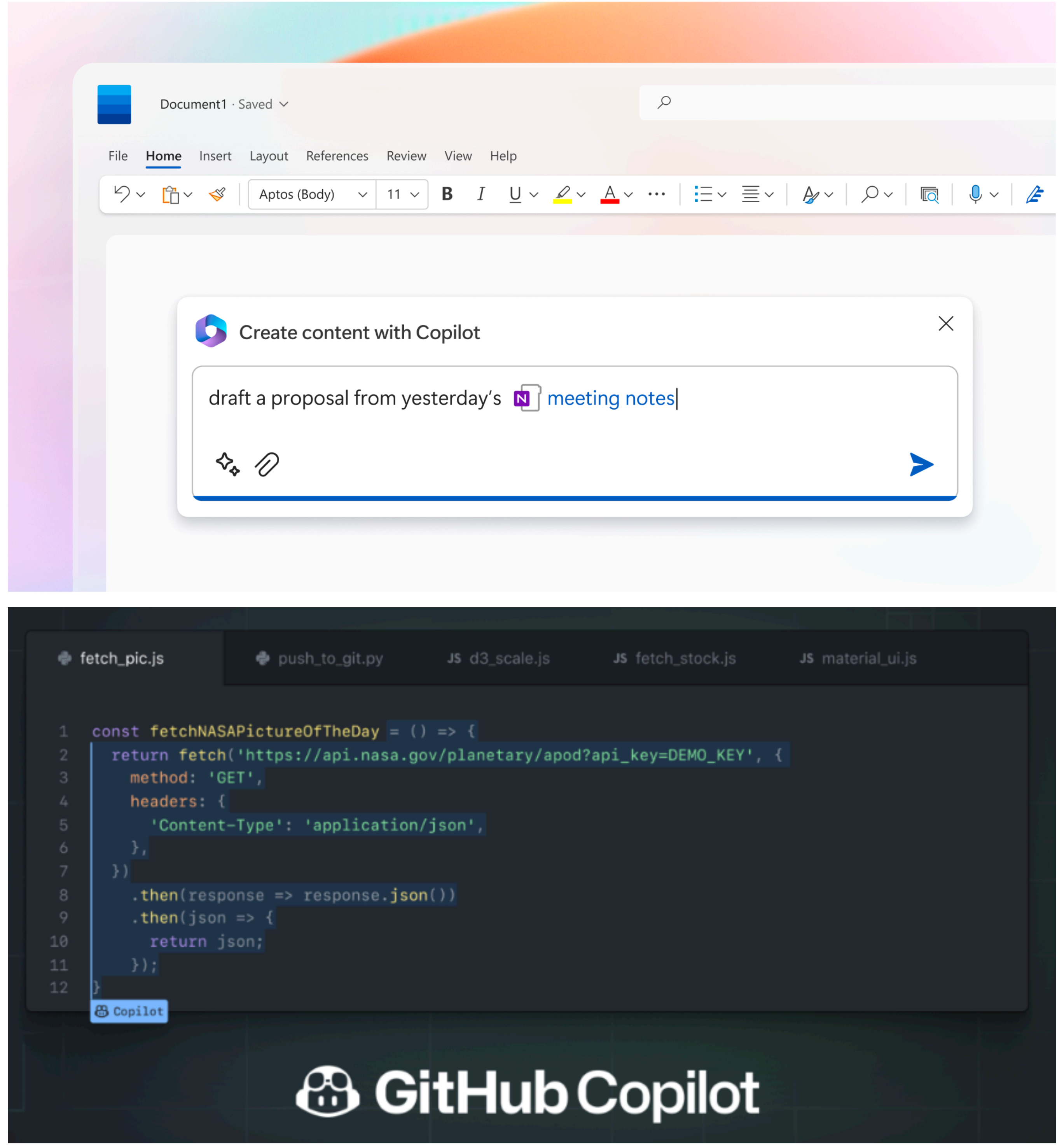
Transformer Inference

From first principles to the current state of the art

Linden Li
NeurIPS 2023

Real-time user interactions require performant inference of large models

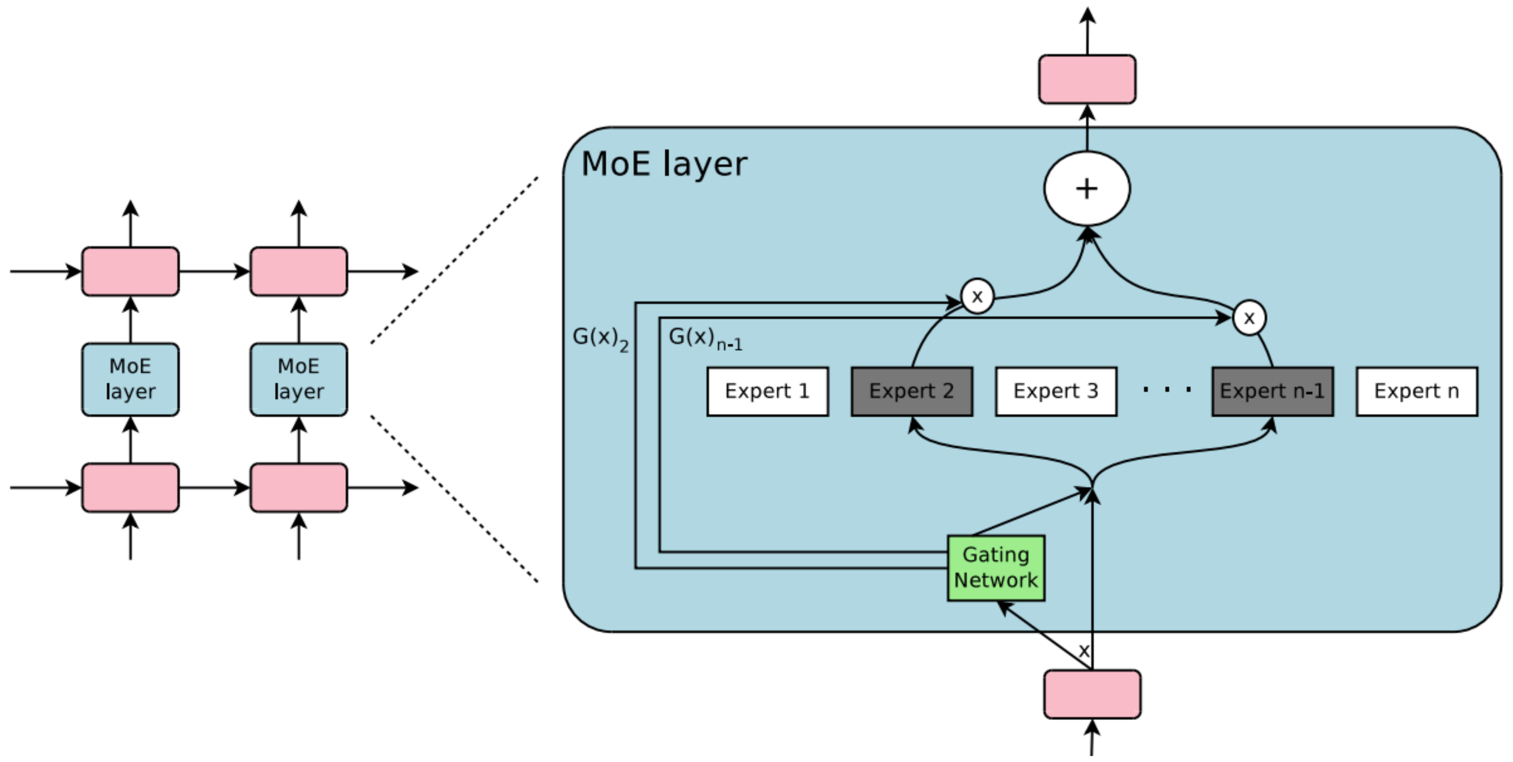
Chatbots: rapid response times after a user message



Copilots: rapid, real time assistive applications with dynamically updated suggestions after a few keystrokes

High throughput batch inference: process several documents at once

Inference constraints affect all portions of the machine learning pipeline




```python
class Attention(nn.Module):
    """Multi-head attention module."""
    def __init__(self, args: ModelArgs):
        """
        Initialize the Attention module.

        Args:
            args (ModelArgs): Model configuration parameters.

        Attributes:
            n_kv_heads (int): Number of key and value heads.
            n_local_heads (int): Number of local query heads.
            n_local_kv_heads (int): Number of local key and value heads.
            n_rep (int): Number of repetitions for local heads.
            head_dim (int): Dimension size of each attention head.
            wq (ColumnParallelLinear): Linear transformation for queries.
            wk (ColumnParallelLinear): Linear transformation for keys.
            wv (ColumnParallelLinear): Linear transformation for values.
            wo (RowParallelLinear): Linear transformation for output.
            cache_k (torch.Tensor): Cached keys for attention.
            cache_v (torch.Tensor): Cached values for attention.

        """
        super().__init__()
        self.n_kv_heads = args.n_heads if args.n_kv_heads is None else args.n_kv_heads
        model_parallel_size = fs_init.get_model_parallel_world_size()
        self.n_local_heads = args.n_heads // model_parallel_size
        self.n_local_kv_heads = self.n_kv_heads // model_parallel_size
        self.n_rep = self.n_local_heads // self.n_local_kv_heads
        self.head_dim = args.dim // args.n_heads
```

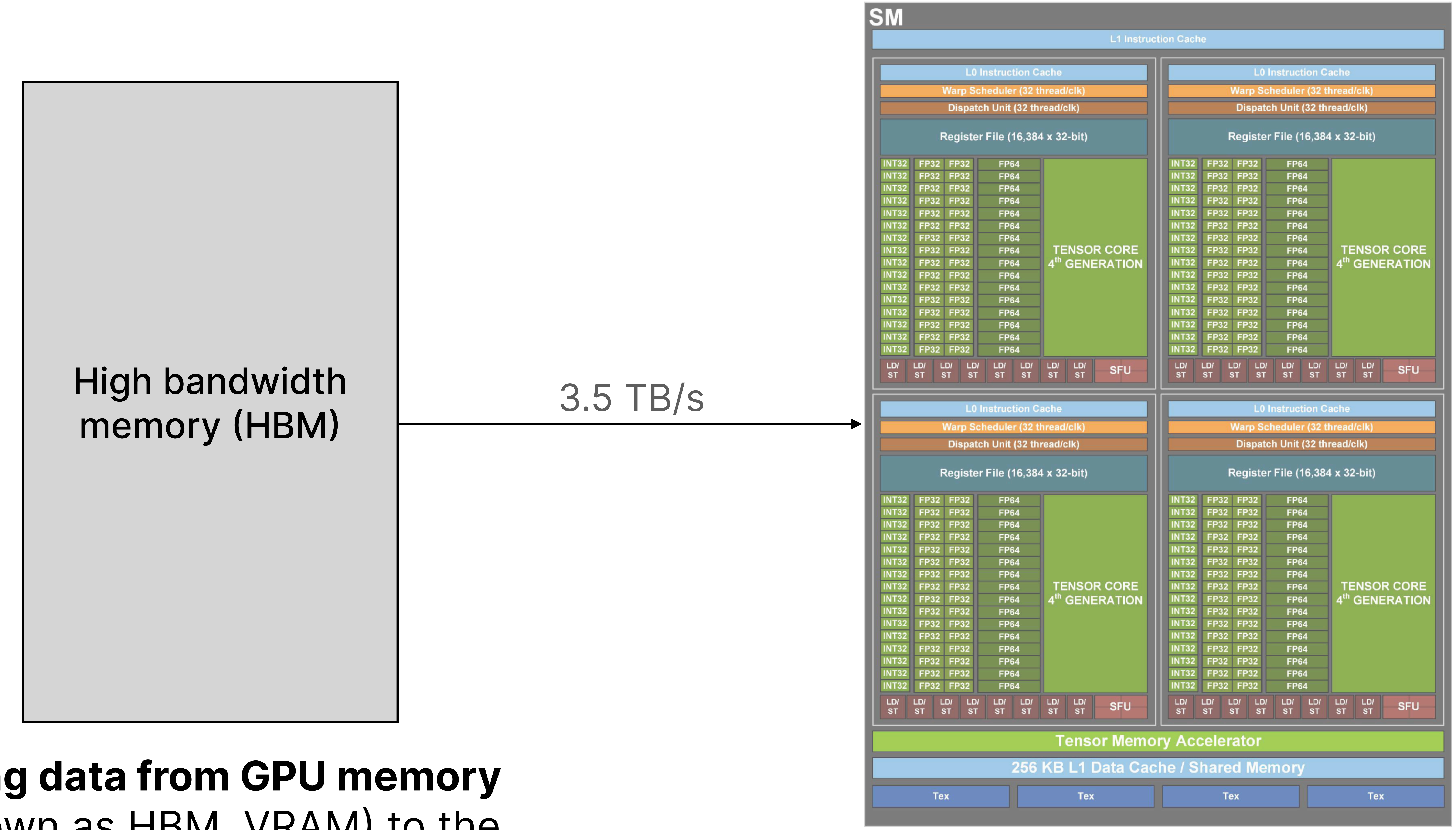
Training: modern architectures like the Mixture of Experts model and grouped query attention (e.g., LLaMA 2) are designed for low inference

Debugging and performance optimization: first principles can be both an important sanity check and roadmap to understand which optimizations are likely to work

UX: understanding how LLM inference work can drive the development of real time user applications

Multiprocessors just load data and do math

A multiprocessor spends time on two operations



A **Streaming Multiprocessor (SM)** in the NVIDIA H100 GPU, with four sub-cores

1. **Loading data from GPU memory** (also known as HBM, VRAM) to the computing unit's SRAM and registers at a specified bandwidth

Source: NVIDIA H100 Whitepaper, Making Deep Learning Go Brrrr From First Principles

A multiprocessor spends time on two operations

High bandwidth memory (HBM)

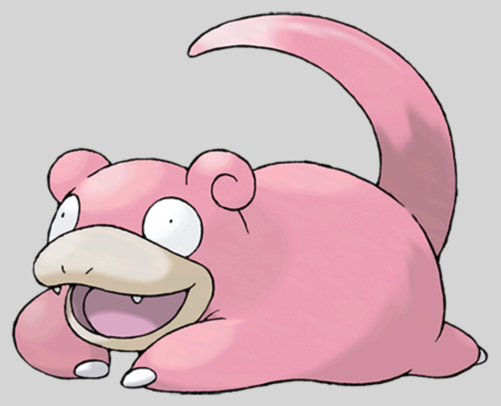
2. **Mathematical operations,** typically matrix-matrix or matrix-vector multiplies taking place in the tensor core

3.5 TB/s

A **Streaming Multiprocessor (SM)** in the NVIDIA H100 GPU, with four sub-cores

Source: NVIDIA H100 Whitepaper, Making Deep Learning Go Brrrr From First Principles

A job is said to be memory bandwidth bound if memory cannot supply work at a rate to keep the processor busy



High bandwidth memory (HBM)

[0.06, -0.01, 0.42, …]

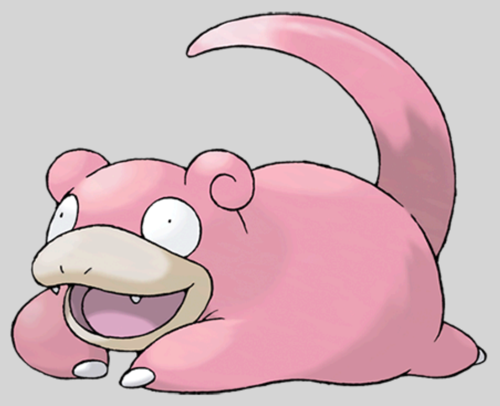
HBM can only send so many bytes/second (the bandwidth)

`F.relu(x) = max(0, x)`

Example: computing activation functions

Source: NVIDIA H100 Whitepaper, Making Deep Learning Go Brrrr From First Principles

A job is said to be memory bandwidth bound if memory cannot supply work at a rate to keep the processor busy



High bandwidth memory (HBM)

HBM can only send so many bytes/second (the bandwidth)

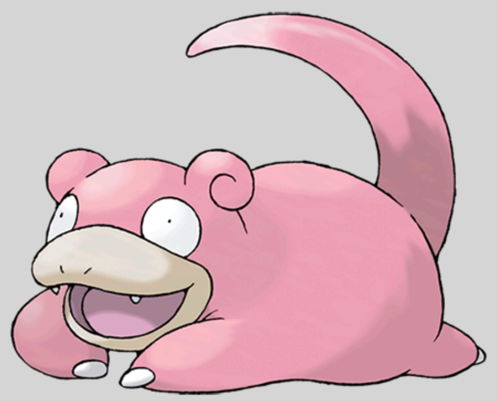
`F.relu(x) = max(0, x)`

`[0.06, −0.01, 0.42, …]`

Example: computing activation functions

Source: NVIDIA H100 Whitepaper, Making Deep Learning Go Brrrr From First Principles

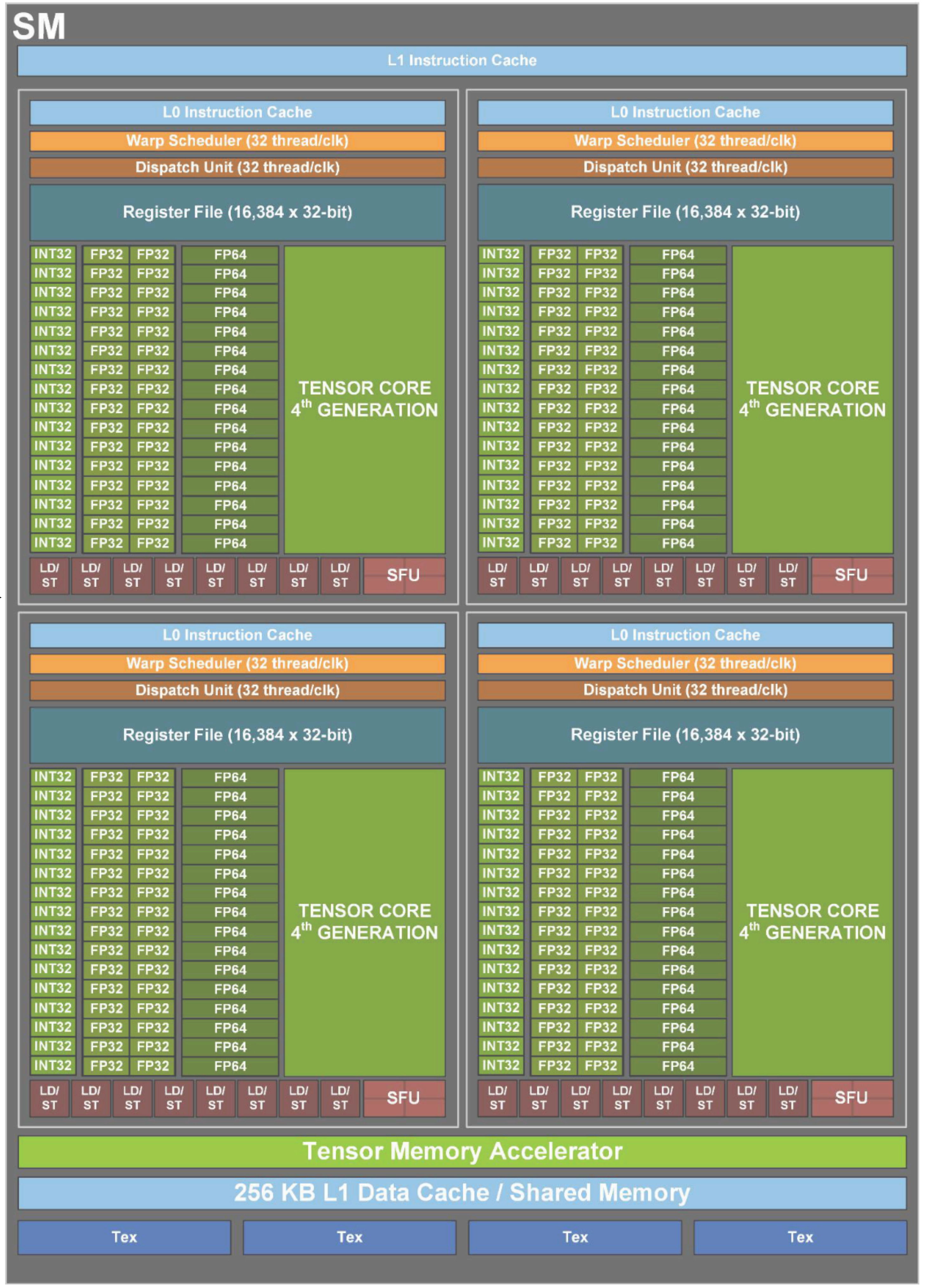
A job is said to be memory bandwidth bound if memory cannot supply work at a rate to keep the processor busy



High bandwidth memory (HBM)

`[0.09, -0.02, 0.54, …]`

HBM can only send so many bytes/second (the bandwidth)



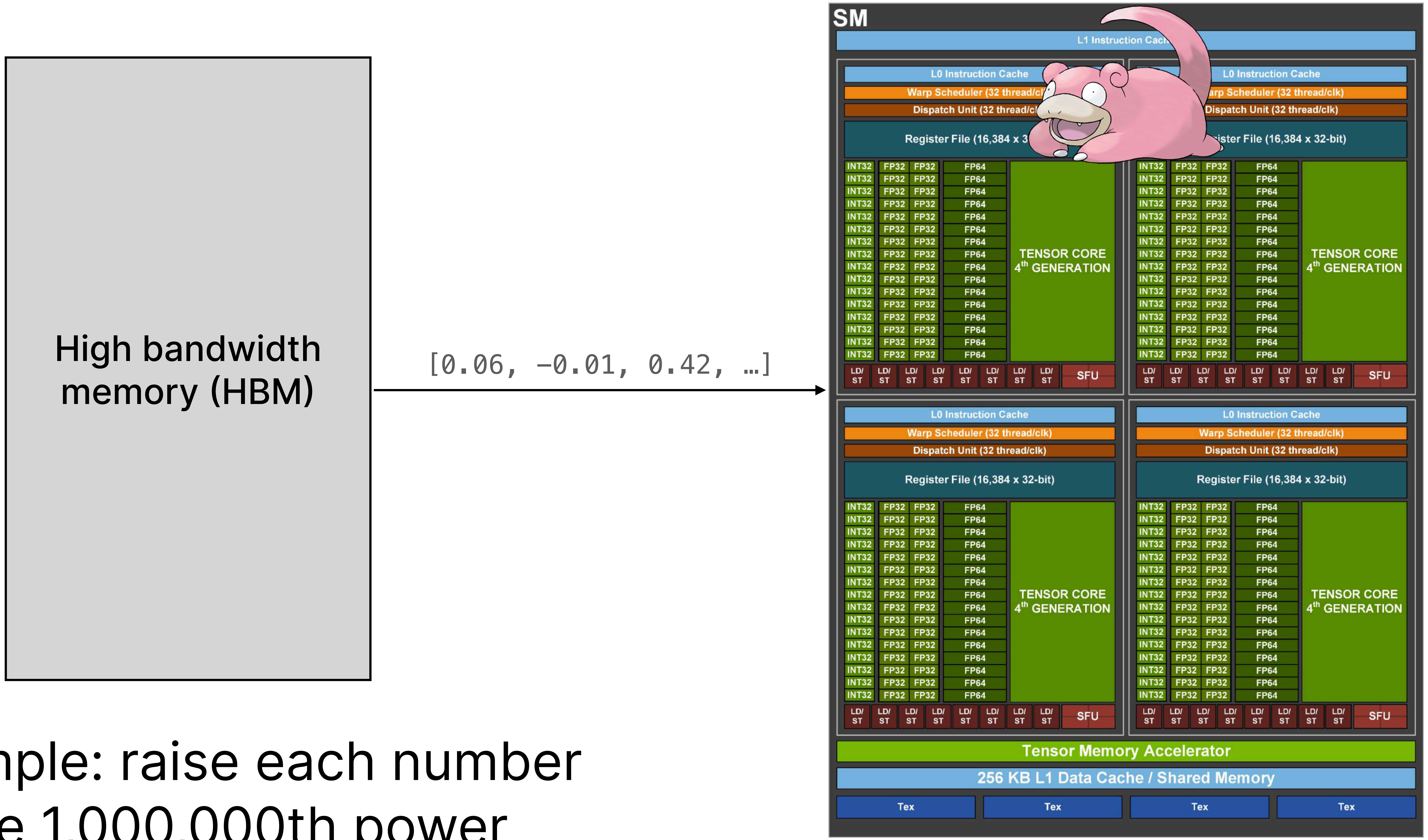
`F.relu(x) = max(0, x)`

Example: computing activation functions

Data has not arrived yet, GPU is idle!

