Large Language Model Distributed Inference

December 17, 2024

Objectives

By the end of this workshop, you will be able to:

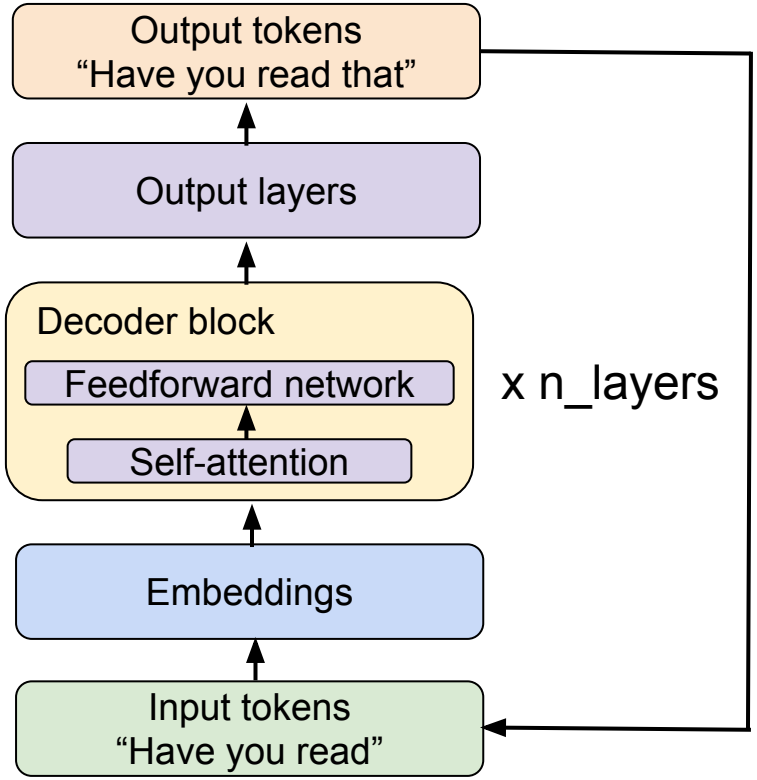
- Explain the basics of distributed computing for large language model inference
- Use vLLM, a popular for LLM inference, to host a server
- Prompt and extract logits from a large Llama model on an HPC cluster
- Use offline batch inference with a large Llama model

Agenda

1. Basics of distributed inference for LLMs
2. Using vLLM for Online Inference
   - Setting up an LLM server
   - Using the LLM server for inference
3. Using vLLM for Offline Inference

Llama Models

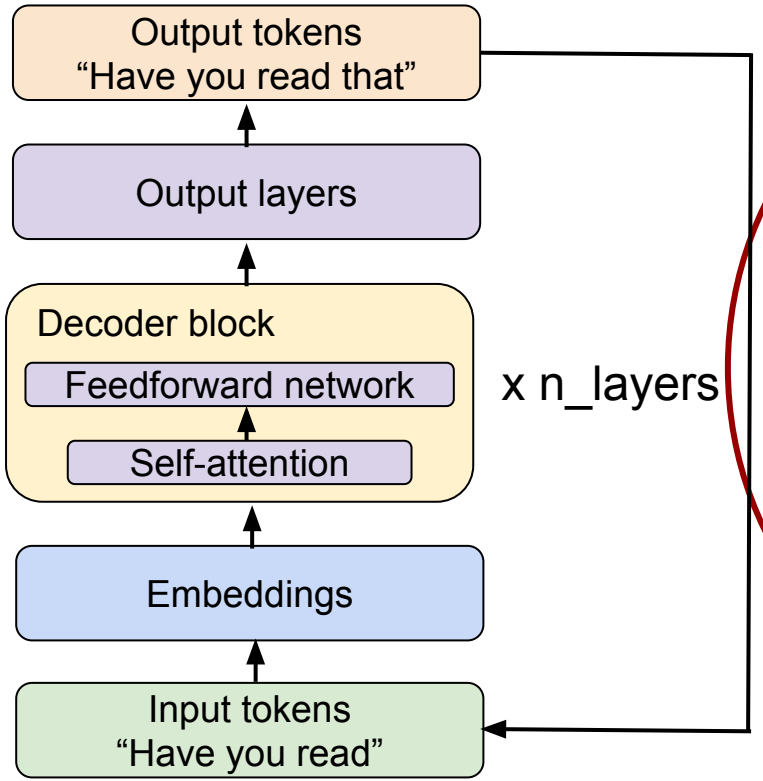
- Family of decoder-only transformer large language models
- Trained model weights released by Meta AI starting in February 2023
- Latest generation released in July 2024
  - Llama 3 models with 8B, 70B, or 405B parameters



Training vs Inference in Llama models

Training

Process of learning model weights by optimizing performance on tasks such as next-token prediction.



Inference

Process of using trained model weights to provide outputs based on given input prompts.

Exercise: Assessing Memory Needs

1) Compute the memory that the model weights of the Llama model with 405 billion weights will take. Assume weights are float16.

2) Will this fit on one GPU? If not, how many H100 GPUs would you need?

