2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA) | 979-8-3503-2658-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISCA59077.2024.00019

Splitwise: Efficient Generative LLM Inference Using Phase Splitting

Pratyush Patel[1], Esha Choukse[2], Chaojie Zhang[2],
Aashaka Shah[2], Íñigo Goiri[2], Saeed Maleki[2], Ricardo Bianchini[2]

[1]University of Washington [2]Microsoft

***Abstract*—Generative large language model (LLM) applications are growing rapidly, leading to large-scale deployments of expensive and power-hungry GPUs. Our characterization of LLM inference shows that each inference request undergoes two phases: a compute-intensive prompt computation phase and a memory-intensive token generation phase, each with distinct latency, throughput, memory, and power characteristics. Despite state-of-the-art batching and scheduling, the token generation phase underutilizes compute resources. Unlike prompt computation, token generation does not need the compute capability of the latest GPUs and can be run with lower power and cost.**

Based on these insights, we propose Splitwise, a model deployment and scheduling technique that splits the two phases of LLM inference requests on to separate machines. Splitwise enables phase-specific resource management using hardware that is well suited for each phase. Request state is transferred efficiently between machines using optimized network libraries on the fast back-plane interconnects available in today's GPU clusters. Using Splitwise, we design homogeneous and heterogeneous LLM inference clusters optimized for throughput, cost, and power. Compared to current designs, Splitwise clusters achieve up to $1.4\times$ higher throughput at 20% lower cost. Alternatively, they can deliver $2.35\times$ more throughput under the same power and cost budgets.

I. Introduction

Recent advancements in generative large language models (LLMs) have significantly improved their response quality and accuracy [18], [71]. These trends have led to the widespread adoption of LLMs across various domains [6], [21]. Most modern LLMs are built using the transformer architecture [77], [78] and exhibit similar characteristics [63]. Transformer model sizes have grown steadily, from the early BERT models [36] having 340 million parameters, to GPT-3 [28] with a staggering 175 billion parameters, and GPT-4 rumored to have even more.

LLMs typically run on expensive and power-hungry GPUs [16]. The sudden and large-scale deployment of LLMs has led to a worldwide GPU capacity crunch [14]. The computational demand for LLM inference far exceeds that of training due to the vast number of applications leveraging LLMs. Furthermore, since training LLMs requires expensive and dedicated supercomputers [56], [60], a large number of inferences are necessary to amortize the high training costs. LLM inference jobs, although orders of magnitude smaller than training, are still expensive given the compute involved.

[1]Work partly done as an intern at Microsoft.

	A100	H100	Ratio
TFLOPs	19.5	66.9	3.43×
HBM capacity	80GB	80GB	1.00×
HBM bandwidth	2039GBps	3352GBps	1.64×
Power	400W	700W	1.75×
NVLink	50Gbps	100Gbps	2.00×
Infiniband	200GBps	400GBps	2.00×
Cost per machine [5]	$17.6/hr	$38/hr	2.16×

TABLE I: NVIDIA A100 vs. H100 specifications.

Generative LLM inference for a single request consists of several forward passes through the model, since the output tokens are generated one by one. This inherently has two contrasting phases of computation. First, the *prompt computation phase*, in which all the input prompt tokens run through the forward pass of the model in parallel to generate the first output token. This phase tends to be computationally intensive and requires the high FLOPs (floating point operations per second) of the latest GPUs today. Second, the *token generation phase*, in which subsequent output tokens are generated sequentially based on the forward pass of the last token and all the cached context from previous tokens in the sequence. Given the lack of compute parallelism, this phase tends to be more memory bandwidth and capacity bound, despite state-of-the-art batching. Running both phases on the same machine often leads to inconsistent end-to-end latencies due to the arbitrary batching of prompt and token phases. Due to these challenges, services need to over-provision expensive GPUs to meet tight inference service level objectives (SLOs) for interactive applications. At the same time, cloud service providers (CSPs) are having to build a lot of new datacenters to meet the GPU demand, and are running into a power wall [19].

The industry continues to release new computationally powerful GPUs, each much more power hungry and expensive than the last. However, as shown in Table I, the high-bandwidth memory (HBM) capacity and bandwidth on these GPUs has not scaled at the same rate recently. The latest NVIDIA H100 GPUs have $3.43\times$ more compute and $1.75\times$ more power compared to their predecessor A100 GPUs. However, their memory bandwidth only grew by $1.6\times$, with no increase in memory capacity.

Our work. Given the distinct properties of prompt computation and token generation phases, we propose splitting the inference

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00019

request and running them on separate machines. Doing so allows us to separately manage hardware resources for each phase, thereby increasing the GPU utilization and the overall efficiency of the system. It also enables using different, better-suited hardware for each phase. To realize such a setup, the cached context from the prompt computation needs to be communicated over from the prompt processing machine to the token generation machine at low latency. We implement these transfers in an optimized manner over the back-end Infiniband interconnects avaialble in datacenters today, allowing us to increase efficiency without any perceived performance loss.

With Splitwise, we design clusters optimized for cost, throughput, and power, using production traces of LLM inference requests [4]. Given the diverging memory and compute scaling rates across GPU generations, we also evaluate different GPUs and power caps for the different inference phases. This allows us to target better performance per dollar (Perf/$) for users, and better performance per watt (Perf/W) for CSPs. Additionally, users can target older GPUs, which are likely more readily available to them.

We show that Splitwise-based LLM inference clusters can achieve 1.4× higher throughput at 20% lower cost than existing clusters. Alternatively, they can deliver 2.35× more throughput with the same cost and power budgets.

Summary. We make the following contributions:

1) An extensive characterization of the differences in the execution and utilization patterns of the prompt and token generation phases in LLM inference on the NVIDIA A100 and H100 GPUs using production traces.
2) Splitwise, our technique for optimized utilization of available hardware, which splits the prompt computation and token generation phases onto separate machines.
3) A design exploration of homogeneous and heterogeneous cluster deployments with Splitwise to optimize the overall cost, request throughput, and provisioned power.
4) An evaluation of the systems designed with Splitwise using production traces.

II. Background

A. Large Language Models

Modern LLMs are based on transformers. Transformer models use attention [77] and multi-layer-perceptron layers to understand the inputs and generate an output, respectively. Transformer-based LLMs include encoder-only [36], [54], decoder-only [67], [69], [71], and encoder-decoder [70] models. Generative LLMs, the focus of this paper, are usually either decoder-only, or encoder-decoder models.

B. Generative LLM inference phases

Figure 1 shows an example of generative LLM inference. Once the prompt query is received, all the input tokens are computed in parallel, within a single iteration, to generate the first token. We call this the prompt processing phase. The context generated from the attention layers during the prompt computation is saved in the key-value (KV) cache, since it is needed for all the future token generation iterations. After the first token is generated, the following tokens only use the last generated token and the KV-cache as inputs to the forward pass of the model. This makes the subsequent token generation more memory bandwidth and capacity intensive than the computationally heavy prompt phase.

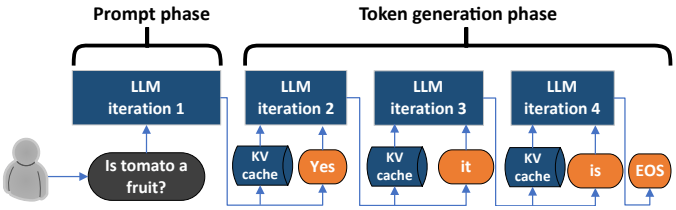


Fig. 1: An LLM inference example.

Metric	Importance to user
End-to-end (E2E) latency	Total query time that the user sees
Time to first token (TTFT)	How quickly user sees initial response
Time between tokens (TBT)	Average token streaming latency
Throughput	Requests per second

TABLE II: Performance metrics for LLMs.

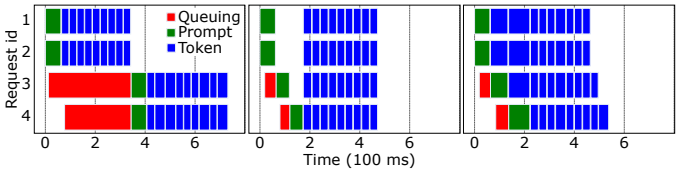
C. Performance metrics for LLMs

Prior work has proposed three main metrics for LLM inference: end-to-end (E2E) latency, time to first token (TTFT), and throughput. We add another latency metric: time between tokens (TBT), to track the online streaming throughput of the tokens as they are generated serially. Table II summarizes the key performance metrics that we consider in this work.

Generative LLMs may be used for a variety of tasks with different kinds of SLOs. For batch tasks (*e.g.*, summarization), TTFT or TBT latency metrics are less important than throughput. On the other hand, for latency-sensitive tasks (*e.g.*, conversational APIs), TTFT and TBT are the more important metrics with tighter SLOs.

D. Batching of requests

Inference requests can be batched together for higher throughput. Several prior works have explored batching [23], [81]. Figure 2 shows the timelines for inference with three common batching mechanisms. The default mechanism only batches at the *request-level* (Figure 2(a)). In this case, ready requests are batched together, but all the forward passes for these requests are completed before any other requests are run. Since requests can have long token generation phases, this can lead to long wait times for requests arriving in between, causing high TTFT and high E2E latencies. An optimization is *continuous batching* [81] (Figure 2(b)). In this case, scheduling decisions are made before each forward pass of the model. However, any given batch comprises either only of requests in their prompt phase or only requests in token phase. Prompt phase is considered more important since it impacts TTFT. Hence, a waiting prompt can preempt a token phase. Although this leads to shorter TTFT, it can substantially increase the tail



(a) Request-level. (b) Continuous. (c) Mixed.