Source: NVIDIA H100 Whitepaper, Making Deep Learning Go Brrrr From First Principles

A job is said to be compute bound if it is bottlenecked by the speed of the processor



Processor can only do so many floating point operations (FLOPs) every second

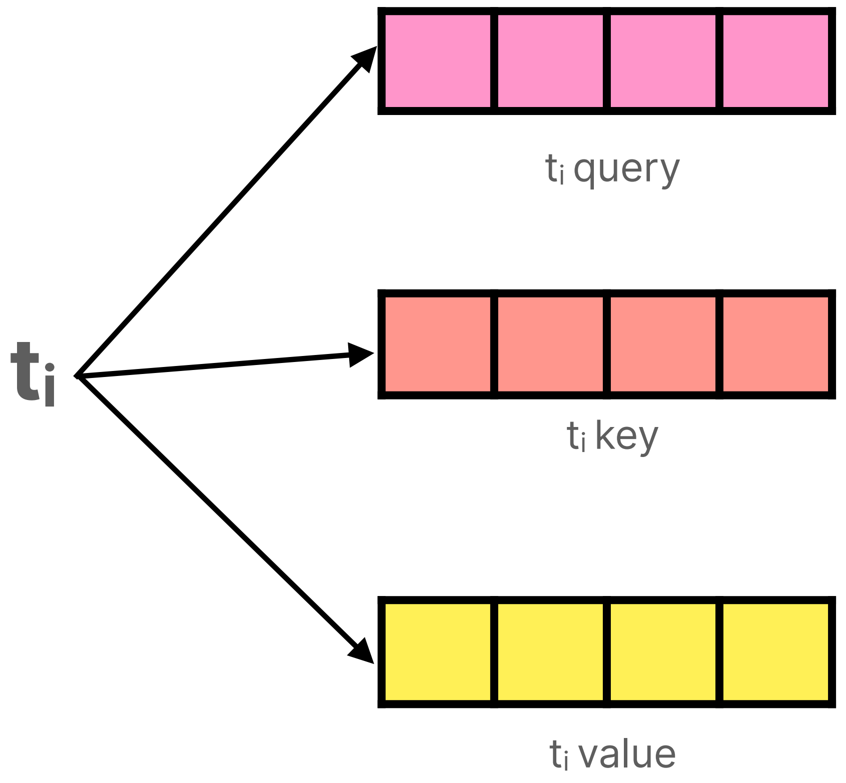
```
for _ in range(1000000):
  x *= x
```

Example: raise each number to the 1,000,000th power

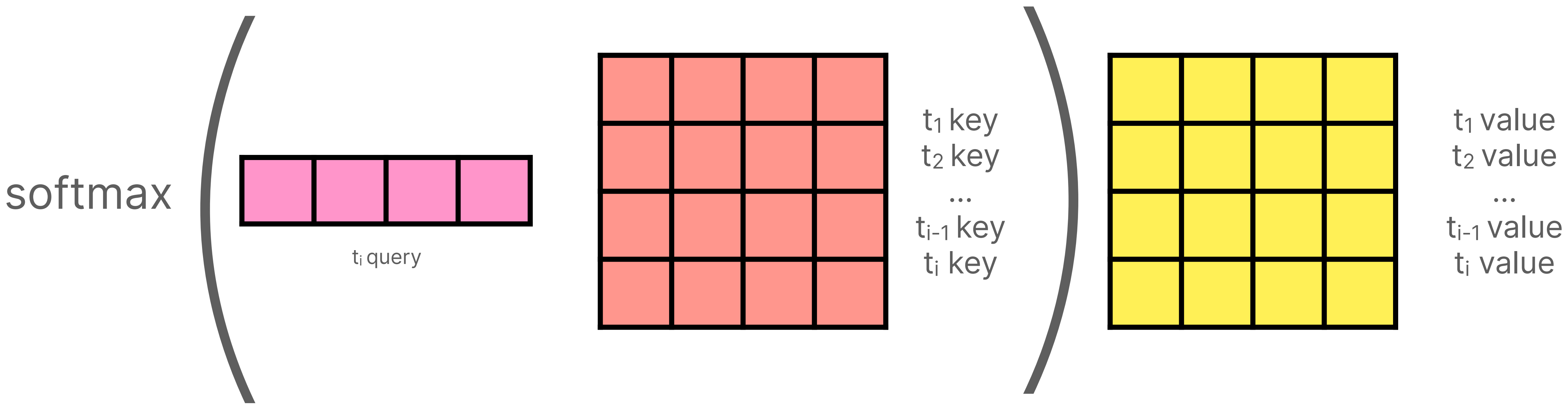
Source: NVIDIA H100 Whitepaper, Making Deep Learning Go Brrrr From First Principles

Inference: a two stage workload

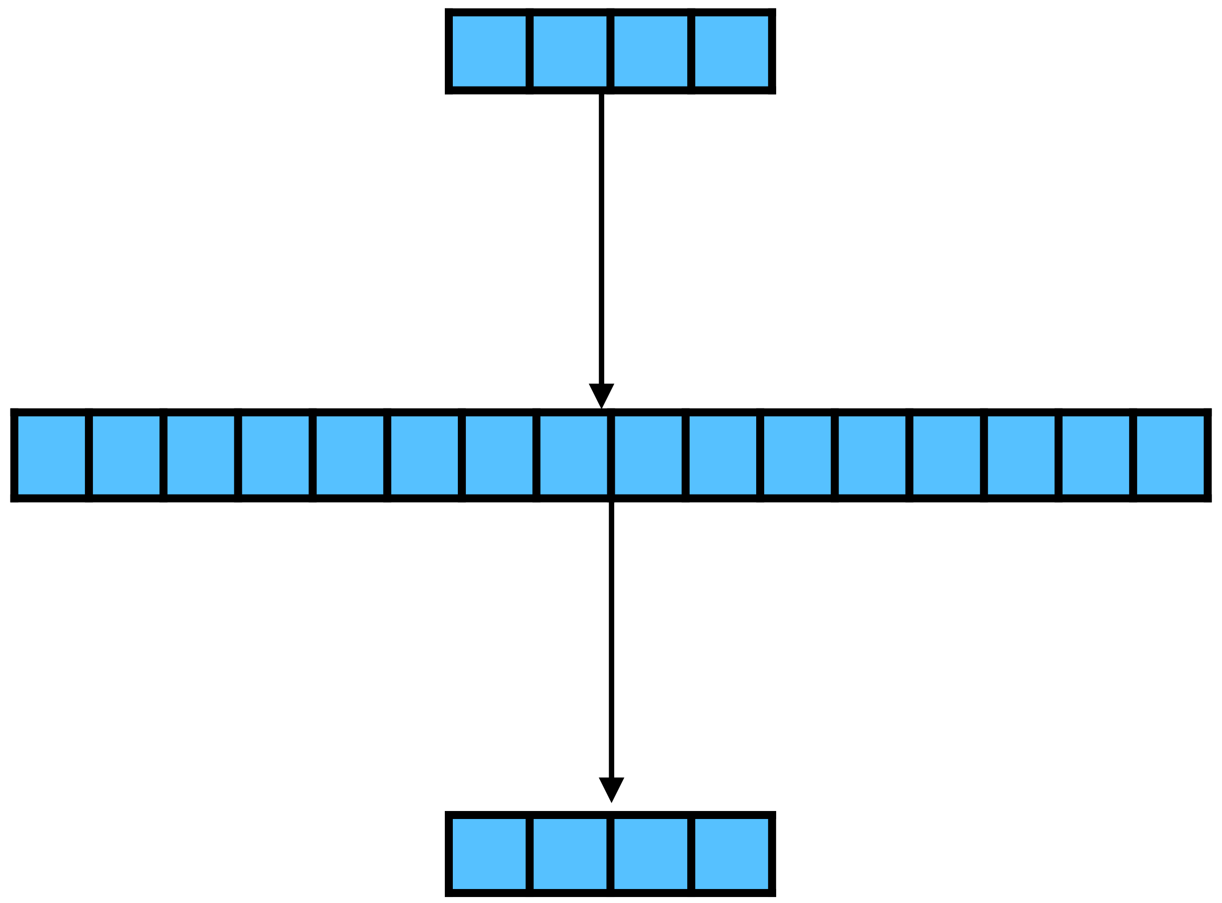
A transformer consists of the following block repeated many times, per token



Compute projections into query, key and value



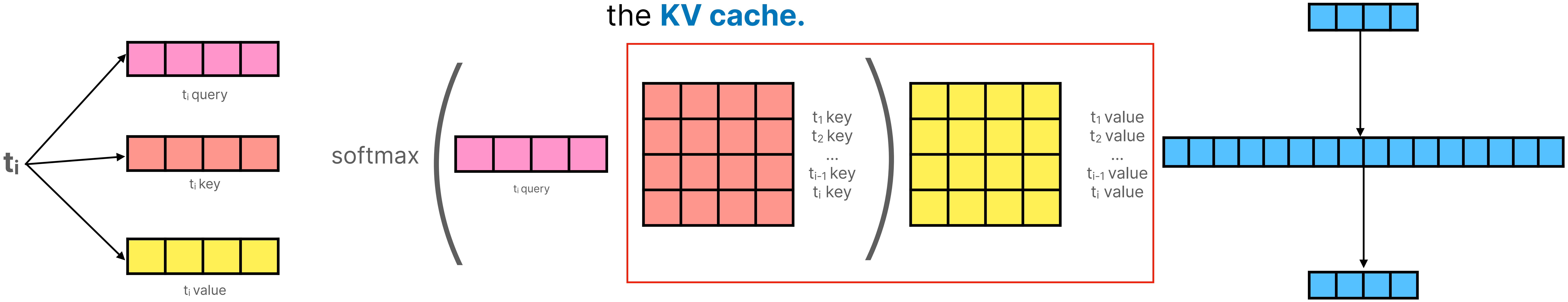
Compute the softmax with the query and all previous keys
Then multiply by all previous values



Project up once and then down project to the original dimension

A transformer consists of the following block repeated many times, per token

Keys and Values for all previous tokens will be reused. It’s a waste to recompute them, so store them in the **KV cache.**

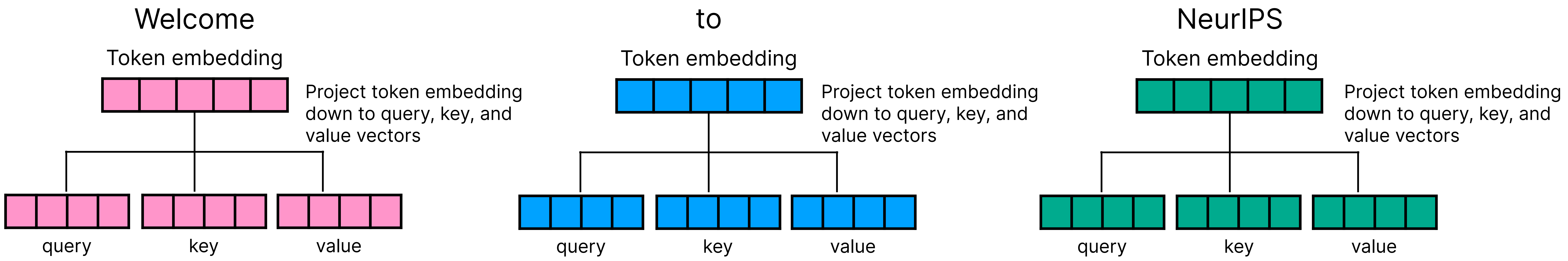


Compute projections into query, key and value

Compute the softmax with the query and all previous keys
Then multiply by all previous values

Project up once and then down project to the original dimension

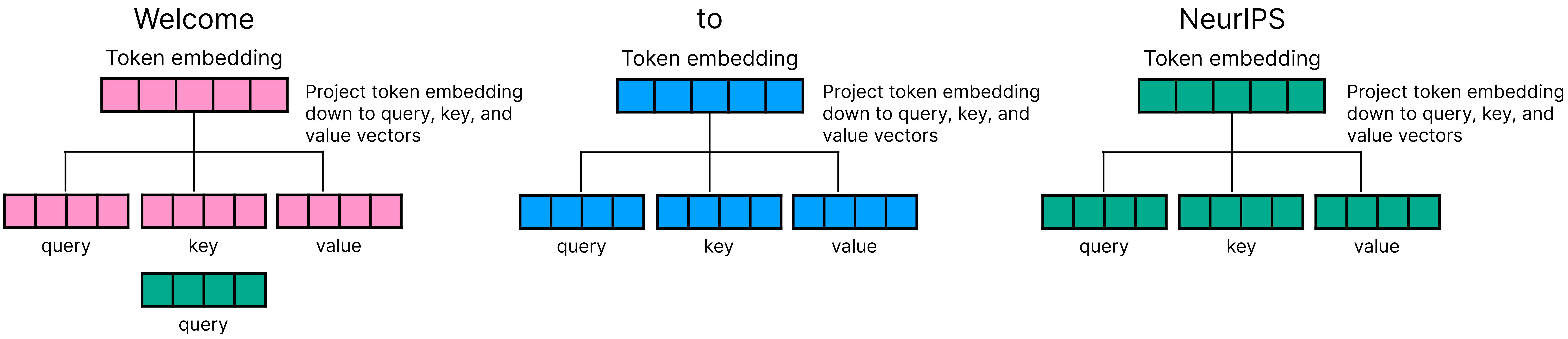
KV caching reduces the need for redundant computation



Attention requires computing the dot product of the current query with all previous keys (and later values)

Inspired by: [Generative LLM inference with Neuron]

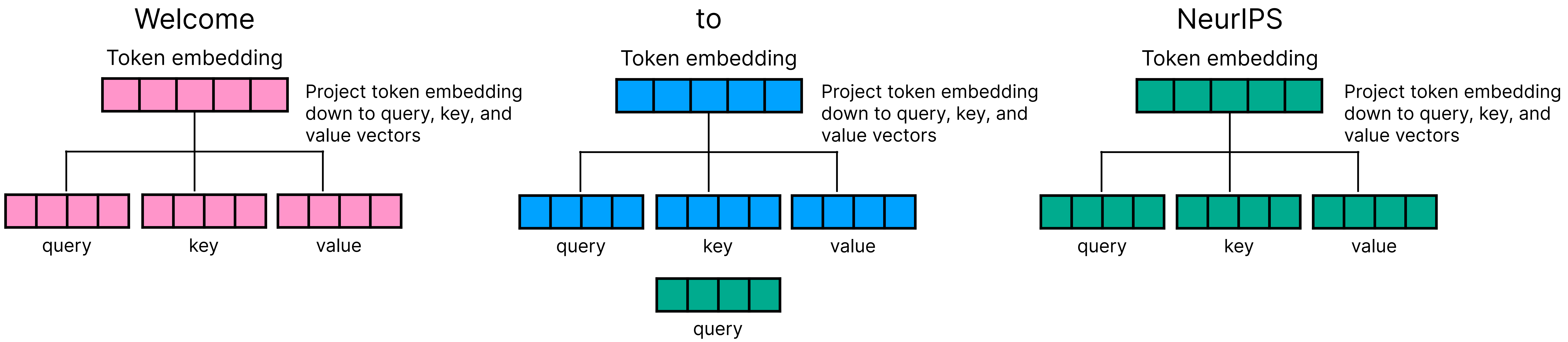
KV caching reduces the need for redundant computation



Attention requires computing the dot product of the current query with all previous keys (and later values)

Inspired by: Generative LLM inference with Neuron

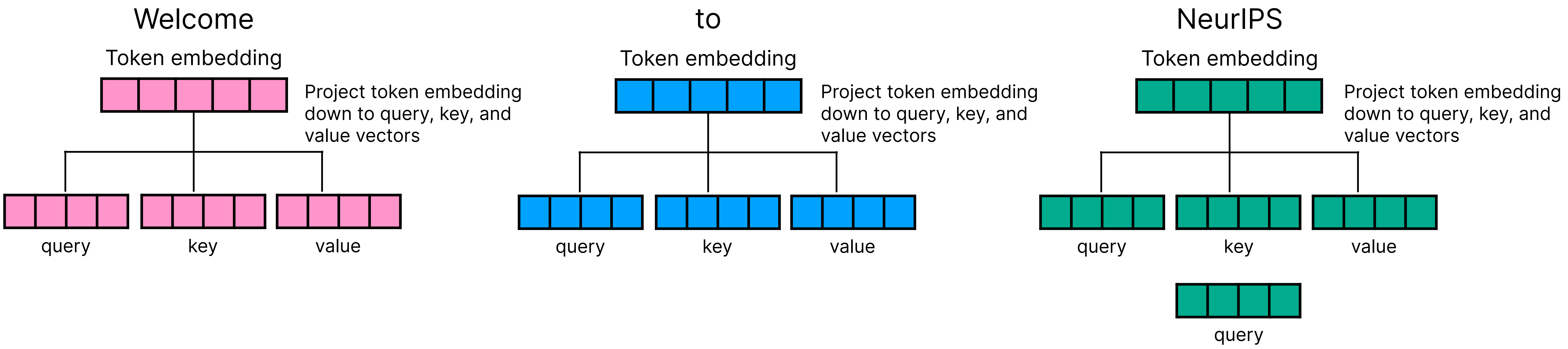
KV caching reduces the need for redundant computation



Attention requires computing the dot product of the current query with all previous keys (and later values)

Inspired by: [Generative LLM inference with Neuron]

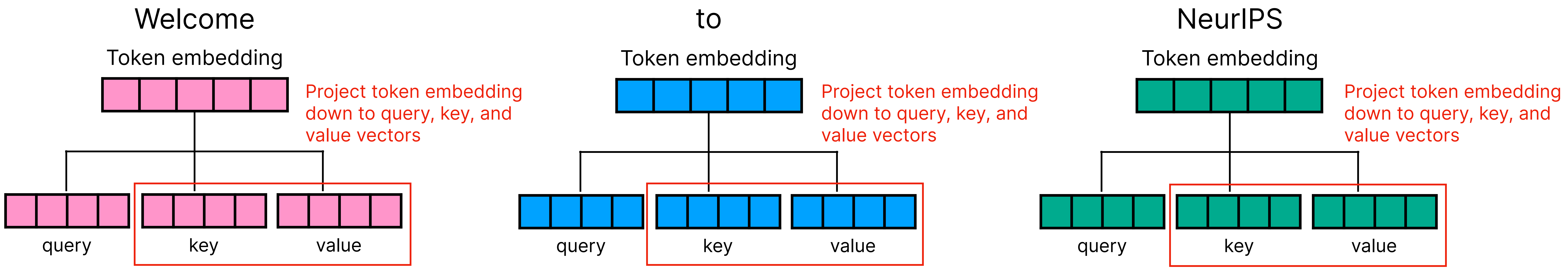
KV caching reduces the need for redundant computation



Attention requires computing the dot product of the current query with all previous keys (and later values)

Inspired by: Generative LLM inference with Neuron

KV caching reduces the need for redundant computation



These values never change, so there's no need to spend FLOPs recomputing them every time

Inspired by: Generative LLM inference with Neuron

`model.generate()` from HuggingFace

```
while True:
    # prepare model inputs
    model_inputs = self.prepare_inputs_for_generation(input_ids, **model_kwargs)

    # forward pass to get next token
    outputs = self(
        **model_inputs,
        return_dict=True,
        output_attentions=output_attentions,
        output_hidden_states=output_hidden_states,
    )

    next_token_logits = outputs.logits[:, -1, :]

    # pre-process distribution
    next_tokens_scores = logits_processor(input_ids, next_token_logits)

    next_tokens = torch.argmax(next_tokens_scores, dim=-1)

    # finished sentences should have their next token be a padding token
    if eos_token_id is not None:
        if pad_token_id is None:
            raise ValueError("If `eos_token_id` is defined, make sure that `pad_token_id` is defined.")
        next_tokens = next_tokens * unfinished_sequences + pad_token_id * (1 - unfinished_sequences)

    # update generated ids, model inputs, and length for next step
    input_ids = torch.cat([input_ids, next_tokens[:, None]], dim=-1)


    # if eos_token was found in one sentence, set sentence to finished
    if eos_token_id_tensor is not None:
        unfinished_sequences = unfinished_sequences.mul(
            next_tokens.tile(eos_token_id_tensor.shape[0], 1).ne(eos_token_id_tensor.unsqueeze(1)).prod(dim=0)
        )

        # stop when each sentence is finished
        if unfinished_sequences.max() == 0:
            this_peer_finished = True

    # stop if we exceed the maximum length
    if stopping_criteria(input_ids, scores):
        break

if streamer is not None:
    streamer.end()
```

Prefill stage processes each token in parallel

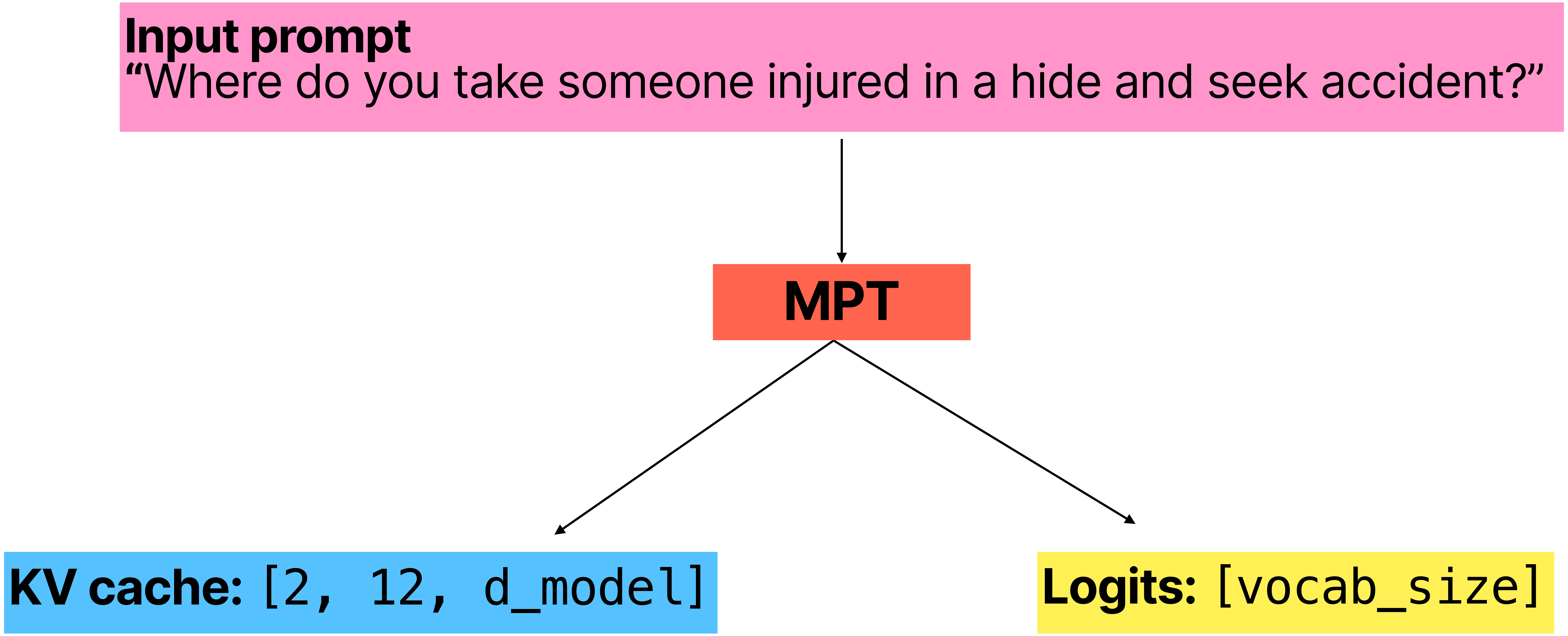
```
while True:
    # prepare model inputs
    model_inputs = self.prepare_inputs_for_generation(input_ids, **model_kwargs)

    # forward pass to get next token
    outputs = self(
        **model_inputs,
        return_dict=True,
        output_attentions=output_attentions,
        output_hidden_states=output_hidden_states,
    )

    next_token_logits = outputs.logits[:, -1, :]

    # pre-process distribution
    next_tokens_scores = logits_processor(input_ids, next_token_logits)

    next_tokens = torch.argmax(next_tokens_scores, dim=-1)

    # finished sentences should have their next token be a padding token
    if eos_token_id is not None:
        if pad_token_id is None:
            raise ValueError("If `eos_token_id` is defined, make sure that `pad_token_id` is defined.")
        next_tokens = next_tokens * unfinished_sequences + pad_token_id * (1 - unfinished_sequences)

    # update generated ids, model inputs, and length for next step
    input_ids = torch.cat([input_ids, next_tokens[:, None]], dim=-1)


    # if eos_token was found in one sentence, set sentence to finished
    if eos_token_id_tensor is not None:
        unfinished_sequences = unfinished_sequences.mul(
            next_tokens.tile(eos_token_id_tensor.shape[0], 1).ne(eos_token_id_tensor.unsqueeze(1)).prod(dim=0)
        )

        # stop when each sentence is finished
        if unfinished_sequences.max() == 0:
            this_peer_finished = True

    # stop if we exceed the maximum length
    if stopping_criteria(input_ids, scores):
        break

if streamer is not None:
    streamer.end()
```

All tokens are present during prefill, so we can process all tokens in the sequence **in parallel**

- Populate KV cache
- Generate probability for the first generated token

Source: Efficiently Scaling Transformer Inference, Pope et al 2022

Prefill stage processes each token in parallel



Prefill stage processes each token in parallel

Input prompt
"Where do you take someone injured in a hide and seek accident?"

Only 1 forward pass needed to process all input tokens in parallel

MPT

KV cache: `[2, 12, d_model]`

Logits: `[vocab_size]`

1 total forward pass ran

Decoding stage processes each token one by one

```
while True:
    # prepare model inputs
    model_inputs = self.prepare_inputs_for_generation(input_ids, **model_kwargs)

    # forward pass to get next token
    outputs = self(
        **model_inputs,
        return_dict=True,
        output_attentions=output_attentions,
        output_hidden_states=output_hidden_states,
    )

    next_token_logits = outputs.logits[:, -1, :]

    # pre-process distribution
    next_tokens_scores = logits_processor(input_ids, next_token_logits)

    next_tokens = torch.argmax(next_tokens_scores, dim=-1)

    # finished sentences should have their next token be a padding token
    if eos_token_id is not None:
        if pad_token_id is None:
            raise ValueError("If `eos_token_id` is defined, make sure that `pad_token_id` is defined.")
        next_tokens = next_tokens * unfinished_sequences + pad_token_id * (1 - unfinished_sequences)

    # update generated ids, model inputs, and length for next step
    input_ids = torch.cat([input_ids, next_tokens[:, None]], dim=-1)

    # if eos_token was found in one sentence, set sentence to finished
    if eos_token_id_tensor is not None:
        unfinished_sequences = unfinished_sequences.mul(
            next_tokens.tile(eos_token_id_tensor.shape[0], 1).ne(eos_token_id_tensor.unsqueeze(1)).prod(dim=0)
        )

        # stop when each sentence is finished
        if unfinished_sequences.max() == 0:
            this_peer_finished = True

    # stop if we exceed the maximum length
    if stopping_criteria(input_ids, scores):
        break

if streamer is not None:
    streamer.end()
```

New token is appended to the input

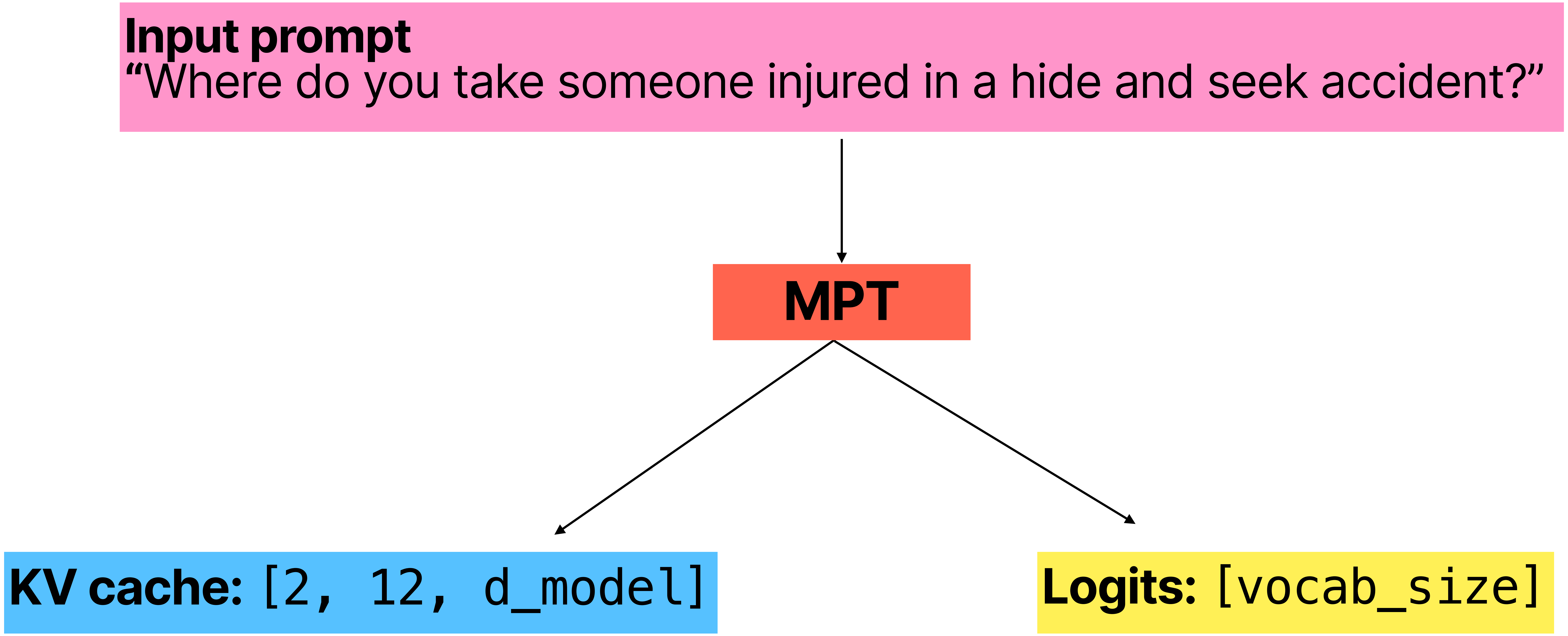
Decoding stage processes each token one by one

```
while True:
    # prepare model inputs
    model_inputs = self.prepare_inputs_for_generation(input_ids, **model_kwargs)

    # forward pass to get next token
    outputs = self(
        **model_inputs,
        return_dict=True,
        output_attentions=output_attentions,
        output_hidden_states=output_hidden_states,
    )

    next_token_logits = outputs.logits[:, -1, :]

    # pre-process distribution
    next_tokens_scores = logits_processor(input_ids, next_token_logits)

    next_tokens = torch.argmax(next_tokens_scores, dim=-1)

    # finished sentences should have their next token be a padding token
    if eos_token_id is not None:
        if pad_token_id is None:
            raise ValueError("If `eos_token_id` is defined, make sure that `pad_token_id` is defined.")
        next_tokens = next_tokens * unfinished_sequences + pad_token_id * (1 - unfinished_sequences)

    # update generated ids, model inputs, and length for next step
    input_ids = torch.cat([input_ids, next_tokens[:, None]], dim=-1)


    # if eos_token was found in one sentence, set sentence to finished
    if eos_token_id_tensor is not None:
        unfinished_sequences = unfinished_sequences.mul(
            next_tokens.tile(eos_token_id_tensor.shape[0], 1).ne(eos_token_id_tensor.unsqueeze(1)).prod(dim=0)
        )

        # stop when each sentence is finished
        if unfinished_sequences.max() == 0:
            this_peer_finished = True

    # stop if we exceed the maximum length
    if stopping_criteria(input_ids, scores):
        break

if streamer is not None:
    streamer.end()
```

