LLM Serving With Efficient KV-Cache Management Using Triggered Operations

Aditya Dhakal
Hewlett Packard Labs
aditya.dhakal@hpe.com

Pedro Bruel
Hewlett Packard Labs
bruel@hpe.com

Gourav Rattihalli
Hewlett Packard Labs
gourav.rattihalli@hpe.com

Sai Rahul Chalamalasetti
Hewlett Packard Labs
sai-rahul.chalamalasetti@hpe.com

Dejan Milojicic
Hewlett Packard Labs
dejan.milojicic@hpe.com

***Abstract*—A large language model (LLM)'s Key-Value (KV) cache requires enormous GPU memory during inference. For faster query processing and conversational memory of chat applications, this cache is stored to answer subsequent user queries. However, if the cache is buffered on the GPU, its large memory requirement prevents GPU multiplexing and requires cache buffering in remote storage. In current systems, transferring and retrieving cache requires the CPU to coordinate with the GPU and push and fetch the data through the network, thereby increasing the overall latency.**

This paper proposes lower overhead KV cache storage and retrieval with SmartNICs capable of triggered operations, such as HPE Slingshot Cassini. Triggered operations enqueue pre-defined data transfer instructions on the Network Interface Cards (NICs). A GPU thread can trigger these instructions once the LLM computes the KV cache for a token. Cassini NIC then transfers the cache, bypassing the CPU and network stack and improving data-transfer latency. Our experiments show that data transfer with triggered operations provides 19× speedup in transfers ranging from 32KB to 5 Terabytes.

I. Introduction

Large Language Models (LLMs) are state-of-the-art deep learning applications trained to generate human language output. Newer LLMs such as GPT-4 and LLaMa-2 produce helpful answers to queries in human language [1]. Consequently, LLMs have been adopted widely for chatbots, knowledge references, and text-generation tools.

LLMs require a large amount of GPU, compute, and memory. Llama-2-70B requires 120GB of GPU memory [2]. As an LLM processes user input (tokens), it caches the results of each token to process the next token. This cache is known as key-value cache (KV-Cache) [3]. KV-cache can grow to 3× the model size as the LLM infers more tokens [4]. Llama-2-70B profiling from NVIDIA [5] shows a requirement of at least 8 GPUs to process a batch of 8 requests. Due to the high reservation cost of cloud GPUs (Table I), dedicating 8 GPUs for a single user would not be cost-effective when planning to serve millions of users.

Multiplexing the GPU by inferring multiple users' requests will greatly increase the GPU utilization and cost-effectiveness of the multi-GPU setups. The current method to serve LLM inference to multiple users is to infer a user's query (prompt) and store the combination of the user's prompt and the LLM's output [7]. The LLM's model's state can then be deleted from the GPUs' memory, freeing up the GPU memory for running another user. When the first user submits a new query later, this new query is appended to the old prompts, and its output is inferred in its entirety. The previous stored queries and output provide context to the new query. This approach has a problem; with every new prompt from the user, the total prompt size increases, thus requiring increased GPU compute and memory [5].

TABLE I: Cost of reserving 8 Cloud GPUs for LLM inference [6]

GPU	Cost GPU/hr	8X GPUs/hr
NVIDIA A100	$2.21	$17.86
NVIDIA H100	$4.76	$38.80

Another solution for multiplexing the accelerators would be to buffer the relevant data necessary for the context of a user's prompt outside the accelerator memory. The free accelerators can then be used for another task. When the first user sends subsequent queries, the relevant data for the user is fetched from storage to the accelerators, and then the inference on the new query is performed. In this approach, only the new prompt needs to be processed, thus lowering the compute required.

The primary data that needs to be buffered is the user's Key-Value cache (KV-cache). The Key and Value vectors are used during the decoder stage in multi-head attention for a generative transformer model (e.g., GPT). The previously generated token's Key and value vectors are used to compute the probability of the next token. Thus, caching the Key and Value vectors eliminates re-computation. This cache is known as a KV-cache. For large batch sizes, the KV-cache size can get larger than the model itself. Transferring the entire KV cache from the accelerators' memory to remote storage is challenging. Utilizing CPUs to pull the data from the accelerators and then push them to the network will increase the overhead of data transfer significantly.

To address these challenges, this paper discusses our approach, in which the transfer and buffering of the KV cache

are offloaded to the SmartNICs. This significantly lowers the CPU overhead and latency of the data transfer. We analyze using "triggered-operations" available in Slingshot Cassini NICs [8] to offload the data transfer to the NIC. We present our evaluation of different data size transfers from GPU to remove NVME drives and evaluate scenarios where the SmartNIC offloading is present. We show that our approach can speed up the data transfer and retrieval from remote NVME drives by more than 19X compared to approaches where the CPU copies the data from the GPU and then pushes it to the network to store it in remote storage.

