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

Anonymous ACL submission

Abstract

Inference for Large Language Models (LLMs) is computationally demanding. To reduce the cost of auto-regressive decoding, Key-Value (KV) cache is used to store intermediate activations, which significantly lowers the computational overhead for token generation. However, the memory required for KV cache 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. Fully overlapping PCIe communication latency gets challenging as the size of KV cache grows and/or the GPU compute capabilities increase. In this paper, we introduce KVPR, an efficient I/O-aware LLM inference method where the CPU first transfers a partial set of activations from which the GPU can start recomputing the KV cache values. While the GPU recomputes the partial KV cache the remaining portion of the KV cache is concurrently transferred from the CPU. This approach overlaps GPU recomputation with KV cache transfer to minimize idle GPU time and maximize inference performance. KVPR 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 KVPR achieves up to 35.8% lower latency and 46.2% higher throughput during decoding compared to state-of-the-art approaches. The code is available at https://anonymous.4open.science/r/KVPR-C176/.

1 Introduction

Large language models (LLMs) have made remarkable progress in recent years, demonstrating their ability to power diverse applications such as machine translation (Zhu et al., 2024), summarization (OpenAI et al., 2024), creative content generation (Gemini Team, 2024), and personalized recommendations (Geng et al., 2022). Real-time applications, including conversational agents and live translation (Li et al., 2023), depend on low latency to provide seamless user interaction, while large-scale deployments require high throughput to support concurrent users and process substantial data efficiently (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).

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 cheaper and plentiful 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. Due to the long PCIe transfer time the overall decoding latency increases and while the token throughput decreases, hindering the overall inference effi-

ciency of the system (Zhao et al., 2024a).

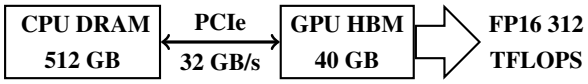


Figure 1: LLM inference system with an A100 GPU.

Model	Hidden Dim	KV Cache (MB)	PCIe Latency (ms)	Comp. Latency (ms)
OPT-6.7B	4,096	512	15.6	0.3509
OPT-13B	5,120	640	19.5	0.4388
OPT-30B	7,168	896	27.3	0.6143

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

To evaluate the impact of communication overhead, we set up an LLM inference serving system (shown in Figure 1) using an NVIDIA A100 GPU. Data transfer between the CPU DRAM and GPU HBM occurs over a PCIe 4.0 16 lanes with a bandwidth of 32 GB/s. Table 1 shows the hidden dimension, KV cache size, PCIe transfer time, and GPU computation latency for KV pair computation. We use FP16 precision with a batch size of 32 and a sequence length of 1024. The results show that PCIe latency exceeds computation latency by over an order of magnitude. The long transfer time leads to large GPU idle time which is detrimental to inference efficiency.

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, the effectiveness of such an overlap is capped by the task that takes the longest time. Hence, PCIe transfer time overshadows GPU computation latency, which is generally 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., 2024a), TwinPilots (Yu et al., 2024), and Park and Egger employ CPU-GPU heterogeneous execution to hide data transfer overhead by performing computations on the CPU. However, as demonstrate later in our results such an approach puts burden on the CPU to satisfy the KV cache computation demands from multiple GPUs attached to a CPU host, thereby limiting scalability.

In this paper, we propose KVPR, a novel approach for efficient LLM inference that balances the GPU computation and PCIe bandwidth tradeoffs. Instead of transferring the entire KV cache from CPU to GPU to compute an attention score, the CPU transfers a partial set of activations, which are smaller in size and are required to generate part of the KV cache, to the GPU. The GPU then starts recomputing the partial KV cache from the input activations. Concurrently the CPU transfers the remaining KV cache over PCIe. KVPR ensures the computation of exact attention scores without approximation, while minimizing GPU idle time and improving overall latency and throughput.

