# Efficient LLM Inference with I/O-Aware Partial KV Cache Recomputation

**Chaoyi Jiang** [1] *, **Lei Gao** [1] *, **Hossein Entezari Zarch** [1], **Murali Annavaram** [1]

[1] University of Southern California
chaoyij@usc.edu, leig@usc.edu, entezari@usc.edu, annavara@usc.edu

## Abstract

Inference for Large Language Models (LLMs) is computationally demanding. To reduce the cost of auto-regressive decoding, Key-Value (KV) caching is used to store intermediate activations, enabling GPUs to perform only the incremental computation required for each new token. This approach significantly lowers the computational overhead for token generation. However, the memory required for KV caching grows rapidly, often exceeding the capacity of GPU memory. A cost-effective alternative is to offload KV cache to CPU memory, which alleviates GPU memory pressure but shifts the bottleneck to the limited bandwidth of the PCIe connection between the CPU and GPU. Existing methods attempt to address these issues by overlapping GPU computation with I/O or employing CPU-GPU heterogeneous execution, but they are hindered by excessive data movement and dependence on CPU capabilities. In this paper, we introduce an efficient CPU-GPU I/O-aware LLM inference method that avoids transferring the entire KV cache from CPU to GPU by recomputing partial KV cache from activations while concurrently transferring the remaining KV cache via PCIe bus. This approach overlaps GPU recomputation with data transfer to minimize idle GPU time and maximize inference performance. Our method is fully automated by integrating a profiler module that utilizes input characteristics and system hardware information, a scheduler module to optimize the distribution of computation and communication workloads, and a runtime module to efficiently execute the derived execution plan. Experimental results show that our method achieves up to 35.8% lower latency and 46.2% higher throughput during decoding compared to state-of-the-art approaches.

## 1 Introduction

Large language models (LLMs) have made remarkable progress in recent years, demonstrating their ability to power diverse applications such as natural language processing tasks like machine translation and summarization (Zhang et al. 2022; OpenAI et al. 2024), creative content generation including text and media (Chowdhery and et al. 2022; Anil and et al. 2024), and personalized recommendation systems tailored to individual users (Geng et al. 2022). Ensuring low latency is crucial for applications requiring real-time interaction, such as conversational agents and live translation services (Li et al. 2023; Hong et al. 2024), where delays can significantly affect user experience and utility. High throughput is equally important for supporting large-scale deployments, enabling these models to serve many concurrent users and process substantial volumes of data efficiently in enterprise and cloud-based environments (Kwon et al. 2023).

Key-Value (KV) cache is essential in auto-regressive decoding for LLMs, as it stores the intermediate key and value activations from earlier steps in the attention mechanism. This reduces the computational complexity of generating each token from quadratic to linear by eliminating the need to recompute these activations for every generated token. However, this comes at a cost: the size of the KV cache grows linearly with batch size, sequence length, and model size, leading to substantial memory demands (Wan et al. 2024; Shi et al. 2024).

GPU memory, while optimized for high-bandwidth access by computation units, is inherently limited and often insufficient to handle the large and growing size of the KV cache. One cost-effective approach to address this limitation is to offload the KV cache to CPU memory, and could be further offloaded to hard disks and network storage (Liu et al. 2024a). While offloading reduces GPU memory pressure, it introduces a new bottleneck: the slow PCIe bus becomes a limiting factor when transferring the KV cache from CPU to the GPU for computation. This not only impacts GPU utilization but also increases latency and reduces throughput, hindering the overall inference efficiency of the system (Zhao et al. 2024; Yu et al. 2024; He and Zhai 2024).

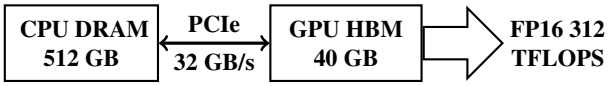



Figure 1: LLM inference system with a single A100 GPU.

| Model | Hidden Dim | KV Cache (MB) | PCIe Latency (ms) | Comp. Latency (us) |
|---|---|---|---|---|
| OPT-6.7B | 4,096 | 512 | 15.6 | 350.9 |
| OPT-13B | 5,120 | 640 | 19.5 | 438.8 |
| OPT-30B | 7,168 | 896 | 27.3 | 614.3 |

Table 1: PCIe latency and computation latency for different KV cache sizes based on the system in Figure 1.

*These authors contributed equally.
*Preprint*, under review.

To measure the communication hurdle we setup an LLM inference serving system (as shown in Figure 1) with an NVIDIA A100 GPU, where data transfer between the CPU DRAM and GPU HBM via the PCIe 4.0 x16 lane (32 GB/s bandwidth). Table 1 reports KV cache sizes, PCIe transfer latency from CPU to GPU, and GPU computation latency for various LLMs, assuming FP16 precision with a batch size of 32 and sequence length of 1,024. The results show that PCIe latency exceeds computation latency by over an order of magnitude, leading to significant GPU idle time and emphasizing the need to mitigate PCIe bandwidth constraints for efficient LLM inference.

To mitigate the issue of slow PCIe bandwidth, FlexGen (Sheng et al. 2023) and PipeSwitch (Bai et al. 2020) attempt to overlap GPU computation of the current layer with KV cache loading for the next layer. However, this approach is useful in setting where the data transfer time and GPU computation time are similar in latency, which is generally not the case with large batch and context sizes. FastDecode (He and Zhai 2024) suggests computing attention scores directly on the CPU, which has faster access to the KV cache compared to the GPU. Similarly, HeteGen (Zhao et al. 2024), TwinPilots (Yu et al. 2024), and (Park and Egger 2024) employ CPU-GPU heterogeneous execution to hide data transfer overhead by performing computations on the CPU. While these methods can improve performance, they depend on CPU resources that may not always be available due to competing workloads (Zhao et al. 2020).

In this paper, we propose a novel approach for efficient LLM inference that optimizes GPU computation and PCIe bandwidth utilization. Instead of transferring the entire KV cache from CPU to GPU to compute an attention score, we propose an alternative strategy: initially, the activations, which are smaller in size, required to generate part of the KV cache are sent to the GPU. The GPU then recomputes this partial KV cache from the input activations while the remaining KV cache over PCIe bus is transferred simultaneously. Our method ensures the computation of exact attention scores without approximation, while minimizing GPU idle time and improving overall latency and throughput.