New token is appended to the input

The forward pass is ran again

Decoding stage processes each token one by one



1 total forward pass ran

Decoding stage processes each token one by one

Input prompt
“Where do you take someone injured in a hide and seek accident?”
You

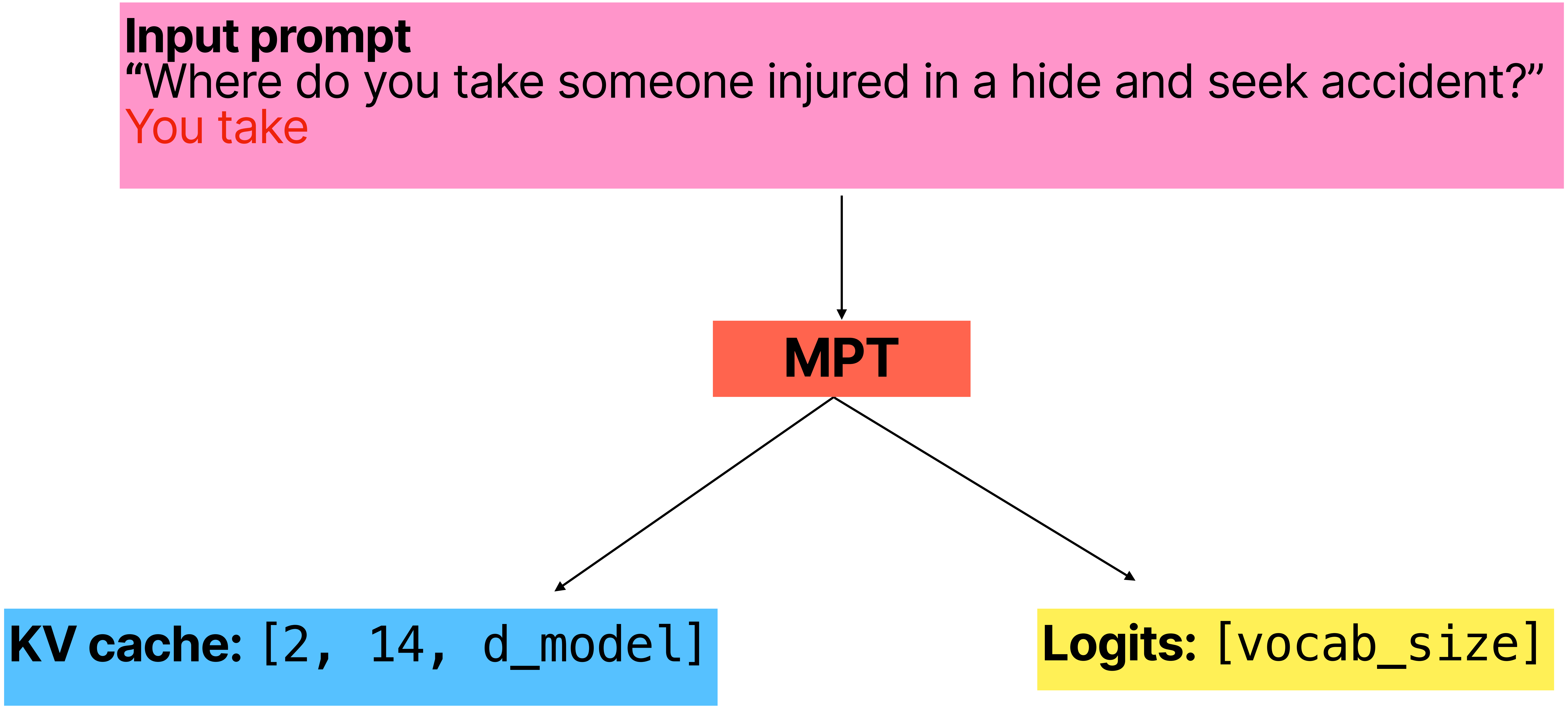
MPT

KV cache: `[2, 13, d_model]`

Logits: `[vocab_size]`

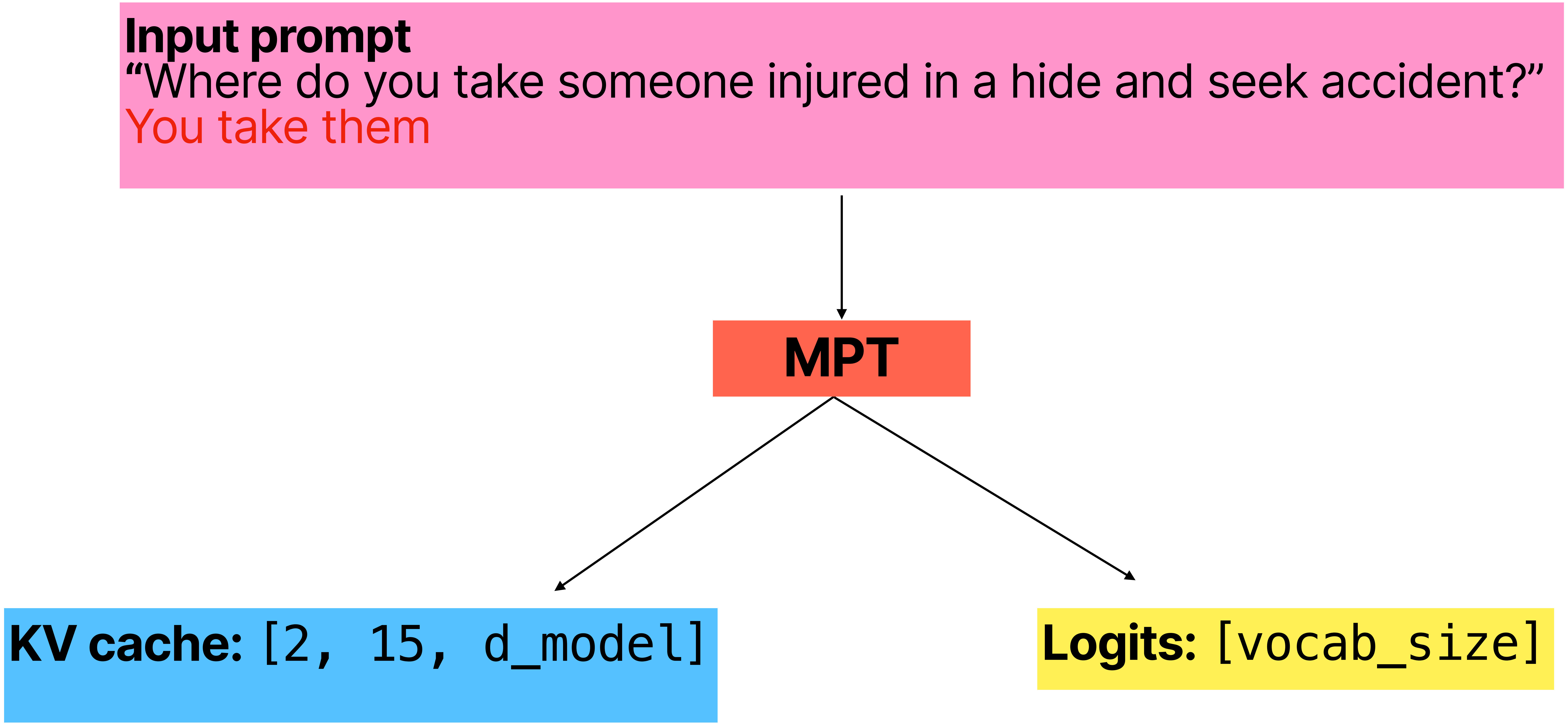
2 total forward passes ran

Decoding stage processes each token one by one



3 total forward passes ran

Decoding stage processes each token one by one



4 total forward passes ran

Decoding stage processes each token one by one

Input prompt
“Where do you take someone injured in a hide and seek accident?”
You take them to

MPT

KV cache: `[2, 16, d_model]`

Logits: `[vocab_size]`

5 total forward passes ran

Decoding stage processes each token one by one

Input prompt
“Where do you take someone injured in a hide and seek accident?”
You take them to the

MPT

KV cache: `[2, 17, d_model]`

Logits: `[vocab_size]`

6 total forward passes ran

Decoding stage processes each token one by one

Input prompt
"Where do you take someone injured in a hide and seek accident?"
You take them to the I.C.U.

MPT

KV cache: `[2, 18, d_model]`

Logits: `[vocab_size]`

7 total forward passes ran

Decoding stage processes each token one by one

Input prompt
"Where do you take someone injured in a hide and seek accident?"
You take them to the I.C.U. 🥁

MPT

KV cache: `[2, 19, d_model]`

Logits: `[vocab_size]`

8 total forward passes ran

Decoding stage processes each token one by one

Input prompt
"Where do you take someone injured in a hide and seek accident?"
You take them to the I.C.U. 🥁

Every single generated token triggered a forward pass through the model

MPT

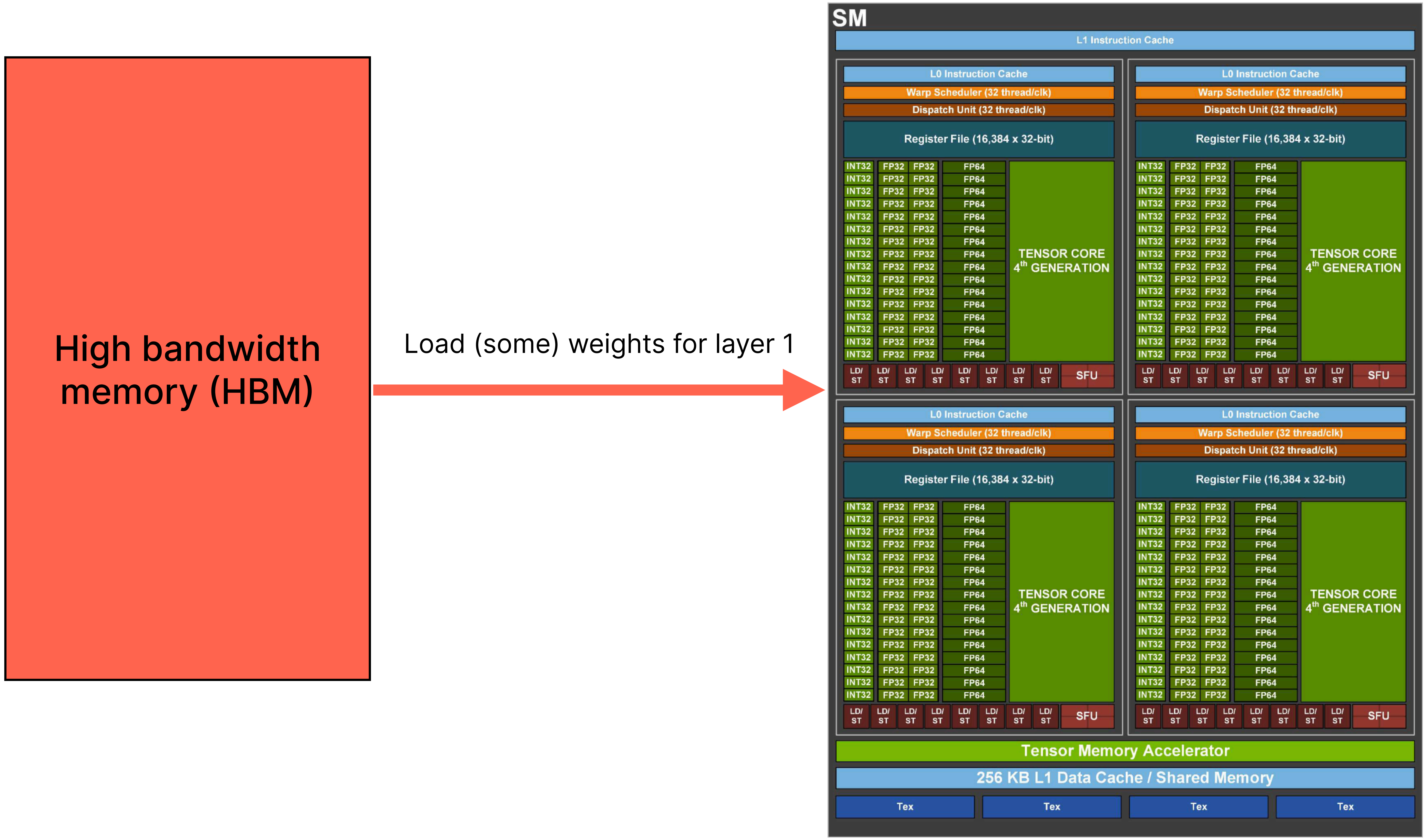
KV cache: `[2, 19, d_model]`

Logits: `[vocab_size]`

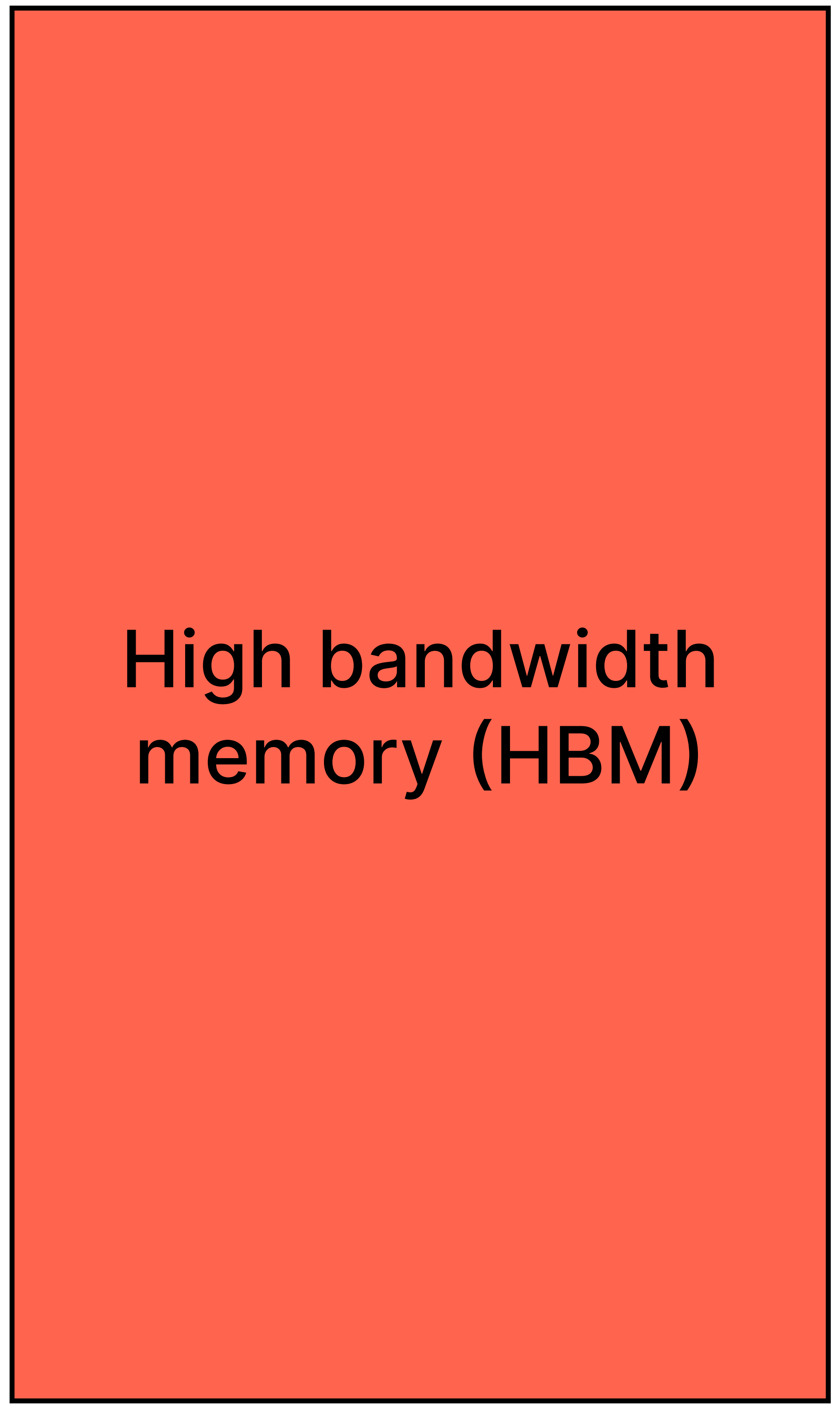
8 total forward passes ran

A forward pass involves moving the weights from HBM to registers on the device

A forward pass involves moving the weights from HBM to registers on the device



A forward pass involves moving the weights from HBM to registers on the device



This matters, since the rate at which bandwidth has been increasing is a lot slower than processor speeds

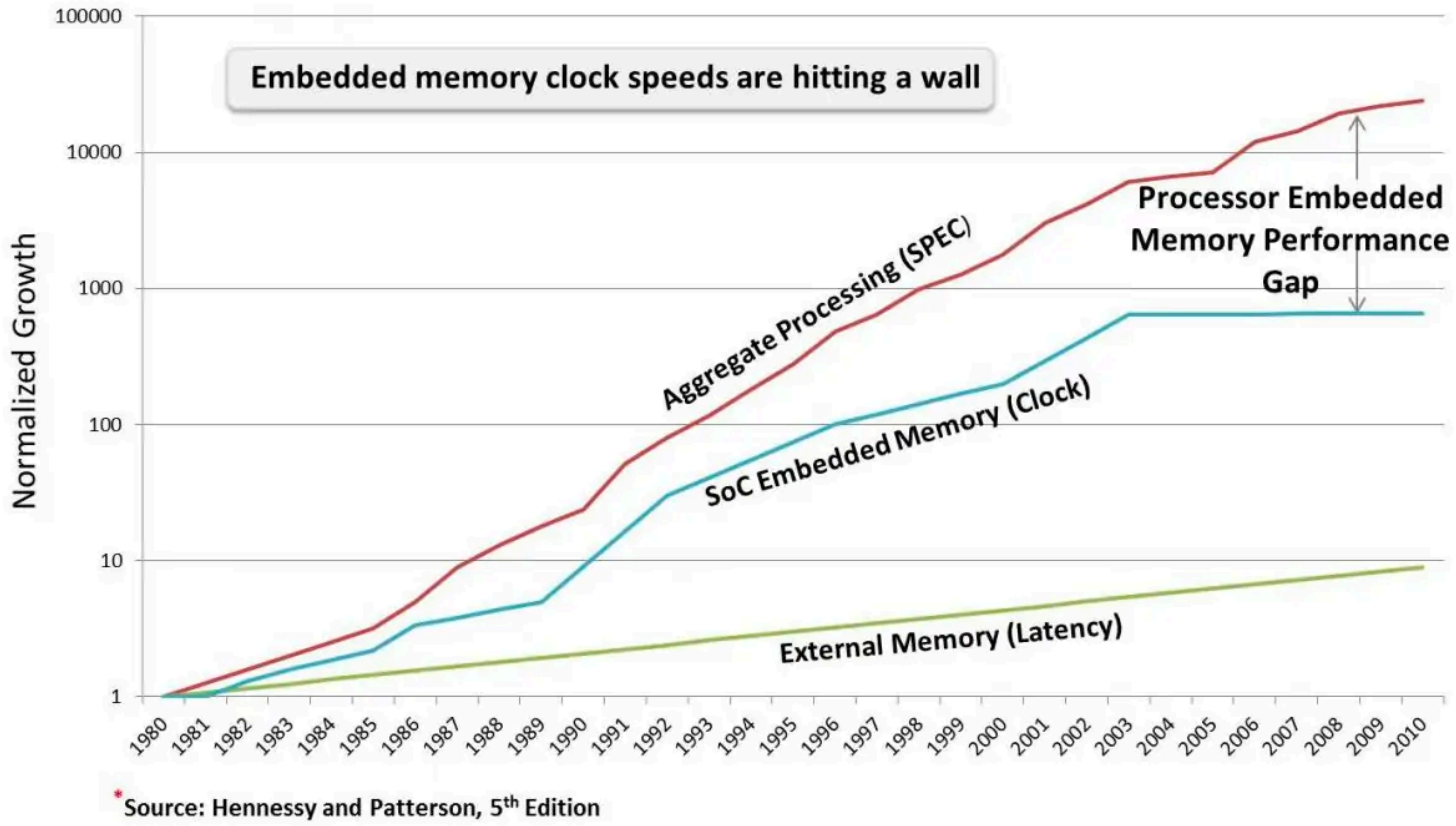


Figure 1: Embedded Memory Performance Gap is Getting Worse

Prefill and decode end up having extremely different characteristics

Prefill loads the model **once** from memory to process all input tokens in parallel

Decode loads the model up to `max_new_tokens` times, once for **every single token generated.** It only processes a *single token*.

Prefill and decode end up having extremely different characteristics

Prefill loads the model **once** from memory to process all input tokens in parallel

Compute bound
High number of operations per byte read

Decode loads the model up to `max_new_tokens` times, once for **every single token generated.** It only processes a *single token.*

Memory bound
Low number of operations per byte read

Serving large models

Memory consists of parameters and the KV cache

Let's say we're serving a Llama2-70B model with:

- `precision = fp16`
- `d_model = 8192`
- `n_layers = 80`
- `batch_size = 4`
- `input_seq_len = 1024`
- `max_new_tokens = 32`
- `head_size = 128`
- `kv_n_heads = 64`

Model size

70e9 params * 2 bytes/param = 140e9 bytes = 140GB

KV cache size

80 * 2 * 2 bytes/param * 64 * 128 * 4 * (1024 + 32) ~= 11e9 bytes = 11 GB

Inspired by: Transformer Inference Arithmetic from kipply's blog

Memory consists of parameters and the KV cache

Let's say we're serving a Llama2-70B model with:

- `precision = fp16`
- `d_model = 8192`
- `n_layers = 80`
- `batch_size = 4`
- `input_seq_len = 1024`
- `max_new_tokens = 32`
- `head_size = 128`
- `kv_n_heads = 64`

Model size

70e9 params * 2 bytes/param = 140e9 bytes = 140GB

KV cache size

80 * 2 * 2 bytes/param * 64 * 128 * 4 * $\boxed{(1024 + 32)}$
~= 11e9 bytes = 11 GB

The workload has this many tokens

Inspired by: Transformer Inference Arithmetic from kipply's blog

Memory consists of parameters and the KV cache

Let's say we're serving a Llama2-70B model with:

- `precision = fp16`
- `d_model = 8192`
- `n_layers = 80`
- `batch_size = 4`
- `input_seq_len = 1024`
- `max_new_tokens = 32`
- `head_size = 128`
- `kv_n_heads = 64`

Model size

70e9 params * 2 bytes/param = 140e9 bytes = 140GB

KV cache size

$80 * 2 * 2 \text{ bytes/param} * 64 * \boxed{128 * 4 * (1024 + 32)}$
~= 11e9 bytes = 11 GB

Each token has a head of size 128

Inspired by: Transformer Inference Arithmetic from kipply's blog

Memory consists of parameters and the KV cache

Let's say we're serving a Llama2-70B model with:

- `precision = fp16`
- `d_model = 8192`
- `n_layers = 80`
- `batch_size = 4`
- `input_seq_len = 1024`
- `max_new_tokens = 32`
- `head_size = 128`
- `kv_n_heads = 64`

Model size

70e9 params * 2 bytes/param = 140e9 bytes = 140GB

KV cache size

The attention heads are concatenated, and there are 64 heads

80 * 2 * 2 bytes/param * $\boxed{64 * 128 * 4 * (1024 + 32)}$
~= 11e9 bytes = 11 GB

Inspired by: Transformer Inference Arithmetic from kipply's blog

Memory consists of parameters and the KV cache

Let's say we're serving a Llama2-70B model with:

- `precision = fp16`
- `d_model = 8192`
- `n_layers = 80`
- `batch_size = 4`
- `input_seq_len = 1024`
- `max_new_tokens = 32`
- `head_size = 128`
- `kv_n_heads = 64`

Model size

70e9 params * 2 bytes/param = 140e9 bytes = 140GB

KV cache size

We need a 2 because there's 1 key and 1 value

$80 * \boxed{2 * 2 \text{ bytes/param} * 64 * 128 * 4 * (1024 + 32)}$
~= 11e9 bytes = 11 GB

Inspired by: Transformer Inference Arithmetic from kipply's blog

Memory consists of parameters and the KV cache

Let's say we're serving a Llama2-70B model with:

- `precision = fp16`
- `d_model = 8192`
- `n_layers = 80`
- `batch_size = 4`
- `input_seq_len = 1024`
- `max_new_tokens = 32`
- `head_size = 128`
- `kv_n_heads = 64`

Model size

70e9 params * 2 bytes/param = 140e9 bytes = 140GB

KV cache size

There are 80 layers

80 * 2 * 2 bytes/param * 64 * 128 * 4 * (1024 + 32)
~= 11e9 bytes = 11 GB

Inspired by: Transformer Inference Arithmetic from kipply's blog

Memory consists of parameters and the KV cache

Let's say we're serving a Llama2-70B model with:

- `precision = fp16`
- `d_model = 8192`
- `n_layers = 80`
- `batch_size = 4`
- `input_seq_len = 1024`
- `max_new_tokens = 32`
- `head_size = 128`
- `kv_n_heads = 64`

