NEO: SAVING GPU MEMORY CRISIS WITH CPU OFFLOADING FOR ONLINE LLM INFERENCE

Xuanlin Jiang [1] **Yang Zhou** [2 3] **Shiyi Cao** [2] **Ion Stoica** [2] **Minlan Yu** [4]

ABSTRACT

Online LLM inference powers many exciting applications such as intelligent chatbots and autonomous agents. Modern LLM inference engines widely rely on request batching to improve inference throughput, aiming to make it cost-efficient when running on expensive GPU accelerators. However, the limited GPU memory has largely limited the batch size achieved in practice, leaving significant GPU compute resources wasted.

We present NEO, an online LLM inference system that offloads part of attention compute and KV cache states from the GPU to the local host CPU, effectively increasing the GPU batch size and thus inference throughput. To this end, NEO proposes asymmetric GPU-CPU pipelining and load-aware scheduling to balance GPU and CPU loads and fully utilize their compute and memory resources. We evaluate NEO on a wide range of workloads (i.e., code generation, text summarization), GPUs (i.e., T4, A10G, H100), and LLM models (i.e., 7B, 8B, 70B). NEO achieves up to 7.5×, 26%, and 14% higher throughput compared to GPU-only approach on T4, A10G, and H100 GPUs, respectively, while maintaining the same latency; with more powerful CPUs, NEO achieves up to 79.3% throughput gain on A10G GPU. To facilitate future research, we open-source our code at https://github.com/NEO-MLSys25/NEO.

1 INTRODUCTION

The advent of auto-regressive transformer-based large language models (LLMs) has significantly reshaped existing technologies such as search engines and chatbots and empowered various new ones, such as autonomous agents and programming assistants. In these online scenarios, LLM inference is directly user-facing and thus requires low latency for immersive interaction; it also desires high throughput, typically via request batching, to efficiently leverage expensive hardware accelerators like GPUs (Kwon et al., 2023).

However, to achieve large batch sizes for high throughput, online LLM inference requires huge GPU memory, but GPUs have limited memory resources. Nowadays LLM models have billions of parameters (e.g., 70B LLaMa-3.1 model (Dubey et al., 2024)) that occupy dozens to hundreds of GB GPU memory; modern LLM inference engines like vLLM (Kwon et al., 2023) additionally store KV cache in the GPU memory to reuse previous computations, whose size increases linearly with prompt and output length. As a result, the memory-bounded LLM inference workloads have created the GPU memory crisis where people demand expensive high-end GPUs with large memory sizes.

Prior work has recognized this problem and proposed various solutions. One line of work is on model quantization (Zhao et al., 2024b; Xiao et al., 2023b) to reduce model memory consumption. However, they come at the cost of lower accuracy. Another line of work is offloading model weights, KV cache, and compute to the CPU, such as FlexGen (Sheng et al., 2023), PowerInfer (Song et al., 2024), TwinPilots (Yu et al., 2024), HeteGen (Zhao et al., 2024a), and FastDecode (He & Zhai, 2024). Memory offloading could increase request batch sizes on the GPU, potentially increasing overall inference throughput; compute offloading avoids repetitively swapping the KV cache between the GPU and CPU, thus preventing PCIe bandwidth from becoming the bottleneck. Unfortunately, most of these work trades inference latency for throughput by using huge GPU batch sizes and layer-by-layer swapping (Sheng et al., 2023). The former means they sacrifice latency for throughput, and the latter means they need to swap the KV-cache back-and-forth between the GPU and CPU, making communication bandwidth the bottleneck. Thus, these designs are not suitable for online inference; the exception is FastDecode, but it uses 8 32-core AMD Epyc CPUs in remote servers to handle the offloaded compute of only one A10G GPU, where the CPUs cost much more than the GPU.

This paper aims to achieve higher throughput for online

[1]Peking University [2]University of California, Berkeley [3]University of California, Davis [4]Harvard University. Correspondence to: Xuanlin Jiang <xljiang@stu.pku.edu.cn>, Yang Zhou <yangzhou.rpc@gmail.com>.

Proceedings of the 8th MLSys Conference, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).

LLM inference *without compromising accuracy or latency* in a cost-efficient way—only using *local host CPUs* as the offloading target that comes with the GPU "as free". Achieving this goal faces two main challenges. First, within each inference iteration, how to balance the compute happening on GPU and CPU to fully utilize their compute and memory resources, while not overloading them? This is challenging because of the vastly different characteristics between GPUs and CPUs. For example, a low-end GPU could easily have nearly TB/s memory bandwidth and hundreds of TFLOPS compute, while a high-end CPU server has only a few hundreds of GB/s memory bandwidth and few TFLOPS compute (He & Zhai, 2024). Prior work like FastDecode fully offloads the KV cache and attention computation to the CPU, making the CPU severely bottleneck the system. Second, across inference iterations, how to adaptively decide the offloading policy for real-world workloads with dynamic input/output lengths? Prior work like FlexGen and FastDecode assumes fixed input/output lengths across requests, and leverages static optimal offloading policies that were determined by one-time offline profiling. As inference iterations proceed, dynamic input/output lengths of requests in real-world workloads would easily break the optimality of such policies.

To address these two challenges, we present NEO[1] with two key designs: *asymmetric GPU-CPU pipelining* within each inference iteration, and *load-aware scheduling* across iterations. Asymmetric pipelining runs two asymmetric sub-batches concurrently: one offloads the decoding attention computation and KV cache of a *subset* of requests into the CPU, and another one runs the rest in the GPU; these two sub-batches overlap with each other to balance GPU and CPU loads. This partial offloading will help keep the offloaded compute and memory bandwidth consumption within CPU capacity. Load-aware scheduling online monitors various request waiting and running queues (for the CPU and GPU), then dynamically decides which requests should be offloaded to the CPU and how to form request batches, based on principled heuristics to optimize for the highest throughput.

Perhaps the closest work to NEO is FastDecode, but NEO differs in several important designs: 1) it features partial offloading to limit the offloaded computation and memory bandwidth consumption without overloading the CPU, 2) it adaptively finds the optimal point to balance GPU and CPU loads for dynamically-changing real-world workloads.

NEO is implemented based on SwiftLLM (interestingLSY/swiftLLM), a simplified copy of vLLM with similar performance, but could be adapted to other frameworks like vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2023). It leverages the Intel ISPC compiler (ispc/ispc) to generate efficient CPU kernels for attention computations. We thoroughly evaluate NEO on AWS GPU instances (`g4` with a T4 GPU and `g5` with an A10G GPU) and our local 8×H100 testbed, both with only the host CPU and memory for offloading. Our evaluation covers two public online inference datasets and three popular LLM models ranging from 7B, 8B, to 70B. NEO achieves up to 14%-7.5× (depending on GPUs) higher throughput over the non-offloaded version while keeping the same inference latency; it further achieves up to 79.3% performance gains with more power CPUs.

To the best of our knowledge, NEO *is the first CPU offloading system for online LLM inference that achieves performance gains over GPU-only systems with the same hardware cost and inference accuracy.* We hope this opens a new door for cost-efficient LLM inference research.

2 BACKGROUND AND MOTIVATION

2.1 LLM Inference and Performance Bottleneck

Auto-regressive transformer-based LLMs (Vaswani, 2017) perceive tokens as the basic elements of languages, with the main task of predicting the next token of the given sequence. Formally, given $[t_1, t_2, \ldots, t_n]$ as input, for each token x in the vocabulary, an LLM needs to return the probability that x is the next token of the sequence, i.e. $\Pr[t_{n+1} = x \mid t_1, \ldots, t_n]$. To do this prediction, as illustrated in Figure 1, the transformer-based LLM first converts tokens to embedding vectors. These vectors are then passed through a series of primary blocks called transformer layers, leveraging the attention mechanism (Vaswani, 2017). The embedding vectors remain the same shape but become more precise and context-aware after passing through each transformer layer, before finally getting through a final fully connected layer that converts embedding vectors to corresponding probabilities for each possible token.

One inference request typically consists of prefilling and decoding stages and heavily involves the KV cache on GPU memory to reuse previous computations. The prefilling stage generates the initial KV cache after consuming all input tokens, while the decoding stage repetitively read-and-appends the KV cache and auto-regressively generates output tokens until an EOS (End Of Sequence). To optimize this process, modern LLM inference engines like vLLM (Kwon et al., 2023) leverage iteration-level scheduling (Yu et al., 2022) to accommodate various input/output lengths of requests, selective batching (Yu et al., 2022) to increase performance by batching matrix multiplications, and paged attention to efficiently manage GPU memory.

The throughput of LLM inference highly depends on the batch size the engine can achieve, which is essentially bounded by the GPU memory size due to the large KV cache (Kwon et al., 2023). Prior work (He & Zhai, 2024; Zhu et al., 2024) has demonstrated that the throughput would

[1]Neo is the protagonist in *The Matrix* who saves humankind.