Our method is fully automated in determining the recomputation and communication split. It includes a profiler module that collects system hardware information, a scheduler module that formulates the problem as a linear programming task to determine the optimal split point, and a runtime module that manages memory allocation on both devices and coordinates data transfer between them. Experimental results show significant improvements in inference latency or throughput, depending on the workload. In summary, our contributions are as follows:

- We propose an efficient CPU-GPU I/O-aware LLM inference method that leverages partial KV cache recomputation and asynchronous overlapping to address the system bottleneck of loading large KV cache.
- We develop a general framework and corresponding theoretical model that provides guidance for achieving optimal computation-communication workload distribution in parallel execution.
- Our experimental results show that our method outperforms the current state-of-the-art approaches up to 35.8% in terms of latency and 46.2% in terms of throughput.

# 2 Background

## 2.1 LLM Inference Process

The inference process of decoder-only LLMs employs an auto-regressive approach, generating tokens sequentially. It consists of two stages: the *prefilling stage* and the *decoding stage*. During the prefilling stage, the model processes the input prompt to compute and store KV pairs in the Multi-Head Attention (MHA) blocks, initializing the KV cache and generating the first output token. In the decoding stage, the model appends the KV pairs of the newly generated token into existing KV cache and generates the output tokens one by one with the KV cache. This process continues until a special `<End-Of-Sentence>` token is generated or the output reaches the maximum length.

## 2.2 Transformer-based LLM

During the *prefilling stage*, the input to the $i$-th decoder layer is denoted as $X^i \in \mathbb{R}^{b\times s\times h}$, where $i \in \{1,\dots,n\}$, $b$ is the batch size, $s$ is the sequence length, and $h$ is the input embedding dimension. The MHA block computes a set of queries ($Q$), keys ($K$), and values ($V$) through linear projections of $X^i$, as shown in Eq. (1):

$$Q^i = X^i \cdot W_Q^i, \quad K^i = X^i \cdot W_K^i, \quad V^i = X^i \cdot W_V^i, \quad (1)$$

where $W_Q^i, W_K^i, W_V^i \in \mathbb{R}^{h\times h}$ are the projection matrices. The self-attention score is computed as follows in Eq. (2):

$$Z^i = \text{Attention}(Q,K,V) = \text{softmax}\left(\frac{Q^i(K^i)^T}{\sqrt{d_{\text{head}}}}\right)\cdot V^i, \quad (2)$$

where $d_{\text{head}}$ represents the dimension of each attention head. Finally, the attention score is applied with a linear projection to produce the output of the MHA block, as shown in Eq. (3):

$$O^i = Z^i \cdot W_O^i, \quad (3)$$

where $W_O^i \in \mathbb{R}^{h\times h}$ is the projection matrix.

The feedforward network (FFN) is followed after the MHA block, which consists of two fully connected layers with a non-linear activation function applied between them. It processes the attention output $O^i$ to generate the input for the next decoder layer, as shown in Eq. (4):

$$X^{i+1} = \text{FFN}(O^i) = \sigma(O^i \cdot W_1^i)\cdot W_2^i, \quad (4)$$

where $W_1^i \in \mathbb{R}^{h\times d_{\text{FFN}}}$ and $W_2^i \in \mathbb{R}^{d_{\text{FFN}}\times h}$ are the weight matrices of the two linear layers, and $\sigma(\cdot)$ denotes the activation function.

In the *decoding stage*, the input to the $i$-th decoder layer is a single token $x^i \in \mathbb{R}^{b\times 1\times h}$. The KV cache is updated by concatenating the newly computed key and value pairs with the existing ones, as shown in Eq. (5):

$$K^i = \text{concat}(K^i, x^i \cdot W_K^i), \quad V^i = \text{concat}(V^i, x^i \cdot W_V^i). \quad (5)$$

The remaining attention and feedforward computations in the decoding stage are identical to those in the prefilling stage.

利用 activation 不是只能 recompute 当前 token 的 KV 吗?

# 3 Proposed Design

In LLM inference, scheduling strategies determine how computations are performed across batches and layers to optimize for specific performance goals, such as minimizing latency or maximizing throughput. In our design, we assume that the KV cache is stored on the CPU memory and is then fetched into GPU memory as needed. Row-by-row schedule processes (as shown in Appendix A.1) all tokens in a batch of sequences for one layer at a time before moving to the next layer. When minimizing latency is the primary goal, this approach is preferred as the KV cache is stored on CPU and is then loaded into GPU on a per-layer basis. In this scenario, model weights are kept in GPU memory whenever feasible. If the model weights are also offloaded to the CPU, both the KV cache and the model weights for a single layer are transferred to the GPU, processed for the current batch, and then cleared. This process is repeated layer by layer until a token is generated. With this approach, all prompts in a batch are fully processed to generate their complete context before proceeding to the next batch.

Column-by-column schedule (as shown in Appendix A.1) is more effective for maximizing throughput. This approach focuses on increasing the *effective batch size* (defined as the number of batches times the batch size) to process more sequences in parallel at the cost of longer latency. In this strategy, the model weights are offloaded to CPU memory to accommodate a large batch size. The model weights and KV cache for a single layer are transferred to GPU memory and processed for the first batch. Instead of moving to the next layer for the current batch, subsequent batches are processed using the same layer while keeping the weights stationary in GPU memory. Once a group of batches are processed for the first layer, the process moves to the second layer for each batch. Note that the effective batch size is limited by the available storage for activations and KV cache, as they must still be stored in CPU memory or external storage.

In both of the approaches described above, a substantial amount of KV cache must be transferred between the CPU and GPU. Our proposed design is independent of the scheduling strategy, whether row-by-row or column-by-column, and aims to overlap the majority of the PCIe transfer time with GPU computations, thereby improving overall efficiency.

