Adit Kolli (aditk), Aryan Joshi (aryanj), Keshav Singh (keshsing), Anup Bagali (abagali)

Summary of “Splitwise: Efficient Generative LLM Inference Using Phase Splitting”

Problem and motivation

Current scheduling policies attempt to optimize for latencies like Time To First Token (TTFT) and Time Between Tokens (TBT) in addition to overall End-To-End (E2E) latency. However, even with various scheduling optimizations, running both the prompt computation and token generation phases of an LLM request on the same machine often leads to inconsistent E2E latencies due to the arbitrary batching of those phases. This leads to service providers provisioning extra GPUs to meet tight inference service level objectives. The prompt computation and token generation phases differ significantly as the prompt computation phase is compute bound while the token generate phase is memory bound. Since these two phases differ significantly, using different machines for different phases could reduce the overall cost while not sacrificing throughput.

Related works

Prior works have studied heterogeneous scheduling for different services and strike a balance between cost, energy, and performance but they still run the entire workload on the same machine [1]. Additionally, other works have investigated the possibility of serving LLMs using CPUs or other relatively low-compute devices [2]. Also, prior work does not account for batching complexities of heterogeneous scheduling.

Solution Overview

Splitwise separates the prompt and token generation phases onto different machines to leverage phase-aware resource management with efficient batching. The key idea is to use this separation to make cost-effective decisions on machine assignment and improve batching.

Prompt machines are responsible for generating the first token for an input query, by processing all the input prompt tokens and generating the KV-cache. The prompt machine sends over the KV-cache to the token machine, which continues the token generation until the response is complete.

Splitwise clusters are organized into prompt pool, token generation pool, and mixed pool. Splitwise has a cluster-level scheduler (CLS) and a machine-level scheduler (MLS). One role of the CLS is to maintain the different pools of machines. The initial assignment of machines to these pools is dependent on expected load and input/output token distributions. Machines can then move between the pools based on loads. The other role of the CLS is request routing. For this, it uses “Join the shortest queue scheduling”.

The machine-level scheduler is responsible for tracking GPU memory utilization and maintaining the pending queue, deciding batch sizes.

Since the prompt and token phases happen on different machines, the KV cache needs to be transferred from the prompt to the token machine. The naive method of transferring is to send the KV cache after the prompt phase has been completed. To optimize the for large KV-caches we can transfer the KV cache in chunks as it is computed layer by layer, which means the transfer will happen in parallel while the next layers are being computed.

Results - Under performance splitwise clusters SLOs achieve 1.76× better throughput with 15% lower power at the same cost, or 2.35× better throughput with the same cost and power than existing designs.

Limitations

Splitwise’s implementation assumes an Infiniband connection between the prompt and token machines. However, although technically feasible, the current implementation is incompatible with Infiniband connections between Nvidia H100 and A100 GPUs. Such transfers would require connections via CPU or Ethernet. The authors suggest that, given Splitwises’s optimized KV-cache transfer method, these lower-bandwidth transfer options would not significantly hinder Splitwise’s benefit. Additionally, they also suggest compressing the KV-cache for a faster transfer.

Current implementations of LLM-based chat APIs require users to continuously send the full conversation context back and forth. However, as services increase their GPU capacity, the entire conversation context could be saved to avoid recomputation. This would alter the authors’ characterization of the memory utilization pattern of the prompt phase.

Splitwise also has similar limitations to Lluminix of fault tolerance and fair scheduling.

Future research directions

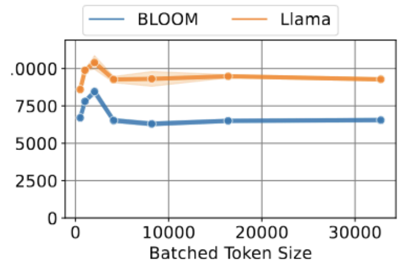
Splitwise characterizes LLM inference in two stages: compute-bound prompt computation and memory-bound token generation. As such Splitwise can run on hardware that aligns with the compute/memory requirement of each phase. However, this method was not implemented on several GPU/heterogeneous computing devices and was left for future work.