One may achieve a *perfect* overlap of PCIe transfer time and GPU recomputation time by determining the optimal fraction of activations that need to be recomputed. KVPR 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 problem 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 workload. In summary, our contributions are as follows:

- We propose an efficient CPU-GPU I/O-aware LLM inference method that leverages KV cache partial recomputation with asynchronous KV cache transfer that overlaps compute and communication to address the system bottleneck of loading large KV cache from CPU memories.
- We develop a framework based on linear integer programming that achieves optimal computation-communication distribution.
- Our experimental results show that KVPR 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

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**. In 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 $i \in \{1, \ldots, n\}$, b is the batch size, s is the prompt length, and h is the input embedding dimension. The Multi-Head Attention (MHA) block computes a set of queries (Q), keys (K), and values

(V) through linear projections of X^i:

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

where $W_Q^i, W_K^i, W_V^i \in \mathbb{R}^{h \times h}$ are the projection matrices. The generated K^i and V^i are stored in the KV cache.

The self-attention score in MHA is computed as:

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

where d_{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:

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

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 follows:

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

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 i-th decoder layer receives 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:

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

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

3 Proposed Method

LLM inference scheduling. Our approach aims at LLM inference systems with large KV caches that are stored on CPU DRAM and fetched into GPU memory as needed. Since LLMs have many layers and many batches of inputs to process there are different scheduling strategies to determine how computations are performed across batches and layers to optimize for specific performance goals, such as minimizing latency or maximizing throughput. Row-by-row schedule (as shown in Appendix A.1) processes a batch of sequences for one layer at a time before moving to the next layer. 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. When minimizing latency is the primary goal, this approach is preferred, because all prompts in a batch are fully processed to generate their complete context before proceeding to the next batch.

Column-by-column scheduling (Appendix A.1) is more effective for maximizing throughput by increasing the *effective batch size* (number of batches times batch size) to process more sequences in parallel, at the cost of longer latency. In this approach, 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 once they exceed the GPU memory capacity.

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 Design Overview

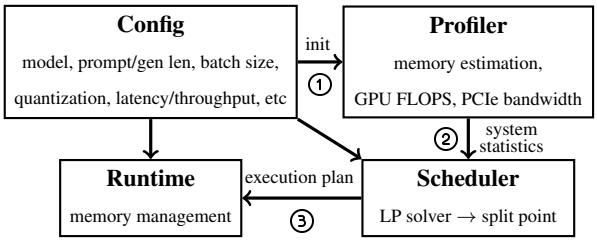


Figure 2: Design overview of KVPR.

To relieve PCIe pressure and improve GPU computation utilization, we propose a novel method, KVPR, that recomputes partial KV cache on GPU while transferring the rest of KV cache to GPU. As shown in Figure 2, KVPR 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 bandwidth during the inference process. The **runtime module**, in turn, utilizes this execution strategy to process user inputs and manages the memory allocation and data transfer streams.

3.2 Scheduler Module

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

Row-by-row schedule with KV cache partial 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 are transferred asynchronously over PCIe 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 this case, only the MHA block will wait the KV cache data being transferred to GPU before starting the computation.

In KVPR, rather than transferring the entire KV cache from CPU memory to GPU memory, the GPU recomputes partial KV cache using corresponding input activations that are transferred from CPU first 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.

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

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

Figure 3: Comparison of two offloading pipelines.

Column-by-column schedule with KV cache partial 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 activations for batch 1 is transferred. Simultaneously, the GPU begins computing the MHA for batch 0. Unlike the row-by-row schedule, which processes all layers sequentially within a single batch before moving to the next batch, the column-by-column schedule processes multiple batches on the same layer. As a result, activations corresponding to the recomputed KV cache must be stored until generation for that batch is complete.

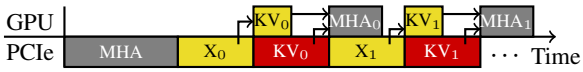


Figure 4: Offloading pipeline for column-by-column schedule with KV cache partial recomputation to maximize throughput.

Determining the optimal KV cache split point. In both scheduling methods, the objective is to identify the optimal split point, which defines the division of the KV cache between the portion recomputed on the GPU and the portion transferred from CPU memory. This problem can be formulated as a linear programming problem. The row-by-row schedule can be viewed as a special case of the column-by-column schedule, where activations for recomputing the KV cache are not transferred. We first formulate the problem for the column-by-column schedule and then demonstrate how it simplifies to the row-by-row schedule.

