CPU-GPU Workload Distribution during Throughput-Oriented LLM Inference on Single-GPU Systems

DAON PARK, Seoul National University, Gwanak-gu, Korea (the Republic of)
BERNHARD EGGER, School of Computer Science and Engineering, Seoul National University, Seoul, Korea (the Republic of) and Lucerne University of Applied Sciences and Arts, Lucerne, Switzerland

Large language models (LLMs) have recently achieved remarkable performance in text generation, capturing the attention of a broad audience. This success, driven by the rapid growth in model parameters, comes at the expense of significantly higher operational costs and decreased processing speed. These costs, combined with privacy concerns around cloud-based deployments, have motivated research into running LLMs on commodity hardware. For example, researchers have used the memory hierarchy to boost throughput by increasing the number of batches. These studies, however, tend to overlook or inefficiently utilize the additional computational resources provided by the CPU.

In this work, we present a dynamic workload allocation technique that efficiently distributes computation across all available hardware resources. The proposed method targets decoder-based models on standard general-purpose hardware, effectively minimizing idle periods for both the CPU and the GPU. Experiments show that our approach achieves up to 30% higher throughput compared to the state of the art, regardless of model architecture, LLM optimizations, and input batch sizes.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; **Natural language generation**.

Additional Key Words and Phrases: CPU offloading, Throughput-latency tradeoff, Large language model generation, CPU-GPU workload distribution, Single commodity-GPU systems

1 Introduction

Today, a wide variety of tasks use deep learning [2, 5, 23], and one of the most researched areas is natural language processing (NLP) [1, 7, 24]. In particular, LLMs have achieved remarkable performance across a wide range of NLP tasks [12, 37, 42], driven by a rapid growth in the number of model parameters. This scaling trend, however, raises significant concerns: operational costs increase with model size, and reliance on cloud-based inference introduces privacy risks for sensitive workloads. These factors have motivated interest in running LLMs on local, commodity hardware. However, these large models are often too big to fit into a single commodity GPU and require expensive high-end hardware. The OPT-30B [48] model using the FP16 datatype, for example, requires 60 GiB of memory to hold all model parameters in memory, and additional memory space is needed during the inference process. When running LLM inference, a frequently employed approach is to cache the entire model in more than one GPU at the expense of requiring the inference servers to be over-provisioned for the peak loads, thereby increasing the total cost of ownership. Another approach is to offload the model to lower-tiered

Extension of Conference Paper. This article extends [33] in three ways. First, we identify orthogonal optimizations applicable to throughput-oriented LLM workloads (Section 2.2) and evaluate them using the CPU-GPU distribution methodology of the prior work (Section 5.6). Second, we introduce a novel operation splitting technique (Section 3.2) and analyze its impact on CPU/GPU utilization (Section 5.3). Third, we present additional experiments examining factors that influence inference performance, including the number of CPU cores allocated for CPU-side computation (Sections 4.1 and 5).

Authors' Contact Information: Daon Park, daon@csap.snu.ac.kr, Seoul National University, Seoul, South Korea; Bernhard Egger (corresponding author), bernhard@csap.snu.ac.kr, Seoul National University, Seoul, South Korea.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 1544-3973/2026/5-ART
https://doi.org/10.1145/3816433

Just Accepted

storage devices such as host DRAM and NVMe storage. This latter approach increases throughput at the cost of latency and is thus more suitable for batch workloads such as data wrangling [27], information extraction [28], and benchmarks [6] that feed thousands of tokens as input.

Recently, noticeable studies regarding the throughput-oriented approach were presented, such as ZeRO-Inference [4], FlexGen [40], HeteGen [45], and CPU-Offload [33]. These studies exploit the slower but larger storage for model accommodation and mitigate the data transfer times on a single commodity GPU system through computation-communication buffering and efficient I/O scheduling with overlaps. Such methods of exploiting slower storage reduce infrastructure costs but significantly impact per-token generation latency. Therefore, these works mainly target running LLMs on single commodity GPU systems with workloads consisting of multiple batches of long sequence inputs and trade-off latency for throughput.

Long contexts are a key feature of LLMs, enabling the processing and generation of text over extended sequences. The primary challenge of supporting long contexts lies in the increased computational cost, which is commonly addressed through KV-cache optimization [10]. While the KV-cache reduces computational cost during the generation phase, it shifts the bottleneck to memory, making inference memory-bound. Substantial research effort has therefore been directed at reducing LLM inference costs associated with KV-cache data, GPU memory management, quantization, compression, and KV-cache selection, among others. However, existing offloading schemes largely fail to combine these optimizations with efficient use of available compute resources. In this paper, we focus on improving the throughput of a single commodity GPU system by employing CPU compute resources, without assuming any particular optimization technique for token generation.

To achieve efficient CPU-GPU workload distribution, we first consider the characteristics of the inference workload as generated by state-of-the-art frameworks to identify where and how computations can be offloaded to the CPU. LLMs typically employ a decoder-only architecture [1, 43, 48], where each decoder layer consists of a multi-head attention (MHA) mechanism and a feed-forward network (FFN). Three sets of weights must be transferred to the GPU for inference: the MHA weights, the FFN weights, and the key-value cache (KV cache) [10] from all of the previous model iterations. The MHA weights and the KV cache are required for MHA layer inference, while the FFN layer can only start after its weights have been transferred. When multiple batches are involved, a KV cache exists for each batch, and every KV cache needs to be transferred for MHA layer inference. If the weights or the cache are compressed to reduce transfer times, decompression causes additional overhead on the GPU. As the model size and the number of input batches increase, relying solely on the GPU for an already memory-bound workload misses significant opportunities for increasing the throughput.

This work specifically aims to employ the CPU to reduce the amount of data transferred and decrease the GPU's idle time. The goal is to split the computation in such a way that computation hides the data transfers whenever possible. Identifying where to divide the model to balance the computation and data transfer times is challenging because the workload's characteristics vary depending on factors such as the input batch size, the use of compression, and the number of batches. We address this challenge with a dynamic division module that profiles the target system to determine the optimal GPU-CPU workload distribution. Our contributions are as follows:

- We exploit CPU compute to reduce data transfer bottlenecks during throughput-oriented LLM inference on commodity hardware.
- We introduce a dynamic division module that profiles the target system and determines the optimal workload split between CPU and GPU based on batch size and decoder layer implementation.
- We show that the approach can be combined with other existing LLM optimizations that do not involve workload distribution, and validate its applicability across different model architectures through fine-grained operator-level integration.

Just Accepted

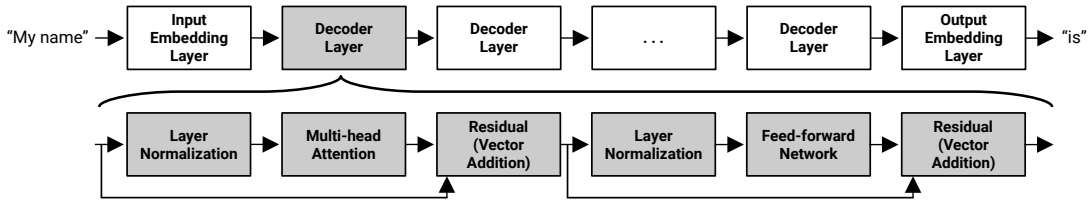


Fig. 1. High-level overview of OPT model and decoder layer architecture.

- We evaluate the presented technique against the state-of-the-art throughput-oriented framework FlexGen [40], achieving improvements of up to 105% with the OPT-30B model and approximately 30% across other LLM workloads and inference configurations.

The remainder of this paper is organized as follows. Section 2 discusses the background of LLM inference and motivates the design of throughput-oriented systems. Sections 3 and 4 detail the implementation of the presented technique. Section 5 presents the evaluation results, Section 6 surveys related work, and Section 7 concludes the paper.

2 Background and Motivation

2.1 Decoder-based Architecture

Recent LLMs are primarily based on decoder-only architectures thanks to their superior zero-shot generalization performance compared to encoder-only or encoder-decoder models [7]. A typical decoder-only model architecture consists of an input embedding layer at the start, an output embedding layer at the end, and multiple decoder layers in between, where each decoder layer is composed of an MHA and an FFN layer, as shown in Figure 1. All of these layers, including the embedding layers, contain large weight matrices that are required for every token generation step.

The auto-regressive nature of LLM inference consists of two phases: the *prefill phase* is the first iteration that processes all input tokens and creates the initial KV-cache [10]. The following *generation phases* take the previous iteration's output and KV-cache to produce a single output token at each forward pass. The KV-cache lowers the computational cost for every decoder layer but increases the already substantial memory footprint of LLM inference, making KV-cache management the primary challenge. The KV-cache size is not fixed and grows linearly with the batch size and the total sequence length, as shown in Equation 1,

$$size_{cache} = 2 \cdot l \cdot b \cdot (i + g) \cdot h \cdot d \cdot p \tag{1}$$

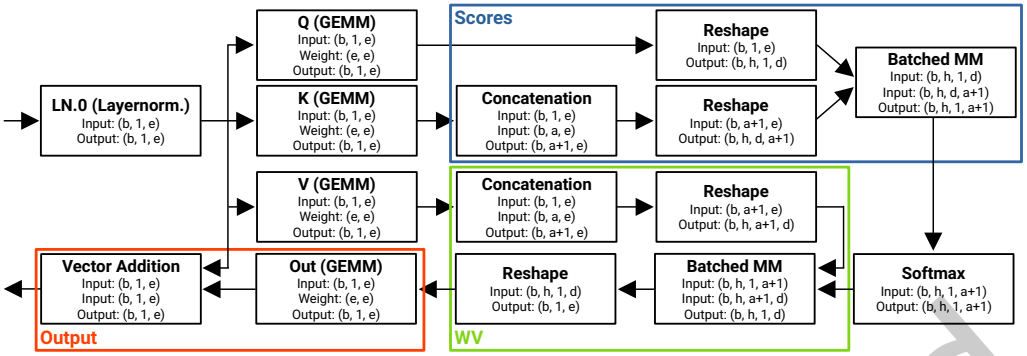
where l denotes the number of MHA layers, b the batch size, i the input token length, g the generated token length, h the number of attention heads, d the head dimension, and p the byte size from a precision.

However, the KV-cache does not reduce the computational demands of the workload, as the large KV-cache is used in batched matrix multiplications (BMMs) for similarity score calculations, as shown in Figure 2a. In addition, the relatively large weight matrices used in the fully connected (FC) layers of the FFN layer, as shown in Figure 2b, further contribute to the computation through heavy generalized matrix multiplications (GEMMs). Such computationally intensive workloads are well-suited for GPUs, but the limited VRAM poses constraints for the memory-bound LLM inference workload.