GPUs:

H100 ⇒ 80 G
A100 ⇒ 40 G

Large Memory Needs for Large Models

Memory Needed for Model Weights

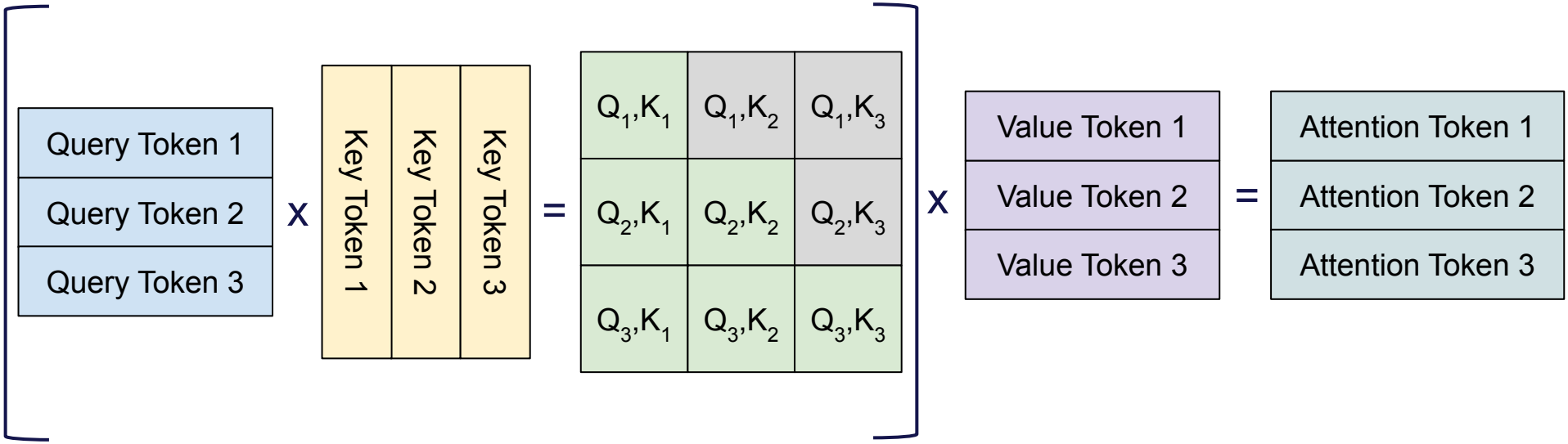
Model Size	Float16	Float8	INT4
8B	16 GB	8 GB	4 GB
70B	140 GB	70 GB	35 GB
405B	810 GB	405 GB	203 GB

Adapted from https://huggingface.co/blog/llama31

KV Caching

- Storing key and values for previous tokens to speed up inference
- Used in decoder architectures when generating multiple tokens

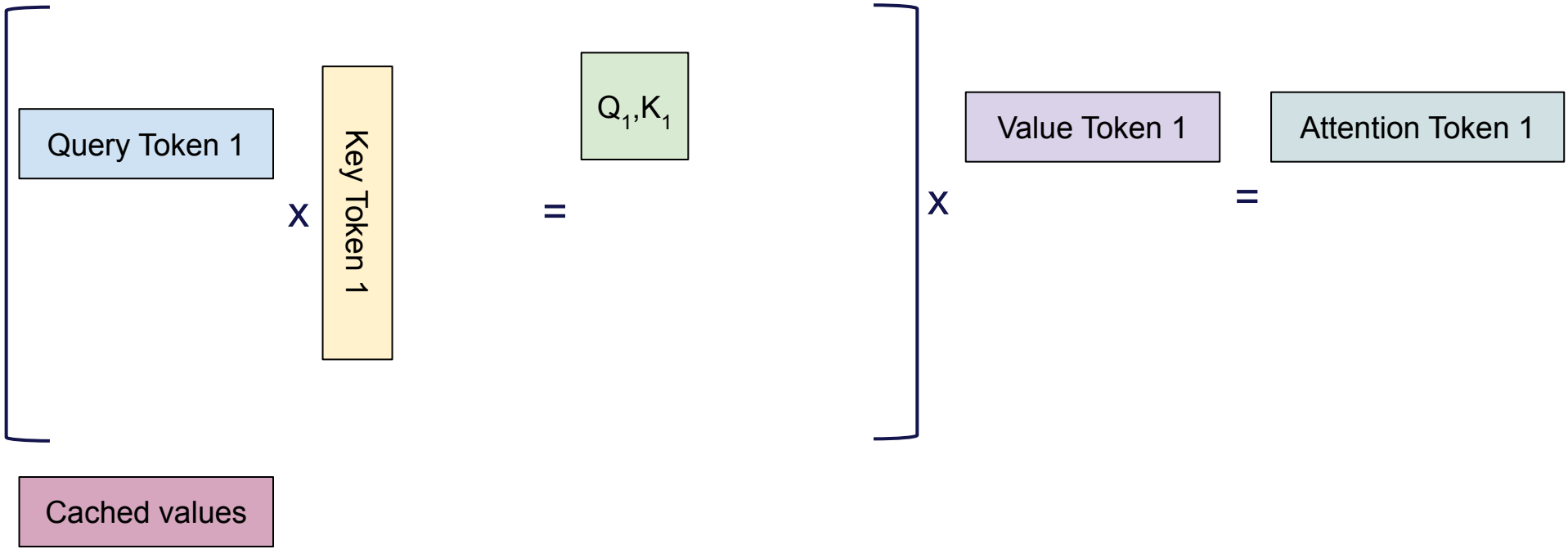
Self-Attention



We keep re-computing earlier key and value tokens!