II. Background and Motivation

In this section, we present the background on LLM and the motivation for our work.

A. Background: LLM Inference and KV-cache

LLMs such as GPT and LLAMA are generative transformer models, where the model's core is the "self-attention mechanism." Self-attention lets the model compute the probability of different tokens (words) in a sequence, thus allowing the model to pick the most probable word. In self-attention, every input token (e.g., a word in a sentence) is transformed into three representations: a query (Q), a key (K), and a value (V). The self-attention mechanism computes the attention scores by taking the dot product of the query vector (Q) of one token with the key vectors (K) of all tokens in the sequence. These scores determine how much focus or "attention" each token should pay to every other token in the sequence. The attention scores are then used to weight the value vectors, essentially determining the output representation of each token based on the information from all other tokens.

The self-attention mechanism utilizes the Key and Value vectors of all the tokens. Key and value vectors of previously generated tokens are cached to reduce the re-computation. This cache is also known as KV-cache. KV-cache allows the computation of LLM to scale as a linear rather than a quadratic function of their token count.

KV-cache can grow very large based on batch size and number of tokens the model supports. We have calculated the size of KV-cache as a function of the model size and presented results in Fig. 1. We have fixed the maximum token size for each model to 512 tokens and the batch size of inference to 64. A larger token size will lead to an even larger KV cache. For some models, the KV cache is larger than the model size, and for others, it is still very large and significant. Current GPUs are very limited in DRAM, with the largest GPUs offering up to 200 GB of DRAM. These large KV-caches cannot reside on the accelerator waiting for the user to send subsequent responses, as they will occupy valuable accelerator space that can be used for other user inference.

B. Motivation: Importance of KV-Cache for Context

We present a motivating scenario that shows that effective management of KV-cache is necessary for conserving the context of LLM chat application.

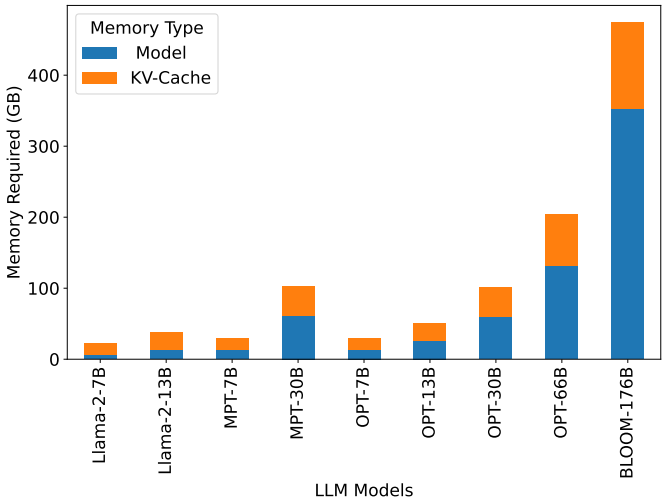

Fig. 1: Accelerator memory occupied by LLM model (16 bit) and KV-cache for different models

User: Please provide me with a recipe for making cheesecake.
LLM: For the cheesecake... 350°F...
User: Only provide temperature in °C from now on.
LLM: OK.
...
User: What temperature should I set the oven on?

The box above shows a snippet of the user starting a conversation with an LLM and asking about a recipe. The LLM faithfully replies to the user with the recipe but with the temperature in Fahrenheit. The user then prompts the LLM to report the temperature in Celsius. The LLM model then generates further responses only in Celsius.

However, the LLMs rely on KV-cache to remember this context, and evicting KV-cache will lead to the model forgetting the context. In this example, if the KV-cache noting the subsequent result published in Celsius is evicted, the damage is quite small. The user will not get a response in Celsius. However, if the KV-cache holds important security information, such as instructions not to reveal personal information, then evicting the KV-cache can be quite harmful. Therefore, it is imperative that the KV-cache is carefully stored and retrieved accurately for the proper functioning of the LLM model.