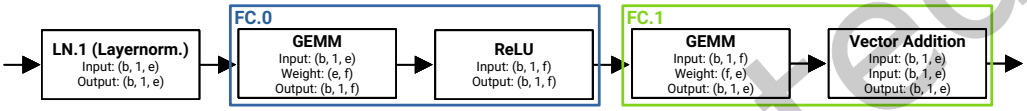
2.2 LLM Optimizations

LLM inference workloads impose substantial computational and memory requirements, necessitating deployment on high-performance computing systems equipped with large main memory and state-of-the-art GPUs, primarily

Just Accepted



(a) MHA layer with input(s), output, and weight shape descriptions.



(b) FFN layer with input(s), output, and weight shape descriptions.

Fig. 2. OPT layer architecture depiction. b denotes the batch size, f the FFN dimension, e the embedding dimension, h the number of heads, d the head dimension, and a the number of tokens stored in the KV-cache. Operation names of Q, K, V, WV, FC denote Queries, Keys, Values, Weighted-values, and fully-connected, respectively.

within data centers. The high cost associated with these workloads has driven extensive research into optimizing both memory and computational efficiency.

One critical area of optimization involves modifications to the MHA layer architecture, as the embedding dimension directly influences weight storage, intermediate activations, and KV cache size. A primary strategy for reducing memory overhead is adjusting the number of attention heads for key and value projections. For instance, Google's multi-query attention (MQA) implementation [38] reduces the number of heads to one, whereas grouped-query attention (GQA) [3] maintains eight heads to align with the number of GPUs or nodes utilized during training. Reducing the number of heads results in a linear decrease in KV-cache size, thereby enabling a longer context window and potentially improving accuracy. However, long-context generation still encounters excessive KV-cache size challenges.

Alternative approaches to optimizing MHA focus on improving data movement efficiency and execution order. A widely-adopted I/O-aware kernel design is FlashAttention [11], which reduces memory traffic by fusing attention operations and carefully structuring computations to maximize on-chip memory reuse. By minimizing costly reads and writes to high-bandwidth memory, this approach significantly accelerates the attention mechanism, particularly for long sequences where memory bandwidth becomes a bottleneck.

Another line of work explores operation reordering [20], which restructures the sequence of attention-related computations to improve cache locality and reduce intermediate memory footprint. By reorganizing how intermediate activations are produced and consumed, this method effectively halves the required cache size and achieves up to 50% higher throughput compared to conventional implementations. Together, these approaches highlight that optimizing memory access patterns and execution ordering is as critical as improving raw compute efficiency in modern MHA implementations.

Another extensively researched technique for optimizing LLM inference, as well as deep neural networks in general, is model quantization [19, 44] and compression [13, 25, 39, 49]. These techniques reduce model weight

Just Accepted

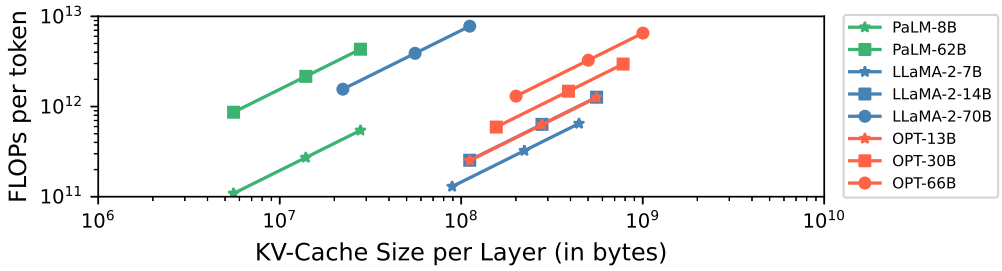


Fig. 3. FLOPs and KV-cache size based on model architecture. The markers within a single line denote 10, 25, and 50 batches, respectively.

and KV-cache size, though they introduce additional computational overhead. The impact on model accuracy depends on the precision levels employed, necessitating careful evaluation. Various methods leverage integer data types or hybrid precision approaches, such as the Any-Precision technique [35], which enables multiple data representations within the same model.

Further efforts to mitigate memory demands during inference involve selective retention of KV-cache elements. The H2O approach [49] employs a fixed sliding window mechanism to limit cache data transfers, similar to the rotating buffers implemented in Mistral AI models [17, 18]. InfiniGen [22] introduces a dynamic selection strategy for cache management, significantly enhancing token generation throughput. However, dynamic cache selection introduces additional computational overhead during both prefill and generation phases, affecting workload distribution.

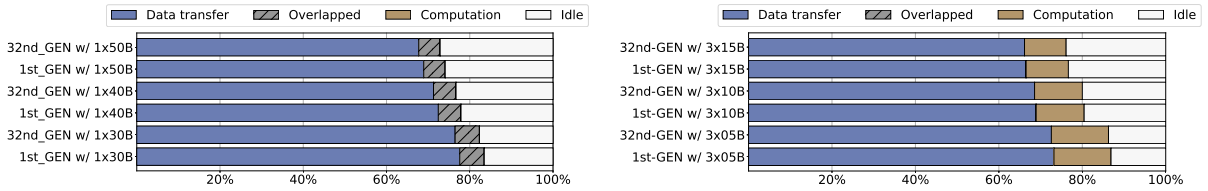
With numerous optimization techniques in mind, the computational characteristics of LLM inference workloads vary depending on batch size, sequence length, and decoder architecture. Figure 3 illustrates the relationship between floating-point operations (FLOPs) and KV-cache size across different workloads and batch sizes. Variability in FLOP requirements arises from differences in implementation strategies, such as rotary embeddings [39] and reductions in attention heads affecting KV-cache formation. The diversity of LLM inference workloads necessitates a comprehensive approach to runtime optimization, considering multiple factors to ensure efficiency and scalability.

2.3 Throughput-oriented Systems

Throughput-oriented workloads often run LLM inference in batches over large input sequences at the expense of latency, making them suitable for offloading systems that use the GPU buffering technique with cheaper but larger storage. Recent offloading frameworks such as ZeRO-Inference [4], HuggingFace Accelerate [16], and FlexGen [40] aim to utilize the host DRAM and NVMe storage to accommodate the whole model and the KV-cache.

There are two types of throughput-oriented workloads regarding LLM inference. One type of workload is the *single-batch workload* that overlaps data transfers with single batch computation and is used as a default workload type in all throughput-oriented frameworks. The optimal single-batch workload of a decoder layer in a generation phase aims to completely overlap the data transfers with the GPU computations to maximize throughput. The other approach is the *multi-batch workload* that reuses the weights as much as possible by completing the computation of a layer for every batch. For example, if there are a hundred batches to compute and the user decides to run ten batches at a time, other related works complete inference for ten batches first, then compute the next ten, and so on. The multi-batch workload, on the other hand, first transfers the weights of a

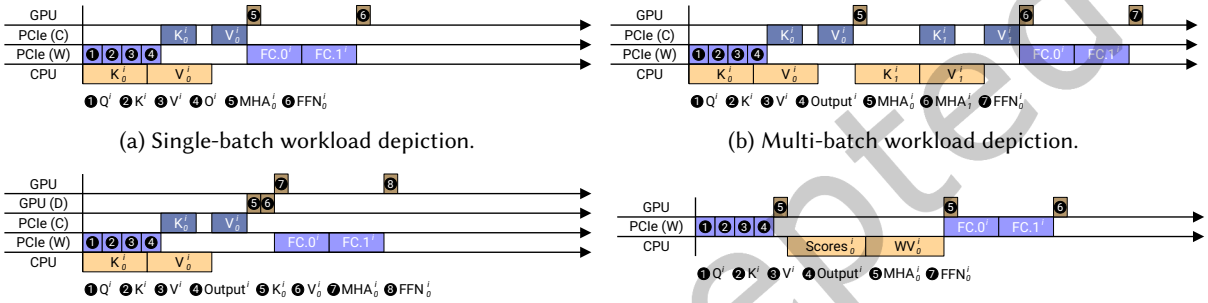
Just Accepted



(a) Single-pipelined workload

(b) Multi-pipelined workload

Fig. 4. Analysis of FlexGen's generation phases using the OPT-30B model with 512 input tokens to generate 32 tokens. `ixjB` denotes the `i` number of batches with batch size `j`.



(a) Single-batch workload depiction.

(b) Multi-batch workload depiction.

(c) Single-batch workload with cache compression. DC denotes decompression routines.

(d) Single-batch workload with CPU computation, which divides the MHA layer compute (⑤) for GPU.

Fig. 5. Illustrations of various generation phases in FlexGen. The GPU, GPU(D), PCIe(C), PCIe(W), and CPU rows denote GPU computations, decompression routines, cache data transfers, weight data transfers, and CPU computations. Each block is associated with the weight of an operation from Figures 2a and 2b, and the superscript and subscript denote the layer index and the batch index, respectively.

Just Accepted

layer (MHA or FFN), processes all hundred batches in sets of ten, and then proceeds with the next layer. FlexGen's zig-zag scheduling uses this multi-batch workload for maximizing throughput performance on a single-GPU system.

While the optimal workloads aim to completely overlap data transfers with computations, GPU computations are considerably faster than data transfers during LLM inference. For this reason, the recent state-of-the-art throughput-oriented framework FlexGen suffers from low GPU utilization. Figure 4 breaks down FlexGen's OPT-30B generation phases in more detail, with the hardware configuration as shown in Table 1. The single-batch workload exhibits low GPU computation time and is dominated by data transfers, while multi-batch workloads suffer even further since only a single KV-cache can be present in the GPU at the same time. In both cases, data transfers constitute the bottleneck and prevent faster processing, accounting for at least 70% of total execution time; single-batch workloads process only a single token per batch and are dominated by layer parameter transfers, whereas multiple KV-cache transfers stall the multi-batch workload.

In addition, the computations are entirely overlapped with data transfers in single-batch workloads, while there is little to no overlap in multi-batch workloads, with idle time accounting for at least 15% of the workload. This is due to the pinning procedures used by FlexGen; CUDA has two types of memory called pageable memory and pinned memory [29], where pinned memory is similar to page-locked memory, allowing the transfer mechanism to identify data for asynchronous transfers. In other words, using pinned memory creates a page-locked host

array to transfer data to the GPU, minimizing the effect of transfer latency. FlexGen pins data prior to transfer by copying it into a new pinned memory region, incurring significant memory copy overhead, as shown in the CPU rows in Figure 5, where the on-the-fly pinning procedures are applied to the KV-cache data only. Such on-the-fly pinning procedures severely limit overlapping opportunities during the generation phase. Furthermore, GPU idle times are higher in intermediate generation phases compared to the first and last phases, as the on-the-fly pinning overhead grows with the KV-cache size.

FlexGen includes two additional modes for the generation phase: compression settings and CPU-compute mode. The compression setting, described in Section 2.2, compresses FP16 weights and KV-cache to allow more input batch creation and improve throughput performance. Compressing the data reduces transfer times, but requires decompression routines before computation, adding to the GPU workload. Figure 5c shows the single-batch workload of a single layer with decompression routines, where decompression occurs only after the weights are transferred, further reducing overlapping opportunities. CPU-compute mode, on the other hand, offloads computation to the CPU as an optional feature, recommending it for I/O-bound situations where keeping the KV-cache stationary in host DRAM is more beneficial than transferring it to the GPU for computation. Enabling this option may reduce long transfer times, but it does not overlap CPU with GPU computations, as shown in Figure 5d. Based on these observations, this paper presents an improved offloading technique that supports dynamic workload division, maximizing the benefits of CPU offloading across all circumstances.

