

# Deploying Foundational Models: Challenges and Best Practices

Ayush Maheshwari

Sr. Solutions Architect, NVIDIA

Interested in research collaboration !

Fill this form - <https://tinyurl.com/collab-nv>





# Sessions

1. Cluster health-check using NCCL, MLPerf, HPL **(1 hour) - Completed**
  - a) Understand the hardware and its performance on multiple GPUs.
  - b) Ensure that your training performance aligns with the h/w benchmarks
  - c) Evaluate the cluster to ensure platform fits within your needs.
2. Large scale data curation for LLM training **(1 hour) - Completed**
  - a) Deep-dive into aspects of data curation
  - b) Mixed-precision training
3. Distributed and stable LLM training on a large-scale cluster **(1.5 hour) - Completed**
  - a) Parallelism techniques
  - b) Frameworks and wrappers
  - c) Recipes and best practices
4. Post-training and evaluation of pre-trained LLM **(1.5 hour) - Today**
  - a) Sync between training data and expected performance
  - b) Algorithms and frameworks
5. Fine-tuning and deployment **(1 hour)**
  - a) Dynamic and static batching, state management, inference server
  - b) Best practices for optimizing model



# Agenda

1. Basics of Inference
2. Inference Metrics
3. Inference in stages: Prefill and Decoding
4. Engines
5. Runtime Optimizations
6. Inference Serving
7. How to calculate GPU inference Budget



# Inference Compute Requirements Scaling Exponentially

Fueled by reasoning models and AI agents



## Larger Models

Hundreds of billions of parameters



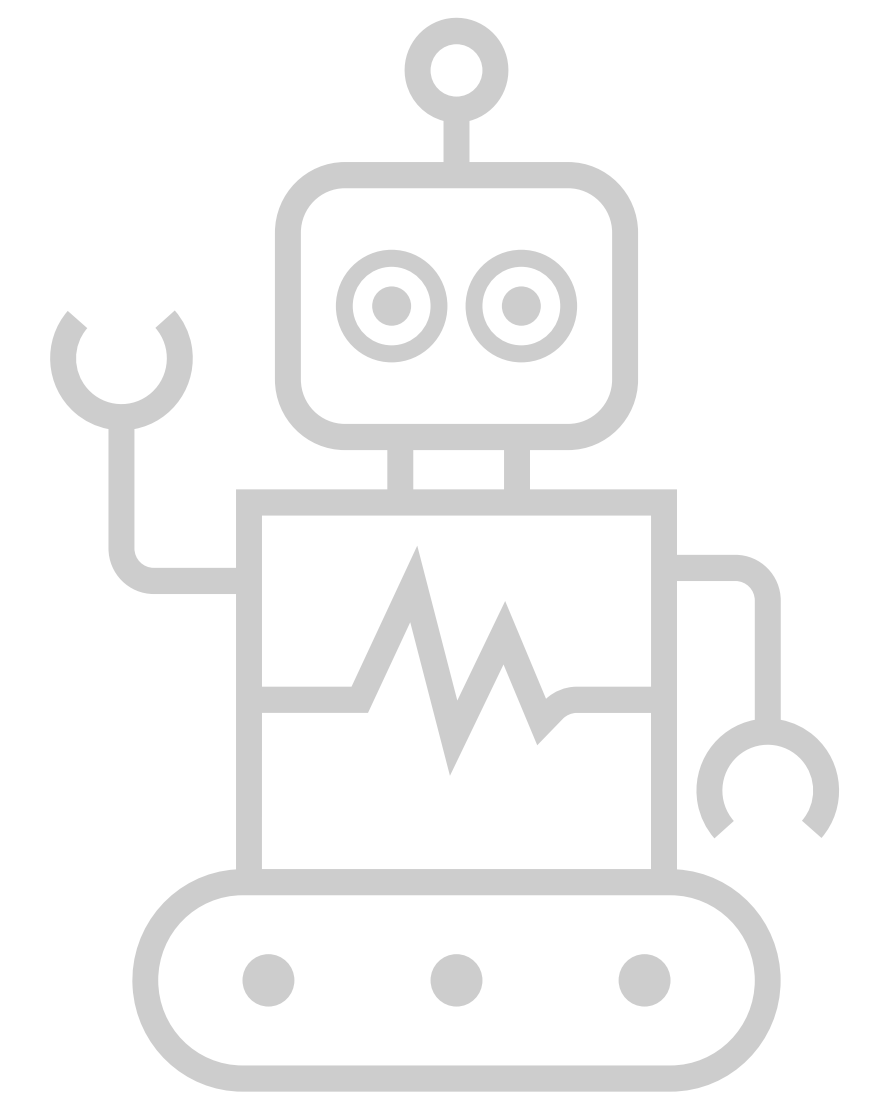
## Long Thinking Time

100x more thinking tokens



## Larger Context

Millions of input tokens



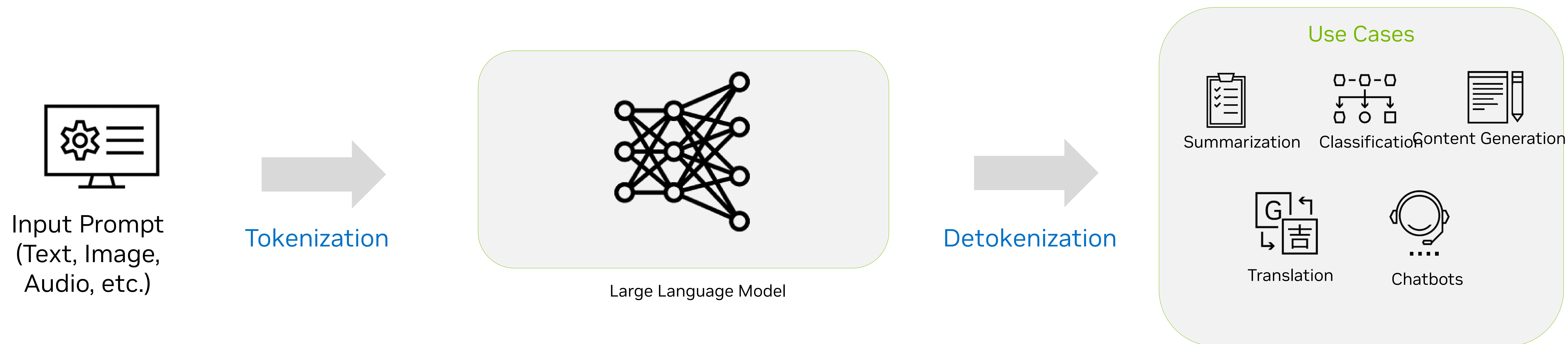
## Agents

One user prompt involves multiple model executions

# What is LLM Inference

The process of getting an output from a model

1. During inference, an AI model receives a prompt and translates into a series of tokens
2. The model processes input tokens to learn the relationships between them
3. It generates its response as tokens and then translates it to the user's expected format



\*Tokenization and detokenization extend beyond text, to multiple modalities such as video, image, audio, gene sequence, etc.

# Understanding Tokens

## The Language and Currency of AI

### Tokens

- Tiny units of data that come from breaking down bigger chunks of information
  - Text:  $\pm$  4 chars in length
  - Images: Group of pixels
- AI models learn by guessing the next token in the sentence (next-token prediction).
- After training, the model can perform inference to generate tokens and produce responses.

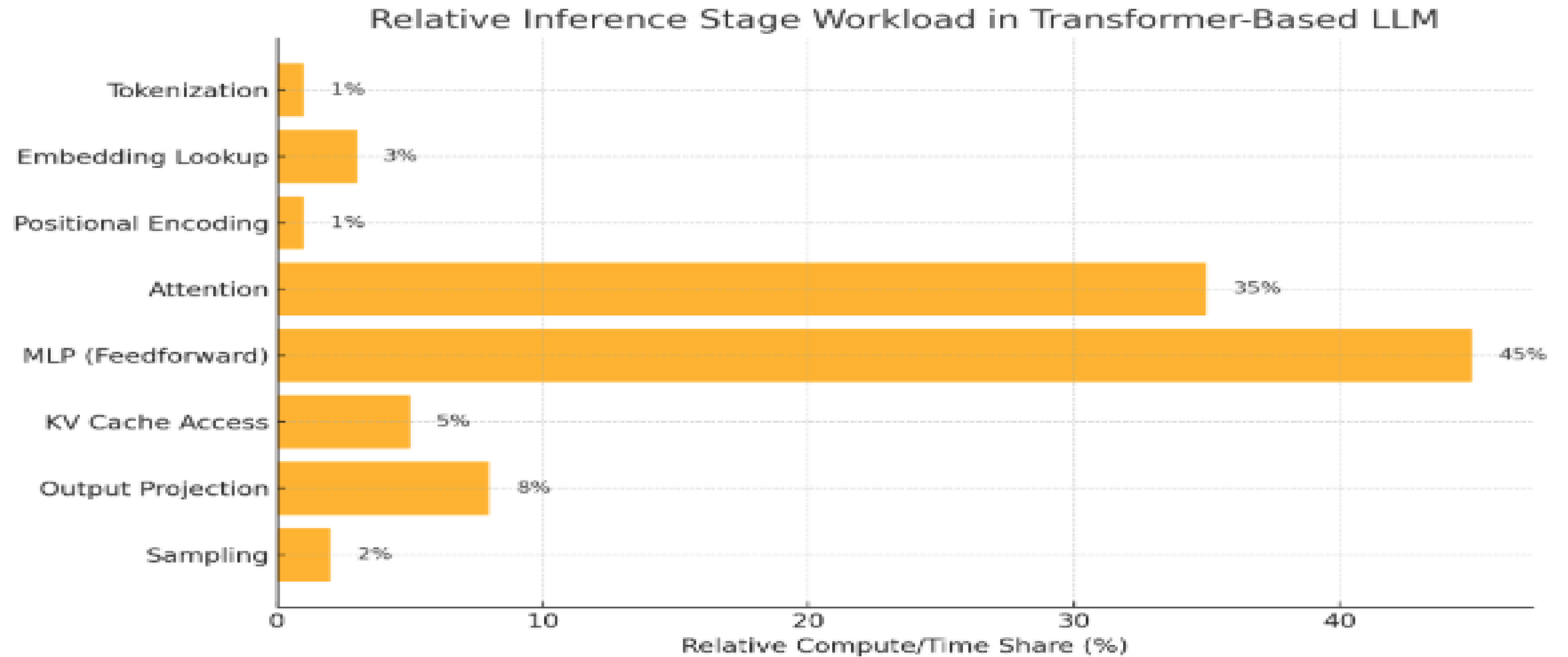
Tokenization breaks a word into  
tokens of about 4 characters

### Value of Tokens

- The faster tokens can be processed, the faster models can learn and respond
- Many AI services measure the value of their products based on the number of tokens consumed and generated
- Many pricing plans are based on a model's rates of token input and output

# Multiple stages

**Relative Inference Stage Workload In Transformer-Based LLM**



- Each stage has a different compute/memory requirement mixture



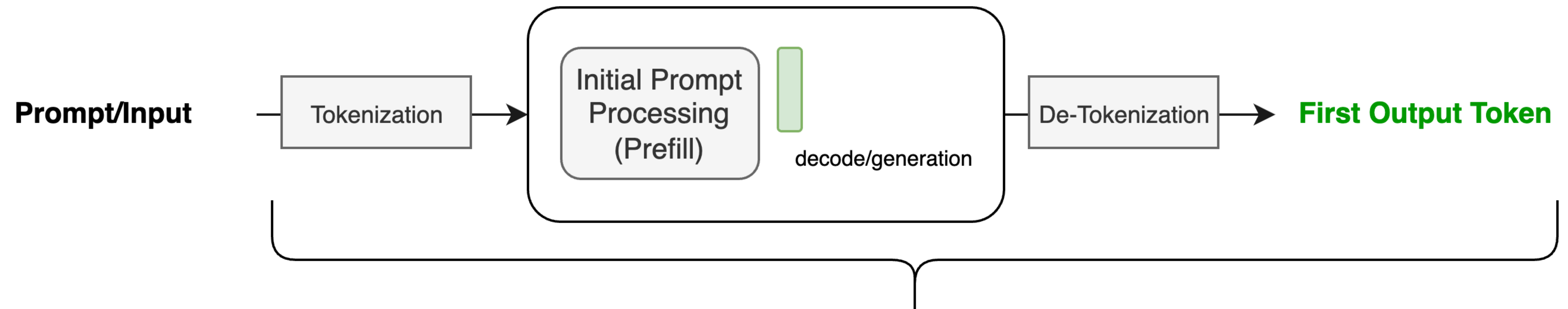


# Metrics



# LLM Inference Measurement – Time to First Token

The time it takes to process the input and generate the first token



## Time to First Token (TTFT)

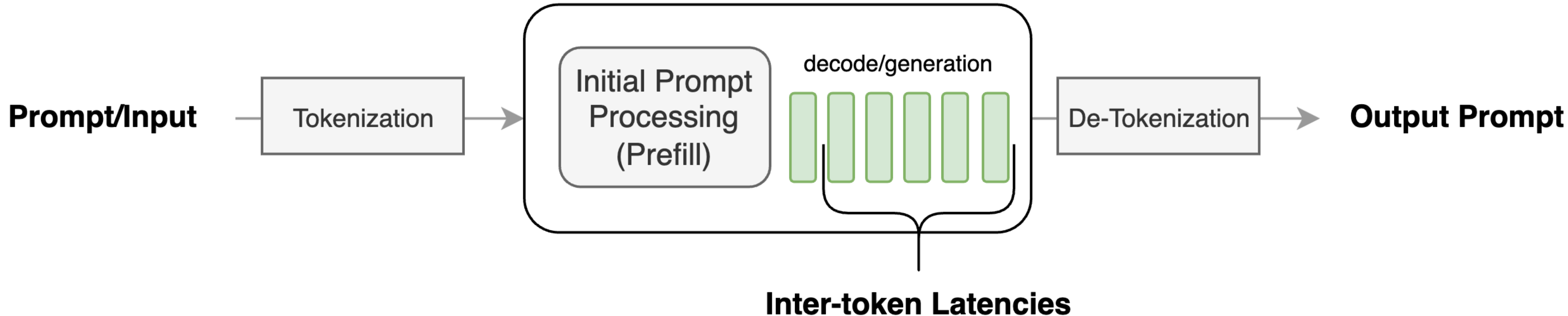
Signifies time it takes to process the prompt and generate the first token

