LLM Inference

As LLMs grow in popularity and as more people, businesses, and enterprises use them, some of the focus is shifted towards having them run efficiently during inference, or actual use. Training a model requires immense compute and money, but so does running it and allowing millions or even billions of people to use it. Inference is the stage where a model actually generates a response for a user. Unlike training, which is a one-time expense, inference happens billions of times a day. Without the speedup methods found in libraries like **vLLM**, serving these models would be prohibitively expensive and frustratingly slow.

Memory Bound vs. Compute Bound

In high performance computing, there is a tradeoff between the available amount of compute (CPU/GPU speed) and memory (moving/storing data). LLM inference is largely memory bound. It takes more time to move weights around than to actually do matrix multiplication. One common solution is **quantization**, reducing precision of weights, from 16-bit floats to 8-bit or 4-bit. This directly reduces the amount of data that needs to be transferred to and from the GPU.

General LLM Inference Flow: 2 stages: prefill and decode

- **Prefill (compute bound)**

  For each layer 1 to n:
  - During prefill, the model takes all the input tokens from the prompt and creates the Q, K and V vectors for each token. It also saves the K and V vectors for each token in its **KV cache**.
  - Using these vectors, it calculates the self attention matrix $Q(K^T)$ where each row corresponds to a token. Then it scales them and applies softmax row wise to get probabilities, then multiplies this with the matrix V (stacked the V vectors of each token).
  - These values are passed through another pretrained layer and then the initial prompt is added to this (both are matrices of the same size tokens x hidden dimensions so they can be added). This is called **Residual Connection.**
  - Now, each token (a token still corresponds to a row) is sent through a feed forward network (FFN). Each row is passed through the FNN independently, but GPUs can make this essentially parallel.
  - *Note: the above will be repeated if there are multiple transformer blocks; large models can have hundreds. This is called 1 layer. We store the KV vector of each such layer, and as the input of one layer is the output of the last, they get more and more unique as they get passed through layers.*
- After the (last) FFN, we get an output of the same size we passed in, and the last row from this is taken and a similarity search (dot product) between it and the vocabulary matrix is done. Most similar token is chosen

- **Decode (memory bound)**

  For each layer 1 to N:
  - It takes the newly generated token and calculates Q, K and V for it and puts the K and V vectors in the KV cache
  - It fetches the K and V values for all previous tokens from the KV cache
  - Multiplies the Q of the new token with all previous Ks for the current layer stored during prefill, generating a row of attention scores.
  - It uses these attention scores as weights and multiples them by the V vectors of all the new and all previous vectors from the current layer stored during prefill. This dot product happens element wise. This results in a context vector.
  - The context vector is multiplied by a trained output matrix. The original vector for the newest input is added onto this (**Residual Connection**)**.**
  - This is passed through dense feed forward layers.
  - *Note: the above will be repeated if there are multiple transformer blocks; large models can have hundreds. This is called 1 layer. We store the KV vector of each such layer, and as the input of one layer is the output of the last, they get more and more unique as they get passed through layers.*
- After the FFN, we get an output of the same size we passed in, and the last row from this is taken and a similarity search (dot product) between it and the vocabulary matrix is done.
  - This step is where temperature comes in, if it is > 0. Once the similarity search is done, temperature can change the final probability distribution so the chosen word can be different.
- The loop repeats until the end token is reached.

Similarities
- It is the same transformer weights and model during both phases
- Same number of layers (as it's the same model)

Differences
- Input passed in.
- For the decode, only the last generated token is passed through whereas in prefill the entire prompt is passed through.
- In prefill the self attention is calculated where every token is compared with all the ones that came before using a causal mask.
- Prefill is $O(N^2)$ as every token "looks at" all the ones before and Decode is O(N). N is the input number of tokens.

The input embeddings of tokens are precalculated, and so are the weights of the model, then given the same prompt, why doesn't the LLM give the same output? The answer is temperature.

Even with temperature = 0, floating point errors and other inconsistencies can occur and cause variations.

Given this high level overview, we can look at some techniques used to make LLMs faster and more convenient. One leading library used that implements these techniques is **vLLM**.

In one **iteration** of vLLM, it goes through the entire process 1 time, but in that one time it can do tasks for multiple users at different points in their generation, such as generating token 23 for a user, 45 for another and prefilling for another. It does this by stacking the inputs for different users in a matrix and running it through the transformer. This way, the responses are mathematically separate at the end as well.

