CSE-234: Data Systems for Machine Learning, Winter 2025

19. LLM-3: Continuous Batching, Paged Attention, Prefill-Decode Disaggregation

Lecturer: Hao Zhang
Scribe: Feng Yao, Aarti Lalwani, Yijiang Li, Junfei Liu, Inan Xu, Weifan Yu, Nandakrishna Kanakapuram Srinivasadeshikachar, Xunyi Jiang, Srikrishna Dantu, Abhinandan Padhi, Sathvik Balakrishna, Veeramakali Vignesh Manivannan

1 Intro: LLM Serving and Inference Challenges

This lecture will cover timely content—LLM serving and inference—which is precisely what leading Silicon Valley companies, such as OpenAI and Google DeepMind, are focusing on. This section will be covering (1) how the LLMs perform the inference process and (2) what the bottlenecks are in LLM serving and inference.

1.1 Recap: LLM Inference Process

LLMs are basically language models, which operate as predicting what word comes next. As shown in Figure 1, LLMs learn to predict the next token by maximizing the probability of the entire sentence, which is a product of all the conditional probability of tokens appearing sequentially.

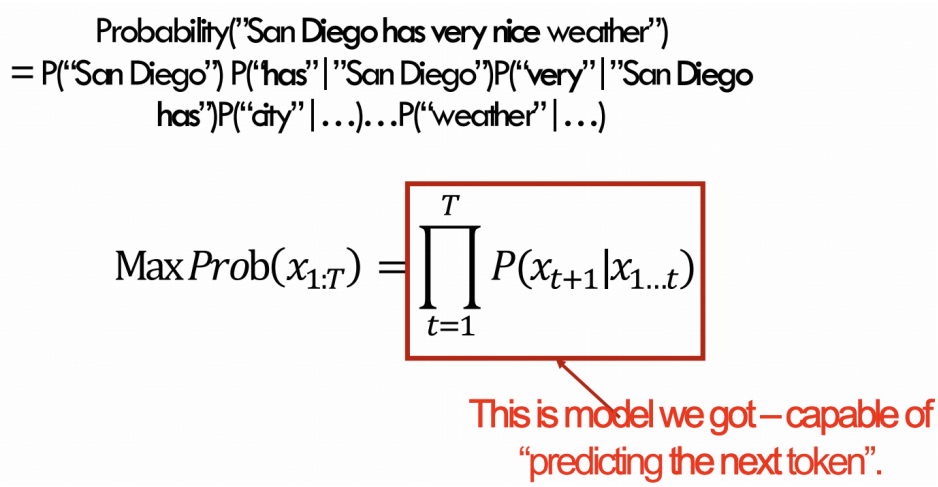


Figure 1: Next token predictor

The LLM inference process is illustrated in Figure 2. The input is called a prompt – a sequence of tokens, which is fed into the LLM to perform a forward pass through all the layers and predict the what the next token is. This is the first step and is called prefill or prompt computation. Then, the next step will be

feeding the newly generated token back to the LLMs to continually generate the next tokens. We repeat this process until we hit the maximum sequence length or the LLM or when the LLM generates a special token called EOS (End-of-sequence).

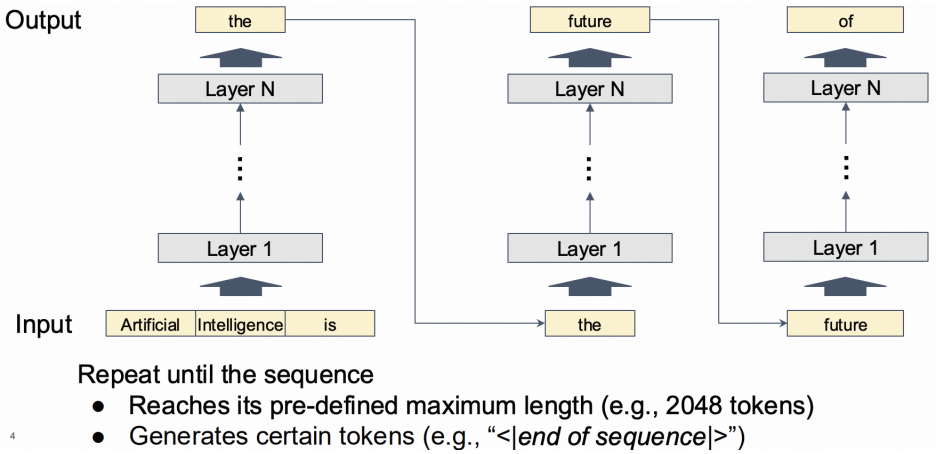


Figure 2: Inference process of LLMs.

From the computation perspective, the LLM inference process can be summarized as two phases:

- **Pre-filling phase**: This phase corresponds to the 0-th iteration of the sequential generation. It is for computing the prompt, where all input tokens are processed together. The pre-fill phase only needs to be conducted once and the new tokens will be processed in the second phase.
- **Decoding phase**: This phase corresponds to all other iterations other than the 0-th one. After the pre-fill phase, the LLM starts to repeat the decoding phase. As the decoding process is autogressive, the LLM only generate one token at each decoding step.

1.2 KV Cache

There is another important data or data structure we need to maintain for LLM inference which is called Key-Value (KV) Cache. The K and V correspond to the Key and Value in the attention computation in Transformer block, and the KV cache here is to help us save some computation. The vanilla computation graph of the attention layer in the Transformer block is shown in Figure 3 below.

Problem: The above computation process or the inference process can be further illustrated by an example shown in Figure 4, where the LLM is generating 4 new tokens one by one. We can tell from this example that every time we generate a new token (e.g., Token 2), we concatenate it with the observed token (e.g., Token 1) generated by the previous step to generate the next token (e.g., Token 3). When we we trying to generate the next token (e.g., Token 3), we only care about the query token of this "next token" (i.e., Query Token 3), and we let this query of next token to attend to the KVs (keys and values) of all previous tokens (e.g., Token 1 and Token 2). In this case, every time we generate a new token, we are computing the Keys and Values of all previous tokens, even if we only need to generate the current token. For example, in step 4, we only want to generate Token 4, but we are still computing the keys and values of Token 1, Token 2, and Token 3, which have already been computed in the previous steps but we are computing them again.