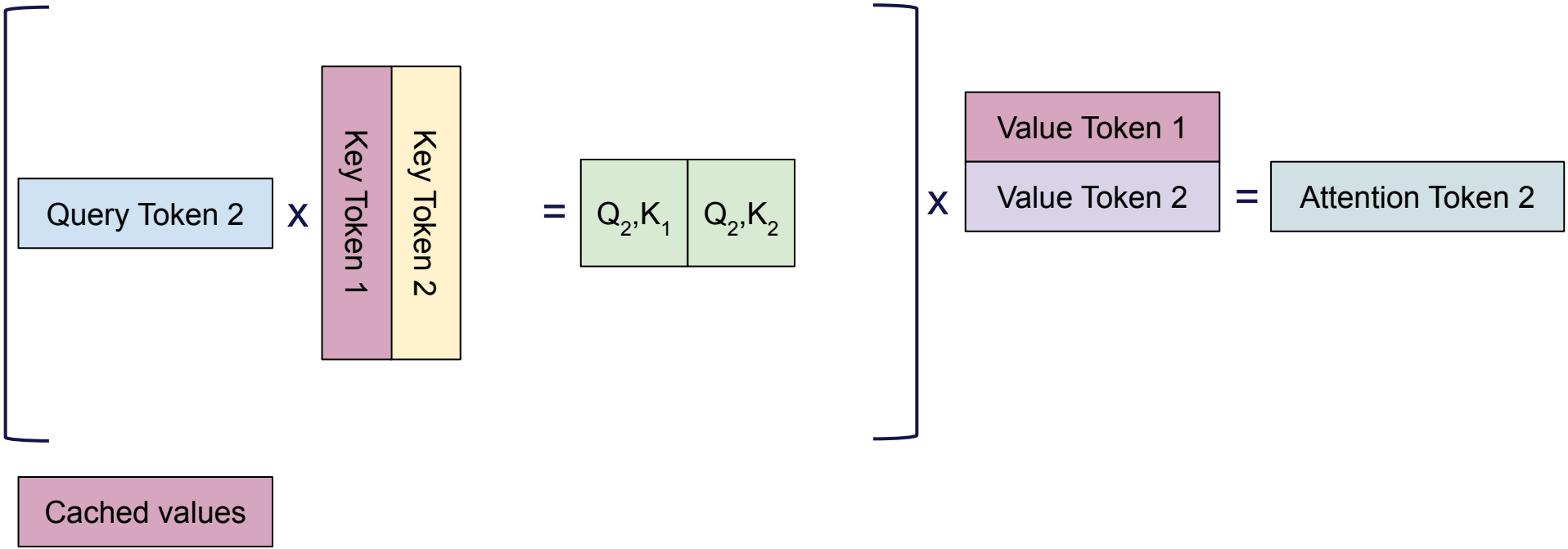
Adapted from https://medium.com/@joaolages/kv-caching-explained-276520203249

Self-Attention with KV-Caching



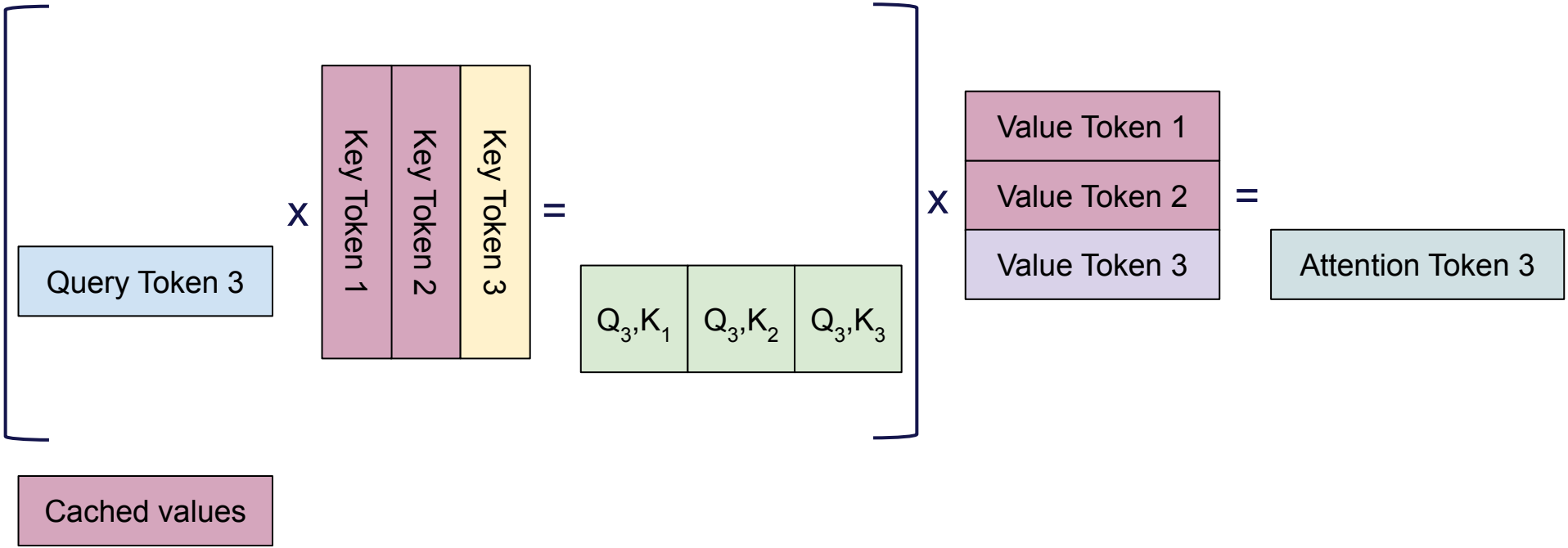
Adapted from https://medium.com/@joaolages/kv-caching-explained-276520203249

Self-Attention with KV-Caching



Adapted from https://medium.com/@joaolages/kv-caching-explained-276520203249

Self-Attention with KV-Caching



Adapted from https://medium.com/@joaolages/kv-caching-explained-276520203249

KV Caching

- Storing key and values for previous tokens to speed up inference
- Used in decoder architectures when generating multiple tokens

