CS 498: Machine Learning System
Spring 2025

Minjia Zhang

The Grainger College of Engineering

Today

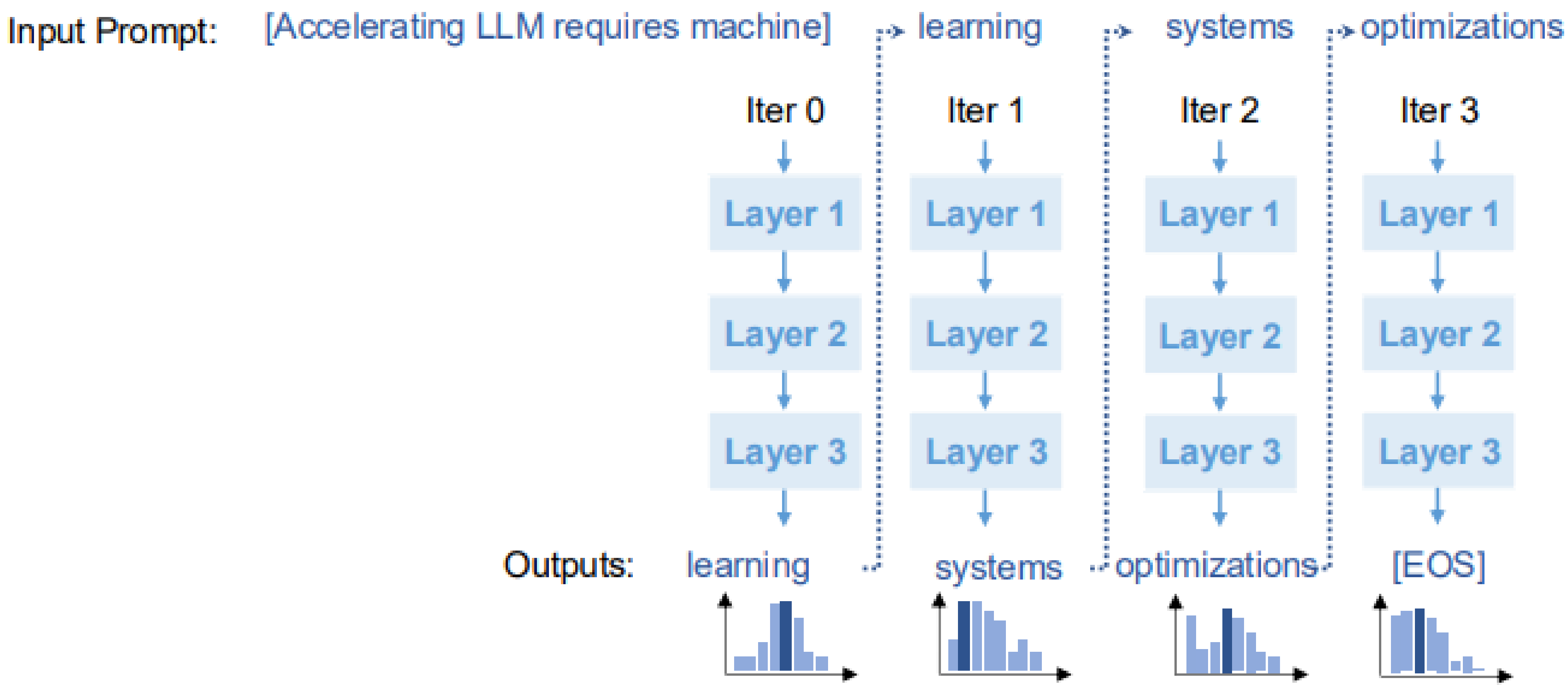
Deep Learning Inference Optimizations

- LLM Inference Basic
- Flash Attention
- Continuous Batching