Solution: To save such duplicated computations of the KV of the previous tokens, we introduce KV cache, which basically means we cache the computation results of keys and values at each step, which can be further

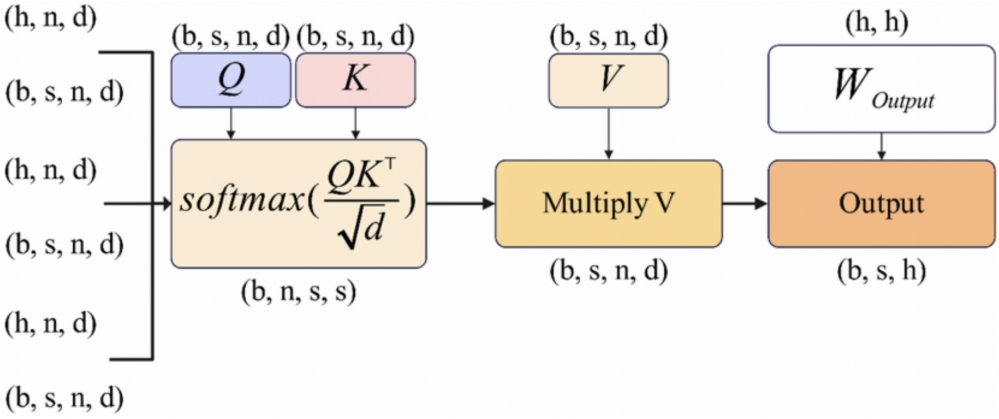


Figure 3: Vanilla computation graph of attention.

utilized by further steps. Figure 5 shows a comparison between the decoding process of the vanilla inference and using KV cache, where the upper half show that in Step 4, we need to calculate the keys and values for all previous tokens (Token 1, Token 2, and Token 3), even if we are only interested in generating the current token (Token 4). In the bottom half, we apply KV cache to all the previous tokens (Token 1, Token 2, and Token 3), where we use the keys and values (marked in purple) that have already been computed in previous steps (Step 1, Step 2, and Step 3). Therefore, KV cache is a kind of computation optimization where we treat some additional memory to save for computation and accelerate the decoding process.

Further Thinking: (1) What happens on KV cache in the prefill phase? There is no KV cache in the prefill phase, because we can see all the tokens, and we just perform one forward across all the tokens and all the KV caches are generate for each token position. So we don't have to cache the previous tokens and wait for the next token to generate. **(2) Do we need to cache Q?** No, because we don't know the query until we generate the "next token". We use the current query to attend to all previous tokens' KVs.

1.3 LLM Inference Bottleneck - Part 1

We look at compute, memory, and communication as potential bottlenecks. In inference, the compute is from the prefill and decoding phases. In prefill, the compute is the same as in training since you observe the entire sequence and feed it to the model. In decoding, since every time we only generate 1 token, s=1. This is a problem and hurts our arithmetic intensity because the matmul sizes are reduced (one dimension becomes 1). For memory, unlike in training when you can compute the KV cache in one pass, compute gradients, update parameters, and throw it away, in inference we can't throw away the KV cache because we are still generating one token at a time, so the KV cache needs to persist till all of the tokens are generated. Compared to training, we do not have the memory pressure from saving intermediate states, parameters, optimizer states from the training backward graph. For communication, it's mostly the same with training. In fact, with inference, parallelism is simpler because we don't have a backward graph (so no gradients or optimizer states related communication).

For batch size b, in training we fix a static batch size, but in inference it's determined by the traffic you receive. We have a customer facing question: should we wait for more requests so we can have a large batch size (increases arithmetic intensity but early users will have to wait a lot) or immediately process each request (will under-utilize GPU)?

Two scenarios: The first is serving, where we have a large b, like Google or OpenAI, lots of requests, and want to minimize cost per query. We want high throughput (how many tokens you can serve per second

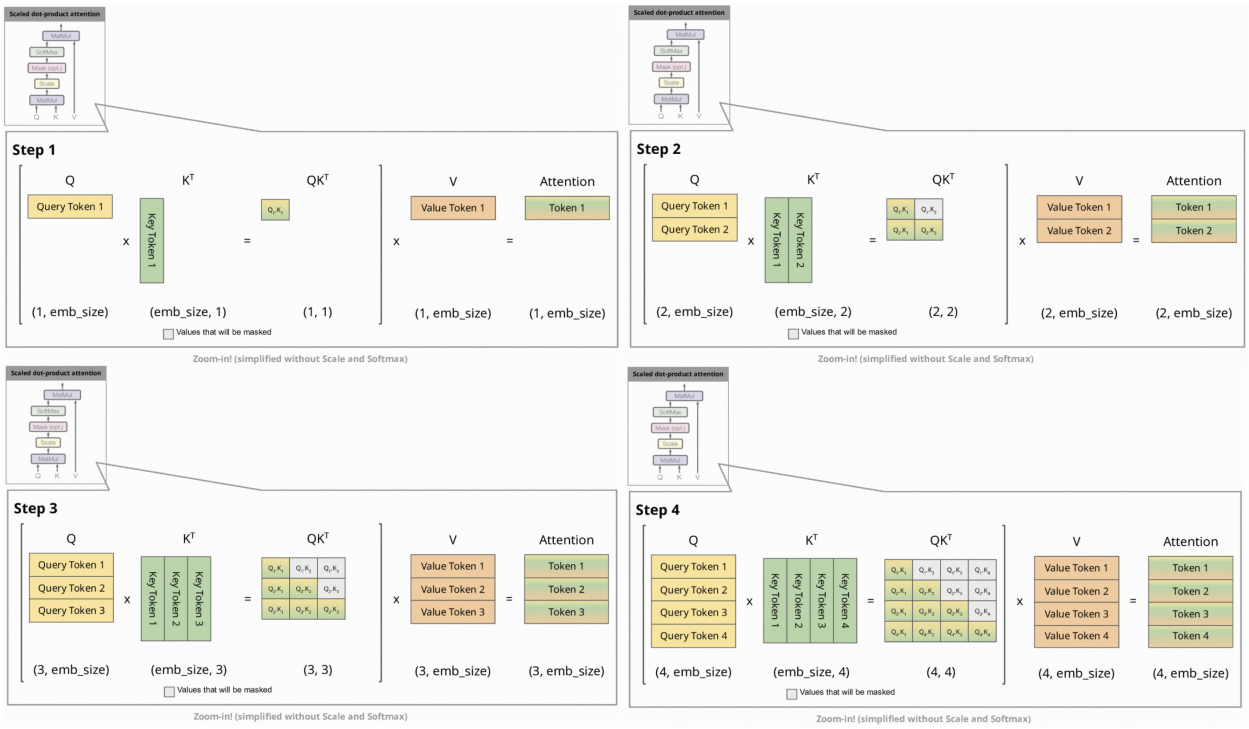


Figure 4: An example of the computation of a LLM generating 4 new tokens during the inference process.

using your fixed number of GPUs).With higher throughput, you can use less GPUs to serve more requests, and get a higher profit margin. The other scenario is inference, where b=1, like when we use the model on our laptop. There are requests, and we want to minimize latency for a single request. Speculative decoding (like EAGLE) helps with this.

Large b: Communication is not a problem still because it's the same as training, if not a little easier. For compute, in prefill the problem with a large batch size is that different users can have different prompts with different lengths, so we need to think about how to batch them. Padding the shortest to the longest is a very bad strategy. In decoding, different prompts have different numbers of tokens generated, which we cannot know. Also s=1 and b is large, so you need a very big batch to make GPU underutilization less severe, which requires strategizing. For memory, KV cache size is linear with b, and remember you can't release the KV cache till its request is finished, so this keeps growing. Continuous batching improves throughput by 20x and PagedAttention helps with the memory problem to manage KV cache, helps you improve throughput by another 3x on top of that, so we've seen 60x improvement in the last 1.5 years.

b=1: Now for memory, the KV cache is not a problem anymore because it's just one request. Communication is the same again. For compute, there's no more problem for prefill since it's just one request, but decoding still has the problem where you don't know how many tokens you'll generate (but this is less of a problem when compared to when b is large). Also for decoding, now s=1 and b=1. Now more dimensions are equal to 1 (like in attention computations) so it's less efficient and GPUs are very underutilized. When b=1 and s=1, GPUs don't perform as efficiently, and we observe lower arithmetic intensity, because the number of floating point operations is small now, but the number of reads between memory hierarchies is the same (to compute one token you have to at least have to move model weights from HBM to SRAM, etc.)