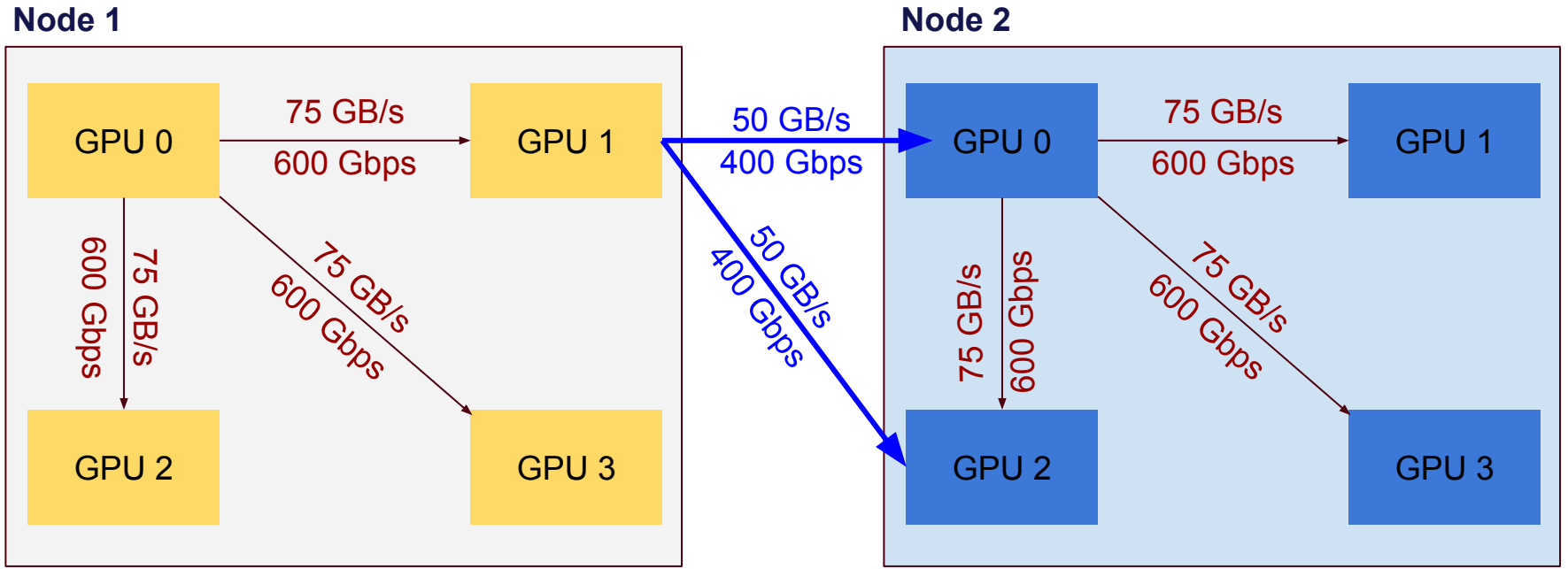
- Pro: small matrices = faster matrix multiplication = faster inference
- Con: need more GPU memory to cache key and value states

Large Memory Needs for Large Models

Model Size	Memory for Float16 Weights	Memory for KV Cache for 128k token request	Number of H100s needed for both
8B	16 GB	15.62 GB	1
70B	140 GB	39.06 GB	2.24
405B	810 GB	123.05 GB	11.66

Adapted from https://huggingface.co/blog/llama31

GPU-to-GPU Communication

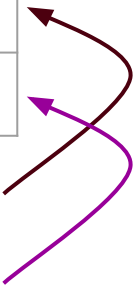


Inside Node (**NVLINK**): Each GPU talks to other three GPUs at 75 GB/s (single direction). This sums up to 900 GB/s all GPU-GPU bidirectional speed. 75 GB/s * 6 * 2 = 900 GB/s

Outside Node (**InfiniBand Network NDR**): Each GPU communicates to other GPUs in another node at 400 Gbps (50 GB/s).

Large Memory Needs for Large Models

Model Size	Memory for Float16 Weights	Memory for KV Cache for 128k token request	Number of H100s needed for both
8B	16 GB	15.62 GB	1
70B	140 GB	39.06 GB	2.24
405B	810 GB	123.05 GB	11.66



Should be on a single node

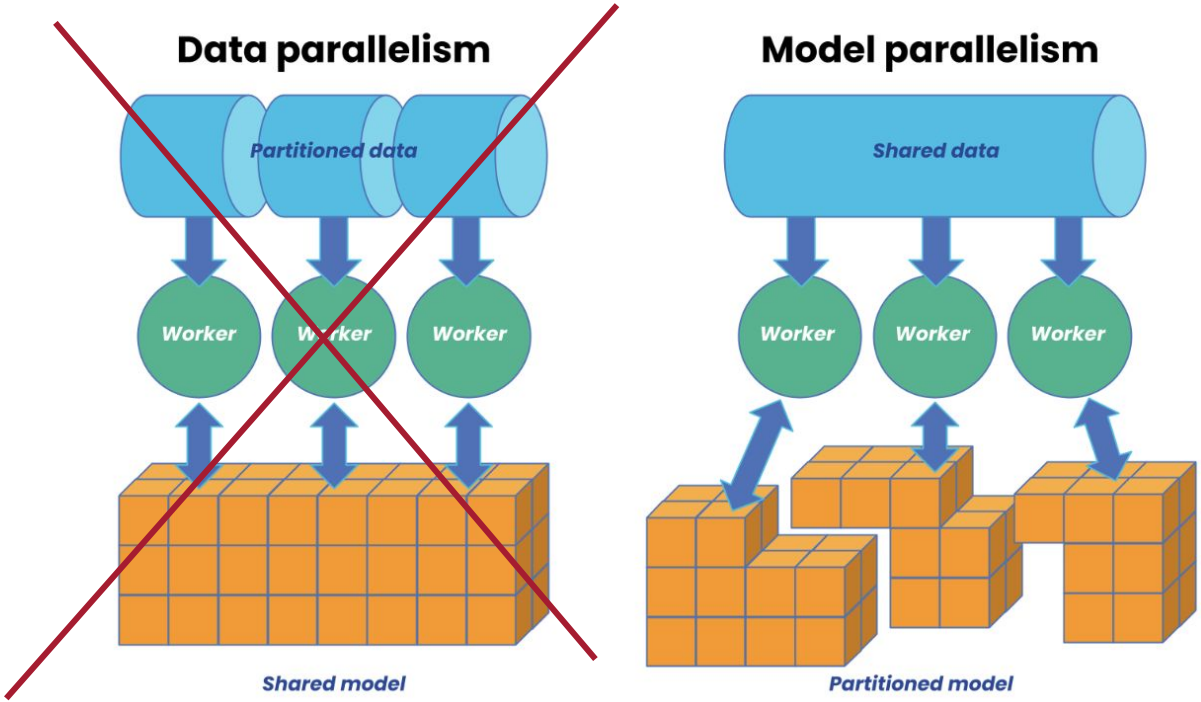
Needs multiple nodes

Adapted from https://huggingface.co/blog/llama31

Virtual Large Language Model (vLLM)