Some Techniques for faster Inference

PagedAttention

vLLM introduced an elegant idea called PagedAttention based on paging used in allocating memory by an OS. As mentioned earlier, the KV cache is essential so we do not have to recompute older values .This grows very large in memory and results in significant wasted space.

Traditionally, the KV cache of a user had to be a contiguous block of memory. Extra memory had to be assigned to each user as there was no way to know how much a user might ask. This situation is similar to that of dynamic arrays. Also, if the initial amount of memory assigned to a user was too large for a particular disk/VRAM, it would have to be saved elsewhere and the rest of this space would be wasted space.

PagedAttention divides the KV pairs into hashable blocks of 16 tokens which can be stored non-contiguously, thus saving using all available memory, minimizing the waste.

Continuous batching

Traditionally, the GPU would finish one batch of users first, and not worry about the others until they were complete. If 4 people were taken in a single batch and the answer for 1 of them was done, it would wait until the other 3 are finished to bring in a new batch. With continuous batching, it brings in a new user as soon as one is answered.

Chunked Prefill

Traditionally, a large prompt (e.g., 4,000 tokens) would be processed in one massive "prefill" burst. Because prefill is compute-intensive, it would "hijack" the GPU, causing a complete pause for all other users currently in the Decode phase. This results in "stuttering" or high inter-token latency (ITL). Chunked Prefill breaks a large prompt into smaller, manageable pieces (e.g., chunks of 512 tokens).

Instead of waiting for a 4,000-token prefill to finish, the GPU prioritizes decoding for existing users, so they get atleast 1 token generated and splits the large new prompt into chunks then processes the next chunk. It balances the workload so that new users get a fast start without ruining the smooth experience of existing users.

This is done using a token budget for each iteration of going through the transformer. It first fills this token budget with decode, then the remaining with prefill from new users.

Prefix Caching (Automatic Prefix Caching)

In many applications (like Chatbots or RAG), users send prompts that share a large amount of identical text, such as a "System Prompt" or a long context document. Traditionally, the model would re-calculate the KV cache for this identical text every single time a new message was sent.

Prefix Caching uses a hashing mechanism to recognize identical blocks of text. vLLM hashes the content of each KV block. If a new request arrives with a prefix that matches a hash already in memory, it simply points to the existing KV blocks instead of recomputing them. This makes processing long, repetitive contexts nearly instantaneous, reducing Time to First Token (TTFT) and saving GPU compute.

Speculative Decoding

LLM generation is memory-bound, and so speculative decoding tries to "guess" multiple tokens at once to make better use of the movement of weights. The main point is to try to guess multiple tokens ahead of the current one and then get them double checked by the LLM.

The LLM can double check these tokens in a single pass due to the causal mask. It applies the causal mask and gets the likely next vocab word for each token position in the input prompt, including the ones guessed. If it matches, the prediction is accepted. If not, the larger model's choice is used and the rest of the predictions are discarded. This cycle continues.

If we recall the prefill and decoding phases mentioned above, there might be a contradiction that the decoding phase only takes 1 token as the input, and uses the KV cache to predict the next token. We need self attention in order to double check these new "guessed" tokens by the smaller LLM. For this method, we essentially use the prefill phase to generate probabilities for each new token, and correct them if needed. The K and V vectors of the

previous tokens are used to check these new token probabilities. Despite this, it is still called decoding as the prefill phase means generating the first new token, and decoding means generating the rest of them.

There are a few methods to try to predict the next few tokens:

1. Standard Speculative Decoding (Draft Model)
Uses a tiny "Draft Model" (e.g., a 100M parameter model) to quickly guess 3–5 tokens in a row. The large "Target Model" (large main model) then checks all those guesses in a single parallel pass.

2. N-Gram Speculative Decoding (Prompt Lookup)
Instead of a second model, this looks at the input prompt to find patterns. If the model is currently writing "The capital of," and that phrase appears elsewhere in the prompt followed by "France," it "guesses" that "France" is the next word.

- This is extremely fast because it requires zero extra neural network math.

3. Medusa / Eagle (Draft-Head Speculative)
Instead of a separate draft model, these add small "heads" (extra layers) to the top/end of the existing large model.

- **Medusa:** Uses multiple independent heads to predict Token+1, Token+2, etc., simultaneously by looking at the hidden state.
- **Eagle:** More advanced than Medusa; it looks at the internal "features" (hidden states) of the model and tries to predict the next hidden states, then makes its prediction. It is currently one of the highest-performing speculative methods.