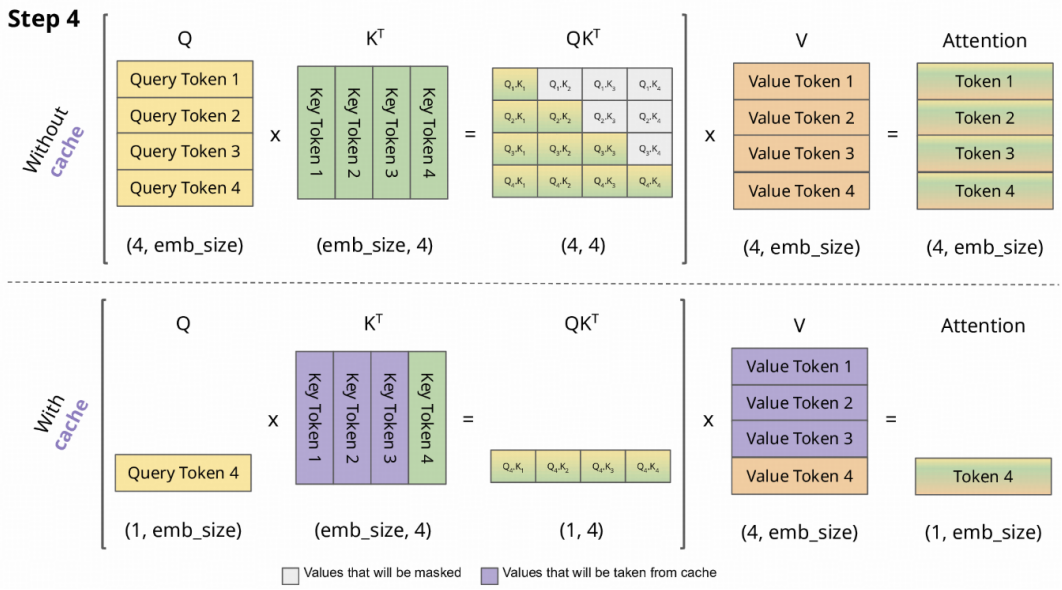


Figure 5: A comparison between the inference processes of not using and using KV cache.

1.4 LLM Inference Bottleneck - Part 2

Here we discuss the inference challenges of LLM in serving from **Compute** and **Memory** perspectives. Note that challenges of LLM in serving (usually with large batch) are different than inference at a smaller scale (small batch, e.g. b=1), as mentioned above. Recall the $AI = \frac{\#ops}{\#bytes}$ for analysis. For s=1 and b=1, compared to serving with large b, the number of operations now become smaller while the number of memory access is still the same, leading to a significantly smaller arithmetic intensity. Speculative decoding works in this case because it reduces the number of tokens decoded during the inference by speculating the decoded tokens by a smaller LM and amortizing the cost of moving from memory to HBM.

Q: Why is speculative decoding not used in large b scenarios? This is because speculative decoding consumes more flops to reduce memory access. However, in cases where b is large, the system becomes compute-bound. And speculative decoding adds more computing, which does not help.

Compute. There are two problems when b becomes large. If batching is random, then different samples in the batch will end up with different prefilling length and different optimal decoding length. For prefilling, the difference in length is obvious, as different prompts contain different words and numbers of tokens. To do batch inference, padding is used to pad all the samples into the maximum length in the batch. Thus samples with smaller lengths need to wait for the sample with the maximum length, wasting a lot of computation. For decoding, different request would lead to different number of optimal decoded tokens. For instance, simple questions and complex reasoning with deep thinking leads to a significantly different number of tokens to be decoded. In batch inference, this also wastes computation, as the request with fewer tokens to be generated needs to wait for the request that are complex with more tokens to be decoded.

Memory. Since batch inference deals with larger b and KV Cache is linear growing with b (since each sample stores a KV cache), the memory usage also grows linearly with b/ If not managed properly, this leads to exploded memory and OOM error, particular in cases with a huge batch size.

2 Continuous Batching

2.1 Traditional Batching

To optimize serving multiple requests, it is desired to batch requests together. Traditionally, requests are batched together, processed, and the output is conveniently returned at the same time due to previous model architectures. However, because the number of generated tokens is unknown in LLMs, this approach poses an inefficiency. Observe that in Figure 6 when S_1 completes at T_6, it must wait until S_2 completes at T_8 to return. The entire batch is bottle necked by the longest number of generated tokens. Requests will complete at different iterations, causing idle GPU cycles while the GPU waits for the longest request to finish generating tokens, since the results are only sent back once all requests complete. Ideally, we want requests to send results immediately and new requests to begin on the next iteration.

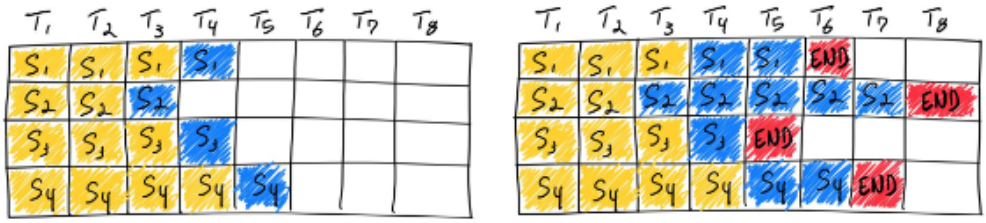


Figure 6: Time diagram of four requests using traditional batching.

Let's explain continuous batching using an example run-through. Suppose at iteration 0, there are two requests waiting to be served. At iteration 1, a new request is received, and at iteration 2, three new requests are received. Each request has a varying sequence length and will generate an unknown number of tokens.

The CPU maintains a request pool that holds requests waiting to be served. The GPU receives and executes batches of requests up to the maximum serving batch size from the CPU. How are the batches executed? Recall that the attention block will not work with varying sequence lengths across the batch. One solution is to pre-pad the sequence lengths for the attention block, which will allow the batch to be processed. The MLP block in the LLM can be easily batched because the MLP module is not dependent on the sequence length.

At iteration 1(Figure 7), while the new request arrives on the CPU, the first two requests need to decode their first tokens. How is the decoding phase batched across requests? While the MLP can be easily batched (with size equal to the number of decoded tokens), the attention block is again troublesome. Modern libraries employ different batching strategies for the MLP and attention blocks. One approach is to independently execute attention for each request, then re-batch the request together for the MLP. During decoding, the batch is un-batched and re-batched again for the attention and MLP blocks respectively.