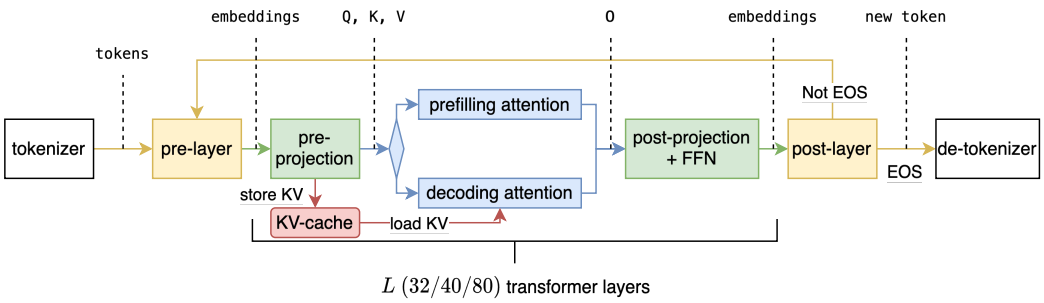


Figure 1. Workflow of transformer-based LLM Inference.

increase almost linearly as the batch size increases up to hundreds on modern A10, V100, and H100 GPUs; meanwhile, unfortunately, these GPUs fail to accommodate hundreds or even dozens of batch size, leaving significant GPU compute resources unutilized.

2.2 CPU Memory as a Possible Rescue

One approach to increase batch size is storing the overflowing KV cache in the main memory of the CPU, transferring it to the GPU when needed, and transferring it back when not needed. However, such repetitive KV cache swap-ins and -outs make the system severely bounded by GPU-CPU PCIe bandwidth, as shown by prior work (Sheng et al., 2023). Fortunately, only the decoding-phase attention operation relies on the KV cache, therefore offloading this part of computation to the CPU would help avoid repetitively transferring the KV cache between the GPU and CPU. Moreover, this operation only takes a tiny proportion of computation (compared to other parts in the transformer architecture) and doesn't require loading model weights.

The decoding attention operation is memory-bandwidth-bounded on both GPU and CPU due to low arithmetic intensity (i.e., FLOP per memory load) (He & Zhai, 2024; Zhu et al., 2024). The memory bandwidth gap between GPUs and CPUs is much smaller than their compute gap. For example, an A10G GPU features 600 GB/s memory bandwidth and 125 TFLOPS, while a modern x86 server has around 200 GB/s memory bandwidth with 1.2 TFLOPS (He & Zhai, 2024; Zhu et al., 2024); modern ARM processors like AWS Graviton4 offer 537.6 GB/s memory bandwidth per socket (WikiChip). Therefore, although there may seem to be a huge compute gap between GPUs and CPUs (i.e., 125 vs. 1.2 TFLOPS), the actual performance gap for the decoding attention operation is relatively small because their memory bandwidths are closer (i.e., 600 vs. 200 GB/s).

Frequent CPU-GPU communication could be another concern for CPU offloading. Fortunately, according to our experiments on AWS g5 instances with LLaMa-3-8B model, dense operations on the GPU take 5-10 times the time of communication for every input token, which means communication can be perfectly hidden by GPU computation.

2.3 Challenges

Despite being promising, there are several challenges when building an efficient LLM inference system that offloads decoding attention compute and memory to CPUs. These challenges stem from the fundamentally different capabilities between GPUs and CPUs (in terms of memory and compute power), and get amplified in real-world dynamically-changing inference workloads (e.g., various input/output lengths).

Challenge #1: How to efficiently overlap the GPU and CPU *within each inference iteration*? GPUs have more compute and memory bandwidth but are limited in memory size, while CPUs have more memory and a decent amount of memory bandwidth, but lack strong compute power. Therefore, we must carefully restructure the pipeline of LLM inference to fit different pipeline modules into the right hardware, while not overloading any hardware. Such restructuring also needs to take care of the complex inter- and intra-transformer-layer data dependencies, without breaking transformer semantics.

Challenge #2: How to schedule inference requests to the GPU and CPU *across inference iterations* to maintain high performance in dynamically-changing workloads? Prior work like FastDecode (He & Zhai, 2024), FlexGen (Sheng et al., 2023), and more (Chen et al., 2024) only consider idealistic settings where request input and output lengths are fixed, and adopt a static scheduling policy (e.g., obtained from offline profiling) to assign requests across the GPU and CPU. However, in real-world dynamic settings, vastly different input/output lengths would make static scheduling no longer work efficiently; instead, it would require an adaptive scheduling policy to determine the best request assignments at the per-iteration level.

3 NEO DESIGN

Figure 2 shows the high-level architecture of NEO. NEO consists of a request scheduler running on the CPU and a set of executors running across the CPU and GPUs. The

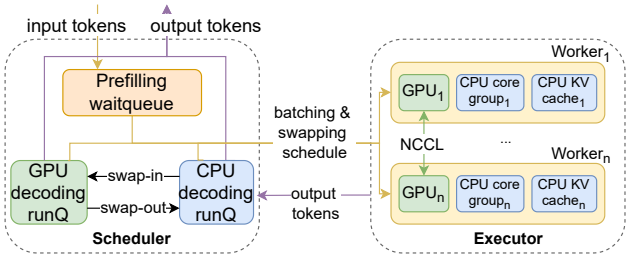


Figure 2. Overall architecture of NEO. "runQ" means "runqueue".

request scheduler maintains a prefilling waitqueue, a GPU decoding runqueue, and a CPU decoding runqueue; this scheduler makes iteration-level adaptive scheduling decisions on whether to run the incoming requests on GPU or CPU. The executors accept batching and swapping schedule from the scheduler, and returns back output tokens that are generated collaboratively by the CPU and GPUs. NEO features two key techniques: 1) asymmetric pipelining to fully leverage the compute resource of both GPU and CPU without overloading them (§3.1), and 2) load-aware scheduling to handle dynamically-changing workloads, e.g., different input/output lengths across requests (§3.2).

3.1 Asymmetric Pipelining

NEO proposes *asymmetric pipelining* to address the GPU-CPU overlapping challenge in §2.3. This design offloads decoding attention (both its compute and KV cache) of a selected portion of inference requests to the CPU. It forms two sub-batches of requests—one mostly runs on the GPU while another runs across the GPU and CPU, and overlaps these two sub-batches to balance the load of GPU and CPU.

To motivate this design, we first explore a simple strawman called *simple offloading* that offloads compute and KV cache to the CPU but leaves the GPU idle. Next, we examine a more intricate strawman called *symmetric pipelining* used by prior work (He & Zhai, 2024; Chen et al., 2024) that tends to leave the GPU idle. Finally, we will arrive at our design of asymmetric pipelining, and show how it effectively addresses the issues in the simple offloading and symmetric pipelining designs.

Strawman #1: simple offloading. As shown in Figure 3, this design extracts the decoding attention and offloads its compute and KV cache to the CPU, while leaving the rest to the GPU. The rest includes prefilling attention and token-wise independent operations—referred to as the linear operations that mainly involve matrix multiplications. However, during these linear operations, the CPU always remains idle. As a result, this design fails to leverage the compute and memory resources of CPUs.

Strawman #2: symmetric pipelining. A straightforward approach to reducing the CPU idle time is to evenly split a single decoding batch into two sub-batches, and overlap their linear operations (on the GPU) and attention operations (on the CPU), as shown in Figure 4. For the prefilling batch, symmetric pipelining just runs it on the GPU without offloading, as the prefilling stage requires high computation for matrix multiplication while not consuming much memory (Zhu et al., 2024). The output of the symmetric pipelining, i.e., the KV cache, will be swapped out to the CPU for offloading. This technique has been used by prior work like FastDecode (He & Zhai, 2024) and more (Chen et al., 2024). Nevertheless, seemingly efficient, this design suffers from three major issues.

- Firstly, this design results in significant underutilization of GPU memory. In this scheme, the GPU solely retains the model weights and runtime activations, while the rest of the memory—which stores the KV cache in non-offloading settings—remains unused.
- Secondly, this design fails to achieve balanced GPU-CPU overlapping. This is because 1) it entirely overlooks the prefilling stage and KV cache swap-out time, during which the CPU stays idle; 2) the linear stage of a decoding sub-batch on the GPU is typically much shorter than the attention stage on the CPU, due to the attention's auto-regressive nature and high memory bandwidth consumption. As a result, the duration of the attention stage will likely exceed that of the linear stage, causing the CPU to become the bottleneck.
- Finally, it is challenging to split two batches evenly in practice. In real-world workloads, it is nearly impossible to ensure that a single batch can be divided into two identical sub-batches, due to different input lengths and unpredictable output lengths. The discrepancies between the batches would likely result in a significant number of idle periods or "bubbles" in the pipeline.