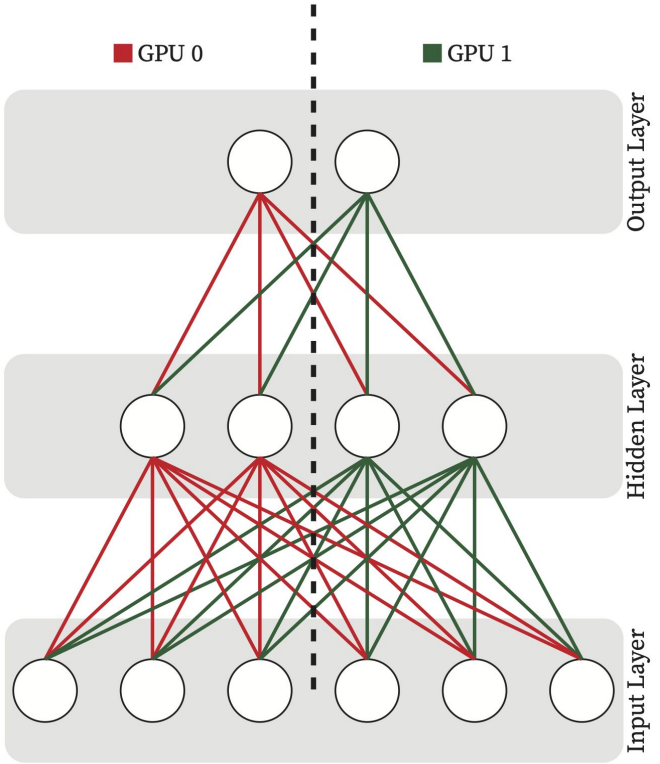
- Open source library that supports fast and efficient inference and serving of LLMs
- Has built-in support for distributed inference through tensor and pipeline parallelism
- We'll use it with Llama models but supports many other models (will discuss this further later)

Types of Parallelism



https://www.anyscale.com/blog/what-is-distributed-training

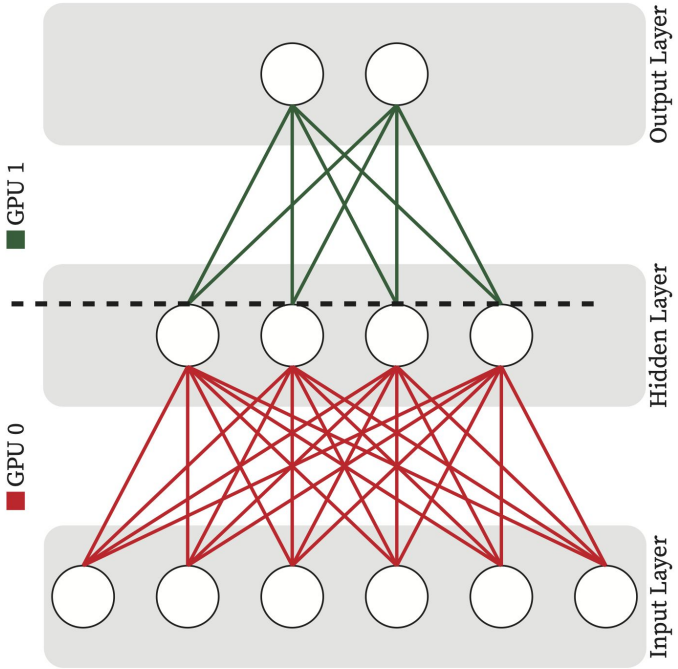
Single-Node Multi-GPU: Tensor Parallel Inference



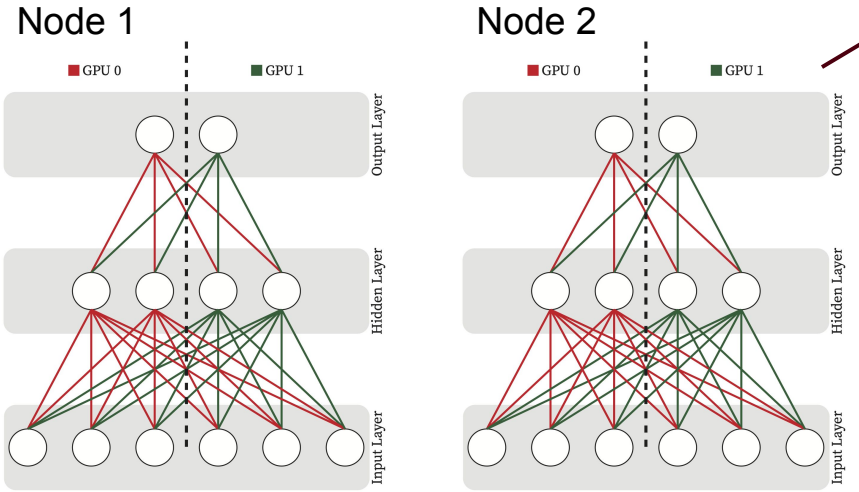
- All GPUs contribute in each layer computation
- Removes the GPU Idle time