Fig. 2: Batching mechanisms and their latency impact on the prompt and token phases.

for TBT, and therefore the E2E latency. Finally, there is *mixed batching* (Figure 2(c)) [23]. With this batching, the scheduling decisions are made at each forward pass, and the prompt and token phases can run together. This reduces the impact on TBT, but does not eliminate it, since token phases scheduled with prompt phases will experience a longer runtime. In the rest of the paper, we use mixed batching unless stated otherwise.

E. Model parallelism

Model parallelism can be used to divide a model onto multiple GPUs, and even multiple machines, for higher efficiency and memory capacity. LLM inference typically uses pipeline and tensor parallelism. Pipeline parallelism (PP) divides the layers of the model among the GPUs, while keeping all the operators and tensors within a layer on the same GPU. Tensor parallelism (TP) divides the tensor across the GPUs, while replicating all the layers on each GPU. Pipeline parallelism requires lower communication across the participating GPUs, while tensor parallelism requires high bandwidth communication for each layer. In general, tensor parallelism performs better for GPUs within the same machine, connected with high bandwidth interconnects like *e.g.* NVLink [15]. In the rest of the paper, we use tensor parallelism across 8 GPUs for the best latency.

F. GPU clusters and interconnects

With the recent rise of LLM use cases, several cloud service providers have expanded the GPU-based offerings, leading to large GPU cluster deployments [5], [56], [57]. Each machine in these AI clusters is generally comprised of 8 flagship NVIDIA GPUs (A100 or H100). Each GPU is connected to all the other GPUs in the cluster with a high bandwidth Mellanox InfiniBand interconnect [10], [13], forming a high bandwidth data plane network. The InfiniBand bandwidth offered in the cloud today ranges from 25 to 50GBps per GPU pair [7], [10].

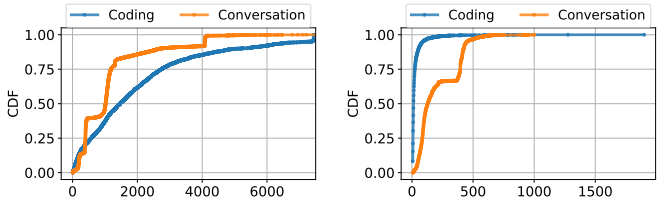
III. Characterization

In this section, we explore the performance and utilization characteristics of LLM inference and draw key insights to guide the design of Splitwise.

Production traces. We use production traces taken from two Azure LLM inference services on November 11^{th} 2023. Our traces represent the most common scenarios in LLM inference today: *coding* and *conversation*. We have released a subset of our traces at https://github.com/Azure/AzurePublicDataset [4]. The traces we use for characterization are 20 minutes long and include the arrival time, input size (number of prompt tokens), and output size (number of output tokens). Due to customer privacy requirements (*e.g.*, GDPR), we do not have visibility into the content of the prompts. We instead use the production traces to guide the input and output sizes, where we send the input prompt with the required number of tokens, and force the model to generate the corresponding number of output tokens for each request. Note that the text of the inputs prompts does not impact the performance metrics that we benchmark, since they depend only on the input and output sizes. For this characterization, we do not reuse the KV-cache between requests to emulate a cloud service with security guarantees.

Model	#Layers	Hidden size	#Heads
Llama2-70B	80	8192	32
BLOOM-176B	70	14336	112

TABLE III: Models we evaluate and their parameters.



(a) Prompt input tokens. (b) Generated output tokens.

Fig. 3: Distribution for prompt and generated tokens.

Models. Table III shows the models that we evaluate. Both BLOOM [69] and Llama2 [71] are state-of-the-art open source LLMs. Both models are decoder-only, transformer-based models. We use the version of each model with the most parameters, since these versions are the most representative for production-class accuracy. Unless stated otherwise, we run BLOOM-176B and Llama-70B on vLLM [51] on a machine with 8 H100 [16] GPUs.

A. Number of prompt and generated tokens

To better understand our traces, we examine the distribution of the number of *prompt input* and *generated output* tokens. Figure 3a shows the distribution of number of prompt tokens. Since the coding LLM inference service is generally used to generate completions as the user is writing code, its input prompt can include large chunks of the code written so far. Thus, it has a large median prompt size of 1500 tokens. On the other hand, the conversation service has a wider range of input prompt tokens since it depends on the user. The median number of prompt tokens for this trace is 1020 tokens.

Figure 3b shows the distribution of the number of generated tokens. Since the coding service typically only generates the next few words in the program as the user types, the median number of output token is 13 tokens. On the other hand, the conversation service has an almost bimodal distribution, with a median of 129 tokens generated.

Insight I: Different inference services may have widely different prompt and token distributions.

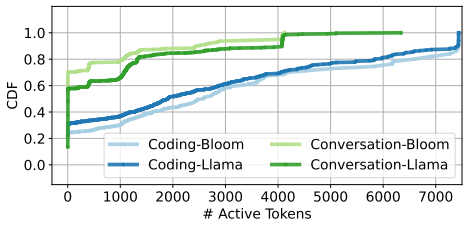
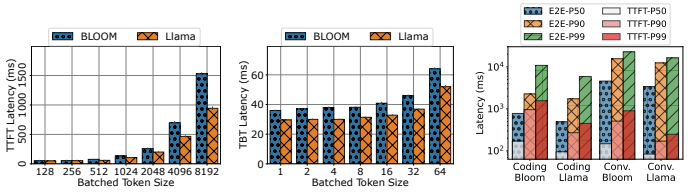
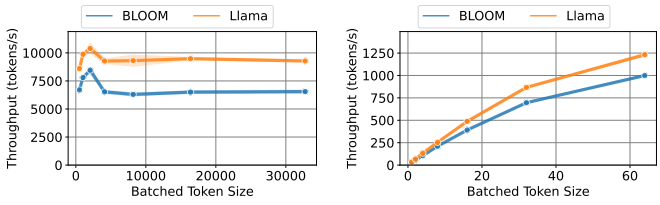


Fig. 4: Cumulative distribution of time spent with various active batched tokens.



(a) TTFT by prompt size. (b) TBT by batch size. (c) Latencies on prod traces (no batching).

Fig. 5: TTFT, TBT, and E2E for BLOOM-176B and Llama-70B on DGX-H100.



(a) Prompt phase. (b) Token generation phase.

Fig. 6: Impact of batching on the throughput for the 2 LLMs.

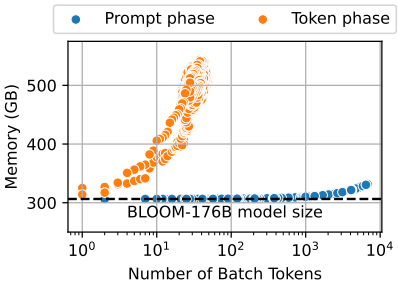


Fig. 7: Required memory with batching in prompt/token phases.

B. Batch utilization

To understand how much can these requests be batched, we measure how often machines run at a given batch size. We use mixed continuous batching as shown in Figure 2. To fit into a single machine, we run a scaled-down version of the coding and conversation traces with 2 requests per second.

Figure 4 shows the distribution of the time spent by the machine running various number of active tokens in a batch. Note that if a prompt of 100 tokens is running in its prompt phase, we count the active tokens as 100. However, once the request is in the token phase, we count it as one active token, since the tokens are generated one at a time (assuming a beam search size of one [51]). We find that most of the time (60–70%) for conversation is spent running only 20 tokens or fewer. Since the coding service has very few output tokens, it experiences even worse batching in the token phase and runs with a single token for more than 20% of the time. Both the LLMs show very similar trends.

Insight II: Mixed continuous batching spends most of the time with very few active tokens batched.

C. Latency

TTFT. Figure 5a shows the impact of the number of prompt tokens on TTFT. The range of sizes was chosen based on the coding and conversation traces. We find that TTFT for both models grows almost linearly as the prompt size increases. This behavior is due to the prompt phase having high GPU utilization and being computationally bound.

TBT. Figure 5b shows the impact of forcefully batching the output tokens of different requests together on the TBT. We observe very little impact on TBT as the batch size grows. With a batch size of 64, there is only 2× impact on TBT.

E2E. Figure 5c shows various percentiles of E2E latency for both models, with no batching. The variability between the request input and output sizes is apparent. Furthermore, we see that most of the E2E time is spent running the token phase. This holds true even for the coding trace, where prompt sizes are large and generated tokens few. In fact, we find that for BLOOM-176B, a prompt phase with 1500 input tokens takes the same time as token phase with only 6 output tokens.

Insight III: For most requests, the majority of the E2E time is spent in the token generation phase.

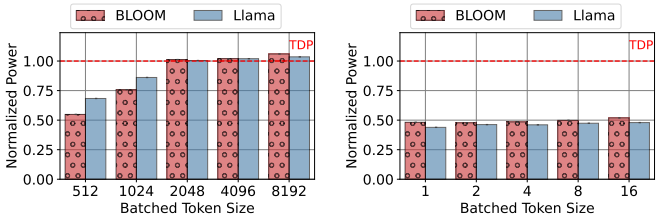
D. Throughput

Figure 6 shows the impact of batching on the throughput (measured as tokens per second). For the prompt phase, we define the throughput as the number of prompt input tokens that are processed per second. We see that the throughput decreases after 2048 prompt tokens, which corresponds to a batch size of less than 2 for the median prompt sizes from the traces. On the other hand, Figure 6b shows that the throughput in the token phase keeps increasing with batching until 64 batch-size, at which point, the machine runs out of memory.

