ZvLLM: Zigzag forward pass with vLLM

Henry Zhu
University of Illinois Urbana-Champaign
zhu109@illinois.edu

Siyuan Chai
University of Illinois Urbana-Champaign
siyuanc3@illinois.edu

Aditya Prerepa
University of Illinois Urbana-Champaign
prerepa2@illinois.edu

Abstract

Deploying Large Language Models (LLMs) on single GPU systems requires efficient resource management due to the limited availability and high demand of computational resources. Traditional methods struggle with balancing computational overhead and memory usage, particularly for managing model weights and key-value (KV) cache. To improve this, we adopt "Zigzag Scheduling" from the FlexGen framework within the vLLM architecture, which delays offloading model weights to secondary storage like CPU memory or disk. This approach allows multiple batches to be processed before offloading, reducing memory transfer frequency and minimizing I/O costs, thereby enhancing system efficiency. Zigzag Scheduling further optimizes hardware usage by overlapping the loading of weights for upcoming layers with the loading and storing of cache and activations for adjacent batches, significantly increasing throughput and reducing idle times. The integration of Zigzag Scheduling into vLLM not only streamlines memory management but also boosts overall performance, showing a 52% improvement in throughput over traditional methods.

1 Intro

Efficient resource management is critical for deploying Large Language Models LLMs on platforms with constrained resources such as single GPU systems. Traditional approaches often face challenges in balancing computational overhead and memory usage, particularly when managing model weights and key-value (KV) cache. To address these challenges, we integrate "Zigzag Scheduling" from the FlexGen framework with the vLLM serving architecture on a single GPU. This method strategically delays the offloading of model weights to secondary storage—such as CPU memory or disk—permitting the processing of multiple batches before any data offloading occurs. This reduces the frequency of memory transfers, thereby minimizing I/O costs and enhancing overall system efficiency.

Zigzag Scheduling optimizes the use of hardware by overlapping multiple operations: it synchronizes the loading of weights for upcoming layers with the loading and storing of cache and activations for adjacent batches. This overlapping maximizes throughput and minimizes idle times, allowing current batch computations to proceed without delay. By integrating Zigzag Scheduling into the vLLM framework, we capitalize on its efficient use of overlapping operations to improve system performance and resource utilization on a single GPU.

Incorporating Zigzag scheduling to vLLM's enhances its ability to manage large KV caches efficiently and maintain high computational performance without the frequent need to reload weights from

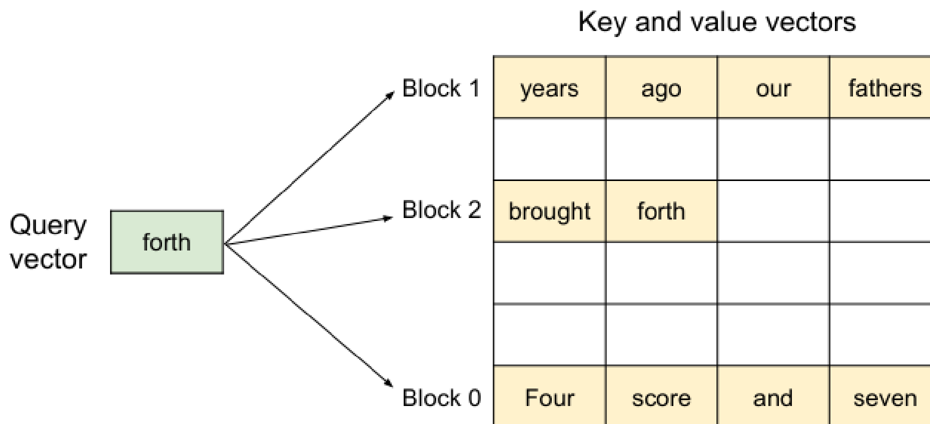
34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

secondary storage. We show a throughput improvement of 52% over the baseline (Huggingface Accelerate)

2 Related Work

PagedAttention PagedAttention addresses the challenge of efficient memory management in the serving of large language models (LLMs), particularly under the constraints imposed by the dynamic and sizeable memory allocation required for the key-value (KV) cache during LLM operations. Traditional methods often suffer from internal and external fragmentation issues because they allocate memory contiguously for KV cache, which can vastly differ in required size during execution, leading to inefficient memory utilization. PagedAttention, inspired by virtual memory systems in operating systems, divides the KV cache into blocks that can be stored non-contiguously, thus reducing memory waste and improving system throughput. This approach allows for flexible and dynamic memory allocation, closely resembling the paging technique of operating systems, which handles memory fragmentation by allowing data that is logically contiguous to be physically non-contiguous.