Asymmetric pipelining, as shown in Figure 5, offers a solution to the aforementioned problems. To fully utilize GPU memory, NEO does *partial offloading*. The KV cache system of NEO is divided into two separate components: the "GPU-cache" located in the GPU's HBM, and the "CPU-cache" located in the CPU main memory. For any request that has already been prefilled in the system, its KV cache will either reside entirely in the GPU-cache—designated as a "GPU-request"—or entirely in the CPU-cache—designated as a "CPU-request". Requests are prioritized for storage in the GPU-cache to maximize GPU memory utilization.

To achieve full GPU-CPU overlapping, NEO integrates the prefilling stage computation into the GPU decoding sub-batch, so that the prefilling stage computation (in GPU) also happens in parallel with the CPU attention computation. This shares a similar spirit as the *selective batching* technique used in Orca (Yu et al., 2022). In our context, this selective batching largely extends the duration of the GPU

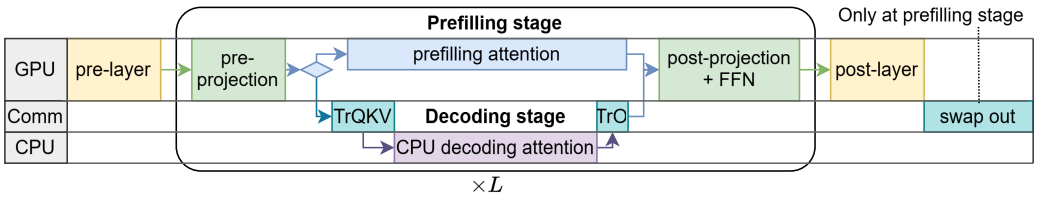


Figure 3. Simple offloading strawman offloads all requests' KV cache and decoding attention computation to the CPU. "Comm" stands for GPU-CPU communication; "TrQKV" means transferring Q,K,V tensors to CPU; "TrO" means transferring attention output to GPU.

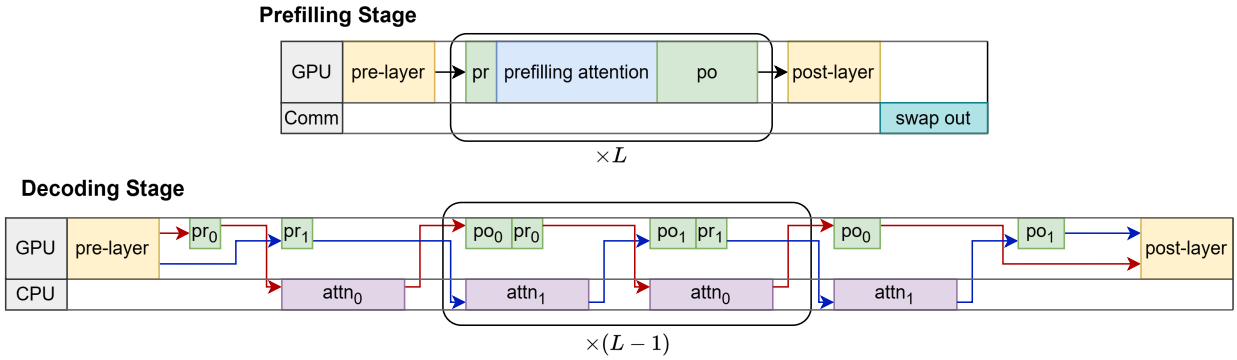


Figure 4. Symmetric pipelining strawman forms two identical sub-batches and overlaps linear and attention operations for the decoding stage. The red and blue arrows depict the data flows of the two sub-batches. "pr" means pre-projection and "po" means post-projection + FFN operations; together they form the linear stage. "attn" means attention operations. "TrQKV"s and "TrO"s are omitted for simplicity.

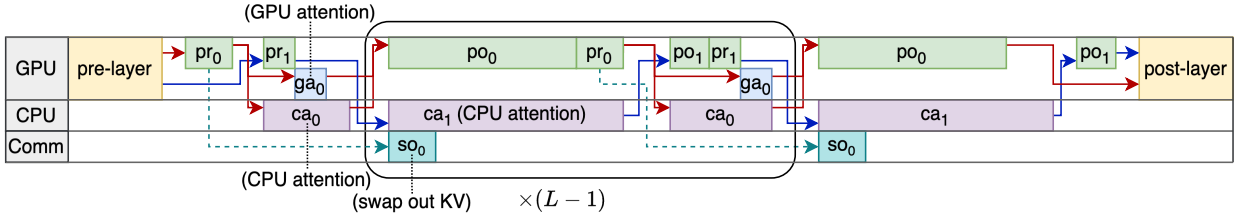


Figure 5. Asymmetric pipelining integrates the prefilling stage into one sub-batch (red arrows) and most of the decoding attention operations into another (blue arrows). "pr" means pre-projection, while "po" means post-projection + FFN operations; "ca" means attention operations done on CPU; "ga" means attention operations done on GPU; "Comm" stands for GPU-CPU communication.

compute, allowing for a longer overall CPU computing time and enabling more CPU-requests to be incorporated into the batch. Further, NEO leverages the *layer-wise swapping* technique to facilitate the overlapping of KV value transmission with computation. Given that the KV values of newly prefilled requests are computed layer by layer, we can initiate PCIe transmission immediately after each layer's KV value is computed, rather than deferring this process until the end of the entire iteration.

To simplify the batch division scheme and minimize idle periods, NEO introduces *asymmetric batch division*. Instead of attempting to create two identical batches, we consolidate all prefilling requests and GPU decoding requests with a few CPU decoding requests to batch-0, while dispatching the majority of CPU decoding requests to batch-1. The two batches are complementary: batch-0 features a long linear stage with little CPU attention computation, whereas batch-1 includes a lengthy attention stage with a very short linear stage. This arrangement not only simplifies system implementation, but also facilitates effective overlapping between the two batches, resulting in an inference iteration characterized by alternative "long-stages" and "short-stages", rather than uniform stages.

Asymmetric pipelining offers even more benefits. As shown in Figure 5, the non-overlapping segments at the beginning and end are minimized due to this intentional asymmetry. Furthermore, this approach reduces GPU kernel launching overhead, which can be substantial in Python due to its limited multi-threading parallelism, consuming considerable CPU time. In asymmetric pipelining, the GPU attention kernel is only invoked once per iteration, as only one sub-batch will need the GPU attention, compared to twice in symmetric pipelining.

3.2 Load-Aware Scheduling

Real-world inference workloads are complex, irregular, and dynamically changing, e.g., different input/output lengths across requests. NEO scheduler considers asymmetric pipelining and adaptively determines whether the incoming request should be placed on the GPU or CPU, to keep both busy while not overloading anyone.

NEO faces a more challenging problem than prior GPU-only inference engines like vLLM. First, NEO needs to form two sub-batches in each iteration, which means we need to consider batch selection and batch splitting at the same time. Secondly, blindly putting as many requests as possible into the CPU decoding runqueue does not work well. This is because too many CPU requests might overload the CPU capacity either in memory bandwidth or compute. Finally, the two-batch asymmetric pipelining does not always work better than GPU-only inference. This is because in asymmetric pipelining, the GPU memory needs to hold two concurrently running sub-batches, and each sub-batch can only achieve half of the maximum batch size; if NEO cannot offload enough KV caches, the sub-batch size might be even smaller than the GPU-only inference, yielding lower throughput.

To address these challenges, NEO follows several principles:

- **Greedy:** At the beginning of each iteration, NEO's scheduler would make a GPU-only inference schedule and a two-batch asymmetric pipelining schedule, and would choose the one with higher estimated throughput as the final decision.
- **Balancing:** For asymmetric pipelining, the scheduler should minimize pipeline "bubbles". That is, the estimated CPU busy time and GPU busy time should be as close as possible.
- **Hiding CPU:** For asymmetric pipelining, there shouldn't be cases when the CPU is busy but the GPU is idle.
- **Maximizing GPU:** For asymmetric pipelining, in each iteration, the scheduler should pick as many requests as possible from the prefilling waitqueue, and GPU and CPU decoding runqueues (that hold requests running on GPU and CPU respectively).

