FLEXINFER: FLEXIBLE LLM INFERENCE WITH CPU COMPUTATIONS

Seonjin Na [1] **Geonhwa Jeong** [2] **Byung Hoon Ahn** [3] **Aaron Jezghani** [1] **Jeffrey Young** [1] **Christopher J. Hughes** [4]
Tushar Krishna [1] **Hyesoon Kim** [1]

ABSTRACT

LLMs have demonstrated remarkable performance across various fields, prompting data centers to use high computation cost accelerators like GPUs and NPUs for model training and inference. However, the immense size of these models and key-value (KV) caches poses substantial memory capacity challenges. While offloading-based approaches utilize CPU memory to store model weights and KV caches—enabling deployment of models exceeding GPU memory capacity—they often suffer from performance degradation due to PCIe transfer bottlenecks. To address the performance limitations of existing offloading-based LLM inference in CPU and memory-limited single GPU systems, this paper proposes FlexInfer. FlexInfer uses a performance estimator to dynamically select the most appropriate execution policy for each phase—prefill and decode—based on hardware configurations and runtime parameters such as sequence length and batch size. Our evaluation results show that by selecting optimal policies for these phases, FlexInfer can significantly reduce end-to-end latency by 75% and 76% on average across two different server configurations, when compared to FlexGen, a state-of-the-art offload-based LLM inference technique.

1 INTRODUCTION

Transformer-based Large Language Models (LLMs) have demonstrated remarkable performance across various tasks, ushering in a new era of generative AI. These LLMs are now widely deployed in numerous applications, including text generation, machine translation, and beyond (Zhang et al., 2022; Touvron et al., 2023; Brown et al., 2020; Amodei et al., 2016). To meet the substantial computational demands of LLMs, companies have introduced specialized units in their existing processors, such as NVIDIA's Tensor Cores (Markidis et al., 2018) in GPUs, Intel's TMUL unit (Nassif et al., 2022) in CPUs, or even custom AI offload accelerators such as Meta's MTIA (Firoozshahian et al., 2023) or Google's TPU (Jouppi et al., 2023). Modern data centers are increasingly adopting these accelerators to efficiently handle LLM training and inference workloads (Hu et al., 2024b).

Despite these advancements, serving LLMs remains expensive, primarily due to their high memory requirements (Kwon et al., 2023; Sheng et al., 2023; Aminabadi et al., 2022). As model sizes continue to grow to achieve higher accuracy based on the scaling law (Kaplan et al., 2020), the number of parameters increases significantly, demanding substantial memory just to store the model weights. For instance, even with the FP16 data type, recent models such as GPT-175B (Brown, 2020) require 350 gigabytes of memory for storing weights alone, which can necessitate five expensive NVIDIA H100 GPUs. Furthermore, models used in industry, such as GPT-4 (Achiam et al., 2023) are known to be even larger (Achiam et al., 2023; Chowdhery et al., 2023; Dubey et al., 2024). In addition to model weights, the KV cache used to optimize repetitive computations in LLM inference also incurs significant memory challenges because the size of the KV cache scales with the sequence length, batch size, and number of model layers. For example, for OPT-66B with a sequence length of 4096 and a batch size of 32, the KV cache can consume as much as 288 GB of memory. Therefore, the memory capacity requirements for the combination of model weights and KV cache presents a considerable challenge for LLM inference serving systems.

To handle LLMs exceeding GPU memory capacity, recent studies (Sheng et al., 2023; Zhang et al., 2024; Aminabadi et al., 2022) have proposed *offloading-based LLM inference* serving techniques, where model weights, activations, and KV cache are stored in CPU memory and transferred one layer at a time via the CPU-GPU interconnect, typically PCIe, during the previous layer's computation on the GPU. While offloading enables handling larger models, it introduces significant PCIe data transfer overhead. With limited PCIe bandwidth, these transfers become a major bottleneck. To address this, prior studies have aimed to

[1]Georgia Institute of Technology [2]Meta [3]University of California, San Diego [4]Intel Labs. Correspondence to: Seonjin Na <seonjin.na@gatech.edu>.

Proceedings of the 8^{th} MLSys Conference, Santa Clara, CA, USA, 2025.

improve the performance of offloading-based LLM inference by reusing model weights across batches and overlapping data transfers with layer computation (Sheng et al., 2023; Aminabadi et al., 2022). However, since LLM layer computation times are much shorter than PCIe transfer times, these systems still face I/O bottlenecks. For instance, when using FlexGen (Sheng et al., 2023), a state-of-the-art offloading-based inference framework, we observe the exposed PCIe data transfer time accounted for about 96-98% of the total execution time. This demonstrates that while offloading-based LLM inference can handle larger models, addressing PCIe transfer overhead is crucial for achieving better performance. An alternative solution using multiple GPUs with model parallelism could address the memory constraints, but this approach substantially increases deployment costs, and well-optimized implementations are not readily available (Narayanan et al., 2021; 2019).

Recent studies demonstrate that CPUs with dedicated GEMM accelerators like Intel AMX can be effective at LLM inference, sometimes outperforming GPU-based solutions (Na et al., 2024; Kim et al., 2024). While most implementations offload execution to GPUs and leave CPU computation resources largely idle, these findings suggest that for LLM inference, performing *some* computation on the CPU is a viable option. Based on these observations, we explore efficient ways to utilize both CPU and GPU resources for LLM inference, considering how to partition work between a CPU and GPU and how to manage CPU-GPU communication. While modern CPUs offer significant memory capacity and dedicated GEMM hardware, their computational throughput lags GPUs. Therefore, careful workload distribution based on phase characteristics is essential for optimal performance.

In this paper, we propose FlexInfer, a system designed to minimize LLM inference latency in memory-constrained system environments by effectively leveraging both CPU and GPU computation. FlexInfer offers three key execution policy options: (1) CPU computation, (2) offloading-based GPU execution, and (3) CPU-GPU static partitioning execution. While this study focuses on these three policies, the framework can be extended to incorporate additional execution strategies as needed. The system dynamically selects the optimal execution policy using an estimator that considers the system's CPU and GPU performance, CPU-GPU communication capability, as well as user-specified parameters such as input sequence length, output sequence length, and batch size. The estimator considers the prefill and decode phases of LLM inference separately. It predicts and uses the best policy for each phase, optimizing performance in different stages of the inference process.

Our experimental results show that FlexInfer significantly improves performance over existing state-of-the-art offloading techniques, reducing average end-to-end latency by 75% and 76% on two different server configurations. These results demonstrate that our approach performs consistently well across different hardware environments.

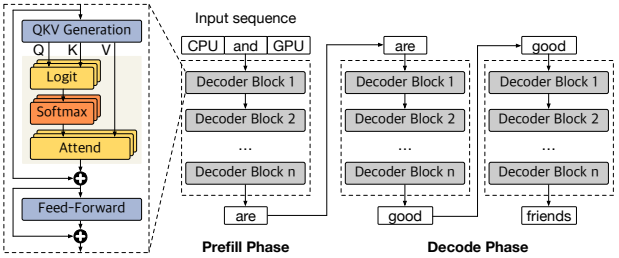


Figure 1. Transformer-based LLM architecture.

We summarize the contributions of this paper as follows:

- We identify and analyze key performance bottlenecks in existing GPU offloading-based approaches as they relate to the prefill and decode phases of modern LLMs.
- We explore the potential of CPU computation in LLM inference and identify trade-offs between various execution policies for hybrid CPU-GPU execution.
- We propose FlexInfer, a new approach that uses an estimator to select execution policies for the prefill and decode phases based on phase-specific characteristics.
- We demonstrate that FlexInfer can significantly reduce LLM inference latency by 75% and 76% on two different server configurations across various models.

