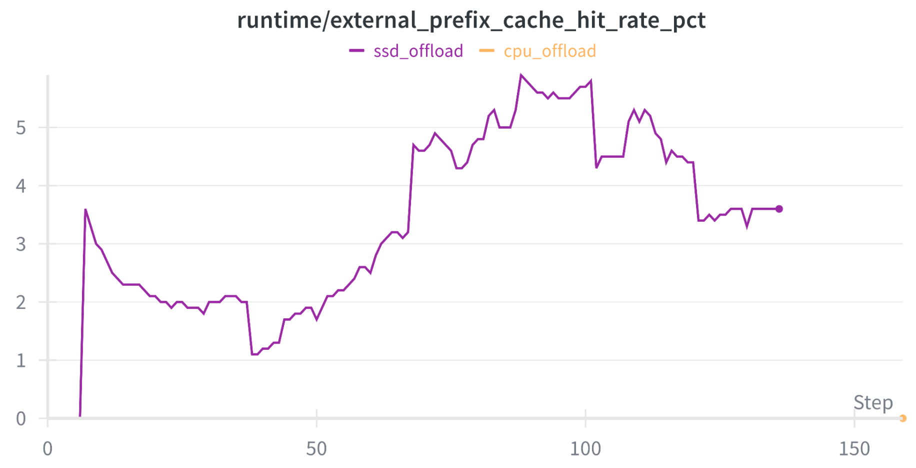

Figure 9: External prefix cache hit rate

Disk I/O statistics further confirm that **SSD-based KV cache is actively writing and functioning correctly**.

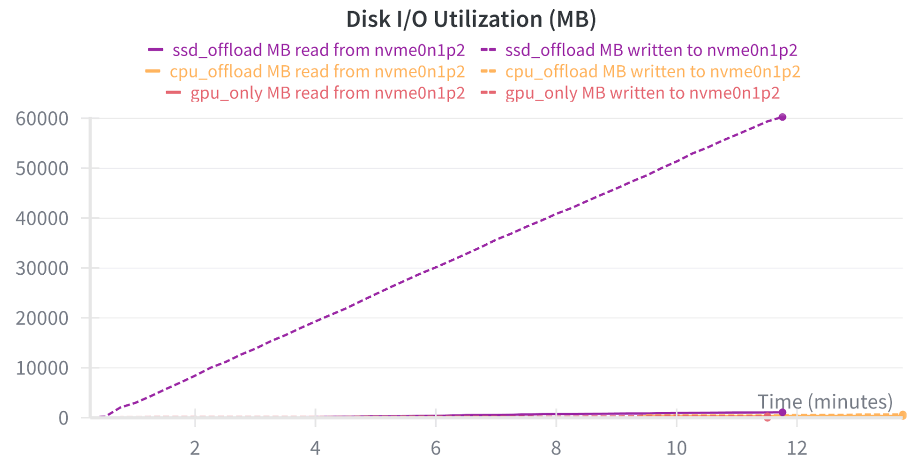

Figure 10: Disk I/O Utilization

3.4 Increasing CPU KV-Cache Capacity

In the previous setup, we limited CPU KV-cache to simulate unified-memory systems. However, on traditional x86 systems, CPU and GPU have separate memory pools. We therefore repeat the test but increase CPU KV-cache to **5 GB**, matching the GPU cache limit.

Under this condition, GPU-only remains the fastest, CPU-cache is still the slowest, and SSD-cache again sits between GPU and CPU in performance.

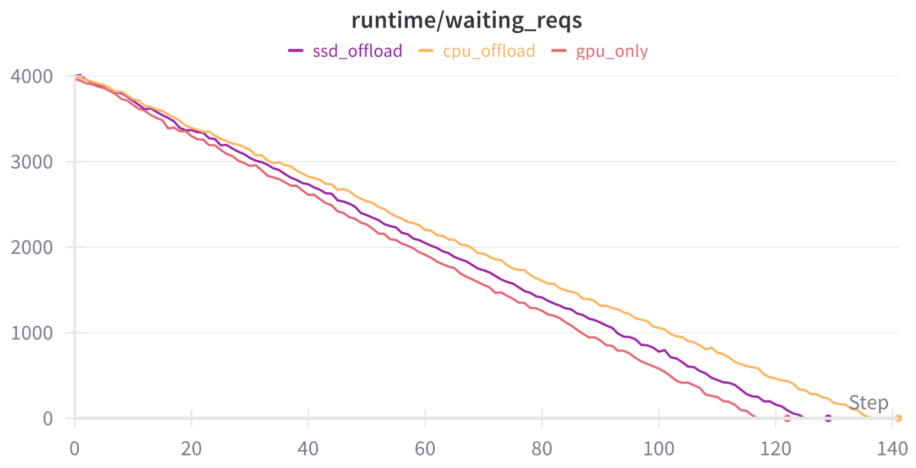

Figure 11: Waiting requests

The cache hit-rate explains this behavior: initially CPU and SSD caching behave similarly, but once CPU reaches its 5 GB limit, SSD continues accumulating cache, leading to higher long-term hit rate and better throughput.