Insight IV: The prompt phase batch size should be limited to ensure good performance. In contrast, batching the token generation phase yields high throughput without any downside.

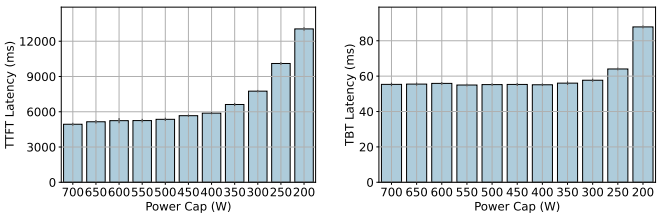
E. Memory utilization

During an LLM inference, the GPU memory is used to host the model weights and activations, as well as the KV caches (Section II-B). As the number of tokens in a batch increase, the memory capacity required for the KV cache also increases. Figure 7 shows the memory capacity utilization during each phase as the number of tokens in the batch increases. During the prompt phase, the input prompt tokens generate the KV



(a) Prompt phase. (b) Token generation phase.

Fig. 8: Maximum and mean power utilization varying the batching size.



(a) Prompt phase. (b) Token generation phase.

Fig. 9: Impact of power cap on the prompt and token generation latency with the maximum batch size possible.

cache. During the output token phase, *each* active generated token that is being processed accesses the KV cache of its *entire context* so far.

Insight V: Batching during the prompt phase is compute-bound, whereas the token phase is limited by memory capacity.

F. Power utilization

When hosting machines, cloud providers need to consider the peak power draw, which has direct impact in the datacenter cost [26]. This is especially important when building GPU clusters, since GPUs consume much higher power than regular compute machines [63], [64]. Figure 8 shows the GPU power draw normalized to the thermal design power (TDP) when running prompt and token generation phases. Since the the prompt phase is compute intensive, its power draw increases with batch size. On the other hand, the token phase is memory bound and its power draw does not vary when increasing the number of tokens to process.

Providers can cap the power usage of the machines to reduce the peak power. Figure 9 shows the impact to latency when increasing the power caps for both prompt and token phases. The prompt phase is highly sensitive to the power cap and the latency increases substantially. On the other hand, the token generation phase incurs almost no latency impact when power capping by over 50% (*i.e.*, 700 to 350W).

Insight VI: While the prompt phase utilizes the power budget of the GPU efficiently, the token phase does not.

G. GPU hardware variations

Given the different characteristics of prompt and token generation phases, we measure the performance impact on the two from running on different hardware. Table I shows the specifications for DGX-A100 [15] and DGX-H100 [16]. The memory-to-compute ratio favors A100 over H100. Table IV shows our findings. We see a lower performance impact on the token generation phase (TBT) as compared to the Prompt phase (TTFT). Since coding requests are dominated by prompt phase, by having very few generated tokens, the E2E latency impact from A100 is worse on coding than conversation. Furthermore, we see that A100 has better or equal inference cost and energy overall compared to H100.

	Coding			Conversation		
	A100	**H100**	**Ratio**	**A100**	**H100**	**Ratio**
TTFT	185 ms	95 ms	0.51×	155 ms	84 ms	0.54×
TBT	52 ms	31 ms	0.70×	40 ms	28 ms	0.70×
E2E	856 ms	493 ms	0.58×	4957 ms	3387 ms	0.68×
Cost [5]	$0.42	$0.52	1.24×	$2.4	$3.6	1.5×
Energy	1.37 Whr	1.37 Whr	1×	7.9 Whr	9.4 Whr	1.2×

TABLE IV: P50 request metrics on A100 vs. H100 without batching on Llama-70B.

Insight VII: Token generation can be run on less compute-capable hardware for better Perf/W and Perf/$ efficiencies.

IV. Splitwise

Based on our characterization insights, we propose Splitwise, a technique to split the prompt and generation phases in the LLM inference on to separate machines.

Figure 10 shows the high-level overview of Splitwise. We maintain two separate pools of machines for prompt and token processing. A third machine pool, the mixed pool, expands and contracts as needed by the workload. All machines are pre-loaded with the model of choice. When a new inference request arrives, the scheduler allocates it to a pair of machines (*i.e.*, prompt and token). The prompt machines are responsible for generating the first token for an input query, by processing all the input prompt tokens in the prompt phase and generating the KV-cache. The prompt machine also sends over the KV-cache to the token machine, which continues the token generation until the response is complete. We use continuous batching at the token machines to maximize their utilization. Machines in mixed pool use mixed continuous batching.

At a lower request rate, we target better latency in Splitwise, while, at a higher request rate, we target avoiding any performance or throughput reduction due to the fragmentation between prompt and token machine pools.

Splitwise uses a hierarchical two-level scheduling as shown in Figure 10. The cluster-level scheduler (CLS) (1) is responsible for machine pool management and for routing incoming inference requests. The machine-level scheduler (MLS) (2) maintains the pending queue and manages batching of requests at each machine.

A. Cluster-level scheduling

Machine pool management. The CLS maintains the prompt, token, and mixed machine pools (3). Splitwise initially assigns

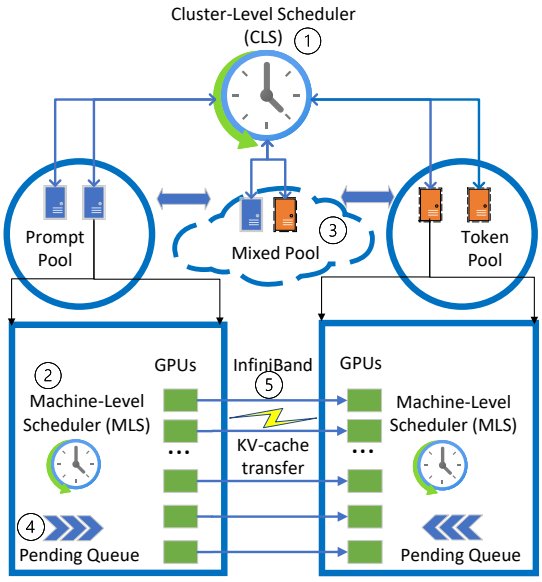
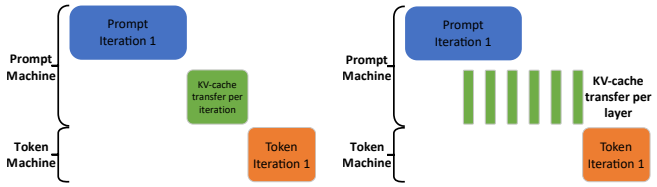


Fig. 10: High-level system diagram of Splitwise.

machines to the prompt or token pool depending on the expected request load and input/output token distributions. Machines from the prompt or token pools may be dynamically moved into and out of the mixed pool to reduce fragmentation and meet SLOs at higher loads. A machine in the mixed pool retains its identity as a prompt or token machine and goes back to its original pool once there are no tasks of the opposite kind in its pending queue. Switching pools does not incur any noticeable latency. If the load distribution deviates considerably from initial assumptions, Splitwise employs a coarse grained *re-purposing of machines* and moves machines between the prompt and token pools. Re-purposing of machines is done infrequently, typically only if they stay in the mixed pool for a considerable amount of time.

Request routing. CLS uses Join the Shortest Queue (JSQ) scheduling [39], [85] to assign a prompt and a token machine to each request. Queue lengths are defined by the number of pending tokens. Each machine regularly communicates to the CLS changes in its memory capacity or pending queue. Note that this does not happen at every iteration boundary. We simultaneously assign both the prompt and token machine when scheduling requests, since we can then overlap KV-cache transfers with prompt computation to reduce transfer overheads (Section IV-C).

When routing requests, if the pending queue is bigger than a certain threshold, the CLS looks for target machines in the mixed pool. If the mixed pool is also full, it proceeds to look in the opposite pool (*i.e.*, a token machine to run prompts and vice versa) and moves the machine into the mixed pool. Machines in the mixed pool operate exactly as a non-Splitwise machine would, with mixed batching. Once the queue of mixed requests is drained, the CLS transitions the machine back to its original pool. For example, when the queue is too long, we can move a prompt machine to the mixed pool to run tokens; once the machine is done running tokens, we transition the machine back into the prompt pool.



(a) Serialized KV-cache transfer. (b) Optimized KV-cache transfer per-layer during prompt phase.

Fig. 11: Optimizing KV-cache transfer in Splitwise.

B. Machine-level scheduling

The MLS runs on each machine and is responsible for tracking the GPU memory utilization, maintaining the pending queue (4), deciding the batch for each iteration, and reporting the relevant status to the CLS.

Prompt machines. The MLS simply uses first-come-first-serve (FCFS) to schedule prompts. The results in Figure 6a show that after 2048 prompt tokens, the throughput degrades. For this reason, the MLS restricts the batching of multiple prompts together to 2048 tokens in total. This is a configurable value, and can change for a different model or hardware.