3 CPU-GPU Workload Distribution

3.1 Single-batch Workload

The single-batch workload is the basic form of the generation phase used in most existing throughput-oriented frameworks, such as ZeRO-Infinity, HuggingFace Accelerate, and FlexGen. Since the host DRAM size far exceeds the VRAM size, the GPU VRAM generally limits the size of a single batch, even though the host DRAM can accommodate larger or more batches. During the single-batch workload, existing frameworks overlap computation with communication for both MHA and FFN layers, i.e., the FFN layer weights are transferred while the MHA layer is computed and vice versa; however, this overlap yields low throughput because data transfer times exceed GPU computation times.

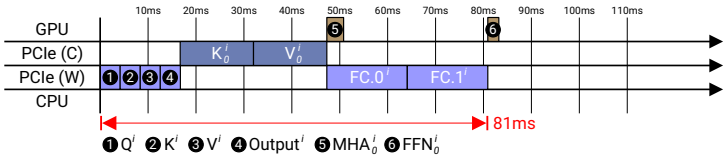
In this work, the CPU cores are employed during the generation phases to perform computation while weights are being transferred to the GPU, ensuring that at least one compute device is active at all times, thereby realizing *pipeline-parallelism*. CPU computational offloading not only reduces decoding latency, but also requires fewer weights to be transferred from the host DRAM to the GPU VRAM; furthermore, pipeline-parallelism introduces fewer synchronization points than model-parallelism [45, 46].

The state-of-the-art throughput-oriented framework FlexGen uses pinned main memory areas to enable fast and asynchronous transfers to the GPU; however, the necessary data copies from pageable memory into pinned memory incur significant overhead. In this work, stationary weights are allocated in pageable memory while data to be transferred to the GPU is placed in pinned memory, eliminating any on-the-fly pinning procedures, removing heavy memory copies, and freeing the CPU for computational offloads. If all data to be transferred to the GPU during inference is pinned before the generation phase begins, the expensive memory copy overhead can be eliminated and idle times can be reduced. Figure 6b shows the generation phase of a single-batch workload without on-the-fly pinning, with actual performance measurements in Figure 6a. Compared to Figure 5a, the idle transfer times are eliminated as a result of removing the on-the-fly pinning procedures. This alone brings significant throughput improvement, as idle times grow in FlexGen with larger batch sizes or when not all weights are pinned before the start of LLM inference. However, GPU computations are still considerably faster than data transfers, precluding complete overlap; our work therefore identifies computation offloading opportunities on the CPU to overlap with the GPU, as shown in Figure 6c.

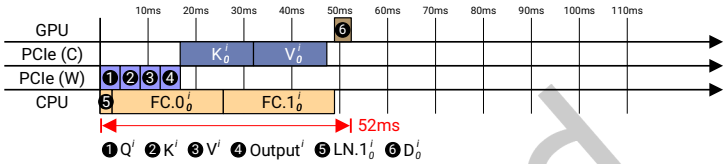
Just Accepted

	H2D	CPU
LN.0	0.001	0.255
Q	4.214	5.775
K	4.214	5.775
V	4.214	5.775
Scores	15.344	77.527
Softmax	0.000	0.776
WV	15.344	77.527
Output	4.214	5.775
LN.1	0.001	0.255
FC.0	16.856	23.216
FC.1	16.856	24.193

(a) Host-to-device (H2D) and CPU computation latency measurements in ms. (OPT-30B, single batch of size 50)



(b) Single-batch workload w/o on-the-fly pinnings.



(c) Single-batch workload with CPU offloads.

Fig. 6. Illustration of improved single-batch generation phases based on actual performance measurements. Performance was measured with hardware in Table 1. Gantt charts follow the same notations from Figure 5.

Just Accepted

Furthermore, the model implementation can be optimized for CPU computations. For example, the current implementation includes a heavy reshape operation after concatenating the produced KV matrices with the KV-cache, as shown in Figure 2a. These reshape operations incur high CPU computational latency, reducing overlapping opportunities since the GPU is orders of magnitude faster at all other operations. However, the reshape operation can be reordered to lower CPU computation latency. Instead of reshaping the concatenated KV matrices at every forward pass, the KV-cache can be directly stored in the target shape. In other words, the K-cache and the V-cache can be stored as (bh, d, a) and (bh, a, d) instead of (b, a, e), respectively. This way, the KV-cache does not need to be reshaped at every forward pass, and only the newly produced KV matrices require reshaping. Eliminating KV-cache reshapes becomes increasingly beneficial as the cache size grows, and a faster attention layer computation on the CPU allows more work to be delegated to the CPU, which in turn minimizes data transfer latency and GPU computation latency.

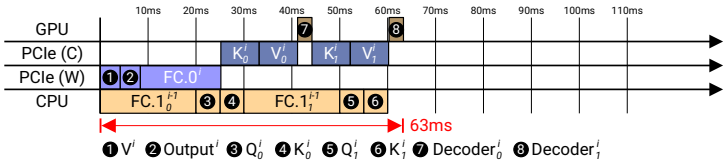
3.2 Multi-batch Workload

The multi-batch workload increases the number of times each weight is used after it has been transferred to the GPU VRAM. Building upon the single-batch workload, the multi-batch workload can benefit from pipeline-parallelism but requires additional consideration. Firstly, since multiple batches are involved, a KV-cache data transfer is required for each batch, as shown in Figure 5b. However, a batch's KV-cache data transfers cannot be overlapped with the previous batch's GPU computations because only a single batch's KV-cache can reside in the GPU at a time. Instead, the CPU must remain active throughout the data transfers to maximize generation throughput. For example, Figure 7 shows that CPU computations need not be stalled by data transfers or GPU computations, thus maximizing CPU utilization for faster decoder layer inference. To implement the multi-batch workload, both data transfer latency and GPU computation latency must be measured for precise workload division, and all CPU computations for a batch must complete before its GPU computation begins for correctness, as discussed in Sections 4.1 and 4.2.

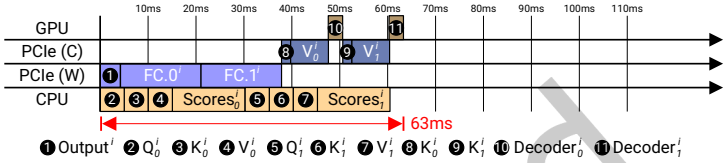
Secondly, to further increase overlapping opportunities, the decoder layer can be split at any point to balance CPU computation against data transfers. Depending on the workload, the latency of the MHA workload on the CPU may be significantly shorter than the time required to transfer the FFN data to the GPU. In such cases, parts of the FFN workload can be executed on the CPU to maximize the overlap of data transfers with CPU

	H2D	CPU	GPU
LN.0	0.001	0.255	0.038
Q	4.214	5.030	0.323
K	4.214	5.030	0.323
V	4.214	5.030	0.323
Scores	7.884	36.450	0.463
Softmax	0.000	0.491	0.066
WV	7.884	36.450	0.463
Output	4.214	5.030	0.323
LN.1	0.001	0.255	0.038
FC.0	16.856	18.910	1.097
FC.1	16.856	19.891	1.174

(a) Host-to-device (H2D), CPU, GPU computation latency measurements in ms. (OPT-30B, two batches of size 25)



(b) Two-batch workload with two split points.



(c) Two-batch workload with a single split point after Scores split.

Fig. 7. Illustration of improved single-batch generation phases based on actual performance measurements. Performance was measured with hardware in Table 1. Gantt charts follow the same notations from Figure 5.

computations, dividing the decoder layer into three sections with two split points. An illustration of splitting the workload twice is given in Figure 7b, where the large cache data is transferred to be handled by the GPU rather than the CPU. Instead of having the lightweight $\mathtt{Q}_i$, $\mathtt{K}_i$, and $\mathtt{V}_i$ operations computed by the CPU – which would introduce idle time – the heavier MLP operation at the end of the sequence is assigned to the CPU to minimize idle time.

While inter-operation split points can balance the CPU and GPU workloads in many cases, they may not always yield a sufficiently fine-grained balance. In such cases, individual operations can themselves be split in an intra-operation split to more precisely align CPU computation times with data transfer times. A simple example is shown in Figure 7c, where the `Scores` operation is split to balance the CPU-GPU workload. Operation splitting can be applied regardless of model-specific optimizations such as GQA [3]. GQA reduces the KV-cache size, but the computational workload remains relatively unchanged compared to multi-head attention. The reduced data transfer volume limits the opportunity to offload computation to the CPU, as it will generally be more advantageous to perform computations on the GPU. When most computation is performed on the GPU, pipeline-parallelism is hampered by long GPU computation times at large batch sizes. To alleviate this overhead, batched matrix multiplications over the KV-cache data can be split into smaller operations to reduce GPU computation latency and device memory copies.

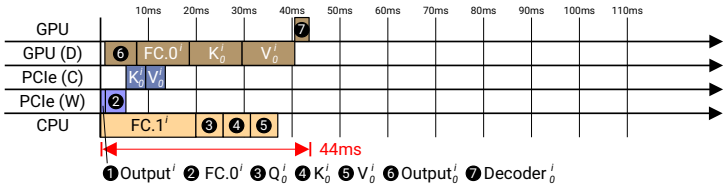
3.3 Compression

Compression can enhance data transfer efficiency in memory-bound workloads by reducing transfer times; however, it requires additional decompression routines. This approach is particularly beneficial in systems with constrained main memory, though the associated GPU decompression overhead may reduce performance in offloaded scenarios. To maintain efficiency, this overhead must be minimized. The decompression process can be partially overlapped with host-to-device data transfers, as decompression can begin immediately after the first data segment is received. As illustrated in Figure 8b derived from actual performance measurements in Figure 8a, the generation phases improve when decompression routines are incorporated, with decompression operations represented in the `GPU(D)` row. In this setup, decompression begins as soon as the corresponding data becomes available on the GPU, such as the decompression of the `Output` weight matrix (⑥) following the completion

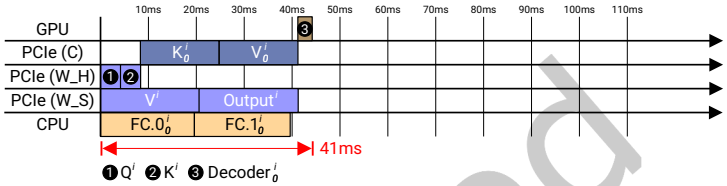
Just Accepted

	H2D(C)	DC	S2D
LN.0	0.001	0.000	0.000
Q	1.083	6.623	20.700
K	1.083	6.623	20.700
V	1.083	6.623	20.700
Scores	4.110	10.963	0.000
Softmax	0.000	0.000	0.000
WV	4.110	10.963	0.000
Output	1.083	6.623	20.700
LN.1	0.001	0.000	0.000
FC.0	4.332	11.446	82.798
FC.1	4.332	11.446	82.798

(a) Compressed H2D, decompression (DC), storage-to-device (S2D) latency measurements in ms. (OPT-30B, single batches of size 50)



(b) Single-batch workload with compression setting.