Model size

70e9 params * 2 bytes/param = 140e9 bytes = 140GB

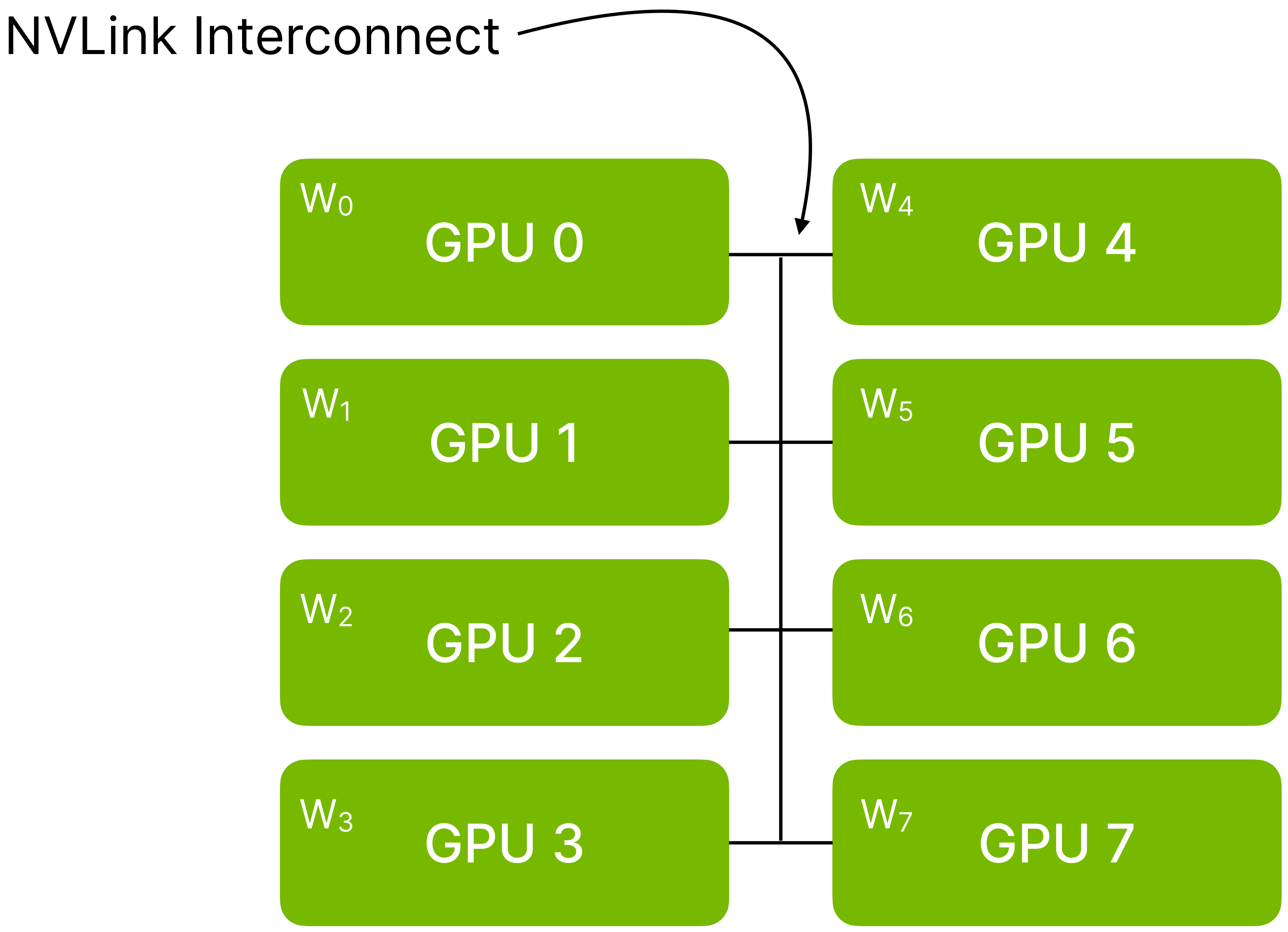
KV cache size

80 * 2 * 2 bytes/param * 64 * 128 * 4 * (1024 + 32) ~= 11e9 bytes = 11 GB

140 GB + 11 GB = 151 GB >> 80 GB

Inspired by: Transformer Inference Arithmetic from kipply's blog

To reduce memory, we use tensor parallelism



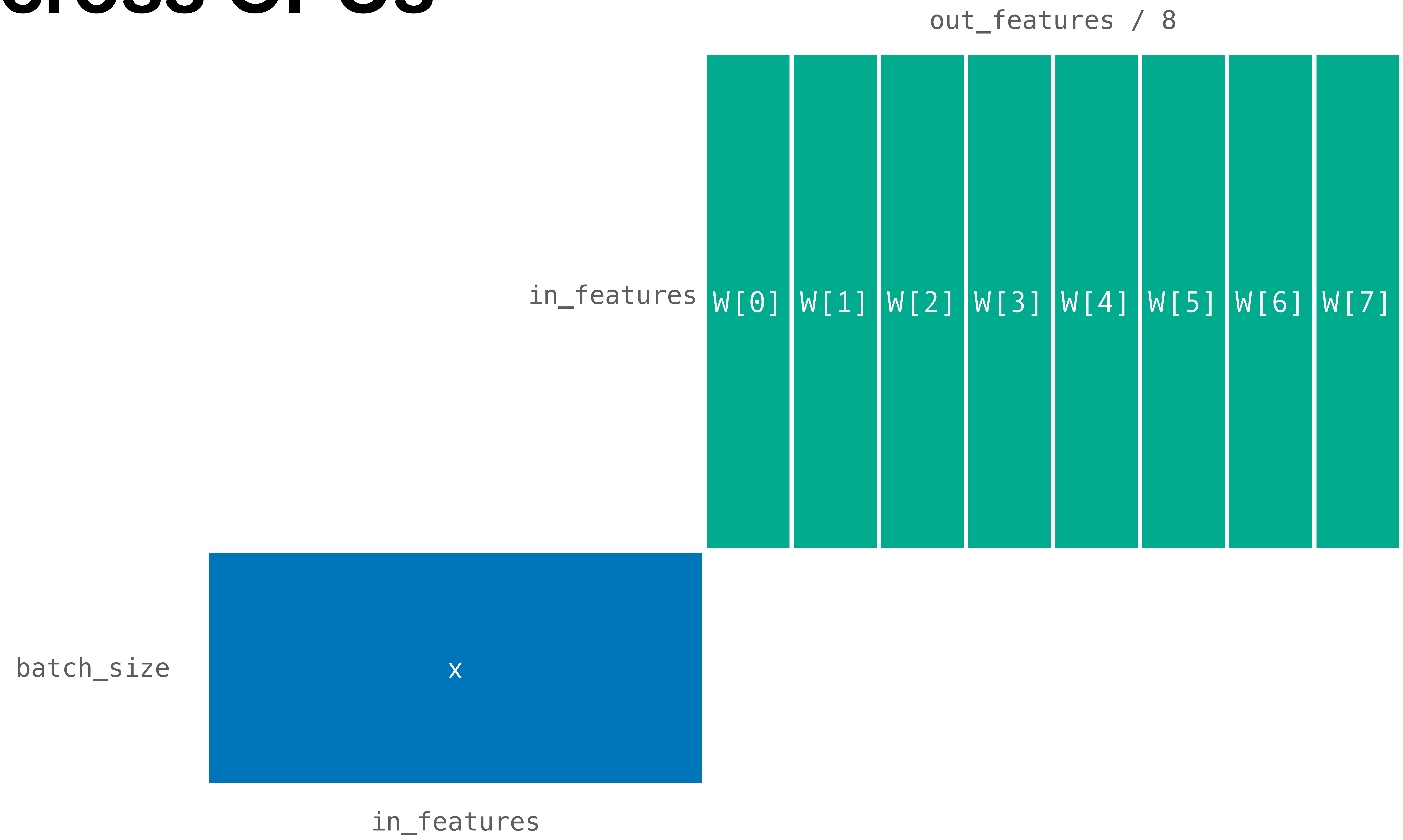
Weights take up lots of memory, so shard them across GPUs

Source: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, Shoeybi et al, 2020

Column parallel shards the output dimension across GPUs

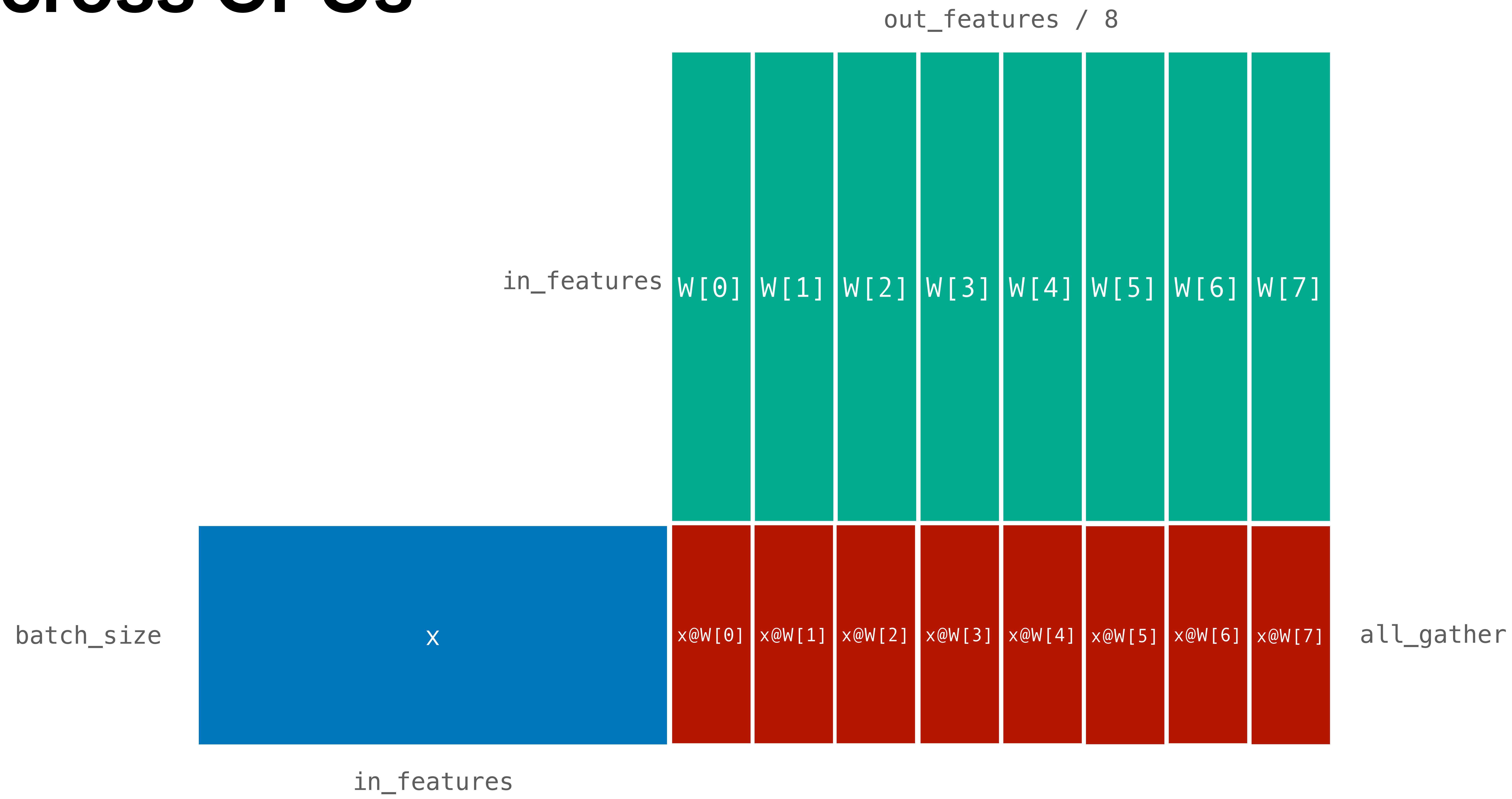
Source: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, Shoeybi et al, 2020

Column parallel shards the output dimension across GPUs



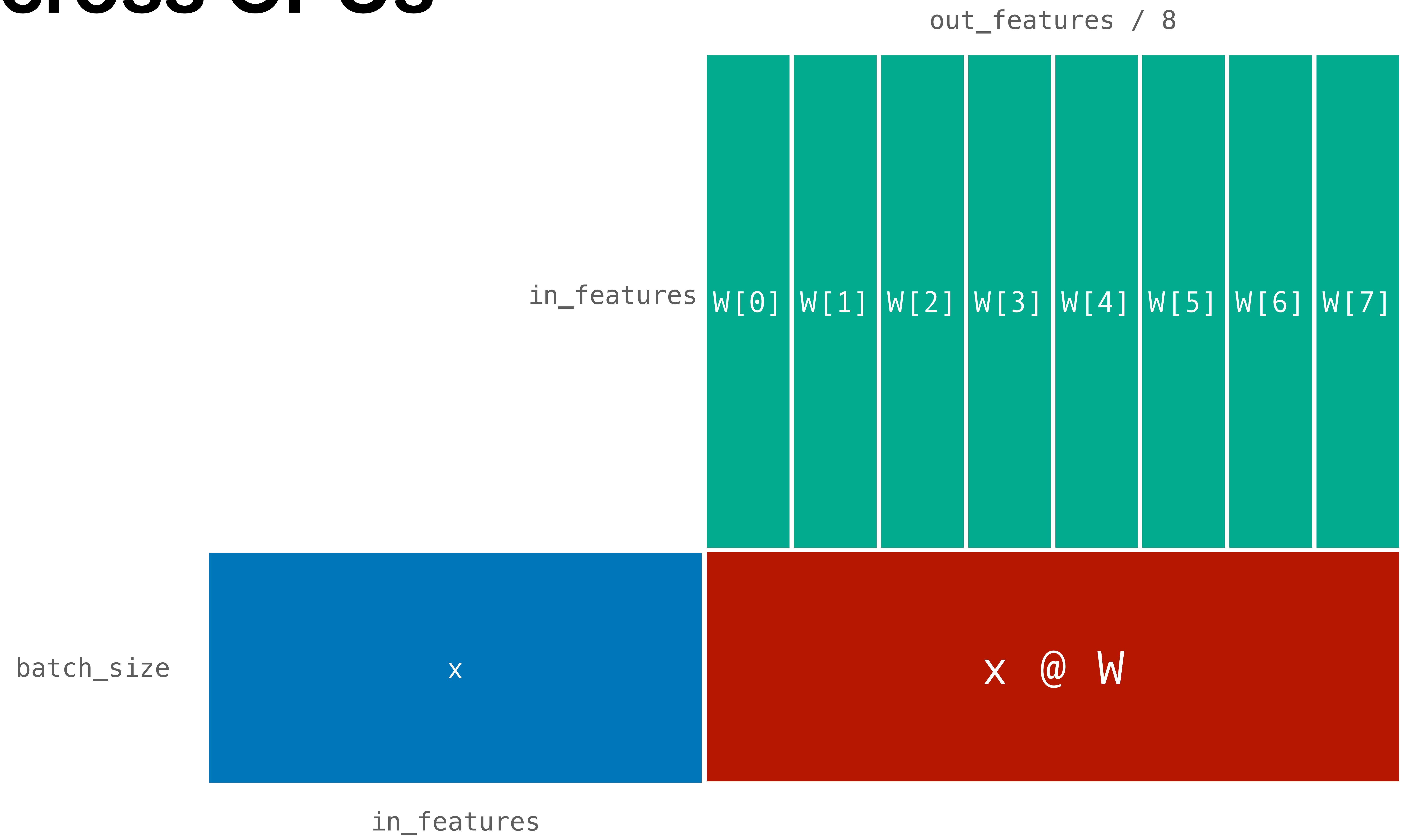
Source: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, Shoeybi et al, 2020

Column parallel shards the output dimension across GPUs



Source: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, Shoeybi et al, 2020

Column parallel shards the output dimension across GPUs

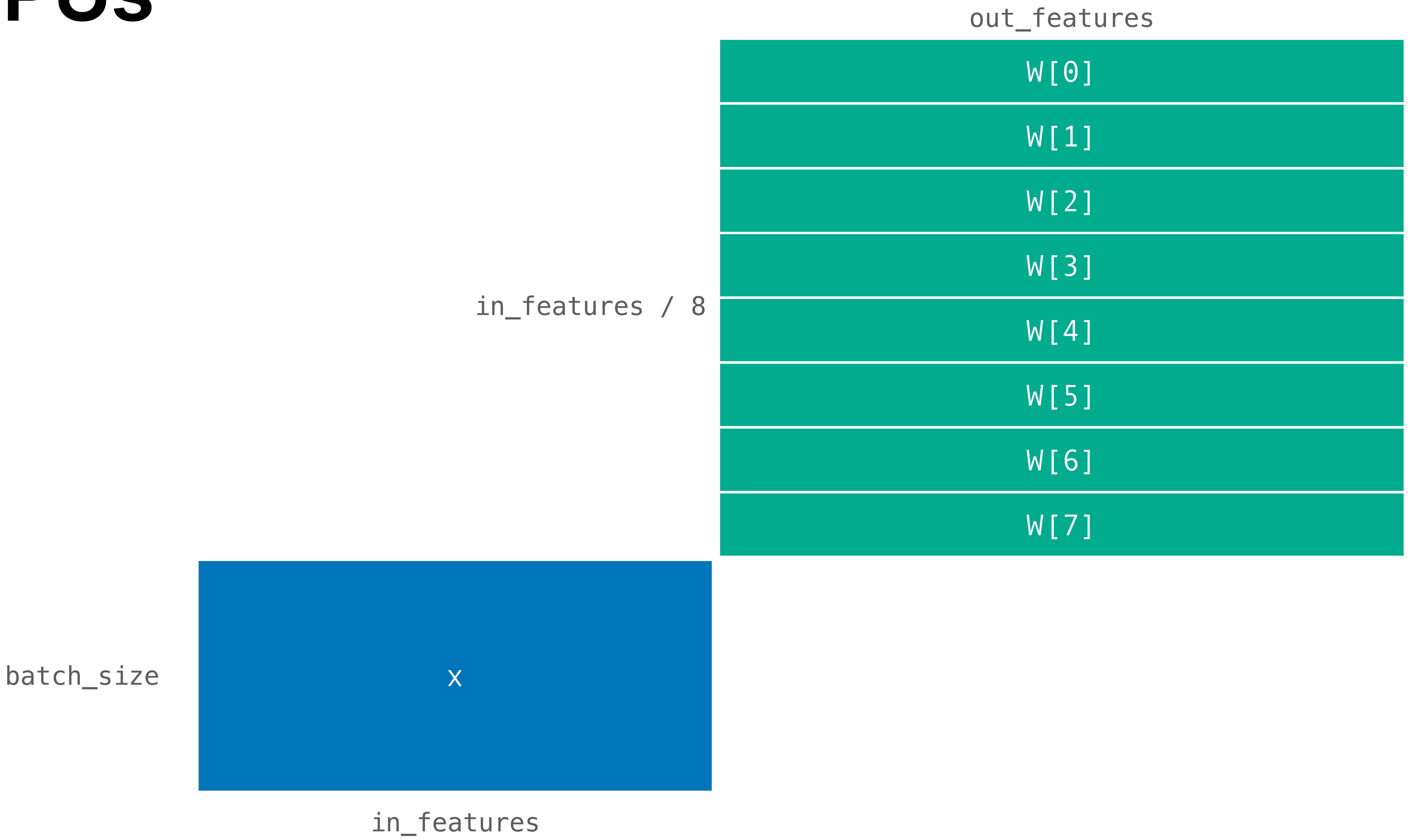


Source: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, Shoeybi et al, 2020

Row parallel divides the input dimension by GPUs

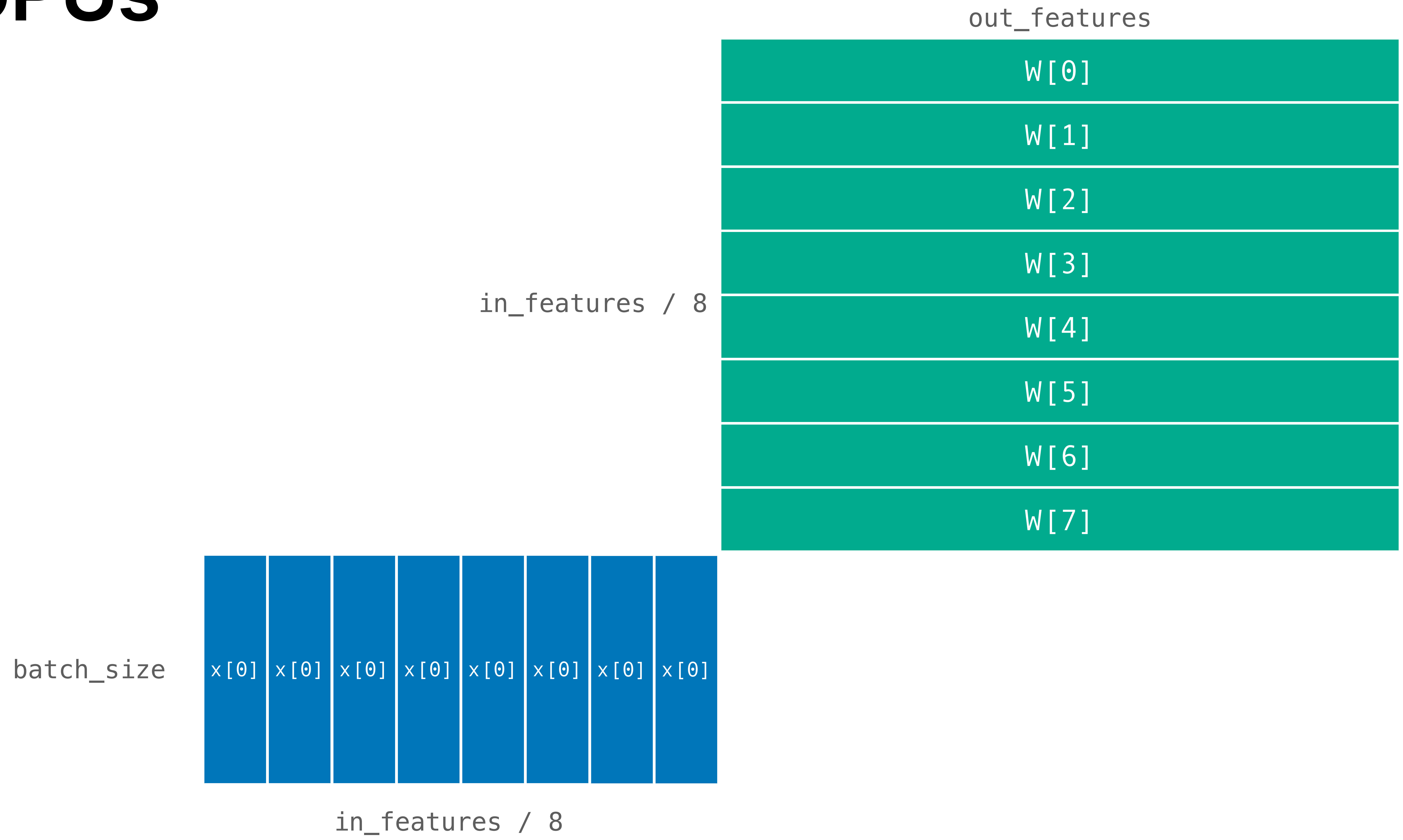
Source: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, Shoeybi et al, 2020

Row parallel divides the input dimension by GPUs



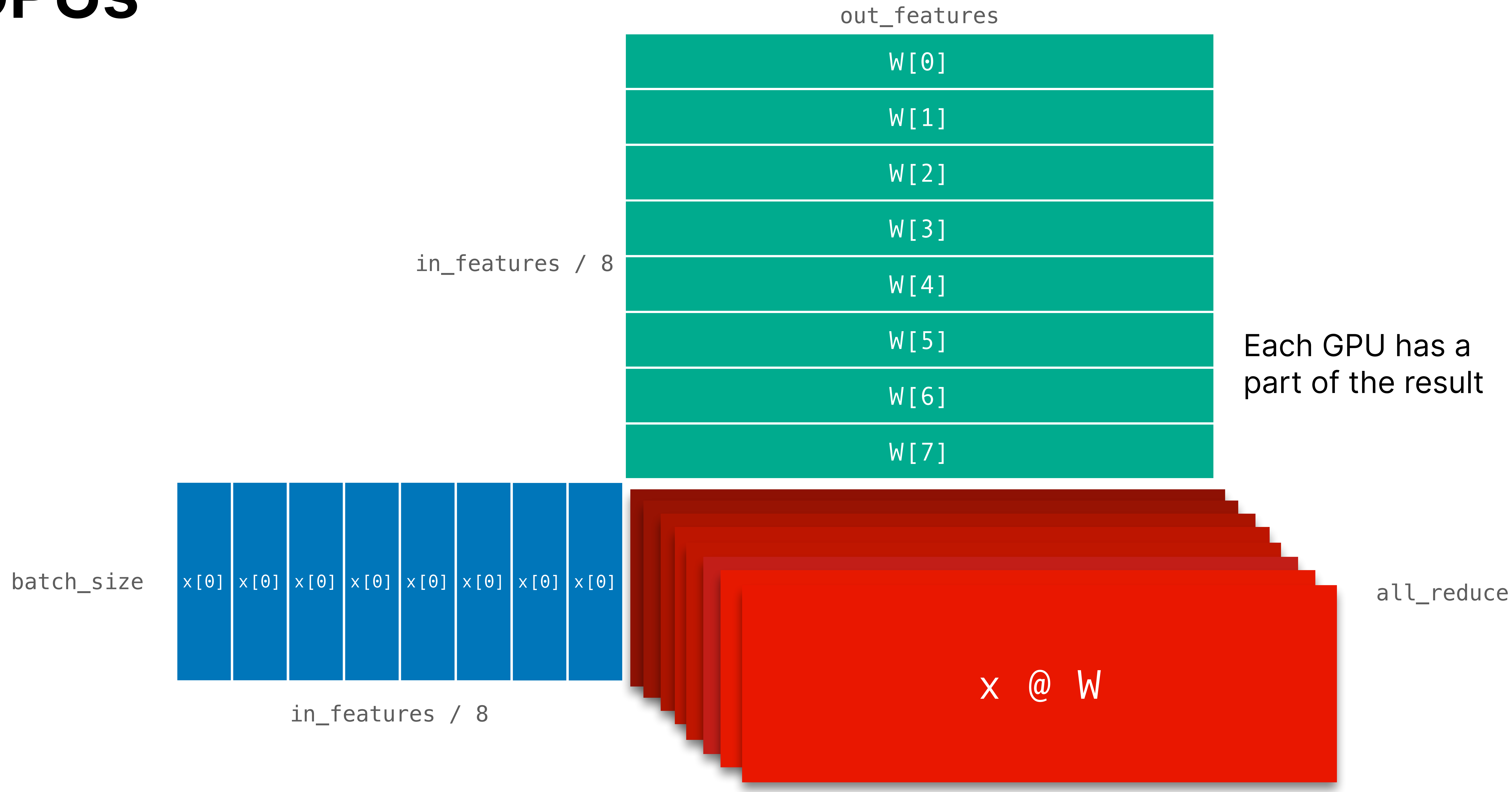
Source: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, Shoeybi et al, 2020

Row parallel divides the input dimension by GPUs



Source: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, Shoeybi et al, 2020