## 3.1 Overview

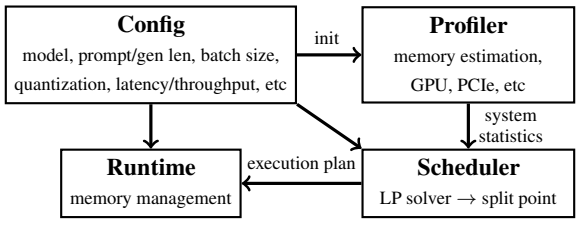



Figure 2: Design overview of our method.

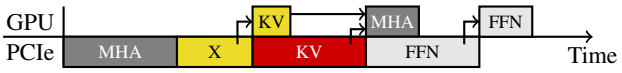
To relieve PCIe pressure and improve GPU computation utilization, we propose a novel method that recomputes partial KV cache on GPU while transferring the rest of KV cache to GPU. As shown in Figure 2, our method comprises three main modules: the profiler, scheduler and runtime. User configuration includes performance objective (i.e., latency or throughput), data parameters such as prompt length, generation length, batch size, and model information like input embedding dimension and number of layers. The **profiler module** gathers system statistics, which provides insights into hardware characteristics like PCIe bandwidth and GPU processing speed. Using this information along with the user configuration, the **scheduler module** calculates the best KV cache split point for recomputation by solving a linear programming problem, aiming to maximize the overlap between the computation and communication operations and utilization of both GPU and PCIe throughout the inference process. The **runtime module**, in turn, utilizes this execution strategy to process user inputs and manages the memory allocation and streams. This method ensures an efficient and system-aware execution plan tailored to the underlying hardware.

## 3.2 Scheduler Module

In this section, we describe how our approach is adopted to either the row-by-row or column-by-column schedule.

**Row-by-row schedule with partial KV cache recomputation.** If the performance objective is to minimize latency, the scheduler module will initiate a row-by-row execution plan. The naive offloading pipeline of a row-by-row schedule is shown in Figure 3(a), where both the KV cache and model weights are offloaded to CPU memory. The required data is transferred asynchronously over PCIe back to the GPU for executing the MHA and FFN blocks. Storing newly generated KV pairs to CPU memory is omitted from the figure for simplicity. Since the KV cache is larger in size compared to the MHA weights, it arrives at the GPU later during the asynchronous transfer. The pipeline is slightly different if model weights are not offloaded to CPU, in which case, only the MHA block will wait the KV cache data being transferred to GPU before starting the computation.


(a) Naive offloading pipeline for row-by-row schedule with asynchronous data transfer. GPU and PCIe denote GPU computation and data transfers, with arrows indicating data dependencies.




(b) Offloading pipeline for row-by-row schedule with partial KV cache recomputation to minimize latency.

Figure 3: Comparison of two offloading pipelines.

In our approach, rather than transferring the entire KV cache from CPU memory to GPU memory, the CPU transfers part of the activations (shown as $X$ on PCIe) required for KV cache recomputation on GPU. This allows the GPU to begin recomputing the corresponding KV activations

while the remaining KV cache is asynchronously transferred to the GPU, as illustrated in Figure 3(b). The GPU then merges the recomputed KV cache with the transferred KV cache to perform MHA computations.

**Determining the recomputation split using linear programming.** Given the current sequence length $s$, the activation transferred to the GPU in the $i$-th layer is represented by $X^i[0:l]$, where $0 \leq l \leq s$. The remaining KV cache for the subsequent tokens is denoted by $K^i[l:s]$ and $V^i[l:s]$. The memory consumption of these embeddings is given by

$$M_{X^i[0:l]} = b \times l \times h \times p, \tag{6}$$

$$M_{KV^i[l:s]} = 2 \times b \times (s-l) \times h \times p. \tag{7}$$

Recomputing the KV cache for $X^i[0:l]$ involves calculating

$$K^i[0:l] = X^i[0:l] \cdot W_K^i, \quad V^i[0:l] = X^i[0:l] \cdot W_V^i. \tag{8}$$

This recomputation on the GPU requires floating-point operations of

$$N_{KV^i[0:l]} = 4 \times b \times l \times h^2. \tag{9}$$

Consequently, the recomputation time $t^i_{gpu}$ for the KV cache is given by

$$t^i_{recomp} = \frac{N_{KV^i[0:l]}}{v_{gpu}}, \tag{10}$$

where $v_{gpu}$ denotes the GPU processing speed. The total time $t^i$ for processing can then be expressed as

$$t^i = \frac{M_{X^i[0:l]}}{v_{com}} + \max\left(t^i_{recomp}, \frac{M_{KV^i[l:s]}}{v_{com}}\right), \tag{11}$$

where $v_{com}$ represents the data transmission speed for activations and KV cache.

The objective is to determine the optimal $l$ that minimizes this total processing time $t^i$, which becomes a linear programming problem shown in Eq. (12).

$$\begin{aligned} \min_{l} \quad & t^i \\ \text{s.t.} \quad & 0 \leq l \leq s \quad \forall i \in \{1, \ldots, n\}. \end{aligned} \tag{12}$$

**Column-by-column schedule with partial KV cache recomputation.** When the performance objective is to maximize throughput, the scheduler module adopts a column-by-column execution plan. This approach, illustrated in Figure 4, accommodates large batch size inference by reusing model weights across multiple batches. As soon as the KV cache for batch 0 is fully transmitted, the activation for batch 1 is transferred. Simultaneously, the GPU begins computing the MHA for batch 0. Unlike row-by-row schedule, which processes each layer sequentially within a single batch before moving to the next layer, column-by-column schedule overlaps the transmission of KV cache and activations with the computation of MHA across multiple batches. The optimal split point, which determines the division of the KV cache between the portion recomputed on the GPU and the portion transferred from the CPU, can still be formulated as shown in Eq. 12.

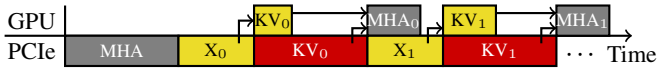



Figure 4: Offloading pipeline for column-by-column schedule with partial KV cache recomputation to maximize throughput.