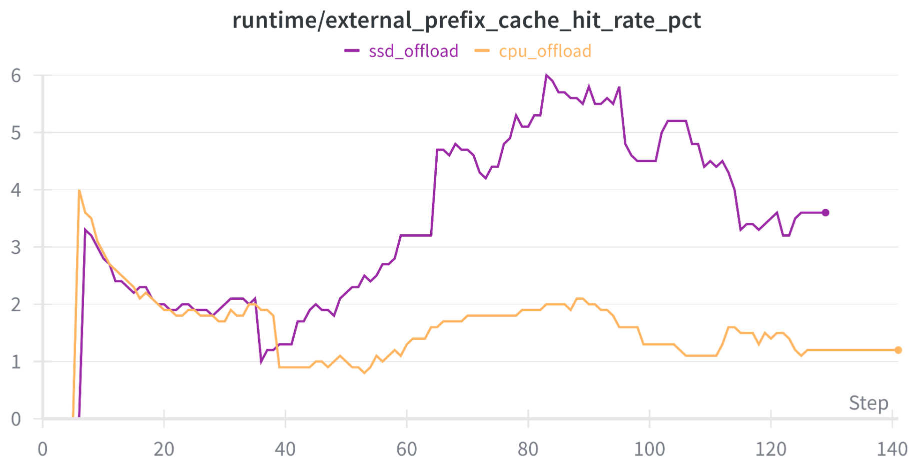

Figure 12: External prefix cache hit rate

3.5 A More Realistic Server Scenario

Previously we restricted CPU KV-cache to 5 GB to mirror consumer-grade hardware. However, in server environments it is more common to have limited GPU VRAM but abundant CPU RAM.

Therefore, we relax CPU cache limits and match CPU cache size to the SSD cache size. In this configuration, SSD-cache remains slightly faster than CPU-cache