Summary of class discussion

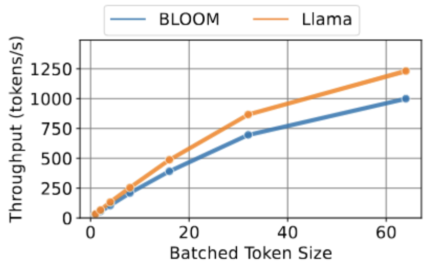
Q: Why is the H100 cheaper in the prompt phase even though each H100 is more expensive?

A : Because many less H100s are needed than A100s for the same performance outcome since the prompt phase is much more compute intensive.

Q: On the graph in slide 8 why is there a sudden drop at 2048 for the prompt phase?
A : The number 2048 here is GPU specific. After batch size of 2048 the GPU can't efficiently provision and handle data for requests.
Followup: How come it flatlines eventually instead of going down. Does the paper mention why?
A: The paper doesn't seem to mention why the flatline occurs.



(a) Prompt phase.



(b) Token generation phase.

Q: Does the Cluster Level Scheduler split requests across instances.
A: This is unrelated to the approach in the paper; the paper only focuses on the benefits of splitting the prompt and token phase across instances.

Q: Does the KV cache transfer affect power usage?
A: Insu says that power is static in network machines so it doesn't make a difference or at least not a significant difference.

Q: How do you know when the KV cache is ready to be used if the cache is sent layer by layer?
A: The prompt machine signals a semaphore that the token machine waits on.

Q: How effective would splitting phases be if you use it with other approaches to reduce memory in the token phase such as sliding window attention?
A: It should be complementary as the different methods should not interfere with each other.

[1] 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.
[2] J. Chen and L. K. John, "Efficient Program Scheduling for Heterogeneous Multi-core Processors," in DAC, 2009.

Summary of “Llumnix: Dynamic Scheduling for Large Language Model Serving”

Problem and motivation

LLM requests consist of a wide variety of tasks, such as summarization, generation, or rewording, each with varying service level objectives (SLOs), and input and output lengths. Additionally, the frequency of requests and the amount of resources each request requires are unknown and can vary greatly, especially with the increase in maximum sequence lengths LLMs output. Static scheduling mechanisms are therefore ineffective and cause inefficiencies such as load imbalances, violation of SLOs, and external fragmentation, all of which increase latency and cause unwanted preemption.

Related works

Much of the prior work with regard to inference optimization is centered around improvements made at the instance level, such as optimizations to memory usage (vLLM), better utilization of hardware (FlashAttention), and instance-level scheduling (Orca), which can be integrated into the global dynamic scheduling that Lluminix proposes.

There have also been numerous publications concerning the scheduling of inference requests to DNNs, but traditional DNN requests only need one-time inference whereas LLM inference is autoregressive and relies on the previously generated output recursively. The most similar of these DNN scheduling works is INFaas, which is a technique that selects an engine to run a given inference request based on the engine’s specific hardware and resource footprint. INFaas is not LLM specific, and is meant for general-purpose DNN inference, and is used throughout the paper as a comparison for the results produced by Lluminix.

Gandiva is a paper that proposes migration for DNN inference, however, it differs from Lluminix in how it does it. Gandiva must migrate the entire model weights while execution is paused since they can change during execution, whereas this is not the case for Lluminix.

Solution Overview

Lluminix uses dynamic scheduling to mitigate inefficiencies static schedulers may face due to variability in LLM requests such as load imbalances, violation of SLOs, and external fragmentation. The approach is similar to how the OS manages threads across CPU cores.

External fragmentation occurs when there is enough memory across all instances to serve a request, but no single instance has sufficient memory to serve a request, which results in unnecessary preempts. To resolve this, Llumnix adopts a dynamic approach, where requests from overloaded machines may be rescheduled at other machines. This mitigates external fragmentation, as it shuffles around requests until we can sufficiently free one of the instances to serve a new request. To monitor the resources used by each request and the amount of resources free at each instance, Llumnix uses a metric called Virtual Usage. Virtual Usage is a function of the memory a request uses at some point in time, and the priority of the request (i.e. how urgent the request is). Once an instance has excessive virtual usage, some of its requests are transferred to other instances with lower virtual usage.