(c) Single-batch workload with GPUDirect Storage.

Fig. 8. Illustration of improved single-batch generation phases based on actual performance measurements. Performance was measured with hardware in Table 1. Gantt charts follow the same notations from Figure 5, with rows `PCIe(W_H)` and `PCIe(W_S)` denoting host-to-device transfers and storage-to-device transfers.

of its transfer (①). As with multi-batch workloads, the latency introduced by decompression must be carefully accounted for to achieve an optimal workload distribution.

3.4 NVMe Utilization

To optimize GPU bandwidth utilization for data transfers, storage-to-device transfers can be employed to directly access weights stored in NVMe devices. NVIDIA's GPUDirect Storage (GDS) framework [31] enables direct data loading from NVMe storage into the GPU, bypassing the intermediate storage-to-host and host-to-device transfer stages, which are traditionally associated with significant latency and overhead. Since the I/O workload does not require DRAM for buffering, this approach reduces DRAM footprint and mitigates bandwidth contention between CPU computations and host-to-device transfers. These factors significantly influence throughput, particularly in CPU-offload scenarios, where reduced DRAM contention allows for increased batch storage in host memory and accelerated CPU computation, ultimately enhancing overall system throughput, as illustrated in Figure 8c derived from actual performance measurements from Figure 8a. Furthermore, GDS facilitates direct weight transfers, eliminating the need for decompression by storing weights in an uncompressed format on the NVMe devices. When GDS is available, a dedicated process is instantiated for each NVMe device, enabling parallel data loading rather than relying on a single process to transfer all weights into host DRAM. The downside is that all weights must be replicated across all NVMe devices, as it cannot be determined in advance which weights will be requested; however, this represents a modest storage overhead relative to the total capacity of the NVMe devices.

4 Framework Implementation

The workload division options presented in the previous section assume advanced knowledge of the split point(s). While the split point(s) for a given model, configuration, and hardware can be determined manually, this process must be repeated for each use case. Therefore, an automatic and dynamic workload division module is implemented to determine the optimal workload balancing configuration without manual intervention. Figure 9 shows the organization of the framework: the profiler first measures the latencies of each workload on the involved devices for different configurations, the device map creator then determines the optimal division points

Just Accepted

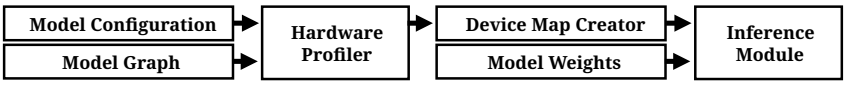
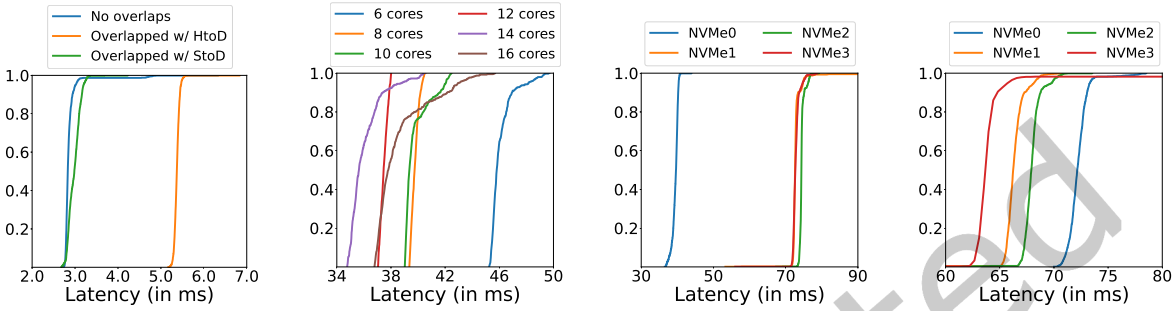


Fig. 9. Block diagram illustrating the overview of the framework.



(a) GEMM latency CDF w/ various transfer types.

(b) GEMM latency CDF w/ various # of CPU cores.

(c) Separate NVMe transfer latency CDF.

(d) Simultaneous NVMe transfer latency CDF.

Fig. 10. Computation and transfer latency CDFs. The GEMM latency was measured with two tensors shaped (1, 7168) and (7168, 7168), and the transfer latency was measured with a (7168, 7168) shaped weight.

and weight device placements from the profiled information, and finally the inference module executes the inference process for the given user inputs.

4.1 Hardware Profiler

The preprocessing phase consists of profiling the target system and creating a device placement map for the runtime module. During the profiling phase, all latency information related to computation, decompression, host-to-device transfers, and, if available, simultaneous storage-to-device transfers is collected.

Accurate measurement of storage-to-device transfer latency is key to achieving accurate workload division. It is important to note that simultaneous CPU computations and memory transfer operations can lead to memory contention that severely degrades the performance of both tasks. Figure 10a shows the CDF of GEMM operation latency compared with the operation overlapped with host-to-device and storage-to-device transfers, illustrating that simultaneous host DRAM access slows down computation increasing GEMM latency by almost double. Based on this observation, the hardware profiler measures CPU computation performance while overlapping with host-to-device data transfers to obtain accurate measurements. Another factor requiring consideration is the number of CPU cores used during computation. Experiments with PyTorch [36], which allows configuring the number of CPU threads used for computation, on the target system (Table 1) show that using all available cores does not always yield optimal results, as shown in Figure 10b. The hardware profiler therefore sweeps the number of cores used during computation to find the lowest achievable CPU latency. After collecting the necessary profiling information, the framework computes the split points that minimize the end-to-end LLM inference latency.

When NVMe devices share PCIe lanes rather than each having a dedicated x16 connection, their effective throughput can vary significantly depending on slot assignment and inter-device interference. Therefore, when using multiple NVMe slots with different speeds, the transfers across devices must be balanced to avoid being bottlenecked by the slowest one. A naive approach of allocating transfer workloads linearly proportional to each

Just Accepted

NVMe device's profiled performance is possible, but can still lead to sub-optimal results. Figures 10c and 10d illustrate the difference between separate and simultaneous transfers of a single weight. Figure 10c shows that one of the four identical NVMe devices achieves almost twice the throughput of the other three on its PCIe lane. However, if transfer sizes are allocated based on single-transfer profiling results, the NVMe throughput may not match expectations under simultaneous transfers, as shown in Figure 10d. In other words, an NVMe device can be over-allocated, disrupting the optimal overlapping scenario. Therefore, the dynamic division procedure iteratively profiles storage-to-device performance while incrementally adjusting the transfer size assigned to each NVMe device after each profiling iteration. For example, consider NVMe devices n_1 and n_2 that require t_1 and t_2 time units to transfer s_1 and s_2 data units, respectively. Given the average latency $t_{avg} = (t_1 + t_2)/2$, the data size units for the next iteration are updated by multiplying by the weighted transfer times, as shown in Equation 2.

$$s_1 = s_1 * \frac{t_{avg}}{t_1},\ s_2 = s_2 * \frac{t_{avg}}{t_2} \tag{2}$$

However, directly applying this multiplier can continuously over-allocate the transfer workload to a single NVMe device and miss the optimal configuration, similar to overshooting a global minimum due to a large step size in stochastic gradient descent. Therefore, only a fraction of the full correction is applied at each step, controlled by a constant factor α as shown in Equation 3. The resulting weight is then applied to the data size, as shown in Equation 4. In our evaluation, approximately 10 profiling iterations were sufficient for the α parameter to converge and achieve a balanced workload distribution across all NVMe devices.

$$w_1 = (1 - \frac{t_{avg}}{t_1}) \cdot \alpha,\ w_2 = (1 - \frac{t_{avg}}{t_2}) \cdot \alpha \tag{3}$$

$$s_1 = s_1 * (1 + w_1),\ s_2 = s_2 * (1 + w_2) \tag{4}$$

4.2 Device Map Creator

All profiled information is passed to the device map creator for the division decision-making process. The profiling information is represented as an array O where $O[i]$ indicates the i-th operation. To determine which subarray $O[i..j]$ must be computed on the GPU and the remainder on the CPU, computation and transfer latencies are measured. The arrays C, G, and H are obtained during the profiling phase, where $C[i]$, $G[i]$, and $H[i]$ denote the CPU computation, GPU computation, and host-to-device transfer latency for operation $O[i]$, respectively. The objective is then to find the split points that minimize the overall latency L,

$$L = \max(\text{CPU}(i, j),\ \text{H2D}(i, j)) \tag{5}$$

where $CPU(i, j)$ and $H2D(i, j)$ return the total CPU computation time and total host-to-device transfer time for a single decoder layer with two split points i and j, as shown in Equations 6 and 7.

$$\text{CPU}(i, j) = \text{sum}(\text{C}[1..i-1]) + \text{sum}(\text{C}[j..\text{len}(O)]) \tag{6}$$

$$\text{H2D}(i, j) = \text{sum}(\text{H}[i..j-1]) \tag{7}$$

If compression is involved, the total decompression latency is calculated by adding the sum of all necessary decompression latencies to the transfer latency of the first weight, since decompression can only begin after the first compressed weight completes its host-to-device transfer. Equation 8 shows the calculation of the total decompression latency, which is then compared against the other components to compute the overall latency L as shown in Equation 9,

$$\text{DC}(i, j) = \text{sum}(\text{D}[i...j])) + \text{H}[i] \tag{8}$$

$$L = \max(\text{CPU}(i, j),\ \text{H2D}(i, j),\ \text{DC}(i, j)) \tag{9}$$

Just Accepted

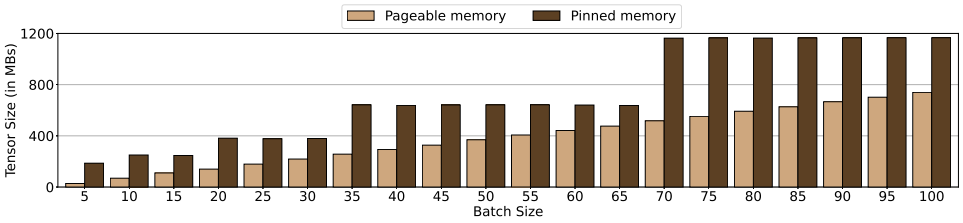


Fig. 11. Comparison of KV-cache memory usage for a single layer of OPT-30B model with 544 tokens.

where $D[i]$ denotes the decompression time of the weight used in operation $O[i]$.