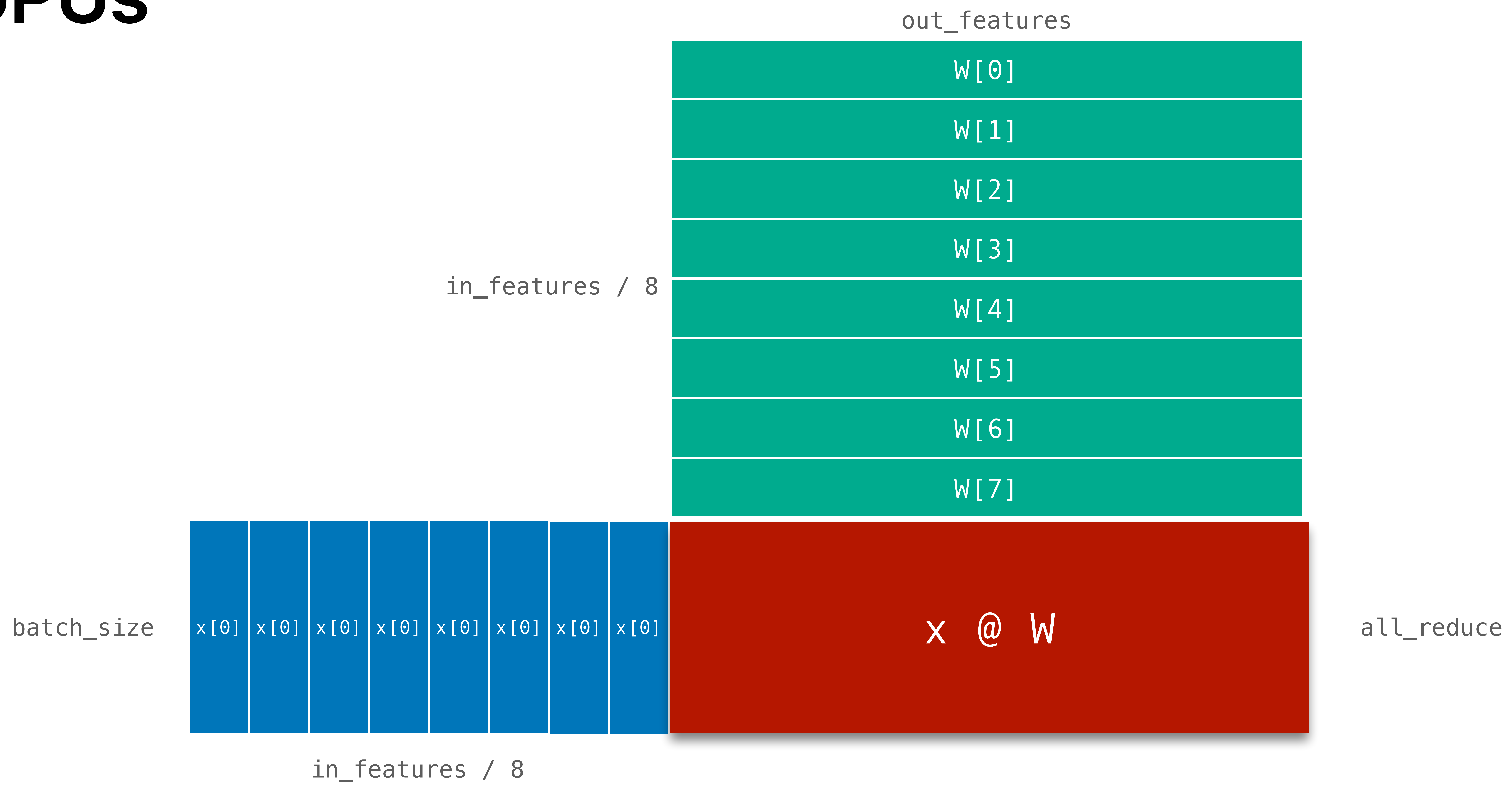
Row parallel divides the input dimension by GPUs



Each GPU has a part of the result

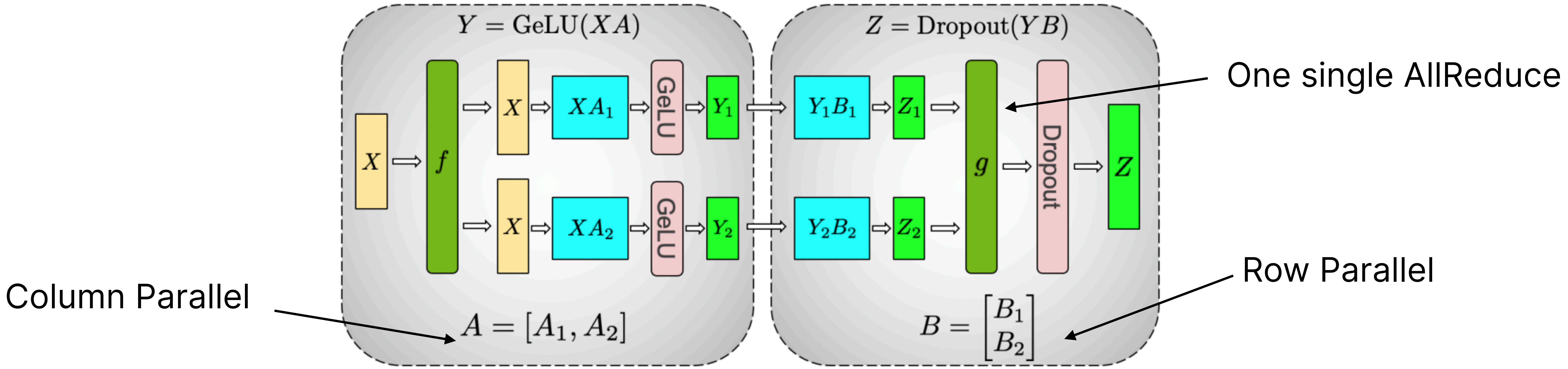
Source: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, Shoeybi et al, 2020

Row parallel divides the input dimension by GPUs

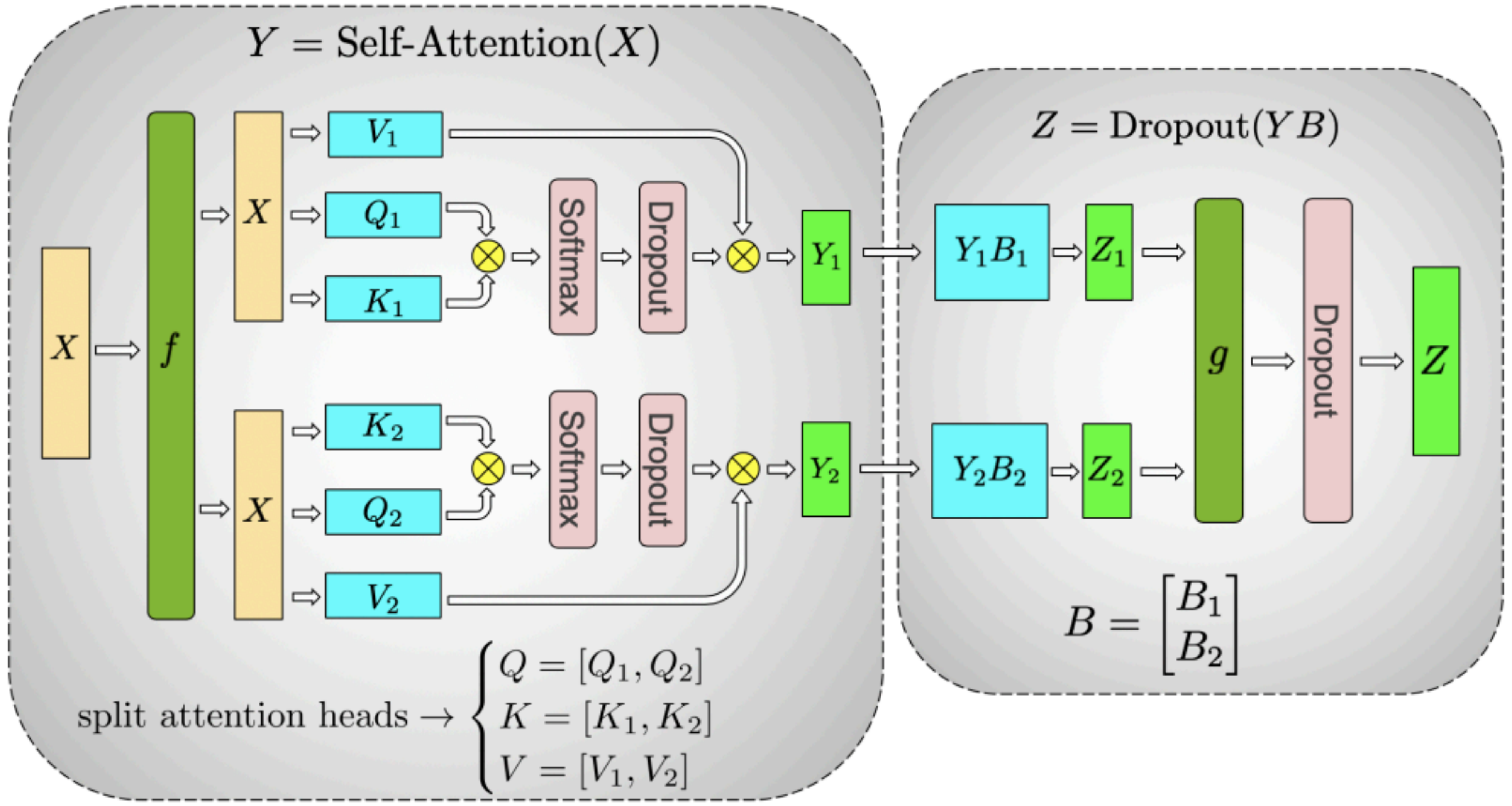


Source: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, Shoeybi et al, 2020

Megatron-LM cleverly combines these tricks, so there's only one synchronization step



(a) MLP



(b) Self-Attention

Source: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, Shoeybi et al, 2020

Back-of-the-envelope inference arithmetic

Say we have 8x A100 40GB GPUs trying to run inference with Llama2-70B

Let:

- `batch_size = 32`
- `input_seq_len = 512`
- `max_output_tokens = 64`

Numbers everyone should know

A100 fp16/bfloat16: 312e12 FLOPs/second (3 and then two 12s)

A100 memory bandwidth: 1.5 TB/second

H100 fp16/bfloat16: 1e15 FLOPs/second (a petaflop)

H100 memory bandwidth: 3.3 TB/second (roughly double A100)

NVLink interconnect: 300 GB/s

Calculating prefill on A100

Let:

- `batch_size = 32`
- `input_seq_len = 512`
- `max_output_tokens = 64`

FLOPS time

There are 70e9 parameters. For a single token each one of them is involved at exactly *one* point in the matrix multiply. It does a single multiplication and an add (think a dot product).

Total FLOPs: $\boxed{2 * 70e9}$ * 32 * 512 ~= 2.3e15

So total time is:

2.3e15 FLOPs / (8 * 312e12 FLOPs/sec) = 0.92s

Calculating prefill on A100

Let:

- `batch_size = 32`
- `input_seq_len = 512`
- `max_output_tokens = 64`

FLOPS time

There are 512 tokens per sequence, and 32 sequences in the batch.

Total FLOPs: $2 * 70e9 * \boxed{32 * 512}$ ~= 2.3e15

So total time is:

2.3e15 FLOPs / (8 * 312e12 FLOPs/sec) = 0.92s

Calculating prefill on A100

Let:

- `batch_size = 32`
- `input_seq_len = 512`
- `max_output_tokens = 64`

FLOPS time

Total FLOPs: 2 * 70e9 * 32 * 512 ~= 2.3e15

So total time is:

2.3e15 FLOPs / (8 * 312e12 FLOPs/sec) = 0.92s

Memory load time

Total bytes: 2 * 70e9 = 140e9

So total load time is:

140e9 bytes / (8 * 1.5e12 bytes / sec) ~= 0.01s

Calculating prefill on A100

Let:

- `batch_size = 32`
- `input_seq_len = 512`
- `max_output_tokens = 64`

FLOPS time

Total FLOPs: 2 * 70e9 * 32 * 512 ~= 2.3e15

So total time is:

2.3e15 FLOPs / (8 * 312e12 FLOPs/sec) = **0.92s**

Memory load time

Total bytes: 2 * 70e9 = 140e9

So total load time is:

140e9 bytes / (8 * 1.5e12 bytes / sec) ~= **0.01s**

Computation and memory loading are overlapped

Prefill time = $\boxed{\text{max(FLOPs time, load time)}}$ = 0.92s

Calculating prefill on A100

Let:

- `batch_size = 32`
- `input_seq_len = 512`
- `max_output_tokens = 64`

FLOPS time

Total FLOPs: 2 * 70e9 * 32 * 512 ~= 2.3e15

So total time is:

2.3e15 FLOPs / (8 * 312e12 FLOPs/sec) = **0.92s**

Memory load time

Total bytes: 2 * 70e9 = 140e9

So total load time is:

140e9 bytes / (8 * 1.5e12 bytes / sec) ~= **0.01s**

Prefill time = max(FLOPs time, load time) = 0.92s - compute bound!

Calculating prefill on A100

Let:

- `batch_size = 32`
- `input_seq_len = 512`
- `max_output_tokens = 64`

FLOPS time

Total FLOPs: 2 ∗ 70e9 ∗ 32 ∗ 512 ~= 2.3e15

So total time is:

2.3e15 FLOPs / (8 ∗ 312e12 FLOPs/sec) = **0.92s**

Memory load time

Total bytes: 2 ∗ 70e9 = 140e9

So total load time is:

140e9 bytes / (8 ∗ 1.5e12 bytes / sec) ~= **0.01s**

Time to First Token (TTFT): how long a user waits before they receive a response to their query.

Prefill time = max(FLOPs time, load time) = 0.92s - compute bound!

Calculating decoding time on A100

Let:

- `batch_size = 32`
- `input_seq_len = 512`
- `max_output_tokens = 64`

FLOPS time, per output token

Total FLOPs: 2 * 70e9 * 32 * 1 ~= 4.48e12

So total time is:

4.48e12 FLOPs / (8 * 312e12 FLOPs/sec) = **0.001s**

Memory load time (from before)

Total bytes: 2 * 70e9 = 140e9

So total load time is:

140e9 bytes / (8 * 1.5e12 bytes / sec) ~= **0.01s**

We only spend 10% of the time doing actual math!

Calculating decoding time on A100

Let:

- `batch_size = 32`
- `input_seq_len = 512`
- `max_output_tokens = 64`

FLOPS time, per output token

Total FLOPs: 2 ∗ 70e9 ∗ 32 ∗ 1 ~= 4.48e12

So total time is:

4.48e12 FLOPs / (8 ∗ 312e12 FLOPs/sec) = **0.001s**

Memory load time (from before)

Total bytes: 2 ∗ 70e9 = 140e9

So total load time is:

140e9 bytes / (8 ∗ 1.5e12 bytes / sec) ~= **0.01s**

For 64, output tokens, 64 ∗ 0.01s = 0.64s

Calculating decoding time on A100

Let:

- `batch_size = 32`
- `input_seq_len = 512`
- `max_output_tokens = 64`

FLOPS time, per output token

Total FLOPs: 2 * 70e9 * 32 * 1 ~= 4.48e12

So total time is:

4.48e12 FLOPs / (8 * 312e12 FLOPs/sec) = **0.001s**

Memory load time (from before)

Total bytes: 2 * 70e9 = 140e9

So total load time is:

140e9 bytes / (8 * 1.5e12 bytes / sec) ~= **0.01s**

For 64, output tokens, 64 * 0.01s = 0.64s.

Prefill was done in ~0.9s. We processed 8x more tokens in just 1.5x the time.

Two numbers to bring these numbers closer to reality

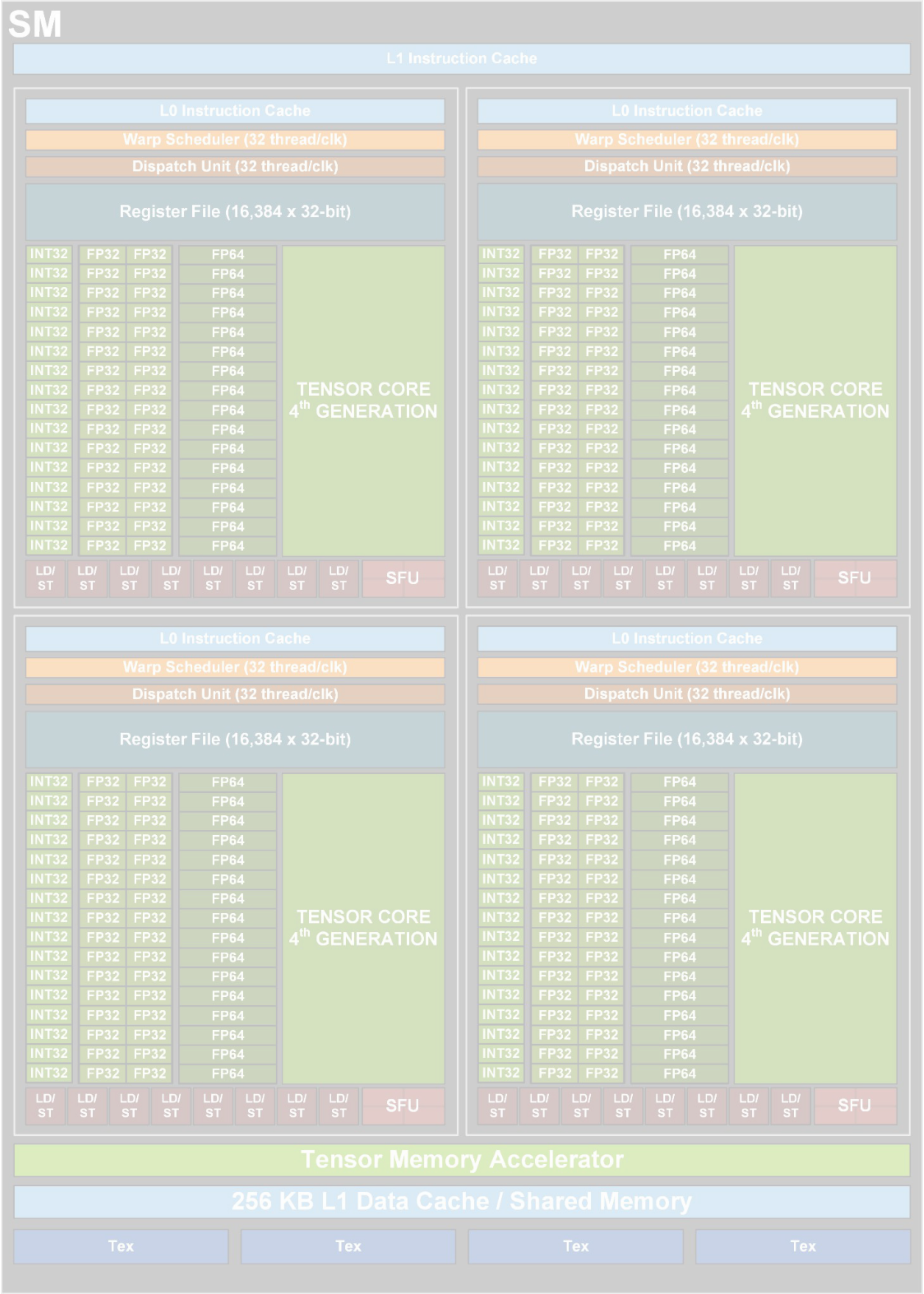
Model Bandwidth Utilization (MBU)

Relative to the advertised system bandwidth, what is the actual bandwidth realized?

High bandwidth memory (HBM)

Model FLOPs Utilization (MFU)

Relative to how fast the accelerator claims to run, what percent of the FLOPs do we actually see when we run the model?



Two numbers to bring these numbers closer to reality

Model Bandwidth Utilization (MBU)

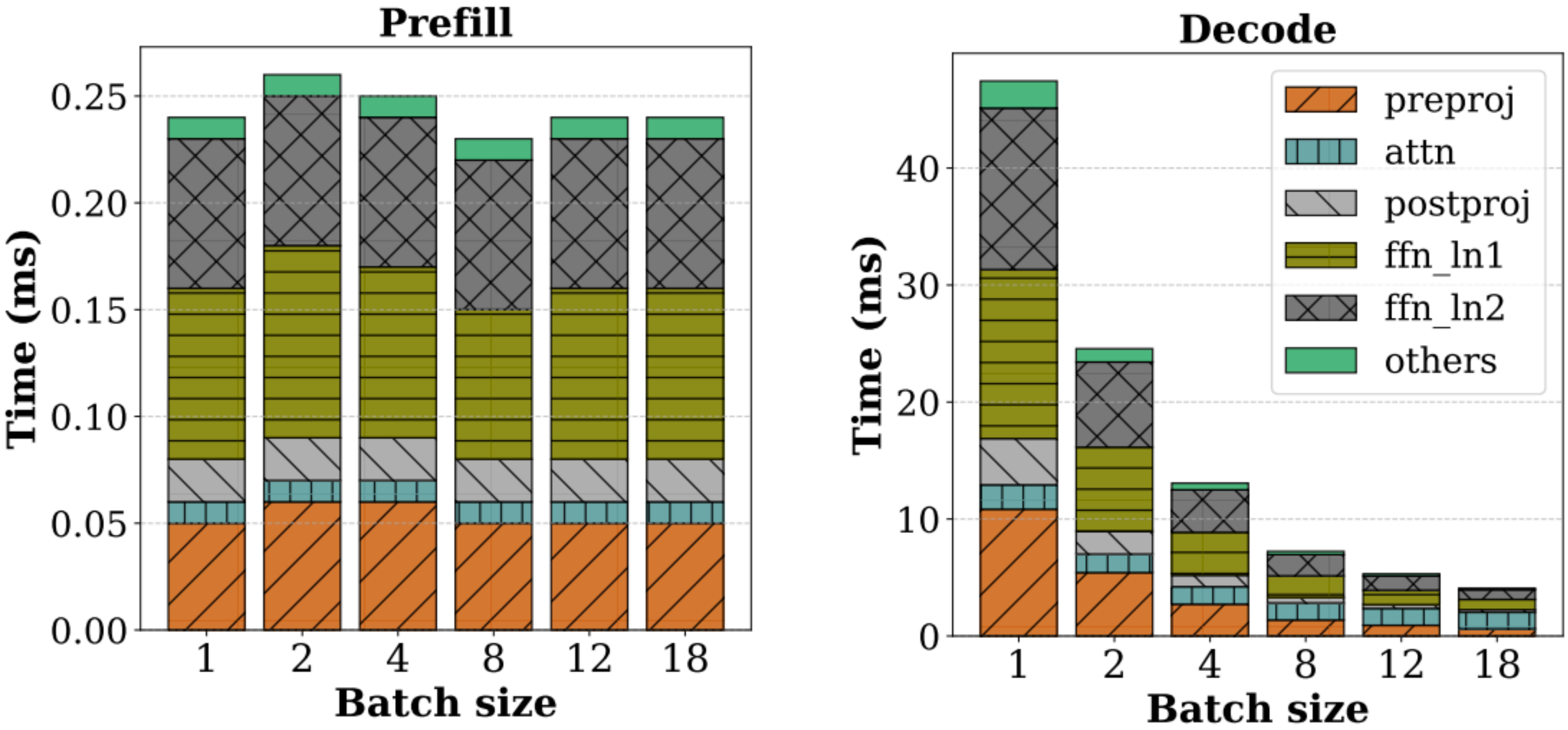
Relative to the advertised system bandwidth, what is the actual bandwidth realized?

High bandwidth memory (HBM)

Model FLOPs Utilization (MFU)

Relative to how fast the accelerator claims to run, what percent of the FLOPs do we actually see when we run the model?

Prefill vs. decode per-token latencies



160x difference, on A6000 GPU

Source: SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills

Prefill vs. decode per-token latencies

Model	Input	Output
8K context	$0.03 / 1K tokens	$0.06 / 1K tokens
32K context	$0.06 / 1K tokens	$0.12 / 1K tokens

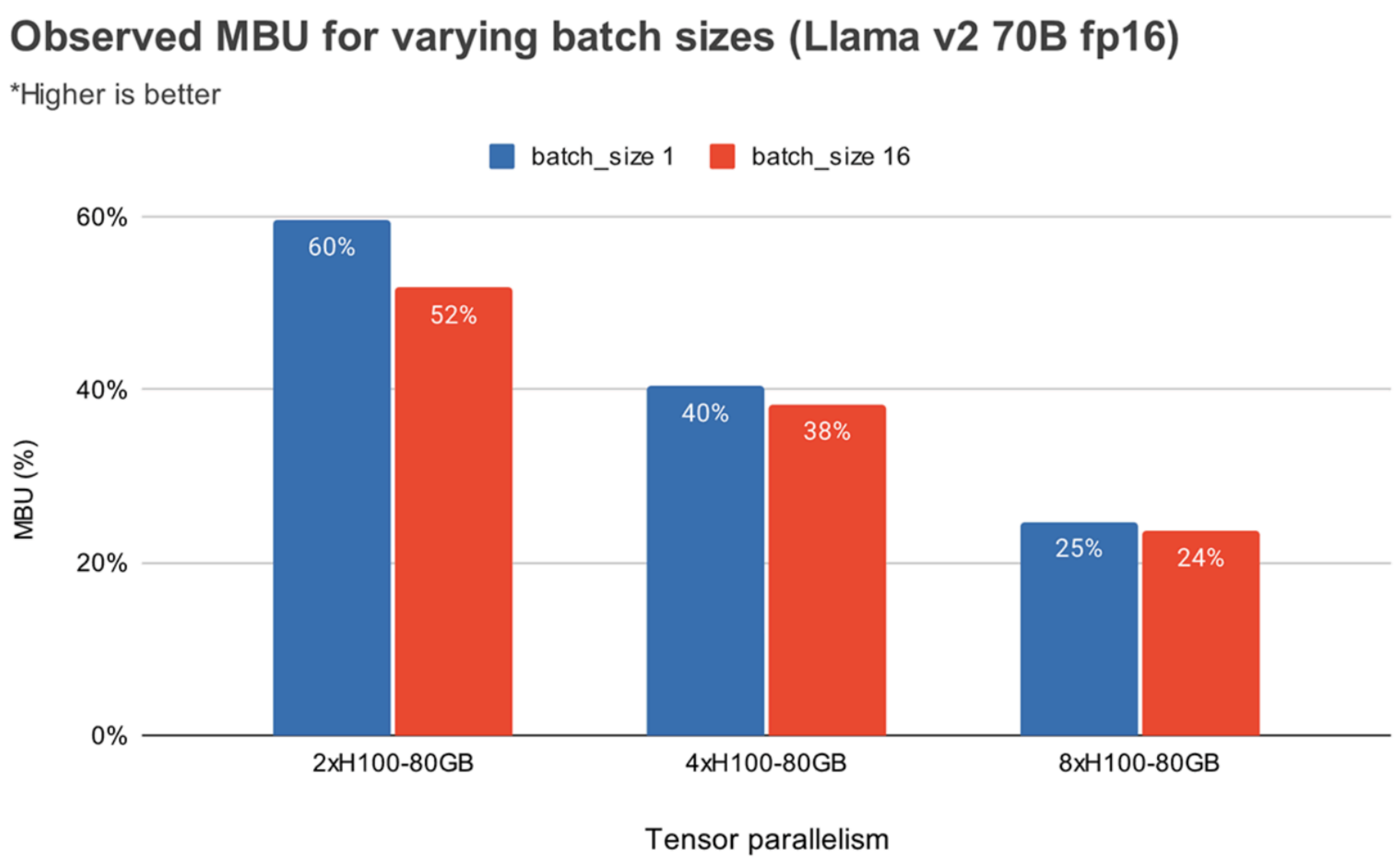
OpenAI GPT-4 token pricing

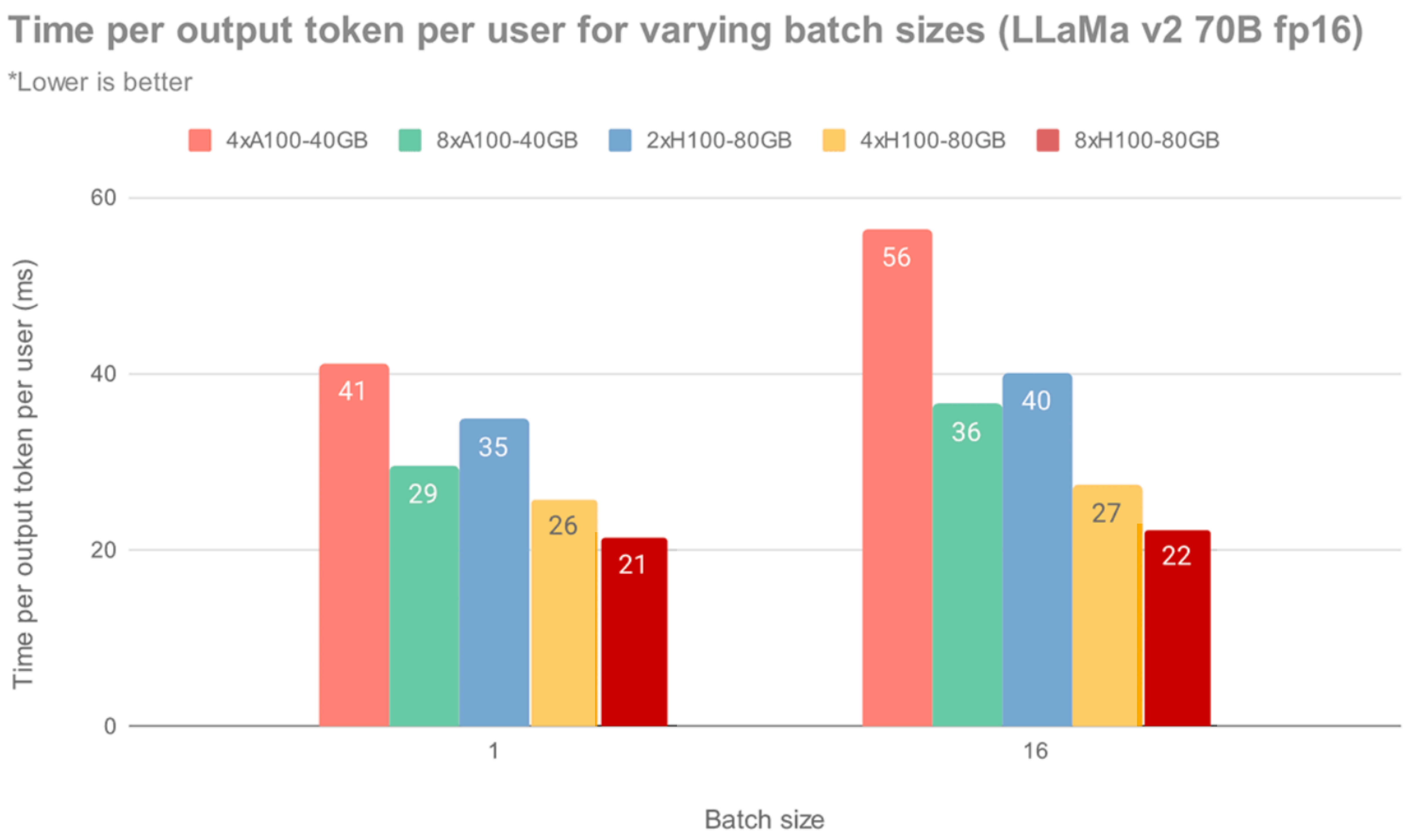
Be careful about tokens/sec. If it includes input and output tokens, the metric could make you think your system is running a lot faster than it actually is.