Llumnix uses virtual usage rather than physical usage to ensure high-priority requests are run on machines that have smaller load, since processing too many requests at a time may lead to additional latency. For high-priority requests, some additional padding is added to the physical usage, which leads to lower-priority requests on the same instance to be rescheduled even when the instance's maximum capacity has not been reached. We see that this approach prevents SLOs from being violated, as we ensure that requests that are latency-sensitive (e.g. LLM chatbot requests) incur the least amount of latency possible.

To reduce the overhead incurred from rescheduling, Llumnix employs Live Migration. We observe that for a request, the KV-cache is only appended, and never overwritten. Thus, we may transfer the KV-cache of a request while we continue computing the request. Since the transfer of memory for each cache entry is faster than the time it takes to compute it, the transfer eventually catches up to the newest token generated for a request, at which point the request is ready to continue at the other instance. This means that the overhead due to migration is only the time it takes to transfer the final block of memory for a request.

By monitoring virtual usage at each instance, the global scheduler effectively balances loads. The global scheduler monitors the usage at each instance, and when it observes an unbalanced load distribution across machines, it initiates migration from higher to lower load machines. Further, virtual usage allows seamless autoscaling, as the global scheduler can detect when the total usage is too high for the number of machines and additional instances are needed. Additionally, when there are more active instances than needed, we may simply mark an instance as inactive by assigning it a dummy request with infinite virtual usage.

Limitations

Llumnix faces some of the limitations that Splitwise does. One limitation is that failure of a particular instance would cause Llumnix to have to recompute the entire computation for all active requests, which causes us to incur additional latency overhead. As proposed in lecture, this may be addressed by checkpointing KV-cache results by storing it in a database, so that we don’t have to restart requests on a failed instance from scratch.

Another Limitation is that Llumnix doesn’t seem to address other desirable scheduling objectives, such as fairness. For example, the Llumnix paper doesn’t seem to directly address certain issues that may occur such as starvation of requests. Future work can be done to address fairness in Llumnix while adding minimal overhead.

Future research directions

One future direction that the authors mentioned is extending Llumnix to run for multiple models. Currently, Llumnix works for a single LLM, but other works such as INFaaS support scheduling requests across multiple models based on performance and accuracy constraints. The authors of Llumnix also discuss the potential to further look into the interplay between Local and Global Scheduling techniques to achieve a more optimal scheduler (i.e where each local scheduler uses its own techniques such as preemptive and fair scheduling).

Summary of class discussion

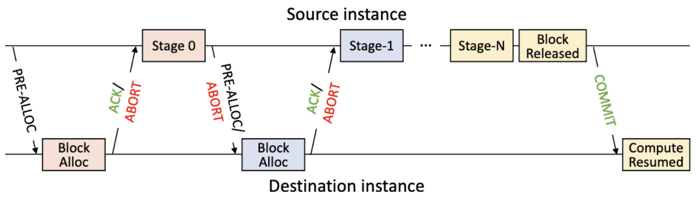
Q. What does Llumnix use for each of the instances?
A. It uses vLLM engines.

Q. Would it be possible to combine aspects of Llumnix and Splitwise, as Splitwise doesn't seem to address External Fragmentation like Llumnix does?
A. Yes it should be possible to incorporate some of the Llumnix optimizations to reduce fragmentation in Splitwise.

Q. Is it ever the case that regular priority requests are forward indefinitely due to too many high priority requests?
A. This doesn't seem to be the case. Overhead due to rescheduling is almost zero for regular priority requests due to rescheduling.

Q. The paper mentions that Goo is used to transfer KV-cache, which transfers memory from GPU to CPU and then back to GPU. Is there any data for how much overhead this transfer causes?
A. The paper shows that there is minimal overhead from this, substantially less than recomputing. This is due to the live migration used by Llumnix when transferring KV-cache.

Q: Is Llumnix able to tolerate failures that occur during the migration portion?
A. Yes, it does so by performing handshake during migration, as shown in the below diagram.


Source instance
Stage 0
Stage-1
Stage-N
Block Released
PRE-ALLOC
ACK/ ABORT
PRE-ALLOC/ ABORT
ACK/ ABORT
COMMIT
Block Alloc
Block Alloc
Compute Resumed
Destination instance