Synonyms:

- First Token Latency (FTL)

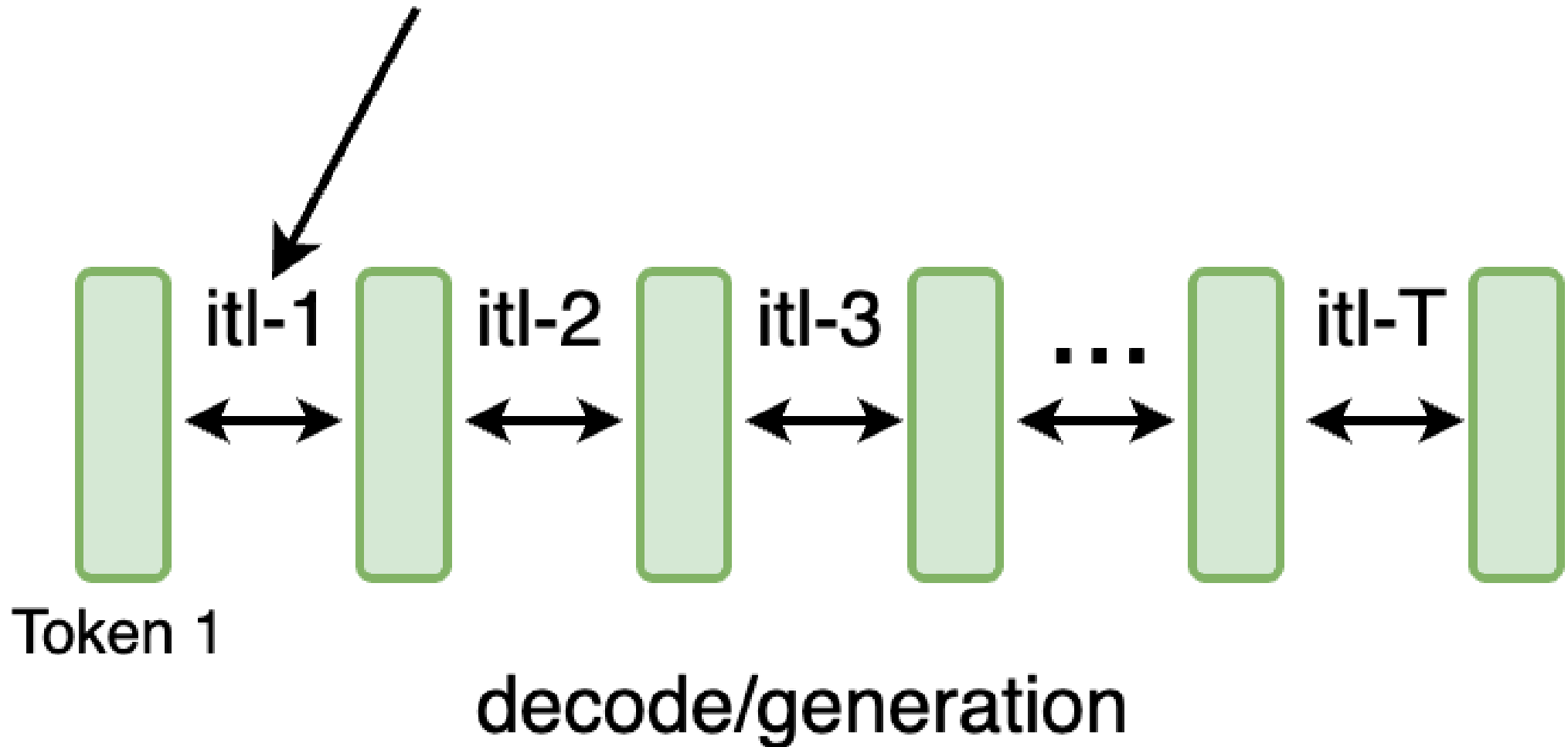
# LLM Inference Measurement – Inter-token Latency

Measuring generation performance when the system is under load



**Inter-token Latencies** - Should be near constant over all token generations.

Near constant time implies efficient generation



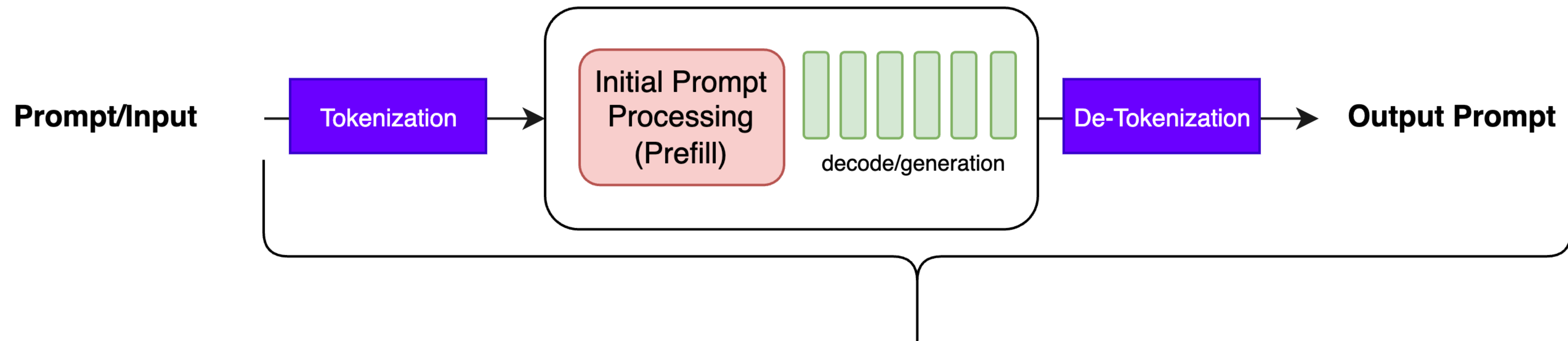
Synonyms:

- Time Per Output Token (TPOT)



# LLM Inference Measurement – Total Time to Generation

The time it takes to process a prompt and generate all the tokens



## Total Time to Generation

Measures to total time to process input and generate all tokens

Synonyms:

- End-to-End Latency (E2E latency)
- Time to last Token

# Throughput Metrics

- Always specify
  - Model
  - Precision
  - Input Length
  - Output Length
  - Concurrency
  - TP
- The most unambiguous metric
  - to measure is **requests/second/instance**
  - to use in sizing **requests/second/GPU**



# Inference Latency

## Tokens

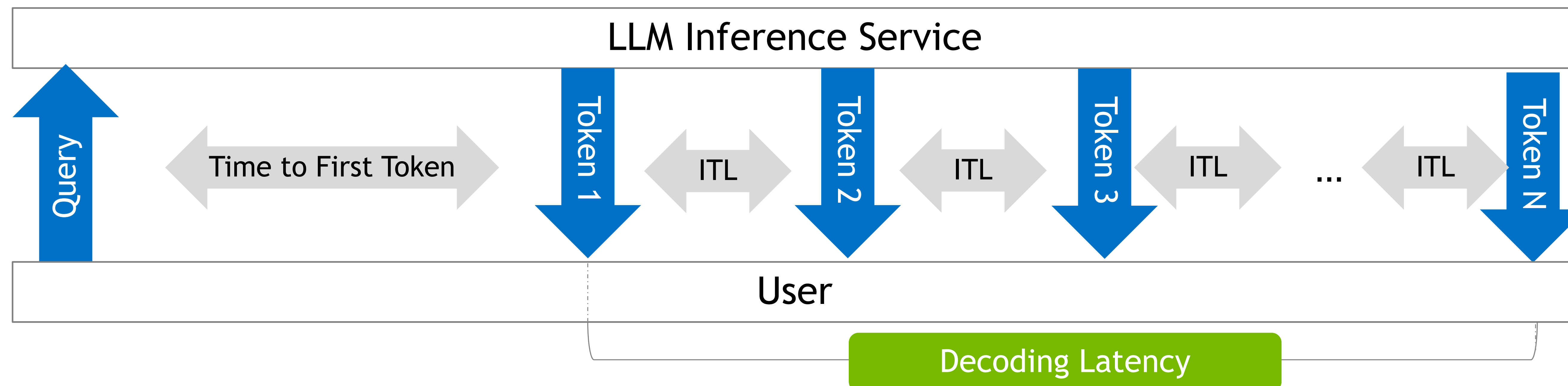
- Input sequence length of **tokens (ISL)** are fed into a model.
- LLM generates of **output sequence length (OSL)** of **tokens** one at a time

## Time to First Token (TTFT)

- Time it takes for the model to generate the first token after receiving a request
- This part of execution is called **Prefill**
- Critical for chat-like applications

## End-to-end Latency (E2E)

- Time it takes for a model to generate a complete response to a user's prompt.
- E2E Latency = TTFT + **Decoding** Latency
- Inter-token Latency (**ITL**) is an average time between output tokens
- Critical, when full response has to be processed further: guardrails, tool calling

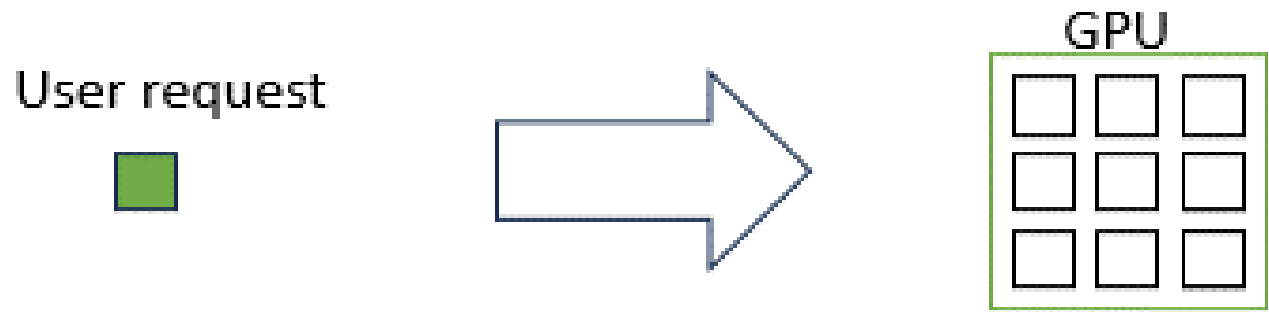


# Two Stages of LLM Execution

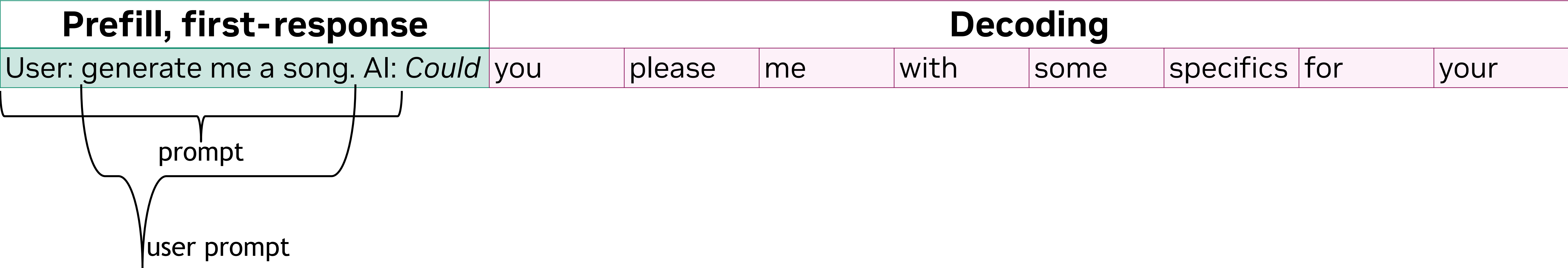
## Prefill vs Decoding

- **Prefill** = time to first token (~word)
  - Loading the user prompt into the system
  - From the request reception to the first token
  - Depends only on the number of input tokens
  - Populate KV-cache for all the tokens from the prompt.
  - Compute-bound for most of the reasonable prompt lengths
- **Decoding** = inter-token latency
  - Generating the response token by token, word by word
  - Inter-token latency depends on the total token number, both input and output tokens.
  - Usually memory-bound

### GPU Utilization Prefill vs Decode Phase



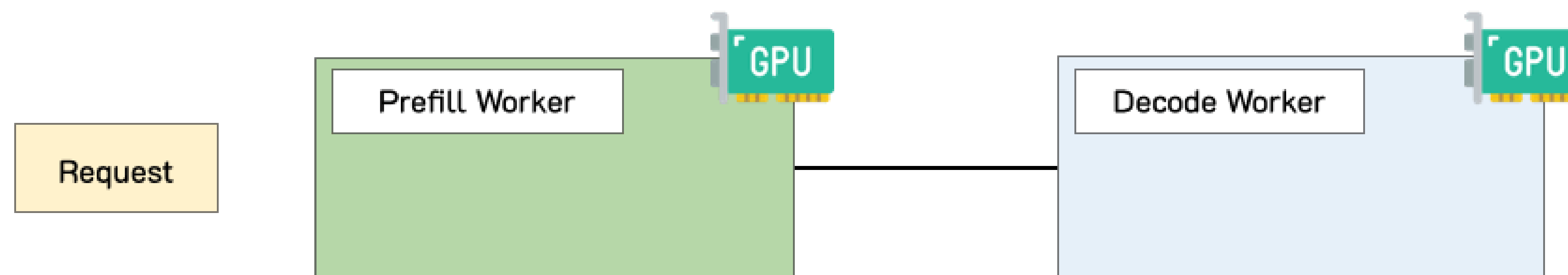
GPU Resource Utilization:  High Utilization  Low Utilization