Token machines. The MLS uses FCFS to schedule tokens and batches as much as possible. Figure 6b shows that the token generation throughput keeps scaling up with the batch size until the machine runs out of memory. For this reason, the MLS tracks the memory and starts queueing tokens once the machine is close to running out of memory.

Mixed machines. To meet the TTFT SLO, the MLS must prioritize running prompts and schedule any new prompts in the pending queue immediately. If the machine is running token phases and has no additional capacity to run the prompt phase, the MLS will *preempt* tokens. To avoid *starvation* of the token phase due to preemption, we increase the priority of the token with age and limit the number of preemptions that each request can have.

C. KV-cache transfer

As discussed in Section II, the KV-cache is generated during the prompt phase of the request, and it continuously grows during the token generation phase. In Splitwise, we need to transfer the KV-cache from the prompt machine to the token machine (5) (shown in Figure 10) to complete the inference. This transfer delay is the main overhead associated with Splitwise. In this section, we discuss the impact of KV-cache transfer and how we optimize it.

Figure 11a shows the Gantt chart for the prompt phase, the KV-cache transfer, and the token generation phase for a single batch of requests when naively transferring the KV cache in a serialized way. The KV-cache transfer starts only after the prompt phase has finished and the first token is generated. Further, it needs to complete before the next output token

can be generated in the token generation phase. This directly impacts the maximum TBT and end-to-end latency of inference.

The time required for the transfer depends on the size of the KV cache (which is directly proportional to the number of prompt tokens) and on the bandwidth of the interconnect between the prompt and the token machines. Even when using fast InfiniBand links, the transfer overhead for large prompt sizes could become a significant fraction of the TBT.

In Splitwise, we optimize the KV-cache transfer by overlapping it with the computation in the prompt phase. As each layer in the LLM gets calculated in the prompt machine, the KV cache corresponding to that layer is also generated. At the end of each layer, we trigger an asynchronous transfer of the KV-cache for that layer while the prompt computation continues to the next layer. Figure 11b shows this asynchronous transfer which reduces the transfer overheads. Layer-wise transfer also enables other optimizations, such as earlier start of the token phase in the token machines, as well as earlier release of KV-cache memory on the prompt machines.

Layer-wise KV-cache transfer happens in parallel with the prompt computation for the next layer. This requires fine-grained synchronization per layer for correctness. Thus, it is possible to incur performance interference and increase the TTFT, especially for smaller prompts. However, for small prompts the total KV-cache size is small and does not need the layer-wise transfer to hide the latency. Since the number of tokens in a batch is already known at the start of computation, Splitwise picks the best technique for KV-cache transfer. It uses serialized KV-cache transfer for smaller prompts and layer-wise transfer and for larger prompts. We show that the overall transfer and interference overheads are relatively small in Section VI-A.

D. Provisioning with Splitwise

We leverage Splitwise to optimize LLM inference cluster deployments for power, cost, and throughput.

Type of machines. We propose four main variants of Splitwise-based systems: *Splitwise-AA*, *Splitwise-HH*, *Splitwise-HA*, and *Splitwise-HHcap*. The nomenclature is simply drawn from the first letter representing the Prompt machine type, and the second letter representing the Token machine type. "A" represents a DGX-A100 machine, "H" represents a DGX-H100 machine, and "Hcap" represents a power-capped DGX-H100 machine. Table V shows a summary of the cost, power, and hardware in each of our evaluated systems.

Splitwise-AA uses DGX-A100 for both prompt and token pools, while Splitwise-HH uses DGX-H100 for both. These two variants represent the commonly available setups in providers where machines are homogeneous and interchangeable.

Splitwise-HA uses DGX-H100 for the prompt pool and DGX-A100 for the token pool. We choose this configuration based on Table IV, and the Insight VII (*i.e.*, A100s can be more cost- and power-efficient for the token phase).

Splitwise-HHcap uses DGX-H100 machines for both prompt and token pools. However, we power cap the token machines down to 70% of their rated power, with each GPU capped by 50% of the power. We propose this design based on Figure 9 and Insight VII (*i.e.*, the prompts phase is impacted by power caps while token has no performance impact with 50% lower power cap per GPU).

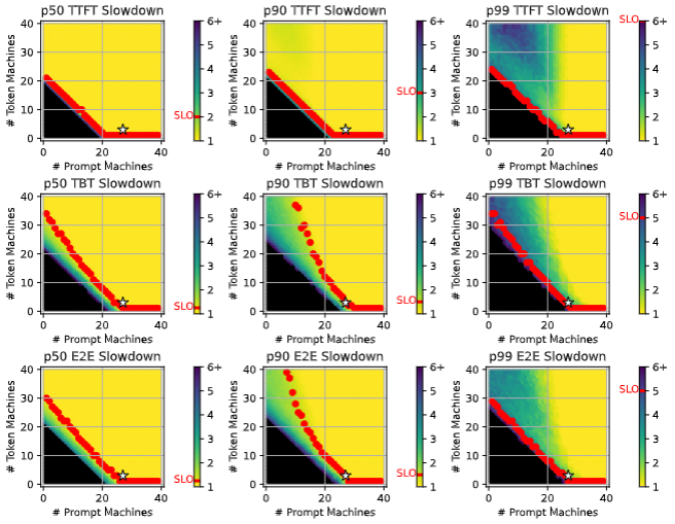


Fig. 12: Design space for provisioning a Splitwise-HH cluster. Cluster configurations targets a peak throughput of 70 RPS. The cost-optimal Splitwise-HH configuration is marked with ⋆ (27 prompt and 3 token machines).

Number of machines. The LLM inference cluster deployment must be sized with the appropriate number of prompt and token machines. Our methodology involves searching the design space using our event-driven cluster simulator, which is described in detail in Section V. We need to provide as input: (1) the target cluster design (*e.g.*, Splitwise-HA or Splitwise-HHcap), (2) an LLM-specific performance model that can estimate the TTFT and TBT at various input, output, and batch sizes, (3) a short trace derived from the target prompt and token size distributions for the service (*e.g.*, Figure 3), (4) the SLOs (*e.g.*, Table VI), (5) the constraints (*e.g.*, throughput), and (6) the optimization goal (*e.g.*, minimize cost). Using this information, our provisioning framework searches the space for the desired optimal point. For example, searching with a throughput constraint and a cost minimization goal gives us iso-throughput cost-optimized clusters across different designs.

Search space. Figure 12 shows an example of the two-dimensional search space for the number of prompt and token machines under Splitwise-HH for the coding workload (using a 2-minute trace). The simulator outputs the various percentiles for TTFT, TBT, and E2E latencies. Then, we select the clusters that meet the SLOs for each of these metrics and optimize our target function. For example, Figure 12 shows a ⋆ for the setup with 27 prompt and 3 token machines with the lowest cost that achieves 70 RPS. We call this setup *iso-throughput cost-optimized*.

Optimization. We can use three optimization goals: *throughput*, *cost*, and *power*. Throughput optimization is important for both,

	Prompt Machine			Token Machine			Prompt-Token
	Type	Cost	Power	Type	Cost	Power	Interconnect Bandwidth
Splitwise-AA	DGX-A100	1×	1×	DGX-A100	1×	1×	1×
Splitwise-HH	DGX-H100	2.35×	1.75×	DGX-H100	2.5×	1.75×	2×
Splitwise-HHcap	DGX-H100	2.35×	1.75×	DGX-H100	2.5×	1.23×	2×
Splitwise-HA	DGX-H100	2.35×	1.75×	DGX-A100	1×	1×	1×

TABLE V: Evaluated Splitwise designs all normalized to DGX-A100

the cloud service provider (CSP) and the user. Cost optimization has different importance levels to the CSP and the user. For the CSP, a higher cost for the same throughput might be acceptable if there are gains in power and space requirements for the cluster. However, for the end-user, a higher cost at the same throughput is generally unacceptable. Finally, power optimization is attractive for a CSP, since it enables more GPUs to be deployed in the same datacenter [62], [63], but it may not be as important to the user. We only consider the provisioned power, and not the dynamic power utilization, in our study.

E. Practical Considerations

Accuracy impact. Splitwise does not impact accuracy since it uses lossless KV-cache transfer and does not add any randomization. It executes inference with the same parameters and state as on a single machine.

Scalability. Since LLM requests are much longer than typical ML requests [37], [38], they incur lower scheduling overhead for similar cluster sizes. However, the CLS may become a scalability bottleneck for large clusters. Insights from prior work on partitioned or replicated scheduling could help improve scalability [27], [61], [72] and are orthogonal to Splitwise.

Reliability and fault tolerance. If the prompt or the token machine fail, Splitwise simply restarts requests from scratch, similar to today's LLM serving systems [44], [51]. Alternatively, Splitwise could checkpoint the KV-cache generated after prompt computation into an in-memory database. To recover, Splitwise can use this cache to skip prompt recomputation, and start right away with the token phase. The KV-cache could also be checkpointed periodically during the token phase. Designing safe and efficient failure recovery is out of scope for our paper.

V. Methodology

A. Experimental setup

To evaluate our proposal on real hardware, we implement Splitwise's KV-cache transfer mechanism on top of vLLM [51]. Our implementation is open source [1]. We run this modified vLLM on two DGX-A100 and two DGX-H10 virtual machines (VMs) on Microsoft Azure with specifications from Table I. These are the VMs used to collect the characterization data in Section III. These machines are connected with InfiniBand and the DGX-H100s have double the bandwidth (*i.e.*, 400 Gbps).