Concretely, denote each iteration time as T and batch size as x, NEO's goal is to minimize T/x. As shown in Figure 5, T consists of the following components: pre-layer time T_{prl}, transformer layer time T_{tr}, post-layer time T_{pol}, while T_{tr} is the major part that usually takes more than 95% of time in each iteration. Therefore, we can estimate T as below:

$$T \approx T_{tr} = L \times (\max\{T_{po_0} + T_{pr_0}, T_{ca_1}\} + \max\{T_{po_1} + T_{pr_1} + T_{ga_0}, T_{ca_0}\})$$

where T_{po_x}, T_{pr_x}, T_{ga_x}, and T_{ca_x} denote time for post-projection, pre-projection, GPU attention, and CPU attention time for batch-x, respectively. For simplicity, we define $T_{l_x} = T_{po_x} + T_{pr_x}$, which stands for time mainly consumed by multiplying activations with model weights in one transformer layer. To estimate T_l, T_{ga}, and T_{ca}, NEO does offline profiling for typical input/output lengths and uses linear interpolation to approximate the values for other lengths. Overall, to stick to principles of "Balancing" and "Hiding GPU", we would like to guarantee $T_{l_0} \geq T_{ca_1}$ and $T_{l_1} + T_{ga_0} \geq T_{ca_0}$, and minimize the gap between the left-hand side and right-hand side of both inequations.

Based on the analysis above, we present our load-aware scheduler, with the following procedures on each iteration:

1. *Initialization.* Initialize two empty batch schedules: batch-0 would mostly run on the GPU and contain requests in both prefilling and decoding phases; batch-1 would mostly run on the CPU and only contain requests that calculate decoding attention.
2. *Schedule GPU decoding requests.* Try to put every request from the GPU decoding runqueue into batch-0. Then swap out requests until GPU memory can hold all new KV cache, or swap in requests if there is ample space on GPU (Maximizing GPU).
3. *Schedule prefilling requests.* Pop the prefilling waitqueue and put requests into batch-0: keep the generated KV cache on GPU if there is enough GPU memory, otherwise, swap out the generated KV cache. Do this repeatedly until the GPU cannot hold the activations in the batch (Maximizing GPU).
4. *Schedule CPU decoding requests.* Scan the CPU decoding runqueue and put requests into either batch-0 or batch-1, maintaining $T_{ca_0} \leq T_{l_1} + T_{ga_0}$ and $T_{ca_1} \leq T_{l_0}$. If putting a request into either batch would violate the inequations, skip this request and leave it for the next iteration (Balancing and hiding CPU).
5. *Reduce prefilling requests.* Remove any prefilling request from batch-0 that would require swapping out the generated KV cache, as long as the above inequations hold. This is to avoid the CPU being idle (Balancing).
6. *Make decisions.* Now the two-batch schedule is made; we make the GPU-only schedule by taking batch-0 and excluding all the CPU decoding requests added in step 4. Finally, we compare their estimated T_{tr}/x values and pick the schedule with a smaller one (Greed).

We further measure and find that the scheduling overhead is less than 3ms per iteration, while each iteration takes hundreds of milliseconds. Therefore, its overhead would not have a big impact on the end-to-end performance. Interested readers can refer to appendix A for more details of the scheduling algorithm.

4 IMPLEMENTATION

We implement NEO based on SwiftLLM (interestingLSY/swiftLLM)—a tiny yet powerful LLM inference system for research purposes. SwiftLLM achieves vLLM-equivalent (single GPU, default scheduling policy) performance with around 2K lines of code. Our NEO implementation consists of 4K lines of Python and 1.5K lines of C++.

Efficient CPU Kernels. Existing vendor-supplied CPU kernel libraries don't support paged attention algorithm, where the inputs are non-continuous tensors and indexed by a block table. Therefore, we implement a custom C++ torch extension library called Paged-Attention-for-CPU (PACPU) and plug it into SwiftLLM. The PyTorch runtime will directly call a multi-threaded procedure written in C++. Under the hood, PACPU calls yet another extension library written in ISPC (ispc/ispc)—a language for writing SPMD programs on CPUs. ISPC code can be compiled to targets with different sets of vectorized instructions, such as AVX2, AVX512, and ARM Neon, making it perfectly portable between different types of CPUs. PACPU utilizes a paged KV cache similar to vLLM with the Paged Attention algorithm to mitigate memory fragmentation.

NEO optimizes memory access performance for decoding attention on CPU, as the decoding attention is heavily bounded by memory bandwidth (even with GQA) (Agrawal et al., 2024; Zhu et al., 2024). 1) Within a single CPU core, we utilize SIMD memory load/store instructions (features provided by ISPC) to minimize instruction overhead and enhance memory throughput. Additionally, we carefully organize the memory access order to ensure its contiguity. 2) Across different CPU cores, we use a parallelism strategy similar to Flash Decoding (Dao et al.). For each request, we partition its computation into individual tasks along the request dimension, allowing each task to access unique and continuous memory at block granularity. These tasks are evenly dispatched to all threads with each thread having an equal number of blocks to process. Finally, we aggregate the partial outputs of each request to obtain the final result.

Reducing Kernel Launching Overhead. Due to Python's poor multi-threaded execution performance incurred by its global interpreter lock (GIL), the data plane CPU kernels could not run in parallel with the control plane CUDA kernel launching calls, making kernel launching blocking on the critical path. To reduce kernel launching overhead, we replaced most of SwiftLLM's Triton-JIT (Tillet et al., 2019; Tillet) kernels with kernels written in CUDA C++. This can effectively reduce the additional kernel selection and launching overhead brought by Triton-JIT. With custom GPU kernels, we reduced the kernel launching overhead of LLaMa-3-8B from 1.2ms to 0.6ms per layer.

Multi-GPU inference. We redesigned SwiftLLM's architecture to support model sharding and tensor parallelism, as it originally only supported single-GPU inference. We utilize Ray actors (Moritz et al., 2018) to hold shards of the model and use PyTorch's pre-built communication library , which is a wrapper of NCCL, to handle cross-GPU communication. The underlying mechanism of splitting tensors and collecting results across GPUs is the same as vLLM (Kwon et al., 2023). As a standard approach, each Ray actor also has its partition of CPU KV cache, each responsible for a portion of KV heads to avoid cross-CPU communication.

5 EVALUATION

5.1 Experiment Setup

Testbeds. We run our experiments on multiple types of single-GPU instances on the AWS EC2 public cloud, including g5.2/4/8/16xlarge with an A10G GPU and g4dn.4xlarge with a T4 GPU. By default, we use g5.4xlarge for all A10G experiments. We also run on an 8×H100 SXM local server to test multi-GPU performance. The specifications of hardware are listed in Table 1. Note that the HGX machine has 4 CPU NUMA nodes. We confine our system to running on a single NUMA node (thus 1/4th of the total memory size and bandwidth) when running 2-GPU experiments.

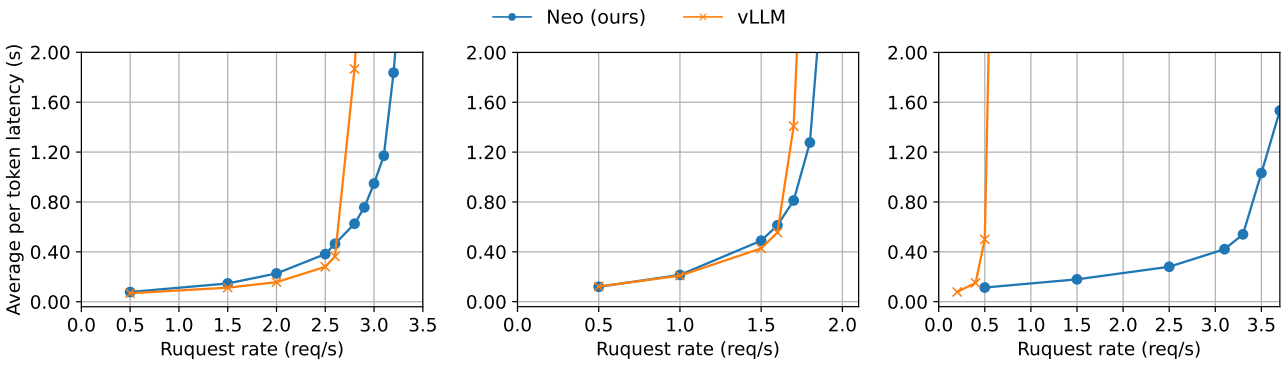
Name	GPU	CPU (#cores)	Memory
g5.$n$xlarge	A10G	EPYC 7R32 ($2n$)	$16n$ GB
g4.4xlarge	T4	Xeon P-8259CL (8)	64 GB
HGX	8×H100	Xeon 8462Y+ (64)	2 TB

Table 1. Hardware specifications of our testbeds.

Models. We evaluate NEO on the representative Llama models including Llama-3.1-8B, Llama-3.1-70B (Dubey et al., 2024) and Llam-2-7B (Touvron et al., 2023).

Baselines. We consider three baselines in our evaluation.