Two metrics we care about:

1. **Time To First Token (TTFT):** how long does it take before the first token is generated?
2. **Time Per Output Token (TPOT):** how long does it take for each output token to be generated?

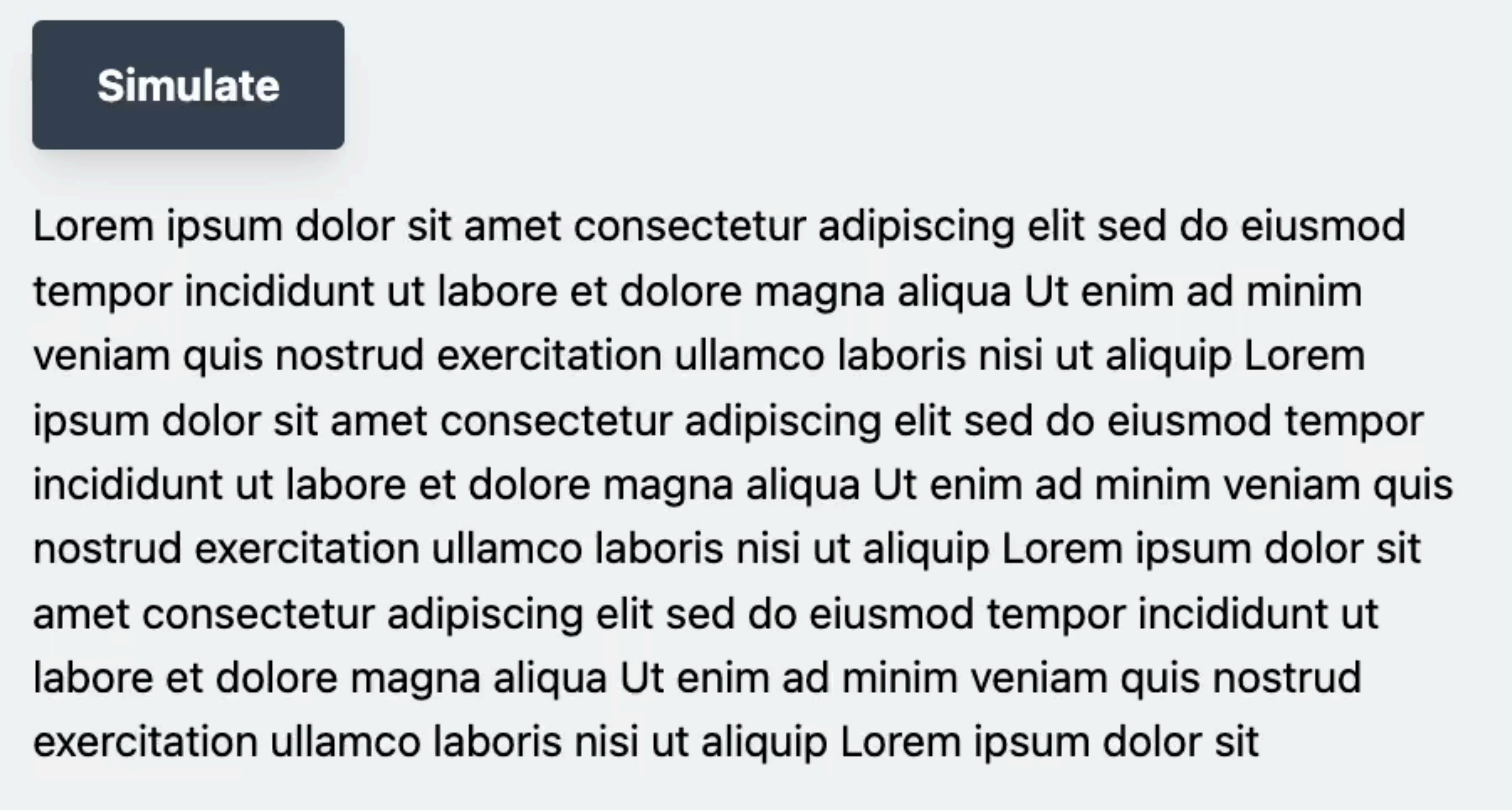
MBU numbers for Llama2-70B





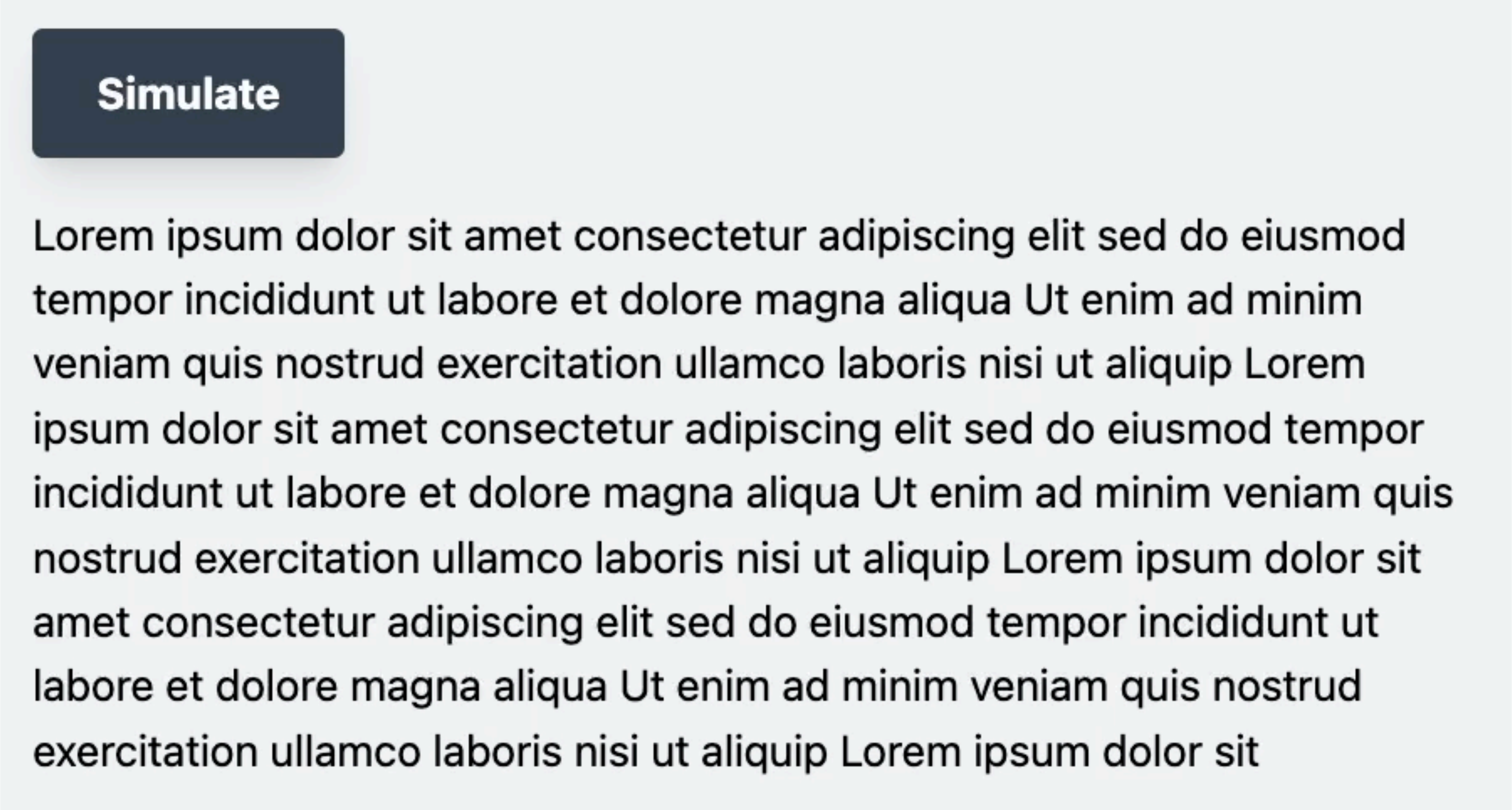
Source: LLM Inference Performance Engineering: Best Practices

Milliseconds are confusing...what do these numbers mean?



Time to first token: 4s
Time per output token: 80ms

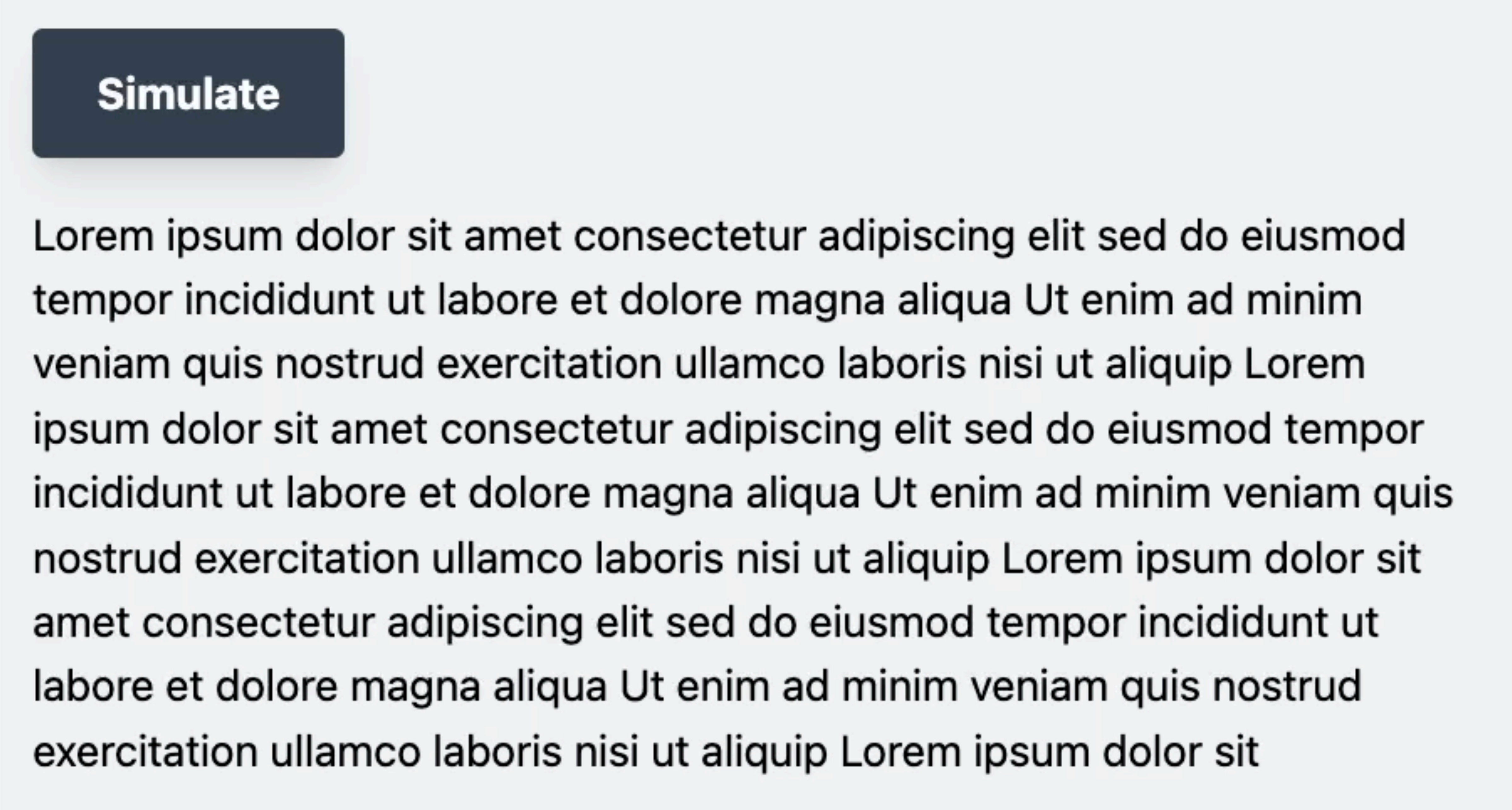
Milliseconds are confusing...what do these numbers mean?



Time to first token: 3s
Time per output token: 46ms

Milliseconds are confusing...what do these numbers mean?

This is a North Star: it’s close to “Copilot” level functionality



Time to first token: 1s
Time per output token: 16ms

Both of these metrics are important

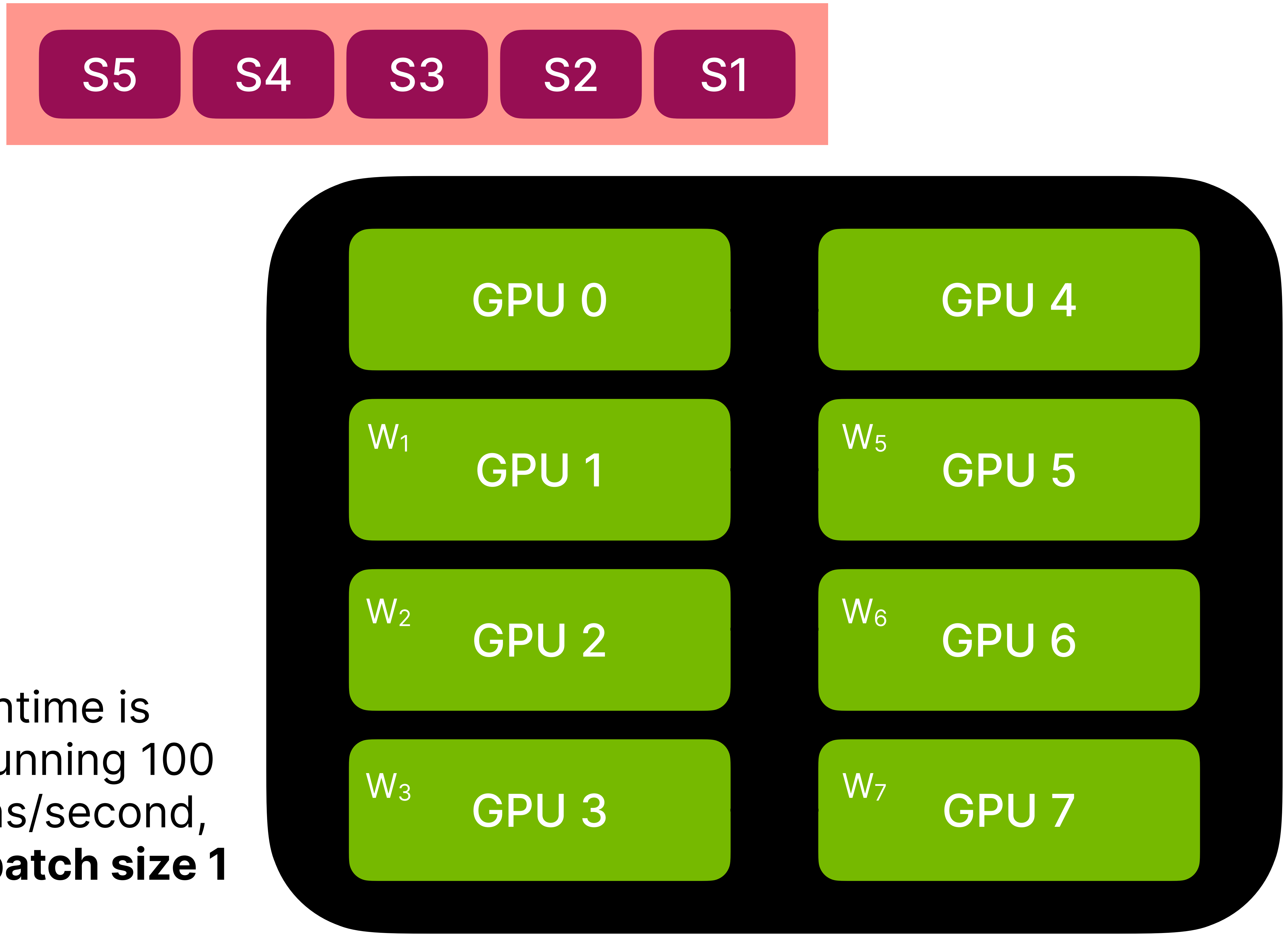
Example: optimize for decoding throughput by only running one sequence at a time

Both of these metrics are important

Example: optimize for decoding throughput by only running one sequence at a time

I have the fastest inference engine out there at 100 output tokens/second!

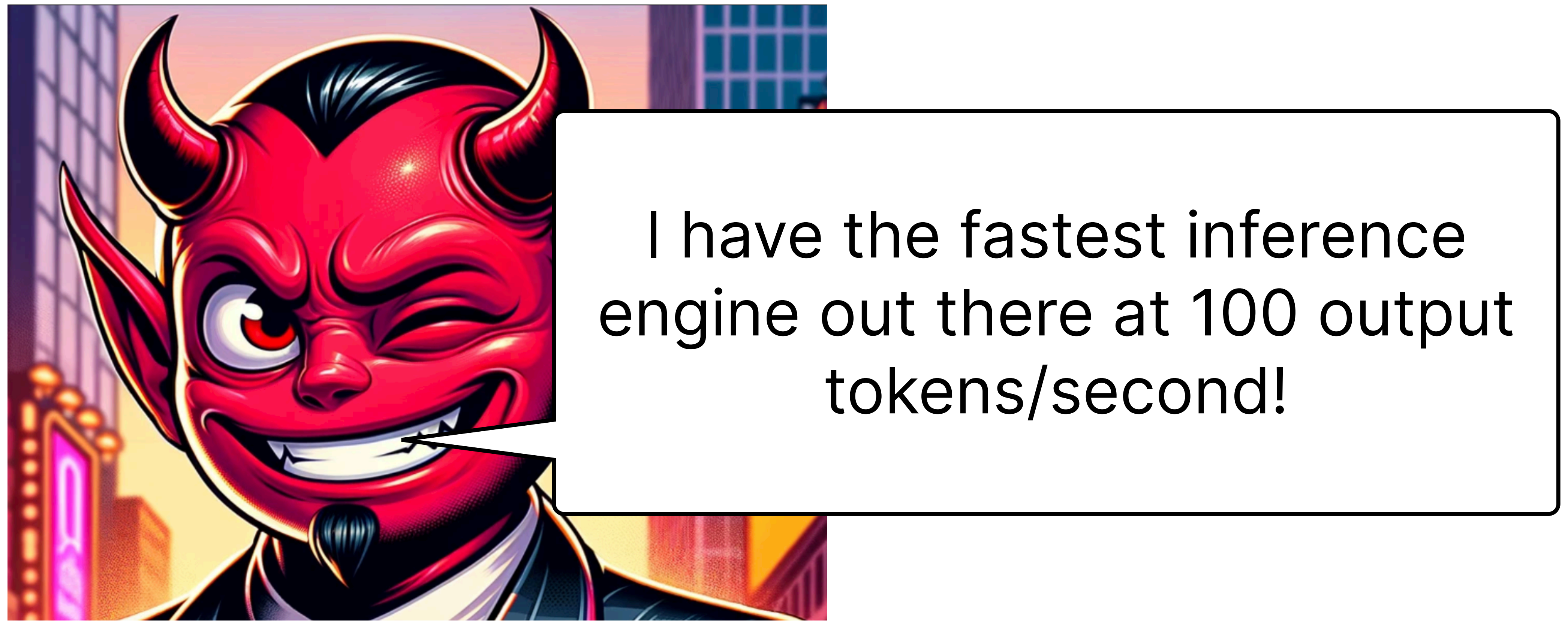
Request queue with sequences

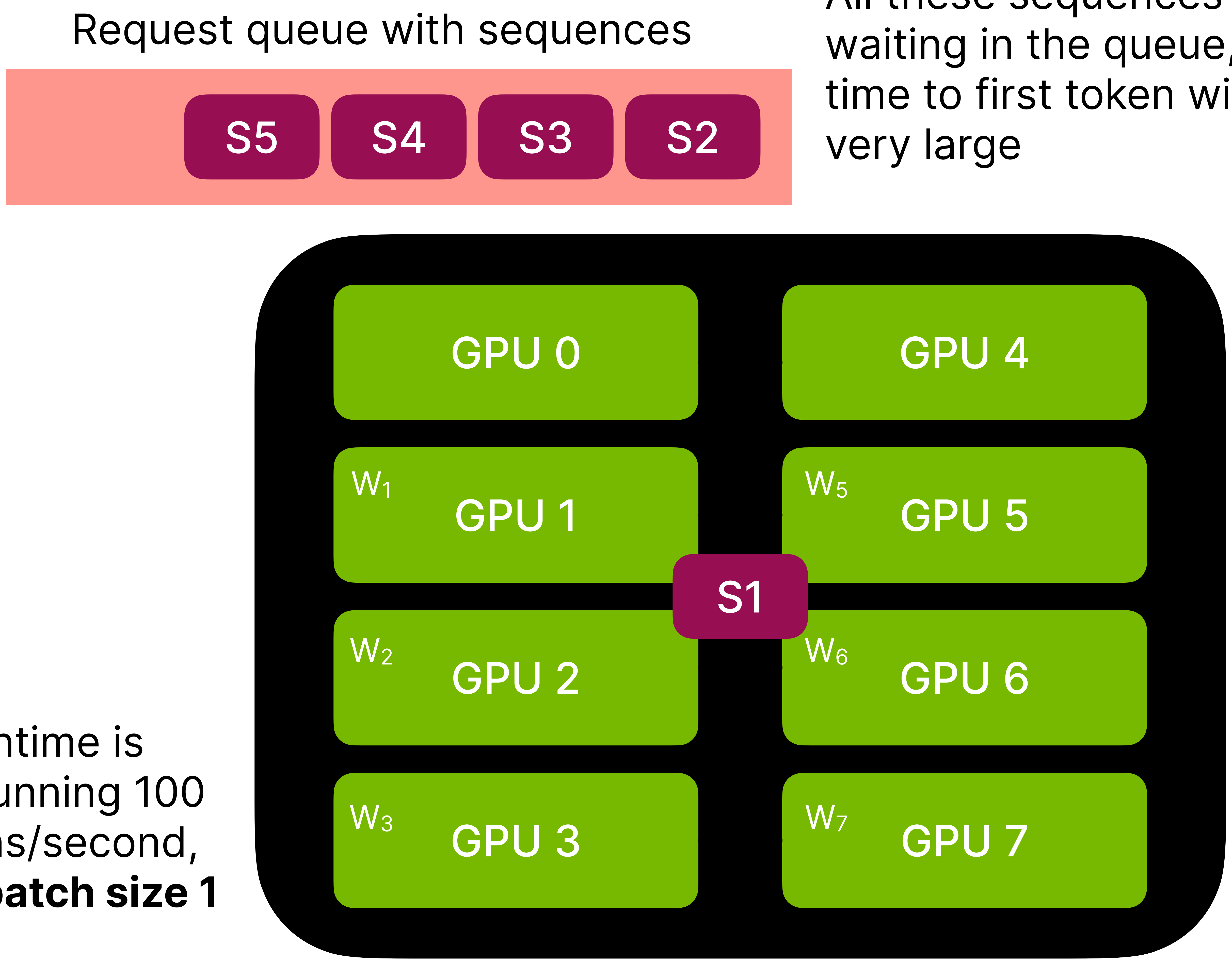


Inference runtime is capable of running 100 output tokens/second, but **only at batch size 1**

Both of these metrics are important

Example: optimize for decoding throughput by only running one sequence at a time





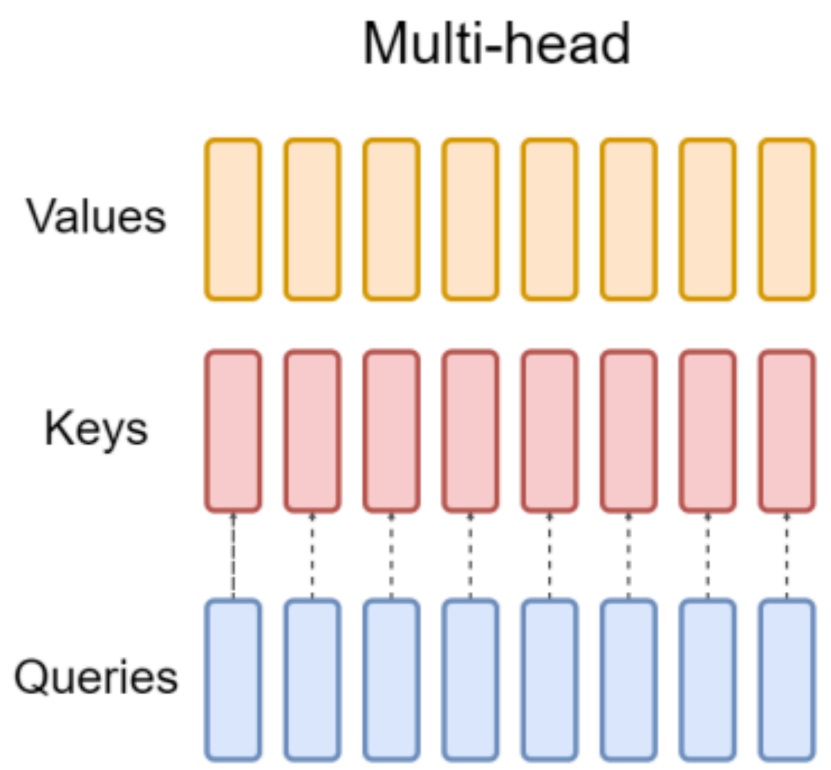
All these sequences are waiting in the queue, so time to first token will be very large

Inference runtime is capable of running 100 output tokens/second, but **only at batch size 1**

How do we speed this up?