Flash Attention

Standard attention from the original Transformer calculates an N x N matrix where N is the number of input tokens. This is very large for bigger inputs and will not fit in the GPU's fast memory (SRAM) so it has to keep writing to and reading from slower memory (HBM). The creates another memory bound problem where moving the weights and numbers takes more time than actual math.

The output of the Attention mechanism: $\text{softmax}(QK^T)$ V is an N x d matrix, N is number of input tokens and d is hidden dimension size.

FlashAttention uses tiling to produce the same matrix piece by piece. Instead of calculating the whole N x N QK^T, it calculates it in pieces, multiplies by V and keeps track of the softmax as it goes through the pieces. It breaks the Q, K, V matrices into small blocks that do fit into SRAM. and calculates the attention matrix block, then multiplies by V while keeping track of Softmax. It will continue to accumulate and save this in the slow (HBM) memory. The key save is not having to move weights continuously to and from HBM. It calculates a whole block, saves it, then overwrites it with the new block calculation, and so on.

Splitting work across multiple GPUs

LLMs which reach several tens of billions of parameters are too large to run on even the most expensive modern GPUs, so they must be split across multiple. This is also taken care of by inference engines like vLLM.

2 main ways to split work across multiple GPUs:

- Tensor Parallelism
    - Splitting the weights in a layer between 2 or more GPUs
    - When an input comes in, it sends it to both GPUs, which each compute half of the attention matrix for example then combine their answers (all-reduce) to a single matrix.
    - Requires GPUs to communicate frequently (twice per layer)
    - Can require special fast communication hardware between GPUs.
- Pipeline Parallelism
    - Used with many GPUs across servers.
    - Splitting the layers themselves between multiple GPUs.
    - The first GPU does its calculations then hands it down to the next one and so on.
    - This can lead to a problem where GPU 1 is doing work while the rest are idle.
    - Fixed by sending different users through the pipeline at different times, like an assembly line.

For large scale deployments with hundreds of GPUs over several servers, vLLM does both these ways. In a single server, GPUs are connected via a fast NVLink so it uses Tensor Parallelism. Between servers, it uses Pipeline Parallelism.

The KV cache was difficult to manage so that different GPUs and servers could interact and view it to avoid recomputing older calculations.

Recently, vLLM introduced support for distributed cache maintenance.

Implementation

I tried out some of these techniques using vLLM in Google Colab. To implement vLLM on Google Colab, I had to overcome the environment's inherent limitations as an interactive notebook rather than a production server. I used the following link as a guide through the process of using vLLM in Google Colab: https://colab.research.google.com/github/apache/beam/blob/master/examples/notebooks/beam-ml/run_inference_vllm.ipynb#scrollTo=OsFaZscKSPvo

Because vLLM operates as an OpenAI-compatible API service that runs indefinitely, I used the `subprocess` and `os` modules to spawn the server in a separate background process group. This allowed the notebook to remain responsive so I could execute test scripts while the model was live. I also implemented a cleanup routine using `os.killpg` and `pkill` to manually terminate the background processes and free up the GPU memory between test runs, preventing the "Out of Memory" errors that occur when multiple model instances collide.

Automatic Prefix Caching

To evaluate the impact of **Automatic Prefix Caching (APC)**, I designed a benchmark using Apache Beam to orchestrate 80 high-concurrency inference requests against a single 3,000-token technical document. In a standard LLM workflow, every unique question about the same document requires the GPU to re-process (prefill) the entire context from scratch, leading to redundant computation and significant latency. By integrating the `vllm_inference` handler within a Beam pipeline, I could systematically measure how caching the initial document's KV (Key-Value) states eliminates this redundancy when the prefix remains identical across multiple prompts.

The technical implementation required using the **Apache Beam** RunInference API to manage the model's lifecycle across a distributed data parallel workflow. I configured two distinct pipeline runs: a baseline with `enable_prefix_caching` set to false and a test run with it enabled. I also implemented a custom `ProcessStats` class as a `DoFn` (Do Function) to inspect the inference objects and extract completion token counts, which allowed me to verify that the model was generating equivalent outputs while the underlying compute time varied.

The results highlighted an efficiency gain, as the cached run bypassed the quadratic cost of the prefill phase for all requests after the first. In the baseline run, the GPU was bottlenecked by repetitive matrix multiplications for the same 3,000 tokens across 80 different iterations. With prefix caching enabled, vLLM identifies the shared "hash" of the document prefix and retrieves its pre-computed KV cache from memory. This shifted the performance profile from a compute-bound prefill task to a memory-bound generation task, resulting in a dramatic speedup factor and a significantly higher tokens-per-second throughput for the L4 GPU.