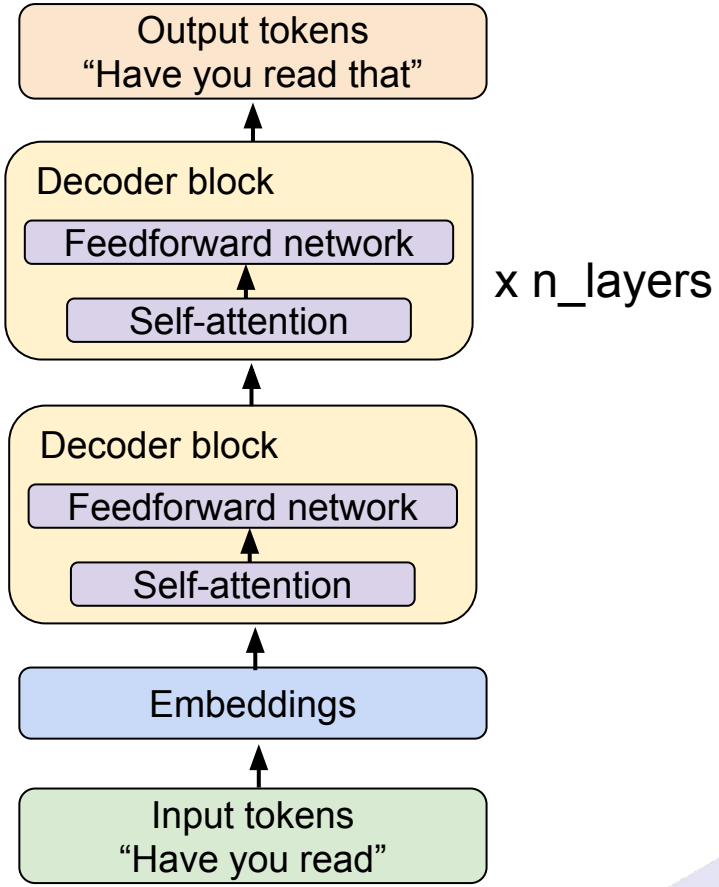
https://docs.vllm.ai/en/latest/serving/distributed_serving.html

Pipeline Parallel Inference



Multi-Node Multi-GPU: Tensor Parallel & Pipeline Parallel Inference





vLLM Recommendations

Tensor parallel size = # of GPUs per node

Pipeline parallel inference size = number of nodes

However on our cluster, communication between nodes is fast enough to support only tensor parallelism with vLLM even for multi-node multi-GPU model hosting

Behind the Scenes of vLLM Distributed Inference

- vLLM uses Megatron-LM's tensor parallel algorithm
- Manages distributed multi-node inference scheduling using Ray:
  - Ray is an open-source framework for scaling AI models

Speed of Inference

Using provided vLLM configs:

70B model: ~50 tokens per second (early in sequence, single prompt)

405B model: ~18-20 tokens per second (early in sequence, single prompt)

Latency: time it takes for a single input to produce a single output

Throughput: rate of output (tokens per second)

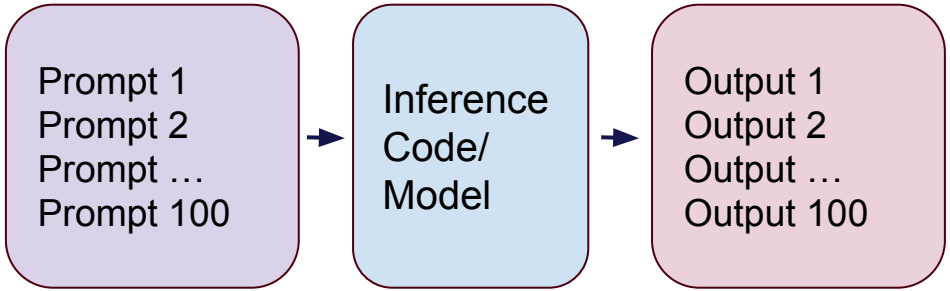
Could be increased by parallelizing prompts rather than doing them one-at-a-time

Agenda

1. Basics of distributed inference for LLMs
2. Using vLLM for Online Inference
   - Setting up an LLM server
   - Using the LLM server for inference
3. Using vLLM for Offline Inference

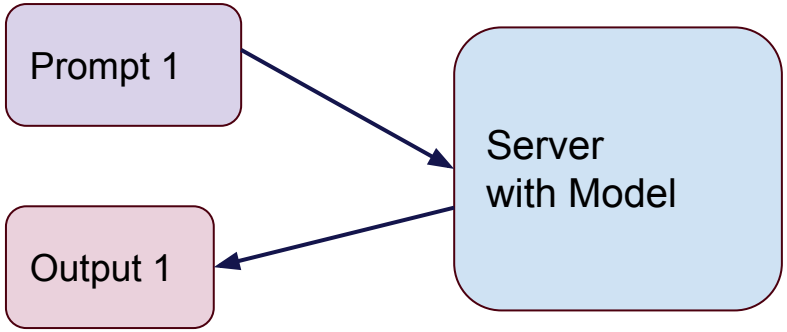
Offline/Batch Inference

- Processing large dataset all at once
- Similar to typical Pytorch module/inference for other types of models
- Asynchronous/no interactivity
- Can be optimized to maximize throughput (high volume of prompts)