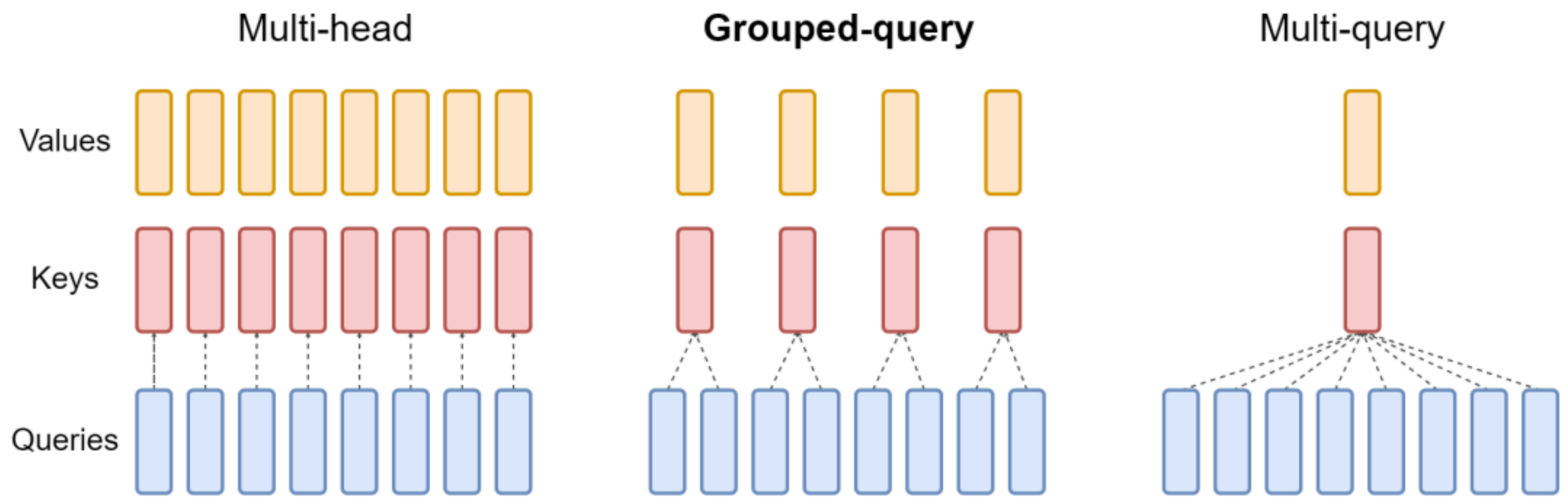
Idea 1: reduce how much memory you need

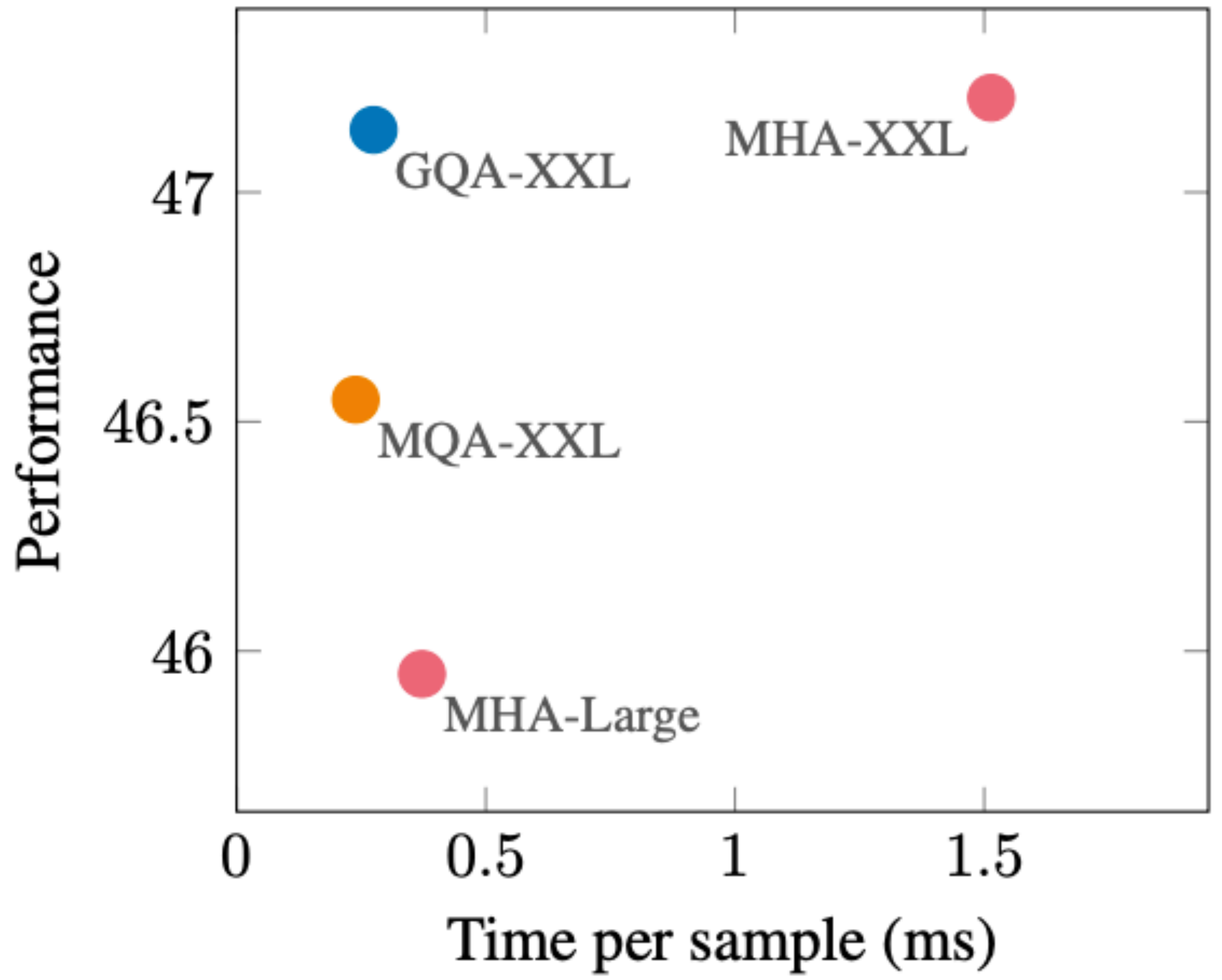
Idea 1.1: make the KV cache smaller?



Source: GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints

Idea 1.1: make the KV cache smaller?

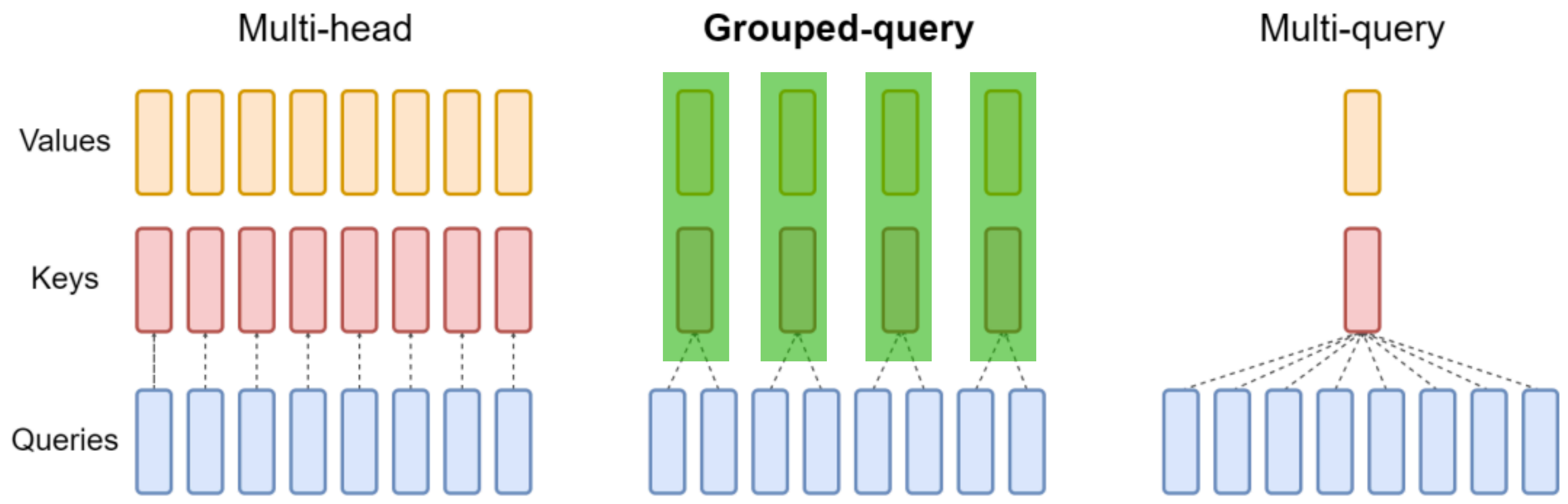


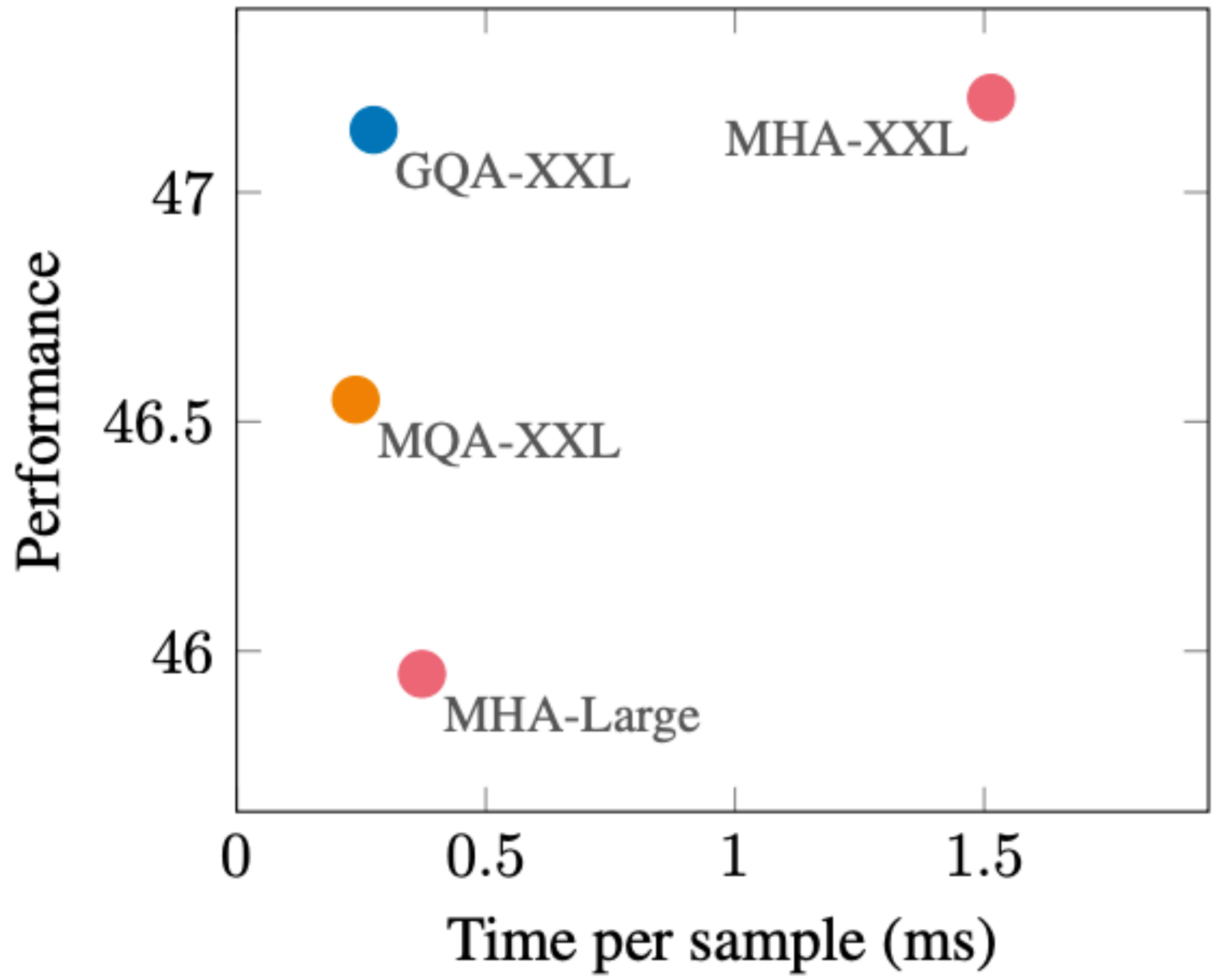


Grouped query attention: reduce the number of key and value heads to some multiple of the number of query heads. Produces negligible performance decrease for large (>2x reduction in inference cost)

Source: GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints

Idea 1.1: make the KV cache smaller?





Serve by partitioning each head on a different GPU

Source: GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints

Idea 1.1: make the KV cache smaller?

Let's say we're serving a
Llama2-70B model with:

- `precision = fp16`
- `d_model = 8192`
- `n_layers = 80`
- `batch_size = 4`
- `input_seq_len = 1024`
- `max_new_tokens = 32`
- `head_size = 128`
- `kv_n_heads = 64`

KV cache size (maximum)

80 * 2 * 2 bytes/param * 64 * 128 * 4 * (1024 + 32) ~= 11e9 bytes = **11 GB**

Idea 1.1: make the KV cache smaller?

Let's say we're serving a
Llama2-70B model with:

- `precision = fp16`
- `d_model = 8192`
- `n_layers = 80`
- `batch_size = 4`
- `input_seq_len = 1024`
- `max_new_tokens = 32`
- `head_size = 128`
- `kv_n_heads = 8`

KV cache size (maximum)

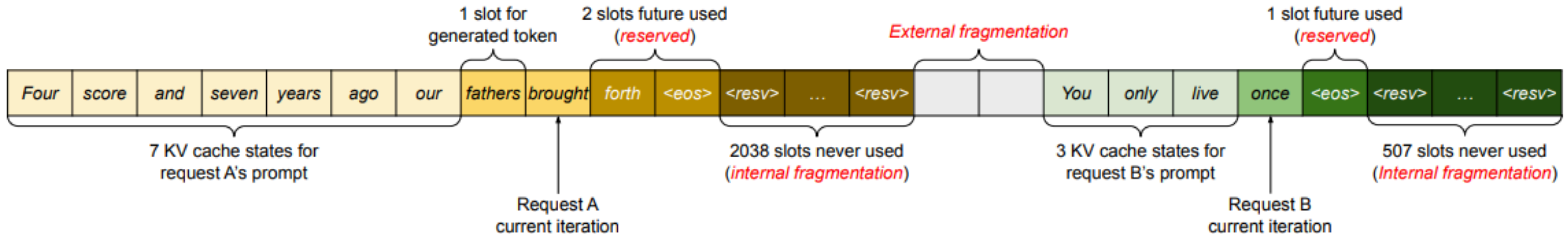
80 * 2 * 2 bytes/param * 8 * 128 * 4 * (1024 + 32)
~= 11e9 bytes ~= **1.3 GB**

8x reduction!

Idea 1.2: how do we actually allocate?

How much memory is required for a request: how many total tokens will the generation take?

Problem: fragmentation, which occurs from allocation and frees



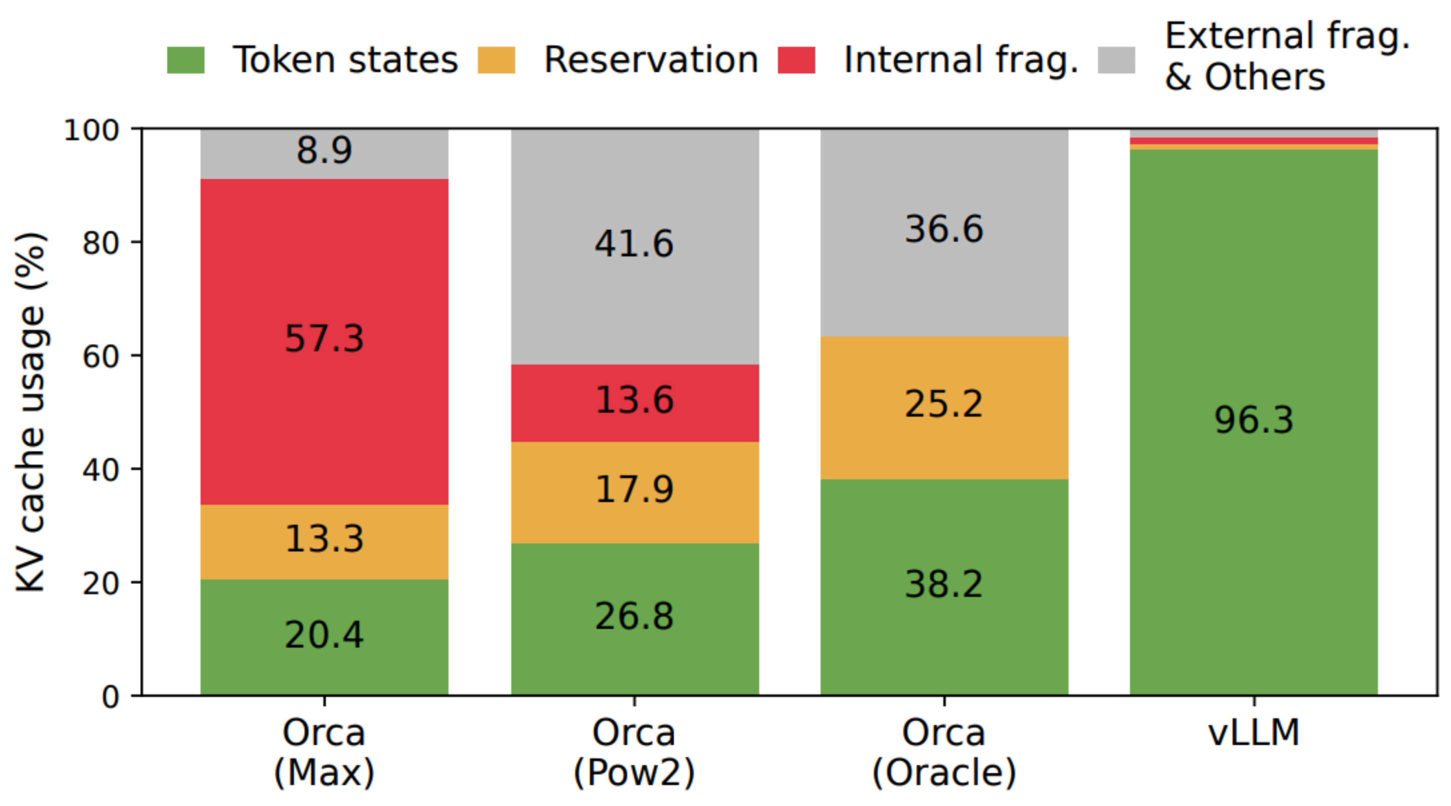
Source: Efficient Memory Management for Large Language Model Serving with PagedAttention

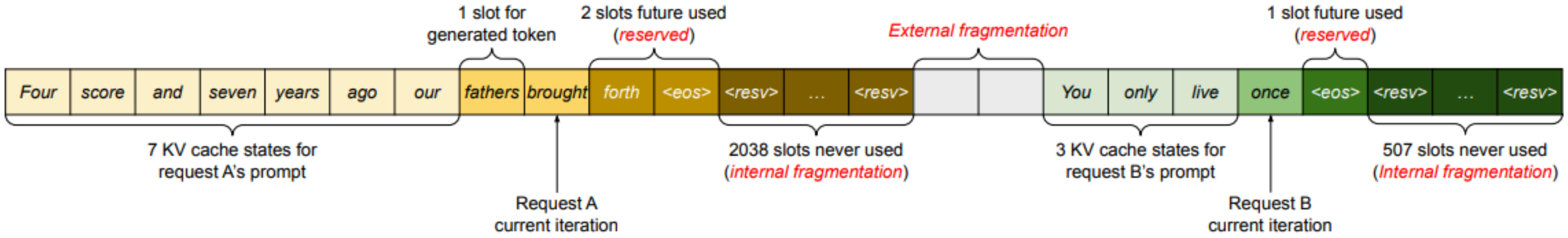
Idea 1.2: how do we actually allocate?

How much memory is required for a request: how many total tokens will the generation take?

Problem: fragmentation, which occurs from allocation and frees

Tons of memory waste!

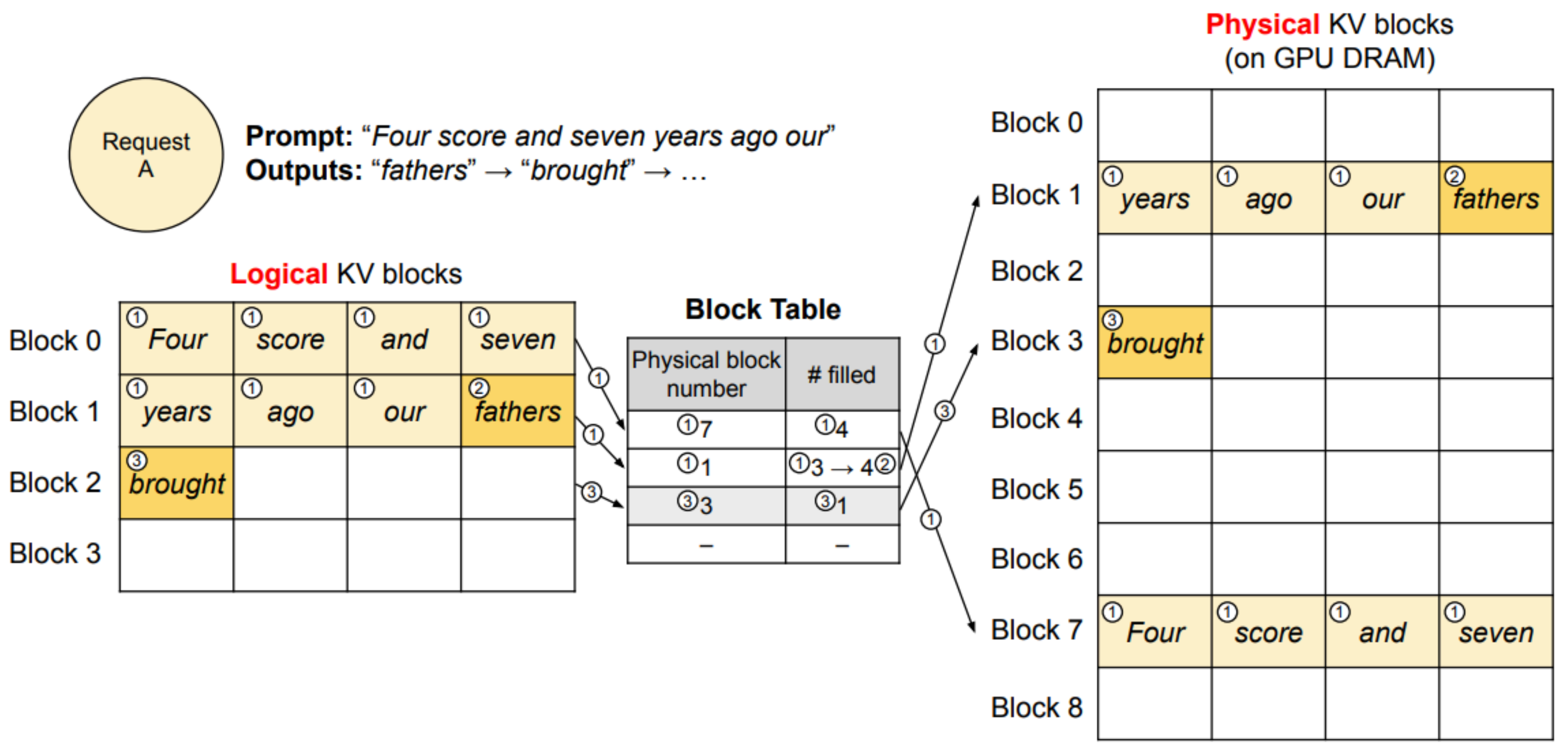




Source: Efficient Memory Management for Large Language Model Serving with PagedAttention

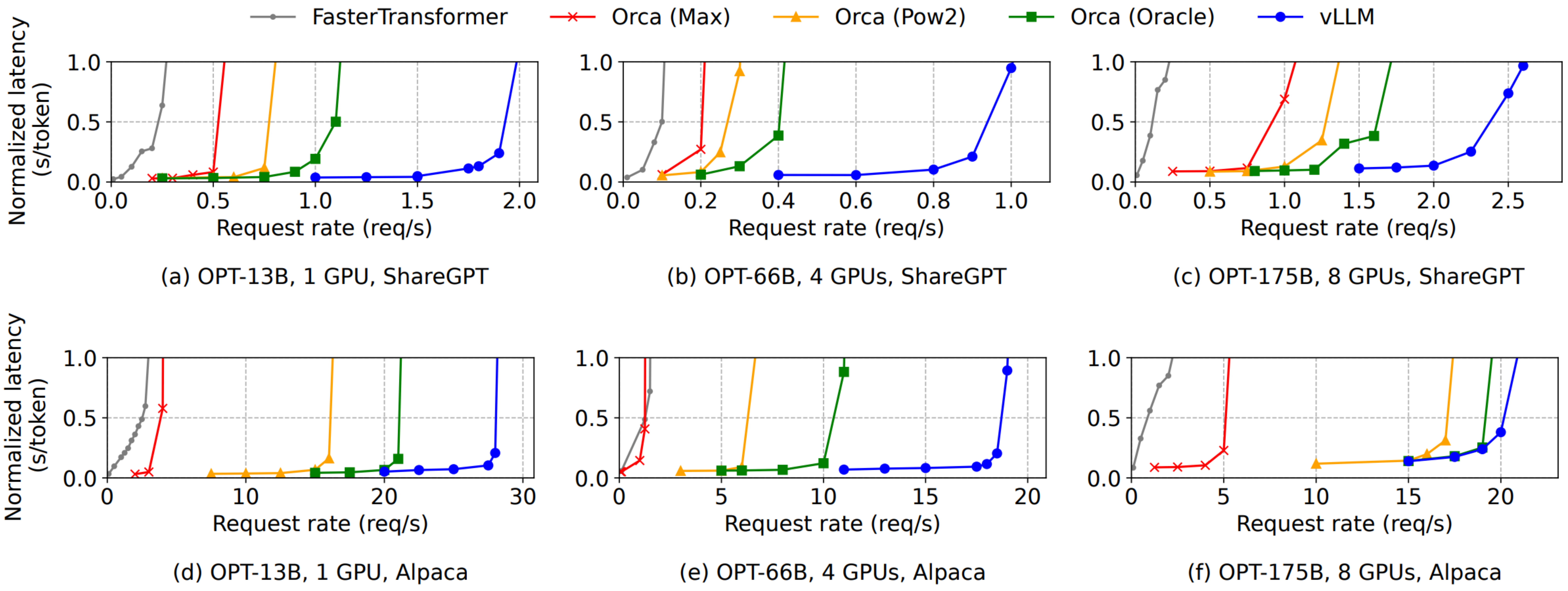
Idea 1.2: how do we actually allocate?