Disaggregation is a technique that

## Request Arrived

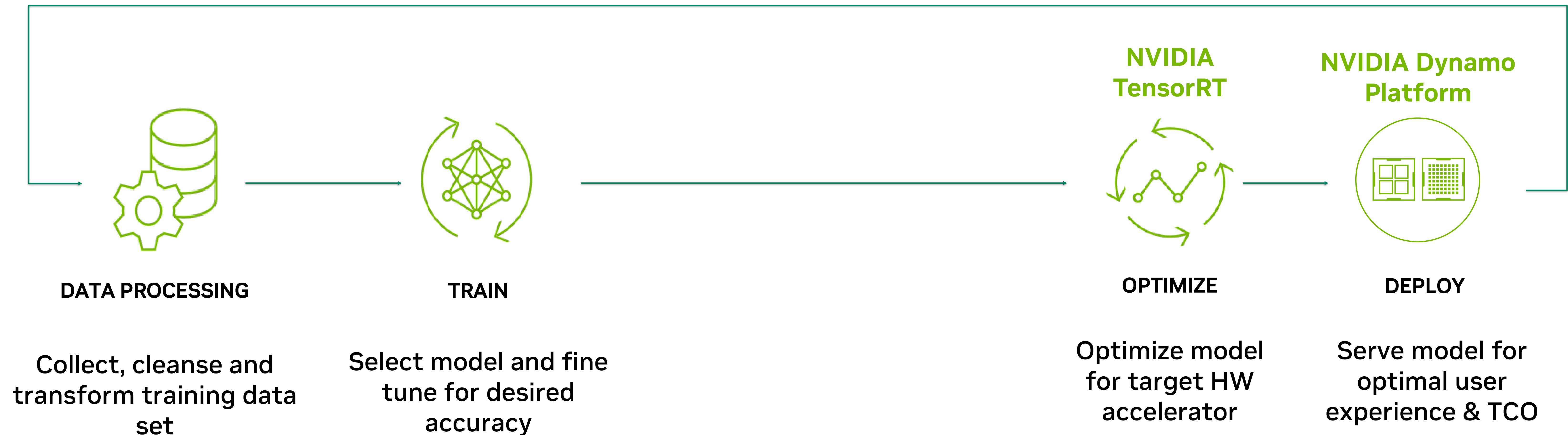


Timeline

# Inferencing in the End-to-end AI Workstream

## AI Training

## AI Inference







# GenAI-Perf to Benchmark


Using GenAI-Perf to Benchmark x +

docs.nvidia.com/nim/benchmarking/llm/latest/step-by-step.html

🔍 ☆ 📄 ⬇️ 👤 ⋮

 **NVIDIA** DOCS HUB 

## NIM for LLM Benchmarking Guide

Search NIM for LLM Benchmarking Guide 

### Topics

▼ Parameters and Best Practices

^ Using GenAI-Perf to Benchmark

Step 1. Setting Up an OpenAI-Compatible LLama-3 Inference Service with NVIDIA NIM

Step 2. Setting Up GenAI-Perf and Warming Up: Benchmarking a Single Use Case

Step 3. Sweeping through a Number of Use Cases

Step 4. Analyzing the Output

Step 5. Interpreting the Results


NVIDIA Docs Hub > NVIDIA NIM > NIM for LLM Benchmarking Guide > Using GenAI-Perf to Benchmark

NIM for LLM Benchmarking Guide |

## Using GenAI-Perf to Benchmark

NVIDIA **GenAI-Perf** is a client-side LLM-focused benchmarking tool, providing key metrics such as TTFT, ITL, TPS, RPS and more. It supports any LLM inference service conforming to the OpenAI API **specification**, a widely accepted de facto standard in the industry. This section includes a step-by-step walkthrough, using GenAI-Perf to benchmark a Llama-3 model inference engine, powered by NVIDIA NIM.

### Step 1. Setting Up an OpenAI-Compatible LLama-3 Inference Service with NVIDIA NIM

 Feedback

# GenAI-Perf command

Sample output generated by GenAI-Perf

```
export INPUT_SEQUENCE_LENGTH=200
export INPUT_SEQUENCE_STD=10
export OUTPUT_SEQUENCE_LENGTH=200
export CONCURRENCY=10
export MODEL=meta/llama3-8b-instruct

cd /workdir
genai-perf \
  -m $MODEL \
  --endpoint-type chat \
  --service-kind openai \
  --streaming \
  -u localhost:8000 \
  --synthetic-input-tokens-mean $INPUT_SEQUENCE_LENGTH \
  --synthetic-input-tokens-stddev $INPUT_SEQUENCE_STD \
  --concurrency $CONCURRENCY \
  --output-tokens-mean $OUTPUT_SEQUENCE_LENGTH \
  --extra-inputs max_tokens:$OUTPUT_SEQUENCE_LENGTH \
  --extra-inputs min_tokens:$OUTPUT_SEQUENCE_LENGTH \
  --extra-inputs ignore_eos:true \
  --tokenizer meta-llama/Meta-Llama-3-8B-Instruct \
  -- \
  -v \
  --max-threads=256
```



LLM Metrics				
Statistic	avg	min	max	p99
Time to first token (ns)	85,485,242	27,402,273	152,621,817	130,194,943
Inter token latency (ns)	8,847,758	2,113,030	74,794,303	9,477,464
Request latency (ns)	1,848,822,497	1,844,511,394	1,924,017,143	1,905,132,459
Num output token	184	177	190	189
Num input token	200	198	201	200

Output token throughput (per sec): 995.61  
Request throughput (per sec): 5.41



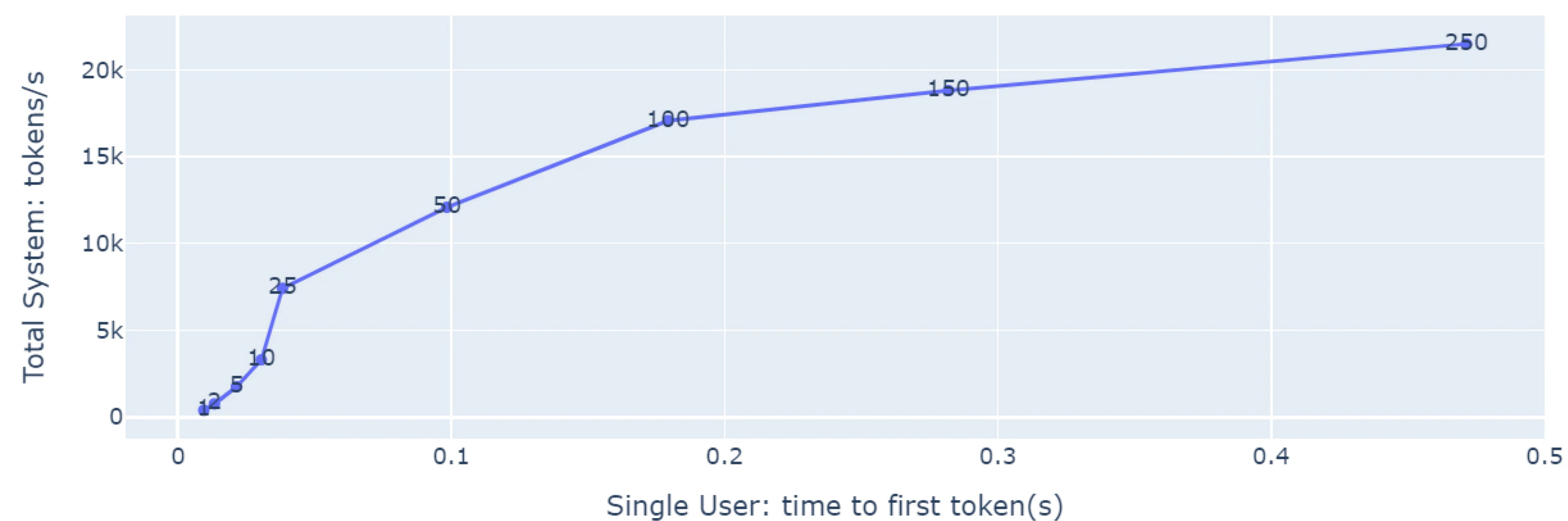
# Sweeping across concurrencies

Running multiple GenAI-Perf calls

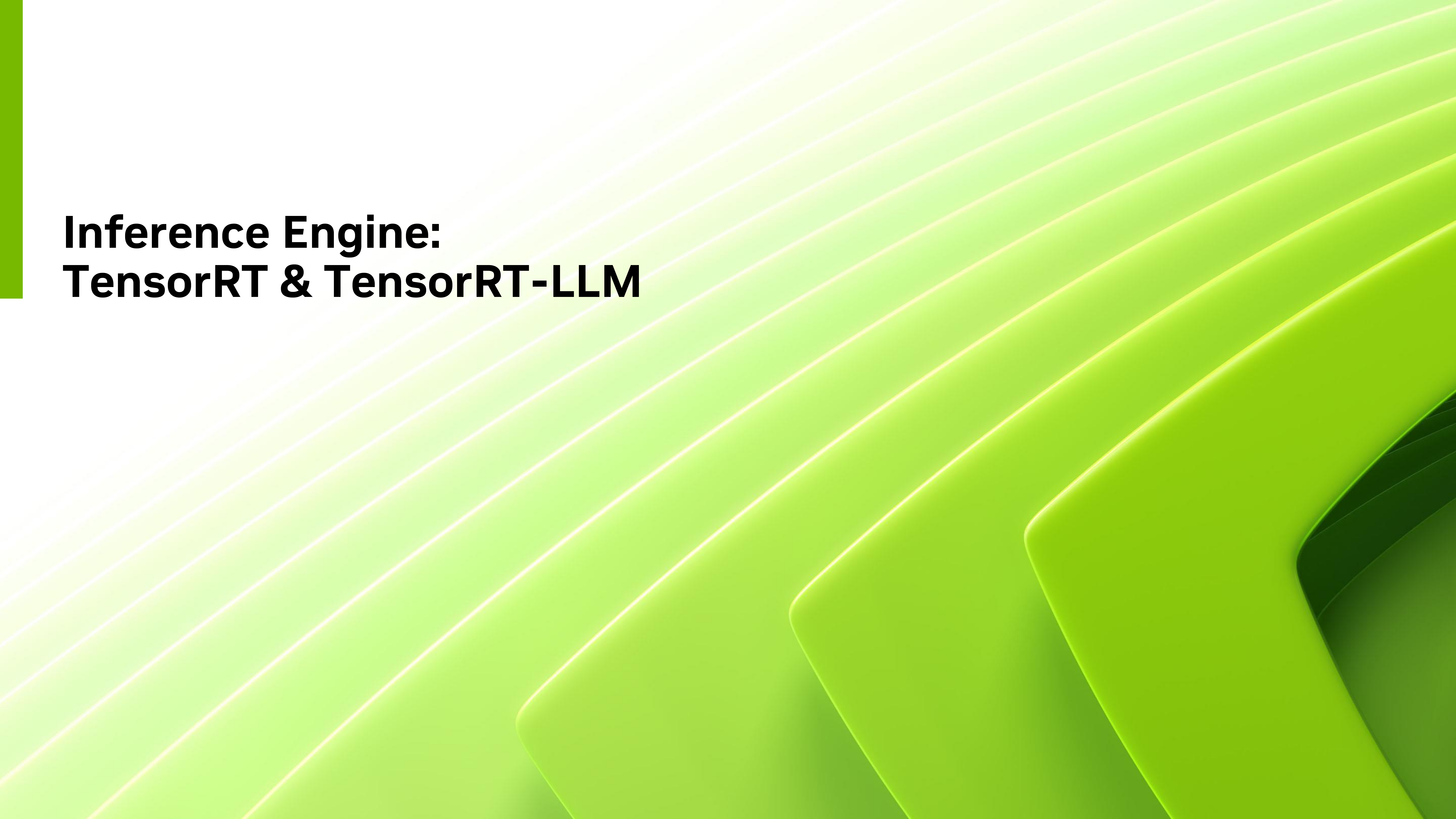
```
for concurrency in 1 2 5 10 50 100 250; do

    local INPUT_SEQUENCE_LENGTH=$inputLength
    local INPUT_SEQUENCE_STD=0
    local OUTPUT_SEQUENCE_LENGTH=$outputLength
    local CONCURRENCY=$concurrency
    local MODEL=meta/llama3-8b-instruct

    genai-perf \
        -m $MODEL \
        --endpoint-type chat \
        --service-kind openai \
        --streaming \
        -u localhost:8000 \
        --synthetic-input-tokens-mean $INPUT_SEQUENCE_LENGTH \
        --synthetic-input-tokens-stddev $INPUT_SEQUENCE_STD \
        --concurrency $CONCURRENCY \
        --output-tokens-mean $OUTPUT_SEQUENCE_LENGTH \
        --extra-inputs max_tokens:$OUTPUT_SEQUENCE_LENGTH \
        --extra-inputs min_tokens:$OUTPUT_SEQUENCE_LENGTH \
        --extra-inputs ignore_eos:true \
        --tokenizer meta-llama/Meta-Llama-3-8B-Instruct \
        --measurement-interval 10000 \
```







# **Inference Engine: TensorRT & TensorRT-LLM**



# TensorRT-LLM in the *DL Compiler* Ecosystem

TensorRT-LLM builds on TensorRT Compilation

- **TensorRT-LLM**

- Inference runtime & compiler specifically designed for LLMs
- LLM specific optimizations:
  - KV Caching & Custom MHA Kernels
  - Inflight batching, Paged KV Cache (Attention)
  - Multi-GPU, Multi-Node
  - Grammar support
  - & more
- *ONLY for LLMs*

- **TensorRT**

- General purpose Deep Learning Inference Compiler
  - Graph rewriting, constant folding, kernel fusion
  - Optimized GEMMs & pointwise kernels
  - Kernel Auto-Tuning
  - Memory Optimizations
  - & more
- *All AI Workloads*

## TensorRT-LLM

LLM specific optimizations:

- KV Caching
- Multi-GPU, Muti-Node
- Custom MHA optimizations
- Paged KV Cache (Attention)
- etc...