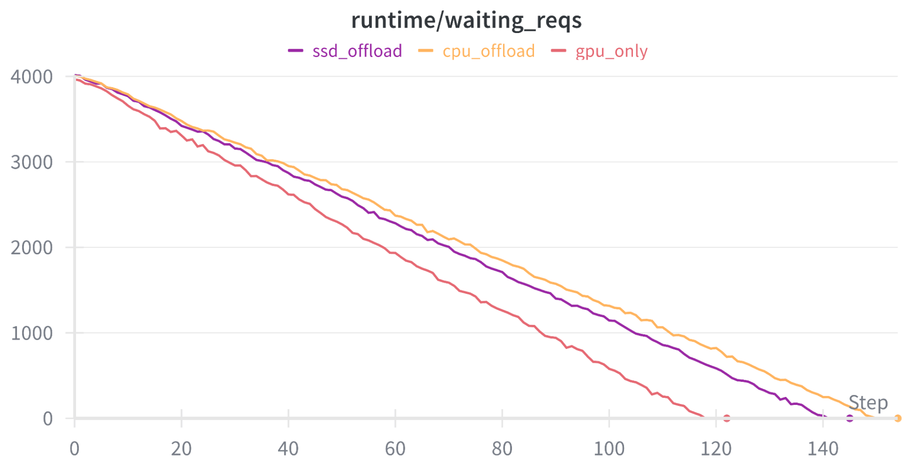

Figure 13: Waiting requests

And their external cache hit-rates are nearly identical

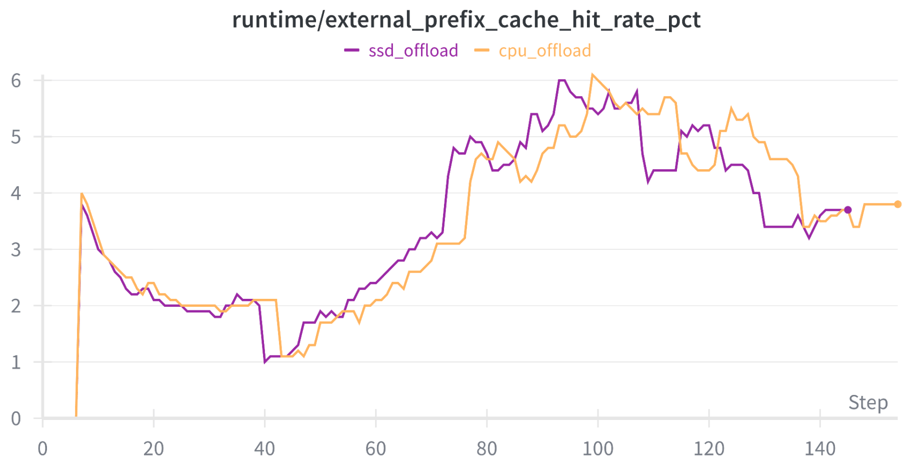

Figure 14: External prefix cache hit rate

4. Discussion

4.1 Overall Assessment and Conclusion

The key contribution of our project is implementing an **SSD-based KV cache**. Building on vLLM's existing KV-cache manager, we extend CPU-level offloading to an SSD tier. Using **NVIDIA GPUDirect**'s Python wrapper KvikIO, we enable direct DMA transfers between GPU memory and NVMe SSDs for KV-block swapping. We also implement an **LRU-based rotation mechanism** to efficiently manage SSD capacity when the cache becomes full.

We benchmark our system to evaluate its effectiveness. First, we test in a high-memory environment and show that our **SSD-based solution introduces minimal overhead**. Next, under extreme memory pressure with unified memory, we demonstrate that SSD offloading significantly outperforms CPU caching. We then compare scenarios where SSD and CPU caches have the same capacity and find that SSD remains faster. Finally, even when CPU memory is plentiful, our SSD approach performs close to the CPU baseline, with comparable cache hit rates.

4.2 Difficulties with the SparkDGX Environment

The NVIDIA DGX Spark we used is a very new hardware platform. Although unified memory between GPU and CPU is promising, we encountered several challenges when setting up the environment for vLLM. Full compilation fails because the build system does not clearly distinguish between ARM and x86 environments, causing errors when compiling oneDNN components. Since our modifications are limited to the Python layer, we used the precompiled ARM binaries instead, which allowed us to modify and test our implementation.

Additionally, the default PyTorch version on the system is incompatible, but PyTorch built for CUDA 13.0 works correctly. Some development and testing dependencies were also missing or incompatible, but we managed to install everything necessary to run our workload. We documented the detailed environment setup steps in our code repository so that others can reproduce our results.

4.3 Future Work

4.3.1 Cross request and long term kv cache on SSD

Our current SSD-based KV-cache implementation depends on vLLM’s original KV-cache manager, which does not support long-term or cross-request KV persistence. It primarily shares KV blocks only across concurrently running requests. Supporting long-term KV on SSD would enable more realistic workloads and improve reuse across multi-turn conversations.

4.3.2 Evaluation on the multi term dataset

Our current evaluation uses only the first prompt of each request, which puts our approach at a disadvantage because it reduces opportunities for KV-cache reuse. A more meaningful evaluation would involve multi-turn conversational datasets, especially code-generation dialogues. These conversations are extremely long, and each new user query typically depends on all prior context. Without a persistent KV cache, the model must recompute large portions of the prefix. Running on such datasets would yield much higher cache hit rates, and in some cases, our SSD-based solution may even outperform GPU-only caching.

5. Related Work

1. Kwon et al., "**Efficient Memory Management for Large Language Model Serving with PagedAttention.**"
Introduces block-level KV-cache management. But they do not provide a SSD swap policy.

2. Ren et al., "**An I/O Characterizing Study of Offloading LLM Models and KV Caches to NVMe SSD.**"
Characterize I/O when offloading model weights and KV-cache. But they are not based on paged attention.

3. **NVIDIA GPUDirect Storage**
https://developer.nvidia.com/gpudirect
We will implement the cache swapping with Nvidia GPU direct storage feature to transfer data between GPU and SSD.

4. **Mastering LLM Techniques: Inference Optimization**
https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/
Introduced the KV-cache mechanism and the importance of having this kind of cache in inference.

6. Contributions

We analyzed the project proposal and discussed approaches to tackle the problem collaboratively. Sakshi Sharma primarily contributed to the SSD-based KV-cache swapping implementation, while Yuhang focused on benchmarking and evaluation using real-world datasets. We then analyzed the results together and co-authored the final report based on our findings.

Code: latest changes with SSD changes and benchmarking - SSDOffloading