In iteration 2 (Figure 8), under a traditional batching scenario, new requests R4 and R5 are received. Meanwhile, the decoding process continues for existing requests R1 and R2. However, a significant inefficiency emerges: requests in the queue (R3, R4, R5) cannot enter the GPU batch currently occupied by R1 and R2. Furthermore, requests that complete early (R2 hits EOS token) cannot exit early, as the traditional batching method waits for the entire batch to complete.

At iteration 3 (Figure 13), these inefficiencies are further exacerbated. Requests R3, R4, and R5 remain waiting in the CPU request pool, unable to enter the GPU execution engine because the GPU batch remains occupied with R1, despite R2 already completing its generation. This prolongs idle GPU cycles, resulting

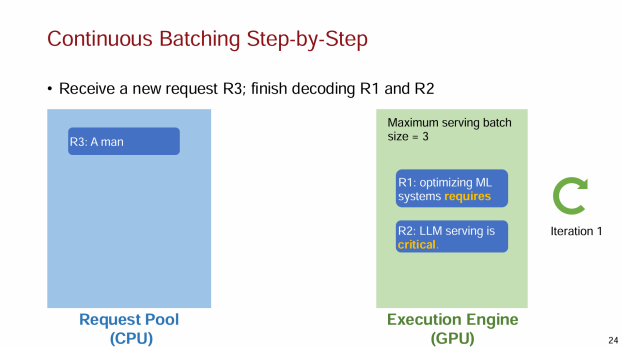


Figure 7: Iteration 1 using traditional batching.

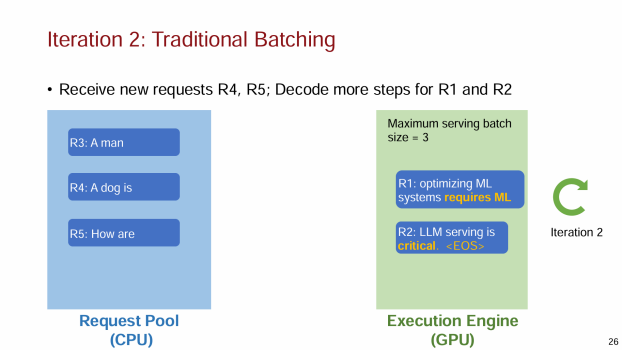


Figure 8: Iteration 2 using traditional batching.

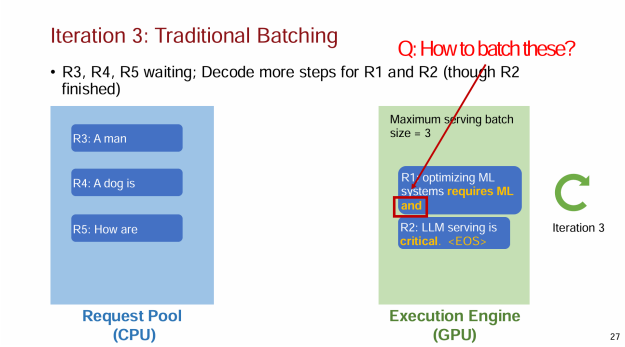


Figure 9: Iteration 3: GPU remains occupied despite early completion of some requests.

in suboptimal GPU utilization, as GPU resources are blocked by ongoing requests whose token generation processes vary significantly in length.

Summary of Traditional Batching Issues:

The primary drawbacks of traditional batching methods are summarized as follows:

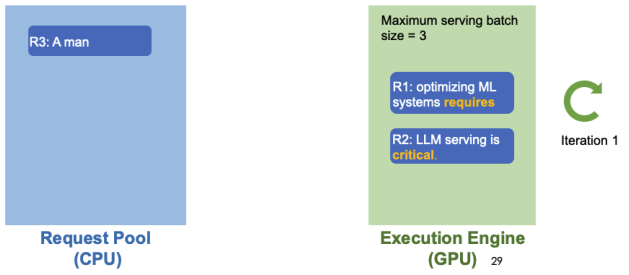
- A batch, once issued, must run to completion before another batch can begin.
- Requests waiting in the CPU pool cannot enter the batch in the middle of an ongoing batch execution.
- Requests that complete generation early (hitting EOS tokens) cannot exit the batch prematurely.
- GPUs may become idle due to the differing and unknown number of generated tokens per request, significantly lowering GPU utilization efficiency.

Continuous batching addresses these limitations by dynamically allowing requests to enter and exit GPU batches, significantly optimizing GPU utilization and throughput.

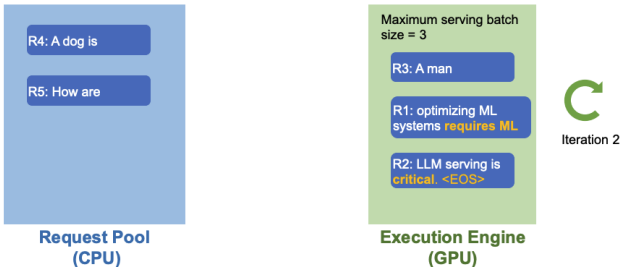
2.2 Continuous batching analysis

We go through a example to explain the process of continuous batching as shown in figure10. In iteration 1, we finish decoding R1 and R2 and receive a new request R3. In iteration 2, we decode R1, R2, R3 and receive R4, R5. At this time, R2 completes. Now, we should pay attention how we batch the R3 with R1 and R2? LLM is basically composed by attention and MLP. For attention part, we cannot batch R3 together with R1 and R2 since the number of token in R3 is quite different from R2 and R1. However, when it comes to MLP layer, R3 can be batched with R1 and R2. The reason is that in MLP, we don't care about the length of the token, we just take each token project up and project down. In other words, it is token independently. Therefore, we can break down attention and MLP, and we do different batching strategy for different two parts of the graph. What's more, according to the previous lecture, we know that MLP is going to take 90% of the flops. For attention, it is acceptable to compute it sequently. Thus, continuous batching is an efficient way to improve inference process. In iteration 3, we decode R1, R3, R4 following the continuous batching until all the requests are completed.

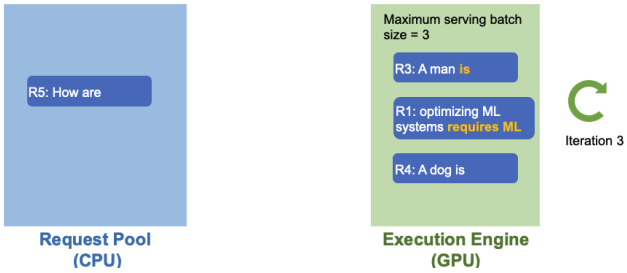
In summary, the advantage of continuous batching includes: (a) handle early-finished and late-arrived requests more efficiently (b) improve GPU utilization.



(a) iteration 1



(b) iteration 2



(c) iteration 3

Figure 10: Continuous batching

3 Page Attention

3.1 KV Cache:

KV Cache is like a temporary storage area in your computer's memory that helps large language models (like AI) work faster. It stores information about words (tokens) that the model is processing. Efficient management of KV Cache is super important for making AI models run smoothly and handle lots of tasks at once (high-throughput LLM serving). If we don't manage it well, the system slows down, wastes memory, and becomes less efficient.