## TensorRT

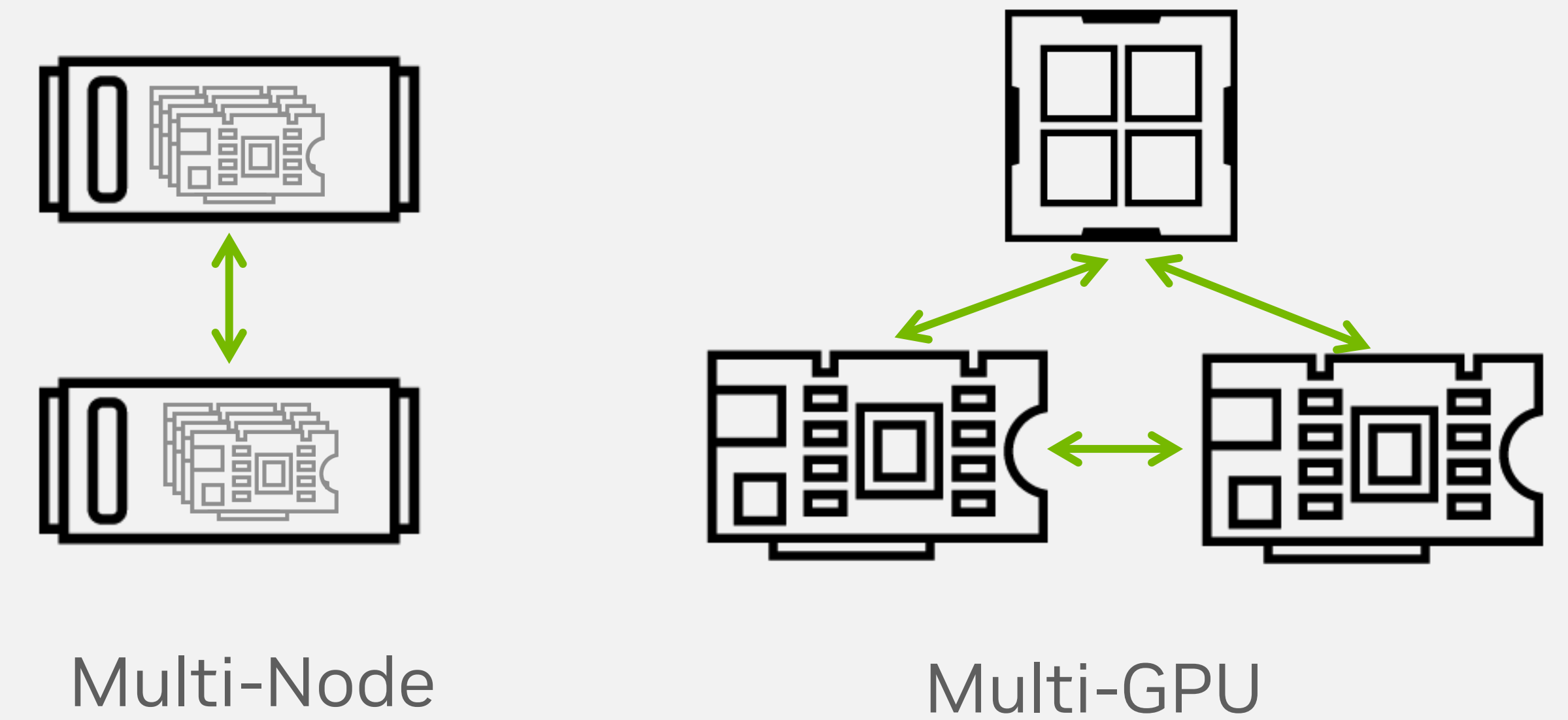
General Purpose Compiler

- Optimized GEMMs & general kernels
- Kernel Fusion
- Auto Tuning
- Memory Optimizations
- Multi-stream execution

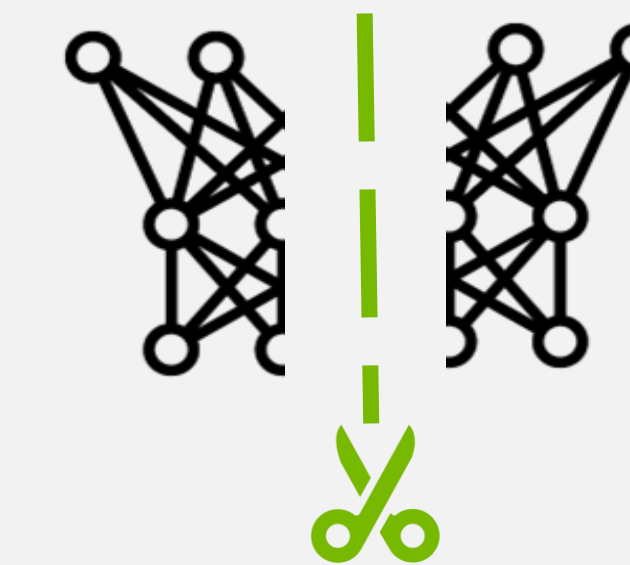
# Multi-GPU Multi-Node

## Sharding Models across GPUs

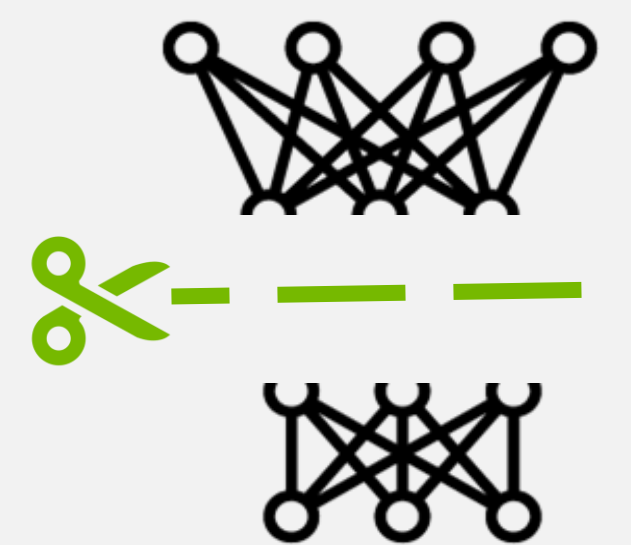
- Supports Tensor & Pipeline parallelism
- Allows for running very large models (tested up to 530B)
- Supports multi-GPU (single node) & multi-node
- TensorRT-LLM handles communication between GPUs
- Examples are parametrized for sharding across GPUs



No Parallelism



Tensor Parallel



Pipeline Parallel





# **Inference Optimizations**

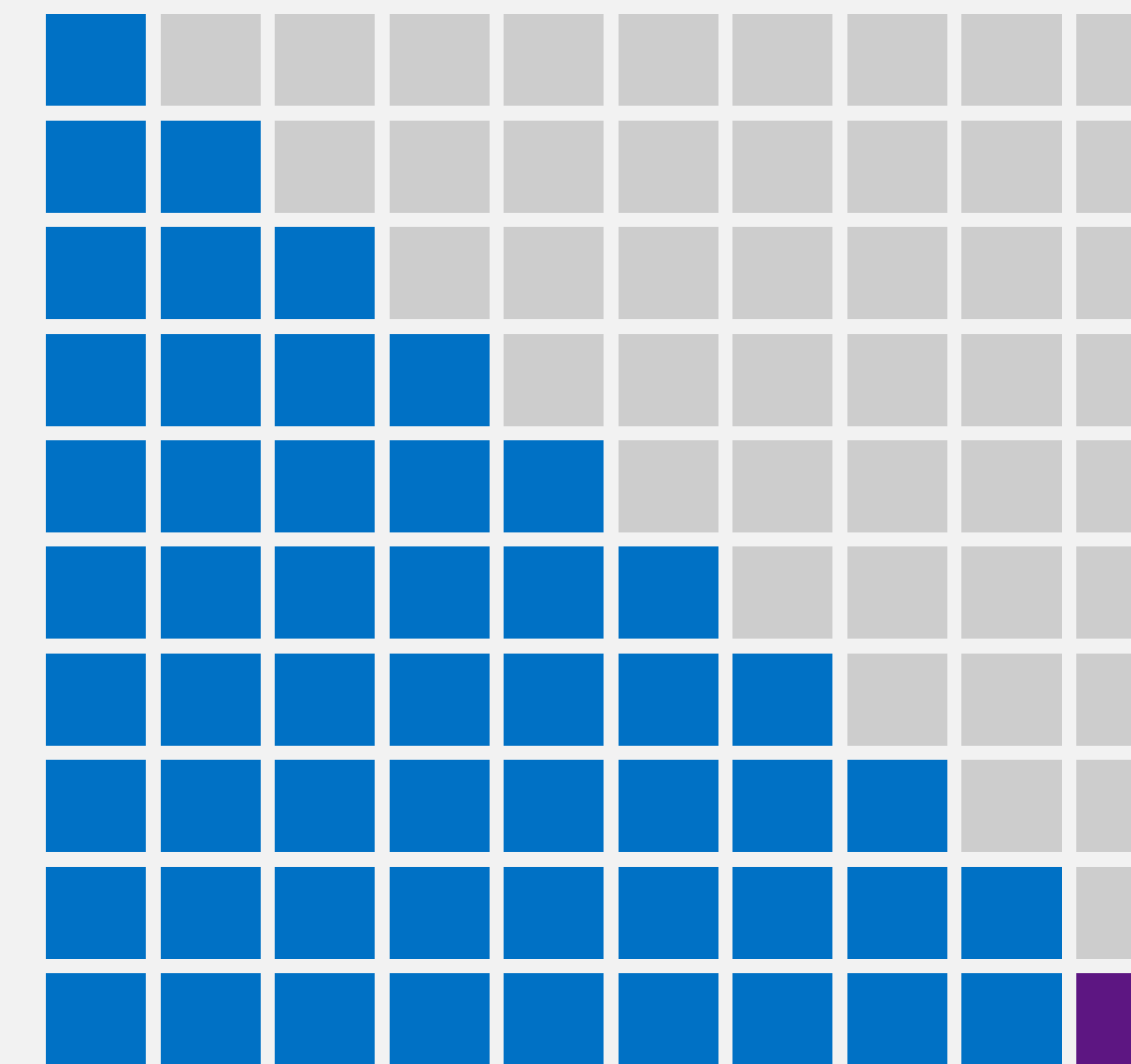


# KV Cache & Attention Techniques

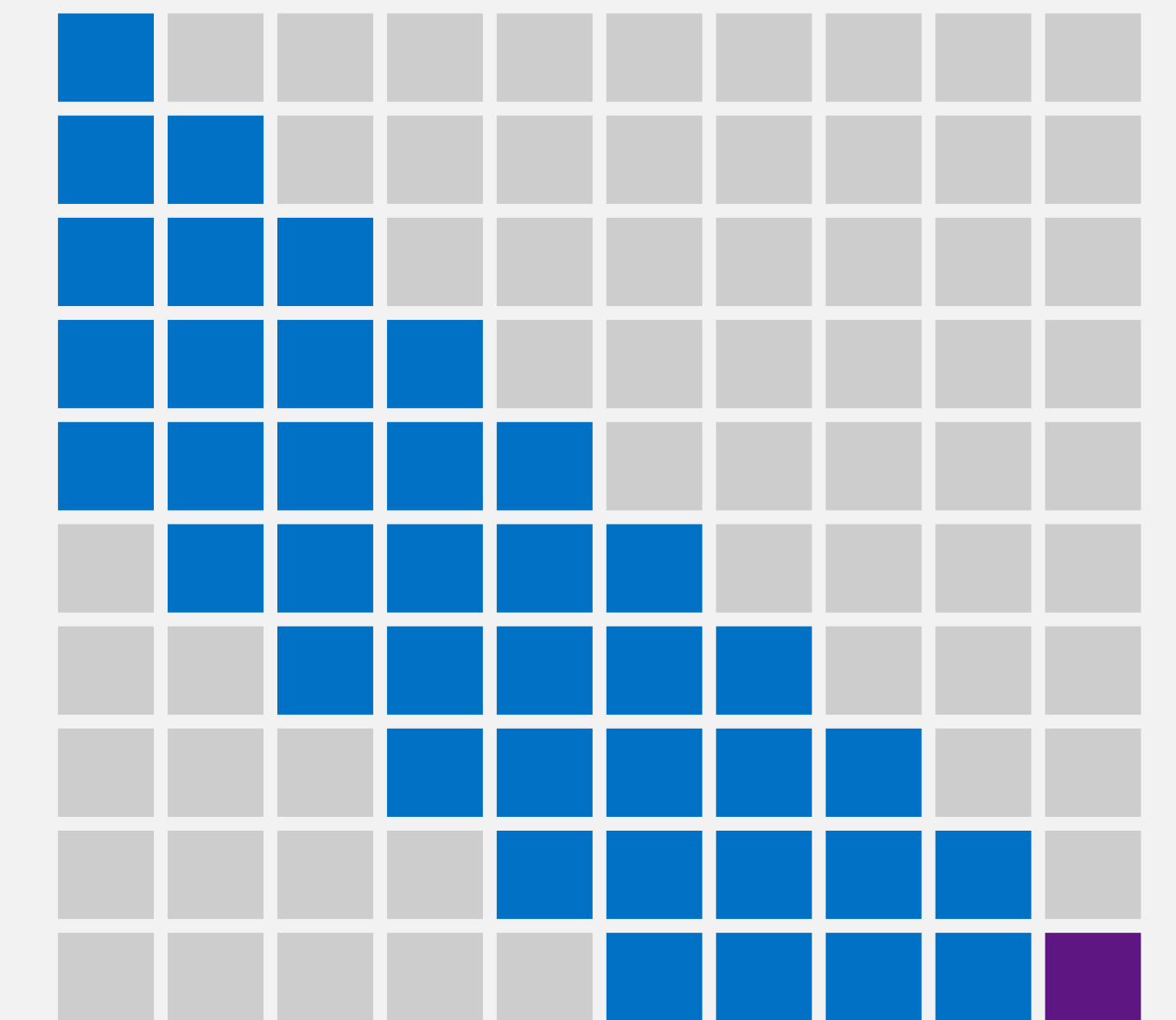
## (Sliding) Window Attention, & Streaming LLM