If NVMe devices are involved in weight transfers, the host-to-device transfer latency calculation must be adjusted. Weights selected for storage-to-device transfers are subtracted from the host-to-device transfer latency and treated as separate variables when computing the overall latency L, as in Equation 10,

$$L = \max(\text{CPU}(i,j),\ \text{H2D}(i,j,nvmes),\ \text{DC}(i,j),\ \text{S2D}(nvmes)) \tag{10}$$

where *nvmes* is an array of weight indices assigned to storage-to-device transfers. Given the weight indices in *nvme* and an array S containing the storage-to-device transfer performance for each weight, $H2D$ and $S2D$ are computed as shown in Equations 11 and 12,

$$\text{H2D}(i,j,nvme) = \text{sum}(\text{H}[i..j-1]) - \Sigma_{k \in nvme} H[k] \tag{11}$$

$$\text{S2D}(nvme) = \Sigma_{i \in nvme} S[i] \tag{12}$$

All possible combinations of split-point pairs and NVMe weight configurations are exhaustively evaluated, and the configuration that minimizes both overall latency and inter-stage imbalance is selected as the workload distribution scheme for runtime execution. The size of the search space is primarily determined by the number of split-point candidates, which depends on the number of operations. For a decoder layer with n operations, there are $(n-1)$ possible single split points. In addition, selecting two split points from these $(n-1)$ candidates yields $\binom{n-1}{2}$ combinations. Therefore, the total number of split-point candidates is as shown in Equation 13.

$$(n-1) + \frac{(n-1)(n-2)}{2} = \frac{n(n-1)}{2} \tag{13}$$

The same procedure applies to the multi-batch workload, with the only difference being the calculation of KV-cache transfer latency, which must account for every batch's KV-cache transfers. Therefore, $\text{H2D}(i,j)$ for the multi-batch scenario becomes

$$\text{H2D}(i,j) = \text{sum}(\text{H}[i..j]) + (n-1)(\text{G}[i..j]) + (n-1)(\text{KV}) \tag{14}$$

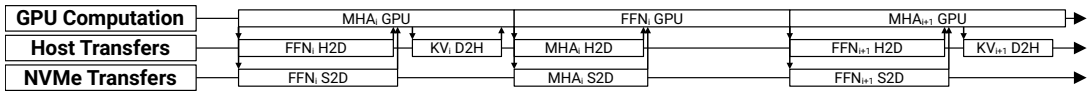
where KV denotes the total transfer latency of all cache data involved. With these equations in place, all combinations of split points and storage-to-device transfers are precomputed and stored prior to the search.

4.3 Inference Module

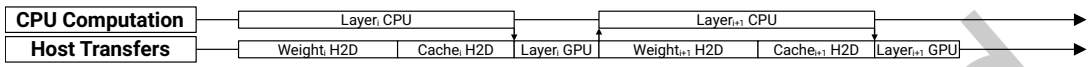
The runtime uses two separate processes for CPU computation and host-to-device transfers. If the GDS mechanism is enabled, additional processes are created – one per NVMe device – to handle storage-to-device transfers. This multi-process design offers two advantages. First, the runtime does not need to issue asynchronous data transfers from within the same process as CPU computations, since transfers are handled in separate CUDA contexts. When a single process submits both computation and transfer kernels to the GPU before starting CPU computations, precisely balancing the GPU and CPU workloads is difficult. With multiple processes, barrier synchronizations can eliminate this balancing complexity and simplify implementation. Second, not all weights

Just Accepted

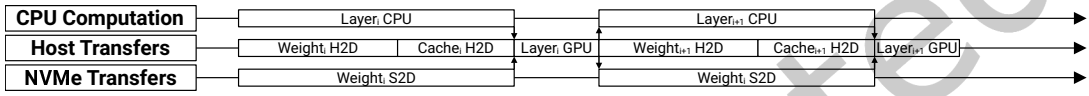
(a) Sequence diagram of a prefill phase using host-to-device only.



(b) Sequence diagram of a prefill phase using host-to-device and storage-to-device transfers.



(c) Sequence diagram of a generation phase using host-to-device transfers only.



(d) Sequence diagram of a generation phase using host-to-device and storage-to-device transfers.

Fig. 12. Sequence diagrams for prefill and generation phases. Subscripts denote layer index.

Just Accepted

need to be pinned before the prefill phase. Pinned memory regions can consume more than double the memory of equivalent pageable regions, as shown in Figure 11; therefore, minimizing pinned memory allocation significantly reduces overall memory requirements during LLM inference.

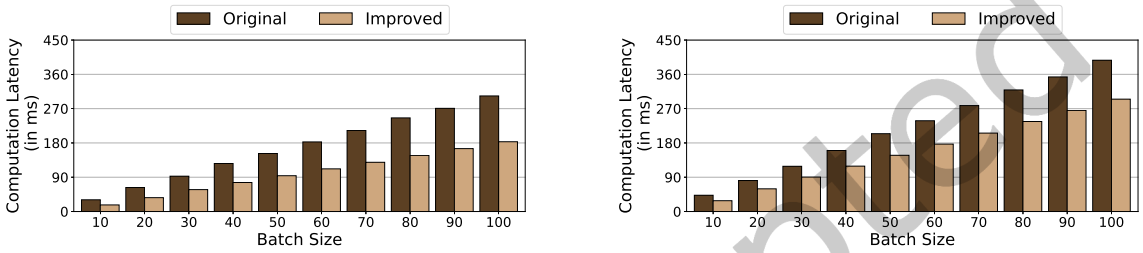
During the prefill phase, no CPU offloading is performed, since throughput-oriented workloads target large input sequences that make the prefill phase computationally intensive; hence, GPU-only execution is more appropriate. Two processes are used for the prefill phase: one for GPU computation and one for weight transfers, as shown in Figure 12a. These two processes are synchronized by the GPU computation process, which signals the transfer process whenever it begins computing a layer and again when the KV matrices have been produced. The first signal initiates host-to-device weight transfers and the second initiates device-to-host KV transfers; both are hidden by the GPU computations. If GDS is enabled, NVMe transfer processes are launched alongside the weight transfer process during the prefill phase, as shown in Figure 12b.

The generation phases employ the CPU as an additional compute device; accordingly, one process is dedicated to CPU computation and another to GPU computation. Figures 12c and 12d show generation phases with separate processes for CPU computation, host-to-device transfers, and GPU computations. The examples illustrate a single-batch workload; additional host-to-device cache transfers are introduced when multiple batches are used. CPU computations and weight transfers notify the GPU computation process upon completion, correctly triggering GPU computations in a pipelined-parallel manner. A notable implementation detail concerns KV-cache placement: if the CPU computation process requires the cache data, it must hold the cache locally. If the cache data resides in another process and must be copied over, this incurs additional memory transfer overhead that degrades throughput. Therefore, once weight placements are determined by the device map creator, the runtime module allocates cache data in the appropriate process from the outset.

A further implementation consideration involves decompression, where correctness requires that decompression begins only after the corresponding transfer completes. The ordering of GPU kernels during decompression is enforced using CUDA streams [30], which guarantee sequential execution of kernels submitted to the same stream. For example, if two weights W_i and W_j are to be transferred and decompressed, submitting the transfer

Table 1. Commodity target system used for evaluation.

CPU	AMD Ryzen 9 7950X 16-core @ 4.5 Ghz (BF16 compute enabled)
GPU	NVIDIA RTX A4000 16GB (PCIe Gen4)
DRAM	Samsung DDR5 32G PC4-44800 x 4
NVMe	SK Hynix Gold P32 NVMe M.2 2TB x 4
M/B	ASROCK X670E PG Lightning
PCIe	(GPU) Gen5x16 (NVMe) Gen5x4, Gen4x4, Gen3x4, Gen4x2
S/W Stack	PyTorch 2.3.0 + CUDA 12.3



(a) Latency comparison w/o data transfer overlaps.

(b) Latency comparison w/ data transfer overlaps.

Fig. 13. Analysis of reshape efficiency comparing Scores computation latency for OPT-30B model.

Just Accepted

and decompression kernels for W_i and W_j to CUDA streams S_i and S_j respectively enforces both intra-stream ordering (transfer before decompression) and independence between S_i and S_j.

5 Evaluation

All experiments evaluate throughput (averaged over 10 runs) on the target system shown in Table 1. The A4000 GPU is the only non-commodity component, used to evaluate NVMe transfers via the GDS feature. The AMD Ryzen 9 CPU supports the BF16 data type; the FP16 model weights are converted to BF16 format for inference. This conversion has only a marginal effect on model accuracy, increasing the perplexity score (lower is better) from 13.3 to 13.37 on the WikiText2-103-test [26] dataset.

5.1 Reshape Efficiency

To assess the computational impact of the reshape operation on CPU execution, as described in Section 3.1, we evaluate both the original and improved implementations and their effects on workload efficiency. Figure 13 presents a comparative evaluation of these two implementations. In the original approach, the entire cache data is reshaped before the BMM operation, resulting in a significant increase in execution latency. Specifically, this method incurs approximately a 70% latency overhead compared to the optimized implementation without overlapped transfers, as shown in Figure 13a.

The improved implementation exhibits increased latency when concurrent with intensive host-to-device data transfers; however, it still achieves a 35% to 50% reduction in execution time relative to the original implementation under the same conditions. This optimization substantially reduces attention layer latency on the CPU, enabling the division module to find a more balanced workload distribution.

Table 2. Preprocessing (offline profiling and split point search) latency based on various configurations. `SC. Split` denotes whether `Scores` operation is split into 8 chunks or not.

Model	Input	SC. Split	Profiling Duration
OPT-30B	1x1	X	38 seconds
OPT-30B	1x100	X	51 seconds
OPT-30B	1x100	O	54 seconds
LLaMA-2-34B	1x1	X	38 seconds
LLaMA-2-34B	1x100	X	52 seconds
LLaMA-2-34B	1x100	O	55 seconds

5.2 Split Point Search

The split-point search relies on an offline profiling phase performed prior to inference execution. In this phase, latency statistics are collected and used to determine the optimal split point. As shown in Table 2, the combined cost of profiling and split-point search is on the order of tens of seconds per configuration. Additional overheads are modest: splitting an operation introduces a few seconds of preprocessing latency, while NVMe profiling incurs approximately one second per iteration (not shown in the table).

When a new configuration is introduced, profiling must be repeated; however, the associated overhead remains small. If the profiler explores multiple CPU core configurations (including varying core frequencies), the total cost scales proportionally with the number of configurations evaluated. To avoid redundant profiling, all results are cached and reused when the same configuration is encountered in subsequent inference runs. The low profiling overhead provides flexibility, allowing re-profiling when necessary, such as in response to hardware changes.

To identify the optimal workload partitioning, an exhaustive search over all possible split points is performed. Representative examples are illustrated in Figure 14, which presents results for all single-point workload divisions. When the overall latency is determined by the maximum of two components – CPU computation and data transfer – the resulting curve exhibits a concave-up shape. As shown in Figures 14a and 14c, distributing the workload between CPU and GPU reduces total latency by 30% and 8%, respectively, compared to a GPU-only baseline, denoted as `LN.0`. Furthermore, Figures 14b and 14d demonstrate that finer-grained partitioning yields additional latency reductions of 7% and 5%, respectively, relative to the CPU-GPU split configuration.