- vLLM (Kwon et al., 2023) is a popular state-of-the-art LLM inference system. vLLM's default scheduling policy doesn't include selective batching; so we set the `--enable-chunked-prefill` flag to enable it.
- FastDecode (He & Zhai, 2024) is an LLM inference system that offloads full decoding attention. Since the original work didn't consider the prefilling stage or implement an end-to-end system, we implemented our own version of FastDecode$^+$. It leverages NEO's asymmetric pipelining and load-aware scheduling, but offloads all requests' decoding attention to the host CPU.
- SwiftLLM (interestingLSY/swiftLLM) is a simplified version of vLLM with similar Pythonic implementation and without offloading, upon which NEO is built. We also make simple modifications to SwiftLLM to support multi-GPU inference. It achieves comparable performance with vLLM on a single GPU, and slightly lower performance than vLLM in 2-GPU settings (§5.5).



(a) 2xH100 + Llama-3.1-70B + AC. (b) A10G + Llama-3.1-8B + AC. (c) T4 + Llama-2-7B + OSC.

Figure 6. Load-latency curve comparison between NEO and vLLM on 3 different settings. For each request, we compute its per-token latency by dividing its full latency by its output token number, and then we take the average among all requests.

Workloads. We use real-world workloads as well as synthetic ones to evaluate our system.

- Azure LLM inference trace for coding (AC) (Microsoft; Patel et al., 2024) is a LLM coding trace collected from Azure cloud's production environment.
- OpenAI summarization comparison (OSC) (CarperAI) is an open dataset of input text, chosen summary, and rejected summary produced in real-world human chatbot interactions.
- Synthetic workloads with various input and output lengths. For a pair of input length l_i and output length l_o, we synthesize requests with input and output lengths sampled independently and uniformly from $[0.9l_i, 1.1l_i]$ and $[0.9l_o, 1.1l_o]$, respectively.

We use the AC trace with relatively longer requests on the H100 and A10G GPUs, while using the OSC trace with shorter requests on the lower-end T4 GPU.

5.2 Online Latency vs. Load

We evaluate the online latency of NEO under various request rates. We sample request arrival timestamps following the Poisson process. As Figure 6 shows, NEO sustains higher loads than vLLM in all listed hardware/model settings while providing comparable latencies at low rates. NEO achieves 14.3% higher throughput on H100 (at 2 sec latency), 6.40% higher on A10G (at 2 sec latency), and 563% higher on T4 (at 1 sec latency). In the T4+Llama-2-7B setting, we achieve nearly 6× throughput gains over vLLM; this is because the T4 GPU has an extremely constrained memory budget for the KV cache, severely limiting vLLM batch size, while NEO could offload the KV cache to the CPU, achieving a much larger batch size.

Figure 6 also shows that NEO achieves comparable latencies at low request rates, slightly higher latencies at intermediate rates, and significantly lower latencies at large rates. This is because of the two-fold impact of increasing the batch size. On the negative side, it will slightly increase execution latency because more requests get batched together; on the positive side, it improves throughput thus reducing queueing delay. These impacts vary at different request rates. When the request rate is very low, batching has a tiny impact since requests are processed one by one and won't queue up. As the request rate grows, the execution delay penalty grows slightly faster than the queueing reduction at first, but will be overtaken shortly as the queue expands.

There are two additional reasons for NEO's higher latency within the intermediate range: 1) vLLM is heavily optimized, while NEO features less engineering optimizations for simplicity and performance clarity—this is especially true in multi-GPU settings. 2) NEO actively seeks opportunities to offload requests, even though offloading may not help, which consequently leads to more system-level overheads in scheduling and swapping.

Figure 7 further compares the latency distributions of NEO and vLLM, and shows that our throughput gains do not come at the cost of latency. Our inference latency is comparable to vLLM's at all percentages.

5.3 Comparison with FastDecode$^+$

We compare NEO and FastDecode$^+$ in terms of both latency and throughput. The results are shown in Figure 8. FastDecode$^+$'s higher latency is caused by its inflexibility to tackle irregular workloads. When there are many requests in the CPU decoding runqueue but few or no requests in the prefilling waitqueue, FastDecode$^+$ would have no choice but to launch CPU batches to make progress, hindering overall performance, while NEO could simply launch GPU batches to utilize GPU resources.

Furthermore, NEO is always better in throughput than baseline because its scheduler can always decide to fall back

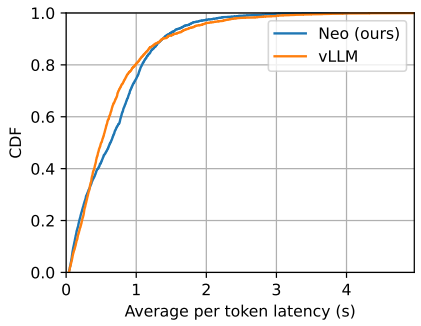
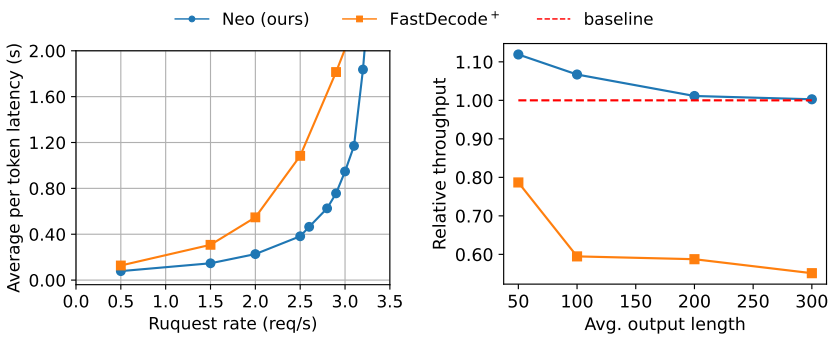


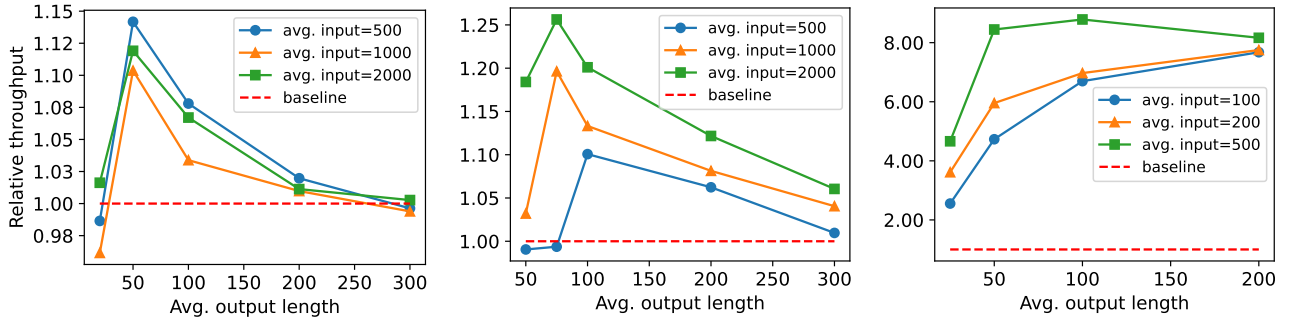
Figure 7. Latency distributions of NEO and vLLM in A10G+Llama-3.1-8B+AC setting at request rate of 1.6/s. The distributions are both skewed because of the skewed request length distribution of the dataset.



(a) Online latency.

(b) Offline throughput.

Figure 8. Comparison of NEO and FastDecode$^+$ in 2xH100+Llama-3.1-70B setting. Figure (a) compares their latency on the AC dataset. Figure (b) shows their relative throughput to the baseline, where we fix the input length to 2000 and vary the output length. We use the GPU-only version of NEO (i.e., SwiftLLM) as the baseline.



(a) 2xH100 + Llama-3.1-70B.

(b) A10G + Llama-3.1-8B.

(c) T4 + Llama-2-7B.

Figure 9. Relative throughput in different settings and different synthetic workloads, with GPU-only NEO (i.e., SwiftLLM) as the baseline.

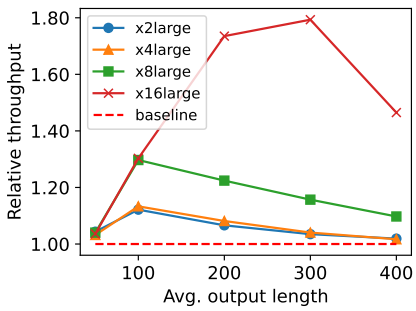
to GPU-only mode. However, as output length grows, FastDecode$^+$ becomes CPU bounded, and its performance drops quickly to less than 60% of baseline's performance.