- Allow for longer (sometimes unlimited) sequence length
  - Reduces KV Cache Memory usage
  - Avoids OOM Errors
- (Sliding) Windowed Attention evict tokens based on arrival
  - Significantly reduces memory usage
  - Can negatively impact accuracy or require recomputing KV
- Streaming-LLM allows for unlimited sequence length
  - Does not evict Attention Sinks (important elements)
  - KV Cache stays constant size
  - Does not require recompute & does not impact accuracy
  - Particularly beneficial for multi-turn (ie. chat) usecases

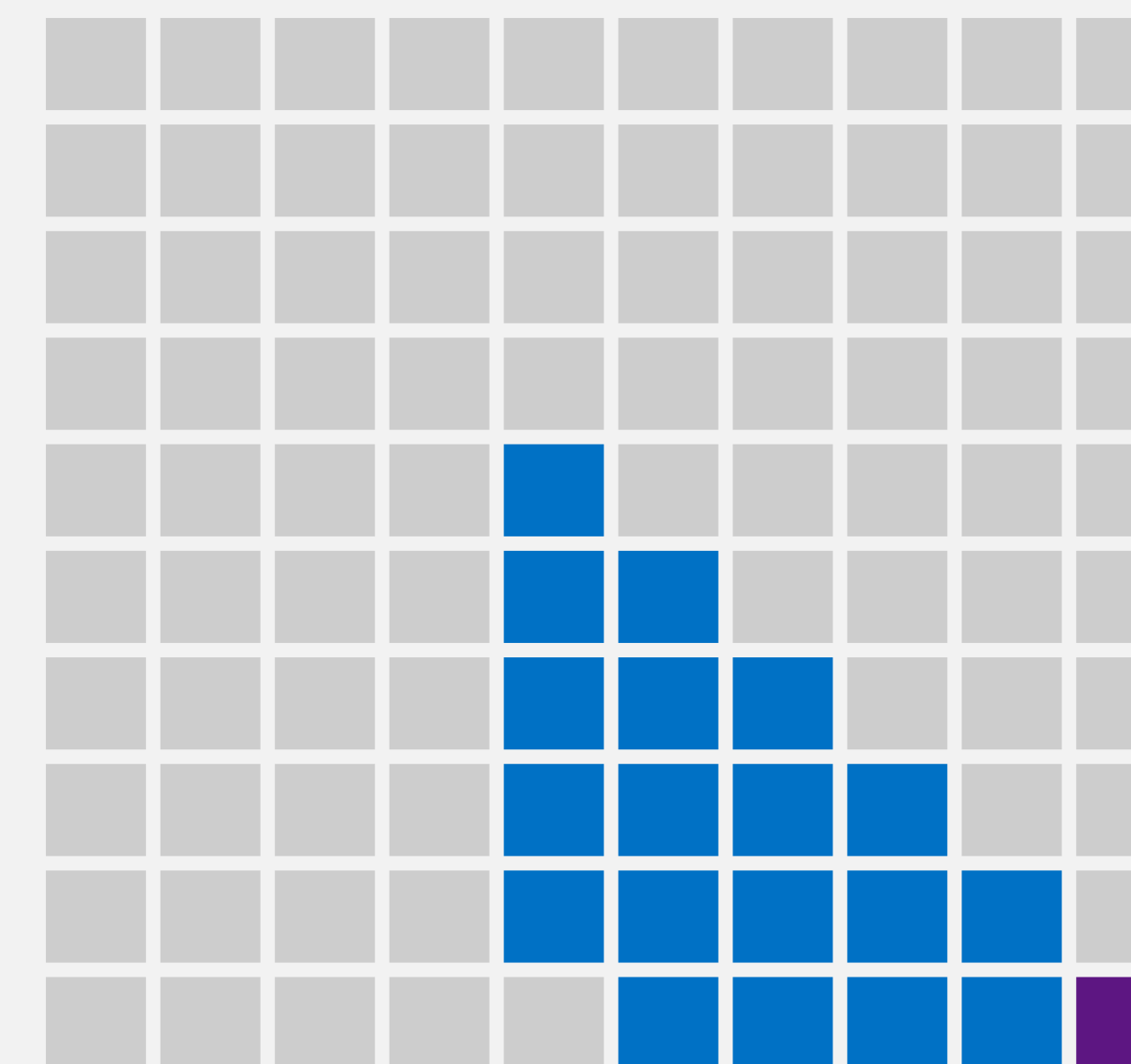
Attention KV Cache Usage (*Less is Better*)



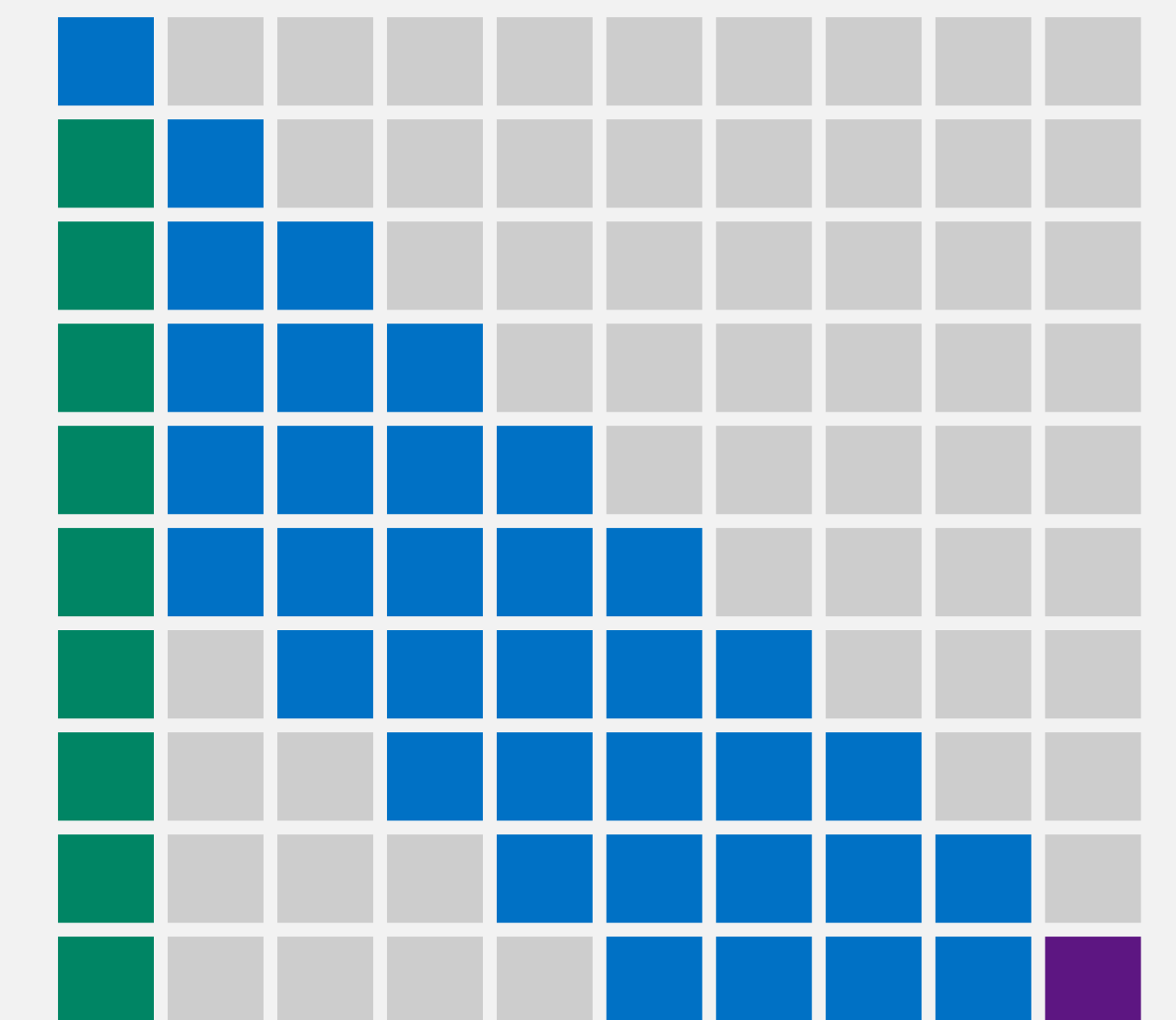
Dense



Windowed



Sliding Window



StreamingLLM

Free Prev. Tokens Curr. Token Attn. Sync

# Inflight Batching

## Maximizing GPU Utilization during LLM Serving

TensorRT-LLM provides custom Inflight Batching to optimize GPU utilization during LLM Serving

- Replaces completed requests in the batch
  - Evicts requests after EoS & inserts a new request
- Improves throughput, time to first token, & GPU utilization
- Integrated directly into the TensorRT-LLM Triton backend
- Accessible through the TensorRT-LLM Batch Manager

Batch Elements	Iteration									...
	1	2	3	4	5	6	7	8	9	
$R_1$						END			$R_5$	...
$R_2$			END						$R_6$	...
$R_3$					END				$R_7$	...
$R_4$								END	$R_8$	...

Static Batching

Batch Elements	Iteration									...
	1	2	3	4	5	6	7	8	9	
$R_1$						END	$R_7$			...
$R_2$			END	$R_5$						...
$R_3$					END	$R_6$		END	$R_8$	...
$R_4$								END	$R_9$	...

Inflight Batching

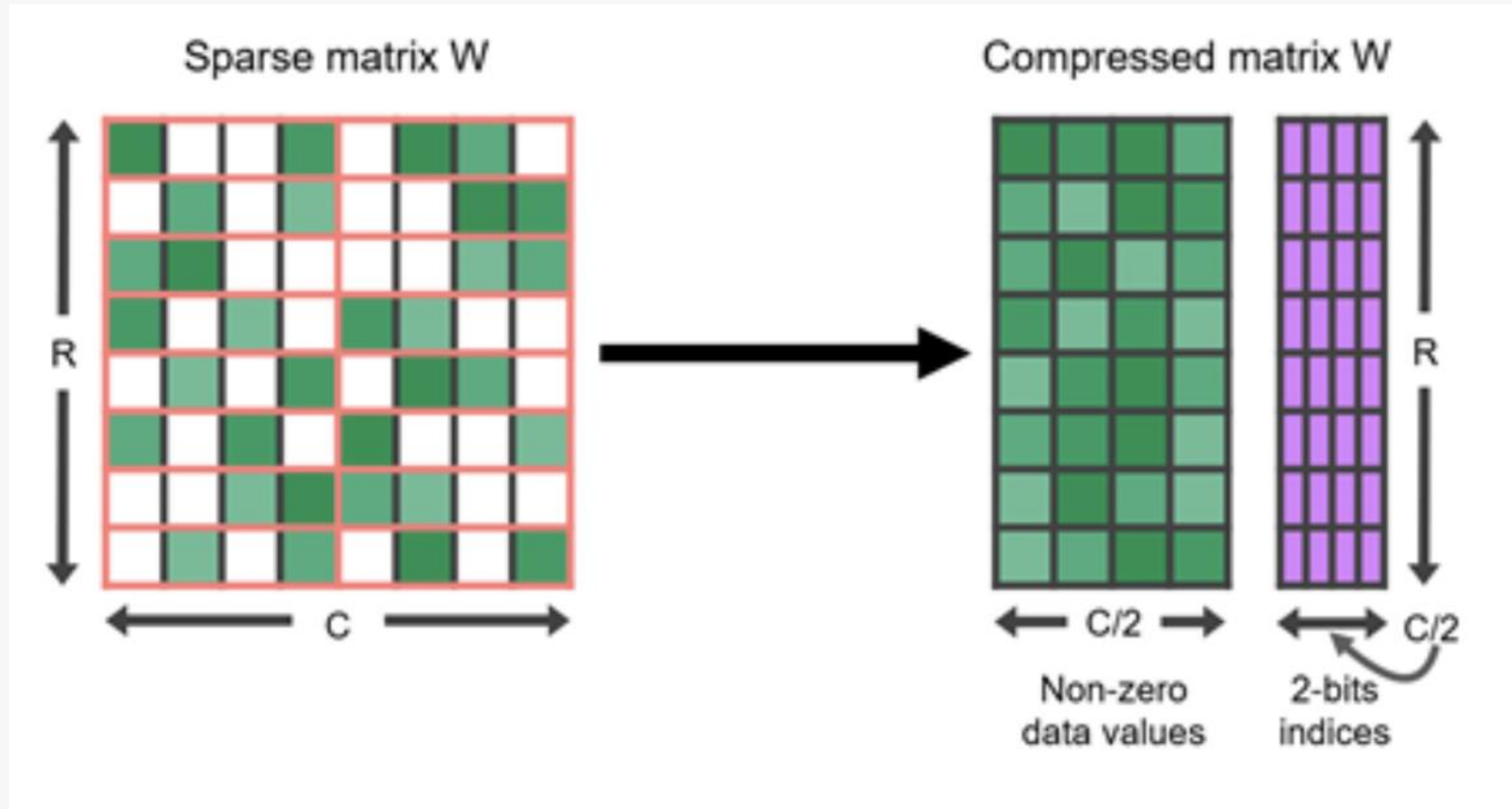
Context Gen EoS NoOp

# Model Optimizer - Sparsity

Model Optimizer offers Sparsity (with fine-tuning or post-training sparsity) to reduce the memory footprint and accelerate inference.