Figure 1: PagedAttention mapping to multiple blocks.



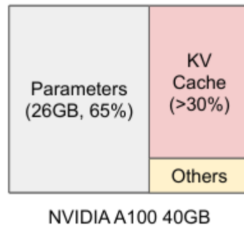
vLLM vLLM builds upon the foundation of PagedAttention to create a high-throughput, efficient LLM serving system. It utilizes a block-level memory management system that significantly reduces the wastage typically associated with KV cache memory allocation in existing systems. vLLM supports various LLM architectures and sizes, making it versatile for different deployment scenarios. By enabling near-zero waste in KV cache memory usage and flexible sharing across requests, vLLM not only enhances throughput but also maintains low latency levels, competitive with state-of-the-art systems like FasterTransformer and Orca. This system demonstrates particular efficiency improvements in scenarios involving complex decoding algorithms, longer sequences, and larger models, showcasing its capability to scale and manage memory more effectively than conventional LLM serving solutions.

FlexGen FlexGen is a system for serving large language models with high efficiency and low latency. It introduces a novel framework for model parallelism that combines tensor and pipeline parallelism to enable efficient execution of large models across multiple devices. FlexGen employs a just-in-time compilation strategy to optimize the execution plan for a given model and hardware configuration, yielding performance gains over traditional model parallelism approaches. It also incorporates techniques for effective memory management and communication optimization, allowing it to scale to models with billions of parameters while maintaining low latency. The authors demonstrate FlexGen's efficacy through benchmarks on various model sizes and hardware setups, showcasing its ability to serve large language models with high throughput and low overhead.

3 Background

The deployment of large language models in real-time applications has introduced significant challenges in terms of computational efficiency, particularly concerning memory management. LLMs, such as those based on the Transformer architecture, require extensive memory resources (see 2) primarily due to their need for maintaining large KV caches. These caches store the states needed for generating text sequences, with each request dynamically altering the memory footprint depending on the sequence length and complexity. Traditionally, memory management for LLMs has involved static allocation, which can lead to inefficiencies such as memory fragmentation. This fragmentation occurs because fixed-size memory blocks do not align well with the variable-sized data stored in KV caches, leading to underutilized memory spaces and reduced throughput.

Figure 2: Rough idea of GPU Memory Usage; Serving Transformer-based models is memory bound.



The introduction of systems like PagedAttention and vLLM marked significant advancements in the field of LLM serving by addressing these memory management issues. PagedAttention borrows concepts from virtual memory systems used in operating systems, such as paging, to manage memory fragmentation more effectively. It breaks down the KV cache into smaller, manageable blocks, allowing them to be stored non-contiguously. This method not only mitigates internal and external memory fragmentation but also enhances memory utilization by allowing more flexible data storage and retrieval.

Building on the foundations laid by PagedAttention, vLLM implements a block-wise memory management system that significantly improves the throughput of LLM serving systems. vLLM organizes the KV cache into blocks, dynamically allocating and deallocating them based on the needs of individual requests. This block-level management reduces wastage and allows for the serving of larger models or more concurrent requests within the same hardware constraints. By reducing the overhead associated with memory management, vLLM helps in maintaining low latency and high throughput, crucial for cost-effective LLM serving.

Despite these improvements, neither PagedAttention/vLLM fully addresses the potential of integrating more advanced memory hierarchies, such as swapping blocks to secondary storage devices. This integration could further optimize memory usage during peak loads or for particularly large requests. The idea of leveraging secondary storage is not new and has been explored in systems like FlexGen, which proposes offloading entire KV caches to secondary storage. However, this approach can lead to significant I/O overhead, especially when the data involved is large and not all of it is needed immediately upon retrieval.