- Map logical blocks to physical blocks on GPU RAM
- Fix the block sizes, only allocate when necessary
- If there are multiple blocks that come in with the same prompt, then increase the reference count of physical KV blocks



Source: Efficient Memory Management for Large Language Model Serving with PagedAttention

Idea 1.2: how do we actually allocate?



Source: Efficient Memory Management for Large Language Model Serving with PagedAttention

Idea 2: get a greater bang for your bytes by increasing the batch size

A first idea: naively take requests and run them through the model

Write me an epic ten-thousand page novel that will win a Nobel Prize.

Who wears a better shacket - Evan or Linden?

Do you like living in San Francisco?

What should I name my pet rock?

What happens when the request is done?

Write me an epic ten-thousand page novel that will win a Nobel Prize.

Who wears a better shacket - Evan or Linden?

Do you like living in San Francisco?

What should I name my pet rock?

As	a	large	language	model	I	don't	know	how	but	I
Evan	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>
Sure	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>
Rocky	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>

Batched LLM processing keeps these sequences idle, as the request latency becomes the maximum of all sequences in the batch.

What happens when the request is done?

Write me an epic ten-thousand page novel that will win a Nobel Prize.

Who wears a better shacket - Evan or Linden?

Do you like living in San Francisco?

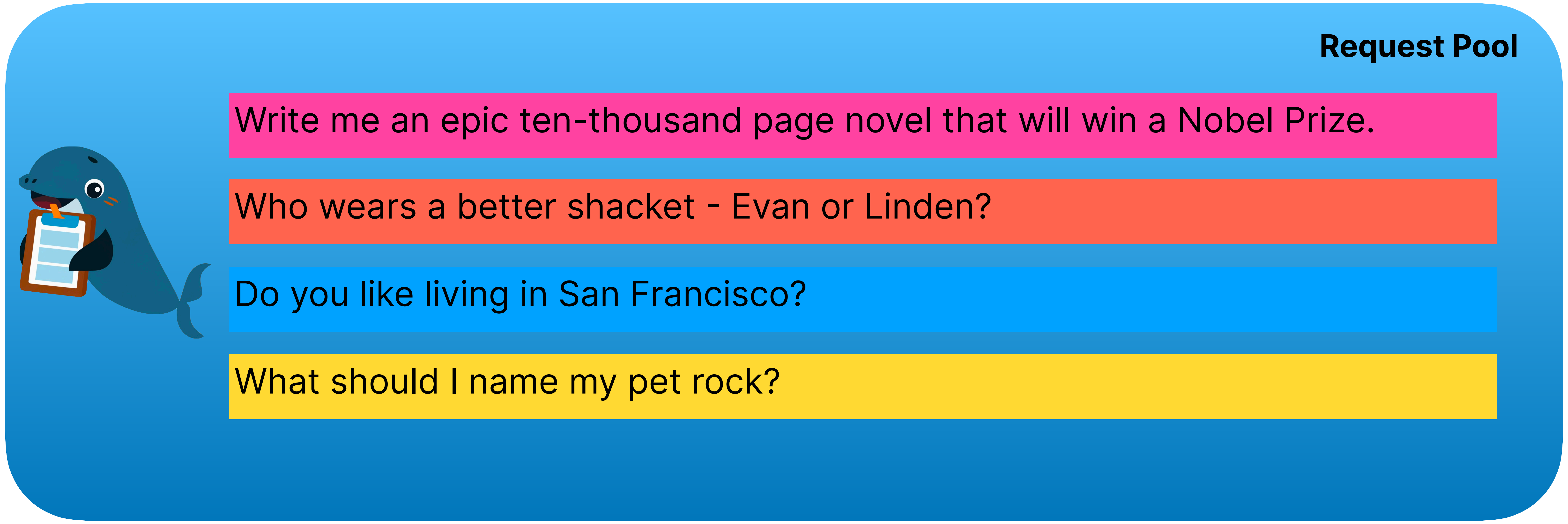
What should I name my pet rock?

As	a	large	language	model	I	don't	know	how	but	I
Evan	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>
Sure	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>
Rocky	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>	</s>

When will AGI be achieved internally at Databricks?

There's room for more requests...but we can't serve them.
IMPORTANT QUESTIONS CAN'T GET ANSWERED!

A better idea: exploit iterations



Introduce the Orca scheduler

1. Select the requests to run next.

A better idea: exploit iterations

Write me an epic ten-thousand page novel that will win a Nobel Prize.

Who wears a better shacket - Evan or Linden?

Do you like living in San Francisco?

What should I name my pet rock?

Introduce the Orca scheduler

1. Select the requests to run next.

A better idea: exploit iterations

`~~model.forward(input_ids)~~ model.run_step(ids)`

Write me an epic ten-thousand page novel that will win a Nobel Prize.

Who wears a better shacket - Evan or Linden?

Do you like living in San Francisco?

What should I name my pet rock?

As
Evan
Sure
Rocky

Request Pool

Introduce the Orca scheduler

1. Select the requests to run next.
2. Run an iteration of the engine

A better idea: exploit iterations

Write me an epic ten-thousand page novel that will win a Nobel Prize.

Who wears a better shacket - Evan or Linden?

Do you like living in San Francisco?

What should I name my pet rock?

As
Evan
Sure
Rocky

Request Pool

When will AGI be achieved internally at Databricks?

Introduce the Orca scheduler

1. Select the requests to run next.
2. Run an iteration of the engine
3. Receive execution results

A better idea: exploit iterations

Write me an epic ten-thousand page novel that will win a Nobel Prize.

Who wears a better shacket - Evan or Linden?

Do you like living in San Francisco?

What should I name my pet rock?

As

Evan

Sure

Rocky

Request Pool

When will AGI be achieved internally at Databricks?

Introduce the Orca scheduler

1. Select the requests to run next.
2. Run an iteration of the engine
3. Receive execution results

A better idea: exploit iterations

Write me an epic ten-thousand page novel that will win a Nobel Prize.

Who wears a better shacket - Evan or Linden?

Do you like living in San Francisco?

What should I name my pet rock?

As	a
Evan	</s>
Sure	</s>
Rocky	</s>

Request Pool

When will AGI be achieved internally at Databricks?

Introduce the Orca scheduler

1. Select the requests to run next.
2. Run an iteration of the engine
3. Receive execution results

A better idea: exploit iterations

Write me an epic ten-thousand page novel that will win a Nobel Prize.

As	a

KV cache evicted
GPU memory freed

Request Pool

When will AGI be achieved internally at Databricks?

Introduce the Orca scheduler

1. Select the requests to run next.
2. Run an iteration of the engine
3. Receive execution results

A better idea: exploit iterations

Write me an epic ten-thousand page novel that will win a Nobel Prize.

When will AGI be achieved internally at Databricks?

As	a

Request Pool

Introduce the Orca scheduler

1. Select the requests to run next.
2. Run an iteration of the engine
3. Receive execution results

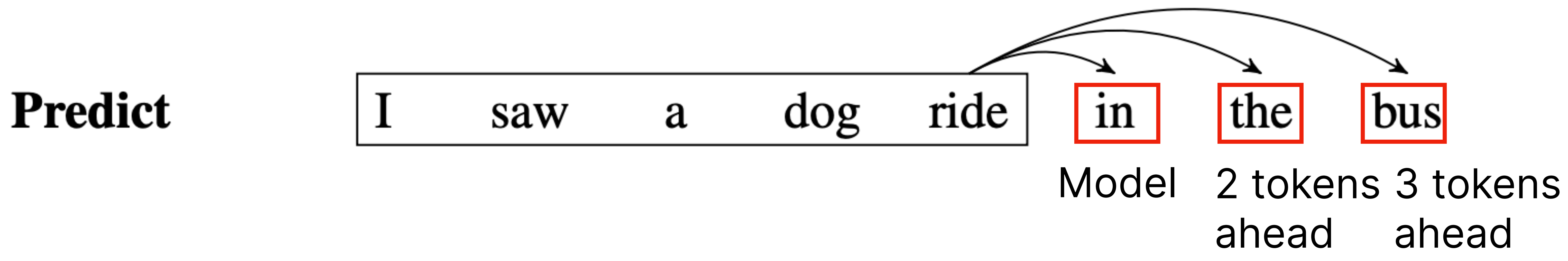
Iteration level batching is a *lot* faster

- Reduced waiting time for a given request
- High GPU utilization from large batch sizes
- Less wasted computation from padding within a simple

Idea 3: speed up decoding by trying to decode more tokens in parallel

What if we decoded several tokens in parallel if the problem is decoding one token at a time?

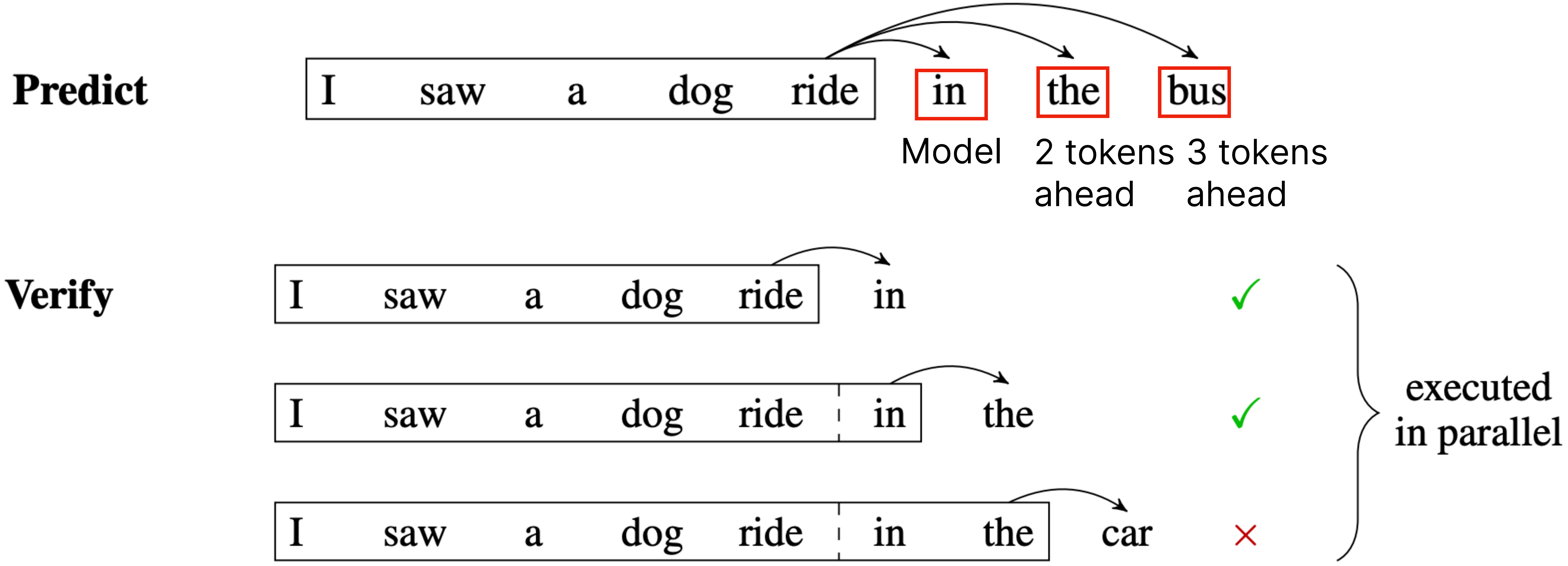
Idea: train auxiliary models that can predict *n* tokens instead (not just 1 token ahead)



Source: Blockwise Parallel Decoding for Deep Autoregressive Models

What if we decoded several tokens in parallel if the problem is decoding one token at a time?

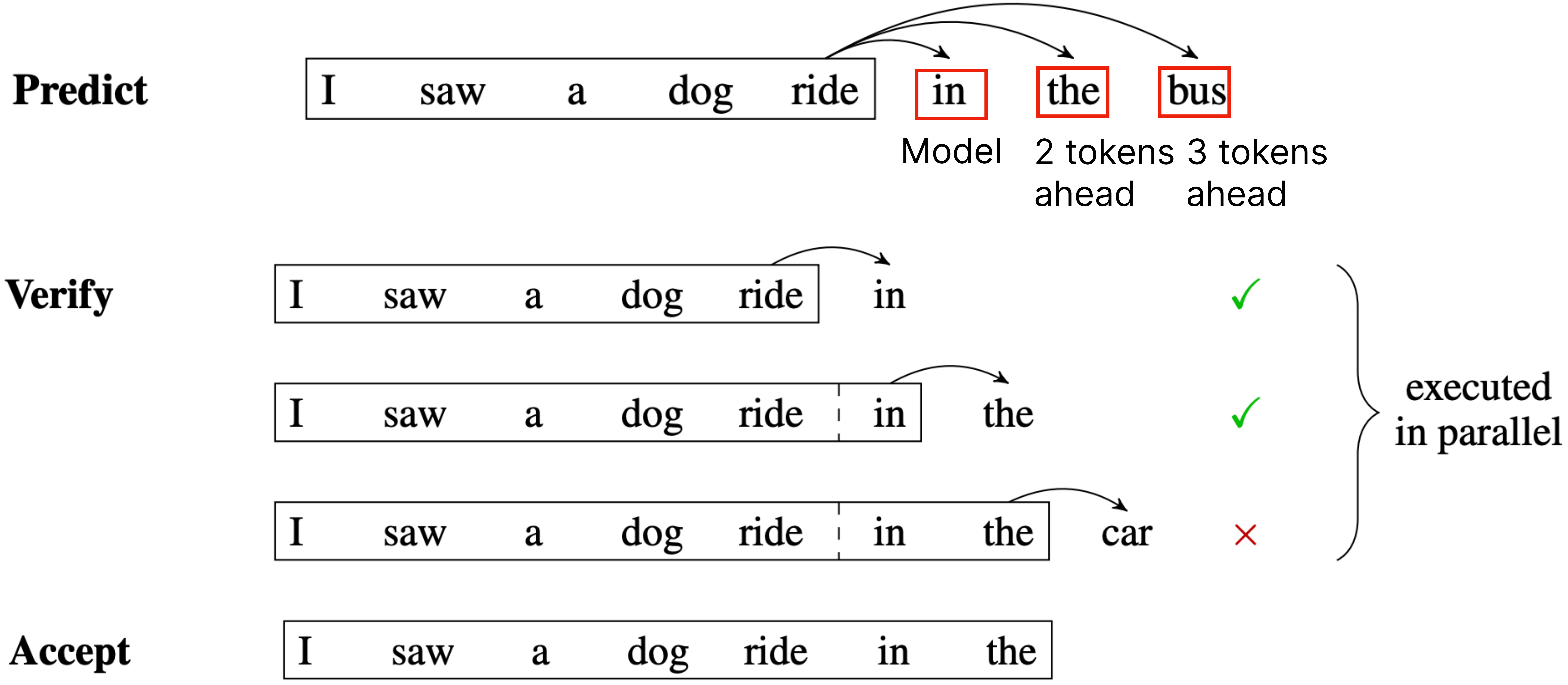
Idea: train auxiliary models that can predict *n* tokens instead (not just 1 token ahead)



Source: [Blockwise Parallel Decoding for Deep Autoregressive Models]()

What if we decoded several tokens in parallel if the problem is decoding one token at a time?

Idea: train auxiliary models that can predict *n* tokens instead (not just 1 token ahead)



Source: Blockwise Parallel Decoding for Deep Autoregressive Models

Why does this make sense?

Decoding several tokens at once reduces the number of forward passes that need to be ran

FLOPS time, per (>1) output token

Total FLOPs: 2 * 70e9 * 32 * 1 ~= 4.48e12

So total time is:

4.48e12 FLOPs / (8 * 312e12 FLOPs/sec) = **0.001s**

Memory load time (from before)

Total bytes: 2 * 70e9 = 140e9

So total load time is:

140e9 bytes / (8 * 1.5e12 bytes / sec) ~= **0.01s**

Results

Model	Source	BLEU	Wall-Clock Speedup
Transformer (beam size 4)	Vaswani et al. (2017)	28.4	
Transformer (beam size 1)	Gu et al. (2018)	22.71	
Transformer (beam size 4)	Gu et al. (2018)	23.45	
Non-autoregressive Transformer	Gu et al. (2018)	17.35	
Non-autoregressive Transformer (+FT)	Gu et al. (2018)	17.69	
Non-autoregressive Transformer (+FT + NPD $s = 10$)	Gu et al. (2018)	18.66	
Non-autoregressive Transformer (+FT + NPD $s = 100$)	Gu et al. (2018)	19.17	
Transformer (beam size 1)	Lee et al. (2018)	23.77	1.20x
Transformer (beam size 4)	Lee et al. (2018)	24.57	1.00x
Iterative refinement Transformer ($i_{\text{dec}} = 1$)	Lee et al. (2018)	13.91	11.39x
Iterative refinement Transformer ($i_{\text{dec}} = 2$)	Lee et al. (2018)	16.95	8.77x
Iterative refinement Transformer ($i_{\text{dec}} = 5$)	Lee et al. (2018)	20.26	3.11x
Iterative refinement Transformer ($i_{\text{dec}} = 10$)	Lee et al. (2018)	21.61	2.01x
Iterative refinement Transformer (Adaptive)	Lee et al. (2018)	21.54	2.39x
Latent Transformer without rescoring	Kaiser et al. (2018)	19.8	
Latent Transformer rescoring top-10	Kaiser et al. (2018)	21.0	
Latent Transformer rescoring top-100	Kaiser et al. (2018)	22.5	
Transformer with distillation (greedy, $k = 1$)	This work	29.11	1.00x
Blockwise parallel decoding for Transformer ($k = 2$)	This work	28.95	1.72x
Blockwise parallel decoding for Transformer ($k = 4$)	This work	28.54	2.69x
Blockwise parallel decoding for Transformer ($k = 6$)	This work	28.11	3.10x
Blockwise parallel decoding for Transformer ($k = 8$)	This work	27.88	3.31x
Blockwise parallel decoding for Transformer ($k = 10$)	This work	27.40	3.04x

Source: Blockwise Parallel Decoding for Deep Autoregressive Models

Take 2: training these are expensive, and not that accurate, so let’s try another model

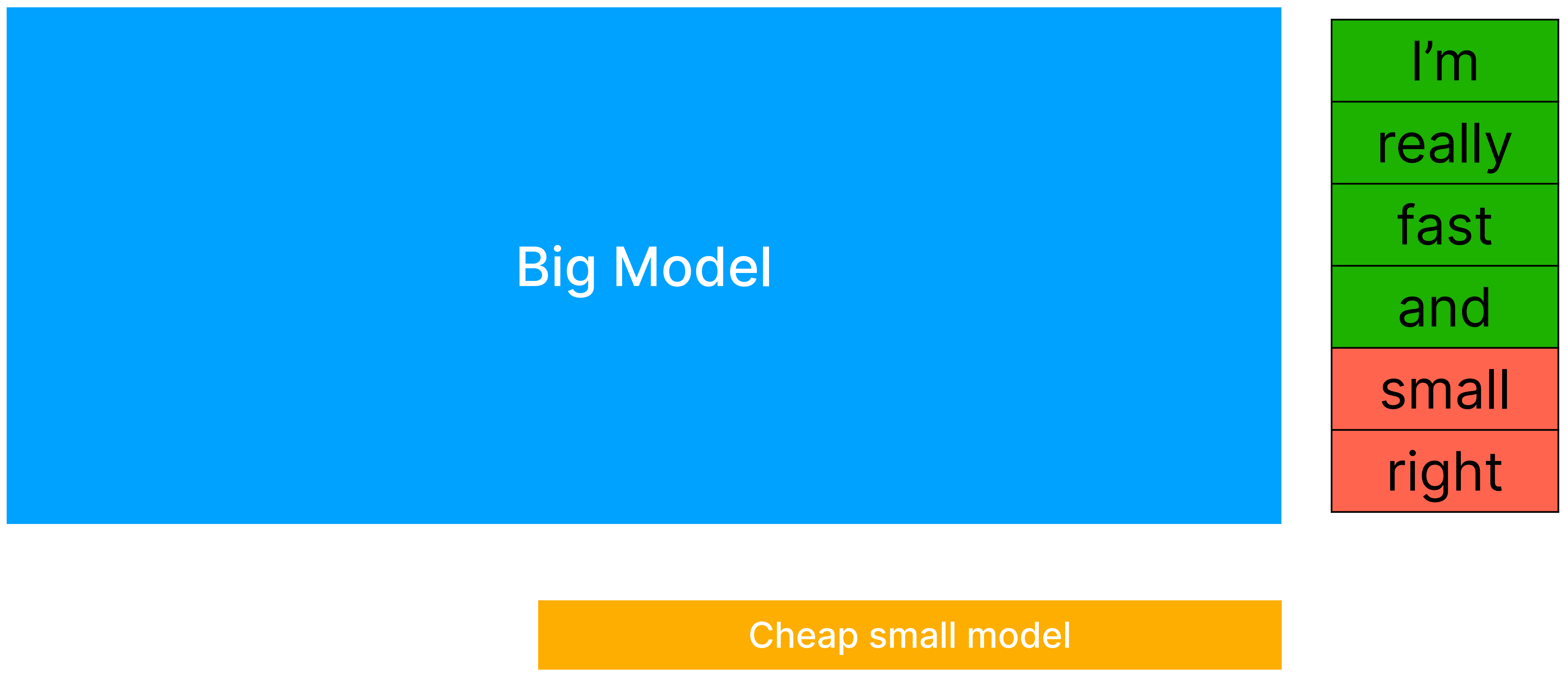
Big Model

Cheap small model

I’m	really	fast	and	small	right

A cheap small model generates tokens

Take 2: training these are expensive, and not that accurate, so let's try another model



The big model verifies them in parallel

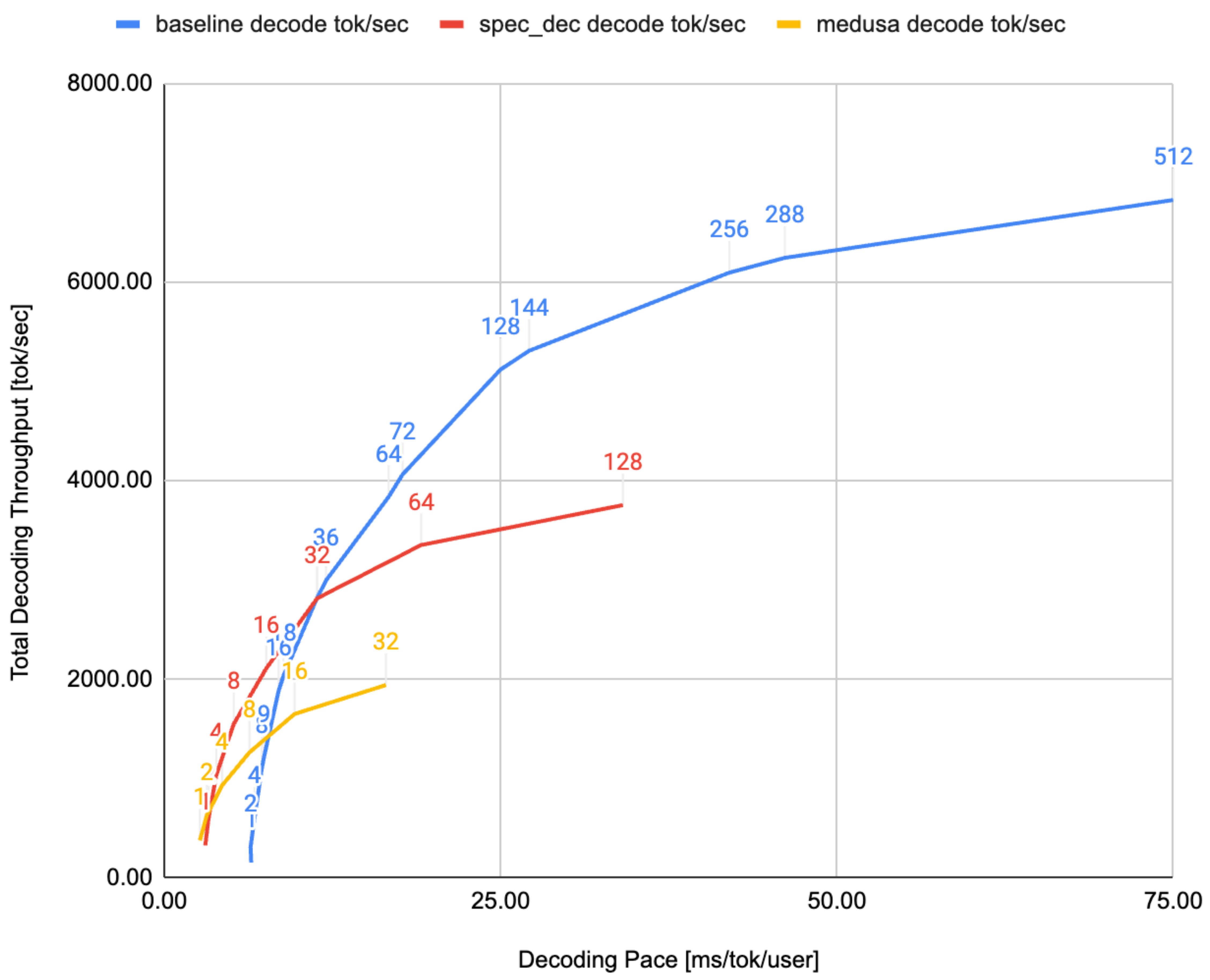
Source: Accelerating Large Language Model Decoding with Speculative Sampling

Results

Sampling Method	Benchmark	Result	Mean Token Time	Speed Up
ArS (Nucleus)	XSum (ROUGE-2)	0.112	14.1ms/Token	1×
SpS (Nucleus)		0.114	7.52ms/Token	1.92×
ArS (Greedy)	XSum (ROUGE-2)	0.157	14.1ms/Token	1×
SpS (Greedy)		0.156	7.00ms/Token	2.01×
ArS (Nucleus)	HumanEval (100 Shot)	45.1%	14.1ms/Token	1×
SpS (Nucleus)		47.0%	5.73ms/Token	2.46×

Source: Accelerating Large Language Model Decoding with Speculative Sampling

Adding this makes sense…but not for high batch sizes



- FLOPs go up but…
- You're doing *k* times as much work, and at batch size *b*, an effective batch size of *k* * *b* might bring you into the compute bound regime
- And that lots of that work is wasted, since you might be wrong

Credit: Abhi Venigalla