Since vanilla vLLM only supports continuous batching with token preemption which can lead to much higher TBT, we implement state-of-the-art mixed continuous batching [81] as discussed earlier in Figure 2(c).

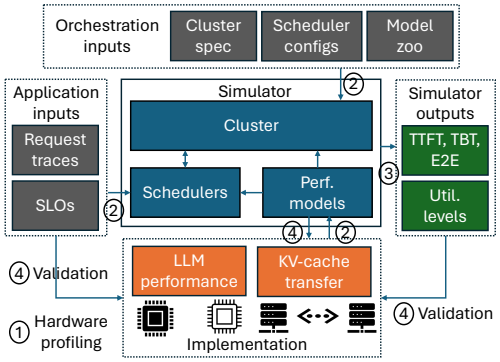


Fig. 13: Overview of the design of the Splitwise simulator.

Our implementation of the Splitwise technique assigns machines either a prompt role, or a token role. As the prompt machine generates the first token, it transfers the KV-cache to the token machine using the technique described in Section IV-C. We use MSCCL++ [11], an optimized GPU-driven communication library, to implement the naive and layer-wise KV cache transfers.

In our implementation, the prompt machine uses the zero-copy one-sided `put` primitive of MSCCL++ to send KV-cache data over InfiniBand as soon as it is ready, without requiring the token machine to issue any receive instructions. Once we have issued a `put` for all layers, the prompt machine signals a semaphore that the token machine waits on. The synchronization done with the help of semaphores uses the same InfiniBand connection used to send KV-cache data. When processing a batch of prompts, each request is assigned a different semaphore since it may be routed to different token machines. We ship the KV-caches block-by-block in vLLM. To minimize the number of transfers, we also consider the contiguity of KV blocks as long as they use the same semaphore.

B. Simulator setup

We build a simulator to explore cluster designs and evaluate Splitwise at scale. The simulator code is open source [20].

Figure 13 shows the design of our simulator. The simulator is event-driven and faithfully models the Splitwise machine pools, schedulers, machine-level memory and queues, and KV-cache transfer. We first profile the LLM on the target hardware with various input/output sizes (1). Based on the characterization profiles, we build a performance model. The simulator takes as input the request traces, SLOs, the performance model, and the configurations for cluster and scheduler (2). For our

	P50	P90	P99
TTFT	2×	3×	6×
TBT	1.25×	1.5×	5×
E2E	1.25×	1.5×	5×

TABLE VI: SLO expressed as slowdown compared to a request running on DGX-A100 under no contention.

evaluation, we use the prompt and token size distributions from the production traces in Section III. We tune the Poisson arrival rate to increase and decrease the load (requests per second) for cluster sizing. The simulator provides the achieved metrics per request (TTFT, TBT, E2E), and the machine utilization levels (3). We cross-validated the performance model with hardware experiments to ensure accuracy; we also validated the simulator end-to-end using production load with over 50K iterations to ensure fidelity (4).

Performance model. We build a piece-wise linear performance model using performance profiles at various batch sizes, input sizes, output sizes, in the required parallelism configuration on A100 and H100 machines from Section III. We validate that our performance model has high accuracy; it incurs a mean absolute percentage error (MAPE) of less than 3% when evaluated with a 80:20 train:test dataset split.

Communication model. In our evaluation, KV-cache transfers cause inter-machine communication, whereas tensor parallelism only causes intra-machine communication. We model inter-machine communication overheads by benchmarking our KV-cache transfer implementation over Infiniband in Section VI-A.

SLOs. To determine the maximum throughput that can be supported by a given cluster design, we use P50, P90, and P99 SLOs for TTFT, TBT, and E2E latency metrics. Table VI shows our SLO definition using DGX-A100 as a reference. We require all nine SLOs to be met. SLOs on TTFT are slightly looser, since it has a much smaller impact on the E2E latency.

Baselines. We compare our Splitwise designs against Baseline-A100 and Baseline-H100. The clusters in these baselines consist of just DGX-A100s and DGX-H100s, respectively. Both baselines use the same mixed continuous batching that Splitwise uses for mixed pool machines (described in Section IV-A).

VI. Evaluation

A. Experimental results

KV-cache transfer latency. We first measure the latency to transfer the KV-cache as the prompt size grows. Figure 14 shows the visible transfer latency on both A100 and H100 setups with the naive and optimized transfer design as discussed in Figure 11. Compared to the prompt computation time, the overhead is minimal (< 7%). The time for serialized transfers linearly increases with the prompt size since the size of the KV-cache also increases. The optimized per-layer transfer, on the other hand, hides much of the latency. For these transfers, we see a constant non-overlapped transfer time of around 8ms for the A100 and around 5ms for the H100 setup. The H100

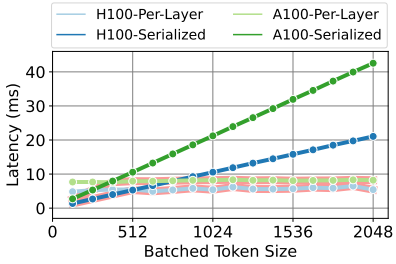


Fig. 14: Overhead of the KV-cache transfer as the prompt size increases on A100s and H100s.

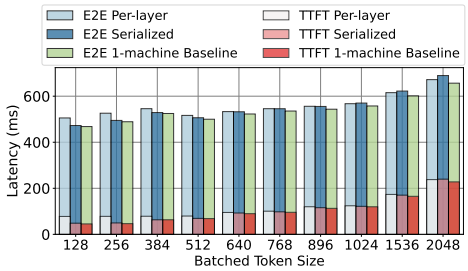


Fig. 15: Overhead of KV cache transfer on TTFT, E2E latency for coding trace for A100 and H100.

setup has double the bandwidth of the A100 setup (*i.e.*, 200 vs 400 Gbps), and the impact of this can be clearly seen with transfers in the H100 setup happening about twice as fast as those in the A100 setup.

As discussed in Section IV-C, for small prompt sizes (< 512 in H100), Splitwise uses the serialized KV-cache transfer and for larger prompts, it uses per-layer transfers.

End-to-end impact. Next, we run the coding trace on the 2-machine Splitwise setups without batching, and compare the observed latency metrics to a 1-machine baseline setup with no batching. Figure 15 shows our results. The latency impact of serially transferring the KV-cache grows up to 3% of the E2E with large prompts. However, Splitwise only incurs 0.8% of E2E. In a user-facing inference, the only visible impact of KV-cache transfer overhead is the latency for the second token. Splitwise adds a 16.5% latency to the second token, as compared to the 64% overhead from a serialized transfer. Overall, the transfer impact in Splitwise is hardly perceivable even in a user-facing inference.

B. Iso-power throughput-optimized clusters

Cluster provisioning. We provision clusters using the methodology described in Section IV-D. We target a specific workload (*e.g.*, conversation) at a peak load with the same power (*i.e.*, iso-power) for each cluster design. For the baseline, we use the power for 40 DGX-H100 machines as our target peak power. For the A100 baseline, we can fit 70 DGX-A100 machines under the same power budget. We denote these two designs as 40P/T and 70P/T respectively, since they both use mixed batching in all machines.

For Splitwise cluster designs under the coding trace, Splitwise-AA provisions 55 prompt machines and 15 for the token pool, denoted as (55P, 15P). Note that like Baseline-A100, Splitwise-AA also provisions 75% more machines than Baseline-H100. The legends in Figure 16 show the different provisioning choices under coding and conversation workloads. Request size distributions reflect in the machine pool sizing. For example, we provision more prompt machines under Splitwise-HH (35P, 5T) for the coding trace, while we provision more token machines (25P, 15T) for the conversation trace.

Latency and throughput. Figure 16 shows a deep dive into all the latency metrics at different input load for each cluster design with the same power (*i.e.*, iso-power). For the coding trace (Figure 16a), Splitwise-HH, Splitwise-HHcap, and Splitwise-AA all perform better than Baseline-H100. As the load increases, Baseline-H100 suffers from high TBT due to mixed batching with large prompt sizes. Although Splitwise-AA can support higher throughput, its TTFT is consistently higher than most designs. Splitwise-HA clearly bridges the gap by providing low TTFT and E2E at high throughput. The mixed machine pool in Splitwise becomes useful at higher loads to use all the available hardware without fragmentation. This benefit can be seen clearly in the P50 TBT chart for Splitwise-HA, where after 90 RPS, H100 machines jump into the mixed machine pool and help reduce TBT.

For the conversation trace (Figure 16b), Splitwise-HHcap clearly does better on all fronts, including latency. This is because its token generation phases typically run for much longer than in the coding trace, which is beneficial for the token machines.

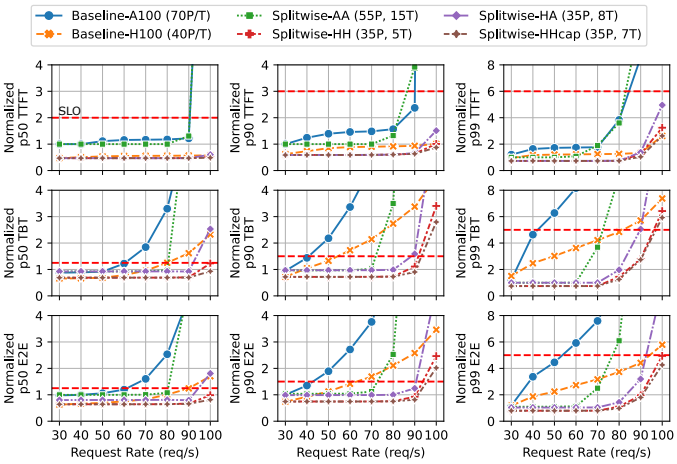
Impact on batched tokens. Figure 17 shows the cumulative distribution of time spent processing a varying number of batched active tokens in an iso-power throughput-optimized cluster. The distributions are collected by running the conversation trace at low (70 RPS) and high (130 RPS) loads.