C. Background: Triggered Operations

Triggered operations [9] allow applications to enqueue data transfer requests in the NICs and defer them for future trigger events. The origin of triggered operations can be traced to Quadrics network [10]. HPE's Cassini NICs also provide triggered operations features. Slingshot NICs initially used the Portals communication library [11]. Currently triggered operations are supported with libfabric [12] communication library. Libfabric library allows users to set a specific event counter and a threshold. The application first has to assign a memory buffer that can be transferred. This transfer is put in a deferred queue. When the event counter is updated and

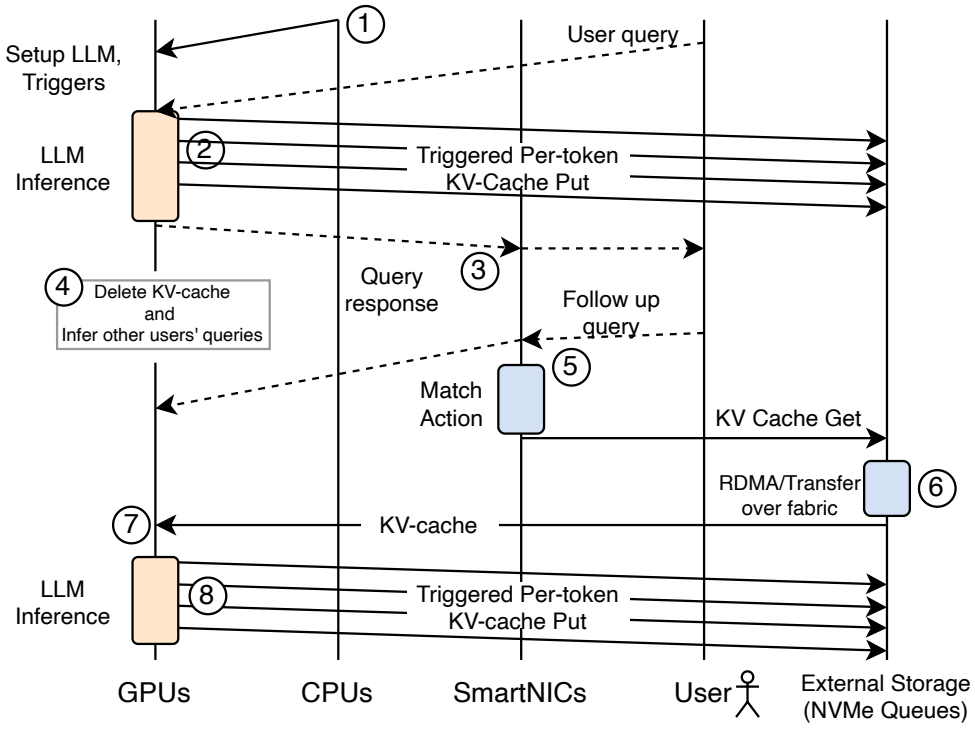

Fig. 2: LLM-based chat application with KV-cache transfer

the threshold is passed, the data from the memory buffer is transferred. This event counter can be updated easily by application. Triggered operations do not involve the CPU, other than the initial trigger setup.

In our work, we explore the use of triggered operations to transfer KV-cache out from GPU memory to the network location. The CPU application registers the GPU buffer where the KV cache is stored. When a token is inferred, a GPU thread can trigger the NIC [8]. The NIC will then transfer the particular buffer to the storage endpoint. This transfer eliminates the need for the CPU to run API functions such as *cudaMemcpy* to move the data from GPU to host memory and call fabric functions, such as socket/verbs. Triggered operations can move the data in a peer-to-peer format from accelerator to network, thus eliminating the need to push the data to the network using the network stack. This two-fold enhancement the triggered operations can greatly reduce the data movement latency as we show in our experiments and simulation results in Section V.

III. System Architecture

In this section, we will first illustrate the user's inference and KV cache transfer. We then explain our proposed system and sub-systems that are necessary for efficient KV-cache transfer and retrieval from GPUs to storage nodes.

A. LLM chat application workflow

In Fig. 2 we present a chat application with user query, LLM's response, and KV cache storage and retrieval. We have illustrated the important bits of execution with circled numbers (1)-(8).

At the beginning, in the step (1), the CPU setups the LLM application. The CPU also sets all the necessary buffers, including the buffer where the KV cache will be stored for the LLM. In the same step, the CPU program uses libfabric to register the KV-cache data buffer for the trigger operations. Here the trigger event can be set as a GPU thread updating a flag in a memory location.

The step (2) shows LLM inference being conducted. First, the user query arrives at the accelerator. An encoder encodes this query. The LLM's transformers then start generating an answer for the user. While the answer is being generated, the KV vectors for each token are also generated and stored in the KV cache. Using triggered operations our setup then transfers the newly created KV-cache for every few tokens with the aid of the smartNICs.