The 'Efficient KV Cache Management for High-Throughput LLM Serving' figure highlights the importance of efficiently managing the KV cache for high-throughput Large Language Model (LLM) serving. It shows that in a 13B LLM deployed on an A100-40GB GPU, 33% of the memory is occupied by the KV cache (13GB), while the remaining 65% is used by model parameters (26GB). The graph compares memory usage with traditional systems (existing systems) and the vLLM approach. In existing systems, memory usage increases linearly with batch size, while vLLM manages memory more efficiently. The dashed blue line indicates the vLLM system, which shows a more controlled growth of memory usage as batch size increases.

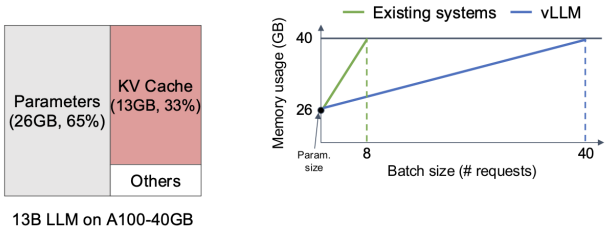


Figure 11: Efficient KV Cache Management for High-Throughput LLM Serving

Efficient KV cache management in vLLM allows for better throughput without excessive memory usage, especially as the number of requests scales up.

Even though KV Cache is super useful, it has some big issues with wasting memory:

1. Reserved but Unused Space: Sometimes, the KV Cache saves space for words it might need in the future, but it doesn't end up using that space. This is called reservation waste.
2. Internal Fragmentation: The KV Cache often allocates more memory than it actually needs because it doesn't know how long the output will be. This is called internal fragmentation.
3. External Fragmentation: Sometimes, the memory gets split into small, unusable chunks. This makes it hard to use the memory efficiently.

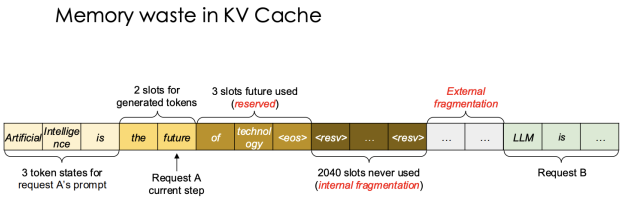


Figure 12: Memory Waste in KV Cache

The 'Memory Waste in KV Cache' figure illustrates memory waste in the Key-Value (KV) Cache of Large Language Models (LLMs) during inference. It shows that memory is reserved for future tokens, leading to internal fragmentation when the reserved slots are not used, and external fragmentation when gaps appear between requests. There are unused slots in the cache, such as 2,040 slots that are never utilized, wasting memory space. Efficient KV cache management is crucial to reduce this fragmentation and improve memory usage during LLM inference.

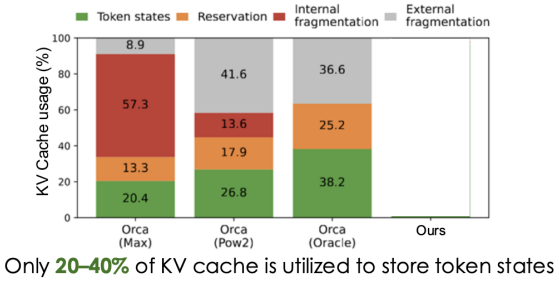


Figure 13: Actual KV cache utilization

In the 'Actual KV cache utilization' diagram, only 20–40% of the KV Cache is actually used to store token states. This means 60–80% of the memory is wasted due to issues like internal and external fragmentation, or unnecessary reservations. Most of the KV Cache is sitting idle, which is a huge waste of resources.

3.2 PagedAttention Algorithm

In paged attention, we first organize the request prompt into logical token blocks, ensuring logical continuity. Using a block table, we map these tokens to physical token blocks, tracking how many tokens are stored in each KV cache line.

During text generation with paged attention, we logically append newly generated tokens and update the block table accordingly. For example, if we predict the token "and", the block table updates the count of filled tokens in the corresponding physical block. If all blocks are filled, a new memory block is allocated to store incoming tokens. While these blocks may not be physically contiguous in memory, they remain logically contiguous.

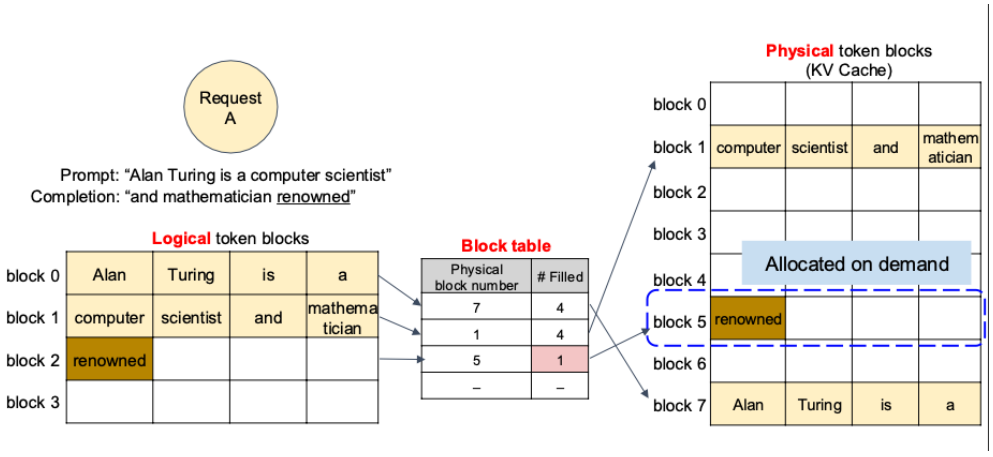


Figure 14: The Workflow for PagedAttention

When handling multiple requests, each request is assigned different physical blocks, which may also be non-contiguous in memory.

This approach significantly improves memory efficiency for large language models (LLMs). In their work, the authors developed an LLM inference platform called vLLM, which minimizes memory waste. In vLLM,

memory overhead is limited to the last block of each sequence, ensuring that the wasted tokens per sequence do not exceed the block size. Typically, the block size is set to 16-32 tokens. Given sequence lengths on the order of O(100)–O(1000) tokens, the maximum possible waste is at most 31 tokens.