Given the current sequence length s', which is greater than the prompt 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 remain-

ing KV cache for the subsequent tokens is denoted by $K^i[l:s']$ and $V^i[l:s']$. The memory usage of these activations is:

$$\begin{aligned} M_{X^i[0:l]} &= b \times l \times h \times p, \\ M_{KV^i[l:s']} &= 2 \times b \times (s' - l) \times h \times p. \end{aligned} \tag{6}$$

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

$$\begin{aligned} K^i[0:l] &= X^i[0:l] \cdot W_K^i, \\ V^i[0:l] &= X^i[0:l] \cdot W_V^i. \end{aligned} \tag{7}$$

This recomputation on the GPU requires floating-point operations of

$$N_{KV^i[0:l]} = 4 \times b \times l \times h^2. \tag{8}$$

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

$$t^i_{recomp} = \frac{N_{KV^i[0:l]}}{v_{gpu}}, \tag{9}$$

where v_{gpu} denotes the GPU processing speed. The total time t^i for processing is as follows:

$$t^i = \frac{M_{X^i[0:l]}}{v_{com}} + \max\left(t^i_{recomp}, \frac{M_{KV^i[l:s']}}{v_{com}}\right), \tag{10}$$

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:

$$\begin{aligned} \min_{l} \quad & t^i \\ \text{s.t.} \quad & 0 \le l \le s \quad \forall i \in \{1, \ldots, n\}. \end{aligned} \tag{11}$$

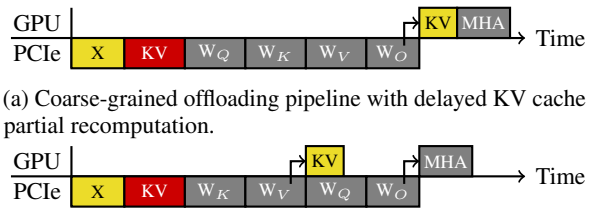
The optimal split point l depends on the current sequence length s', which increases during generation and must therefore be determined adaptively. Fortunately, solving this linear programming problem is computationally negligible because there is only one integer variable. If the first term in Eq. (10) is omitted, the problem simplifies to the row-by-row schedule.

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 Appendix A.2. By incorporating double buffering and prefetching techniques, it simultaneously loads weights for the next layer, and 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 transfer, 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 KV cache partial 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 KV cache partial recomputation may degrade inference performance. This occurs because 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. (7)), 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 KV cache partial 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.



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

4 Experiments

Hardware. In our experiments, we utilize an NVIDIA A100 GPU with 40 GB of memory, connected to the CPU 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 KVPR using OPT models (Zhang et al., 2022) with parameter sizes ranging from 6.7B to 30B. 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 et al., 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 KVPR 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 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.

To provide accurate comparisons we use the same datasets as those in FlexGen (Sheng et al., 2023) 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 KVPR does not impact prefilling performance. We report an average decoding latency and throughput across five test runs, respectively.

Baseline. In our experiments, we use DeepSpeed Inference (Aminabadi et al., 2022), Hugging Face Accelerate (Gugger et al., 2022) as the baseline for latency-oriented workload experiments, as Hugging Face Transformers library 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. KVPR is implemented on top of Hugging Face Transformers (v4.46.1) (Wolf et al., 2020) and FlexGen (Sheng et al., 2023) frameworks to ensure fair comparison with baselines. In the Transformers implementation, we utilize double buffering in GPU memory to overlap KV cache transfer across decoder layers. For both the Transformers and FlexGen implementations, we utilize CUDA streams to enable asynchronous overlapping as described in Algorithm 1.

4.1 Latency-oriented Experiments

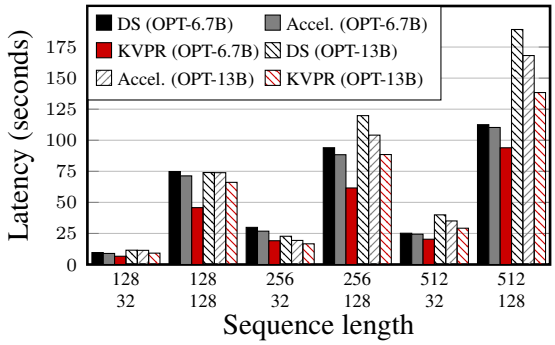


Figure 6: Decoding latency for a single batch of size 64 across different sequence lengths.

We evaluate the decoding latency required to complete a single batch for settings of different sequence lengths. Figure 6 shows that KVPR consistently outperforms the baselines, DeepSpeed Inference and Hugging Face Accelerate, for both OPT-6.7B and OPT-13B. The experimental results show that KVPR 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% compared to Hugging Face Accelerate. Detailed experiential results including KV cache size, GPU peak memory usage and optimal recomputation split points over the generation process are provided in Appendix A.3 and A.4.