LLMs are Slow and Expensive to Serve

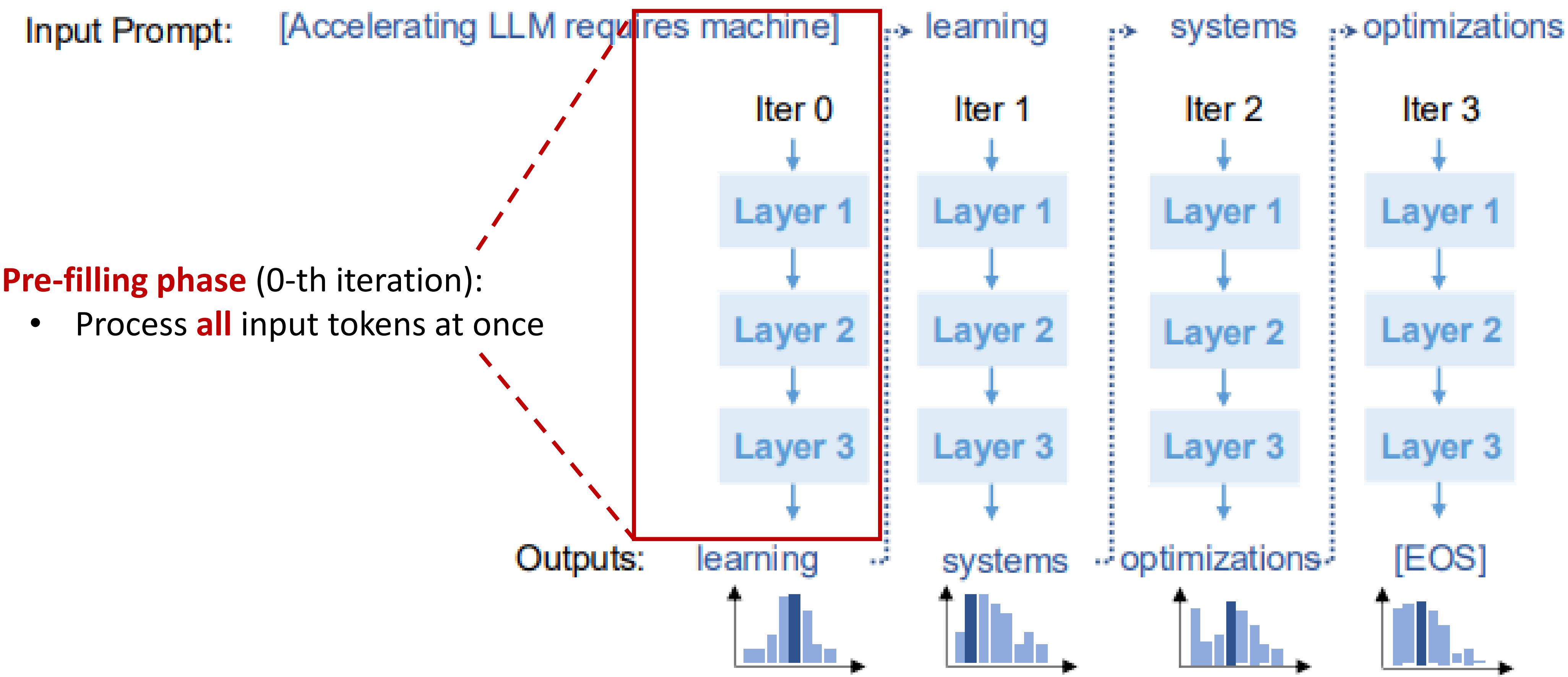
- At least ten A100-40GB GPUs to serve 175B GPT-3 in half-precision
- Generating 256 tokens takes ~20 seconds
- Cannot process requests in parallel
  - Per-request key-value cache takes 3GB GPU memory