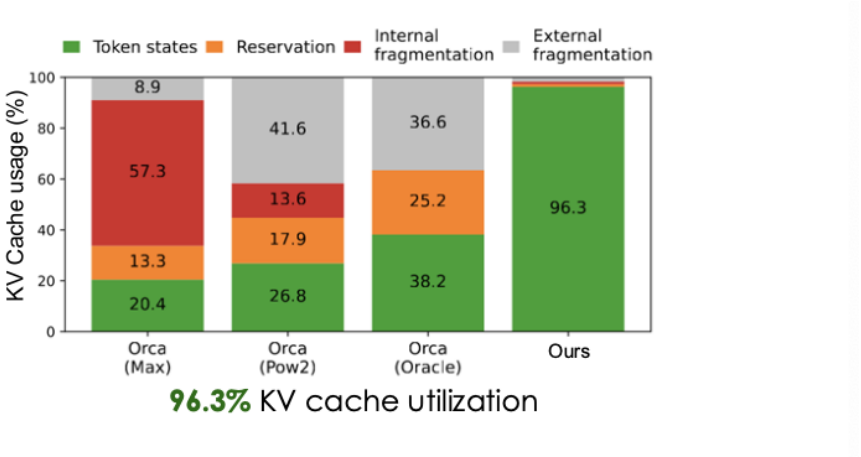


Figure 15: KV Cache Usage of Orca and vLLM

3.3 vLLM: Efficient memory management for LLM inference

In traditional operating systems, virtual memory allows processes to access memory beyond what is physically available. Paging divides memory into fixed-size pages that can be loaded or swapped as needed. It helps prevent fragmentation, allows for efficient multitasking, and enables virtual memory, making it possible for processes to use more memory than what is physically available. The same approach extends to vLLM, applying this concept to LLM inference. Instead of memory pages, we have Token Blocks in the KV Cache. Requests dynamically fetch token blocks without storing the entire sequence in memory. This optimizes KV Cache usage, allowing multiple requests to share cached tokens, reducing redundancy and memory overhead.

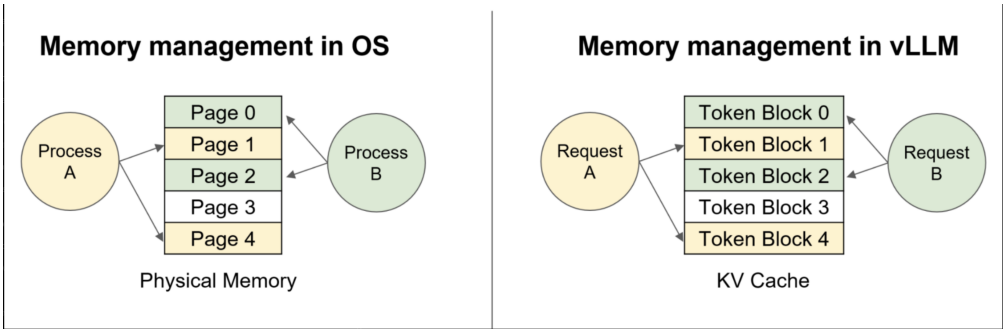


Figure 16: Similarity in Memory Management in OS and vLLM

So what are **token blocks**? It is a fixed-size contiguous memory chunk used to store token states. Each token in a sequence needs to store key-value (KV) pairs, which are used during attention computations. Instead of storing KV pairs inefficiently, vLLM organizes them into blocks. The KV Cache stores precomputed key and value embeddings for previously generated tokens, allowing for faster inference by avoiding recomputation.

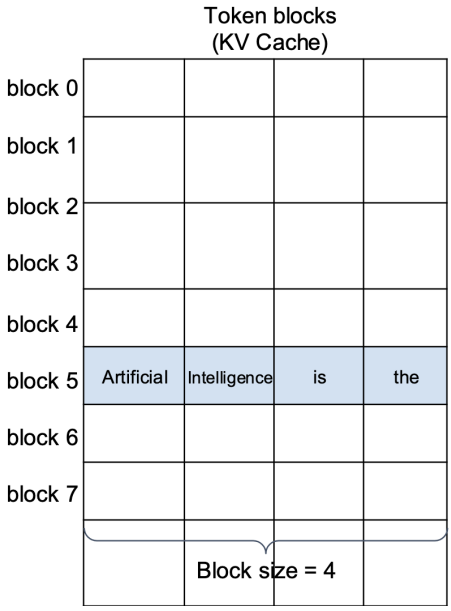


Figure 17: Example

For example, we have a sequence of words "Artificial Intelligence is the". Each word (token) is mapped to numerical values (tensor representations), which are stored in Block 4 inside the KV Cache. The block size is 4, meaning each block can hold up to 4 tokens before needing to use the next available block. Block 4 stores key-value tensors for the words "Artificial", "Intelligence", "is", and "the". These words are now allocated a contiguous chunk of memory, allowing fast retrieval during inference. The stored values (820 KB per token for LLaMA-13B) show that memory usage can become very large, reinforcing the need for efficient caching mechanisms like vLLM's paged attention.

3.4 Logical vs Physical Token Blocks

PagedAttention allows models to store keys and values in non-contiguous memory blocks, improving efficiency. This approach is inspired by virtual memory paging in OS, where logical pages don't need to be stored in contiguous physical memory. This is illustrated by Figure 19 where logical token blocks display how the model conceptually groups tokens, and how physical token blocks reflect how they are stored in memory.

Additionally, there exists a block table serving as a mapping index, linking logical blocks to their corresponding physical locations in KV cache. We can think of this as a page table in OS virtual memory, mapping logical pages to actual physical memory locations.

For example, for a prompt "Alan Turing is a computer scientist" is already stored in logical token blocks and mapped to physical KV cache blocks. Now, the model generates the next token: "and". Since Logical Block 0 (Alan, Turing, is, a) is mapped to Physical Block 7, the newly generated token "and" would be added to the next available position in a partially filled block (Block 1) and increment the **Filled** count until it was full. Instead of allocating a new block, tokens are placed into available slots within existing blocks. This avoids unnecessary memory fragmentation while enabling fast token retrieval.

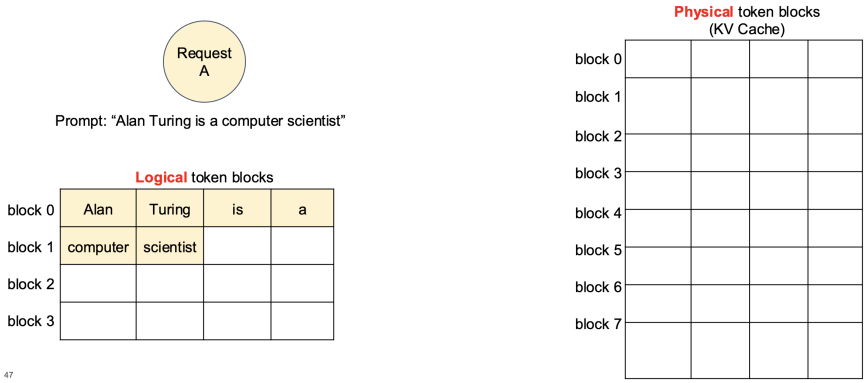


Figure 18: Logical vs Physical Token Representation

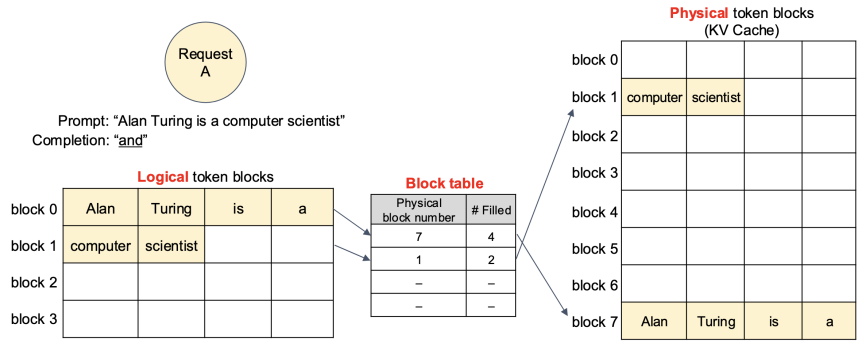


Figure 19: Logical vs Physical Token Representation

4 Prefill-Decode Disaggregation

4.1 Throughput vs. SLOs

In recent years, especially 2023 and 2024, ML researchers have tried to maximize the so-called "throughput" metric, which involves maximizing the number of tokens that can be served per second given a limited number of GPUs. Recent works have pushed the throughput value increasingly higher - for instance, some works have attained 3.5x and even 24x higher throughput compared to the previous state-of-the-art.