5.4 Varying Input/Output Lengths

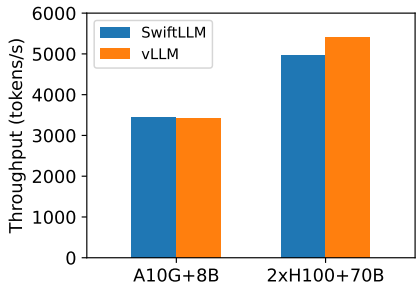
We further examine NEO's performance over various workloads with different input/output lengths. As shown in Figure 9, where we fix input length and tweak output length, NEO achieve up to 14%, 26%, and 750% throughput gains on H100, A10G, and T4, respectively. When the output length is short, NEO may perform slightly worse than the baseline because NEO still attempts to put some requests onto the CPU and swap them back in later, incurring slight overheads. As the output length increases, NEO's gains first grow to the maximum point, where GPU and CPU times are exactly balanced, then gradually drop as the system launches a larger proportion of GPU-only batches. NEO performance would be close to the baseline when output length gets large enough, sometimes slightly worse due to suboptimal scheduling decisions caused by the inevitable inaccuracy of the offline performance profiling.

5.5 Sensitivity Study

Impact of CPU capacity. We first study the impact of CPU capacity on throughput gains. To examine the impact of CPU memory bandwidth, we use 4 kinds of AWS EC2 instances, namely g5.2xlarge, g5.4xlarge (what we used in previous experiments), g5.8xlarge, and g5.16xlarge. These instances all have 1 A10G GPU and thus the same baseline (non-offloading) performance. However, their offloading performance varies due to different CPU cores, memory sizes, and memory bandwidths. A g5.$n$xlarge generally has $2n$ CPU cores (i.e., $4n$ hyperthreads) and $16n$ GB CPU memory; g5.2xlarge has nearly the same peak memory bandwidth as g5.4xlarge, while g5.8xlarge has about twice the bandwidth of g5.4xlarge, and g5.16xlarge has about twice the bandwidth of g5.8xlarge. In experiments, we set the CPU KV cache size proportionally to the number of cores. Figure 10a shows the results. NEO achieves up to 12.2%, 13.3%, 29.7%, and 79.3% higher throughput over the baseline under different CPU capacities. When the output length is short, these instances have nearly the same throughput gains as the workload mainly runs on the GPU. As output length increases, the instances with less CPU memory bandwidth start to drop performance earlier. The peak throughput



(a) CPU sensitivity.



(b) SwiftLLM vs. vLLM.

Figure 10. Sensitivity study. Figure (a) shows the relative throughput of NEO on different AWS EC2 `g5` instances, with GPU-only NEO (i.e., SwiftLLM) as the baseline. Figure (b) compares SwiftLLM's throughput on the AC dataset with vLLM in both multiple and single GPU settings. Note that we implement the multi-GPU support for SwiftLLM (§4).

gain is positively related to the CPU memory bandwidth. This supports the fact that the memory bandwidth, rather than computing power (i.e., number of cores), is the factor that determines the performance of attention operation on CPUs.

SwiftLLM vs. vLLM. We now examine the gap between our baseline system and vLLM. We feed the Azure Code trace all at once to both systems and examine the GPU token throughput, i.e., the total time elapsed divided by the total number of tokens processed (input length + output length). We evaluate in both single-GPU (A10G + Llama-3.1-8B) and multi-GPU (2xH100 + Llama-3.1-70B) settings.

Figure 10b shows the results. As SwiftLLM is initially targeted at single-GPU inference and mimics vLLM implementation, it achieves comparable throughput with vLLM in a single A10G + Llama-3-8B setting. However, SwiftLLM is slightly worse than vLLM (8.8% lower throughput) in the 2-GPU setting; this is due to our less optimized tensor parallelism implementation compared to production-grade vLLM. We leave as future work integrating NEO into vLLM to measure our 4-GPU and 8-GPU performance gains.

6 DISCUSSION

Compare to chunked-prefill. NEO's performance benefits are essentially two-fold: 1) larger GPU batch size, and 2) shifting some unbatchable decoding attention operations to CPU, thus making up room on GPU for other batchable operations. The second benefit source shares a similar spirit as the chunked-prefill technique in Sarathi-Serve (Agrawal et al., 2024), which breaks a prompt into multiple chunks to launch more prefill-decode mixed batches and thus less decoding-only batches (that contain unbatchable decoding attention operations).

However, chunked-prefill suffers from several drawbacks that NEO does not have. First, chunked-prefill consumes significantly more GPU memory bandwidth, because the KV cache of all previous chunks needs to be loaded repetitively to compute for the subsequent chunk (Zhong et al., 2024). Second, chunking-prefill would not work well on memory-constrained GPUs, because the resulting small batch size would limit the opportunity of piggybacking decode on prefill chunks while saturating the GPU. In contrast, NEO relies on large CPU memory to ensure throughput gains. Readers can refer to appendix B for more experimental results.

Nevertheless, we believe NEO could integrate with chunked-prefill NEO can compute all chunked prefilling on the GPU, and doesn't offload the KV-cache until the full prompt of a request is prefilled. With chunked prefill, we can modify the step 5 of NEO's load-aware scheduling: instead of removing the whole prefilling requests, NEO could remove chunks of prefilling requests, thereby having a finer-grained control for CPU-GPU balancing. Chunked prefill would be especially useful for NEO when context length gets very large while GPU memory is limited.

Offloading other parts to CPU. NEO implicitly assumes putting all model weights on the GPU and only offloading attention computation to the CPU is the most efficient way to balance GPU and GPU loads. This is because of, as mentioned in section 2, the huge gap between the CPU and GPU's computing power. Meanwhile, according to profiling results of Table 2 in (Zhu et al., 2024), all non-communication operations are compute-bounded except decoding attention. However, in prior work like (Sheng et al., 2023) and (Song et al., 2024), people offload components other than attention to the CPU. This is not only useful when the GPU's memory is too constrained to hold all model weights, but also potentially beneficial even if the GPU could hold all the model weights. For example, when requests in the workload have too few output tokens, NEO would be bounded by GPU computation, while the CPU is mostly idle. Therefore, offloading some of the dense operations to the CPU could alleviate the GPU's pressure in these extreme workloads. Nevertheless, the actual gain needs to be validated by further exploration.

NEO usage scenarios. NEO works best in scenarios where the GPU memory is constrained such that it limits the batch size and underutilizes the GPU compute. These scenarios will likely hold for a long time, as the GPU compute capacity continues growing while its memory size stays relatively stagnant, e.g., H100 triples the compute of A100 but with the same 80GB memory size (NVIDIA, 2022; 2024). NEO will degrade to non-offloading mode when the GPU has enough memory to reach a batch size that can saturate the GPU. Another usage scenario of NEO is the economic serving of LLM models by leveraging cheap and abundant CPU resources that already exist in current datacenters.

Using remote CPUs. NEO focuses on improving inference throughput by only using the host CPU. In order to gain more throughput, NEO could be extended to support remote CPU workers. However, CPU memory bandwidth in current commercial clouds is still expensive, and as (He & Zhai, 2024) shows, cross-machine transfer latency could be yet another bottleneck. We leave this for future work.

Money and energy cost NEO targets the following scenarios: typical public cloud like AWS, Azure, and GCP allocates host CPU proportionally with the GPU capacity and count, and such CPU resources have been calculated into the VM price. In these scenarios, NEO tries to exploit these host CPU resources (that **were not** effectively used and thus wasted) to accelerate LLM serving. We note that all our performance comparisons on A10G and T4 GPUs in the paper are done using the same standard AWS instances for both NEO and GPU-only; therefore, from the perspective of cloud users, NEO does not incur additional costs. But we do agree that for on-premises testbeds, NEO may not achieve better perf-TCO.

NEO achieves performance gains mainly through utilizing the host CPU, which will increase energy usage compared to GPU-only baselines. However, from the perspective of cloud users, the energy cost of the host CPU has already been calculated and added to the GPU VM price. From the environmental perspective, the throughput-per-watt of GPU-CPU offloading may not be superior to pure GPU in all cases; but in the case of T4 GPU, given our 8x throughput improvement, GPU-CPU offload will achieve better throughput-per-watt. We intend to leave it as future work to study the broad topic of energy consumption in LLM serving.