Generative LLM Inference: Autoregressive Decoding

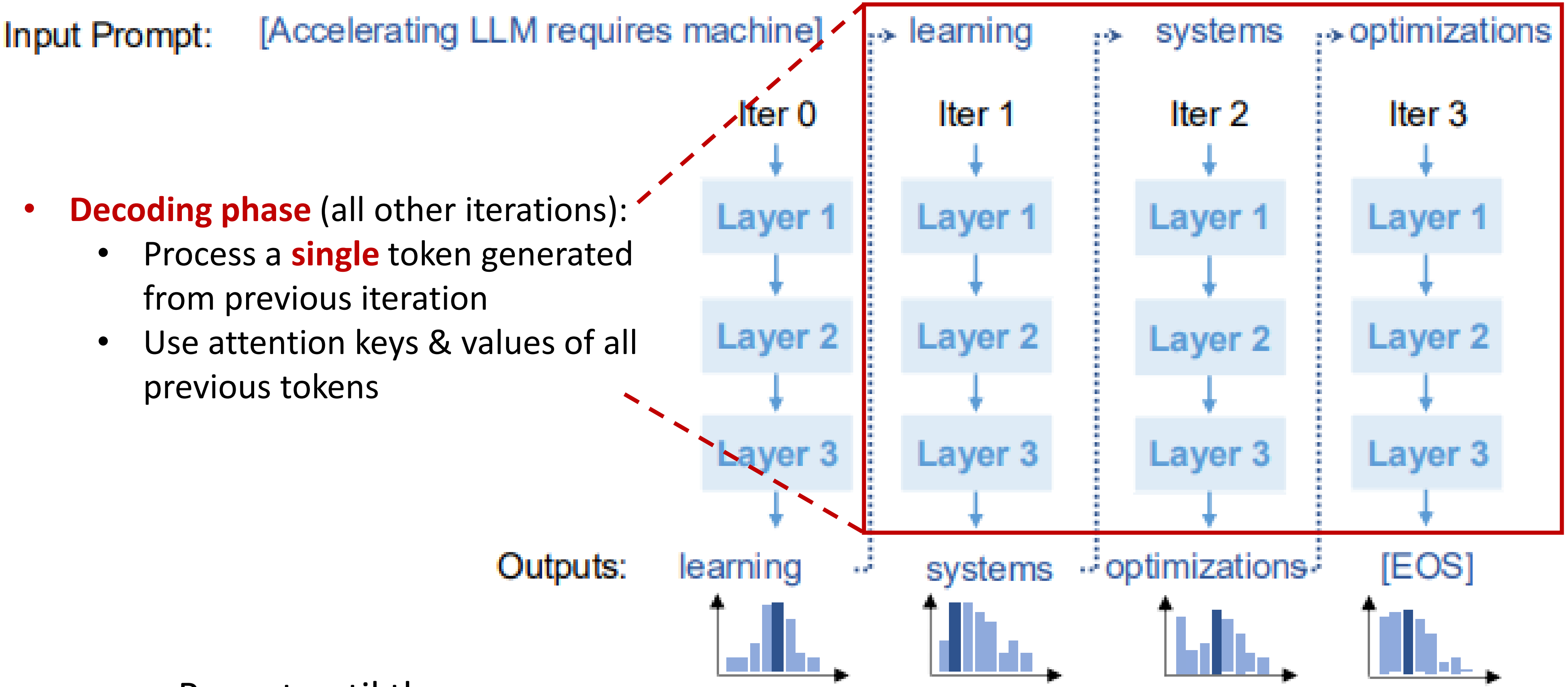


Generative LLM Inference: Autoregressive Decoding

- **Pre-filling phase** (0-th iteration):
  - Process **all** input tokens at once



Generative LLM Inference: Autoregressive Decoding



- **Decoding phase** (all other iterations):
  - Process a **single** token generated from previous iteration
  - Use attention keys & values of all previous tokens

Repeat until the sequence
- Reaches the pre-defined maximum length (e.g., 2048 tokens)
- Generates the stop tokens (e.g., “<|end of sequence|>”)

Generative LLM Inference: Important Metrics

- **Time to First Token (TTFT)**: Measures how quickly users begin to see the model output token after submitting a query.
  - Critical for real-time interactions
  - Driven by prompt processing time and the generation of the first token
- **Time per Output Token (TPOT):** Time taken to generate each output token.
  - Impacts user perception of speed (e.g., 100ms/token = 10 tokens/second)
- **E2E Latency = TTFT + (TPOT * the number of generated tokens)**
  - Total time to generate the complete response
- **Throughput:** Number of tokens generated per second across all requests by the inference server

[LLM Inference Performance Engineering: Best Practices | Databricks]