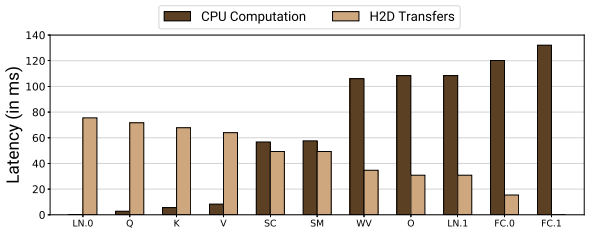
Figure 14 captures only two factors – CPU computation and host-to-device data transfer – as these are the primary drivers of workload balancing in the framework. Modifications to the model implementation or underlying hardware can significantly affect these factors. For instance, faster data transfer rates enabled by newer GPUs or higher-generation PCIe interfaces reduce the opportunity for overlapping computation, whereas slower transfer rates increase it. As the overlap opportunity grows, the benefit of workload balancing correspondingly increases.

5.3 End-to-end Latency

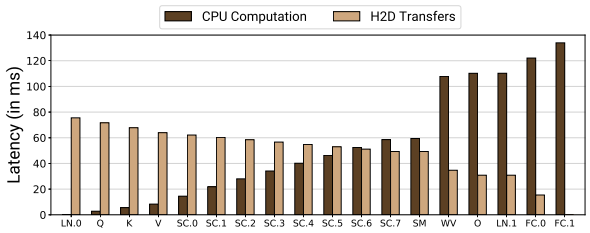
In this section, we evaluate the proposed methodology using the BF16 data type against FlexGen across various workload configurations, including single-batch, single-batch with compression, and multi-batch with compression. FlexGen serves as the baseline, and two variants of our methodology are compared: one without CPU offloading (`GPU-only`) and one with CPU offloading (`GPU+CPU`).

Figure 15 shows inference latency for the OPT-30B model across various token generation lengths with 512 input tokens, with prefill and generation latencies shown as stacked bars. Our approach demonstrates significant performance improvements over FlexGen, with maximum throughput increases of 105%, 80%, 55%, and 95% across the different workload configurations when leveraging both CPU and GPU for computation. Notably, improvements are observed in both the prefill and generation phases. In the prefill phase, FlexGen performs

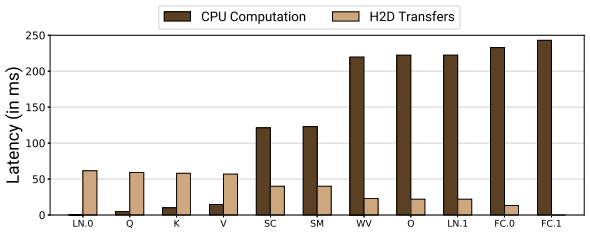
Just Accepted



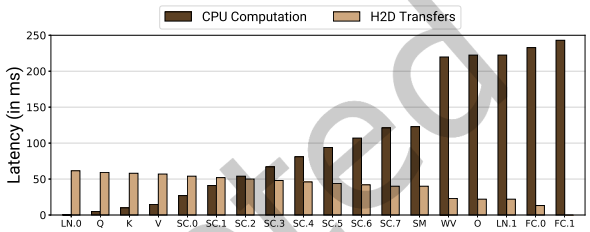
(a) Search result for OPT-30B model with 1x50 batch size.



(b) Search result for OPT-30B model with 1x50 batch size and divided `Scores(SC)` operation.



(c) Search result for OPT-30B model with 1x100 batch size and compression.



(d) Search result for OPT-30B model with 1x100 batch size, compression, and divided `Scores(SC)` operation.

Fig. 14. Single-point workload division result across all available split points. The x-axis labels show the operations in an OPT decoder layer, where each label denotes the split point. For example, `V` indicates that all operations up to and including `V` are computed by the CPU, and the remainder by the GPU.

Just Accepted

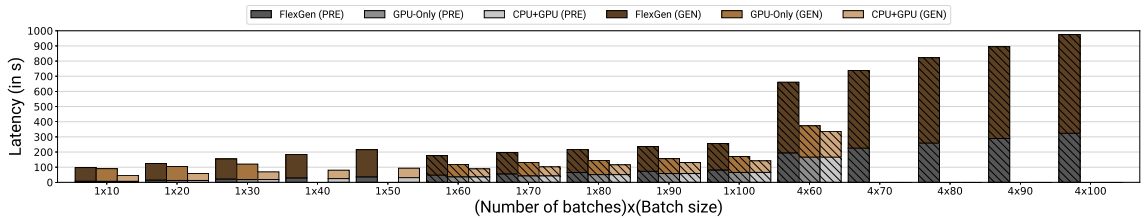
synchronous KV-cache transfers, which introduce increasing delays as batch size grows. Our implementation instead uses separate processes for inference and data transfers, effectively alleviating this bottleneck. During the generation phase, the performance gains stem from several improvements: the optimized attention layer implementation and an efficient AVX BF16 kernel together allow the dynamic division module to offload a larger portion of computation to the CPU, maximizing utilization of available hardware resources.

Figure 16 shows inference latency for the LLaMA-3-8B and LLaMA-2-34B models across various token generation lengths with 512 input tokens. Since FlexGen does not support the LLaMA model, CPU-GPU inference latency is compared against a GPU-only baseline. All LLaMA-based evaluations show smaller gains compared to OPT-30B due to the elimination of on-the-fly pinning procedures, but the workload distribution still provides a maximum of 51% latency reduction in non-compression workloads and 18% in compression workloads.

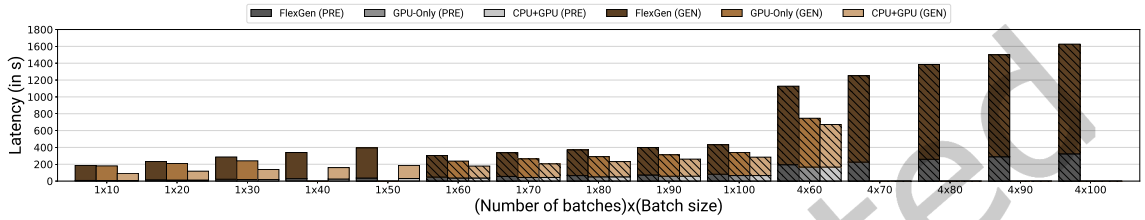
5.4 GPUDirect Storage

The use of pinned memory in our implementation introduces a fundamental trade-off between data transfer latency and overall computational throughput. In FlexGen, data is dynamically pinned during the inference stage. While this on-demand pinning incurs noticeable overhead and increases per-transfer latency, it allows larger batches to be created. For example, FlexGen can accommodate up to four batches of 100 inputs, whereas our implementation is limited to four batches of 60. In contrast, our implementation pre-allocates and pins memory at the start of inference, which reduces runtime pinning overhead but increases memory pressure, limiting the number of batches that can be supported concurrently.

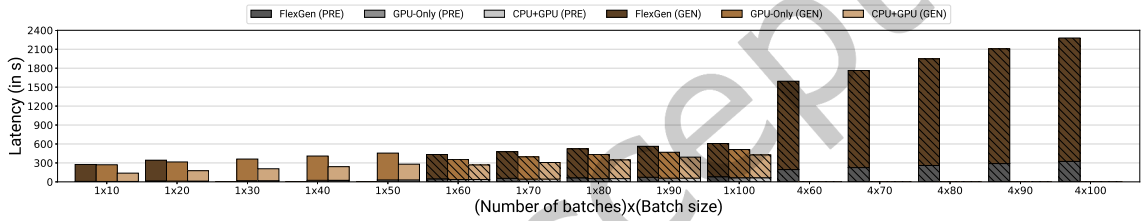
To overcome this limitation and enable larger batch configurations, our work leverages GDS to offload model weights to NVMe devices. This reduces the memory footprint in main memory, freeing space for additional input



(a) Latency comparison with various batch configurations for 32 token generations.



(b) Latency comparison with various batch configurations for 64 token generations.



(c) Latency comparison with various batch configurations for 128 token generations.

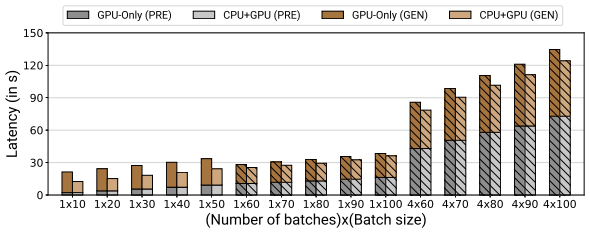
Fig. 15. OPT-30B model inference latency comparison with 512 input tokens. Generation phase latencies are stacked on top of the prefill phase latency. Bars with hatches denote application of compression, and empty bars show out-of-memory errors.

Just Accepted

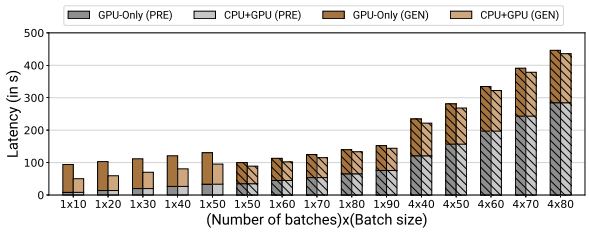
batches and intermediate activations, as shown in Figure 17. However, the performance benefits of NVMe offloading are workload-dependent. Without compression, NVMe-based transfers achieve performance comparable to the GPU+CPU baseline. With compression, however, NVMe performance degrades due to resource contention on the GPU: concurrent execution of host-to-device transfers, storage-to-device transfers, and GPU computation leads to contention for shared resources such as memory bandwidth and DMA engines, reducing the efficiency of each component and ultimately lowering overall throughput.

5.5 CPU-GPU Overlaps

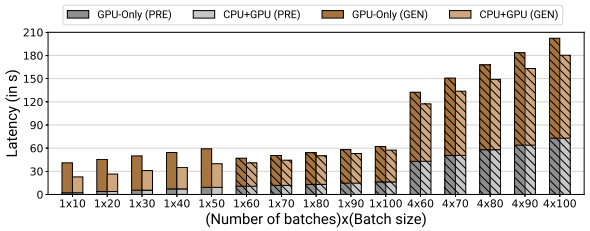
We also analyze CPU and GPU utilization within a decoder layer during distributed inference. As shown in Figure 18a, the single-batch workload achieves approximately 98% overlap between host-to-device transfers and CPU computation. In the multi-batch workload, shown in Figure 18b, only up to 70% of the decoder workload is handled by CPU computation, due to the high computational demands of the decoder layer that cannot be fully offloaded to the CPU. With operation splitting, however, the CPU overlap increases to up to 90% of the workload, as illustrated in Figure 18c. This improved utilization yields approximately a 5% reduction in generation phase latency compared to the CPU+GPU configuration without operation splitting, as shown in Figure 19.