The step (3), the LLM response is returned to the user. In this scenario, we assume the user takes time to write a follow-up query. Meanwhile, the match action module understands the request and starts retrieving the relevant KV cache for that particular user. The step (6) shows the KV Cache Get to the storage servers and RDMA of the KV-cache data to the GPU. The Get request is initiated by the SmartNIC and enqueued in the storage device's NVME drive. When the NVME drives fetch the data, the SmartNICs in the storage nodes will RDMA the KV-cache data to the relevant GPU buffers as shown in step (7). Finally, in step (8), the follow-up query is inferred, generating a new result. Similar to previous step, the KV-cache

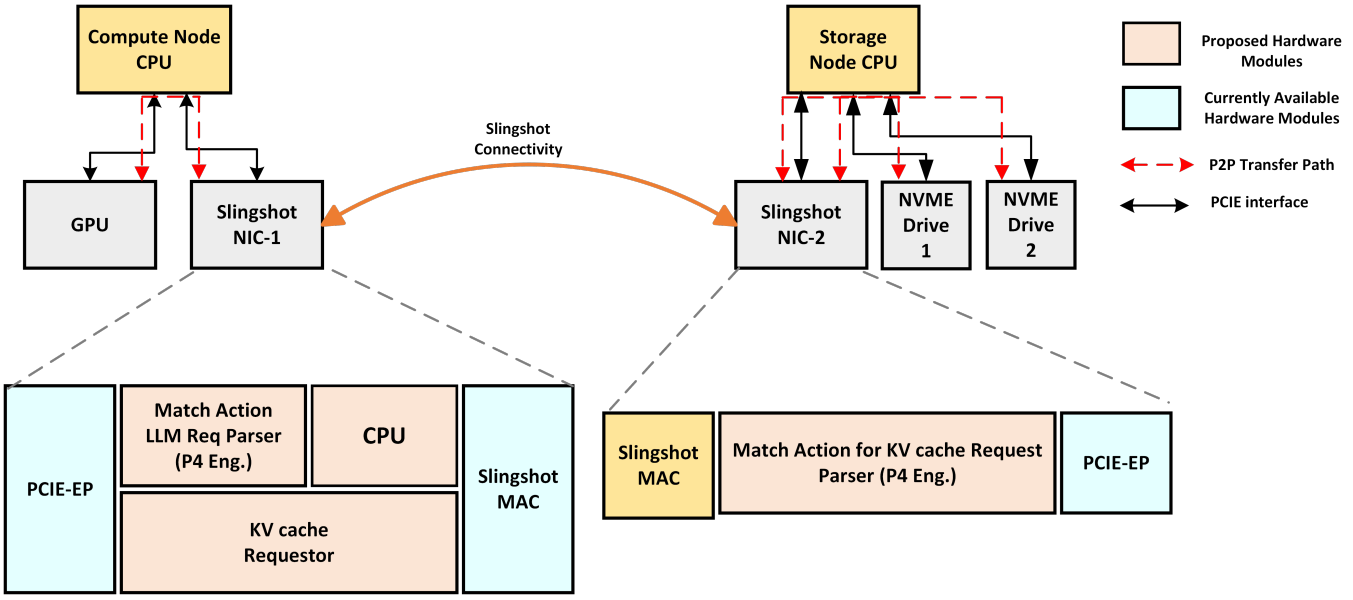

Fig. 3: System Architecture Setup with Slingshot NICs and other proposed hardware

is transferred to storage with trigger operations. This process is repeated whenever a user makes a new LLM inference request.

B. System Architecture Description

In this subsection, we describe the components in the SmartNIC and the Storage node necessary for the transfer and retrival of KV-cache from the GPUs.

The proposed system architecture is shown in Fig. 3. It depicts two system nodes (compute and storage node) with GPUs and NVME drives, respectively, connected through a network fabric such as Slingshot. Due to the storage density of the KV cache, NVME drives are suitable. For easier understanding, new hardware blocks required for compute and storage node NICs are represented in different colors.

We describe the function of the proposed hardware modules in the smartNIC. When the compute node NIC (NIC-1) receives a follow-up LLM request, a match action is performed on the incoming packet with the *Match Action module* to match the query to the user. The *Match action module* matches the incoming requests using the metadata such as user-ID, application context IDs, and network 5-tuple. The match action module uses a match action table to point to all the relevant KV-cache for the user.