It supports [NVIDIA 2:4 Sparsity](#) sparsity pattern and various sparsification methods, such as ([NVIDIA ASP](#)) and ([SparseGPT](#)).

- **MLPerf-Inference v4.0 Results** ([blog](#))
  - . Uses 2:4 sparsity on a Llama2-70B.
  - . Post-training sparsification (PTS) with SparseGPT yields no accuracy drop.
  - . PTS reduces model size by **37%**, facilitating deployment on a single H100 SXM with TP=1, PP=1, and achieving a **1.3x speedup**.



A 2:4 structured sparse matrix W, and its compressed representation

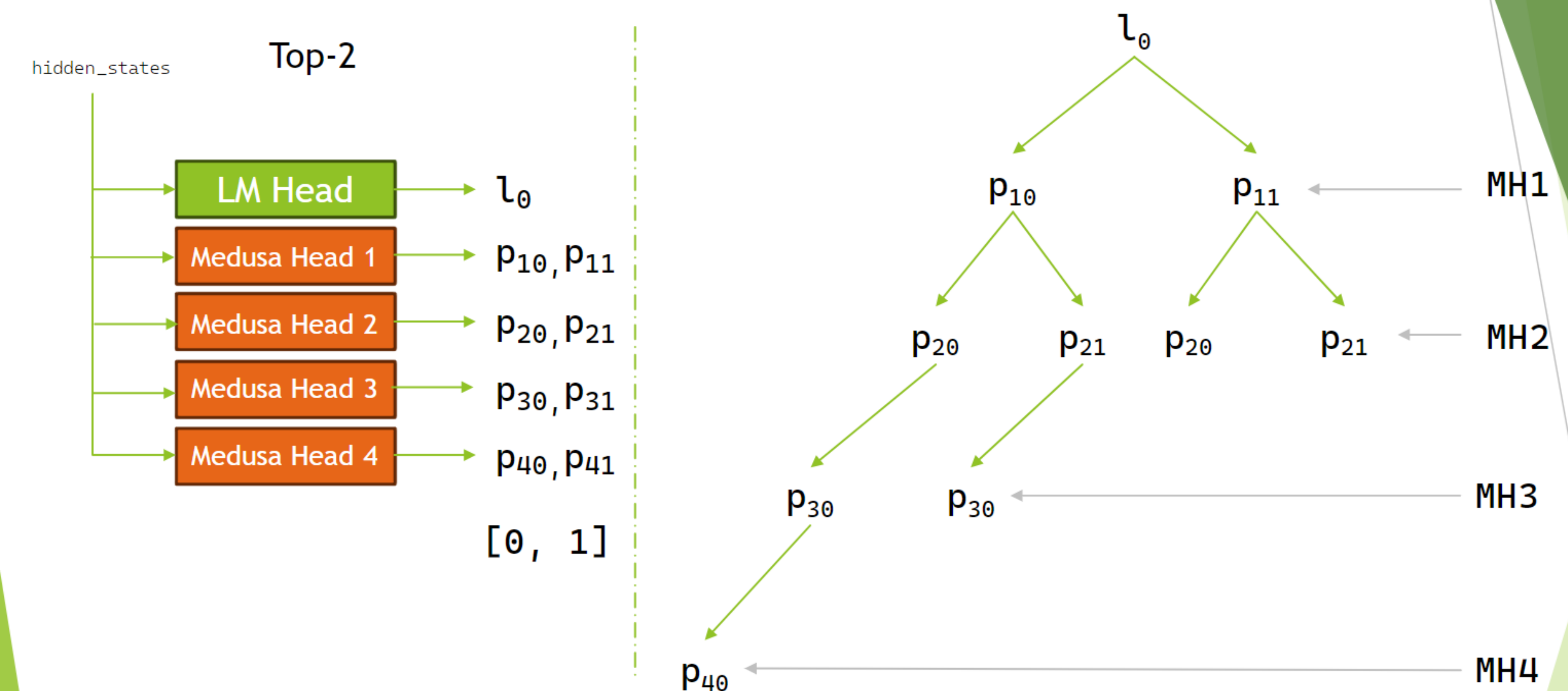
Model	Batch_size	Inference Speedup (Compared to the FP8 dense model with the same batch size)
Sparsified Llama 2-70B	32	1.62x
	64	1.52x
	128	1.35x
	896 (MLPerf)	1.30x

Performance improvement with sparsity



# Speculative Decoding

- Generate of more than one token per forward pass iteration
- Helpful if only if the GPU is underutilized due to small batch sizes.
- Perform more computations, keeping memory access almost the same
- Supports: separate draft model, Medusa, ReDrafter



# of paths: 4 (16 if all MHs have Top-2)

Paths:  $l_0 p_{10} p_{20} p_{30} p_{40}$ ,  $l_0 p_{10} p_{21} p_{30}$ ,  $l_0 p_{11} p_{20}$ ,  $l_0 p_{11} p_{21}$

# of candidates: 10 ( $l_0$ ,  $l_0 p_{10}$ ,  $l_0 p_{10} p_{20}$ ,  $l_0 p_{10} p_{20} p_{30}$ ,  $l_0 p_{10} p_{20} p_{30} p_{40}$ ,  $l_0 p_{10} p_{21}$ ,  $l_0 p_{10} p_{21} p_{30}$ ,  $l_0 p_{11}$ ,  $l_0 p_{11} p_{20}$ ,  $l_0 p_{11} p_{21}$ )





**Inference serving**



# LLM serving frameworks



**Dynamo**

# vLLM

## Running vLLM on multiple nodes

If a single node does not have enough GPUs to hold the model, you can run the model using multiple nodes. It is important to make sure the execution environment is the same on all nodes, including the model path, the Python environment. The recommended way is to use docker images to ensure the same environment, and hide the heterogeneity of the host machines via mapping them into the same docker configuration.

The first step, is to start containers and organize them into a cluster. We have provided the helper script [examples/online\\_serving/run\\_cluster.sh](#) to start the cluster. Please note, this script launches docker without administrative privileges that would be required to access GPU performance counters when running profiling and tracing tools. For that purpose, the script can have `CAP_SYS_ADMIN` to the docker container by using the `--cap-add` option in the docker run command.

Pick a node as the head node, and run the following command:

```
bash run_cluster.sh \  
    vllm/vllm-openai \  
    ip_of_head_node \  
    --head \  
    /path/to/the/huggingface/home/in/this/node \  
    -e VLLM_HOST_IP=ip_of_this_node
```

On the rest of the worker nodes, run the following command:

```
bash run_cluster.sh \  
    vllm/vllm-openai \  
    ip_of_head_node \  
    --worker \  
    /path/to/the/huggingface/home/in/this/node \  
    -e VLLM_HOST_IP=ip_of_this_node
```

Then you get a ray cluster of **containers**. Note that you need to keep the shells running these commands alive to hold the cluster. Any shell disconnect will terminate the cluster. In addition, please note that the argument `ip_of_head_node` should be the IP address of the head node, which is accessible by all the worker nodes. The IP addresses of each worker node should be specified in the `VLLM_HOST_IP` environment variable, and should be different for each worker node. Please check the network configuration of your cluster to make sure the nodes can communicate with each other through the specified IP addresses.

```
from vllm import LLM, SamplingParams  
  
prompts = [  
    "Hello, my name is",  
    "The president of the United States is",  
]  
  
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)  
  
def main():  
    llm = LLM(model="TinyLlama/TinyLlama-1.1B-Chat-v1.0")  
  
    outputs = llm.generate(prompts, sampling_params)  
  
if __name__ == "__main__":  
    main()
```



# SGLang



Search Ctrl + K

## Installation

Install SGLang

## Backend Tutorial

DeepSeek Usage

Llama4 Usage

Sending Requests

OpenAI APIs - Completions

OpenAI APIs - Vision

OpenAI APIs - Embedding

SGLang Native APIs

Offline Engine API

Server Arguments

Sampling Parameters

Hyperparameter Tuning

Attention Backend



## Multi-Node Inference on SLURM

This example showcases how to serve SGLang server across multiple nodes by SLURM. Submit the following job to the SLURM cluster.

```
#!/bin/bash -l

#SBATCH -o SLURM_Logs/%x_%j_master.out
#SBATCH -e SLURM_Logs/%x_%j_master.err
#SBATCH -D ./
#SBATCH -J Llama-405B-Online-Inference-TP16-SGL

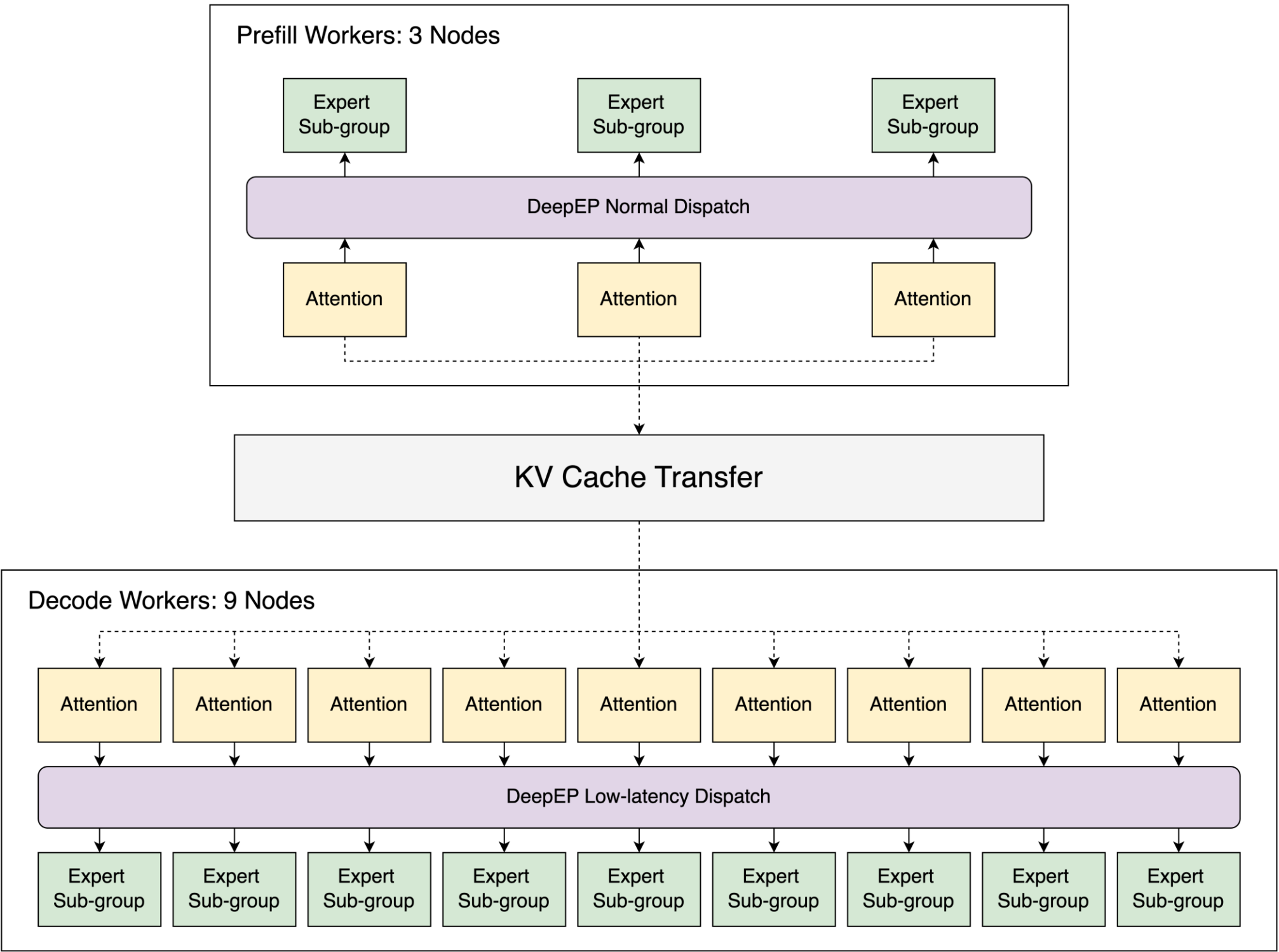
#SBATCH --nodes=2
#SBATCH --ntasks=2
#SBATCH --ntasks-per-node=1 # Ensure 1 task per node
#SBATCH --cpus-per-task=18
#SBATCH --mem=224GB
#SBATCH --partition="lmsys.org"
#SBATCH --gres=gpu:8
#SBATCH --time=12:00:00

echo "[INFO] Activating environment on node $SLURM_PROCID"
if ! source ENV_FOLDER/bin/activate; then
    echo "[ERROR] Failed to activate environment" >&2
    exit 1
fi

# Define parameters
model=MODEL_PATH
tp_size=16

echo "[INFO] Running inference"
```