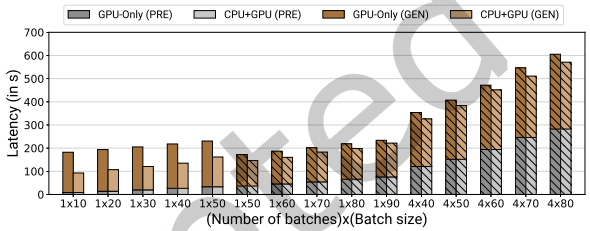
(a) LLaMA-3-8B latency comparison with various batch configurations for 32 token generations.



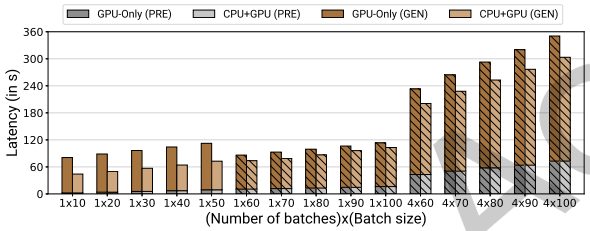
(b) LLaMA-2-34B latency comparison with various batch configurations for 32 token generations.



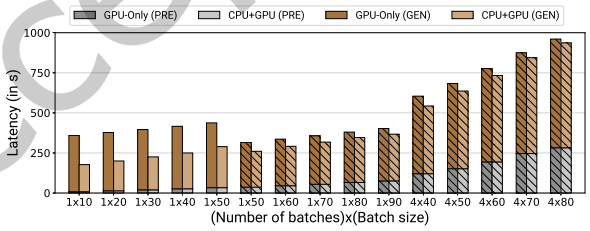
(c) LLaMA-3-8B latency comparison with various batch configurations for 64 token generations.



(d) LLaMA-2-34B latency comparison with various batch configurations for 64 token generations.



(e) LLaMA-3-8B latency comparison with various batch configurations for 128 token generations.



(f) LLaMA-2-34B latency comparison with various batch configurations for 128 token generations.

Fig. 16. LLaMA-3-8B and LLaMA-2-34B model inference latency comparison with 512 input tokens. Generation phase latencies are stacked on top of the prefill phase latency, and bars with hatches denote application of compression.

Just Accepted

5.6 LLM Optimizations

In this section, we evaluate the performance of the presented methodology across a variety of LLM optimization settings. Specifically, batch sizes ranging from 1 to the maximum achievable on our hardware were tested with a prefill phase of 512 tokens followed by generation of 32 tokens. The optimization strategies considered here are selected to assess the feasibility and effectiveness of CPU-GPU distributed inference.

To explore the generality of the methodology, experiments are conducted with two distinct models and optimization combinations. The first combines InfiniGen [22] with the OPT-30B model to evaluate KV-cache selection. The second applies ASC [20] to the OPT-30B model to evaluate operation reordering. Each configuration was implemented using open-sourced code or implementation details from the respective paper. Correctness was maintained in all cases except ASC, where only the operation ordering described in the paper could be reproduced.

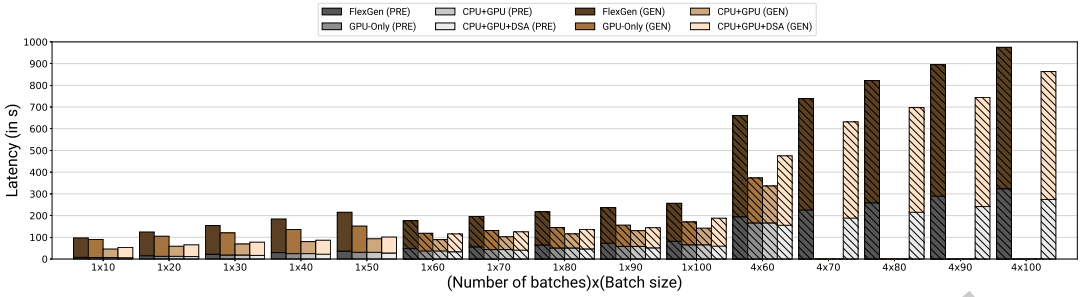
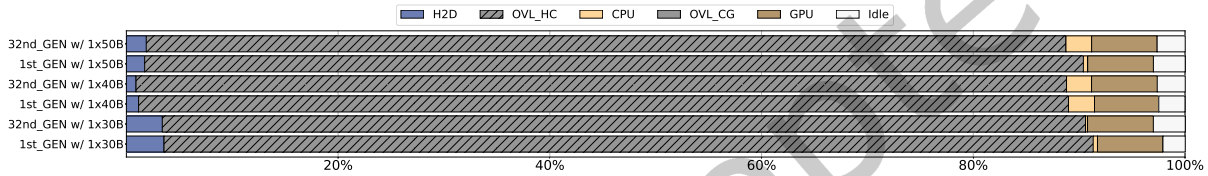
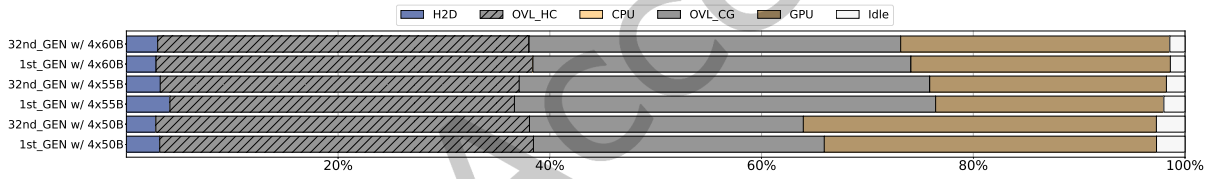


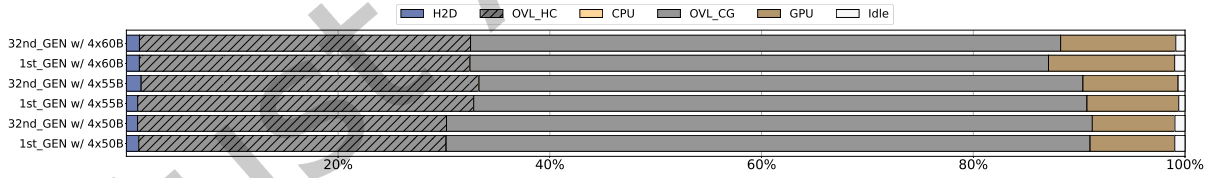
Fig. 17. OPT-30B model inference latency comparisons with 512 input tokens and 32 token generations. Generation phase latencies are stacked on top of the prefill phase latency. Bars with hatches denote application of compression, and empty bars show out-of-memory errors.



(a) Single-batch workload breakdown.



(b) Multi-batch workload breakdown.

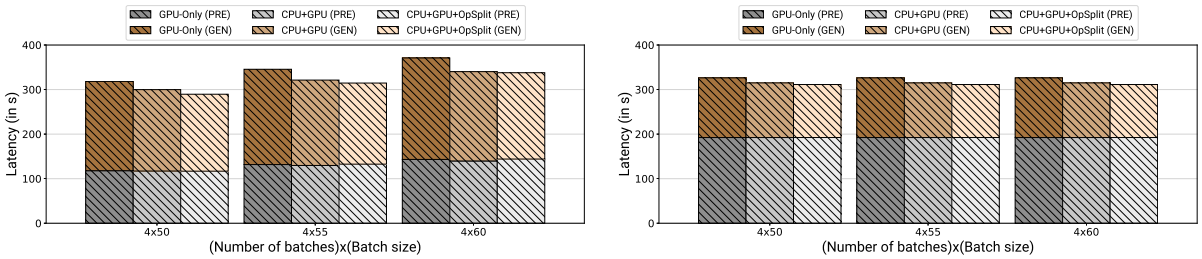


(c) Multi-batch workload with operation split.

Fig. 18. Analysis of generation phases using the OPT-30B model with 512 input tokens to generate 32 tokens. H2D denotes host-to-device transfers, OVL_HC the overlap between H2D and CPU computations, OVL_CG the overlap between CPU and GPU computations, and GPU denotes GPU computations.

Figure 20 presents single-batch workload results across all three optimization settings, comparing against the GPU-only baseline. A substantial latency reduction is observed: approximately 50% for non-compressed workloads and 20% for compressed workloads when using distributed CPU-GPU inference. These gains are purely from the generation phase, as the prefill phase shows no improvement over the GPU-only setting. The smaller gains under compression can be attributed to the increased GPU workload, which reduces the available

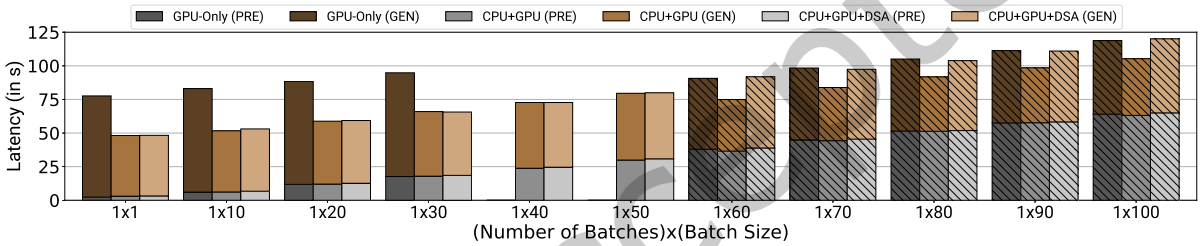
Just Accepted



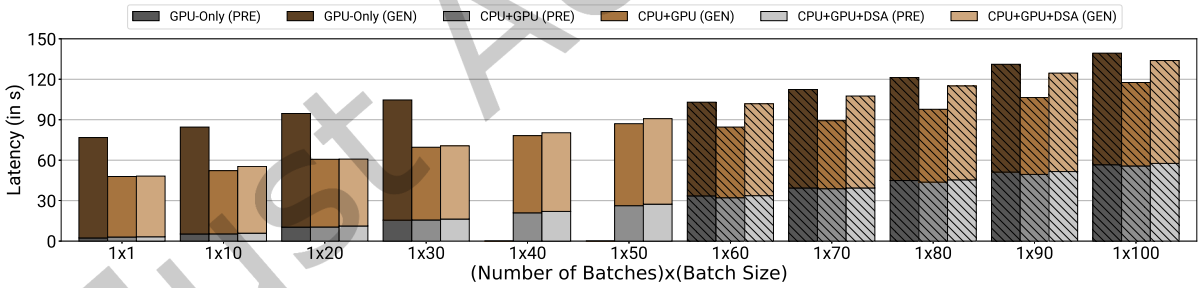
(a) OPT-30B inference latency comparisons.

(b) LLaMA-2-34B inference latency comparisons.

Fig. 19. Inference latency comparisons with 512 input tokens and 32 token generations. Generation phase latencies are stacked on top of the prefill phase latency, and bars with hatches denote application of compression. `OpSplit` denotes operation splitting.



(a) OPT-30B latency comparison on various single-batch workload configurations w/ InfiniGen.



(b) OPT-30B latency comparison on various single-batch workload configurations w/ ASC.