The *KV-cache Requester* is responsible for the KV-cache storage and retrieval operation, including hash operation for mapping of user requests to corresponding hardware addresses where KV-cache is stored. The *KV-cache Requester* updates a hashmap every time the KV-cache is transferred from the accelerators to the storage node. This hashmap maps the KV-cache to the storage node. When the KV-cache needs to be retrieved, the cache requester will send the retrieval requests to the storage node NIC (NIC-2). After a match action of the retrieval request (sent by NIC-1) on the storage NIC (NIC-2), the incoming request is forwarded to the respective drive.

We propose using the NVME-oF environment with RDMA for NIC-2 to program the submission and completion queues of the NVME drive. Once the NVME command is processed by the NVME drive NIC-2 would initiate RDMA write to GPU memory for KV-cache *get* and RDMA read from GPU memory for KV-cache *put* operation. For keeping track of user requests and the stored cache memory management, such as Least Recently Used Eviction and Timer-based Eviction of values we propose using *low power CPU* core inside the NIC.

TABLE II: Bandwidth parameters used in simulation, where m is the message size in bytes.

Parameter	Value $\left(\frac{1}{\text{bytes/s}}\right)$	Description
β_{PCIE}	8.12×10^{-12}	PCIE bandwidth
β_{VRAM}	4.55×10^{-13}	GPU memory bandwidth
β_{CPU}	4.55×10^{-13}	CPU estimated bandwidth
β_{DRAM}	1.21×10^{-12}	Node memory bandwidth
$\beta_{\text{NIC\_serial.}}$	5.13×10^{-11}	NIC serialization bandwidth
$\beta_{\text{NIC\_mem}}$	1.43×10^{-11}	NIC memory bandwidth
$\beta_{\text{NIC\_alu}}$	5.22×10^{-12}	NIC ALU bandwidth
$\beta_{\text{NVMe\_write}}$	$\frac{1}{2.34 \times 10^{9} + 6.83m}$	NVMe write bandwidth model
$\beta_{\text{NVMe\_read}}$	$\frac{1}{5.22 \times 10^{9} + 19.4m}$	NVMe read bandwidth model

TABLE III: Latency parameters used in simulation, where m is the message size in bytes.

Parameter	Value (s)	Description
λ_{PCIE}	3.2×10^{-8}	PCIE latency
$\lambda_{\text{NVMe\_write}}$	$3.3 + 4.10 \times 10^{-7}m$	NVMe write latency model
$\lambda_{\text{NVMe\_read}}$	$2.03 + 2.63 \times 10^{-7}m$	NVMe read latency model
λ_{network}	1.21×10^{-12}	Network latency

IV. Experimental Setup

We built a performance model and simulator using latency and bandwidth for HPE Cassini NICs on a compute node with two Xeon Intel CPUs and 8 H100 Nvidia GPUs, assuming one NIC per GPU. We conducted experiments in existing real hardware to get the benchmarks and created an analytical model to produce performance metrics for end-to-end computation. We have utilized the vLLM LLM inference serving platform. We have used HPE ProLiant DL380 Gen10 Server as the inference and storage servers. We have modeled the link between the servers with a 100Gbps connection. The storage node has NVMe drives. Our modeled storage node also used Cassini NICs and we modeled typical NVMe read and write latency and bandwidth. We tested KV-cache sizes produced by popular models listed in Fig. 1.

A. Simulation setup

The communication between inference and storage servers was modeled analytically. Not all components and delays were modeled, resulting in an approximation of real behavior. Tables II and III show the estimated bandwidth and latency parameters used by each modeled component. We decided to model the bandwidth and latency for the read and write operations of NVMe drives to attempt to account for delays due to message queuing. In this model, we found that a linear model was capable of representing satisfactorily the performance of NVMe reads and writes, but future work will include more complex latency and bandwidth models for all simulated components.

V. Results

In this section, we present the results of measurement in NVMe drives as well as results obtained from the simulation of the end-to-end model of KV-cache transfer.

A. NVMe Latencies

We first measure the NVMe read and write latency. These NVMe drives would be present in the storage nodes. We need accurate measurements to build the end-to-end model. We present micro-benchmarks from NVMe drives in Fig. 4. FIO benchmark [13] is used to measure the latency of sequential read and write, and random read and write benchmarks, by performing a sweep from 512B to 128MB data sizes. We can see that NVMe drives have 10-100 ms latency while reading small-size data, but latency increases quite rapidly to 2000 ms when reading data of larger size ($\geq$64 MB). We do not see a great difference between sequential and random reads.