Optimizing LLM Inference: Goals and Tradeoffs

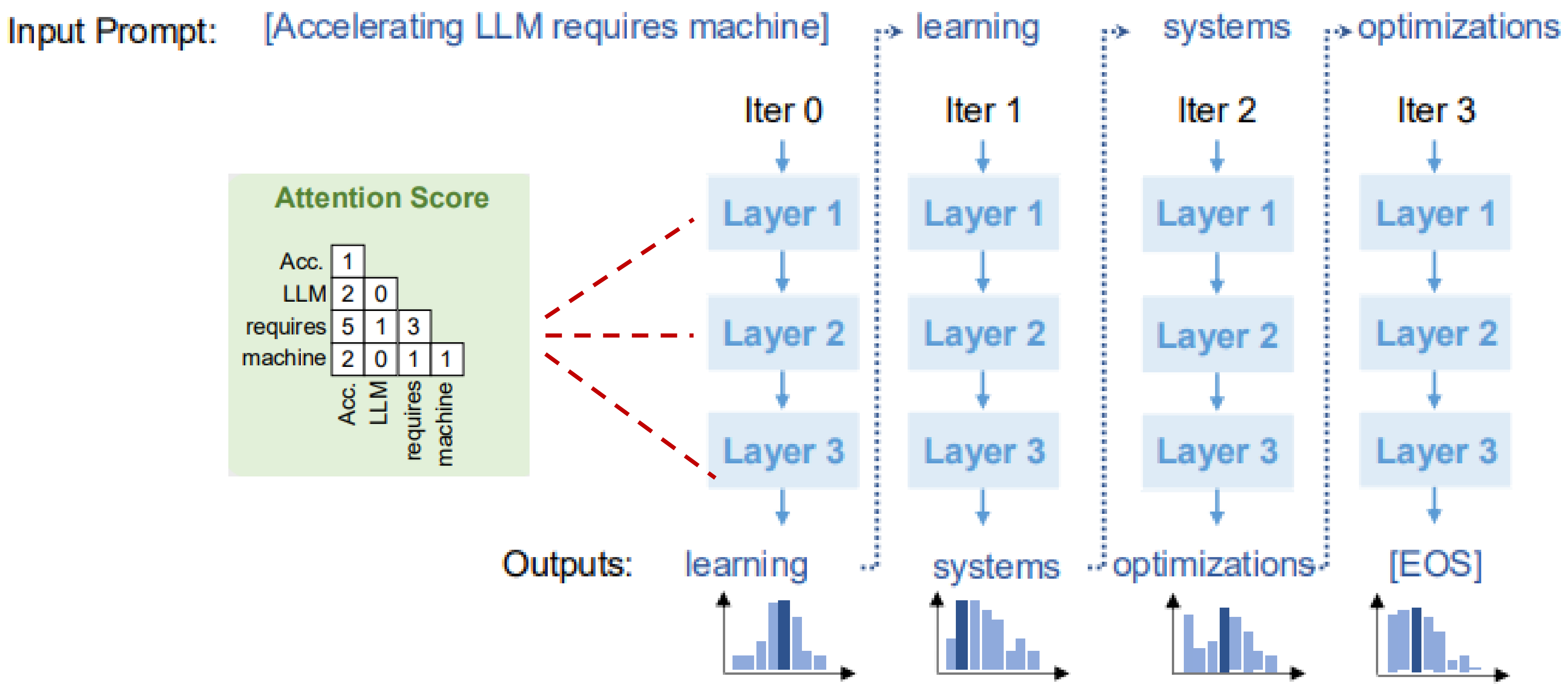
- **Goal:** Minimize TTFT, maximize throughput, and reduce TPOT
- **Throughput vs. TPOP Tradeoff**: Processing multiple queries concurrently increases throughput extends TPOT for each user.

Today

DL Inference

- LLM Inference
- FlashAttention (cont.)
- Continuous Batching

LLM Inference: Bottleneck from Attention Calculation

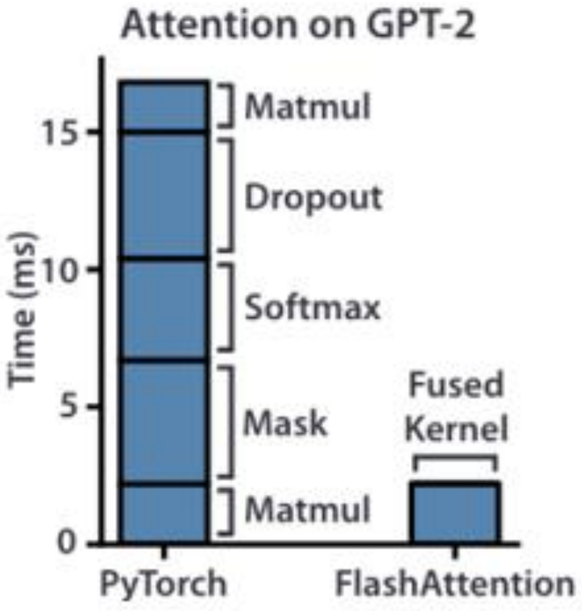


FlashAttention

- First introduced at HAET workshop @ICML July 2022
- Published @ NeurIPS Dec 2022
- Very useful even though many people probably don't even know they are using it!