### 3.3 Runtime Module

**Asynchronous overlapping.** To enable concurrent execution of GPU computation and CPU-GPU communication, the runtime module employs a communication parallelism strategy with six processes: weight loading, KV cache loading, activation loading, recomputed activation loading, KV cache storing, and activation storing, as detailed in Algorithm 1. By incorporating double buffering and prefetching techniques, it simultaneously loads weights for the next layer, retrieves activations for KV cache recomputation, and KV cache for the next batch, while storing cache and activations from the previous batch and processing the current batch.

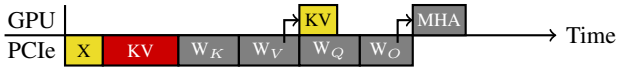
**Pinned memory.** To optimize data transfers, like prior works (Sheng et al. 2023; Yu et al. 2024), we utilize pinned CPU memory for recomputed activation and the weights that are transferred to the GPU. Using pinned memory enables faster and asynchronous transfer, as it avoids the need to page data in and out.

**Hiding partial KV cache recomputation.** If both the KV cache and model weights are offloaded, and the size of the transferred KV cache is smaller than the size of the model weights, a coarse-grained computation pipeline with partial KV cache recomputation may degrade inference performance. This occurs because partial KV recomputation waits until all MHA weights ($W_Q$, $W_K$, $W_V$, and $W_O$) are fully loaded, as shown in Figure 5(a), which delays the MHA computation. However, KV cache recomputation only requires $W_K$ and $W_V$ (Eq. 8), making it unnecessary to wait for the complete weight loading process. To address this, we implement a fine-grained MHA pipeline that prioritizes loading $W_K$ and $W_V$ first. Once these weights are available, KV cache recomputation can begin immediately. As illustrated in Figure 5(b), $W_K$ and $W_V$ are used for partial KV cache recomputation, followed by the use of $W_Q$ and $W_O$ for MHA computation. This approach effectively overlaps KV cache recomputation with weight loading, ensuring that in the worst-case scenario, the method performs no worse than the baseline bottlenecked by weights loading.

## 4 Experiments

**Hardware.** In our experiments, we utilize an NVIDIA A100 GPU with 40 GB of memory, connected to the CPU


(a) Coarse-grained offloading pipeline with delayed partial KV cache recomputation.




(b) Fine-grained offloading pipeline overlapping KV cache recomputation with weights loading.

Figure 5: Comparison of offloading pipelines with different levels of granularity in the MHA layer.

through a PCIe 4.0 x16 interface, which provides a bandwidth of 32 GB/s. The CPU is an AMD EPYC processor with 64 cores, operating at 2.6 GHz. Our method and implementation are hardware-agnostic, which allows for flexible deployment across diverse system architectures.

**Model.** We evaluate our proposed method using OPT models (Zhang et al. 2022) with parameter sizes ranging from 6.7 billion to 30 billion, alongside other baseline methods. While our experiments focus on OPT models, the recomputation technique presented in this work is compatible with other LLM architectures, such as LLaMa (Touvron et al. 2023) and GPT-3 (Brown 2020), due to their similar attention mechanisms (Vaswani et al. 2017). This compatibility also extends to models employing grouped-query attention (Touvron et al. 2023).

**Workload.** We evaluate our method on two types of workloads: latency-oriented and throughput-oriented. In the latency-oriented workload, the model weights are retained in GPU memory to avoid the costly repeated loading. Due to the limited memory size of a single GPU, experiments are conducted using OPT-6.7B and OPT-13B. In the throughput-oriented workload, model weights are offloaded to the CPU after computation to free more GPU memory for handling larger batches. This setup is evaluated using OPT-6.7B, OPT-13B, and OPT-30B.

We use synthetic datasets with prompts uniformly padded to the same length, with models configured to generate 32 or 128 tokens per prompt. To evaluate performance across different input scenarios, our evaluation uses prompt lengths of 256, 512, and 1024 tokens. Performance metrics include decoding latency (time taken to generate tokens) for latency-oriented workloads and decoding throughput (tokens generated per second) for throughput-oriented workloads, as our method does not impact prefilling performance.

**Baseline.** In our experiments, we use Hugging Face Transformers (v4.46.1) (Wolf et al. 2020) with Accelerate (Gugger et al. 2022) library as the baseline for latency-oriented workload experiments, as it currently supports KV cache offloading to CPU memory while still retaining the model weights in GPU memory. We use FlexGen (Sheng et al. 2023) as the baseline for throughput-oriented workload experiments, as it supports column-by-column schedule by offloading both model weights and KV cache to CPU.

**Implementation.** Our method is implemented on top of both Hugging Face Transformers and FlexGen (Sheng et al. 2023) frameworks to ensure a fair comparison with the baselines. In the Hugging Face implementation, we utilize double buffering in GPU memory to overlap the KV cache transfer across decoder layers. For both the Hugging Face and FlexGen implementations, we utilize CUDA streams to enable asynchronous overlapping as described in Algorithm 1.

## 4.1 Latency-oriented Workload Experiments

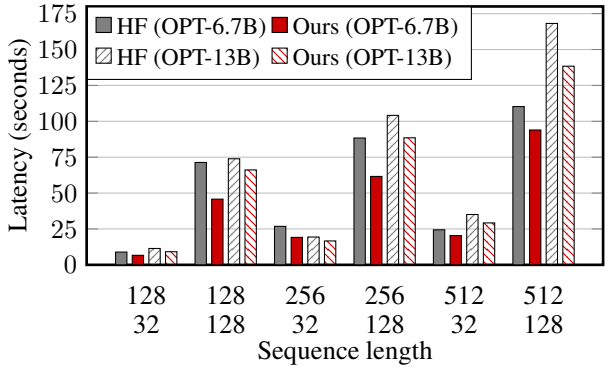



Figure 6: Decoding latency for a single batch of size 64 across different sequence lengths.

We evaluate the decoding latency required to complete a single batch for settings of different sequence lengths. Figure 6 shows that our approach consistently outperforms the baselines, Hugging Face Transformer with Accelerate library, for both OPT-6.7B and OPT-13B. The experimental results show that our method reduces decoding latency, especially at longer generation lengths. For instance, OPT 6.7B at a prompt length of 128 with 128 tokens generated, latency is reduced by approximately 35.8%. Detailed experiential results including KV cache size and GPU peak memory usage are provided in Appendix A.3.