We have also benchmarked the writes done to the NVMe drives. Using FIO, CPU threads populate the NVMe queues to write onto the drive. We see that write latency increases exponentially as the data size becomes ($\geq$128 MB). Both write and rand-write suffer from very large latency. With this data in mind, we have opted for our technique to transfer the KV cache of few generated tokens at a time to utilize network bandwidth as well as write latency of NVMe better. We have also included this data in the simulation of end-to-end KV-Cache transfer.

B. End-to-End Simulation results

We primarily compare between *CPU-* and *Triggered-Operations-initiated* transfer of KV-cache. CPU-initiated transfer refers to one where the data is transferred from GPU to host memory using API calls such as *cudaMemcpy*. These API calls are executed in the host CPU and need to be called after the KV-cache for a few tokens is computed. We have considered transferring the KV cache in chunks of 512 MB (unless transferring a smaller size KV cache). *Triggered-operation-initiated* are the operations that execute with the help of NIC and transfer the data to storage without involving host CPUs. We should note that this simulation only produces the data transfer latencies between GPUs and storage nodes and does not simulate the rest of LLM inference.

Our results in Fig. 5 show the difference in time taken for CPU-centric and GPU-initiated KV cache transfer for a small amount of data. CPU-centric transfer moves the KV-cache from GPU to host memory using the CPU, then, the CPU pushes the data to the remote storage. GPU-initiated transfer utilizes GPU-threads-initiated trigger operation in the NIC and queues inside the NVME drive to move the data from the GPU to the remote storage. We have particularly focused on this data size as single user inference might produce a few hundred megabytes of KV-cache at a time. We can see in Fig. 5 that GPU-initiated transfer provides more than 20$\times$ speed up compared to CPU-centric data transfer.

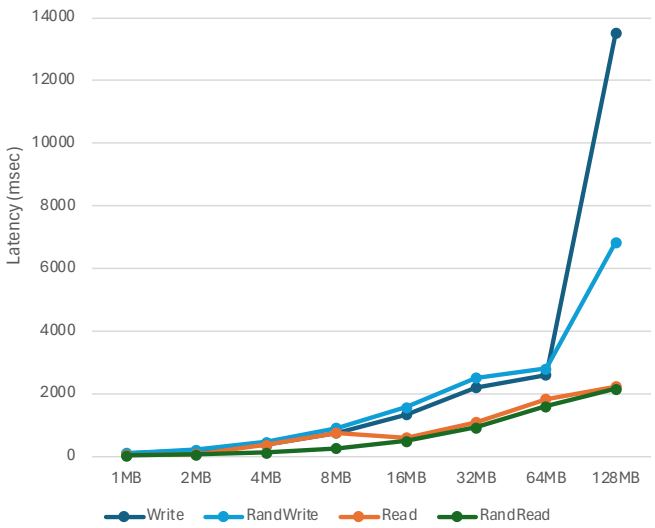

Fig. 4: NVME Read/Write latencies

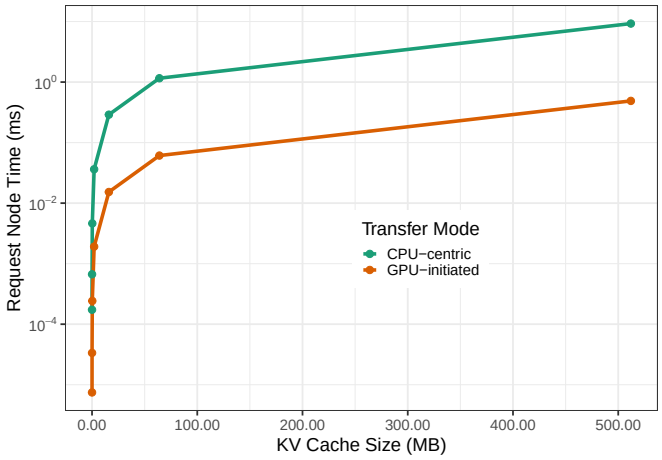

Fig. 5: CPU-Centric transfer vs. Triggered initiated transfer

We extend our results to simulating the transfer of terabytes of data at once. This scenario envisions the transfer of KV-caches out of multiple GPUs at once for a very large LLM model. We present the simulated time taken in Fig. 6. We can see the CPU-initiated still lags behind trigger-ops-initiated transfer by a huge margin (>19x).

We also present the Speedup of GPU-centric triggered operations KV-cache transfer when compared to CPU-centric approach for different KV-cache data sizes in Fig. 7. We can see that the speed-up of data transfer is larger, with 23X improvement at small data sizes. The speedup is consistently larger than 19X in larger data sizes. With these results, we can see that offloading the data transfer and avoiding the use of CPU for API and network functions calls that add large overheads greatly reduce the data transfer time from GPUs to storage node.