At low load, all 40 Baseline-H100 machines spend 70% of the time running ≤15 tokens, and the rest running mixed batches with large prompts, which affects TBT and E2E. The 35 Splitwise-HH prompt machines are mostly idle, and when active, run much larger batches of tokens. The 15 Splitwise-HH token machines also do a better job at batching. Overall, Splitwise machines have better batching and latency at 70 RPS. At high load, since the mixed pool is utilized more, the batch sizes start looking similar across prompt and token machines.

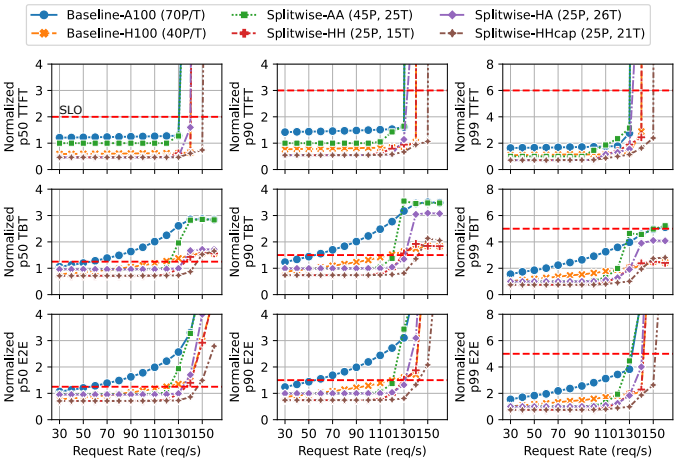
Summary plot. Figure 18a summarizes the results across all cluster metrics for iso-power throughput-optimized designs for the conversation trace. We use Baseline-A100 as the baseline. Compared to Baseline-A100, Splitwise-AA delivers 2.15× more throughput at the same power and cost. Splitwise-HA delivers 1.18× more throughput at 10% lower cost and the same power.

C. Other cluster optimizations

We have described *iso-power throughput-optimized clusters* in detail. For the rest of the cluster optimization evaluation, we only discuss the summary plots.



(a) Coding trace.



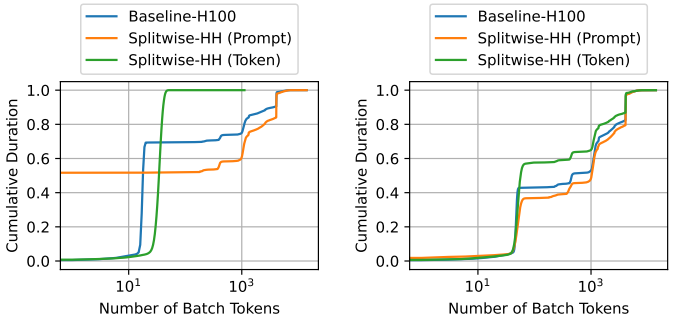
(b) Conversation trace.

Fig. 16: Latency metrics across input loads for iso-power throughput optimized clusters. Dashed red lines indicate SLO.

Iso-cost throughput-optimized. Figure 18b shows the summary plot for iso-cost clusters, with their space, throughput, and power requirements. We find that Splitwise-AA provides the best throughput for the same cost, namely 1.4× more throughput than Baseline-H100, running at 25% more power, and 2× the space. This is an interesting operational point for most customers who may not care about power and space, instead preferring the 40% higher throughput using older, more easily available GPUs. In contrast, the preferable choice for the CSP is less clear.

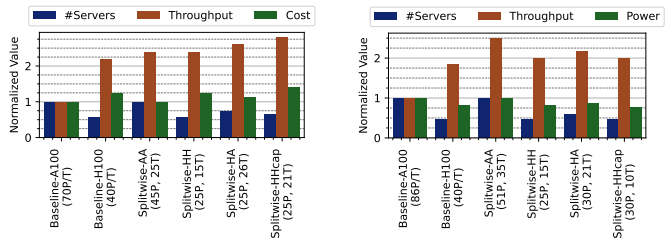
Iso-throughput power-optimized. Figure 19a shows cluster designs that yield same throughput at the least power. Splitwise-HHcap can achieve the same throughput as Baseline-H100 at 25% lower power at the same cost and space. This can be a clear win for the CSPs.

Iso-throughput cost-optimized. Figure 19b shows the cost-optimized versions of the iso-throughput design. Note that there are no changes to any of the homogeneous designs between Figures 19a and 19b. This is because the prompt and token machines have the same cost and power. However, Splitwise-



(a) Low load (70 RPS). (b) High load (130 RPS).

Fig. 17: Cumulative distribution of time spent at various batched token sizes for iso-power throughput-optimized design.



(a) Iso-power. (b) Iso-cost.

Fig. 18: Summary of throughput-optimized cluster designs.

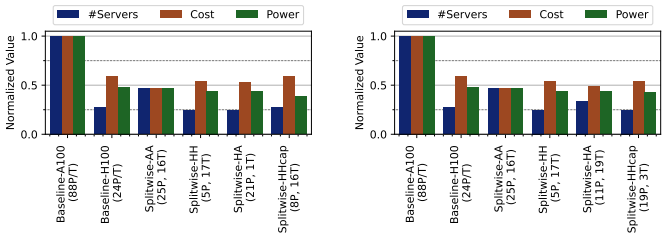
HA and Splitwise-HHcap arrive at slightly different results with the cost and power optimizations. Figure 19b shows that with Splitwise-AA, customers can achieve the same throughput as Baseline-H100 at 25% lower cost.

D. Impact of workload changes

So far, we have tested a trace and a model on clusters optimized for a specific workload pattern and model. To test the Splitwise' robustness, we now run conversation trace on a cluster meant for coding service, and Llama-70B on a cluster meant for BLOOM-176B. Figure 20 shows these results for iso-power throughput-optimized clusters.

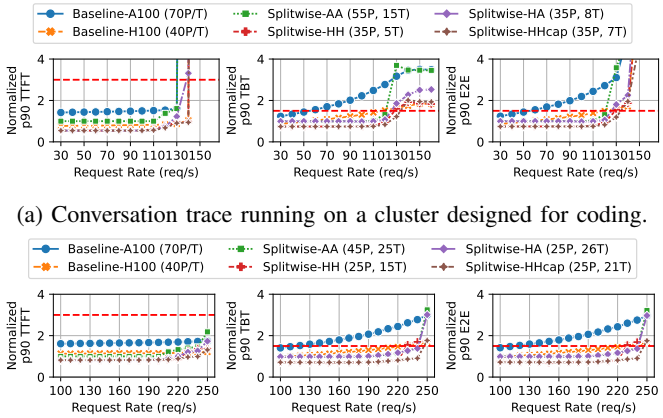
Changing workload trace. Compared to Figure 16b, we find that in Figure 20a, the Baseline clusters are similarly sized and incur no throughput or latency impact. Splitwise-AA and Splitwise-HH with the mixed pool morph well to meet the requirements of the new workload, and they see no throughput or latency impact. Since Splitwise-HA and Splitwise-HHcap have different types of machines in the prompt and token pools, they experience a throughput setback of 7% from the respective cluster optimized designs for conversation trace. Note that all the Splitwise designs still perform much better than any of the Baseline designs.

Changing model. Figure 20b shows that Llama-70B can support much higher throughput in the same cluster design than BLOOM-176B, given its fewer parameters (Table III). All the Splitwise designs out-perform both the Baseline designs at higher load. Furthermore, Splitwise-HH and Splitwise-HHcap consistently achieve the best latency, even as the load increases.



(a) Power-optimized. (b) Cost-optimized.

Fig. 19: Summary of iso-throughput cluster designs.



(a) Conversation trace running on a cluster designed for coding.

(b) Llama-70B, on a cluster designed for BLOOM-176B.

Fig. 20: Latency impact of running a workload on a cluster designed for another workload. Dashed red lines indicate SLO.

Summary. Based on these two experiments, we conclude that Splitwise can morph according to the requirements of the workload using its smart scheduling, and it is robust to changes in the LLMs, request load, and token distributions.

E. Cluster design for batch job

We design various clusters with Splitwise under strict latency SLOs, even when we are optimizing for throughput. This is unnecessary for batch jobs, which can be stressed to high load for a high token generation throughput. We find that upon stressing our iso-power throughput-optimized clusters, Baseline-A100 and Splitwise-AA have the best throughput per cost at 0.89 RPS/$. At high load, Splitwise devolves into the iso-count Baseline, since it starts mixed batching with all the machines in the mixed pool. The same holds true for Splitwise-HH and Baseline-H100, which achieve 0.75 RPS/$.