FlashAttention

- First introduced at HAET workshop @ICML July 2022
- Published @ NeurIPS Dec 2022
- Very useful even though many people probably don't even know they are using it!

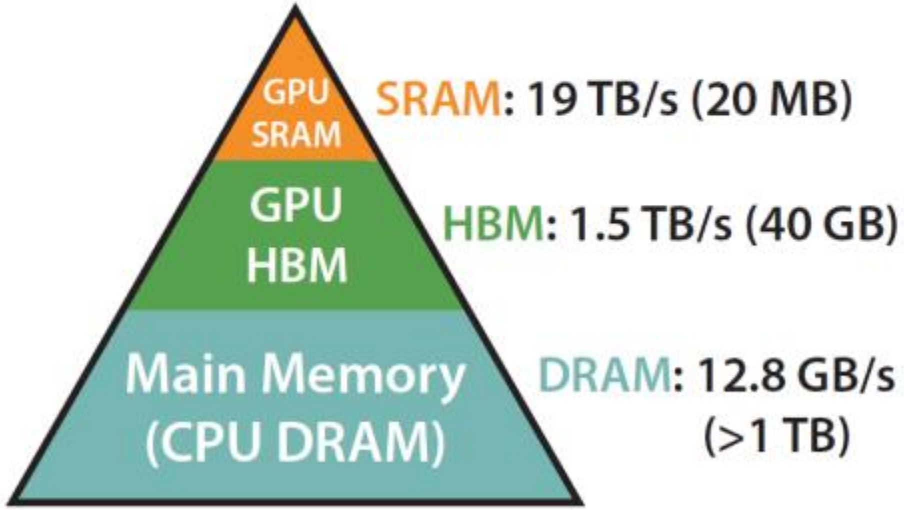


Massive adoption (5 months)

Revisit: GPU Memory Hierarchy

Memory is arranged hierarchically

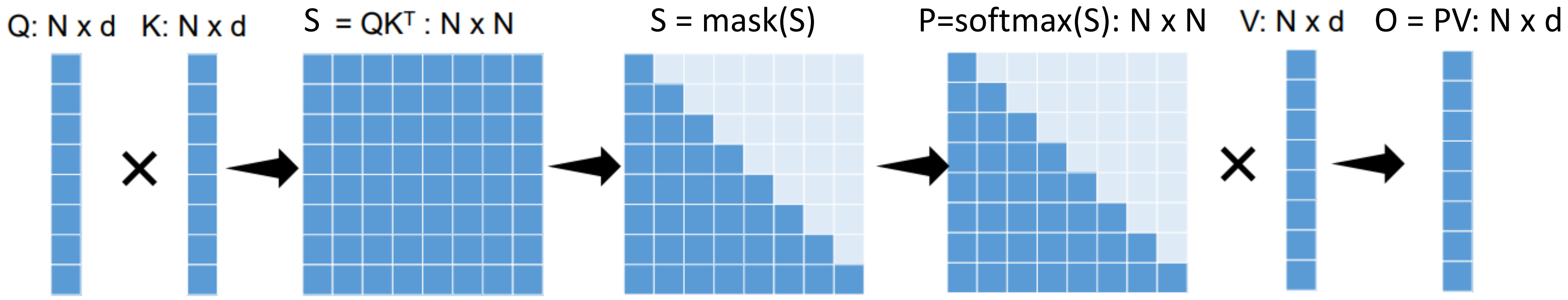
- GPU SRAM is small, and supports the fastest access
- GPU HBM is larger but with much slower access
- CPU DRAM is huge, but the slowest of all



Memory Hierarchy with Bandwidth & Memory Size

Standard Attention

Attention: $O = \text{Softmax}(QK^T)\ V$



$$\mathbf{S} = \mathbf{QK}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \text{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{PV} \in \mathbb{R}^{N\times d},$$