4.2 Throughput-oriented Experiments

We also evaluate throughput performance during the decoding stage, as KVPR does not affect the prefilling 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 KVPR consistently outperforms FlexGen under settings of all sequence lengths for different models. It achieves up to 15.1%, 46.2%, and 29.0%

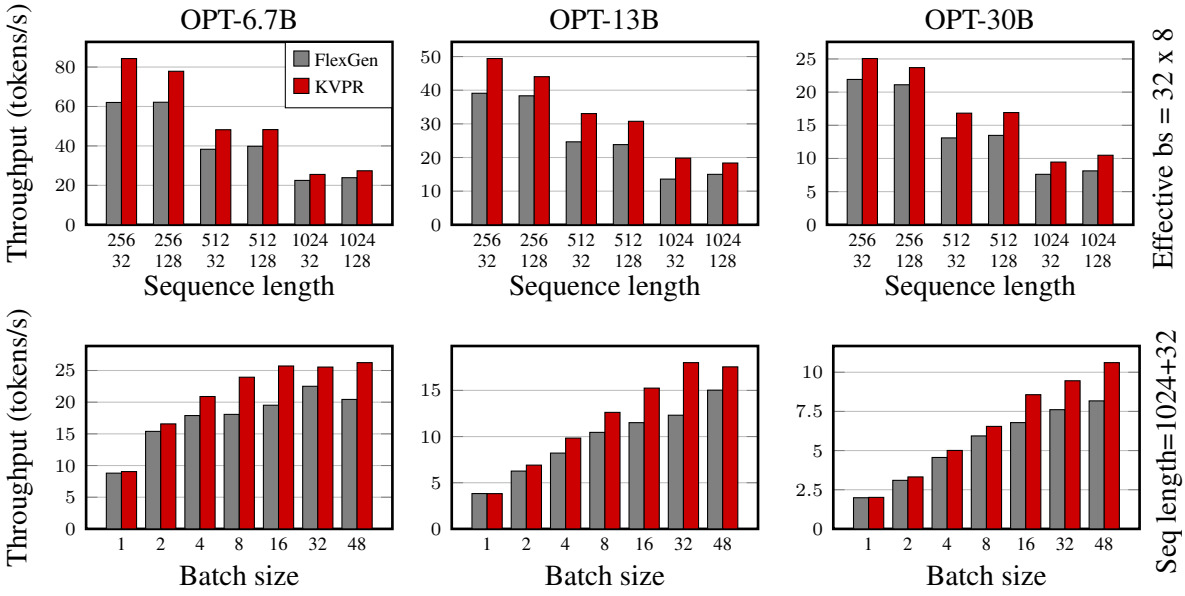


Figure 7: Throughput comparison for various models and configurations.

speedup in throughput for OPT-6.7B, OPT-13B, and OPT-30B, respectively. Additional experimental results on a low-end GPU system are provided in Appendix A.5.

We also compare KVPR 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. KVPR consistently outperforms FlexGen across all batch sizes. As the KV cache grows larger, KVPR shows greater performance benefits due to reduced KV cache transfer over the PCIe bus.

4.3 GPU Utilization

To evaluate the efficiency improvement, we analyze the temporal resource utilization of KVPR and FlexGen as shown in Figure 8. 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, KVPR enhances GPU utilization, increasing it from 85% to 99% on average by overlapping GPU computations with CPU-GPU data transfer, while maintaining the same peak memory usage indicated by the black lines.

4.4 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 9 shows that applying compression reduces the amount of data transferred to the

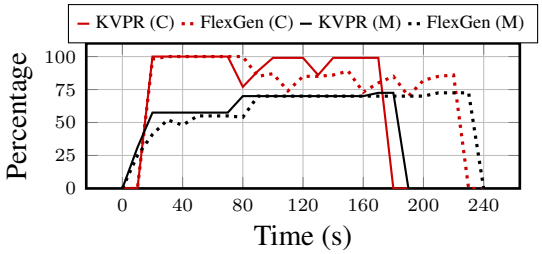


Figure 8: Computation and memory resource usage of KVPR and FlexGen during decoding stage.