Fig. 20. OPT-30B model inference latency comparisons with 512 input tokens and 32 token generations combined with other LLM optimization techniques. Generation phase latencies are stacked on top of the prefill phase latency. Bars with hatches denote application of compression, and empty bars show out-of-memory errors.

Just Accepted

opportunities for CPU offloading. Overall, the results demonstrate that the dynamic division module achieves significant inference performance improvements across different model configurations and input sizes.

6 Related Work

Offloading Systems. ZeRO-Inference [4] pins the entire model weights in DRAM or NVMe and streams them layer-by-layer into the GPU for inference, leveraging layer prefetching to hide portions of the transfer latency. HuggingFace Accelerate [16] pursues the same goal of offloading weights to DRAM and NVMe to reduce VRAM pressure, but uses a hook mechanism rather than explicit streaming. FlexGen [40] leverages a zigzag scheduling strategy that divides batches into sets and processes one set at a time, rather than completing the full inference pass before moving to the next. While these throughput-oriented frameworks target maximum throughput, none of them considers CPU computation to alleviate data transfer bottlenecks.

DeepSpeed-FastGen [15] introduces a token-level scheduling strategy that decomposes long prompts into fixed-size chunks and fuses them with generation tokens to maintain uniform forward-pass sizes. ExeGPT [32] presents a constraint-aware LLM inference system that explicitly decouples the encoding and decoding phases and formulates schedule optimization as a monotonic search problem solved via branch-and-bound. However, neither DeepSpeed-FastGen nor ExeGPT considers CPU computation during LLM inference.

Both TwinPilots [46] and HeteGen [45] employ hybrid CPU-GPU parallelism by combining pipeline- and tensor-parallel methods during LLM inference to increase token generation throughput. However, TwinPilots targets high-end server environments comprising multiple nodes or GPUs, whereas our work focuses on a single commodity GPU system. HeteGen focuses on single-input tasks to demonstrate its effectiveness in reducing latency in offloading scenarios, whereas our work considers batch inference settings where latency is traded for higher throughput.

InfiniGen [22] and PowerInfer [41] both exploit sparsity during LLM inference, leveraging KV-cache sparsity and activation sparsity, respectively. KV-cache sparsity reduces the volume of data transfers during inference, while activation sparsity reduces the FLOPs executed on the GPU by offloading sparse weights and activations to the host CPU. The former is a GPU-centric approach complementary to our work, while the latter assigns the CPU only the computation of activated neurons, in contrast to our approach which offloads a full portion of the inference workload.

Finally, llama.cpp [14] is a widely adopted open-source framework that supports a broad range of hardware configurations for running LLMs. While not strictly an offloading framework, it supports offloading a configurable number of layers and KV-cache data to host DRAM during GPU-based inference. However, its approach executes the two devices sequentially rather than in an overlapped fashion, making it better suited for pipeline-parallel multi-user settings than for throughput-oriented workloads where the full model cannot fit in GPU VRAM.

Memory Management. vLLM [21] and Orca [47] use iterative scheduling and paged attention to eliminate memory waste and handle a larger number of concurrent requests. RadixAttention [50] introduces a modified scheduler that prioritizes requests matching cached prefixes to improve throughput. These works focus on efficient memory management at the model server level, rather than targeting a single commodity GPU system.

Attention Architectures. MQA [38] reduces the number of KV heads to one, thereby decreasing KV-cache size, while GQA [3] generalizes this by matching the number of KV heads to the number of training nodes [43]. Alternative approaches to MHA optimization include I/O-aware kernel design via FlashAttention [11] for accelerated attention computation, and operation reordering [20], which has been shown to halve cache size and improve throughput by up to 50% compared to standard implementations. The sliding window technique [17] bounds KV-cache size by constraining sequence lengths to a fixed window. These works motivated the design of the dynamic division module, as they give rise to varied workload characteristics during LLM inference.

Quantization and Compression. SqueezeLLM [19] applies sensitivity-based non-uniform quantization and filters out outliers to effectively compress model weights. SmoothQuant [44] achieves accuracy-preserving W8A8 quantization through per-channel smoothing. H2O [49] and Scissorhands [25] reduce KV-cache size by enforcing a token budget and evicting tokens once that budget is exceeded. The FastGen framework [13] builds on attention

Just Accepted

patterns and targets accuracy preservation by minimizing maximum approximation error. These techniques reduce model or KV-cache size and are compatible with our framework, provided the target hardware supports their computational requirements.

CPU-GPU Workload Distribution. Dopia [9] and OnTheFly [8] both dynamically distribute workloads between the CPU and GPU, but target integrated CPU-GPU architectures rather than systems with a dedicated discrete GPU. The work most closely related to ours is our prior work on CPU offloading [34], which differs primarily in that it does not support dynamic workload distribution through a profiling stage.

7 Conclusion

This paper demonstrates that strategically employing available CPU resources during host-to-device transfers can significantly reduce LLM inference latency on commodity hardware. Modern hardware capabilities, such as BF16 and FP16 compute support and GPUDirect Storage, combined with software-level optimizations, such as restructuring algorithms for more efficient CPU execution, offer further avenues for improvement. To relieve the user from manually identifying optimal workload division points, we present a dynamic workload division module that automatically determines the best split point configuration for a given target system. The proposed approach outperforms existing techniques across all tested configurations and two different LLM architectures, achieving a maximum speedup of over 2× compared to the state of the art.

Just Accepted

Acknowledgments

The authors thank the anonymous reviewers and the associate editor for their thorough feedback and constructive suggestions, which helped improve the quality of this paper.

This work was funded, in part, by the Korean National Research Foundation through grant 21A20151113068 (BK21 Plus for Pioneers in Innovative Computing - Dept. of Computer Science & Engineering, SNU), by the Ministry of Trade, Industry and Energy (MOTIE) and the Korea Evaluation Institute of Industrial Technology (KEIT) through grant 10077609, and by the EU Horizon program within the f-inference project (grant no. 101298393). ICT at Seoul National University provided research facilities for this study.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] David Ahmedt-Aristizabal, Mohammad Ali Armin, Simon Denman, Clinton Fookes, and Lars Petersson. 2021. Graph-based deep learning for medical diagnosis and analysis: past, present and future. *Sensors* 21, 14 (2021), 4758.

[3] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

[4] 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*. IEEE, 1–15.

[5] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934* (2020).

[6] Rishi Bommasani, Percy Liang, and Tony Lee. 2023. Holistic evaluation of language models. *Annals of the New York Academy of Sciences* 1525, 1 (2023), 140–146.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[8] Younghyun Cho, Florian Negele, Seohong Park, Bernhard Egger, and Thomas R Gross. 2018. On-the-fly workload partitioning for integrated CPU/GPU architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–13.

[9] Younghyun Cho, Jiyeon Park, Florian Negele, Changyeon Jo, Thomas R Gross, and Bernhard Egger. 2022. Dopia: online parallelism management for integrated CPU/GPU architectures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 32–45.
[10] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive Language Models beyond a Fixed-Length Context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2978–2988.
[11] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.
[12] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations*.
[13] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2023. Model Tells You What to Discard: Adaptive KV Cache Compression for LLMs. In *The Twelfth International Conference on Learning Representations*.
[14] Georgi Gerganov. 2023. llama.cpp Github. https://github.com/ggml-org/llama.cpp
[15] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. 2024. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671* (2024).
[16] HuggingFace. 2022. HuggingFace Accelerate. https://huggingface.co/docs/accelerate/index.
[17] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B.
[18] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).
[19] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. 2023. SqueezeLLM: Dense-and-Sparse Quantization. *arXiv preprint arXiv:2306.07629* (2023).
[20] Sowoong Kim, Eunyeong Sim, Youngsam Shin, YeonGon Cho, and Woongki Baek. 2024. Activation Sequence Caching: High-Throughput and Memory-Efficient Generative Inference with a Single GPU. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*. 78–90.
[21] 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*. 611–626.
[22] 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)*. 155–172.
[23] Wookey Lee, Jessica Jiwon Seong, Busra Ozlu, Bong Sup Shim, Azizbek Marakhimov, and Suan Lee. 2021. Biosignal sensors and deep learning-based speech recognition: A review. *Sensors* 21, 4 (2021), 1399.
[24] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023).
[25] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2024. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems* 36 (2024).
[26] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer Sentinel Mixture Models. In *International Conference on Learning Representations*.
[27] Avanika Narayan, Ines Chami, Laurel Orr, and Christopher Ré. 2022. Can Foundation Models Wrangle Your Data? *Proc. VLDB Endow.* 16, 4 (Dec. 2022), 738–746. https://doi.org/10.14778/3574245.3574258
[28] Shashi Narayan, Shay B. Cohen, and Mirella Lapata. 2018. Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Brussels, Belgium, 1797–1807. https://doi.org/10.18653/v1/D18-1206
[29] Nvidia. 2012. How to Optimize Data Transfers in CUDA C/C++. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.
[30] Nvidia. 2015. CUDA Streams Simplify Concurrency. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency.
[31] Nvidia. 2023. Magnum IO GPUDirect Storage. https://developer.nvidia.com/gpudirect-storage.
[32] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. 2024. Exegpt: Constraint-aware resource scheduling for llm inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 369–384.

Just Accepted

[33] 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*. 233–245.
[34] Daon Park, Sungbin Jo, and Bernhard Egger. 2023. Improving Throughput-oriented Generative Inference with CPUs. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*. 37–42.
[35] Yeonhong Park, Jake Hyun, SangLyul Cho, Bonggeun Sim, and Jae W. Lee. 2024. Any-precision LLM: low-cost deployment of multiple, different-sized LLMs. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 1607, 20 pages.
[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: an imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*.
[37] Roshan M Rao, Jason Liu, Robert Verkuil, Joshua Meier, John Canny, Pieter Abbeel, Tom Sercu, and Alexander Rives. 2021. MSA transformer. In *International Conference on Machine Learning*. PMLR, 8844–8856.
[38] Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150* (2019).
[39] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8815–8821.
[40] 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 *International Conference on Machine Learning*. PMLR, 31094–31116.
[41] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 590–606.
[42] Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. 2019. BERT4Rec: Sequential recommendation with bidirectional encoder representations from transformer. In *Proceedings of the 28th ACM international conference on information and knowledge management*. 1441–1450.
[43] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[44] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*. PMLR, 38087–38099.
[45] Zhao Xuanlei, Bin Jia, Haotian Zhou, Ziming Liu, Shenggan Cheng, and Yang You. 2024. HeteGen: Efficient Heterogeneous Parallel Inference for Large Language Models on Resource-Constrained Devices. *Proceedings of Machine Learning and Systems* 6 (2024), 162–172.
[46] 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*. 91–103.
[47] 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)*. 521–538.
[48] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).
[49] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2O: heavy-hitter oracle for efficient generative inference of large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*.
[50] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2023. Efficiently Programming Large Language Models using SGLang. *arXiv preprint arXiv:2312.07104* (2023).

Received 22 August 2025; revised 25 March 2026; accepted 2 May 2026

Just Accepted