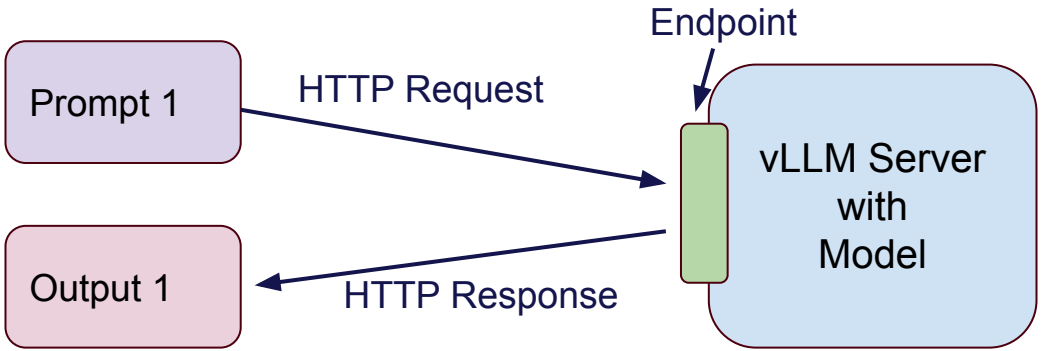
Online Inference

- "Real-time" interactive inference of single or small batch at a time
- Similar to ChatGPT/Claude
- Can be optimized to minimize latency (time for single prompt)



vLLM Server

- Server: computer or software program that provides services to other computers (called clients). Acts as a central point to fulfill requests from clients and serve data/applications.
- Server endpoint: specific location/address on a network that serves as entry/exit point for communication
- In vLLM case, we're going to persistently host an LLM model on multiple GPUs
- We can then send prompt requests to this server and get the outputs back.
- This server implements OpenAI API protocol



Exercise: Setting up your LLM server

1) Log into the FASRC cluster
2) Clone Kempner vLLM github repo and cd in

   git clone https://github.com/KempnerInstitute/distributed-inference-vllm.git

3) Find relevant SLURM script: **examples/workshops/70B_slurm.sh**. Update output and error file paths
4) Run the SLURM script. We'll dive into what it's doing next!

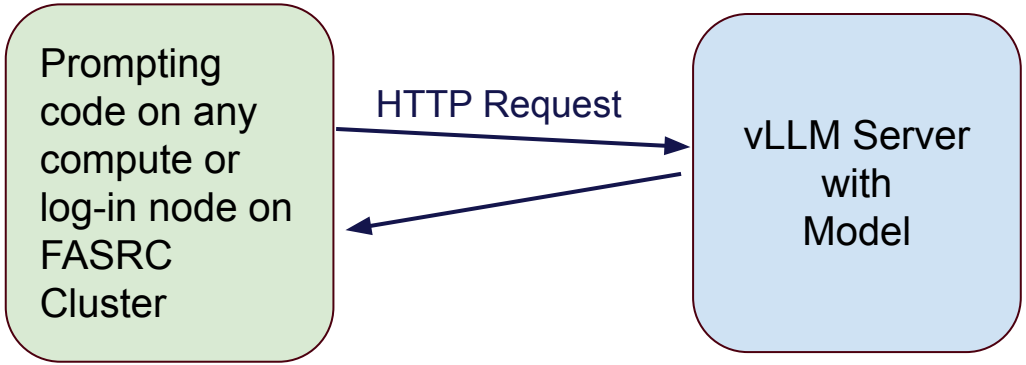
   sbatch examples/workshops/70b_slurm.sh

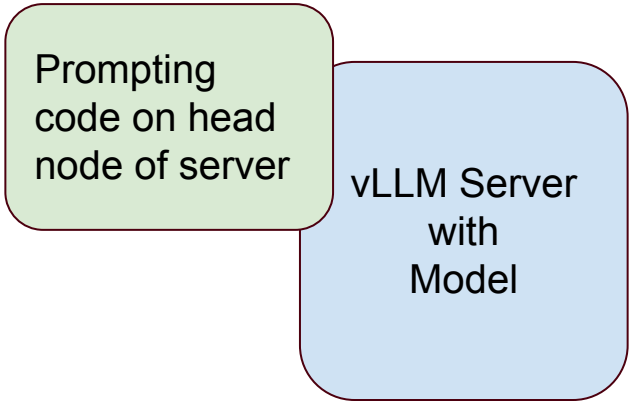
5) Check it's running (**sacct**) (might take a few minutes to show). Look at error log periodically. Should eventually see something like:

```
Loading safetensors checkpoint shards:   0% Completed | 0/191 [00:00<?, ?it/s]

Loading safetensors checkpoint shards:   1% Completed | 1/191 [00:04<13:59,  4.42s/it]
```

Interacting with server





Exercise: Getting set up for inference

We'll be working through the tutorial in the file examples/workshops/inference.ipynb

Need to set up to use this jupyter notebook on the cluster:

- Recommended:
  - Use OpenOnDemand. You do not need GPUs so use partition test, a single cpu, and 4 GBs of memory. No need to specify modules or environment
- If you're familiar with VSCode remote dev:
  - Option 1: Find head GPU of LLM server in your output logs. Can use this as compute node
  - Option 2: Start a different interactive session using partition test, a single cpu, and 4 GBs of memory

vLLM Output Format

Dictionary with these keys:

- id: request id (defined by the server)
- object: type of request (text_completion in our case)
- created: timestamp of request
- model: model id (path of the model)
- choices: generated output of the model
- usage: statistics about input and output tokens

Exercise: Inference for a single prompt

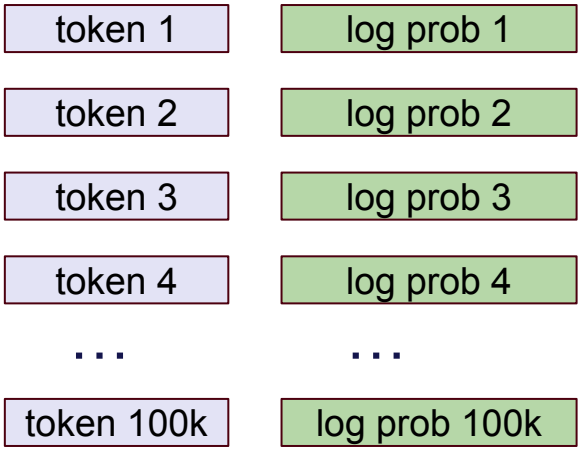
1) Fill in the IP of the node hosting the server: this is the head node ip in the output logs of the SLURM job. You can use localhost if your tutorial notebook is running on the relevant compute node.
2) Run the code and look at the outputs.
3) Change the max_tokens and min_tokens values and rerun the code.
4) Change the temperature value. Run the code a couple times for each temperature parameter you try. What do you notice about the output text when temperature is 0 vs higher?
5) Change the frequency_penalty value. It can take any value between [-2, 2]. Can you figure out what this parameter does?
6) Change the top_k value and have temperature = 1. How do the outputs change?