GPU, leading to further improvements in decoding throughput. These results showcase the compatibility of KVPR with KV cache compression and its potential to achieve additional performance gains by alleviating PCIe bandwidth bottlenecks.

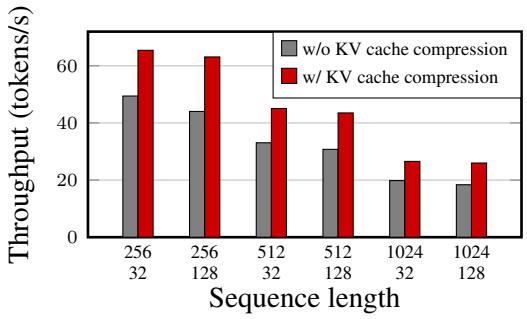


Figure 9: Decoding throughput improvement with KV cache compression enabled on OPT-13B model.

4.5 Ablation Study

Hiding KV cache partial recomputation. To evaluate the effectiveness of the fine-grained offload-

ing 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, KVPR without hiding KV cache recomputation, and KVPR with hiding. When the batch size is 1 and the KV cache size is the smallest, FlexGen can outperform KVPR without hiding. By overlapping the transfer of MHA weights with KV cache recomputation, KVPR ensures performance that is no worse than FlexGen under this scenario, particularly when weight loading is the primary bottleneck. This result shows that KVPR works well for both small and large batch size settings, thereby providing a unified approach to improve decoding performance.

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
KVPR (w/o. hiding KV recomputation)	1.749	3.461	**6.766**	12.930	23.613	43.462
KVPR (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.

Runtime breakdown. Figure 10 presents the runtime breakdown of an MHA block in KVPR and FlexGen during decoding stage. KVPR 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 KVPR 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.

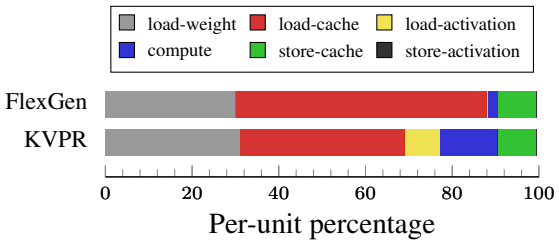


Figure 10: Runtime breakdown of KVPR and FlexGen.

5 Related Works

To address the memory demands of LLMs in resource-constrained settings, offloading techniques aim to minimize the latency of data transfer 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., 2024a) 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) overlap GPU linear projection computations with CPU-based attention computations across multiple batches to improve resource utilization.

ALISA (Zhao et al., 2024b) compresses the KV cache based on sparsity and offloads KV cache exceeding GPU memory capacity. When loading the KV cache to the GPU, ALISA recomputes a portion of KV cache first and then transfers the remainder, where we propose overlapping the recomputation and transfer by adaptively determining the optimal split point. Furthermore, ALISA addresses only for the row-by-row schedule, while KVPR extends to column-by-column schedule. KVPR is orthogonal to CPU-assisted and KV cache compression approaches, making it compatible for integration with these techniques to further improve overall system performance. As shown in the additional experiments provided in Appendix A.6, we demonstrate that the CPU can become a bottleneck in certain distributed system configurations. In contrast, KVPR optimizes GPU utilization and data transfer efficiency without relying on additional CPU resources or approximations of the KV cache.

6 Conclusion

In this paper, we introduce KVPR, an efficient CPU-GPU I/O-aware LLM inference method designed to accelerate KV cache loading. KVPR minimizes the data transfer between the CPU and GPU by leveraging KV cache partial recomputation. By overlapping this recomputation with data transmission, KVPR significantly reduces idle GPU time and enhances overall inference performance.

7 Limitations

Our study represents an important step towards optimizing the efficiency of LLM inference by leveraging KV cache partial recomputation. However, KVPR has certain limitations that suggest avenues for future research. First, our methodology is currently limited to single-GPU and data-parallel multi-GPU inference. It does not yet extend to advanced distributed systems, such as model or tensor parallelism. Expanding this approach to these paradigms could enable support for larger model sizes. Second, while we address PCIe bandwidth bottlenecks in CPU-GPU communication, we do not consider scenarios where the KV cache is loaded from disk or network storage. Nevertheless, KVPR could potentially be adapted to accelerate the prefilling stage in such setups. Third, the current implementation performs system profiling only at the start of inference, assuming static hardware conditions throughout the process. Incorporating dynamic profiling and runtime adaptive optimization could enhance the robustness and efficiency of the approach, particularly in heterogeneous or multi-tenant environments.