4 Motivation

Overcome Single GPU's Memory Limitation While vLLM enabled blockwise memory management for KV cahce, such PagedAttention algorithm does not apply for model weights. Thus, vLLM fails to run models with size exceeding the GPU memory size on a single GPU. Nowadays, modern workloads have over 10B parameters which easily oversize a single GPU memory size. For example, Llama (2) has 70B, DBRX (6) has over 132 B, and Command R+ (1) has 104B parameters. Such increasingly large model size can not be fit on a single GPU which has max memory size of 80GiB; not to mention that commercial GPUs like RTX 3090 or 4090 have memory size of 24GiB. Even if for smaller models like OPT 7B and 13B, GPU still quickly runs out of memory; for example, OPT 13B using FP16 will have at least 26GiB of memory.

To help overcome such challenges, we enroll CPU memory and SSD to help. Previous works like ZERO-offlaod (4) and ZERO-infinity (3) offload offload part of weights and KV cache from GPU memory to CPU memory and SSD. As shown in figure 5, CPU memory and storage which have much larger capacity than a single GPU's memory can be used to offload weights. Existing framework like Flexgen has already enabled offloading to CPU memory by specifying the offloading percentage for weights, cache, and activation. We ported the offloading strategy to vLLM for weights of model so that vLLM can overcome the memory limitation of a single GPU.

Higher Throughput The other goal we wish to further improve the throughput. vLLM improves throughput by increasing memory utilization and reducing memory fragmentation. We wish to further improve the througput from the perspective of scheduling policy. We experiment the zig-zag policy analyzed in Flexgen on vLLM. We anticipate an increment in throughput as it balances the offloading of weights and KV cache.

5 Design & Implementation

Figure 3: Example of memory hierarchy

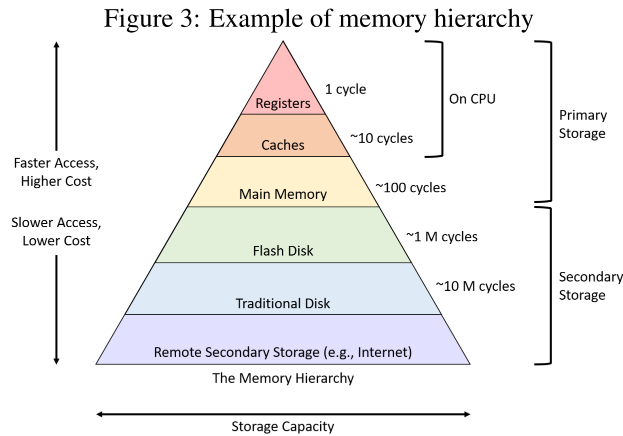
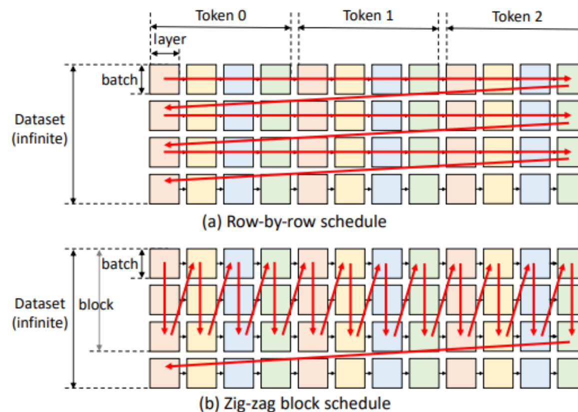


Figure 4: Row by row schedule and zig-zag schedule



Our approach hinges on two critical components. First, the finite capacity of GPU memory, currently capped at 80GB, poses a significant constraint for larger models. To mitigate this limitation, we leverage offloading techniques, which allow us to store weights in alternative storage mediums like CPU memory and SSDs. While these memory tiers offer significantly larger capacity, as illustrated in Figure 5, their longer access latency may hurt the throughut performance of the models. they also introduce new challenges that must be addressed.