2 Background

2.1 Large Language Model (LLM) Inference

LLM architecture: Recent LLMs (Brown, 2020; Dubey et al., 2024) employ decoder-based transformer models that generate tokens in an autoregressive manner, as illustrated in Figure 1. The transformer architecture includes multiple decoder blocks (Vaswani, 2017) with two key components: a Multi-Head Attention cell that uses an attention mechanism to understand relationships between tokens and a Feed-Forward layer that performs nonlinear transformations to refine the sequence representations.

LLM inference phases and characteristics: LLM inference consists of two phases, prefill and decode, as illustrated in Figure 1. In the prefill phase, the model takes in the input prompt and generates one new token. This phase is computationally intensive as it requires computing hidden representations for the entire input sequence simultaneously with an attention mechanism that has quadratic complexity over sequence length.In the decode phase, LLMs iteratively generate one token at a time until they reach a predefined sequence length or end-of-sequence token. The decode phase generally involves KV caching, an optimiza-

Schemes	Tensor Offload	CPU Computation	Flexible Execution Policy
Accelerate (Wolf, 2019)	✓	✗	✗
DeepSpeed-ZeRO (Aminabadi et al., 2022)	✓	✗	✗
FlexGen (Sheng et al., 2023)	✓	▲	✗
HeteGen (Xuanlei et al., 2024)	✓	✓	✗
FlexInfer (Ours)	✓	✓	✓

(▲: Only used for attention score operation)

Table 1. Comparisons with prior offloading techniques

tion to eliminate redundant computation of key-value matrix contents across iterations. This typically makes the decode phase memory-bound.

2.2 Offloading-based LLM Inference

The large memory footprint of LLM inference, caused by model weights and the KV cache, often exceeds the memory capacity of recent GPUs. To address this limitation, several offloading techniques have been proposed (Wolf, 2019; Aminabadi et al., 2022; Sheng et al., 2023; Xuanlei et al., 2024). Table 1 compares the characteristics of previous offloading-based LLM inference techniques. These techniques reduce the GPU memory requirements by storing model weights and the KV cache in CPU memory or on disk. While enabling large models to run on modest capacity GPUs, these approaches introduce significant performance overhead due to data transfers over the slow PCIe bus. For example, despite attempts to overlap transfers with computation, PCIe bandwidth limitations still result in substantial transfer time, underutilizing GPU resources.

To mitigate performance degradation due to I/O bottleneck, recent research has explored incorporating CPU computation into offloading-based LLM inference. For instance, FlexGen (Sheng et al., 2023) partially leverages CPU computation for attention score calculations in the decode phase, while HeteGen (Xuanlei et al., 2024) employs CPU-GPU tensor parallelism—using CPU-based operations (e.g., linear layers)—to exploit both CPU and GPU while hiding data transfer overhead. However, these solutions often rely on limited CPU utilization or adopt fixed execution policies that do not consider varying runtime parameters and hardware configurations. In particular, the benefits of CPU-GPU tensor parallelism can diminish at larger batch sizes or with longer input sequences, where CPU throughput becomes a bottleneck.

In contrast, FlexInfer dynamically determines the most suitable execution strategy for each phase, taking into account runtime parameters and hardware configurations. This flexible execution policy more effectively balances work between CPU and GPU, reducing PCIe transfer overhead and minimizing offloading-based LLM inference latency on GPUs with limited memory.

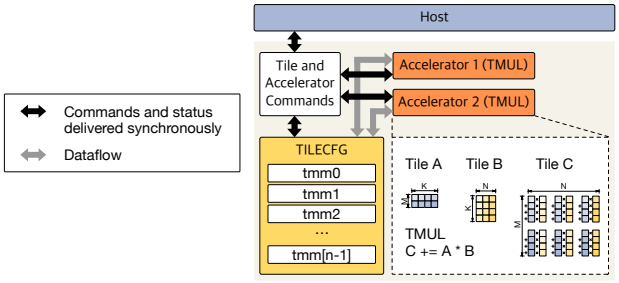
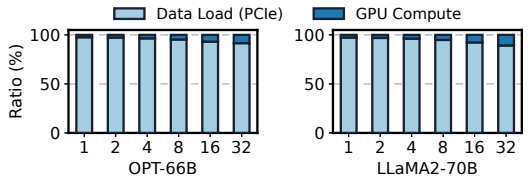
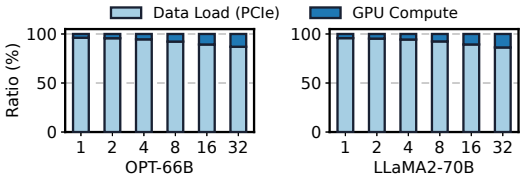


Figure 2. Overview of Intel AMX architecture (Intel, 2023).



(a) Execution time breakdown result on A100 GPU



(b) Execution time breakdown result on H100 GPU

Figure 3. Offloading-based LLM inference execution time breakdown analysis for larger models (OPT-66B and LLaMA-2 70B) on A100 and H100 GPU using FlexGen (Sheng et al., 2023)

2.3 Matrix Multiplication Accelerators on CPUs

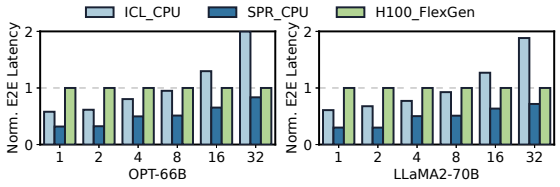
Recent CPUs include dedicated matrix multiplication hardware to meet the demands of machine-learning applications. Leading CPU vendors have introduced specialized hardware extensions: Intel's Advanced Matrix Extensions (AMX) (Nassif et al., 2022), IBM's Matrix Multiply Assist (MMA) (de Carvalho et al., 2022), and Arm's Scalable Matrix Extension (SME) (Weidmann, 2021). Intel's AMX, introduced with Sapphire Rapids (SPR) processors (Nassif et al., 2022), consists of two key components shown in Figure 2: Tile Matrix Multiply Units (TMUL), which accelerate matrix operations on BF16 and INT8 data formats (Nassif et al., 2022; Jeong et al., 2021; 2023), and Tiles, which are 2D registers for matrix storage.

3 Motivation

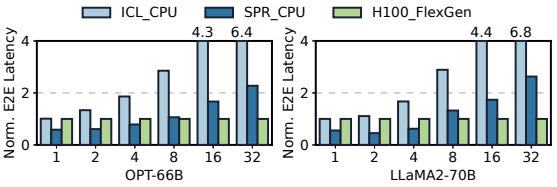
3.1 Offloading LLM Inference Analysis

To analyze offloading-based LLM inference performance, we conduct experiments on two different servers, with different generations of CPUs (ICL/SPR), GPUs (A100/H100), and PCIe interconnects (PCIe 4.0/5.0). Detailed server configurations are described in Section 6.

Execution time breakdown result: Figure 3 shows



(a) Execution time comparison when the input length is set to 128 and the output length to 32.



(b) Execution time comparison when the input length is set to 1024 and the output length to 32.

Figure 4. End-to-end latency comparison with varying batch sizes for OPT-66B and LLaMA2-70B models. Figure (a) shows the results for an input length of 128, while Figure (b) presents the results for an input length of 1024.

the execution time breakdown of running OPT-66B and LLaMA2-70B models using FlexGen on A100 and H100 GPUs with an input length of 1024 while generating outputs of length 32, across batch sizes ranging from 1 to 32. In this setup, the model weights and KV cache are stored in CPU memory, and the system overlaps the computation of the current layer with the loading of the next layer's weights and KV cache via PCIe to minimize idle GPU time. Using NVIDIA's Nsight profiling tool (NVIDIA, 2025), we analyze the detailed execution timeline and break down the total execution time into two components: exposed PCIe data transfer time that could not be fully overlapped (Data Load) and GPU computation time (GPU compute).