## 4.2 Throughput-oriented Workload Experiments

We also evaluate throughput performance during the decoding stage, as our method does not affect the pre-filling stage. To maximize throughput, we set the effective batch size to be 32 by 8, meaning each layer computes on 8 batches of size 32 sequentially before moving to the next layer. The first row of Figure 7 shows the results, demonstrating that our method consistently outperforms FlexGen under settings of all sequence lengths for different models. It achieves up to 15.1%, 46.2%, and 29.0% speedup in throughput for OPT-6.7B, OPT-13B, and OPT-30B, respectively.

We also compare our method with FlexGen for varying batch sizes from 1 to 48 with a fixed prompt length of 1,024 and a generation length of 32, as shown in the second row of Figure 7. Our technique consistently outperforms FlexGen across all batch sizes. As the KV cache grows larger, our method shows greater performance benefits due to reduced KV cache transfer over the PCIe bus.

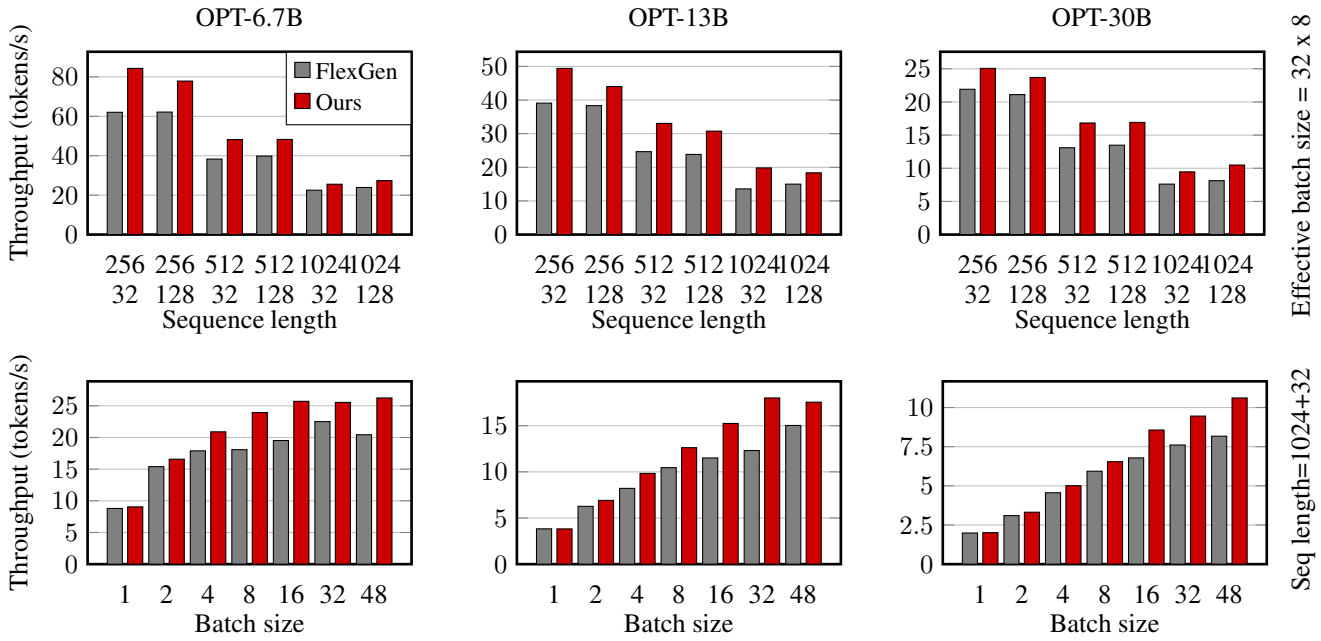



Figure 7: Throughput comparison for various models and configurations.

### 4.3 Ablation Study

**Hiding partial KV cache recomputation.** To evaluate the effectiveness of the fine-grained offloading pipeline that overlaps KV cache recomputation with weight loading, we conduct experiments using the OPT-6.7B model. In this ablation, we use a small KV cache size to ensure that MHA weights always arrive at the GPU later than the KV cache. Table 2 presents decoding latency across varying smaller batch sizes, comparing three configurations: FlexGen, our method without hiding KV cache recomputation, and our method with hiding. When the batch size is 1 and the KV cache size is the smallest, FlexGen can outperform our method without hiding. By overlapping the transfer of MHA weights with KV cache recomputation, our method ensures performance that is no worse than FlexGen under this scenario, particularly when weight loading is the primary bottleneck.

| Batch size | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| KV cache (MB) | 3 | 6 | 12 | 24 | 48 | 64 |
| FlexGen | 1.761 | 3.488 | 6.646 | 12.826 | 23.795 | 41.210 |
| Ours (w/o. hiding KV recomputation) | 1.749 | 3.461 | **6.766** | 12.930 | 23.613 | 43.462 |
| Ours (w. hiding KV recomputation) | **1.774** | **3.586** | 6.696 | **12.986** | **24.557** | **43.945** |

Table 2: OPT-6.7B model with prompt and generation lengths of 256 and 64, respectively. Each MHA block ($W_Q$, $W_K$, $W_V$, and $W_O$) requires 128 MB of memory.

**KV cache compression.** We apply group-wise 4-bit quantization to compress the KV cache, which has been shown to have minimal impact on model accuracy (Sheng et al. 2023). Figure 8 shows that applying compression reduces the amount of data transferred to the GPU, leading to further improvements in decoding throughput. These results showcase the compatibility of our method with KV cache compression and its potential to achieve additional performance gains by alleviating PCIe bandwidth bottlenecks.