With offloading as our foundation, we delve into a comparative analysis of two scheduling techniques: zig-zag block schedule and row-by-row schedule. These scheduling strategies play a pivotal role in managing the transfer of weights between different memory tiers. By evaluating their performance under the constraints of GPU memory limitations and the benefits of offloading, we aim to uncover insights that can enhance the efficiency and effectiveness of model training for large-scale deep learning applications.

Through our investigation, we seek to optimize the interplay between GPU memory limitations and the potential of offloading weights to alternative storage. Our analysis compares the effectiveness of zig-zag block scheduling and row-by-row scheduling techniques in managing weight storage across different memory tiers. By elucidating the strengths and weaknesses of each approach, our study aims to provide valuable insights for enhancing the performance and scalability of deep learning models.

5.1 Offloading

The integration of offloading techniques has been a focal point of research in recent years, with notable advancements such as the percentage offloading method employed in FlexGen (5). Our implementation extends this approach, allowing for the offloading of specific percentages of weights to the GPU, CPU, and disk, providing a flexible and dynamic approach to memory management.

In our design, we modify the forward pass to facilitate offloading, ensuring that weights are seamlessly swapped to the CPU after completing a pass. We built the offloading strategy on top of existing vLLM implementation.

However, while offloading presents a promising solution, it poses challenges when integrated with models like vLLM. Notably, vLLM pre-allocates CPU/GPU KV blocks, limiting our ability to manage memory allocation in these blocks. As a result, our allocator cannot directly influence how vLLM handles KV blocks, necessitating a nuanced approach to memory management in such contexts.

5.2 Zig-zag block schedule

The conventional approach to model computation involves passing one batch through each layer until all layers have processed the input, resulting in one output. Figure 5-a illustrates the row by row scheduling policy. However, this method becomes costly when employing offloading, as each input and weight must be reloaded to GPU from CPU for every pass through each layer. An alternative is to traverse the computation graph in a column all the way to the end; it computes next layer's input for all batches of data given current layer's weight. This avoids weights reloading, but it's typically impossible to store KV cache and activation all in GPU memory.

To mitigate this, we adopt the zig-zag approach, a compromise between row and column traversal. It aimes to maximize GPU utilization and reduce data transfer overhead. In the zig-zag approach, we prioritize keeping the layer on the GPU for as long as possible to compute a particular input. This involves passing multiple batches through a single layer consecutively. Each batch is loaded onto the GPU, and computations are performed using the weights already residing on the GPU. While this approach increases throughput by processing multiple batches before transferring data, it comes at the cost of increased latency, as the entire layer must be processed before moving to the next layer.

By employing the zig-zag approach, we effectively trade off latency for throughput, optimizing the use of GPU resources and minimizing data transfer overhead in offloaded deep learning computations.

5.3 Scheduling overlapping

The other design option we find to be performance critical but didn't have time to include in our implemantion is overlapping. As elaborated in flexgen paper, the technique parallelizes the oading weights of the next layer, loading cache/activation of the next batch, storing cache/activation of the previous batch, and the computation of the current batch as there's no dependency between them. Algorithm 1 summarizes the technique. More details can be found in algorithm 1 in Flexgen (5). We found it critical to have the overlapping technique in our Section 6.

6 Evaluation

In this study, we conduct a comprehensive benchmarking analysis of model throughput across several frameworks: Huggingface Accelerate (baseline), FlexGen without offloading, FlexGen with offloading, vLLM with offloading, and vLLM with the addition of the zigzag forward pass technique. Our experiments are centered around the OPT-13B model, a significant and complex model widely used in natural language processing tasks. To execute these experiments, we utilize a robust hardware setup, including two RTX 3090 GPUs (PCIe 4.0) with 24GiB single GPU memory, an AMD 7800x3D with 128GB CPU memory, and a 4TB M.2 PCIe SSD.

One of the key challenges we address in this study is the size of the OPT-13B model when quantized to fp16, which amounts to 52GiB. This size exceeds the capacity of both GPUs combined, necessitating a strategic approach to model distribution and computation. To navigate this limitation, we explore the effectiveness of offloading techniques, particularly in scenarios where the entire model cannot fit into GPU memory simultaneously. Additionally, we investigate the impact of the zigzag forward pass approach in vLLM, which aims to optimize GPU utilization and reduce data transfer overhead.