VII. Discussion

Extensibility to new models. Despite the plethora of model sizes from 2B parameters [47], [84] to 176B parameters [69] or more [18], all modern transformer-based generative LLMs have the distinct prompt processing and token generation phases. Similarly, even modifications and flavors like Mixture-of-Experts (MoEs) have these phases. Since Splitwise is built solely by exploiting these phases, it is applicable to all of the

current and upcoming LLMs, as long as the auto-regressive nature of the workload requires these two phases. Note that as shown in Section VI-D, clusters provisioned with Splitwise for one model can also efficiently serve other models.

Alternative compute hardware. In this work, we use NVIDIA H100 and A100 GPUs since they are commonly used for LLM inference in datacenters today [17]. Smaller datacenter GPUs like NVIDIA T4 lack enough memory to run modern LLMs efficiently. In general, our methodology is applicable to any hardware (including CPUs, FPGAs, ASICs [33]) that aligns with the computational requirements of prompt and token phases. Our characterization suggests that prompt phases need high compute capability and memory bandwidth with low memory capacity, whereas token phases need moderate compute capability with high memory capacity and bandwidth. Thus, GPUs like AMD MI-250 [2] and CPUs like Intel Sapphire-Rapids (with HBM) [9] could be effective token machines. Since we do not have access to such hardware and/or optimized LLM implementations, we leave this to future work.

Interconnect between prompt and token machines. In this work, we assume Infiniband connection between the prompt and token machines in all the designs (albeit, lower bandwidth when A100s were involved). Although this is common for all homogenous machines, Splitwise-HA is not be readily available with an Infiniband connection between H100s and A100s, even though technically feasible. The alternative could be HPC clouds, with Infiniband connections through the CPU [3], or Ethernet, using RoCE [58]. Given our optimized KV-cache transfer that helps reduce critical latency, an interconnect with $10\times$ lower bandwidth would likely still be beneficial. To further reduce our bandwidth utilization, we could also compress the KV-cache before transferring it across the network [55].

Heterogeneous prompt/token machines. Although Splitwise is robust to varied models and input traces, we recognize that fragmenting a data center with different types of GPUs (*e.g.*, Splitwise-HA) may bring its own challenges for the CSP.

Conversation back and forth. Chat APIs for LLMs today require the user to send the complete context of the conversation so far [18]. However, in the future, services may have enough GPU capacity to cache the context and avoid recomputation. This could sway the memory utilization pattern of the prompt phase from our characterization. Furthermore, it may require transferring the KV-cache back to a prompt machine to be ready for the next conversation request.

VIII. Related Work

Heterogeneous scheduling and dataflow systems. Prior work has studied heterogeneous scheduling for a variety of interactive services [65], [68], [83]. These works exploit hardware heterogeneity to strike a balance between different objectives such as cost, energy, and performance. However, they run the entire workload on the same machine. Research on heterogeneous multiprocessor CPU scheduling attempts to match workload heterogeneity to hardware heterogeneity [29], [40], [41], [50], [76], [80]. These works use profiling or online monitoring with metrics like request length or hardware performance counters to identify workload phases and allocate them appropriately on heterogeneous processors. However, they do not consider the complexities with batching. Distributed dataflow systems orchestrate large-scale computational graphs and aim to provide general-purpose programmability [34], [46], [75], [82]. LLM inference under Splitwise can be viewed as a static computational graph with two stages, so it could be implemented using distributed frameworks that provide efficient GPU abstractions [59]. Splitwise differs from these works since it uses a specialized two-phase design for generative LLM inference and leverages phase-aware resource management with efficient batching.

Model serving systems. LLM inference serving is a rapidly developing field, with several recent works optimizing batching [23], [25], [51], [53], [81], scheduling [22], [42], [51], [66], [73], [79], and memory usage [32], [35], [51], [74]. Prior work has also proposed using CPUs and lower compute capability devices for LLM serving [8], [12]. These approaches use the same machine for both prompt and token phase. With Splitwise, they could improve throughput and latency by splitting phases.

Prior work on video and ML serving focuses on scheduling model chains with data dependencies under latency constraints [24], [31], [43], [49], [68]. Such schedulers rely on model profiling to make efficient allocation decisions and manage requests across machines. Recommendation system inference exhibits compute/memory heterogeneity both within and across models. Prior work exploits such heterogeneity to selectively schedule requests between CPUs and accelerators [38], [52], colocate models with complementary memory usage [30], and partition compute/memory on heterogeneous hardware resources [45], [48]. Similarly, Splitwise exploits the heterogeneity within LLM inference requests. However, it uses different optimizations due to the differences in LLM workload characteristics and requirements.

IX. Conclusion

We extensively characterized the prompt computation and token generation phases of LLM inference to draw out differences in their system utilization patterns. Based on our insights, we designed Splitwise to separate these phases onto different machines and enable phase-specific resource management. Using Splitwise, we explored cluster designs optimized for throughput, cost, and power, and showed that they perform well even as workloads change. Splitwise clusters under performance SLOs achieve $1.76\times$ better throughput with 15% lower power at the same cost, or $2.35\times$ better throughput with same the cost and power than existing designs.

Acknowledgements

We thank the reviewers for their helpful feedback. We thank Chetan Bansal, Srikant Bhardwaj, Suriya Kalivardhan, Ankur Mallick, Deepak Narayanan, and Amar Phanishayee for insightful discussions. Pratyush Patel was partially supported by NSF CNS-2104548 and a research grant from VMware.

REFERENCES

[1] Add Splitwise Implementation to vLLM. GitHub. [Online]. Available: https://github.com/vllm-project/vllm/pull/2809

[2] AMD Instinct™ MI250 Accelerator. [Online]. Available: https://www.amd.com/en/products/server-accelerators/instinct-mi250

[3] Azure InfiniBand HPC VMs. [Online]. Available: https://learn.microsoft.com/en-us/azure/virtual-machines/overview-hb-hc

[4] Azure Public Dataset: Azure LLM Inference Trace 2023. GitHub. [Online]. Available: https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2023.md

[5] CoreWeave - Specialized Cloud Provider. [Online]. Available: https://www.coreweave.com

[6] Google Assistant with Bard. [Online]. Available: https://blog.google/products/assistant/google-assistant-bard-generative-ai/

[7] HPC Interconnect on CoreWeave Cloud. [Online]. Available: https://docs.coreweave.com/networking/hpc-interconnect

[8] Intel BigDL-LLM. [Online]. Available: https://github.com/intel-analytics/BigDL

[9] Intel Sapphire Rapids with HBM. [Online]. Available: https://www.anandtech.com/show/17422/intel-showcases-sapphire-rapids-plus-hbm-xeon-performance-isc-2022

[10] Microsoft Azure ND A100 v4-series . [Online]. Available: https://learn.microsoft.com/en-us/azure/virtual-machines/nda100-v4-series

[11] MSCCL++: A GPU-driven communication stack for scalable AI applications. [Online]. Available: https://github.com/microsoft/mscclpp

[12] Numenta Inference on CPUs. [Online]. Available: https://www.servethehome.com/numenta-has-the-secret-to-ai-inference-on-cpus-like-the-intel-xeon-max/

[13] NVIDIA Accelerated InfiniBand Solutions. [Online]. Available: https://www.nvidia.com/en-us/networking/products/infiniband/

[14] NVIDIA Chip Shortage. [Online]. Available: https://www.wired.com/story/nvidia-chip-shortages-leave-ai-startups-scrambling-for-computing-power/

[15] NVIDIA DGX A100: Universal System for AI Infrastructure. [Online]. Available: https://resources.nvidia.com/en-us-dgx-systems/dgx-ai

[16] NVIDIA DGX H100. [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-h100/

[17] NVIDIA Hopper GPUs Expand Reach as Demand for AI Grows. [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-hopper-gpus-expand-reach-as-demand-for-ai-grows

[18] OpenAI ChatGPT APIs. [Online]. Available: https://openai.com/blog/introducing-chatgpt-and-whisper-apis

[19] Power Availability Stymies Datacenter Growth. [Online]. Available: https://www.networkworld.com/article/972483/power-availability-stymies-data-center-growth.

[20] SplitwiseSim: LLM Serving Cluster Simulator. GitHub. [Online]. Available: https://github.com/Mutinifni/splitwise-sim

[21] The New Bing. [Online]. Available: https://www.microsoft.com/en-us/edge/features/the-new-bing?form=MT00D8

[22] TurboMind Inference Server. [Online]. Available: https://github.com/InternLM/lmdeploy

[23] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills," *arXiv preprint arXiv:2308.16369*, 2023.

[24] H. Albahar, S. Dongare, Y. Du, N. Zhao, A. K. Paul, and A. R. Butt, "SchedTune: A heterogeneity-aware GPU Scheduler for Deep Learning," in *CCGrid*, 2022.

[25] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, and Y. He, "DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale," in *SC*, 2022.

[26] L. A. Barroso, U. Hölzle, and P. Ranganathan, "The Datacenter as a Computer: Designing Warehouse-Scale Machines," *Synthesis Lectures on Computer Architecture*, 2018.

[27] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing," in *OSDI*, 2014.

[28] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," *arXiv preprint arXiv:2005.14165*, 2020.

[29] J. Chen and L. K. John, "Efficient Program Scheduling for Heterogeneous Multi-core Processors," in *DAC*, 2009.

[30] Y. Choi, J. Kim, and M. Rhu, "Hera: A Heterogeneity-Aware Multi-Tenant Inference Server for Personalized Recommendations," *arXiv preprint arXiv:2302.11750*, 2023.