Chunked Prefill

The second major challenge involved managing the GPU's memory and execution flow to simulate real-world traffic. Since vLLM typically reserves 90% of available VRAM for its KV Cache, I tuned the `--gpu-memory-utilization` to 0.85 to provide a safety buffer for Colab's system overhead. To demonstrate the performance difference between sequential and interleaved processing, I utilized Python's threading library to launch two concurrent requests: a "Blocker" with a massive 3,500-token context and a "Victim" with a simple query. This setup

forced the vLLM scheduler to manage contention, simulating how a server handles multiple users in a production environment.

The experiment successfully demonstrated the impact of "Chunked Prefill" on system latency. In the sequential configuration, the "Victim" request was forced to wait for the entire prefill of the "Blocker" to complete, leading to high Time-to-First-Token (TTFT) metrics. By enabling `--enable-chunked-prefill` and setting a specific `--max-num-batched-tokens` limit, I allowed the scheduler to break the large prefill task into smaller slices. This enabled the smaller "Victim" request to be "interleaved" between chunks of the larger task, significantly reducing the wait time and illustrating how modern inference engines optimize for responsiveness without sacrificing total throughput. One interesting point to note was that when chunked prefill was enabled, it sped up both the victim and the blocker times, not just the victim.

Concurrency Stress Test

I conducted a concurrency stress test to measure the impact of vLLM's **continuous batching** and dynamic memory management. By launching 150 simultaneous requests, I compared a restricted baseline against an optimized configuration where I increased `max_num_seqs` to 128 and expanded the GPU memory utilization to 90%. This allowed vLLM to fill the available "slots" in the KV cache more aggressively, processing many different sequences in parallel rather than queuing them in small, static batches.

The results demonstrated the engine's ability to maintain high throughput under high load by overlapping the computation of different requests. While the baseline configuration was limited by conservative sequence caps, the optimized vLLM settings utilized the L4 GPU's compute cores more effectively, ensuring that as soon as one token was generated for a request, its slot was immediately available for the next. This maximized the "utilization density" of the hardware, resulting in a significant speedup factor and proving that proper scheduler tuning is critical for scaling LLM applications to handle high-volume user traffic.

N-gram Speculative Decoding

To benchmark this, I constructed a prompt featuring a 4,000-token repetitive legal and technical manual. The technical "hoop" involved dynamically re-configuring the vLLM server via a JSON-based `--speculative-config` flag and ensuring a clean GPU state between runs using `torch.cuda.empty_cache()` and `fuser`. I specifically tuned the `num_speculative_tokens` to 10, a high value that would normally be risky but was effective here because the "ground truth" text is so predictable.
The results demonstrated a 6% speedup in tokens per second (TPS) compared to the vanilla baseline. This experiment illustrates a powerful optimization.

Draft Model Example

In this experiment, I evaluated a **Draft-and-Verify pipeline** by orchestrating a handoff between two models of different scales: a lightweight 0.5B parameter Draft model and a high-capacity 7B parameter Large model. This setup simulates the conceptual foundation of Speculative Decoding, where a small, inexpensive model rapidly generates a "draft" (a sci-fi hook) that is subsequently refined or expanded by the larger, more capable model, as described above.

The technical execution on a single L4 GPU required memory management and hardware-specific optimizations. To fit both models sequentially in the constrained VRAM environment, I implemented a `cleanup()` function that utilized `fuser` to force-kill GPU-attached processes and `torch.cuda.empty_cache()` to reset the memory state. Furthermore, I optimized the vLLM engine by setting `--kv-cache-dtype` to `fp8`. This reduced the memory footprint of the Key-Value cache, which, combined with strict `gpu-memory-utilization` caps (0.25 for the draft and 0.65 for the large model), allowed for a seamless transition between the two distinct inference server instances.

Results

Optimization Technique	Baseline Performance	Optimized Performance	Speedup / Improvement
Automatic Prefix Caching	1.18 Tokens/Sec (126.85s)	**1.30 Tokens/Sec** (115.36s)	**1.10x**
Concurrency Stress Test	Standard Batching	**Continuous Batching**	**1.09x**
N-Gram Speculative Decoding	21.21 Tokens/Sec	**22.45 Tokens/Sec**	**1.06x** (5.8%)