References

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, 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*, pages 1–15. IEEE.

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2024. Flashattention: fast and memory-efficient exact attention with io-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22.

Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. 2023. Flash decoding: Advances in efficient text generation.

Google Gemini Team. 2024. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Shijie Geng, Shuchang Liu, Zuohui Fu, Yingqiang Ge, and Yongfeng Zhang. 2022. Recommendation as language processing (rlp): A unified pretrain, personalized prompt & predict paradigm (p5). In *Proceedings of the 16th ACM Conference on Recommender Systems*, RecSys '22, page 299–315.

Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab Mangrulkar, Marc Sun, and Benjamin Bossan. 2022. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate.

Jiaao He and Jidong Zhai. 2024. Fastdecode: High-throughput gpu-efficient llm serving using heterogeneous pipelines. *Preprint*, arXiv:2403.11421.

Coleman Richard Charles Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Xuanlin Jiang, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu. 2024. Neo: Saving gpu memory crisis with cpu offloading for online llm inference. *Preprint*, arXiv:2411.01142.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 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)*, pages 155–172.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for "mind" exploration of large language model society. *Preprint*, arXiv:2303.17760.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024a. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 38–56.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024b. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 32332–32344.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, and et al. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Daon Park and Bernhard Egger. 2024. Improving throughput-oriented llm inference with cpu computations. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, pages 233–245.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: high-throughput generative inference of large language models with a single gpu. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *Preprint*, arXiv:2302.13971.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30.

Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Jiachen Liu, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, Mosharaf Chowdhury, and Mi Zhang. 2024. Efficient large language models: A survey. *Transactions on Machine Learning Research*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*.

Chengye Yu, Tianyu Wang, Zili Shao, Linjie Zhu, Xu Zhou, and Song Jiang. 2024. Twinpilots: A new computing paradigm for gpu-cpu parallel llm inference. In *Proceedings of the 17th ACM International Systems and Storage Conference*, SYSTOR '24, page 91–103.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. Opt: Open pre-trained transformer language models. *Preprint*, arXiv:2205.01068.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Re, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Xuanlei Zhao, Bin Jia, Haotian Zhou, Ziming Liu, Shenggan Cheng, and Yang You. 2024a. Hetegen: Efficient heterogeneous parallel inference for large language models on resource-constrained devices. In *MLSys*.

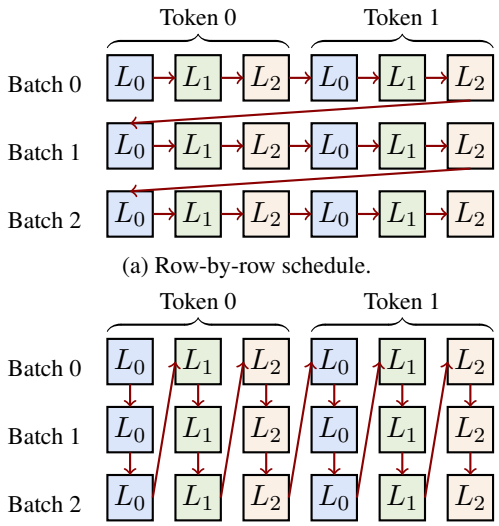
Youpeng Zhao, Di Wu, and Jun Wang. 2024b. ALISA: Accelerating Large Language Model Inference via Sparsity-Aware KV Caching . In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 1005–1017.

Wenhao Zhu, Hongyi Liu, Qingxiu Dong, Jingjing Xu, Shujian Huang, Lingpeng Kong, Jiajun Chen, and Lei Li. 2024. Multilingual machine translation with large language models: Empirical results and analysis. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2765–2781.

A Appendix

A.1 Scheduling 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 KV Cache Partial Recomputation with Overlapping

Built on FlexGen (Sheng et al., 2023)'s computation and communication overlapping technique, we adapt it to support KV cache partial 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 the rest of KV cache and activations for the next batch, storing the KV 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 KV Cache Partial 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 KVPR and the baseline (Hugging Face Transformer with KV cache offloading) in terms of GPU peak memory, decode latency, and throughput across various configurations. Notably, KVPR consistently achieves lower latency while maintaining comparable memory usage.