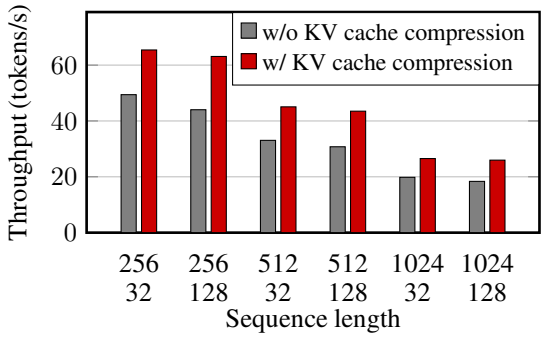



Figure 8: Decoding throughput improvement with KV cache compression enabled on OPT-13B model.

**Runtime breakdown.** Figure 9 presents the runtime breakdown of our method and FlexGen. Our method achieves a substantial reduction in KV cache transfer time, decreasing it from 58% to 38%, with activation transfer contributing only 8% of the total runtime. By recomputing the partial KV cache from the transferred activations, GPU computation time increases from 2.3% to 13.3%. This demonstrates that our method effectively overlaps GPU computation with CPU-GPU communication, substantially reducing the data transfer volume from CPU to GPU and alleviating the PCIe bottleneck that limits LLM inference performance.

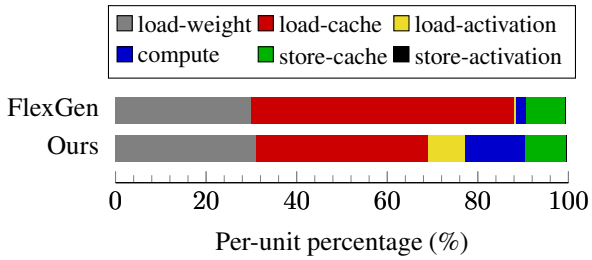



Figure 9: Runtime breakdown of our method and FlexGen.

**GPU utilization.** To evaluate the efficiency improvement, we analyze the temporal resource utilization of our technique and FlexGen as shown in Figure 10. At first in the prefilling stage, both methods reach full GPU utilization since prefilling stage is compute-bound. However, in the decoding stage, in contrast to FlexGen, our method enhances GPU utilization, increasing it from 85% to 99% on average by overlapping GPU computations with CPU-GPU data transfers, while maintaining the same peak memory usage indicated by the black lines.

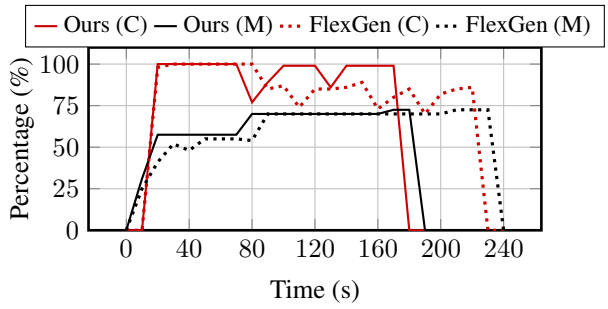



Figure 10: Comparison of resource usage during inference of a single layer. Our method achieves higher GPU utilization across the entire pipeline compared to FlexGen.

## 5 Related Works

**GPU-efficient LLM inference.** Maximizing GPU utilization is crucial for serving LLMs efficiently to achieve low latency and high throughput. Orca (Yu et al. 2022) employs iteration-level scheduling to handle batches with varying output lengths, returning completed sequences immediately to serve new ones. PagedAttention (Kwon et al. 2023) observes that the KV cache grows and shrinks dynamically as tokens are generated, though the sequence lifetime and length are not predetermined. It addresses this by managing the KV cache as non-contiguous memory blocks. FlashAttention (Dao et al. 2022) combines attention operations into a single kernel and tiles QKV matrices into smaller blocks to optimize GPU SRAM usage and reduce HBM access overhead. DeepSpeed-Inference (Aminabadi et al. 2022) enhances multi-GPU inference for both dense and sparse Transformer models by combining GPU memory and employing a hybrid inference technique with CPU and NVMe memory. Flash-Decoding (Dao et al. 2023) accelerates long-context inference by splitting keys and values into smaller chunks, enabling parallel attention computations and combining results for the final output.

**Offloading systems for LLM inference.** To address the memory demands of LLMs in resource-constrained settings, offloading techniques aim to minimize the latency of data transfers between CPUs and GPUs. FlexGen (Sheng et al. 2023) proposes to offload weights, activations, and KV cache to CPU memory or external storage and maximizes throughput for lager batch size by formularizing the optimization as a graph traversal problem. HeteGen (Zhao et al. 2024) uses the CPU for partial computation on offloaded weights while transferring the remaining workload to the GPU. TwinPilots (Yu et al. 2024) further optimizes workload balancing between the CPU and GPU at the operator level. FastDecode (He and Zhai 2024) reduces KV cache data movement by offloading the KV cache and attention computation entirely to the CPU. Park and Egger and Neo (Jiang et al. 2024) overlaps GPU linear projection computations with CPU-based attention computations across multiple batches to improve resource utilization. Our method is orthogonal to these CPU-assisted approaches, as it focuses on optimizing GPU utilization and data transfer efficiency without relying on additional CPU resources. Furthermore, it can be integrated on top of these methods to further enhance overall system performance.

**KV cache optimization.** Efficient KV cache management enhances inference performance through compression or eviction strategies. KIVI (Liu et al. 2024b) introduces a tuning-free 2-bit quantization method to compress key cache per channel and value cache per token. Similarly, KVQuant (Hooper et al. 2024) applies 3-bit compression by combining per-channel quantization with pre-rotary positional embedding quantization for LLaMA. For eviction, H2O (Zhang et al. 2023) formulates KV cache eviction as a dynamic submodular problem, prioritizing critical and recent tokens to improve throughput. StreamingLLM (Xiao et al. 2023) uses window attention with a fixed-size sliding window to retain the most recent KV caches, maintaining constant memory usage and decoding speed once the cache reaches capacity. InfiniGen (Lee et al. 2024) stores low-rank key cache in GPU memory, offloads value cache to the CPU, and selectively retrieves important values based on approximate attention scores.