7 Related Work

GPU-efficient LLM inference. There is a line of work on optimizing the efficiency of LLM inference on GPUs, including general inference systems from Orca (Yu et al., 2022), vLLM (Kwon et al., 2023), SGLang (Zheng et al., 2023), FastServe (Wu et al., 2023), Sarathi-Serve (Agrawal et al., 2024), NanoFlow (Zhu et al., 2024), DistServe (Zhong et al., 2024), and more (Patel et al., 2024; Strati et al., 2024; Hu et al., 2024), and low-level GPU kernel optimizations from FlashDecoding (Dao et al.), FlashDecoding++ (Hong et al., 2023), and FlashInfer (flashinfer ai/flashinfer). NEO leverages several techniques from these work such as selective batching (Yu et al., 2022) and paged attention (Kwon et al., 2023), and could be used in parallel with others, e.g., leveraging NEO to optimize the decoding phase in DistServe (Zhong et al., 2024). Another line of work leverages sparcification and quantization techniques to trade accuracy for performance, including AWQ (Lin et al., 2024), SparseGPT (Frantar & Alistarh, 2023), AlphaTuning (Kwon et al., 2022), GPT3.int8() (Dettmers et al., 2022), GPTQ (Frantar et al., 2022), ZeroQuant (Yao et al., 2022), SmoothQuant (Xiao et al., 2023a), StreamingLLM (Xiao et al., 2023b), and more (Beltagy et al., 2020; Shen et al., 2020; Hoefler et al., 2021). Different from them, NEO does not compromise accuracy.

Offloading for LLM serving. Many existing work offloads LLM models, activations, KV cache, or computations to the CPU for offline scenarios that trade latency for throughput, such as FlexGen (Sheng et al., 2023), HeteGen (Zhao et al., 2024a), PowerInfer (Song et al., 2024), and TwinPilots (Yu et al., 2024). InstInfer (Pan et al., 2024) further offloads to computational SSD to lower the inference cost. FastDecode (He & Zhai, 2024) targets similar online scenarios as NEO, but lacks critical designs to address the load imbalance between the GPU and CPU (see §1).

8 Conclusion

NEO is a CPU offloading system for online LLM inference to increase GPU batch sizes and improve inference throughput. It features asymmetric pipelining and load-aware scheduling to fully leverage both GPU and CPU resources without overloading them. NEO achieves up to 14%-7.5× (depending on GPUs) higher throughput than GPU-only inference systems across a variety of workloads and model sizes, while maintaining the same latency. We will open source NEO codebase to encourage more research on cost-efficient LLM inference.

Acknowledgments

Xuanlin Jiang and Minlan Yu are supported in part by NSF NeTS 2107078. Yang Zhou, Shiyi Cao, and Ion Stoica are supported in part by gifts from Accenture, AMD, Anyscale, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, and VMware.

REFERENCES

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B., Tumanov, A., and Ramjee, R. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In Proceedings of USENIX OSDI, pp. 117–134, 2024.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The Long-Document Transformer. arXiv preprint arXiv:2004.05150, 2020.

CarperAI. OpenAI summarize comparisons. https://huggingface.co/datasets/CarperAI/openai_summarize_comparisons.

Chen, S., Lin, Y., Zhang, M., and Wu, Y. Efficient and Economic Large Language Model Inference with Attention Offloading. arXiv preprint arXiv:2405.01814, 2024.

Dao, T., Haziza, D., Massa, F., and Sizov, G. Flash-Decoding for Long-Context Inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html.

Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. GPT3.int8(): 8-bit Matrix Multiplication for Transformers at Scale. Advances in Neural Information Processing Systems, 35:30318–30332, 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.

flashinfer ai/flashinfer. FlashInfer: Kernel Library for LLM Serving. https://github.com/flashinfer-ai/flashinfer/.

Frantar, E. and Alistarh, D. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. In International Conference on Machine Learning, pp. 10323–10337. PMLR, 2023.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv preprint arXiv:2210.17323, 2022.

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

Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. Journal of Machine Learning Research, 22(241):1–124, 2021.

Hong, K., Dai, G., Xu, J., Mao, Q., Li, X., Liu, J., Chen, K., Dong, Y., and Wang, Y. FlashDecoding++: Faster Large Language Model Inference on GPUs. arXiv preprint arXiv:2311.01282, 2023.

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, 2024.

interestingLSY/swiftLLM. SwiftLLM. https://github.com/interestingLSY/swiftLLM.

ispc/ispc. Intel Implicit SPMD Program Compiler. https://github.com/ispc/ispc.

Kwon, S. J., Kim, J., Bae, J., Yoo, K. M., Kim, J.-H., Park, B., Kim, B., Ha, J.-W., Sung, N., and Lee, D. AlphaTuning: Quantization-Aware Parameter-Efficient Adaptation of Large-Scale Pre-Trained Language Models. arXiv preprint arXiv:2210.03858, 2022.

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 ACM SOSP, pp. 611–626, 2023.

Lin, J., Tang, J., Tang, H., Yang, S., Chen, W.-M., Wang, W.-C., Xiao, G., Dang, X., Gan, C., and Han, S. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. Proceedings of Machine Learning and Systems, 6:87–100, 2024.

Microsoft. Azure LLM inference trace 2023. https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2023.md.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., et al. Ray: A distributed framework for emerging {AI} applications. In Proceedings of USENIX OSDI, pp. 561–577, 2018.

NVIDIA. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf, 2022.

NVIDIA. NVIDIA H100 Tensor Core GPU. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet, 2024.

Pan, X., Li, E., Li, Q., Liang, S., Shan, Y., Zhou, K., Luo, Y., Wang, X., and Zhang, J. Instinfer: In-Storage Attention Offloading for Cost-Effective Long-Context LLM Inference. arXiv preprint arXiv:2409.04992, 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, S., Dong, Z., Ye, J., Ma, L., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 34, pp. 8815–8821, 2020.

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. 2024.

Strati, F., Mcallister, S., Phanishayee, A., Tarnawski, J., and Klimovic, A. DejaVu: KV-cache Streaming for Fast, Fault-tolerant Generative LLM Serving. arXiv preprint arXiv:2403.01876, 2024.

Tillet, P. Introducing Triton: Open-source GPU programming for neural networks. https://openai.com/index/triton/.

Tillet, P., Kung, H.-T., and Cox, D. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, pp. 10–19, 2019.

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.

WikiChip. AWS Graviton4 - Annapurna Labs (Amazon). https://en.wikichip.org/wiki/annapurna_labs/graviton/graviton4.

Wu, B., Zhong, Y., Zhang, Z., Huang, G., Liu, X., and Jin, X. Fast Distributed Inference Serving for Large Language Models. arXiv preprint arXiv:2305.05920, 2023.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. In International Conference on Machine Learning, pp. 38087–38099. PMLR, 2023a.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient Streaming Language Models with Attention Sinks. arXiv preprint arXiv:2309.17453, 2023b.

Yao, Z., Yazdani Aminabadi, R., Zhang, M., Wu, X., Li, C., and He, Y. ZeroQuant: Efficient and Affordable Post-Training Quantization for Large-Scale Transformers. Advances in Neural Information Processing Systems, 35: 27168–27183, 2022.

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

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A Distributed Serving System for Transformer-Based Generative Models. In Proceedings of USENIX OSDI, pp. 521–538, 2022.

Zhao, X., Jia, B., Zhou, H., Liu, Z., Cheng, S., and You, Y. HeteGen: Heterogeneous Parallel Inference for Large Language Models on Resource-Constrained Devices. arXiv preprint arXiv:2403.01164, 2024a.

Zhao, Y., Lin, C.-Y., Zhu, K., Ye, Z., Chen, L., Zheng, S., Ceze, L., Krishnamurthy, A., Chen, T., and Kasikci, B. Atom: Low-Bit Quantization for Efficient and Accurate LLM Serving. Proceedings of Machine Learning and Systems, 6:196–209, 2024b.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. SGLang: Efficient Execution of Structured Language Model Programs. arXiv preprint arXiv:2312.07104, 2023.

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. In Proceedings of USENIX OSDI, pp. 193–210, 2024.

Zhu, K., Zhao, Y., Zhao, L., Zuo, G., Gu, Y., Xie, D., Gao, Y., Xu, Q., Tang, T., Ye, Z., et al. NanoFlow: Towards Optimal Large Language Model Serving Throughput. arXiv preprint arXiv:2408.12757, 2024.

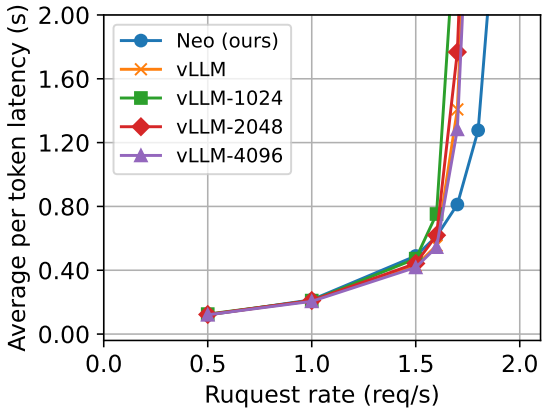
A Pseudo Code for Load-Aware Scheduling

Here we show the pseudo-code for the load-aware scheduling algorithm mentioned in section 3.2. The code is written in Python-like syntax.