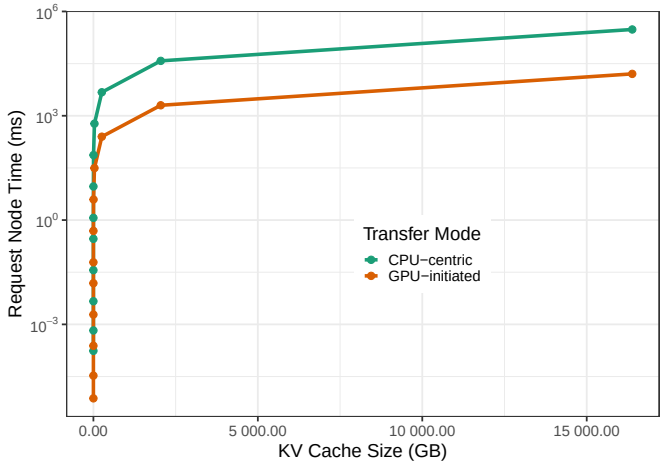

Fig. 6: CPU-Centric transfer vs Triggered transfer for very large data sizes

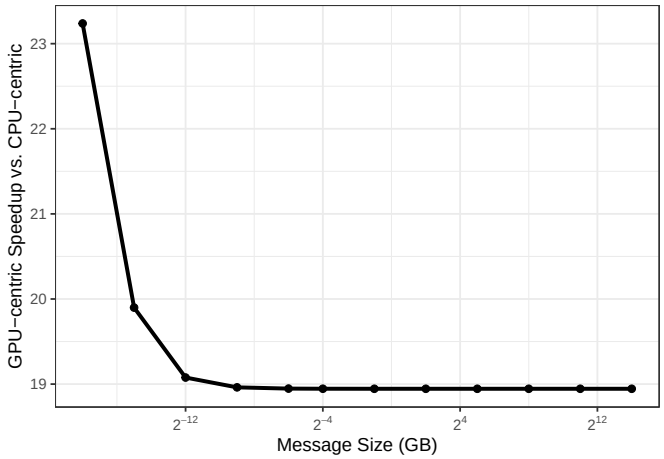

Fig. 7: Speedup obtained by Triggered transfer vs. CPU-Centric Transfer

VI. Related Work

The Key-Value (KV) cache is one of the significant constraints in expanding the capabilities of Large-Language Models. The increasing demand for inference is creating substantial challenges for cache performance. Key-Value cache provides an effective way to accelerate the generation speed for Large-Language Models inference [14].

The significance of the KV cache in LLM inference warrants the need for more optimized utilization. Kwon et al. [3] proposed a vLLM system that uses the PagedAttention algorithm that significantly reduces the wastage of KV cache memory by efficiently managing it using non-contiguous memory allocation and lookup tables. The vLLM system is built on top of this algorithm and allows for a near-zero waste in KV cache memory.

Compared to vLLM, DeepSpeed-FastGen presents an optimized system specialized for text generation. One of the key components of this is the Dynamic SplitFuse, which optimized

the composition of text to significantly reduce latency and subsequently improve the throughput. Compared to vLLM, the authors claim to have increased throughput by about 2 times. It also uses non-contiguous KV caches to allow for increased occupancy and high responsiveness [15]. The usage of such non-contiguous KV cache is not unique to this system and has also been used in DistAttention [16], HuggingFace TGI [17] and Nvidia TensorRT-LLM [18].

Another such system ORCA [19], uses iteration-level scheduling. This allows the system to schedule at the granularity of individual iteration rather than a request. Such a system allows for more efficient usage of memory, thus improving KV-cache performance. It also uses selective batching to enable better utilization of computational resources by batching only selected operations within the Transformer model.

No Token Left Behind [20] describes the importance of storing and retrieving KV-cache correctly. The paper shows the security scenarios where missing KV-cache can generate the prompts that have been explicitly banned by previous prompts. This paper presents the importance of conserving the KV-cache accurately so the generation of information follows the context set by previous prompts.

Previous works such as NetML [21, 22] and popular GPU libraries such as GPUDirect and GDRCopy [23] utilize GPU's threads and DMA engine to transfer data between GPU and other PCIe devices without utilizing host CPU. These works show drastic improvement in data transfer latency when the CPU is not involved. However, unlike our work, these tools and projects are limited to moving data within a system only.