## 6 Conclusion

In this paper, we introduce an efficient CPU-GPU I/O-aware LLM inference method designed to accelerate KV cache loading. Our approach minimizes the data transfer between the CPU and GPU by leveraging partial KV cache recomputation. By overlapping this recomputation with data transmission, our method significantly reduces idle GPU time and enhances overall inference performance. Future work could extend our method to tolerate KV cache loading from remote network storage or scale to large multi-GPU infrastructure, further enhancing its applicability and performance in diverse deployment scenarios.

## References


Aminabadi, R. Y.; Rajbhandari, S.; Awan, A. A.; Li, C.; Li, D.; Zheng, E.; Ruwase, O.; Smith, S.; Zhang, M.; Rasley, J.; et al. 2022. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–15. IEEE.

Anil, R.; and et al. 2024. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805.

Bai, Z.; Zhang, Z.; Zhu, Y.; and Jin, X. 2020. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 499–514.

Brown, T. B. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Chowdhery, A.; and et al. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:2204.02311.

Dao, T.; Fu, D.; Ermon, S.; Rudra, A.; and Ré, C. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35: 16344–16359.

Dao, T.; Haziza, D.; Massa, F.; and Sizov, G. 2023. Flash Decoding: Advances in Efficient Text Generation.

Geng, S.; Liu, S.; Fu, Z.; Ge, Y.; and Zhang, Y. 2022. Recommendation as Language Processing (RLP): A Unified Pretrain, Personalized Prompt & Predict Paradigm (P5). In *RecSys '22*. Association for Computing Machinery.

Gugger, S.; Debut, L.; Wolf, T.; Schmid, P.; Mueller, Z.; Mangrulkar, S.; Sun, M.; and Bossan, B. 2022. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate.

He, J.; and Zhai, J. 2024. FastDecode: High-Throughput GPU-Efficient LLM Serving using Heterogeneous Pipelines. *arXiv preprint arXiv:2403.11421*.

Hong, S.; Zhuge, M.; Chen, J.; Zheng, X.; Cheng, Y.; Wang, J.; Zhang, C.; Wang, Z.; Yau, S. K. S.; Lin, Z.; Zhou, L.; Ran, C.; Xiao, L.; Wu, C.; and Schmidhuber, J. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations*.

Hooper, C.; Kim, S.; Mohammadzadeh, H.; Mahoney, M. W.; Shao, Y. S.; Keutzer, K.; and Gholami, A. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*.

Jiang, X.; Zhou, Y.; Cao, S.; Stoica, I.; and Yu, M. 2024. NEO: Saving GPU Memory Crisis with CPU Offloading for Online LLM Inference. arXiv:2411.01142.

Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C. H.; Gonzalez, J.; Zhang, H.; and Stoica, I. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 611–626.

Lee, W.; Lee, J.; Seo, J.; and Sim, J. 2024. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 155–172.

Li, G.; Hammoud, H. A. A. K.; Itani, H.; Khizbullin, D.; and Ghanem, B. 2023. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. arXiv:2303.17760.

Liu, Y.; Li, H.; Cheng, Y.; Ray, S.; Huang, Y.; Zhang, Q.; Du, K.; Yao, J.; Lu, S.; Ananthanarayanan, G.; Maire, M.; Hoffmann, H.; Holtzman, A.; and Jiang, J. 2024a. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*. Association for Computing Machinery.

Liu, Z.; Yuan, J.; Jin, H.; Zhong, S.; Xu, Z.; Braverman, V.; Chen, B.; and Hu, X. 2024b. KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache. In *Proceedings of the 41st International Conference on Machine Learning*.

OpenAI; Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; and et al. 2024. GPT-4 Technical Report. arXiv:2303.08774.

Park, D.; and Egger, B. 2024. Improving Throughput-oriented LLM Inference with CPU Computations. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, 233–245.

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

Shi, L.; Zhang, H.; Yao, Y.; Li, Z.; and Zhao, H. 2024. Keep the Cost Down: A Review on Methods to Optimize LLM' s KV-Cache Consumption. arXiv:2407.18003.

Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.-A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention Is All You Need. arXiv:1706.03762.

Wan, Z.; Wang, X.; Liu, C.; Alam, S.; Zheng, Y.; Liu, J.; Qu, Z.; Yan, S.; Zhu, Y.; Zhang, Q.; Chowdhury, M.; and Zhang, M. 2024. Efficient Large Language Models: A Survey. *Transactions on Machine Learning Research*.

Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; Davison, J.; Shleifer, S.; von Platen, P.; Ma, C.; Jernite, Y.; Plu, J.; Xu, C.; Scao, T. L.; Gugger, S.; Drame, M.; Lhoest, Q.; and Rush, A. M. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics.

Xiao, G.; Tian, Y.; Chen, B.; Han, S.; and Lewis, M. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*.

Yu, C.; Wang, T.; Shao, Z.; Zhu, L.; Zhou, X.; and Jiang, S. 2024. TwinPilots: A New Computing Paradigm for GPU-CPU Parallel LLM Inference. In *Proceedings of the 17th ACM International Systems and Storage Conference*, 91–103.

Yu, G.-I.; Jeong, J. S.; Kim, G.-W.; Kim, S.; and Chun, B.-G. 2022. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 521–538.

Zhang, S.; Roller, S.; Goyal, N.; Artetxe, M.; Chen, M.; Chen, S.; Dewan, C.; Diab, M.; Li, X.; Lin, X. V.; et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*.

Zhang, Z.; Sheng, Y.; Zhou, T.; Chen, T.; Zheng, L.; Cai, R.; Song, Z.; Tian, Y.; Ré, C.; Barrett, C.; et al. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36: 34661–34710.

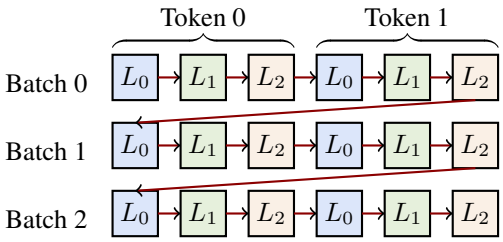
Zhao, H.; Cui, W.; Chen, Q.; Leng, J.; Yu, K.; Zeng, D.; Li, C.; and Guo, M. 2020. CODA: Improving Resource Utilization by Slimming and Co-locating DNN and CPU Jobs. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*.