Sampling Parameters

max_tokens & min_tokens denote the maximum and minimum possible tokens generated. The LLM might generate fewer tokens than the max

top_k: only consider top k log probability tokens when samping

frequency_penalty: penalizes repetition of tokens within generation



Temperature of 0 -> deterministic, just take top log prob

Otherwise, divide log probs by temperature before sampling. Higher temperature squishes probabilities together so more random tokens generated

vLLM Output (Choices)

index: the index in the choices array

text: generated text

finish_reason: why the generation stopped

stop_reason: we do not use this, related to another vLLM sampling parameter

logprobs: (requires parameter) log probabilities of each token in the generated output

prompt_logprobs: (requires parameter) log probabilities of each token in the prompt

vLLM Log Probabilities

vLLM has two sampling parameters (logprobs and prompt_logprobs) for computing log probabilities

- logprobs refers to the generated output tokens
- prompt_logprobs refers to the input tokens

These parameters are non-negative integers

If the parameter is set to k, then you get the log probabilities of the generated/input token, as well as the k most probable options for that token

vLLM Log Probabilities

logprobs=1


```
“logprobs”: {
  "top_logprobs":[
    {
      "20":-4.005917549133301,
      "49":-1.7559176683425903
    },
    {
      "-minute":-1.359967827796936
    }
  ]
}
```


prompt_logprobs=1


```
“prompt_logprobs”: [
  {
    "24661":{
      "logprob":-8.533774375915527,
      "rank":284,
      "decoded_token":"San"
    },
    "14924":{
      "logprob":-1.1587743759155273,
      "rank":1,
      "decoded_token":"Question"
    }
  },
]
```

vLLM Log Probabilities

logprobs=1

```
“logprobs”: {
  "top_logprobs":[
    {
      "20":-4.005917549133301,
      "49":-1.7559176683425903
    },
    {
      "-minute":-1.359967827796936
    }
  ]
}
```

“20” is generated,
“49” is most probable

prompt_logprobs=1

```
“prompt_logprobs”: [
  {
    "24661":{
      "logprob":-8.533774375915527,
      "rank":284,
      "decoded_token":"San"
    },
    "14924":{
      "logprob":-1.1587743759155273,
      "rank":1,
      "decoded_token":"Question"
    }
  },
]
```

vLLM Log Probabilities

logprobs=1

```
“logprobs”: {
  "top_logprobs":[
    {
      "20":-4.005917549133301,
      "49":-1.7559176683425903
    },
    {
      "-minute":-1.359967827796936
    }
  ]
}
```

“-minute” is most probable and generated

prompt_logprobs=1

```
“prompt_logprobs”: [
  {
    "24661":{
      "logprob":-8.533774375915527,
      "rank":284,
      "decoded_token":"San"
    },
    "14924":{
      "logprob":-1.1587743759155273,
      "rank":1,
      "decoded_token":"Question"
    }
  },
]
```

vLLM Log Probabilities

logprobs=1

```
“logprobs”: {
  "top_logprobs":[
    {
      "20":-4.005917549133301,
      "49":-1.7559176683425903
    },
    {
      "-minute":-1.359967827796936
    }
  ]
}
```

prompt_logprobs=1

```
“prompt_logprobs”: [
 {
  "24661":{
    "logprob":-8.533774375915527,
    "rank":284,
    "decoded_token":"San"
  },
  "14924":{
    "logprob":-1.1587743759155273,
    "rank":1,
    "decoded_token":"Question"
  }
 },
]
```

“San” is the first token in the prompt,
“Question is the most probable first token

Multithreading

Allows sending multiple request to the server in parallel

Performant even when handling prompts with varying completion times

- Use a job queue (ThreadPoolExecutor) so that as one request finishes, a new one is sent

Agenda

1. Basics of distributed inference for LLMs
2. Using vLLM for Online Inference
   - Setting up an LLM server
   - Using the LLM server for inference
3. Using vLLM for Offline Inference

Other Models

https://docs.vllm.ai/en/v0.6.2/models/supported_models.html

`JambaForCausalLM`	Jamba	`ai21labs/Jamba-v0.1`, etc.
`LlamaForCausalLM`	Llama 3.1, Llama 3, Llama 2, LLaMA, Yi	`meta-llama/Meta-Llama-3.1-405B-Instruct`, `meta-llama/Meta-Llama-3.1-70B`, `meta-llama/Meta-Llama-3-70B-Instruct`, `meta-llama/Llama-2-70b-hf`, `01-ai/Yi-34B`, etc.
`MiniCPMForCausalLM`	MiniCPM	`openbmb/MiniCPM-2B-sft-bf16`, `openbmb/MiniCPM-2B-dpo-bf16`, etc.
`MiniCPM3ForCausalLM`	MiniCPM3	`openbmb/MiniCPM3-4B`, etc.
`MistralForCausalLM`	Mistral, Mistral-Instruct	`mistralai/Mistral-7B-v0.1`, `mistralai/Mistral-7B-Instruct-v0.1`, etc.
`MixtralForCausalLM`	Mixtral-8x7B, Mixtral-8x7B-Instruct	`mistralai/Mixtral-8x7B-v0.1`, `mistralai/Mixtral-8x7B-Instruct-v0.1`, `mistral-community/Mixtral-8x22B-v0.1`, etc.

Thank you