We observe that as batch size increases, the amount of computation grows and the system attempts to overlap computation with communication. However, data transfer time still dominates the total execution time across all configurations. On the A100 GPU, PCIe data transfers dominate the execution time, ranging from 91.6% to 97.6% of the total runtime for the OPT-66B model and from 89.2% to 97.3% for the LLaMA2-70B model. Similarly, even on the H100 GPU server with higher PCIe bandwidth, PCIe data transfers still consume from 87% to 96.2% of the execution time for the OPT-66B model and from 86.3% to 95.8% for the LLaMA2-70B model. These results highlight that even with state-of-the-art GPU hardware, the limited PCIe bandwidth incurs a severe bottleneck for offloaded LLM inference.

3.2 Opportunities for Exploiting CPU Computation

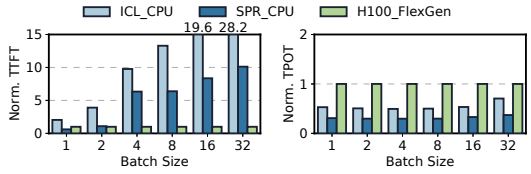
Given the PCIe data transfer overhead in offloading-based LLM inference, the emergence of GEMM accelerators in recent CPUs, and the larger memory capacity compared to GPU, we investigate the potential of exploiting CPU computation in LLM inference. We conduct experiments using two Intel Xeon CPUs: the 6454S (SPR) with AMX support and the 8352Y (ICL) without AMX. For CPU-based LLM inference, we use Intel Extension for PyTorch (Intel, 2020), which provides CPU-specific kernel optimizations. These include AMX-based optimizations tailored for SPR CPUs and AVX-512 optimizations for ICL CPUs.

End-to-end latency comparison: Figure 4 compares the execution time of LLM inference directly on the CPU with FlexGen for OPT-66B and LLaMA2-70B, normalized to FlexGen running on the H100 GPU. As batch size and input sequence length increase, the computation required for both prefill and decode phases grows, resulting in a performance decline for the CPU compared to the GPU, since compute throughput is much higher on the GPU. This is especially noticeable with larger batch sizes or longer input sequences, underscoring the CPU's limitations in these scenarios.

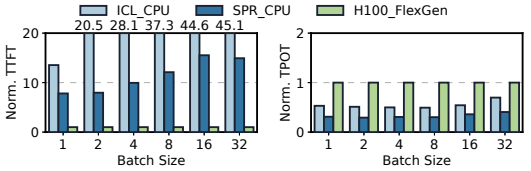
As shown in Figure 4(a), for short inputs, the ICL CPU outperforms GPU-based FlexGen at small batch sizes (up to 8 for OPT-66B and LLaMA-70B), with execution times ranging from 5% to 42.1% faster for OPT-66B and from 7.4% to 39.2% faster for LLaMA-70B. However, for larger batch sizes, the ICL CPU is slower than the GPU, by $1.3\times$ to $2\times$ for OPT-66B and $1.3\times$ to $2\times$ for LLaMA2-70B. The SPR CPU with AMX shows superior performance across most batch sizes, with execution times ranging from 16.5% to 68.2% faster for OPT-66B and from 28.3% to 70% faster for LLaMA2-70B.

Figure 4(b) shows the results for an input length of 1024 and an output length of 32. With this increased input length, the ICL CPU experiences a slowdown ranging from $1.3\times$ to $6.4\times$ for the OPT-66B model and from $1.1\times$ to $6.6\times$ for the LLaMA2-70B model compared to GPU offloading, starting from batch size 2. For the SPR CPU, execution time begins to increase from batch size 8 onward, with slowdowns ranging from $1.1\times$ to $2.3\times$ for OPT-66B and $1.3\times$ to $2.6\times$ for LLaMA2-70B, highlighting the impact of larger input sequence lengths on CPU performance. These results show that while CPUs have potential for handling models that exceed GPU memory capacity, their lower computation throughput limits their performance, especially with larger batch sizes and longer input sequences.

Prefill and decode phase comparison: As discussed in Section 2, LLM inference consists of two phases: prefill and decode. We evaluate phase-specific performance using two standard metrics (Gim et al., 2024; Patel et al., 2024): Time to First Token (TTFT) for the prefill phase and Time



(a) TTFT and TPOT comparison when the input length is set to 128 and the output length to 32.



(b) TTFT and TPOT comparison when the input length is set to 1024 and the output length to 32.

Figure 5. TTFT and TPOT comparison for OPT-66B.

Per Output Token (TPOT) for the decode phase.

Figure 5 presents TTFT and TPOT measurements for the OPT-66B model normalized to FlexGen on H100, varying input length and batch size, while keeping the output length fixed at 32. The prefill phase is computationally intensive as it processes all input tokens, while the decode phase is memory-bound, processing one new token per sequence at a time using the KV cache. Despite transferring weights and KV caches from CPU memory, the TTFT is much better for the GPU execution with offloading in most cases due to its higher computational throughput. The ICL CPU sees a slowdown of $2\times$ to $28.2\times$ for an input length of 128 and a slowdown of $13.5\times$ to $45.1\times$ for an input length of 1024. The SPR CPU shows 39% faster TTFT due to its dedicated hardware for GEMM operations when input length is 128 with batch size 1, but it is still slower by $1.1\times$ to $10.1\times$ for an input length of 128 from batch size 2, and $7.8\times$ to $15.5\times$ for an input length of 1024.

For the decode phase, CPU execution benefits from having all required data (weights, activations, and KV cache) in memory, while GPU execution with offloading, despite overlapping PCIe transfers with computation, still experiences significant overhead due to data movement during each layer's processing. As a result, as shown in Figure 5, both the ICL CPU and SPR CPU show lower TPOT than the H100 GPU. For the ICL CPU, TPOT is 29.5% to 50.5% lower for an input length of 128 and 30.3% to 50.7% lower for an input length of 1024. The SPR CPU demonstrates even better performance, with TPOT values 62.9% to 70.3% lower for an input length of 128 and 59.4% to 70.8% lower for an input length of 1024.

In summary, GPUs outperform CPUs in the compute-intensive prefill phase despite data transfer overhead, whereas CPUs achieve better performance in the decode phase due to lower computational demands of the decode phase and the lack of PCIe data transfers.

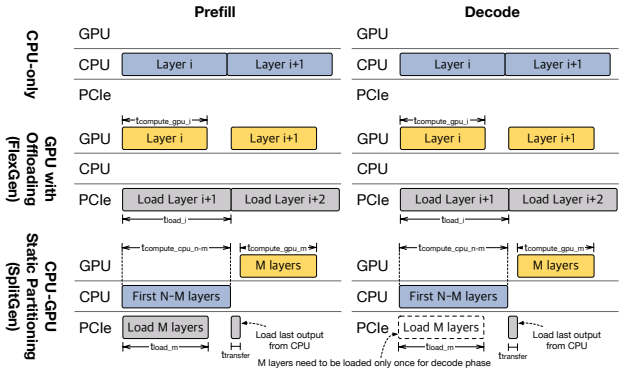


Figure 6. Timeline of different execution policies.

4 Baseline Execution Policies in CPU-GPU Systems

In this section, we outline the various execution policies that might be used with our proposed FlexInfer approach in a CPU-GPU system. As shown in Figure 6, a policy determines the distribution of model layers between the CPU and GPU. The effectiveness of a policy depends on several factors such as input length, batch size, and the computational throughput of the CPU and GPU.

4.1 CPU-only

In the CPU-only execution policy, all computation is performed on the CPU, utilizing its large memory capacity to store model weights, activations, and the KV cache. Given the CPU's relative weakness in computation throughput, this policy favors memory-bound scenarios, i.e., larger models and longer output sequences. Furthermore, this policy is more effective for the decode phase, which is less compute intensive, as discussed in Section 3.1.

4.2 GPU with Tensor Offloading (FlexGen)

GPU with tensor offloading involves transferring the model weights and KV cache that reside in CPU memory to the GPU via PCIe for computation. In this work, we use FlexGen (Sheng et al., 2023) as our offloading baseline. FlexGen overlaps the execution of layer i ($t_{\text{compute\_gpu\_i}}$) and the loading of layer $i+1$ ($t_{\text{load\_i}}$) so that the latency per layer is $max(t_{\text{compute\_gpu\_i}}, t_{\text{load\_i}})$ as illustrated in Figure 6. If data transfers are slower than the computation on the GPU ($t_{\text{compute\_gpu\_i}} < t_{\text{load\_i}}$), they will be the bottleneck and can actually dominate execution time. This is more likely when computation is small, e.g., in the decode phase, as discussed in Section 3.2.

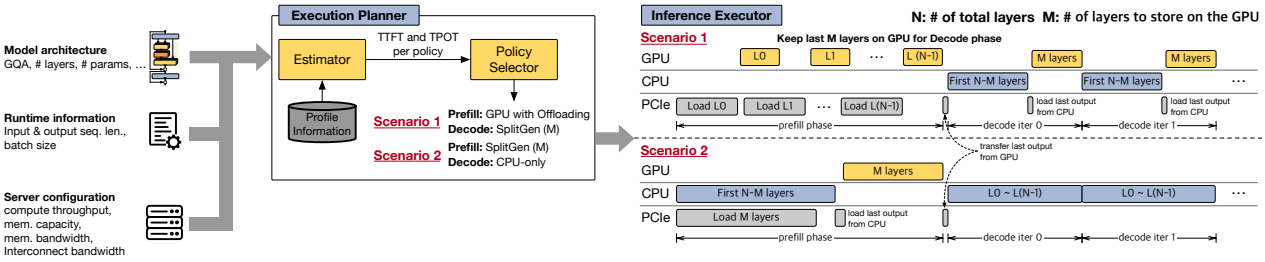


Figure 7. Overview of FlexInfer.

4.3 CPU-GPU Static Partitioning (SplitGen)

In addition to the CPU-only and GPU with offloading approaches, we explore a hybrid execution policy we call "SplitGen" where the model is split statically between CPU and GPU. As shown in Figure 6, with this execution policy the first $N - M$ layers are assigned to the CPU and the remaining M layers are assigned to the GPU.

To determine M, SplitGen considers the GPU memory size and tries to fit as many layers as possible on the GPU. To minimize the overall latency, during the prefill phase, SplitGen overlaps the CPU execution of the first $N - M$ layers ($t_{\text{compute\_cpu\_n-m}}$) and the loading of the final M layers ($t_{\text{load\_m}}$) as illustrated in the figure. The prefill phase latency is $max(t_{\text{compute\_cpu\_n-m}}, t_{\text{load\_m}}) + t_{\text{transfer}} + t_{\text{compute\_gpu\_m}}$ where $t_{\text{compute\_gpu\_m}}$ is the execution time of the last M layers on the GPU and t_{transfer} is the time to transfer the output of the last layer computed on the CPU to the GPU. During the decode phase, we do not need to re-transfer the last M layers to the GPU because SplitGen keeps them in GPU memory without eviction. As such, this is especially helpful in prefill phases with short input sequences and in the decode phase as it minimizes CPU-GPU communication.

4.4 Limitations of Homogeneous Execution Policies in LLM inference

LLM inference performance can be significantly affected by runtime parameters such as input length, output length, and batch size, as well as hardware capabilities including CPU/GPU compute throughput and memory/interconnect bandwidth. Depending on these factors, different execution strategies—CPU-only, GPU with offloading, and CPU-GPU static partitioning—can yield varying performance. To minimize LLM inference latency, it is crucial to dynamically select execution policies that consider both the distinct characteristics of prefill and decode stages, as well as runtime parameters and hardware configurations.

Variables	Description
N	Number of decoder blocks (layers) in model
M	Number of layers executed on the GPU
TH_{Device}	Compute throughput of the device
$\text{BW}_{\text{Device}}$	Memory bandwidth of the device
BW_{ICN}	interconnect bandwidth
$\text{Eff}_{\text{Device}}$	Efficiency correction factor for the device
$\text{C}_{\text{Prefill}}$	Computation amount per decoder block in prefill phase
$\text{D}_{\text{Prefill}}$	Data amount per decoder block in prefill phase
C_{Decode}	Computation amount per decoder block in decode phase
D_{Decode}	Data amount per decoder block in decode phase
$\text{D}_{\text{KVCache}}$	Data amount for KV cache per decoder block
D_{Output}	Data amount for output per decoder block
$\text{D}_{\text{CPU}\rightarrow\text{GPU}}$	Data transferred from CPU to GPU per decoder block

Table 2. Notations for TTFT and TPOT estimation

5 FlexInfer Design

5.1 Overview

To overcome the limitations of using a single execution policy for inference and to effectively minimize LLM inference latency under varying runtime and hardware configurations, we propose FlexInfer. As shown in Figure 7, FlexInfer comprises two main components: the Execution Planner and the Inference Executor. The Execution Planner selects a policy for each of the prefill and decode phases by analyzing the model architecture, runtime information, and server hardware configurations. The Inference Executor performs inference according to the chosen policies.

5.2 Execution Planner

The Execution Planner determines the best execution policies for the prefill and decode phases based on the specific LLM, input length, output length, batch size, and hardware configuration (i.e., CPU and GPU compute throughput, memory bandwidth, and CPU-GPU interconnect bandwidth). While the Execution Planner currently focuses on three baseline policies—CPU-only, GPU with offloading, and SplitGen—for the prefill and decode phases, it can be extended to incorporate additional execution strategies when necessary.

We develop an estimator that uses an analytical model to predict TTFT and TPOT for each execution policy. The estimator first analyzes the workload characteristics, including the amount of computation required for each layer's

decode block (e.g., Attention and Feed-Forward Network (FFN) modules), the memory footprint, and the volume of CPU-GPU data transfers. It then combines that workload information with the system's hardware properties, such as the compute throughput and memory bandwidth of each component, to predict TTFT and TPOT for each policy. Additionally, to account for the gap between theoretical and achievable hardware performance (such as GEMM acceleration hardware not reaching full utilization), we incorporate profiling results from an offline phase. These results provide insights into a component's actual efficiency across different input dimensions, allowing the estimator to more accurately handle variations in throughput.

Table 2 lists the notation used in TTFT and TPOT estimation. Using the notation defined in Table 2, we estimate TTFT and TPOT for the three execution policies: CPU-only execution, GPU with offloading, and SplitGen. Equations 1-3 demonstrate how to calculate the estimated TTFT and TPOT for each execution policy. To bridge the gap between theoretical and actual hardware performance, we incorporate component-specific efficiency factors (e.g., $\text{Eff}_{\text{CPU\_Comp}}$, $\text{Eff}_{\text{GPU\_Mem}}$, Eff_{ICN}), which are derived from offline profiling.

Equation 1 represents the CPU-only execution where all computations are performed on the CPU. Equation 2 shows the GPU offloading where the entire model is executed on the GPU by overlapping the data transfer with the layer computation. Finally, Equation 3 describes the estimated TTFT and TPOT for SplitGen policy, where M is the estimated maximum number of layers that loaded on the GPU and the first $(N-M)$ layers are executed on the CPU while the final M layers are processed on the GPU.

$$\begin{aligned}\text{TTFT} &= \left(\frac{\text{C}_{\text{Prefill}}}{TH_{\text{CPU}} \times \text{Eff}_{\text{CPU\_Comp}}} + \frac{\text{D}_{\text{Prefill}}}{\text{BW}_{\text{CPU}} \times \text{Eff}_{\text{CPU\_Mem}}}\right) \times N \\ \text{TPOT} &= \left(\frac{\text{C}_{\text{Decode}}}{TH_{\text{CPU}} \times \text{Eff}_{\text{CPU\_Comp}}} + \frac{\text{D}_{\text{Decode}} + \text{D}_{\text{KVCache}}}{\text{BW}_{\text{CPU}} \times \text{Eff}_{\text{CPU\_Mem}}}\right) \times N\end{aligned} \tag{1}$$

$$\begin{aligned}\text{TTFT} &= \max\left(\frac{\text{C}_{\text{Prefill}}}{TH_{\text{GPU}} \times \text{Eff}_{\text{GPU\_Comp}}} + \frac{\text{D}_{\text{Prefill}}}{\text{BW}_{\text{GPU}} \times \text{Eff}_{\text{GPU\_Mem}}}, \frac{\text{D}_{\text{CPU}\to\text{GPU}}}{\text{BW}_{\text{ICN}} \times \text{Eff}_{\text{ICN}}}\right) \times N \\ \text{TPOT} &= \max\left(\frac{\text{C}_{\text{Decode}}}{TH_{\text{GPU}} \times \text{Eff}_{\text{GPU\_Comp}}} + \frac{\text{D}_{\text{Decode}} + \text{D}_{\text{KVCache}}}{\text{BW}_{\text{GPU}} \times \text{Eff}_{\text{GPU\_Mem}}}, \frac{\text{D}_{\text{CPU}\to\text{GPU}}}{\text{BW}_{\text{ICN}} \times \text{Eff}_{\text{ICN}}}\right) \times N\end{aligned} \tag{2}$$

$$\begin{aligned}\text{TTFT} &= \max\left(\frac{\text{C}_{\text{Prefill}}}{TH_{\text{CPU}} \times \text{Eff}_{\text{CPU\_Comp}}} + \frac{\text{D}_{\text{Prefill}}}{\text{BW}_{\text{CPU}} \times \text{Eff}_{\text{CPU\_Mem}}}, \frac{\text{D}_{\text{CPU}\to\text{GPU}}}{\text{BW}_{\text{ICN}} \times \text{Eff}_{\text{ICN}}}\right) \times (N-M) \\ &\quad + \frac{\text{D}_{\text{Output}}}{\text{BW}_{\text{ICN}} \times \text{Eff}_{\text{ICN}}} + \left(\frac{\text{C}_{\text{Prefill}}}{TH_{\text{GPU}} \times \text{Eff}_{\text{GPU\_Comp}}} + \frac{\text{D}_{\text{Prefill}}}{\text{BW}_{\text{GPU}} \times \text{Eff}_{\text{GPU\_Mem}}}\right) \times M \\ \text{TPOT} &= \underbrace{\left(\frac{\text{C}_{\text{Decode}}}{TH_{\text{CPU}} \times \text{Eff}_{\text{CPU\_Comp}}} + \frac{\text{D}_{\text{Decode}}}{\text{BW}_{\text{CPU}} \times \text{Eff}_{\text{CPU\_Mem}}}\right) \times (N-M)}_{\text{given } M \text{ layers are already loaded on GPU}} \\ &\quad + \frac{\text{D}_{\text{Output}}}{\text{BW}_{\text{ICN}} \times \text{Eff}_{\text{ICN}}} + \left(\frac{\text{C}_{\text{Decode}}}{TH_{\text{GPU}} \times \text{Eff}_{\text{GPU\_Comp}}} + \frac{\text{D}_{\text{Decode}} + \text{D}_{\text{KV\_Cache}}}{\text{BW}_{\text{GPU}} \times \text{Eff}_{\text{GPU\_Mem}}}\right) \times M\end{aligned} \tag{3}$$

5.3 Inference Executor

Once the Execution Planner selects an execution policy for each phase, the Inference Executor is responsible for carrying out the inference process. The Executor manages the work done by the CPU and GPU, as well as communication between them. It ensures that each phase is executed according to the selected policy, whether the phases use the same policy or different policies. It also manages data transfers between CPU memory and GPU memory, ensuring that each phase has the necessary data to complete its computation. Additionally, The Executor handles synchronization between the CPU and GPU, ensuring that data transfers occur in parallel with computation to minimize idle time.

For instance, if the Execution Planner selects GPU with offloading for the prefill phase and the SplitGen policy for the decode phase, as shown in Figure 7 (Scenario 1), the Executor overlaps layer computation with the loading of weights for the next layer during the prefill phase. In the decode phase with SplitGen, the Execution Planner executes the first N-M layers on the CPU, transfers the final output activation from the CPU to the GPU, and runs the last M layers on the GPU. To prepare for this, the Executor stores the last M layers in GPU memory during the prefill phase, along with any key-value cache generated for these M layers, while all other layers' weights and key-value caches are stored in CPU memory.

5.4 Implementation

We extended the HuggingFace (Wolf, 2019) library to support all of our execution policies and to allow different policies for the prefill and decode phases. Additionally, we modified the Intel Extension for PyTorch (IPEX) (Intel, 2020), which originally supports only CPU execution, to enable selective execution of certain layers or operations (e.g., Attention and FFN in the decode block) on the GPU. For communication, we overlap CPU-GPU data transfers with computation using a separate CUDA stream. For layers executed on the CPU, we leverage IPEX-optimized kernels to improve performance. Currently, FlexInfer supports the LLaMA (Touvron et al., 2023; Dubey et al., 2024) and OPT (Zhang et al., 2022) model families. However, it can be extended to support other models compatible with the HuggingFace library and the IPEX framework.

	Server 1	Server 2
CPU	2× Xeon 8352Y	2× Xeon 6454S
GPU	NVIDIA A100	NVIDIA H100
CPU Throughput (BF16)[1]	9.0 TFLOPS	144.2 TFLOPS
CPU Memory	256 GB	512 GB
CPU Memory Bandwidth [2]	311.8 GB/s	407.8 GB/s
GPU Throughput (BF16)[3]	312 TFLOPS	756 TFLOPS
GPU Memory	40 GB	80 GB
GPU Memory Bandwidth[4]	1299.9 GB/s	1754.4 GB/s
CPU-GPU Interconnect	PCIe 4.0, 32 GB/s	PCIe 5.0, 64 GB/s

Table 3. Server configurations.

6 EVALUATION

6.1 Evaluation Methodology

Experimental setup: We evaluate the performance of FlexInfer on two different servers: Server 1 with an Ice-Lake CPU (no AMX) and A100-40GB GPU connected via PCIe 4.0 (32 GB/s per direction), and Server 2 with a Sapphire Rapids CPU (with AMX) and H100-80GB GPU connected via PCIe 5.0 (64 GB/s per direction). Table 3 provides the details for these servers. For code running on the CPU, we utilize the Intel Extension for PyTorch framework (Intel, 2020) to execute kernels optimized for each CPU.

Models: We use OPT (Zhang et al., 2022) (30B, 66B), LLaMA-2 (Touvron et al., 2023) (70B), and LLaMA-3 (Dubey et al., 2024) (70B) models. Unless otherwise specified, we use an input sequence length of 512, an output sequence length of 32, and vary the batch size from 1 to 32. In Sections 6.4 and 6.5, we further explore the performance impact of varying the input and output sequence lengths.

Metrics: To measure performance, we use widely used metrics from prior studies (Kwon et al., 2023; Sheng et al., 2023; Zhang et al., 2024): (1) end-to-end latency (E2E latency), the total time to generate the output sequence; (2) the time to first token (TTFT), the time to generate the first token; and (3) the time per output token (TPOT), the average time per token during the decode phase.

Baselines: We compare the performance of FlexInfer with several approaches: (1) CPU-only inference (on ICL or SPR CPU), (2) FlexGen (Sheng et al., 2023), a state-of-the-art offloading-based technique, (3) FlexGen_Opt[5], which strategically allocates portions of model weights, activations, and KV cache to GPU memory based on available capacity, and (4) SplitGen, which splits model layers between CPU and GPU. In FlexGen configuration, all model weights and KV cache are stored in CPU memory, with the required data transferred from CPU to GPU memory for each layer computation.

[1] At CPU base frequency.

[2] Measured on 64 cores with the STREAM benchmark. (McCalpin, 2006).

[3] Dense tensor core compute throughput.

[4] Measured with the STREAM benchmark. (McCalpin, 2006).

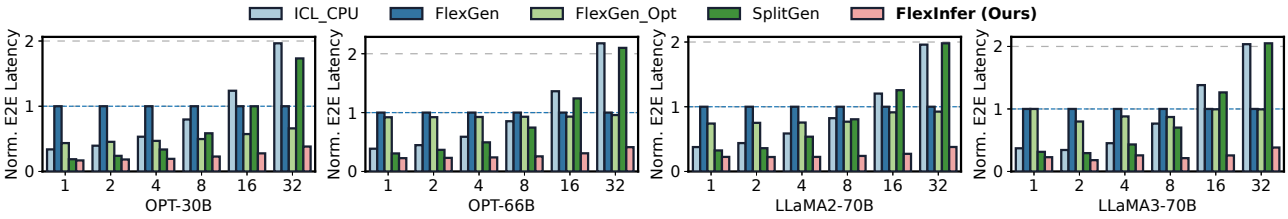
[5] Initial weight loading time is included in all measurements.

6.2 Performance Results

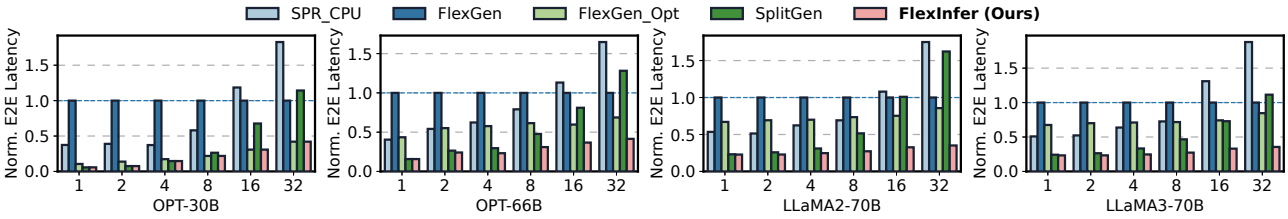
End-to-end latency comparison: Figure 8 shows the end-to-end latency for our evaluated models across various execution policies on each server. In this evaluation, we vary batch size from 1 to 32 and each result is normalized to FlexGen. As shown in the graph, FlexInfer achieves the lowest LLM inference latency across all evaluated models and configurations on both servers. On average, compared to FlexGen, FlexInfer reduces end-to-end latency by 75% and 76% on Servers 1 and 2, respectively. This demonstrates that FlexInfer can select the efficient execution policy for the prefill and decode phases and it is beneficial to reduce LLM inference latency significantly.

FlexGen_Opt and SplitGen both provide better performance than FlexGen by leveraging GPU memory to reduce PCIe traffic. FlexGen_Opt, which maximizes weights and KV cache placement in GPU memory, reduces latency by 23% and 50% on Servers 1 and 2, respectively. Similarly, SplitGen, which utilizes both CPU and GPU, reduces latency by 37% and 61.7% on Servers 1 and 2, respectively. However, these policies both have limitations when compared to FlexInfer. FlexGen_Opt reduces exposed PCIe transfer time proportionally to the fraction of model weights and KV cache that fits within GPU memory capacity. For Server 1, which has insufficient GPU memory for the large OPT-66B, LLaMA2-70B, and LLaMA3-70B models, this results in only marginal performance improvements over FlexGen. Additionally, during the decode phase, FlexGen_Opt still needs to transfer model weights and KV caches over PCIe. SplitGen always leverages CPU computation, which causes performance degradation in compute-intensive prefill phases, particularly with longer sequences and larger batch sizes. As a result, compared to FlexGen, its inference latency increases by up to 2.1× on Server 1 and up to 1.6× on Server 2.

TTFT and TPOT comparison: Figure 9 compares TTFT and TPOT for the OPT-66B model on Servers 1 and 2 across batch sizes of 1, 4, and 16, where the input sequence length is set to 512 and the output sequence length is set to 32. All results are normalized to the corresponding FlexGen result on the same server. FlexInfer consistently selects the optimal execution policy for both metrics, reducing TPOT by 77.7% and 79.6% on Servers 1 and 2, respec-

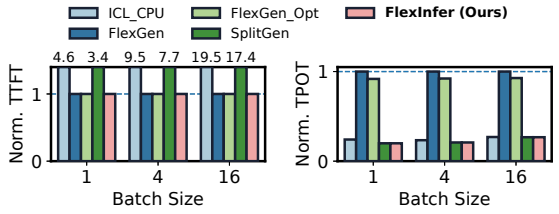


(a) End-to-end latency comparison result for Server 1.

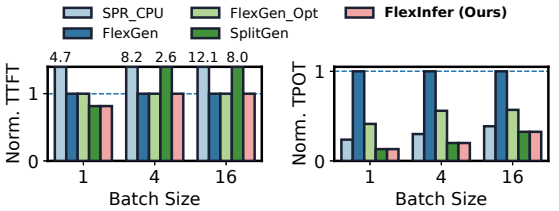


(b) End-to-end latency comparison result for Server 2.

Figure 8. End-to-end latency comparison results for OPT-30B, OPT-66B, LLaMA2-70B, and LLaMA3-70B with an input length of 512, an output length of 32, and batch sizes ranging from 1 to 32.



(a) TTFT and TPOT comparison result for OPT-66B on Server 1.



(b) TTFT and TPOT comparison result for OPT-66B on Server 2.

Figure 9. TTFT and TPOT comparison results for OPT-66B, with an input length of 512, an output length of 32, and batch sizes of 1, 4, and 16.

tively. For TTFT, it achieves a 6.6% reduction on Server 2 while matching FlexGen on Server 1, where the limited compute throughput of ICL CPU makes GPU offloading more efficient during prefill phase.

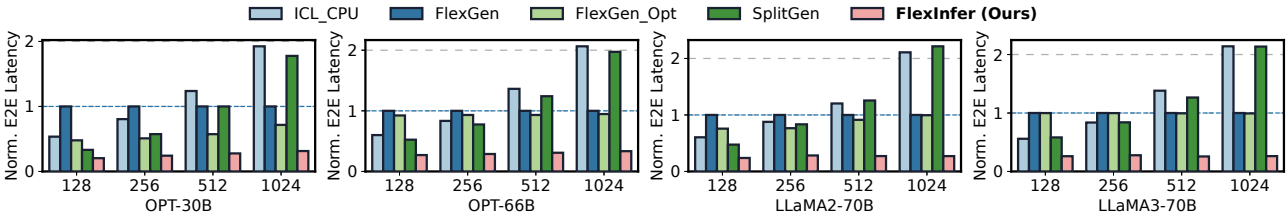
In contrast, schemes that apply the same execution policy to both the prefill and decode phases (i.e., CPU-only, FlexGen, FlexGen_Opt, and SplitGen) show either limited benefits or performance degradation in TTFT and TPOT. For instance, with SplitGen, CPU becomes a bottleneck as batch size increases, leading to an average increase in TTFT by 7.7× and 2.6× on Servers 1 and 2. For FlexGen_Opt, each layer's model weight is stored in GPU memory as much as capacity allows. On Server 1, with limited GPU memory and PCIe bandwidth, TPOT decreases by only 7.7%, whereas on Server 2, with larger memory and higher bandwidth, it improves by 49.1% on average. On the other hand, FlexInfer dynamically selects optimal policies for each phase based on runtime parameters and hardware configurations, highlighting the importance of adaptive policy selection.

6.3 PCIe Traffic Analysis

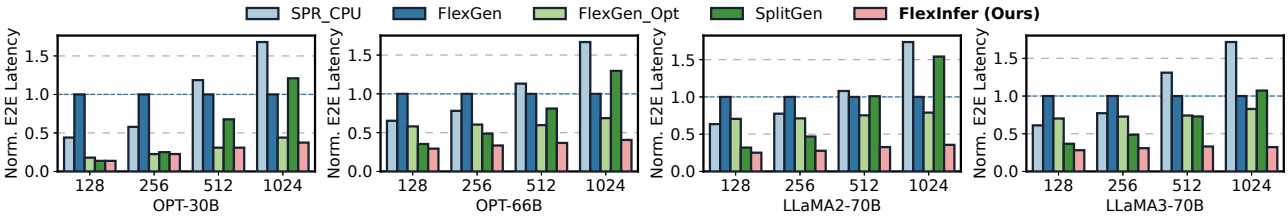
Figure 11 analyzes PCIe traffic between CPU and GPU for different execution policies when running OPT-66B on each server. CPU-only execution eliminates PCIe transfers entirely as all model components remain in CPU memory. SplitGen minimizes transfer volume by only moving the outputs from the last CPU-executed layer to GPU, minimizing CPU-GPU data transfer. FlexGen generates the highest traffic from weight and KV cache transfers, while FlexGen_Opt reduces this by storing partial data in GPU memory. FlexInfer dynamically selects optimal policies based on runtime parameters and hardware configurations—using FlexGen for compute-intensive prefill phase and SplitGen for memory-bound decode phase. This approach leverages GPU acceleration where beneficial while minimizing PCIe transfers, significantly improving performance compared to other methods.

6.4 Performance Impact of Input Sequence Length

Figure 10 compares performance across execution policies and models on both servers with batch size 16, varying input sequence lengths from 128 to 1024. Compared to FlexGen, CPU-only and SplitGen show significant performance degradation as input sequence length increases. This trend

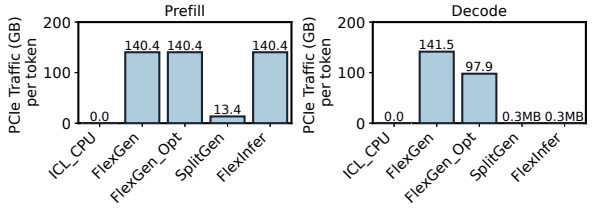


(a) End-to-end latency comparison result for Server 1.

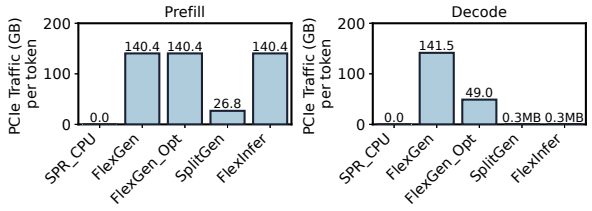


(b) End-to-end latency comparison result for Server 2.

Figure 10. End-to-end latency comparison for OPT-30B, OPT-66B, LLaMA2-70B, and LLaMA3-70B models with batch size 16, output length of 32, and input sequence length ranging from 128 to 1024.

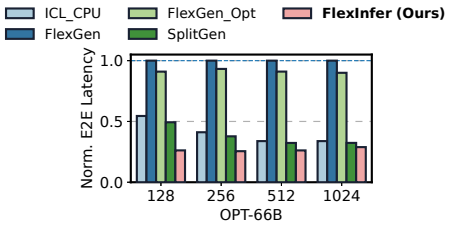


(a) PCIe data traffic for OPT-66B on Server 1.

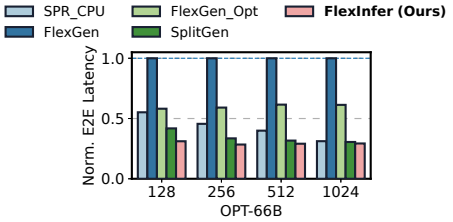


(b) PCIe data traffic for OPT-66B on Server 2.

Figure 11. PCIe data traffic between CPU and GPU comparison for OPT-66B with an input length of 512, output length of 32, and batch size of 16.



(a) End-to-end latency comparison result for OPT-66B on Server 1.



(b) End-to-end latency comparison result for OPT-66B on Server 2.

Figure 12. End-to-end latency comparison for OPT-66B model with an input sequence length of 512, batch size of 16, and output sequence length ranging from 128 to 1024.

occurs because longer inputs increase the proportion of time spent in the compute-intensive prefill phase. FlexInfer achieves the lowest end-to-end latency across all input lengths and models, significantly outperforming FlexGen. On average, FlexInfer reduces LLM inference latency by 72.9% on Server 1 and 70.2% on Server 2. This improvement is attributed to its flexibility in selecting the optimal policy for each phase.

6.5 Performance Impact of Output Sequence Length

Figure 12 shows performance results for different execution strategies running the OPT-66B model on each server as the output length increases from 128 to 1024, while maintaining a fixed input length of 512 and batch size of 16. As output sequence length increases, the decode phase dominates the end-to-end latency. FlexGen and FlexGen_Opt exhibit proportional latency increases due to PCIe data transfers for each generated token. In contrast, CPU-only computation (which eliminates data transfers entirely) and approaches that minimize data movement such as SplitGen and FlexInfer demonstrate lower end-to-end latency as output length grows. FlexInfer consistently achieves the lowest latency, reducing inference time by 73.3% on Server 1 and 70.5% on Server 2 compared to FlexGen by dynamically selecting optimal policies that minimize PCIe bottlenecks during inference.

7 Related Work

7.1 Deep Learning Acceleration on CPUs

Prior studies have explored CPU-specific optimizations for deep learning workloads (Liu et al., 2019; Shen et al., 2023; Georganas et al., 2018; Heinecke et al., 2016; Gong et al., 2022). NeoCPU (Liu et al., 2019) accelerates CNN inference through graph-level optimizations, while Graphite (Gong et al., 2022) enhances GNN performance through hardware-software co-design. RASA (Jeong et al., 2021) and VEGETA (Jeong et al., 2023) improve deep learning operations with specialized matrix multiplication engines. Recent works (Shen et al., 2023; He et al., 2024) enable efficient CPU-based LLM inference through weight quantization and optimized kernels.

7.2 LLM Inference Optmizations

Offloading-based LLM Inference: Several studies (Aminabadi et al., 2022; Wolf, 2019; Sheng et al., 2023; Xuanlei et al., 2024) have proposed offloading-based techniques to handle LLMs that exceed GPU memory capacity by utilizing CPU memory and disk storage. FlexGen (Sheng et al., 2023), DeepSpeed-ZeRO (Aminabadi et al., 2022) and HuggingFace Accelerate (Wolf, 2019) reduce GPU memory usage by offloading model weights to CPU memory and disk. Recent works have explored CPU computation for offloading-based LLM inference (Song et al., 2024; Xuanlei et al., 2024; Park & Egger, 2024). PowerInfer (Song et al., 2024) leverages activation sparsity by processing hot tensors on GPU and cold tensors on CPU, though primarily works under sparse activation assumptions. HeteGen (Xuanlei et al., 2024) applies tensor-parallelism across CPU and GPU, but becomes less effective with increasing batch sizes or model scales due to CPU computational limitations. In contrast, FlexInfer dynamically selects optimal execution policies based on phase characteristics, hardware capabilities, and runtime parameters. Park and Egger (Park & Egger, 2024) enhance LLM throughput by offloading attention blocks and portions of linear layers to CPU while optimizing data transfers. While their approach excludes CPU computation during the prefill phase, our FlexInfer demonstrates that leveraging high-throughput CPUs with AMX during prefill is also beneficial for maximizing overall inference performance.

Disaggregated LLM Inference: Recent studies have proposed executing prefill and decode phases on separate GPUs to exploit their distinct characteristics (Patel et al., 2024; Hu et al., 2024a; Zhong et al., 2024). TetriInfer (Hu et al., 2024a) separates phases to avoid interference, while Splitwise (Patel et al., 2024) optimizes GPU resource usage through phase-aware scheduling. DistServe (Zhong et al., 2024) maximizes performance by tailoring resource allocation and parallelism for each phase. However, these approaches focus only on GPU execution without utilizing CPU computation resources, and do not consider phase-specific execution policies. In contrast, FlexInfer optimizes performance by selecting suitable policies for each phase while leveraging both CPU and GPU resources.

8 Conclusions

This paper presents FlexInfer, a phase-aware LLM inference system that leverages both CPU and GPU resources effectively. Unlike prior approaches that suffer from PCIe bottlenecks and underutilized CPU resources, FlexInfer dynamically selects optimal execution policies for each inference phase, resulting in end-to-end latency reductions of 75% and 76% across different generations of hardware. Importantly, this work demonstrates that existing approaches for overlapping PCIe data transfer with GPU computation can be coupled with effective usage of CPU computation resources to both minimize the amount of data transfer and to more fully utilize both types of compute accelerators.

9 Acknowledgement

This work was supported in part through research infrastructure and services provided by the Rogues Gallery testbed (Young et al., 2019) hosted by the Center for Research into Novel Computing Hierarchies (CRNCH) at Georgia Tech. The Rogues Gallery testbed is primarily supported by the National Science Foundation (NSF) under NSF Award Number #2016701. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s), and do not necessarily reflect those of the NSF. Additionally, this research was partially supported by NSF PPOSS Award #2119523 and Intel. We also thank the anonymous reviewers for their feedback on improving the paper.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

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

Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J.,

Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G., Chen, J., et al. Deep Speech 2: End-to-end speech recognition in English and Mandarin. In *International Conference on Machine Learning (ICML)*, 2016.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

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

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

de Carvalho, J. P., Moreira, J. E., and Amaral, J. N. Compiling for the ibm matrix engine for enterprise workloads. *IEEE Micro*, 42(5):34–40, 2022.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Firoozshahian, A., Coburn, J., Levenstein, R., Nattoji, R., Kamath, A., Wu, O., Grewal, G., Aepala, H., Jakka, B., Dreyer, B., et al. Mtia: First generation silicon targeting meta's recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pp. 1–13, 2023.

Georganas, E., Avancha, S., Banerjee, K., Kalamkar, D., Henry, G., Pabst, H., and Heinecke, A. Anatomy of high-performance deep learning convolutions on simd architectures. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 830–841. IEEE, 2018.

Gim, I., Chen, G., Lee, S.-s., Sarda, N., Khandelwal, A., and Zhong, L. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

Gong, Z., Ji, H., Yao, Y., Fletcher, C. W., Hughes, C. J., and Torrellas, J. Graphite: optimizing graph neural networks on cpus through cooperative software-hardware techniques. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 916–931, 2022.

He, P., Zhou, S., Li, C., Huang, W., Yu, W., Wang, D., Meng, C., and Gui, S. Distributed inference performance optimization for llms on cpus. *arXiv preprint arXiv:2407.00029*, 2024.

Heinecke, A., Henry, G., Hutchinson, M., and Pabst, H. Libxsmm: accelerating small matrix multiplications by runtime code generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 981–991. IEEE, 2016.

Hu, C., Huang, H., Xu, L., Chen, X., Xu, J., Chen, S., Feng, H., Wang, C., Wang, S., Bao, Y., et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024a.

Hu, Q., Ye, Z., Wang, Z., Wang, G., Zhang, M., Chen, Q., Sun, P., Lin, D., Wang, X., Luo, Y., et al. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 709–729, 2024b.

Intel. Intel Extension for PyTorch. https://github.com/intel/intel-extension-for-pytorch, 2020.

Intel. Intel architecture instruction set extensions programming reference. *Intel Corp., Mountain View, CA, USA, Tech. Rep*, pp. 319433–030, 2023.

Jeong, G., Qin, E., Samajdar, A., Hughes, C. J., Subramoney, S., Kim, H., and Krishna, T. Rasa: Efficient register-aware systolic array matrix engine for cpu. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 253–258. IEEE, 2021.

Jeong, G., Damani, S., Bambhaniya, A. R., Qin, E., Hughes, C. J., Subramoney, S., Kim, H., and Krishna, T. Vegeta: Vertically-integrated extensions for sparse/dense gemm tile acceleration on cpus. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 259–272. IEEE, 2023.

Jouppi, N., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., Patil, N., Subramanian, S., Swing, A., Towles, B., et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pp. 1–14, 2023.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Kim, H., Ye, G., Wang, N., Yazdanbakhsh, A., and Kim, N. S. Exploiting intel® advanced matrix extensions (amx) for large language model inference. *IEEE Computer Architecture Letters*, 2024.

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

Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V., and Wang, Y. Optimizing {CNN} model inference on {CPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 1025–1040, 2019.

Markidis, S., Der Chien, S. W., Laure, E., Peng, I. B., and Vetter, J. S. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pp. 522–531. IEEE, 2018.

McCalpin, J. Stream: Sustainable memory bandwidth in high performance computers. *http://www. cs. virginia. edu/stream/*, 2006.

Na, S., Jeong, G., Ahn, B., Young, J., Krishna, T., and Kim, H. Understanding performance implications of llm inference on cpus. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2024.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pp. 1–15, 2019.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

Nassif, N., Munch, A. O., Molnar, C. L., Pasdast, G., Lyer, S. V., Yang, Z., Mendoza, O., Huddart, M., Venkataraman, S., Kandula, S., et al. Sapphire rapids: The next-generation intel xeon scalable processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pp. 44–46. IEEE, 2022.

NVIDIA. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems, 2025.

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

Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., and Bianchini, R. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024.

Shen, H., Chang, H., Dong, B., Luo, Y., and Meng, H. Efficient llm inference on cpus. *arXiv preprint arXiv:2311.00502*, 2023.

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

Song, Y., Mi, Z., Xie, H., and Chen, H. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pp. 590–606, 2024.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Vaswani, A. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

Weidmann, M. Introducing the scalable matrix extension for the armv9-a architecture, 2021.

Wolf, T. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Xuanlei, Z., Jia, B., Zhou, H., Liu, Z., Cheng, S., and You, Y. Hetegen: Efficient heterogeneous parallel inference for large language models on resource-constrained devices. *Proceedings of Machine Learning and Systems*, 6:162–172, 2024.

Young, J. S., Riedy, J., Conte, T. M., Sarkar, V., Chatarasi, P., and Srikanth, S. Experimental insights from the rogues gallery. In *2019 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8, Nov 2019. doi: 10.1109/ICRC.2019.8914707.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V.,

et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.