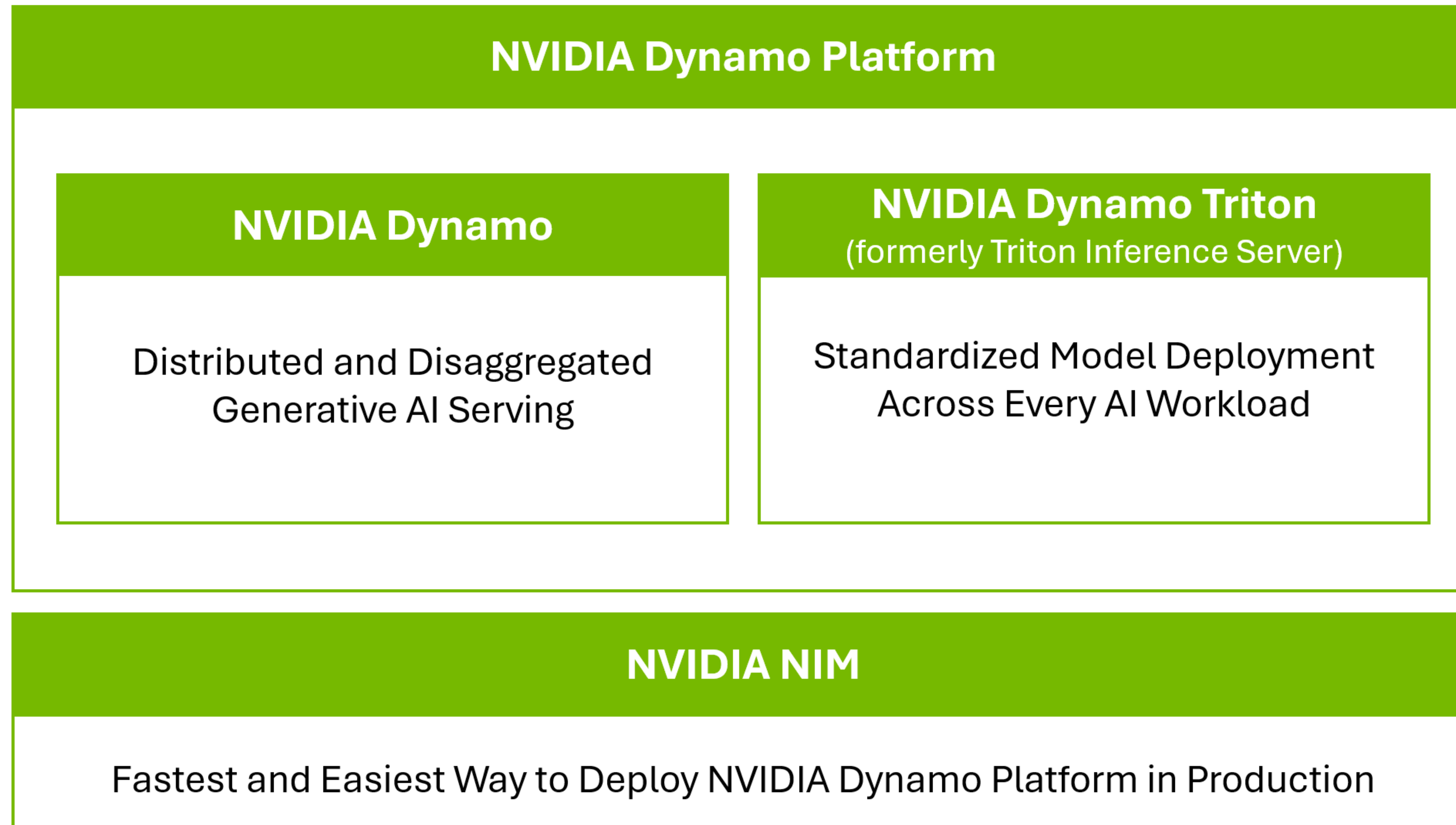
[https://docs.sglang.ai/references/multi\\_node.htm](https://docs.sglang.ai/references/multi_node.htm)  
!



<https://lmsys.org/blog/2025-05-05-large-scale-ep/>

# NVIDIA Dynamo Platform

The Operating System for AI Factories

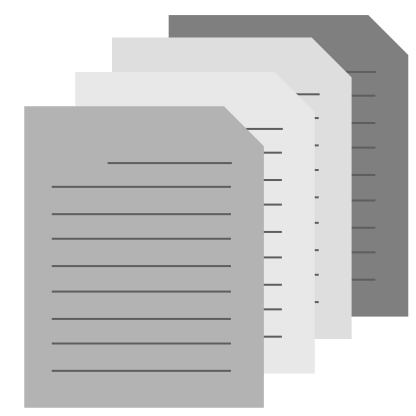




# NVIDIA Dynamo Triton (formerly Triton Inference Server)

Deploy models from all popular frameworks across GPUs and CPUs

Any Framework



Supports Multiple Framework Backends Natively e.g., TensorFlow, PyTorch, TensorRT, XGBoost, ONNX, Python, TensorRT-LLM, vLLM & More

Any Query Type



Optimized for Real Time, Batch, Streaming, Ensemble Inferencing

Any Platform

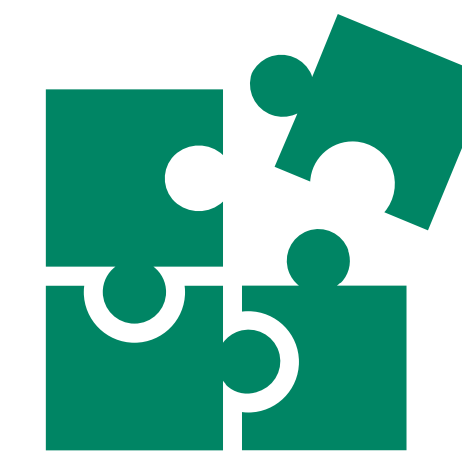


X86 CPU | Arm CPU | NVIDIA GPUs | MIG

Linux | Windows | Virtualization

Public Cloud, Data Center and Edge/Embedded (Jetson)

DevOps & MLOps



Integration With Kubernetes, KServe, Prometheus & Grafana

Available Across All Major Cloud AI Platforms

Performance & Utilization

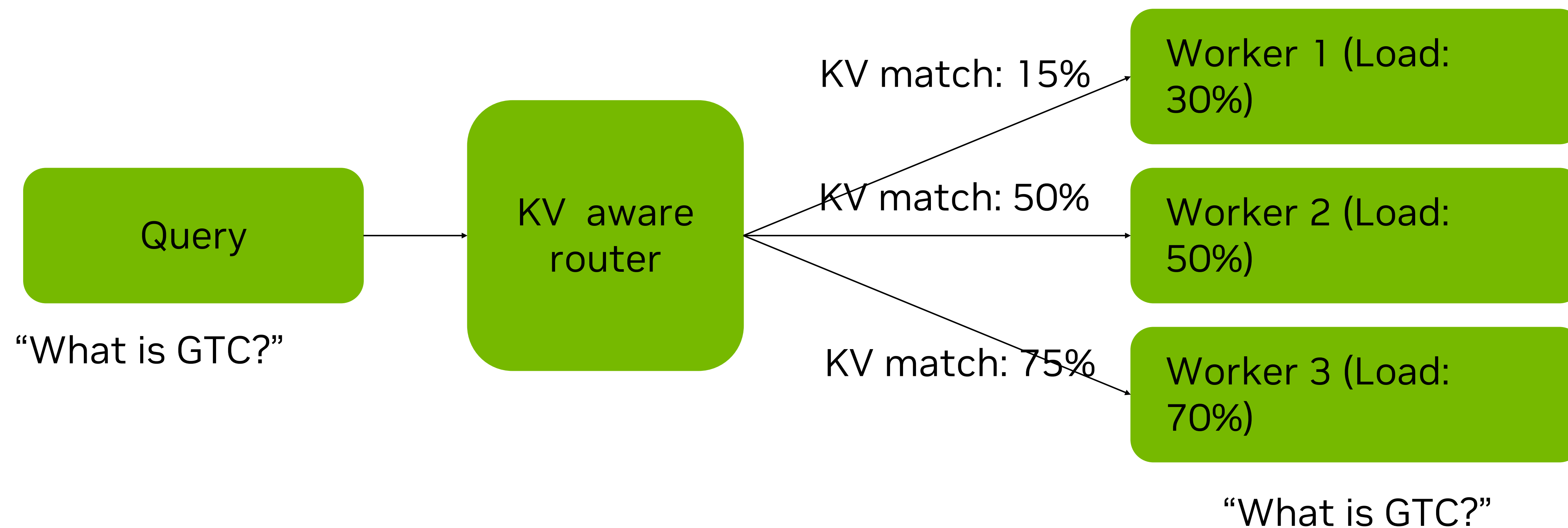
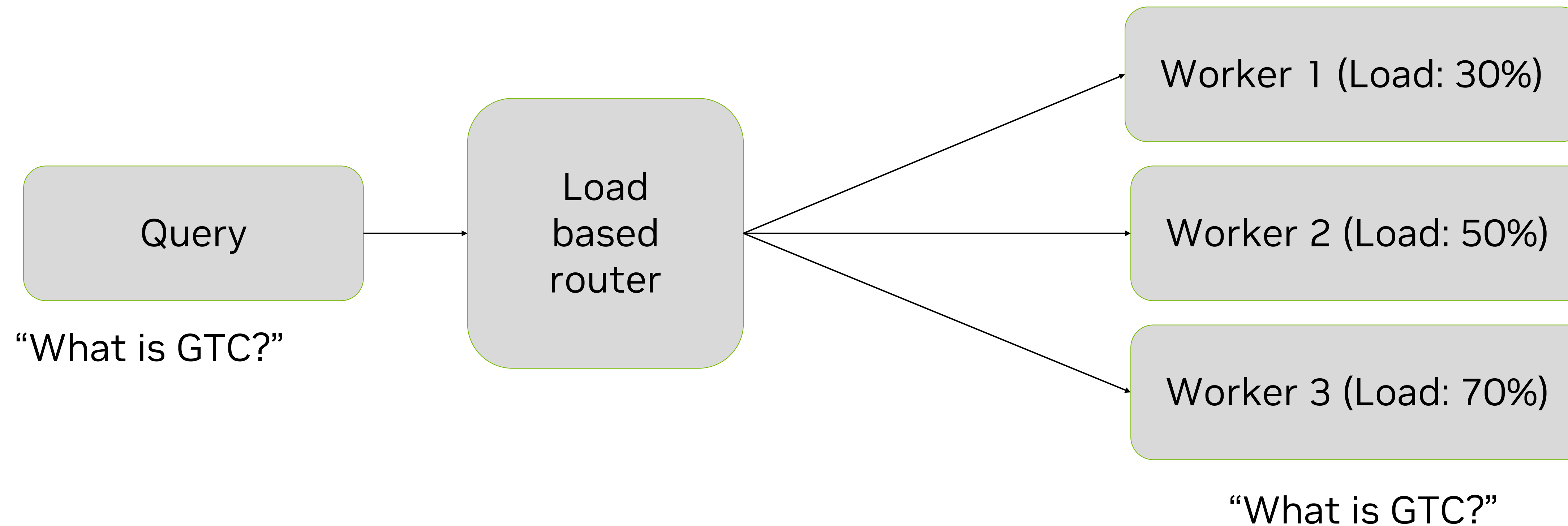


Model Analyzer for Optimal Configuration

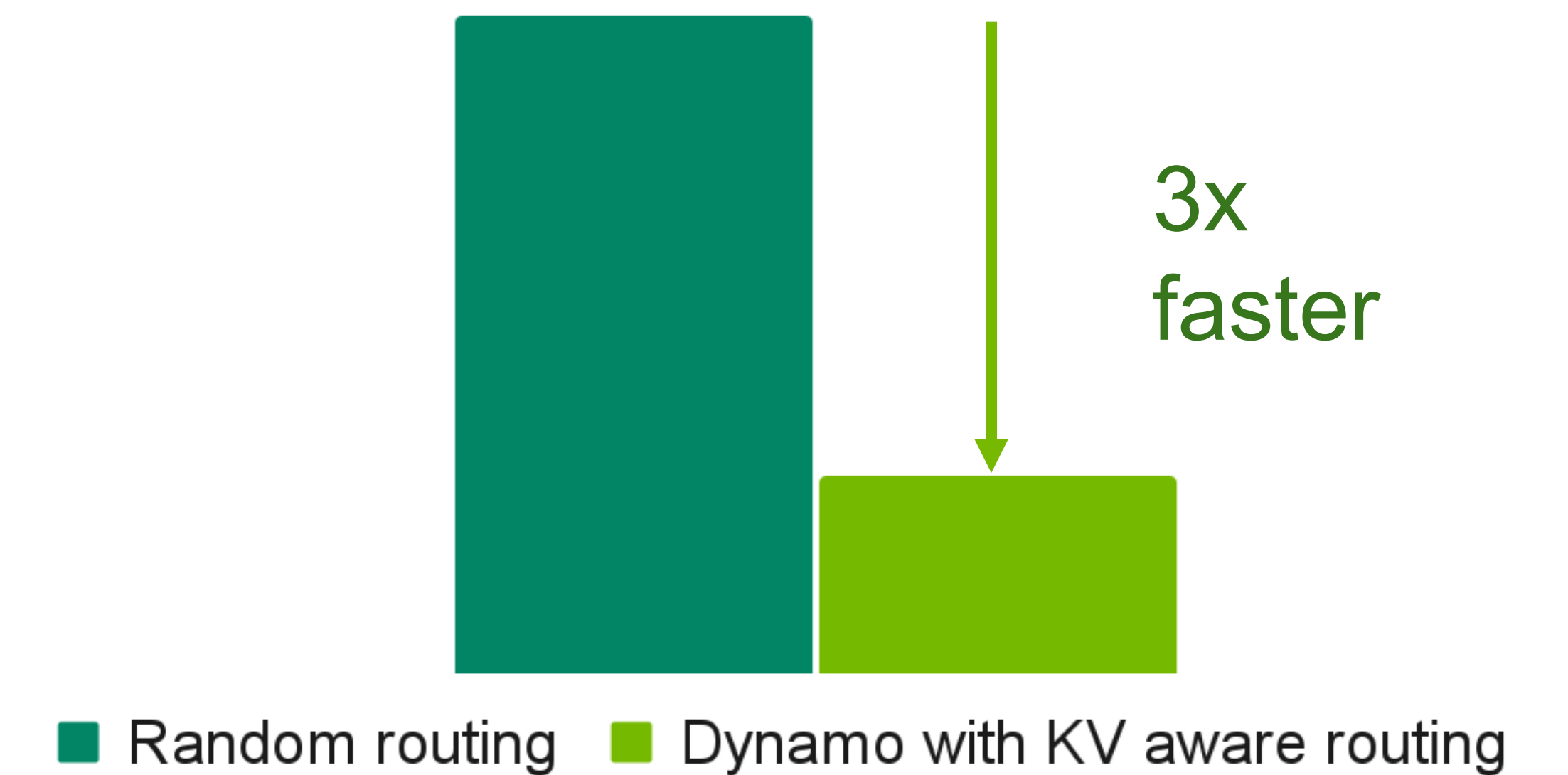
Optimized for High GPU/CPU Utilization, High Throughput & Low Latency