Zhao, X.; Jia, B.; Zhou, H.; Liu, Z.; Cheng, S.; and You, Y. 2024. HeteGen: Efficient Heterogeneous Parallel Inference for Large Language Models on Resource-Constrained Devices. *Proceedings of Machine Learning and Systems*, 6: 162–172.

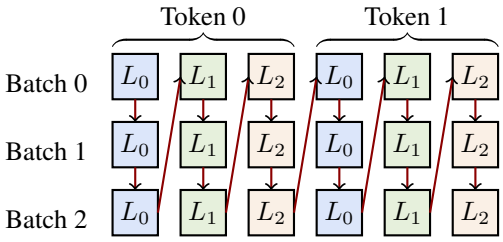
# A Appendix

## A.1 Schedule Methods

Figures 11 illustrates two decoding schedules for generating 2 tokens from a model with three layers ($L_0$, $L_1$, and $L_2$) during the decoding stage. In Figure 11(a), the row-by-row schedule processes each batch across all layers before moving to the next batch. In contrast, Figure 11(b) shows the column-by-column schedule, where each layer is reused to process a group of batches before moving to the next layer.




(a) Row-by-row schedule.




(b) Column-by-column schedule.

Figure 11: Two different schedules, with arrows indicating the scheduling order.

## A.2 Partial KV Cache Recomputation with Overlapping

Built on FlexGen's computation and communication overlapping technique, we adapt it to support partial KV cache recomputation. Algorithm 1 enables simultaneous execution of tasks within the innermost loop, including loading weights for the next layer, loading activations for KV cache recomputation, recomputing the partial KV cache, loading cache and activations for the next batch, storing cache and activations for the previous batch, and performing computation for the current batch. Although the algorithm is designed for column-by-column scheduling, the row-by-row schedule with a single batch is a special case of it.

Algorithm 1: Partial KV Cache Recomputation with Overlapping

**for** $i = 1$ **to** `generation_length` **do**
  **for** $j = 1$ **to** `num_layers` **do**
    **for** $k = 1$ **to** `num_GPU_batches` **do**
      // Load the weight of the next layer
      `load_weight`$(i, j+1, k)$
      // Load the activation for KV cache recomputation of the next batch
      `load_activation_recompute`$(i, j, k+1)$
      // Load the KV cache and activation of the next batch
      `load_cache`$(i, j, k+1)$
      `load_activation`$(i, j, k+1)$
      // Compute this batch
      `compute`$(i, j, k)$
      // Store the KV cache and activation of the previous batch
      `store_activation`$(i, j, k-1)$
      `store_cache`$(i, j, k-1)$
      // Synchronize all devices
      `synchronize`$()$
    **end for**
  **end for**
**end for**

## A.3 Detailed Experimental Results

Table 3 and 4 present detailed experimental results for latency-oriented workloads using OPT-6.7B and OPT-13B. The results show the performance differences between our method and the baseline (Hugging Face Transformer library with KV cache offloading) in terms of GPU peak memory, decode latency, and throughput across various configurations. Notably, our method consistently achieves lower latency while maintaining comparable memory usage.

| Method | Batch size | Prompt length | Generation length | Cache size (GB) | GPU peak mem (GB) | Decode latency (sec) | Decode throughput (tokens/s) |
|---|---|---|---|---|---|---|---|
| HF | 64 | 128 | 32 | 5.0 | 14.427 | 8.905 | 222.788 |
| | 64 | 128 | 128 | 8.0 | 14.708 | 71.327 | 113.954 |
| | 64 | 256 | 32 | 9.0 | 16.337 | 26.825 | 73.961 |
| | 64 | 256 | 128 | 12.0 | 16.618 | 88.354 | 91.993 |
| | 64 | 512 | 32 | 17.0 | 20.154 | 24.390 | 81.344 |
| | 64 | 512 | 128 | 20.0 | 20.576 | 110.277 | 73.705 |
| Ours | 64 | 128 | 32 | 5.0 | 14.364 | 6.651 | 298.284 |
| | 64 | 128 | 128 | 8.0 | 14.645 | 45.766 | 177.598 |
| | 64 | 256 | 32 | 9.0 | 16.212 | 19.138 | 103.666 |
| | 64 | 256 | 128 | 12.0 | 16.493 | 61.597 | 131.955 |
| | 64 | 512 | 32 | 17.0 | 19.904 | 20.349 | 97.501 |
| | 64 | 512 | 128 | 20.0 | 20.951 | 93.932 | 86.531 |

Table 3: Detailed experimental results for OPT-6.7B corresponding to Figure 6.

| Method | Batch size | Prompt length | Generation length | Cache size (GB) | GPU peak mem (GB) | Decode latency (sec) | Decode throughput (tokens/s) |
|---|---|---|---|---|---|---|---|
| HF | 64 | 128 | 32 | 7.812 | 26.083 | 11.409 | 173.891 |
| | 64 | 128 | 128 | 12.500 | 26.434 | 73.896 | 109.993 |
| | 64 | 256 | 32 | 14.062 | 28.087 | 19.381 | 102.368 |
| | 64 | 256 | 128 | 18.750 | 28.439 | 104.115 | 78.068 |
| | 64 | 512 | 32 | 26.562 | 32.851 | 35.066 | 56.579 |
| | 64 | 512 | 128 | 31.250 | 34.146 | 168.155 | 48.336 |
| Ours | 64 | 128 | 32 | 7.812 | 26.005 | 9.148 | 216.867 |
| | 64 | 128 | 128 | 12.500 | 26.356 | 66.119 | 122.929 |
| | 64 | 256 | 32 | 14.062 | 27.931 | 16.654 | 119.127 |
| | 64 | 256 | 128 | 18.750 | 28.337 | 88.492 | 91.850 |
| | 64 | 512 | 32 | 26.562 | 33.203 | 29.215 | 67.911 |
| | 64 | 512 | 128 | 31.250 | 34.615 | 138.377 | 58.738 |

Table 4: Detailed experimental results for OPT-13B corresponding to Figure 6.