VII. Conclusion and Future work

In this work, we have proposed a system that will transfer the KV cache generated during the generation phase of transformer-based LLMs to make way for multiplexing the GPUs. We propose using triggered operation capabilities present in Cassini NICs to transfer the KV cache from the compute node to the storage node and vice-versa. We discuss the necessary hardware and software components in a SmartNIC to enable KV cache transfer with triggered operations.

We create a mathematical model with available hardware data and measure end-to-end KV-cache transfer latencies. Our proposed GPU-triggered cache transfer provides more than 19× speedup while transferring data compared to CPU-centric data transfer. In the future, we are adapting the vLLM project [24] to work with programmable SmartNICs to cache the KV into remote storage. We use libfabric trigger operations [9] to enqueue data transfer after completing the LLM inference.

References

[1] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[2] NVIDIA, "Llm sizing guide," https://docs.nvidia.com/ai-enterprise/workflows-generative-ai/0.1.0/sizing-guide.html, 2023.

[3] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[4] Z. Liu, A. Desai, F. Liao, W. Wang, V. Xie, Z. Xu, A. Kyrillidis, and A. Shrivastava, "Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time," *arXiv preprint arXiv:2305.17118*, 2023.

[5] NVIDIA, "Llama-2 inference performance," https://docs.nvidia.com/nemo-framework/user-guide/latest/performance/llama.html, 2023.

[6] CoreWeave, "Coreweave gpu cloud pricing," https://www.coreweave.com/gpu-cloud-pricing, 2024, accessed on 2024-01-18.

[7] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Re, I. Stoica, and C. Zhang, "FlexGen: High-throughput generative inference of large language models with a single GPU," in *Proceedings of the 40th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, 23–29 Jul 2023, pp. 31 094–31 116. [Online]. Available: https://proceedings.mlr.press/v202/sheng23a.html

[8] N. Namashivayam, K. Kandalla, T. White, N. Radcliffe, L. Kaplan, and M. Pagel, "Exploring gpu stream-aware message passing using triggered operations," *arXiv preprint arXiv:2208.04817*, 2022.

[9] "Libfabric trigger operations," https://ofiwg.github.io/libfabric/v1.9.1/man/fi_trigger.3.html, 2024.

[10] F. Petrini, W.-c. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The quadrics network: High-performance clustering technology," *Ieee Micro*, vol. 22, no. 1, pp. 46–57, 2002.

[11] B. Barrett, R. B. Brightwell, R. Grant, K. Pedretti, K. Wheeler, K. D. Underwood, R. Riesen, A. B. Maccabe, T. Hudson, and S. Hemmert, "The portals 4.1 network programming interface," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2017.

[12] "Libfabric," https://ofiwg.github.io/libfabric/, 2024.

[13] FIO, "Fio benchmark," https://fio.readthedocs.io/en/latest/fio_doc.html, 2023.

[14] H. Kang, Q. Zhang, S. Kundu, G. Jeong, Z. Liu, T. Krishna, and T. Zhao, "Gear: An efficient kv cache compression recipe for near-lossless generative inference of llm," 2024.

[15] C. Holmes, M. Tanaka, M. Wyatt, A. A. Awan, J. Rasley, S. Rajbhandari, R. Y. Aminabadi, H. Qin,

A. Bakhtiari, L. Kurilenko, and Y. He, "Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference," 2024.

[16] B. Lin, T. Peng, C. Zhang, M. Sun, L. Li, H. Zhao, W. Xiao, Q. Xu, X. Qiu, S. Li, Z. Ji, Y. Li, and W. Lin, "Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache," 2024.

[17] "Huggingface: Text generation inference," https://github.com/huggingface/text-generation-inference.

[18] "Nvidia: Tensorrt-llm," https://github.com/NVIDIA/TensorRT-LLM.

[19] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for Transformer-Based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, 2022, pp. 521–538. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/yu

[20] J. Y. Yang, B. Kim, J. Bae, B. Kwon, G. Park, E. Yang, S. J. Kwon, and D. Lee, "No token left behind: Reliable kv cache compression via importance-aware mixed precision quantization," 2024.

[21] A. Dhakal and K. Ramakrishnan, "Netml: An nfv platform with efficient support for machine learning applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 396–404.

[22] A. Dhakal, S. G. Kulkarni, and K. Ramakrishnan, "Gslice: controlled spatial sharing of gpus for a scalable inference platform," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 492–506.

[23] "Gdrcopy," https://github.com/NVIDIA/gdrcopy.

[24] "vllm," https://github.com/vllm-project/vllm, 2024.