[31] D. Crankshaw, G.-E. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov, "InferLine: Latency-aware Provisioning and Scaling for Prediction Serving Pipelines," in *SoCC*, 2020.

[32] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and Memory-efficient Exact Attention with IO-Awareness," in *NeurIPS*, 2022.

[33] David Patterson. Domain Specific Architectures for Deep Neural Networks: Three Generations of Tensor Processing Units (TPUs). Allen School Distinguished Lecture. [Online]. Available: https://www.youtube.com/watch?v=VCScWh966u4

[34] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, 2008.

[35] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale," *arXiv preprint arXiv:2208.07339*, 2022.

[36] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *NAACL*, 2019.

[37] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up," in *OSDI*, 2020.

[38] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "DeepRecSys: A System for Optimizing End-to-end At-scale Neural Recommendation Inference," in *ISCA*, 2020.

[39] V. Gupta, M. Harchol Balter, K. Sigman, and W. Whitt, "Analysis of Join-the-Shortest-Queue Routing for Web Server Farms," *Performance Evaluation*, 2007.

[40] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services," *ACM SIGPLAN Notices*, 2015.

[41] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, "Exploiting Heterogeneity for Tail Latency and Energy Efficiency," in *MICRO*, 2017.

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

[43] Y. Hu, R. Ghosh, and R. Govindan, "Scrooge: A Cost-effective Deep Learning Inference System," in *SoCC*, 2021.

[44] Huggingface. Text Generation Inference (TGI). [Online]. Available: https://github.com/huggingface/text-generation-inference

[45] R. Hwang, T. Kim, Y. Kwon, and M. Rhu, "Centaur: A Chiplet-based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations," in *ISCA*, 2020.

[46] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *EuroSys*, 2007.

[47] M. Javaheripi and S. Bubeck, "Phi-2: The Surprising Power of Small Language Models," *Microsoft Research Blog*, 2023.

[48] W. Jiang, Z. He, S. Zhang, K. Zeng, L. Feng, J. Zhang, T. Liu, Y. Li, J. Zhou, C. Zhang *et al.*, "FleetRec: Large-scale Recommendation Inference on Hybrid GPU-FPGA Clusters," in *KDD*, 2021.

[49] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks," in *EuroSys*, 2019.

[50] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction," in *MICRO*, 2003.

[51] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *SOSP*, 2023.

[52] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning," in *MICRO*, 2019.

[53] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, and I. Stoica, "AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving," in *OSDI*, 2023.

[54] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A Robustly Optimized BERT Pretraining Approach," *arXiv preprint arXiv:1907.11692*, 2019.
[55] Z. Liu, J. Wang, T. Dao, T. Zhou, B. Yuan, Z. Song, A. Shrivastava, C. Zhang, Y. Tian, C. Re, and B. Chen, "Deja Vu: Contextual Sparsity for Efficient LLMs at Inference Time," in *ICML*, 2023.
[56] Meta. Introducing the AI Research SuperCluster — Meta's Cutting-Edge AI Supercomputer for AI Research. [Online]. Available: https://ai.facebook.com/blog/ai-rsc/
[57] "Azure OpenAI Service," Microsoft Azure, 2022. [Online]. Available: https://azure.microsoft.com/en-us/products/ai-services/openai-service
[58] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting Network Support for RDMA," *arXiv preprint arXiv:1806.08159*, 2018.
[59] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A Distributed Framework for Emerging AI Applications," in *OSDI*, 2018.
[60] OpenAI. Scaling Kubernetes to 7,500 Nodes. [Online]. Available: https://openai.com/research/scaling-kubernetes-to-7500-nodes
[61] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, Low Latency Scheduling," in *SOSP*, 2013.
[62] P. Patel, E. Choukse, C. Zhang, Í. Goiri, B. Warrier, N. Mahalingam, and R. Bianchini, "POLCA: Power Oversubscription in LLM Cloud Providers," *arXiv preprint arXiv:2308.12908*, 2023.
[63] P. Patel, E. Choukse, C. Zhang, Í. Goiri, B. Warrier, N. Mahalingam, and R. Bianchini, "Characterizing Power Management Opportunities for LLMs in the Cloud," in *ASPLOS*, 2024.
[64] P. Patel, Z. Gong, S. Rizvi, E. Choukse, P. Misra, T. Anderson, and A. Sriraman, "Towards Improved Power Management in Cloud GPUs," in *IEEE CAL*, 2023.
[65] P. Patel, K. Lim, K. Jhunjhunwalla, A. Martinez, M. Demoulin, J. Nelson, I. Zhang, and T. Anderson, "Hybrid Computing for Interactive Datacenter Applications," *arXiv preprint arXiv:2304.04488*, 2023.
[66] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently Scaling Transformer Inference," in *MLSys*, 2023.
[67] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language Models are Unsupervised Multitask Learners," *OpenAI blog*, 2019.
[68] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated Model-less Inference Serving," in *USENIX ATC*, 2021.
[69] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, J. Tow, A. M. Rush, S. Biderman, A. Webson, P. S. Ammanamanchi, T. Wang, B. Sagot, N. Muennighoff, A. V. del Moral, O. Ruwase, R. Bawden, S. Bekman, A. McMillan-Major, I. Beltagy, H. Nguyen, L. Saulnier, S. Tan, P. O. Suarez, V. Sanh, H. Laurençon, Y. Jernite, J. Launay, M. Mitchell, and C. Raffel, "BLOOM: A 176B-Parameter Open-access Multilingual Language Model," *arXiv preprint arXiv:2211.05100*, 2022.
[70] P. Schmid. Fine-tune FLAN-T5 XL/XXL using DeepSpeed & Hugging Face Transformers. [Online]. Available: https://www.philschmid.de/fine-tune-flan-t5-deepspeed
[71] P. Schmid, O. Sanseviero, P. Cuenca, and L. Tunstall. Llama 2 is here - Get it on Hugging Face. [Online]. Available: https://huggingface.co/blog/llama2
[72] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *EuroSys*, 2013.
[73] Y. Sheng, S. Cao, D. Li, B. Zhu, Z. Li, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Fairness in Serving Large Language Models," *arXiv preprint arXiv:2401.00588*, 2023.
[74] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU," in *ICML*, 2023.
[75] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *MSST*, 2010.
[76] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE)," *ACM SIGARCH Computer Architecture News*, 2012.
[77] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is All You Need," in *NeurIPS*, 2017.
[78] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. v. Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art Natural Language Processing," in *EMNLP*, 2020.
[79] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, "Fast Distributed Inference Serving for Large Language Models," *arXiv preprint arXiv:2305.05920*, 2023.
[80] H. Yang, Q. Chen, M. Riaz, Z. Luan, L. Tang, and J. Mars, "PowerChief: Intelligent Power Allocation for Multi-stage Applications to Improve Responsiveness on Power Constrained CMP," in *ISCA*, 2017.
[81] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A Distributed Serving System for Transformer-Based Generative Models," in *OSDI*, 2022.
[82] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *NSDI*, 2012.
[83] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving," in *USENIX ATC*, 2019.
[84] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "OPT: Open Pre-trained Transformer Language Models," *arXiv preprint arXiv:2205.01068*, 2022.
[85] W. Zhu, "Analysis of JSQ Policy on Soft Real-time Scheduling in Cluster," in *HPCAsia*, 2000.

APPENDIX

A. Abstract

We open source critical components needed to evaluate Splitwise; these could be repurposed to also evaluate future LLM inference serving systems. Our artifact includes:

- Production traces from two LLM inference services at Microsoft Azure.
- A prototype implementation of Splitwise's KV-cache transfer mechanism in vLLM [51].
- SplitwiseSim, a discrete event simulator to evaluate model serving in LLM inference clusters.

Artifact functionality was only tested for the traces and SplitwiseSim due to limited hardware availability.

B. Artifact check-list (meta-information)

- **Data set:** Production traces available as a part of the artifact.
- **Run-time environment:** Linux / Ubuntu.
- **Hardware:** Two machines connected over GPU Infiniband for the vLLM prototype (*e.g.* NVIDIA DGX-A100, NVIDIA DGX-H100). x86-64 CPU machine for SplitwiseSim.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT.
- **Data licenses (if publicly available)?:** CC-BY.
- **Archived (provide DOI)?:** 10.5281/zenodo.11003049.

C. Description

How to access. The entire artifact is available as an archive on Zenodo: https://doi.org/10.5281/zenodo.11003049. Individual components are also available online as follows:

- The production traces can be downloaded from the Azure Public Dataset GitHub repository [4].
- The KV-cache transfer prototype can be downloaded from the vLLM GitHub repository, currently available as a pull request [1].

- SplitwiseSim, and the associated experiment and plotting scripts, can be downloaded from a separate GitHub repository [20].

Hardware dependencies. The KV-cache transfer prototype requires two GPU machines connected over Infiniband, such as NVIDIA DGX-A100s or NVIDIA DGX-H100s. SplitwiseSim requires a standard x86-64 CPU machine; multiple machines may be used to parallelize simulation runs.

Software dependencies. The KV-cache transfer prototype is built on top of vLLM [51] and MSCCL++ [11]. SplitwiseSim depends on a small set of publicly available Python packages, which can be installed via the included `requirements.txt`.

Data sets. Coding and conversation traces from Microsoft Azure are available online as a part of the artifact release [4].

D. Installation and Experiment Workflow

Please refer to the README files within the artifact for installation and usage instructions.

E. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html