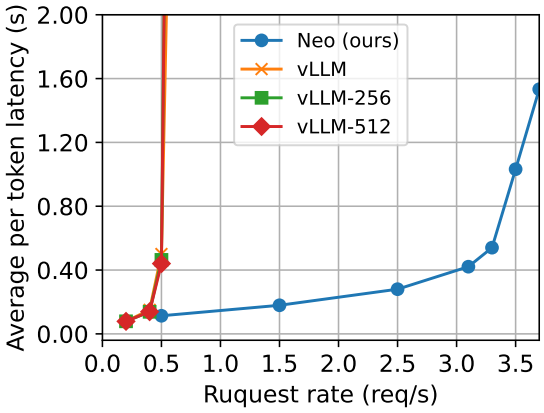
```
# 1. Initialization
batches = [EMPTY_BATCH] * 2

# 2. Schedule GPU decoding requests
while NEED_SWAP_OUT_LAST_GPU_DECODE_REQ:
    req = gpu_decoding_q.pop()
    cpu_decoding_q.prepend(req)
    swap_out(req)

while CAN_SWAP_IN_LAST_CPU_DECODE_REQ:
    req = cpu_decoding_q.popfront()
    gpu_decoding_q.append(req)
    swap_in(req)

for req in gpu_decoding_q:
 batches[0].add_gpu_decode(req)

# 3. Schedule prefilling requests
for req in waiting_q:
 if CAN_PREFILL_TO_GPU(req):
        gpu_decoding_q.append(req)
     batches[0].add_gpu_prefill(req)
    elif CAN_PREFILL_TO_CPU(req):
      cpu_decoding_q.append(req)
     batches[0].add_cpu_prefill(req)
    else:
     break

# 4. Schedule CPU decoding requests
for req in cpu_decoding_q:
    for i in [1, 0]:
        batches[i].add_cpu_decoding(req)
        if cpu_time_exceed_gpu(batches):
            batches[i].pop_cpu_decoding()
        else:
            break

# 5. Reduce prefilling requests
gpu_only_batch =
    batches[0].remove_all_cpu_decodings()
reduce_cpu_prefill(batches[0])
reduce_cpu_prefill(gpu_only_batch)

# 6. Make decisions
if token_rate([gpu_only_batch]) > \
    token_rate(batches):
    batches = [gpu_only_batch]
update_queues(
    batches,
    waiting_q,
    gpu_decoding_q,
    cpu_decoding_q
)

return batches
```

B Comparison with Chunked Prefill

We further conduct more experiments to compare NEO with chunked prefill by leveraging the chunked prefill implementation in vLLM. As Figure 11 shows, chunked prefill slightly reduces the latency, but achieves a similar maximum throughput with vLLM which is lower than NEO. The reason why chunked prefill has little impact on perfomance is that the GPU memory is so limited that the number of piggybacked decoding requests with a chunked prefilling request, the main source of gain, cannot be sufficiently large.



(a) A10G + LLaMa-3.1-8B.



(b) T4 + LLaMa-2-7B.

Figure 11. Load-latency curve comparison between NEO and vLLM with chunked-prefilling functionality. The numbers after vLLM stand for the chunk size, e.g., vLLM-256 for vLLM under chunk size of 256.

C ARTIFACT APPENDIX

C.1 Abstract

This appendix describes the complete workflow of running NEO and generating some representative results used in this paper, including Figure 6c and Figure 10a

C.2 Artifact check-list (meta-information)

- **Algorithm:** Continuous batching; paged attention; grid profiling
- **Compilation:** Both GNU g++-11 and g++-13 compilers are required. CMake must be version 3.18 or higher. The Intel ISPC compiler should be version 1.23. The Nvidia nvcc compiler must be version 12.2 or later.
- **Data set:** OpenAI summarization comparison (OSC) (CarperAI); synthetic workloads
- **Run-time environment:** Ubuntu 22.04
- **Hardware:** AWS g4dn.4xlarge instance; AWS g5.16xlarge instance
- **Execution:** Automatic scripts are included (Please refer to README.md of the repository for details).
- **Metrics:** Average per token latency; relative throughput to baseline
- **Output:** Plots in pdf files and evaluation traces including numerical results under NEO/evaluation
- **How much disk space required (approximately)?:** 15-20GB (mainly model weights)
- **How much time is needed to prepare workflow (approximately)?:** less than 30 minues
- **How much time is needed to complete experiments (approximately)?:** 10-15h for full reproduction; less than 1h for reduced reproduction
- **Publicly available?:** https://github.com/NEO-MLSys25/NEO
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.14964833

C.3 Description

C.3.1 How delivered

The code can be downloaded through a public Github repository.

C.3.2 Hardware dependencies

AWS g4dn.4xlarge and g5.16xlarge instances.

C.3.3 Software dependencies

- Both g++-11 and g++-13
- CMake 3.18 or later
- Intel ISPC compiler 1.23
- Nvidia CUDA toolkit 12.2 or later
- OpenMP library
- Python 3.10 or later
- Pytorch 2.4 or later
- vLLM
- Other python dependencies described in requirement.txt of the repository

C.3.4 Data sets

We use a subset of OpenAI summarization comparison (OSC) (CarperAI) for Figure 6c and synthetic workloads for 10a. Please refer to 5 for more details.

The size of the datasets can be reduced manually for a faster run at the price of accuracy. Please refer to the README in our repository for more details.

C.4 Installation

1. Launch a g4dn.4xlarge/g5.16xlarge instance with AWS community AMI named neo-ae-g4-image/neo-ae-g5-image in us-east-1 region. Then all you need to do is run `mamba activate neo` and `cd NEO` in your login shell. The dependencies should already be installed, and you can skip the steps below. However, it is still recommended to re-download the model weights for faster evaluation. Please refer to the README document of our code repository for details.
2. If you do not have access to the AMI. Please launch the same instances using the "amazon/Deep Learning OSS Nvidia Driver AMI GPU PyTorch 2.6.0 (Ubuntu 22.04)" AMI and run the following commands in your shell to install NEO and its dependencies:

```
sudo add-apt-repository \
    ppa:ubuntu-toolchain-r/test
sudo apt update
sudo apt install g++-13 libomp-dev numactl
sudo snap install ispc --channel latest/edge
git clone \
    https://github.com/NEO-MLSys25/NEO.git
cd NEO
pip install -r requirements.txt
pip install -e csrc
pip install -e .
cd pacpu
bash build.sh llama2_7b 1
cd ..
```

   NOTE: Change the second last line to `bash build.sh llama3_8b 1` on the g5.16xlarge instance.
3. Prepare LLaMa-2-7B/LLaMa-3-8B model weights. You can download it from Huggingface or Meta's official repository. Note that the weights should be in .safetensors format. Please also change the "model_path" entry in evaluation/configs/config-t4-7b.json on g4dn.4xlarge or evaluation/configs/config-a10-8b.json on g5.16xlarge to the actual path to the model weigths.
4. For reproducing Figure 6c, please also install vLLM by `pip install vllm`. Please note that our original results were obtained with vLLM 0.6.3.post1. However, installing this version directly via pip may lead to an SQL-related bug. The latest vLLM (0.7.3 by March 2nd, 2025) does not exhibit this issue and demonstrates comparable performance to vLLM 0.6.3.post1 in our experiments.

C.5 Experiment workflow

On g4dn.4xlarge instance:

```
cd NEO
python evaluation/reproduce-fig6c.py
```

On g5.16xlarge instance:

```
cd NEO
python evaluation/reproduce-fig10a.py
```

C.6 Evaluation and expected result

Reproducing Figure 6c Running the reproduce-fig6c.py script on g4dn.4xlarge instance with proper environment setup will reproduce this figure. By default, the script only uses a small subset (100 requests) of the original input data (2000 requests) used in the original experiment. This is for the purpose of demonstration and quick verification of the results for faster evaluation. As a result, the latency would be lower than the original figure due to the lower average queuing latency. You can change the number in `evaluation/benchmark.py`, line 114 to use more requests for more accurate results.

Reproducing Figure 10a Running the reproduce-fig10a.py script on g5.16xlarge instance with proper environment setup will reproduce this figure. Only the "x16large" and the "baseline" lines in the original figure will be drawn. The default number of requests is 2000, the same as in the original paper. You can reduce the number for faster evaluation. However, the result may not be as accurate as the original one because of warm-up and cool-down effects. It is not recommended to set this number below 800.

C.7 Experiment customization

Users can reduce or add custom data points through very straightforward modifications of the scripts. Users can also modify the number of data used in experiments. See the code for more detailed instructions.

C.8 Notes

On the first run of NEO, the server process starts very slowly for two reasons. First, it needs to load the model weights from storage to the main memory. Secondly, it might (not if you use our provided AMI) need to do the initial performance profiling before starting the service. The whole process would take less than ten minutes. You can check the logs in NEO/evaluation for details if you find no output from the console.

C.9 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging