VASUKI: MINIMIZING MAKESPAN FOR OFFLINE LLM BATCH INFERENCE (TEAM 8)

Kasra A. Sohrab [*1] **Lisa L. Shi** [*1] **Shravan Cheekati** [*1] **Akaash R. Parthasarathy** [*1]

ABSTRACT

The current state-of-the-art for large language model (LLM) inference serving is focused on the online scenario, where new requests continuously enter the system, and the main priority is meeting tight latency SLO constraints. These systems are not optimized for the use case of offline inference, which involves generating outputs for a series of prompts that are all known before serving begins. In the offline case, metrics such as time-to-first-token (TTFT) and time-between-tokens (TBT) are unimportant, and the primary focus is on minimizing makespan (end-to-end completion time) and cost subject to memory constraints. We propose Vasuki, an offline inference-focused LLM serving system built on top of Sarathi-Serve that utilizes KV cache offloading and bin packing to effectively navigate this memory-makespan-cost tradeoff space. We demonstrate that our method provides a 9% end-to-end speedup for summarization tasks and 36% end-to-end speedup for translation tasks compared to SoTA on the same hardware. Vasuki also contributes to a reduction in total cost, due to reducing active GPU time by minimizing the makespan and by enabling the use of cheaper hardware to approach the performance of hardware with twice the memory capacity running a SoTA inference system.

1 INTRODUCTION

LLM offline inference involves running inference against large amounts of input data at a time. Offline inference in production is often run with scheduled jobs repeated after some time interval with large amounts of data (Kamsetty et al., 2023). The objective for offline inference can simply be viewed as minimizing the makespan, the end-to-end time for serving a batch of data (MathWorks, 2024). This differs from online inference, where the environment must continuously serve requests that enter. In particular, the objective for online inference is to effectively manage the latency/throughput tradeoff space since requests may arrive at different points of time, and the system needs to show continuous progress for better user experience.

Today, vLLM and the Sarathi paper are the state-of-the-art solutions for both offline and online inference (Kwon et al., 2023) (Agrawal et al., 2023). However, these frameworks do not treat the problem of offline inference differently even though the key objective and the a priori information differ. There is an opportunity gap to deliver a better solution by taking advantage of the extra information and fewer constraints present when a workload is offline (order of request completion and latency do not matter).

In addition, the cost dimension has not been explicitly optimized on for offline inference workloads. Cheaper GPU hardware introduces the tradeoff of less memory, and since memory is crucial to increase batch sizes (and thus throughput), optimizing for memory-constrained scenarios is vital to a cost-aware solution.

We present Vasuki, a first-class solution for offline LLM inference that utilizes KV cache offloading and bin packing to effectively navigate this memory-makespan-cost tradeoff space. The main contributions of this paper are:

1. Dynamic KV-cache CPU memory offloading based on predicted offloading and batch execution time
2. Greedy request bin-packing scheme to help navigate the memory constraints introduced by cheaper GPU variants

2 RELATED WORK

The state-of-the-art in LLM inference serving primarily addresses online scenarios, where latency-centric metrics like time-to-first-token (TTFT) and time-between-tokens (TBT) dominate the optimization goals. However, several systems, while designed for different use cases, offer insights or partial solutions relevant to offline inference.

vLLM introduces an innovative memory management sys-

[*]Equal contribution [1]College of Computing, Georgia Institute of Technology, Atlanta, GA, United States. Correspondence to: Akaash R. Parthasarathy <akaashrp@gatech.edu>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2024.

tem optimized for online inference workloads by leveraging a dynamic Key-Value (KV) cache system. This approach effectively reduces memory fragmentation and maximizes GPU utilization, achieving low latency and high throughput (Kwon et al., 2023). While vLLM excels in handling bursty, real-time requests, its focus on online inference leaves it suboptimal for offline scenarios. The absence of mechanisms like batch reordering or speculative execution tailored to minimize makespan hinders its applicability to offline workloads, where such features could exploit the static nature of task batches for efficiency gains.

Sarathi-Serve builds off of vLLM and introduces chunked-prefills in order to allow for stall-free schedules because new requests get added to a batch without any pauses in the decode (Agrawal et al., 2024b). By doing so, Sarathi-Serve is able to collapse the tradeoff space of throughput and time-between-tokens (TBT) tail latency, and as a result, serve high throughput with low latency. However, Sarathi-Serve is not fully optimized for offline serving, so our project builds off the chunked-prefill technique to develop a serving system that prioritizes optimizing throughput for inference.

SpecInfer proposes speculative decoding techniques to accelerate LLM inference by parallelizing partial token generation across multiple speculative paths (Miao et al., 2024). This method mitigates latency bottlenecks in online settings by reducing the dependence on sequential decoding. Although SpecInfer's speculative decoding could be adapted for offline inference to speed up the generation of predetermined batches, the system's design prioritizes real-time token generation rather than optimizing for offline-specific metrics like cost or makespan. Thus, SpecInfer highlights a promising avenue—leveraging speculative decoding—but does not comprehensively address the offline inference paradigm.

Work such as FlexGen has attempted to tackle the tradeoff space induced by offline LLM inference serving. FlexGen focuses on enabling efficient LLM inference under strict resource constraints, particularly in low-memory environments (Sheng et al., 2023). By employing offloading techniques to CPU and disk, FlexGen introduces flexibility in balancing memory and compute trade-offs. While primarily targeting resource efficiency, its offloading mechanisms align well with the needs of offline inference, where cost minimization is often critical. However, FlexGen does not fully exploit the deterministic nature of offline workloads for optimizations like bin packing or scheduling strategies.

ConServe discusses co-locating offline batch jobs with online serving and harvesting GPUs from online serving to make progress on offline jobs (Qiao et al., 2024). This work introduces a new way to reduce the cost of offline serving but does not contribute to minimizing an offline job's total makespan, since the running offline jobs will be preempted by online jobs at peak usage times.

Finally, DéjàVu proposes a KV cache streaming scheme as part of its solution. But it neither focuses on cost nor considers offloading of the KV cache at the same granularity as Vasuki. DéjàVu focuses on the KV cache at the micro-batch pipeline level, and they always stream the KV cache layer-by-layer. They perform no bandwidth prediction and thus assume favorable hardware deployments (Strati et al., 2024).

3 Background [Lisa]

In this section, we describe the architecture and typical inference procedure for LLMs. We also discuss metrics typically used in online LLM inference serving and why these are unimportant for the offline use case.

3.1 LLM Architecture and LLM Inference

LLM Architecture Currently, most popular LLMs are decoder-only transformer models, such as GPT-3 (Brown et al., 2020) and LLaMA (Grattafiori et al., 2024). These models have a stack of layers, each structured in a similar way. Specifically, each layer has a self-attention module and a feed-forward network (FFN) (Alammar, 2018).

The self-attention module is a core part of the transformer architecture and allows for each part of a sequence to take into account the previous parts for generating a contextual representation. During computation, specific Query (Q), Key (K) and Value (V) vectors correspond to each input token. These input tokens are obtained via a linear transformation. From there, the attention operator computes a semantic relationship between the tokens by performing a dot-product of the Q and K vectors of all preceding tokens. Afterwards, in order to compute a weighted average of the V vector, the softmax operation is used to obtain the weights. Additionally, the attention computation can have multiple heads, where all the outputs can then be combined using a linear transformation.

The FFN typically has two linear transformations, with a non-linear transformation in between the two layers. The first transforms an input token embedding of dimension h to a higher dimension. From there, there is an activation function applied, such as ReLU. Finally, the second linear layer takes the token embedding back to the original dimension h.

LLM Inference Due to the nature of the LLM architecture, LLM inference can be considered as consisting of two distinct phases: a prefill and decode phase. During the prefill phase, the input prompt is processed and then produces the first output token. Next, the decode phase takes over and generates output tokens one at a time. Finally, the token

generated in the previous step is passed through the model to generate the next token. This process continues until an end-of-sequence token is generated. One key aspect is that the decode phase requires access to all the keys and values associated with the previously processed tokens to perform the attention operation. One common technique to avoid recomputation is for activations to be stored in a KV cache.

The prefill phase is compute-bound since the prompt tokens can be processed in parallel in a single iteration, allowing for saturation of arithmetic intensity. On the other hand, decode phase is memory-bound. This is because a full forward pass of the model is required over a single token generated from the previous iteration.

As a result, naively processing multiple requests sequentially results in under-utilization of GPU compute. Thus, LLM serving systems often leverage batching to process multiple requests concurrently, which is effective when requests are in the decode phase. PagedAttention allows for even more requests to execute concurrently by addressing the fragmentation issue in the KV cache (Kwon et al., 2023). Sarathi-Serve also helps to mitigate the issue by implementing stall-free batching by using chunked-prefills to execute prefills during the decode phase, which takes advantage of the compute slack (Agrawal et al., 2023). However, there are still memory limitations that affect throughput, where large chunk sizes are not possible when they exceed memory capacity or when they result in too much swapping.

3.2 Offline LLM Inference Serving Metrics

Online serving metrics: Online LLM serving systems typically have service-level objectives (SLOs) that they need to meet. The subject of these SLOs is often tail latency. Typical metrics that online LLM inference serving systems use include TTFT (time to first token) and TBT (time between token) (Nvidia, 2024). However, these metrics are not ideal measures of success for offline serving systems because latency is not a crucial concern.

Offline serving metrics: In offline inference serving, one key metric we focus on instead is to measure offline model serving cost, given by

$$\text{model serving cost} = \frac{\text{QPS}}{\$/\text{FP32 Gigaflop/Watt}}$$

We measure QPS (queries per second) as the total number of requests processed over the total amount of time. This allows us to consider cost by taking into account the overall throughput of the system, which is a useful metric for our use case of offline serving systems. The unit of cost we choose to use also takes into consideration the compute power and energy used over time. This is also an important consideration in measuring the effectiveness of offline serving system in commodity hardware settings. We also examine makespan as a key metric. Considering makespan allows us to prioritize overall system throughput. In this manner, tail latency is no longer considered a key success metric.

Offline serving is especially relevant to large companies, which may need to run inference on a large amount of data, in contexts that are not tail latency-sensitive. In these scenarios, prioritizing high throughput (measured via makespan) allows for more efficient usage of hardware, and thus lower costs.

4 MOTIVATION

In this section, we discuss the need for a first-class solution for offline LLM batch inference and the specific challenges and opportunities that commodity hardware presents.

4.1 Lack of First-Class Solution

The vast majority of the current state-of-the-art for LLM inference serving is focused on online use cases (Kwon et al., 2023) (Agrawal et al., 2024b). In online inference serving, new requests continuously arrive, and systems that are tailored for such scenarios often need to handle bursty, unpredictable workloads. As discussed in 3.2, such systems must optimize for tail latency-related metrics such as TTFT and TBT in order to provide a steady stream of tokens to end users. Although offline inference also involves generating outputs for a series of prompts, all of these prompts are known prior to the beginning of serving. In addition, since offline jobs do not need to optimize for user experience, tail latency does not accurately quantify the goal of offline serving systems. In particular, as discussed in 3.2, offline serving systems must optimize for makespan and model serving cost.

Despite offline inference serving having several use cases for large companies, there does not exist any dedicated system for this task. As discussed in 2, the current state-of-the-art systems for LLM inference serving, Sarathi-Serve and vLLM, do not explicitly handle the offline use case or make adjustments to their serving mechanism for this entirely different paradigm. That is, there exists *no first-class solution for offline LLM inference serving*. In particular, since offline serving systems need to optimize for different metrics, existing online-focused systems must be modified to enable them to effectively tackle the offline scenario. We discuss concrete changes that can be made to existing LLM inference serving systems in section 5.

4.2 Commodity Hardware Challenges and Opportunities

The key to optimizing on the cost dimension is the choice of hardware. For the purposes of this paper, we define com-

Table 1. A10 and A40 Hardware Specifications

GPU	A10	A40
$/FP32 GIGAFLOP/WATT	$12	$52
MEMORY CAPACITY	24 GB	48 GB
COMPUTE	125 TF	150 TF
MEMORY BANDWIDTH	600 GB/S	696 GB/S
INTERCONNECT	PCIE GEN 4: 64 GB/S	NVLINK: 112 GB/S, PCIE GEN 4: 64 GB/S

modity hardware as GPUs that have significantly cheaper costs and similar compute within the same generation but with a much lower memory capacity. This is evident from 1, which compares the various dimensions of the NVIDIA A10 and A40 GPUs (Morgan, 2021). In particular, although A10s and A40s have similar compute capabilities, an A10 has half the memory capacity of an A40. The lack of a high-speed interconnect in commodity GPUs also presents certain challenges in terms of deciding data, tensor, and pipeline parallelism configurations. However, we focus on the single-GPU case in this work.

4.3 Designing for Throughput

In order to examine which scheduler to build on top of, we compared the execution time in three key setups: vLLM on 24 GB, Sarathi on 24 GB, and Sarathi on 1000 GB, as seen in Figure 1. The first two setups were the main motivations behind using Sarathi to build our solution. The chunked prefills utilized by Sarathi are able to better distribute memory overhead better than vLLM. This was significant to consider in our case, because of our focus on throughput and memory efficiency. After deciding on Sarathi, we wanted to explore if memory limitations still played a role by conducting an experiment in Vidur where we simulated essentially unlimited memory for the same compute. We found that memory limitations were still apparent since an increase in chunk size can lead to memory capacity being exceeded as well as too much swapping. These results motivated us to build our scheduler on top of Sarathi and more closely examine memory efficiency in order to improve throughput. These results also showed that compute is not the limiting factor in execution time, though we also note that at larger chunk sizes more decodes are batched together and this memory pressure means more requests will restart and contribute to a slightly higher end-to-end time.

5 VASUKI: DESIGN AND IMPLEMENTATION

We now discuss the design and implementation of Vasuki, a system that minimizes makespan for offline LLM batch inference via two key techniques: KV cache offloading and bin packing.

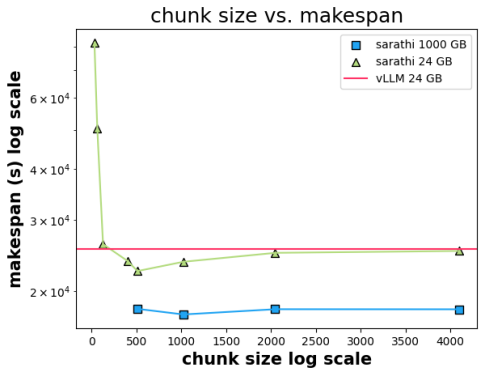

Figure 1. Running Vidur on the arxiv summarization dataset to compare Sarathi execution time with varied memory capacity, against the baseline of vLLM

5.1 KV Cache Offloading

Given the memory limitations imposed by commodity hardware, we discuss algorithms for mitigating memory pressure through the popular technique of offloading key-value (KV) caches. Here, offloading refers to the technique of migrating tensors from GPU to CPU memory in order to reduce GPU memory utilization. Since the KV cache size is directly proportional to the number of layers and the batch size and grows quadratically with sequence length, offloading is crucial to allow models to handle long sequences and operate on larger batch sizes. This not only alleviates memory pressure but also improves compute utilization.

For the particular use case of LLM inference serving, we observe that it is possible to offload the KV cache associated with certain decode layers of the model based on the offloading time for these caches and the execution time for a given layer. In order to estimate offloading time, we integrated bandwidth profiling for device-to-host (d2h) and host-to-device (h2d) transfers on top of Vidur (Agrawal et al., 2024a). In particular, we measured the d2h and h2d transfer time for a variety of configurations up to 16GB and calculate the corresponding bandwidths. We observed that,

Algorithm 1 Maximize Number of Layers Offloaded

1: $l = 0$
2: $r = \lfloor \frac{\text{num\_layers}}{2} \rfloor$
3: **while** $l < r$ **do**
4: $mid = \lfloor \frac{l+r}{2} \rfloor$
5: bandwidth = GetOffloadingBandwidth(layer_mem$\times$ mid)
6: $d2h\_time$ = CalculateTransferTime(layer_mem $\times$ mid, bandwidth.d2h)
7: $h2d\_time$ = CalculateTransferTime(layer_mem $\times$ mid, bandwidth.h2d)
8: offload_time = $(d2h\_time + h2d\_time)$
9: offload_time += (alloc_time + dealloc_time) $\times$ num_requests $\times$ mid
10: **if** offload_time $\leq$ exec_time$\times$(num_layers$-2\times mid$) **then**
11: $l = mid + 1$
12: **else**
13: $r = mid - 1$
14: **end if**
15: **end while**
16: num_offload_layers $= l - 1$

Algorithm 2 Minimize Number of Layers in Memory

1: $l = 0$
2: $r = \lfloor \frac{\text{num\_layers}}{2} \rfloor$
3: **while** $l < r$ **do**
4: $mid = \lfloor \frac{l+r}{2} \rfloor$
5: bandwidth = GetOffloadingBandwidth(layer_mem$\times$ mid)
6: $d2h\_time$ = CalculateTransferTime(layer_mem $\times$ mid, bandwidth.d2h)
7: $h2d\_time$ = CalculateTransferTime(layer_mem $\times$ mid, bandwidth.h2d)
8: offload_time = $(d2h\_time + h2d\_time)$
9: offload_time += (alloc_time + dealloc_time) $\times$ num_requests $\times$ mid
10: **if** offload_time $\leq$ exec_time $\times$ mid **then**
11: $r = mid - 1$
12: **else**
13: $l = mid + 1$
14: **end if**
15: **end while**
16: layers_in_memory $= (l + 1) \times 2$
17: num_offload_layers = num_layers $-$ layers_in_memory

up to a certain threshold, the bandwidth increases linearly with data size and then saturates for both d2h and h2d transfer time. We estimate this saturation point based on the percentage change in bandwidth from one data size to the next. For data sizes below this threshold, we extend Vidur's random forest-based execution time predictor to estimate transfer bandwidths (Agrawal et al., 2024a).

Based on the predicted offloading and loading bandwidths and execution time, we propose an algorithm in 1 for KV cache offloading. We choose to offload the KV cache of decode layers in the model to the extent to which communication time can be perfectly overlapped with computation, adding no additional makespan overhead due to offloading and loading. Given a batch of requests, the algorithm begins by calculating the execution time for the batch, the number of decode requests in the batch, and the total memory required to store the KV caches associated with these decode requests. It then performs binary search on the bounds $l = 0$ and $r = \lfloor \frac{num\_decode\_layers}{2} \rfloor$ (lines 1-2) to determine the maximum number of layers that can be offloaded while perfectly overlapping communication with computation. Note that num_decode_layers refers to the number of decode blocks in the LLM model. The algorithm determines the bandwidth, d2h transfer time, and h2d transfer time and calculates the total offloading time accordingly (lines 5-9). Finally, the algorithm checks if it is able to offload the KV cache for the first mid layers and load the KV cache for the last mid layers within the time it takes to complete the execution time of the remaining $(num\_decode\_layers - 2 \times mid)$ layers (line 10). An example of the working of this algorithm is provided in the file `animations/videos/offloading_algorithm1.mp4` in the supplementary material.

We observe that it is impossible to offload the KV cache associated with more than $\lfloor \frac{num\_decode\_layers}{2} \rfloor - 1$ layers using this algorithm. Since this is less than half of the number of decode layers in the model, we also propose a cyclic algorithm in 2 that allows offloading the KV cache of up to $num\_decode\_layers - 2$ layers. Prior to the binary search, this algorithm has the same initial steps as the previously described algorithm. However, rather than directly maximizing the number of layers that can be offloaded, this algorithm minimizes the number of layers that need to be in GPU memory at any given time. Specifically, given mid layers, the algorithm checks if it is able to offload the KV cache of the previous mid layers and load the KV cache of the next mid layers within the time it takes to complete the execution time of the current layers (line 11). The algorithm thus cyclically offloads the previous mid layers and loads the next mid layers, ensuring that only $2 \times mid$ layers are in memory at a given time. This algorithm requires the assumption that the execution time for each decode layer of the LLM is the same across a given batch. Since SoTA LLMs, such as Llama3, generally have identical decode blocks, we find that this assumption is reasonable, especially for our current single-GPU focus (Grattafiori et al., 2024). An example of the working of this algorithm is provided in the file `animations/videos/offloading`

`_algorithm2.mp4` in the supplementary material.

Depending on the trace. model selection, or hardware deployment, one algorithm may perform worse than the other. As such, we use the maximum of the values calculated by the two algorithms to determine both which algorithm to use and the maximum number of layers that can be offloaded while perfectly overlapping communication with computation. Offloading in such a manner allows us to minimize memory pressure on the GPU and increase the number of decode requests that can fit in a batch, leading to an overall decrease in makespan.

5.2 Bin Packing

We noted first that there is a compute cost in evicting a running request (restart) when the scheduler runs out of memory and second, the observation from the Sarathi paper that decodes can become compute-bound at high batch sizes (Agrawal et al., 2023).

An optimal use of memory to limit the number of batch iterations with the vLLM scheduling policy is closely formulated with the MILP in Appendix A. This formulation takes unreasonably long to run even for a small number of prompts and does not consider the growing rate of the prompt (1 new token each time it is scheduled) and additional complexities of the Sarathi scheduler, where prefills can be running for multiple time steps before decode can begin. However, this motivates finding an approximate bin packing scheme that approaches the optimal value of this MILP.

Thus to avoid restarts at high batch sizes, the memory allocation scheme first allocates enough GPU blocks to fit all the KV layers for the prompt tokens. Since going forward, not all the KV cache layers will be stored GPU memory, greedily allocating in this format will help reduce restarts.

6 EVALUATION

6.1 Experimental Setup

We evaluate the efficacy of Vasuki's offloading and bin packing technique extensively. The setup for our experiments is described below.

6.1.1 Profiling Framework and Tools

We utilized Vidur, a simulator for large-scale LLM inference workloads, as the primary framework for experimentation (Agrawal et al., 2024a). Vidur provides detailed profiling capabilities for various LLM components, such as attention and MLP execution times. These features allowed us to build accurate extensions to execution time predictors for the simulation of CPU-GPU interconnect transfer times and analyze bandwidth usage under different configurations. To simulate real-world workloads, we incorporated representative traces from offline inference scenarios, as detailed below.

6.1.2 Hardware Profiling

Our experiments spanned GPU architectures including NVIDIA RTX 4090, NVIDIA A40, and NVIDIA A10 GPUs. This enabled a comprehensive analysis of Vasuki's performance across various hardware configurations. Profiling metrics included memory bandwidth, kernel execution times, and interconnect transfer rates, all critical for understanding the efficiency of bin packing and offloading strategies.

Unfortunately, we were not able to gain access to NVIDIA A10 GPUs. However, to enable using A10s in simulation testing, we treat profiling data from A40 as the same as A10 since the compute is similar between the two, as seen in Table 1.

6.1.3 Workloads

Our evaluation employed traces representative of offline inference scenarios, focusing on two datasets:

1. BWB (Bilingual Web Book): This dataset consists of Chinese novels translated into English. With a low Prompt-to-Decode token (P:D) ratio (< 1 on average), this workload represents tasks where the translation output is close to or longer than the input (Jiang et al., 2022).

2. arXiv Summarization: This dataset involves summarizing academic papers. However, this features a higher P:D ratio, this workload reflects scenarios where the output (summary) is significantly shorter than the input (Cohan et al., 2018).

6.1.4 Models and Other Parameters

We tested context lengths up to 4096 tokens, capturing a subset of realistic scenarios for both datasets while ensuring compatibility with modern LLMs. We used LLama3-8B for all experiments, which has 32 layers. Finally, the chunk size was fixed to 512 for all the following experiments.

6.2 Results

6.2.1 Batch Size

First, we demonstrate in 2 and 3 that we can enable larger batch sizes by having more GPU memory available.

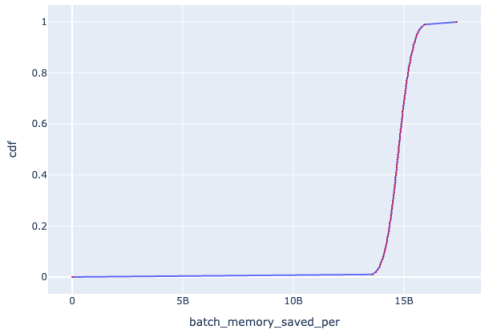

Figure 2. CDF of the amount of memory in bytes Vasuki was able to offload to CPU memory in the BWB offline case using A10

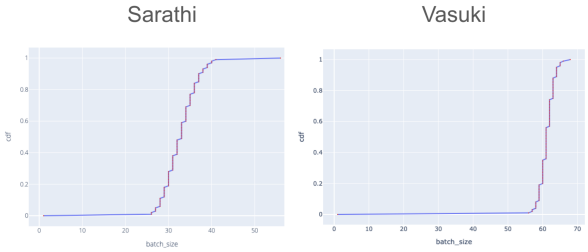

Figure 3. Batch size CDF for offline inference with the BWB trace using A10

Memory saved per batch is calculated as

$$\begin{aligned}\text{memory saved} = &\text{ number of requests in batch} \\ &\times \text{number of offloaded layers} \\ &\times \text{size of decode layer}\end{aligned} \quad (1)$$

This significant amount of memory saving, where 99% of the memory saved per batch is at least 14GB, which enables batch sizes to be twice as large.

6.2.2 *Makespan*

4 and 5 demonstrate the makespan reduction using Vasuki compared to Sarathi. Vasuki approaches the makespan of Sarathi on A40 and outperforms by a consistent amount on both single GPU RTX4090 and A10. More decode pressure present in the BWB translation trace means our improved memory allocation approach will better display its gains.

6.2.3 *Cost*

On the cost dimension, we show the cost-effectiveness of Vasuki by comparing the QPS / cost in Figure 6. Vasuki is an order of magnitude more cost-effective by this metric in the summarization task since the makespans are similar but

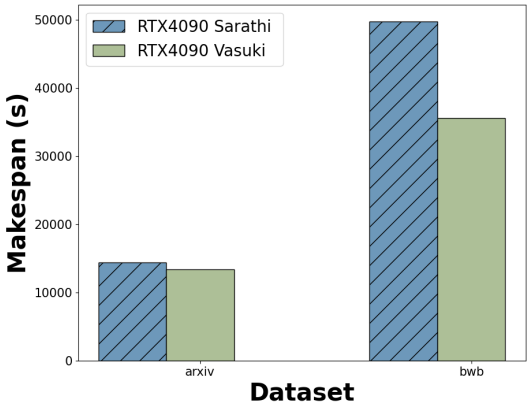

Figure 4. Comparing makespan of Sarathi and Vasuki with RTX4090

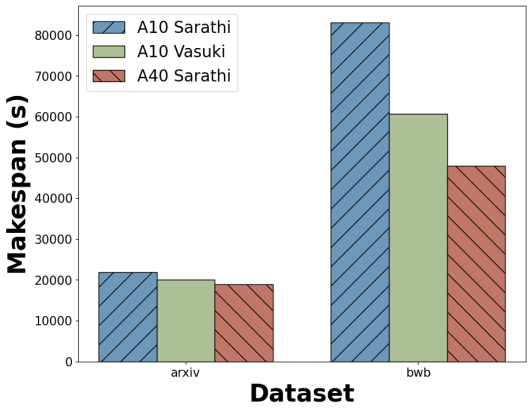

Figure 5. Comparing makespan of Sarathi and Vasuki

Vasuki uses much cheaper hardware. In the translation task, Vasuki helps deal with the greater memory pressure and is 3.4x more cost-effective.

6.2.4 *Correctness*

We show that the current results are reasonable and correct in simulation by examining the total offloading time compared to the execution time of a single batch in Figure 7. Total offloading time is consistently an order of magnitude smaller than the total batch execution time, making it possible to offload 30 of the 32 total layers in Llama3-8b.

7 DISCUSSION

In this paper, we begin to explore the opportunities to increase throughput in memory-limited scenarios. However, there are still several challenges to address in future work.

First, we wish to investigate further what configurations may favor one offloading approach over another, to see if the algorithm can be simplified further. We anticipate test-

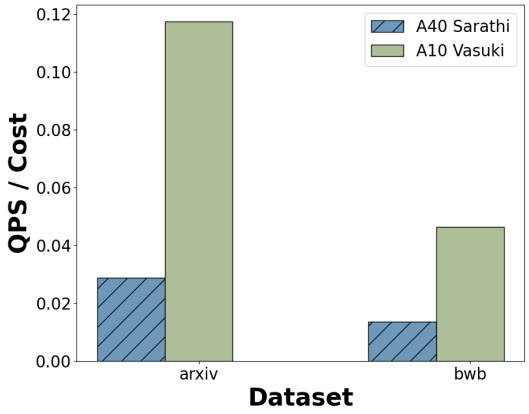

Figure 6. Comparing QPS/cost using the equation from 3.2 and cost data from Table 1

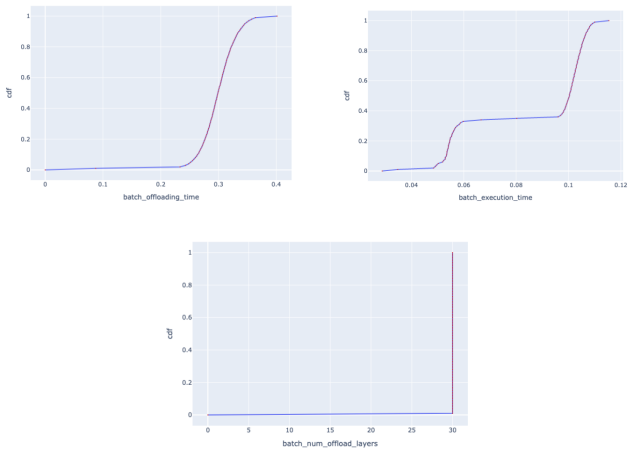

Figure 7. CDFs of model execution time (in s) and offloading time (in ms), showing that we were able to ultimately offload 30 layers without computation overlap using A10

ing with more models, hardware deployments, and longer context lengths will help answer this.

Second, future work may help determine if we can improve bin-packing by dynamically de-allocating blocks if all blocks are not in use after prefill. If we are able to offload more layers than we allocated from the previous iterations, it could permit more requests to fit in the batch and further increase throughput, especially in higher P:D situations like summarization.

Third, supporting weight offloading will be critical to a comprehensive offline serving solution. This will allow the system to serve models that otherwise could not fit in memory. Fourth, scaling up to multiple GPUs will require some scheduling consideration to find a good allocation scheme, between data, tensor, and pipeline parallelism, especially in the absence of high-speed interconnect. Works such as Varuna give a direction for tackling this problem, including focusing primarily on pipeline and data parallelism due to the expected networking costs (Athlur et al., 2022).

Finally, we believe speculative decoding can be part of a future offline serving solution, once weight offloading is supported efficiently since it can better utilize compute (Leviathan et al., 2023). Using an initial configuration explorer and some profiling of acceptance rates we can determine the best drafter model for throughput.

8 Conclusion

Offline LLM batch inference is a challenging problem that currently has no first-class solution. We discuss the primary differences between online and offline LLM inference serving and presented metrics that are crucial to the offline serving case, which current systems do not explicitly optimize on. Further, we discuss cost optimization in the context of hardware choice and present challenges associated with memory-limited hardware. To address these challenges, we introduce Vasuki, a system that minimizes makespan and maximizes QPS/cost for offline LLM batch inference via two key techniques: KV cache offloading and bin packing [todo: align with section 5]. In particular, Vasuki offloads the KV cache associated with particular layers during the generation of a decode token in order to reduce GPU memory pressure and increase the extent of batching. This is supplemented by bin packing, which allows Vasuki to maximize the number of requests in each batch, resulting in an overall decrease in makespan. Our evaluation shows Vasuki provides a 9% end-to-end speedup for summarization tasks and 36% end-to-end speedup for translation tasks compared to SoTA on the same hardware. Vasuki also contributes to a reduction in total cost, due to reducing active GPU time by minimizing the makespan and by enabling the use of cheaper hardware to approach the performance of hardware with twice the memory capacity running a SoTA inference system.

Acknowledgements

This work would not have been possible without the support of Amey Agrawal and Professor Alexey Tumanov throughout the semester. We are thankful to them for all of their suggestions and guidance.

References

Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023. URL https://arxiv.org/abs/2308.16369.

Agrawal, A., Kedia, N., Mohan, J., Panwar, A., Kwatra,

N., Gulavani, B., Ramjee, R., and Tumanov, A. Vidur: A large-scale simulation framework for llm inference, 2024a. URL https://arxiv.org/abs/2405.05465.

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., Tumanov, A., and Ramjee, R. Taming throughput-latency tradeoff in llm inference with sarathi-serve, 2024b. URL https://arxiv.org/abs/2403.02310.

Alammar, J. The illustrated transformer [blog post], 2018. URL https://jalammar.github.io/illustrated-transformer/.

Athlur, S., Saran, N., Sivathanu, M., Ramjee, R., and Kwatra, N. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pp. 472–487, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391627. doi: 10.1145/3492321.3519584. URL https://doi.org/10.1145/3492321.3519584.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL https://arxiv.org/abs/2005.14165.

Cohan, A., Dernoncourt, F., Kim, D. S., Bui, T., Kim, S., Chang, W., and Goharian, N. A discourse-aware attention model for abstractive summarization of long documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pp. 615–621, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-2097. URL https://aclanthology.org/N18-2097.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., and et al. The Llama 3 Herd of Models. *arXiv e-prints*, art. arXiv:2407.21783, July 2024. doi: 10.48550/arXiv.2407.21783.

Jiang, Y. E., Liu, T., Ma, S., Zhang, D., Sachan, M., and Cotterell, R. A bilingual parallel corpus with discourse annotations, 2022. URL https://arxiv.org/abs/2210.14667.

Kamsetty, A., Chen, H., and Xie, L. How bytedance scales offline inference with multi-modal llms to 200 tb data. Technical report, Anyscale, 2023.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. 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.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding, 2023. URL https://arxiv.org/abs/2211.17192.

MathWorks. Minimize makespan in parallel processing, 2024.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Wang, Z., Zhang, Z., Wong, R. Y. Y., Zhu, A., Yang, L., Shi, X., et al. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 932–949, 2024.

Morgan, T. P. Nvidia rounds out "ampere" lineup with two new accelerators, 2021. URL https://www.nextplatform.com/2021/04/15/nvidia-rounds-out-ampere-lineup-with-two-new-accelerators/.

Nvidia. Nim for llm benchmarking guide, 2024. URL https://docs.nvidia.com/nim/benchmarking/llm/latest/metrics.html#time-to-first-token-ttft.

Qiao, Y., Anzai, S., Yu, S., Ma, H., Wang, Y., Kim, M., and Xu, H. Conserve: Harvesting gpus for low-latency and high-throughput large language model serving, 2024. URL https://arxiv.org/abs/2410.01228.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pp. 31094–31116. PMLR, 2023.

Strati, F., Mcallister, S., Phanishayee, A., Tarnawski, J., and Klimovic, A. Déjàvu: Kv-cache streaming for fast, fault-tolerant generative llm serving, 2024. URL https://arxiv.org/abs/2403.01876.

A MILP FORMULATION OF OPTIMAL REQUEST ORDERING FOR VLLM

Algorithm 3 MILP Formulation of Optimal Request Ordering

{Decision Variables}
$x_{i,t} \in \{0,1\}$: Binary variable indicating when request i starts processing
$e_{i,t} \in \{0,1\}$: Binary variable indicating when request i completes processing
$y_{i,t} \in \{0,1\}$: Binary variable tracking request i's presence at time t
$c \in Z^+$: The maximum number of tokens the system can hold at any scheduling step
$S_i \in Z^+$: The maximum length of request i
$M \in Z^+$: Makespan variable (total processing time, unit is scheduling batch steps)
{Objective: Minimize Total Processing Time}
$\min \quad M$
Request Initiation Constraint:
$\sum_{t \in T} x_{i,t} = 1 \quad \forall i \in I$ {Each request must start exactly once}
Request Completion Constraint:
$\sum_{t \in T} e_{i,t} = 1 \quad \forall i \in I$ {Each request must complete exactly once}
Request Lifecycle Constraint:
$e_{i,t+\text{decode\_steps}_i} = x_{i,t} \quad \forall i \in I, t \in T$ {Exit time matches start time plus decoding steps}
Makespan Tracking Constraint:
$M \geq t \cdot e_{i,t} \quad \forall i \in I, t \in T$ {Track maximum completion time, which is the makespan}
Request Presence Constraints:
$y_{i,t} \geq \sum_{\tau \leq t} x_{i,\tau} - \sum_{\tau < t} e_{i,\tau}$
$y_{i,t} \leq \sum_{\tau \leq t} x_{i,\tau} - \sum_{\tau < t} e_{i,\tau}$ {Track request's active duration}
System Capacity Constraint:
$\sum_{i \in I} y_{i,t} \cdot S_i \leq c \quad \forall t \in T$ {Limit total tokens in system}
Optimal request scheduling minimizing makespan number of batch steps

B STATEMENT OF COLLABORATOR CONTRIBUTION

Shravan Cheekati (LS, KS, ARP): Shravan worked on initial algorithm planning, profiling hardware for commodity hardware, benchmarking SOTA commodity hardware algorithms, item tracking for Vidur offloading.

Kasra Sohrab (SC, LS, ARP): Kasra worked on throughput experiments, speculative decoding motivation experiments we used for the midterm, the various iterations of the bin packing algorithm, and fixes and metrics related to the offloading algorithms.

Lisa Shi (SC, KS, ARP): Lisa worked on running experiments to demonstrate the motivation behind designing for throughput (showing that memory bound was a limiting factor of makespan), analyzing the makespan and cost results, and helping to develop the first iteration of the bin-packing algorithm.

Akaash Parthasarathy (SC, LS, KS): Akaash integrated transfer time and bandwidth profiling into Vidur, added support for bandwidth prediction by extending Vidur's execution time predictor, and implemented the two KV cache offloading algorithms described in the paper.