However, it is important to note that many LLM applications are offered as a service to users. Therefore, for some services, there are some other metrics that are equally important besides throughput - these are called **Service-Level Objectives (SLOs)**. An SLO is a metric that many companies want to guarantee, and applications can have diverse SLOs.

4.1.1 Motivation: Applications that have Diverse SLOs

In LLMs, there are two main SLOs:

1. **Time To First Token (TTFT)**: represents the initial response time.
2. **Time Per Output Token (TPOT)**: is the average time between two subsequent generated tokens.

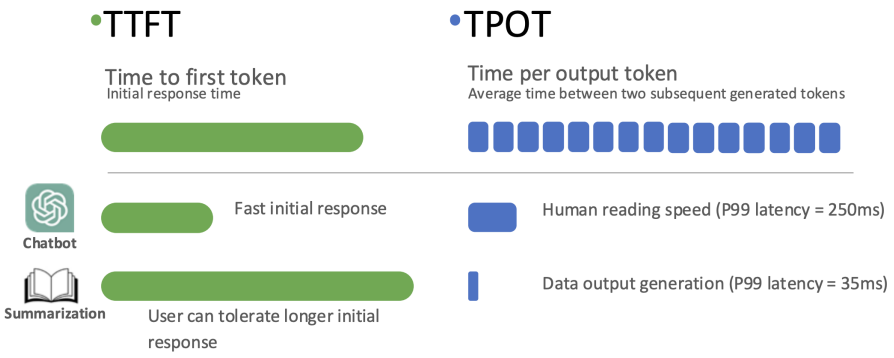


Figure 20: TTFT and TPOT of a Chatbot vs. LLM Summarization.

A chatbot, like ChatGPT, needs to have a relatively fast initial response, i.e. a lower TTFT. Put simply, when a user submits a prompt to the chatbot, they expect to see the first output word as soon as possible. In such a case, if the user needs to wait, then the user experience is impacted. As an example, in ChatGPT, each token is streamed one-by-one, meaning that once the first token is generated, the user starts reading from this first token. For a low TTFT, this first token must be generated quickly.

Generally, as long as the service provider can generate tokens faster than a user can read the tokens (i.e. reading speed), the user experience is good. This is because the user is unlikely to read the tokens faster than they are generated. Hence, in order to maintain a positive user experience, the TTFT needs to be low. But, after the first token is generated, the generation of the subsequent words (measured by TPOT) does not need to be faster than the average human reading speed (approximately 150 English words per minute).

In contrast, for a different task such as LLM summarization, the user can tolerate a longer initial response because when the user tries to submit a batch of documents to the LLM, they do not want to see the first

token immediately. Instead, in this case, the user would likely want to see the eventual results as early as possible. Hence, now the TPOT is more important, with a lower priority placed on TTFT.

The comparison of TTFT and TPOT for a chatbot versus LLM summarization is illustrated in figure 20.

4.1.2 High Throughput vs. High Goodput

In the presence of the two SLOs, TTFT and TPOT, throughput may not be a sufficiently good metric for LLM serving systems. To explain why, we look at an example, which is portrayed in figure 21.

Suppose we have a system that can serve 10 requests per second (i.e. the system capacity). Now, we place constraints on TTFT and TPOT, i.e. we require that the system also satisfy the SLOs. In figure 21, we specify that TTFT stays within 200 milliseconds, and TPOT stays within 50 milliseconds. However, after placing the above constraints, we see that many requests do not satisfy the constraints - these do not meet the service level quality.

We call the portion of the throughput that is subject to the constraints as **"Goodput"**, i.e. the "good" part of the throughput (this concept could be associated with a similar concept in computer networks). In the example shown in figure 21, the system has a low goodput of only 3 requests per second, despite having high throughput.

Thus, high throughput does not always translate into high goodput once we factor in latency, which ultimately leads to a poor user experience (UX). As such, we must ask the following question: why do existing systems fail to achieve a high goodput? To answer this, we need to explore the challenges faced by traditional continuous batching.

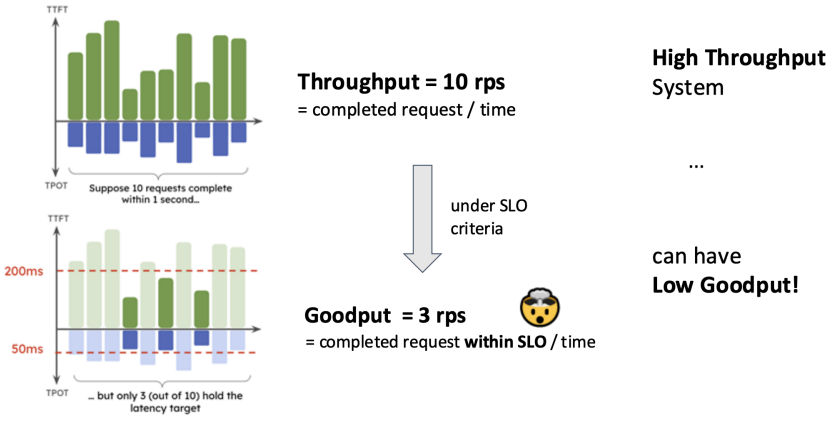


Figure 21: Throughput vs. Goodput

4.2 Challenges with Traditional Continuous Batching in LLMs

4.2.1 How a request goes through an LLM (Traditional Method)

In the traditional method, once an LLM recieves and incoming request, the request leaves the system only after all the prefill and decode steps are all done. That is, the prefill and decode steps are batched together. If Prefill step is heavy, decode steps are going to be delayed because of the high workload of prefill.

4.2.2 Distinct Computational Characteristics:

Prefill and decode both have very distinct computational characteristics. The Prefill step is compute bound, and just one prefill can potentially saturate the compute. Decode, on the other hand is memory bound. During decode step, since s = 1, a lot of requests needs to be batched together inorder to saturate compute.

4.2.3 Problem with continuous batching:

Continuous batching can put both prefill and decode step into the same batch. These two phases can cause interference with each other and cause delay, as shown in figure 22. In the left part of the figure, we see that after R1's decode step is done, there is significant delay since it is waiting for R2's prefill step to be done. This can cascade to mugh higher delays when more requests are batched together in a single GPU, as shown in the right side of figure 22.