Serving	Throughput (tokens/sec)
FlexGen	25.175
FlexGen w/o overlapping	22.801
Accelerate	16.950
vLLM with offloading	24.100
vLLM + zigzag	24.320

In our experimental evaluations, we observe interesting trends in the performance of different frameworks of another classic technique overlapping. FlexGen demonstrates superior performance over our solution when overlapping is considered. However, when overlapping is not enabled, vLLM with offloading outperforms FlexGen. Interestingly, the introduction of the zigzag forward pass technique in vLLM does not significantly impact performance in the absence of overlapping. This leads us to believe that overlapping is a crucial factor for the effectiveness of zigzagging.

The results suggest that the performance gains achieved by FlexGen are primarily attributed to its ability to overlap computation and communication, efficiently utilizing available resources. In contrast, vLLM with offloading leverages offloading techniques effectively, particularly when overlapping is not enabled in FlexGen.

7 Future works

7.1 Overlapping computation, loading and storing

Algorithm 1 Block Schedule with Overlapping

```
 1: for i = 1 to generation length do
 2:     for j = 1 to num layers do
 3:         for k = 1 to num GPU batches do
 4:             load weight(i, j + 1, k)         ▷ Load the weight of the next layer
 5:             store activation(i, j, k − 1)    ▷ Store activation of the previous batch
 6:             store cache(i, j, k − 1)         ▷ Store cache of the previous batch
 7:             load cache(i, j, k + 1)          ▷ Load the cache of the next batch
 8:             load activation(i, j, k + 1)     ▷ Load the activation of the next batch
 9:             compute(i, j, k)                 ▷ Compute this batch
10:             synchronize()                    ▷ Synchronize all devices
11:         end for
12:     end for
13: end for
```

As discussed in Section 6, overlapping saves performance of FlexGen, and suggests a strong potential to further improve our vLLM + zigzag performance.

7.2 Linear Programming to Find Optimal Offloading Strategy

A core contribution from Flexgen is that it proposes a linear programming-based search algorithm to optimize the throughput within the search space. It considers the capacity and peak usage of GPU, CPU, disk, and search for an optimal placement strategy of placing weight, KV cache, and activation on GPU, CPU and disk. We can potentially employ this technique with our vLLM implemenation. A wild idea is to take advantage of vLLM's blockwise management strategy, so that KV cache can be managed at finer granularity, and the vLLM's logical to physical table can naturally embed the information where the physical block is located.

8 Conclusion

Deploying Large Language Models (LLMs) on single GPU systems requires efficient resource management due to the limited availability and high demand of computational resources. Traditional methods struggle with balancing computational overhead and memory usage, particularly for managing model weights and key-value (KV) cache. To improve this, we adopt "Zigzag Scheduling" from the FlexGen framework within the vLLM architecture, which delays offloading model weights to secondary storage like CPU memory or disk. This approach allows multiple batches to be processed before offloading, reducing memory transfer frequency and minimizing I/O costs, thereby enhancing system efficiency. Zigzag Scheduling further optimizes hardware usage by overlapping the loading of weights for upcoming layers with the loading and storing of cache and activations for adjacent batches, significantly increasing throughput and reducing idle times.

References

[1] AI, C. Command r+ documentation, April 2024.

[2] META PLATFORMS, I. Llama-2-70b-chat-hf, 2023.

[3] RAJBHANDARI, S., RUWASE, O., RASLEY, J., SMITH, S., AND HE, Y. Zero-infinity: Breaking the GPU memory wall for extreme scale deep learning. *CoRR abs/2104.07857* (2021).

[4] REN, J., RAJBHANDARI, S., AMINABADI, R. Y., RUWASE, O., YANG, S., ZHANG, M., LI, D., AND HE, Y. Zero-offload: Democratizing billion-scale model training. *CoRR abs/2101.06840* (2021).

[5] 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* (2023), PMLR, pp. 31094–31116.

[6] TEAM, T. M. R. Introducing dbrx: A new state-of-the-art open llm, March 2024.