# KV Cache Aware Routing

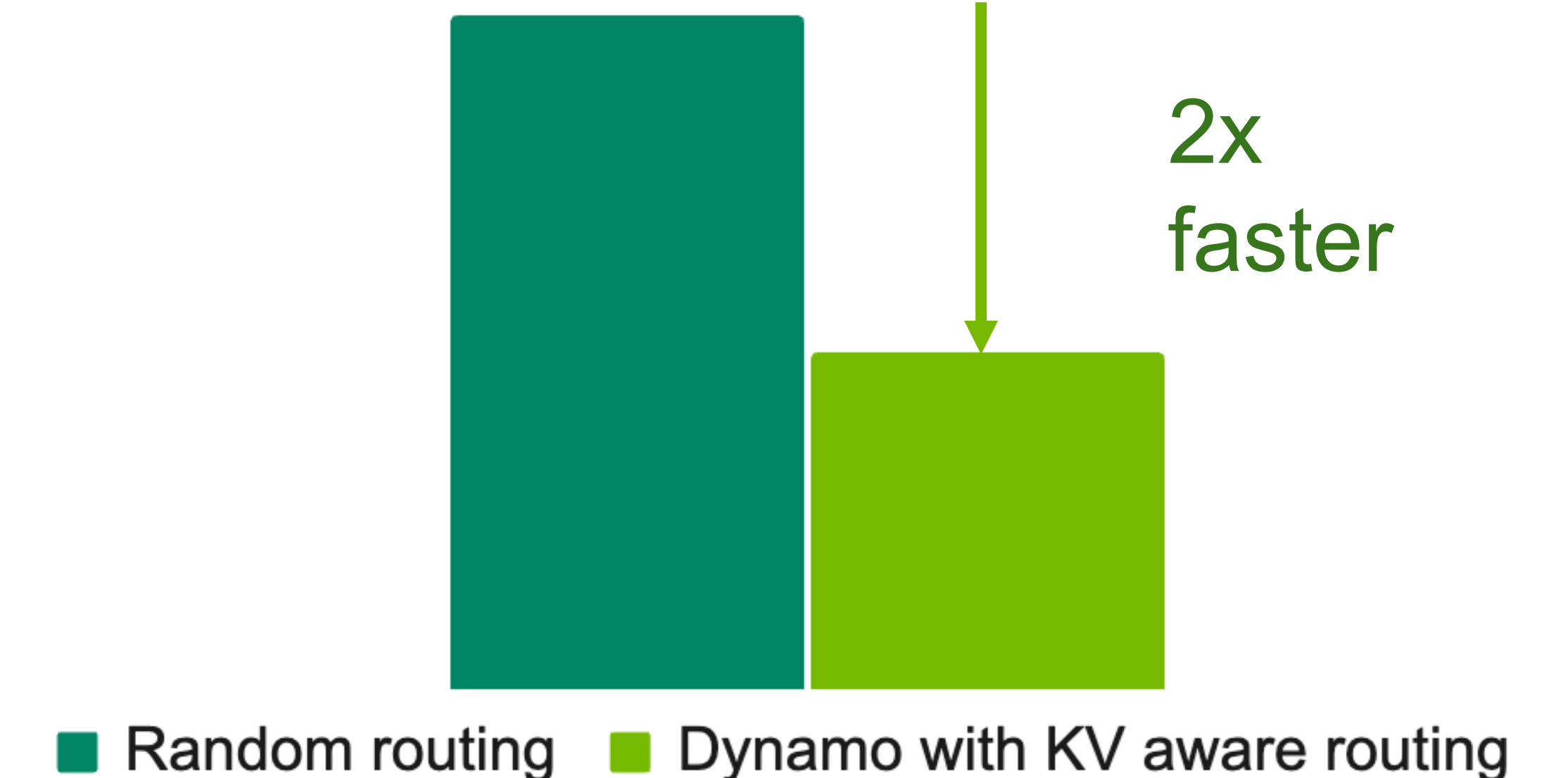
Significant boost in TTFT and End to End Latency with real data (100K requests with R1)



Time To First Token



Avg request latency



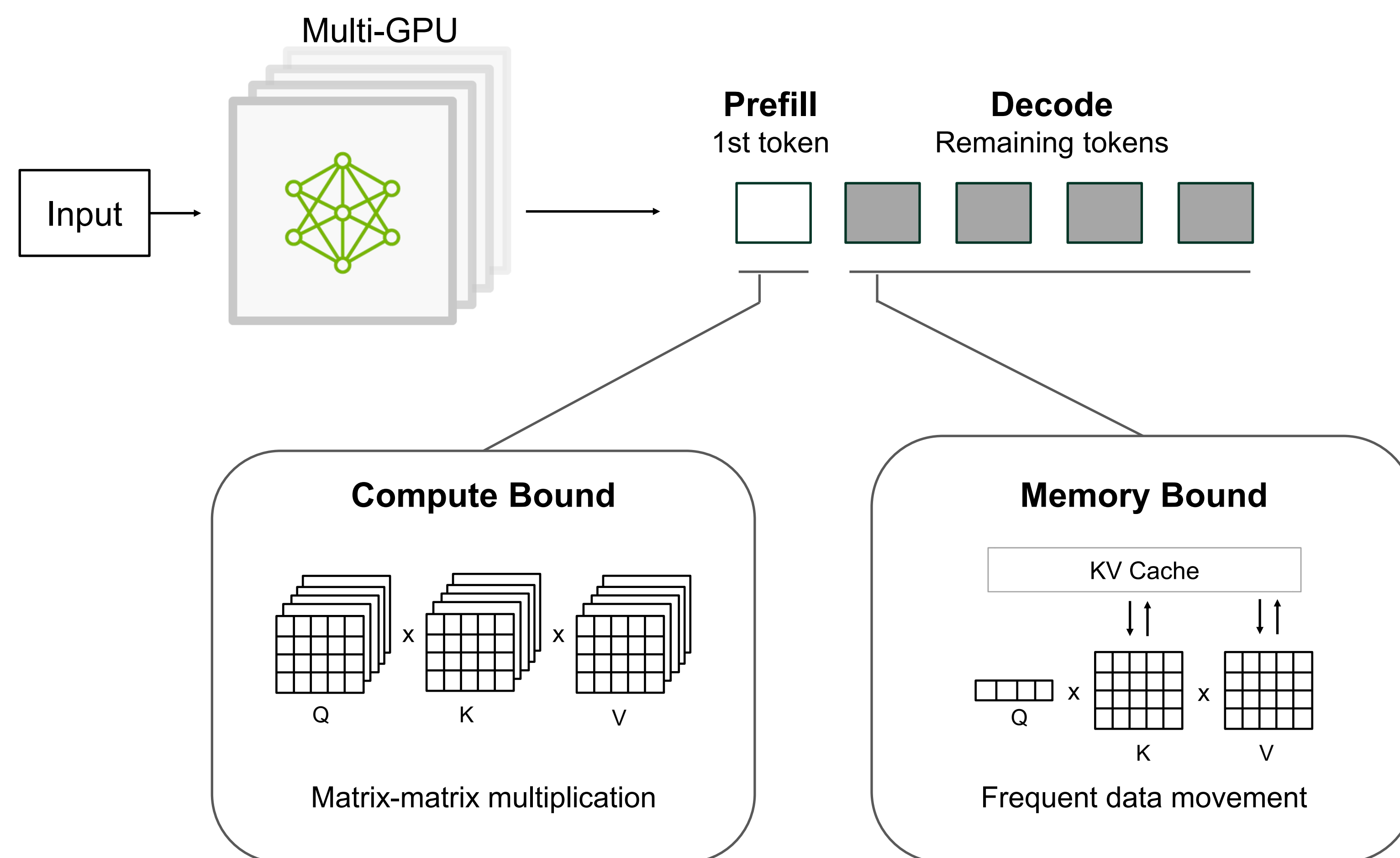
Tested with R1 Distilled Llama 70B over 2 nodes of 8 x H100s with vLLM 0.7.3



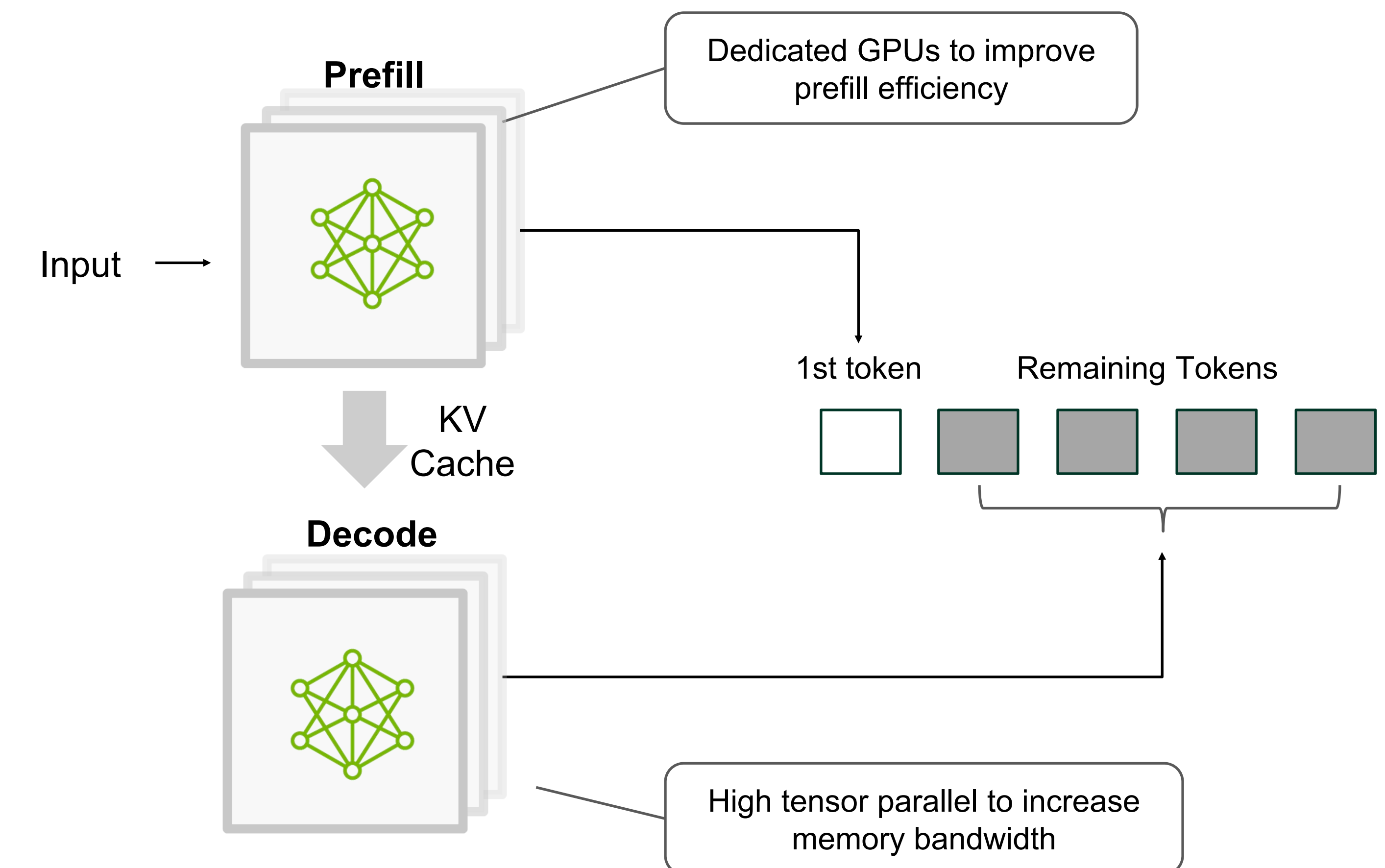
# New Inference Optimization Techniques to Boost Inference

Disaggregated serving separates prefill and decode allowing each to be optimized independently

## Traditional Serving



## Disaggregated Serving



More flexibility to optimize cost and user experience

# Example: Number of Tokens per Use Case

Use Cases	Context Size	Avg Number of Input Tokens	Avg Number Output Tokens
Language Translation - Code refactoring	Short	200	200
	Long	1000	1000
Generation of emails - Stories - Content search		500	2000
Summarization - RAG	Long	5000	500
	Extremely long	20000	2000
Reasoning	Medium	1000	20000