A.4 Optimal KV Cache Split Points

Figure 12 presents the optimal KV cache split points l, obtained by solving the linear programming problem defined in Eq. (11), for the first setting of the latency-oriented workload experiments in Section 4 (prompt length of 128 and generation length of 32). Based on system profiling statistics and KV cache size, the optimal split point l is 182 when the generation length is 1, and l increases to 128 when the generation length is 32.

Method	Batch size	Prompt length	Generation length	Cache size (GB)	GPU peak mem (GB)	Decode latency (sec)	Decode throughput (tokens/s)
Accel.	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
KVPR	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)
Accel.	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
KVPR	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.

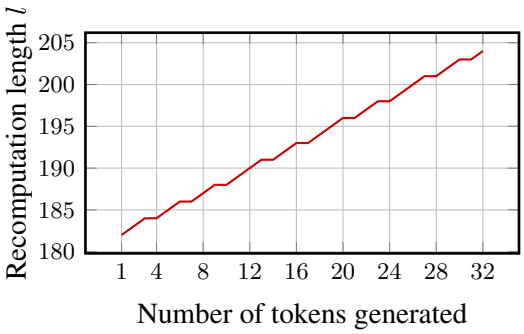


Figure 12: Optimal KV cache split points l over the generation process.

A.5 System Performance with a Low-end GPU

To further demonstrate the adaptability of KVPR, we evaluate it on a low-end system with an AMD EPYC 32-Core CPU and an NVIDIA Quadro RTX 5000 GPU (16 GB HBM, 89.2 TFLOPS FP16 peak performance) connected via PCIe 4.0 x8 (16 GB/s bandwidth). GPU TFLOPS, GPU memory and PCIe bandwidth are lower in this system setting than those in the default system we used earlier. Despite the reduced GPU speed and bandwidth, KVPR achieves up to 15% higher throughput than FlexGen for OPT-6.7B in the same throughput-oriented workload, as shown in Table 5.

Seq len	**256/32**	**256/128**	**512/32**	**512/128**	**1024/32**	**1024/128**
FlexGen	50.057	46.779	29.614	28.650	15.778	16.194
KVPR	53.976	49.860	33.666	32.277	18.285	18.108

Table 5: Throughput (tokens/s) comparison on a low-end GPU system.

A.6 Comparing with CPU-assisted Approaches in Distributed System Setup

In this experiment, we compare the performance of the CPU-assisted offloading approach, FastDecode (He and Zhai, 2024), with KVPR on a GPU node equipped with 8 NVIDIA A100 GPUs and a single CPU, which is the same AMD EPYC processor (64 cores with PCIe 4.0 128 lanes), as described in Section 4.

We run multiple concurrent processes of FastDecode and KVPR on the available GPUs, with each GPU dedicated to a single process. This setup simulates scenarios where either multiple users share a single computing node or a single user performs data-parallel inference. FastDecode relies on the CPU for attention computations, resulting in a performance drop as the CPU becomes a bottleneck when managing multiple concurrent inference processes. In contrast, KVPR eliminates CPU dependency entirely and instead optimizes data transfer over the PCIe bus.

Figure 13 demonstrates that while FastDecode suffers a significant decline in throughput as the number of processes increases, KVPR exhibits better scalability, maintaining stable performance in systems with a single CPU and multiple GPUs.

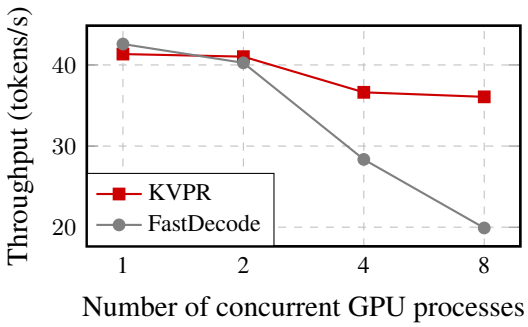


Figure 13: Throughput comparison between KVPR and FastDecode with different GPU workload.

A.7 Extended 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., 2024) combines attention operations into a single kernel and tiles QKV matrices into smaller blocks to optimize GPU SRAM usage and reduce HBM access overhead, while our work mainly focuses on optimizing PCIe bandwidth. 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.

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., 2024) 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.