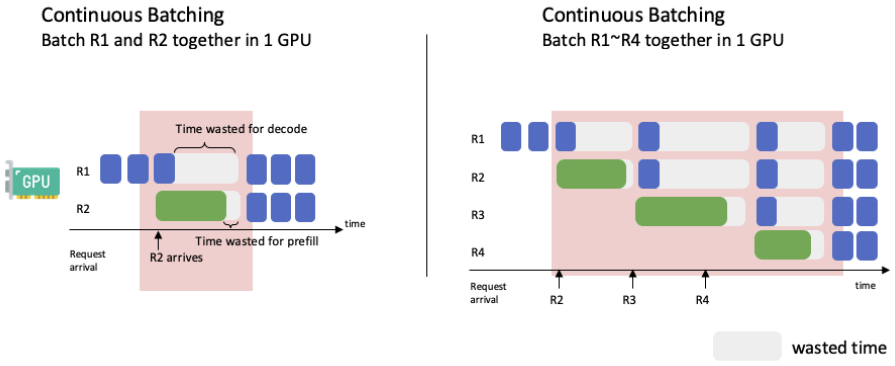


Figure 22: Continuous batching cause interference

4.2.4 Potential solution:

This problem can potentially be solved by using multiple GPUs, and separating prefill and decode steps such that they run on separate GPUs. As shown in figure 23, we see that by adding another GPU, R1 and R2 can run separately, and hence there's no delays due to interference.

However, even though this solution leads to a better user experience, it leads to over-provisioning of resources in order to meet the SLA, which results in higher cost.

4.3 Addressing the Challenges

As discussed above, one approach to mitigating the delays caused by continuous batching is *Colocation*. This technique aims to enhance system performance by allocating multiple GPUs to meet Service Level Objective (SLO) expectations. However, while colocation improves processing efficiency, it comes at a high cost. To further optimize resource utilization and achieve better goodput per GPU, a more effective approach called *Disaggregation* is introduced.

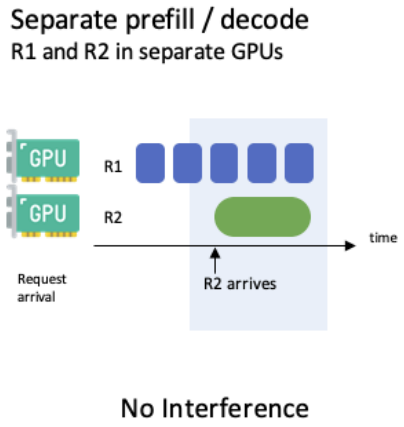


Figure 23: Increasing number of GPUs

4.3.1 Colocation

Colocation involves over-provisioning resources by employing multiple GPUs to ensure that the system meets SLO requirements. In this setup, both the prefill and decode phases remain on the same device, ensuring minimal latency in data transfer between stages. While this method effectively reduces processing delays and enhances system responsiveness, it incurs a significant cost due to the increased resource allocation.

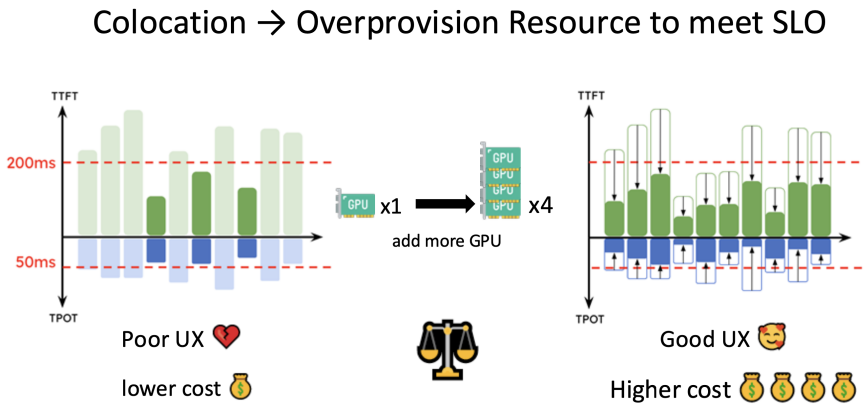


Figure 24: Improved SLOs with Colocation but with a high cost

4.3.2 Disaggregation

Disaggregation provides a more efficient utilization of resources, driven by the observation that the prefill phase takes considerably more time compared to the decode phase. By performing the prefill and decode phases on different devices, the decode process does not remain idle while waiting for the prefill process to complete, thereby reducing resource wastage. One potential drawback of this approach is the need to communicate the KV cache to another device after the prefill phase. However, since this transfer occurs only once per request, its impact on overall goodput remains minimal.

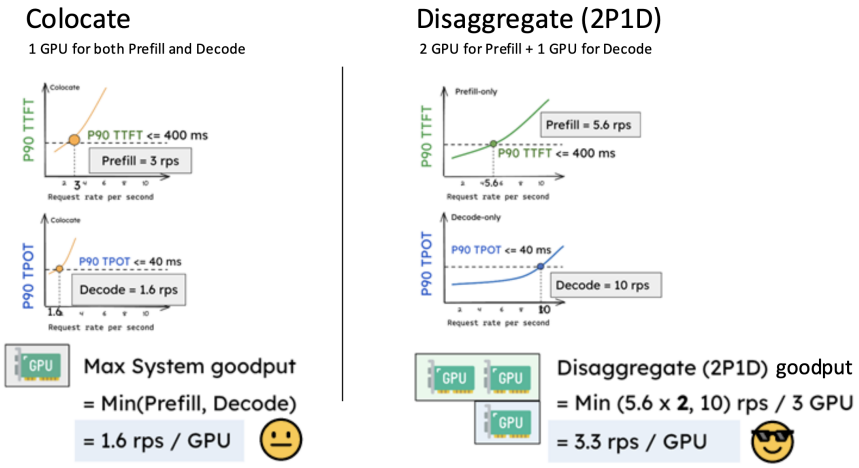


Figure 25: Disaggregation shows 2x improvement of goodput per gpu

In conclusion, Disaggregation, introduced in 2024, quickly became the preferred architecture for large-scale systems, replacing continuous batching. Its adoption is exemplified by Deepseek-v3, which leverages prefill-decode disaggregation alongside various parallelism techniques to enhance performance and efficiency.

5 Scribe Contribution

- **Feng Yao**: section 1.1 & 1.2
- **Aarti Lalwani**: section 1.3
- **Yijiang Li**: section 1.4
- **Junfei Liu & Inan Xu**: section 2.1
- **Weifan Yu**: section 2.2
- **Nandakrishna Kanakapuram Srinivasadeshikachar**: section 3.1
- **Xunyi Jiang**: section 3.2
- **Srikrishna Dantu**: section 3.3 & 3.4
- **Abhinandan Padhi**: section 4.1
- **Sathvik Balakrishna**: section 4.2
- **Veeramakali Vignesh Manivannan**: section 4.3