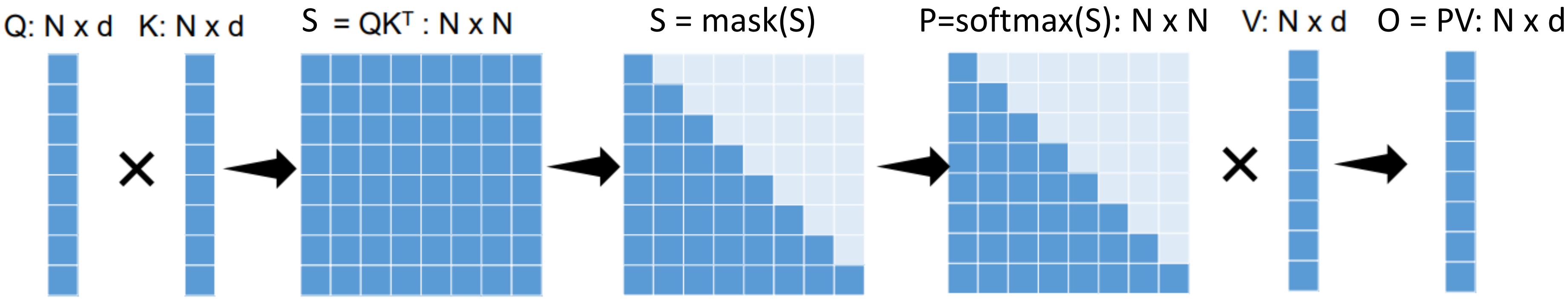
Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{QK}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \text{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{PV}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Standard Attention

Attention: O = Softmax(QK^T) V



$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d},$$

Algorithm 0 Standard Attention Implementation

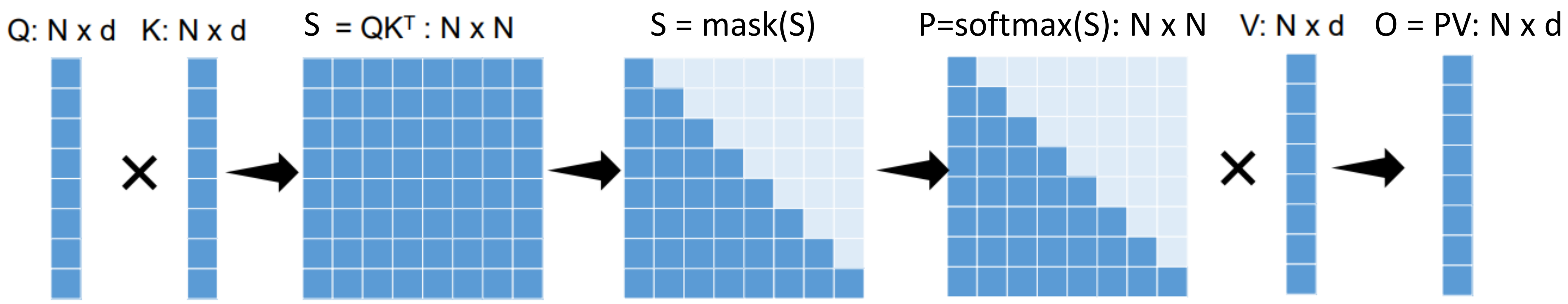
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Question: What are limitations of standard attention implementation?

Standard Attention

Attention: $O = \text{Softmax}(QK^T)\ V$



Challenges:

- Repeated reads/writes from GPU HBM
- Large intermediate results
- Cannot scale to long sequences due to $O(N^2)$ intermediate results

FlashAttention

- Three key ideas are combined to obtain FlashAttention
  - Operator fusion: Use a single kernel that includes all operators during attention computation to avoid kernel launching overhead and intermediate data movement
  - Tiling: compute the attention block by block so that we don’t have to load everything into SRAM at once
  - Recomputation: don’t store the full attention matrix in forward, but just recompute during the backward pass

Operator Fusion

Version A: Usually, we compute a neural network one operator at a time by moving operation input to GPU SRAM (fast/small), doing some computation, then returning the output to GPU HBM (slow/large)

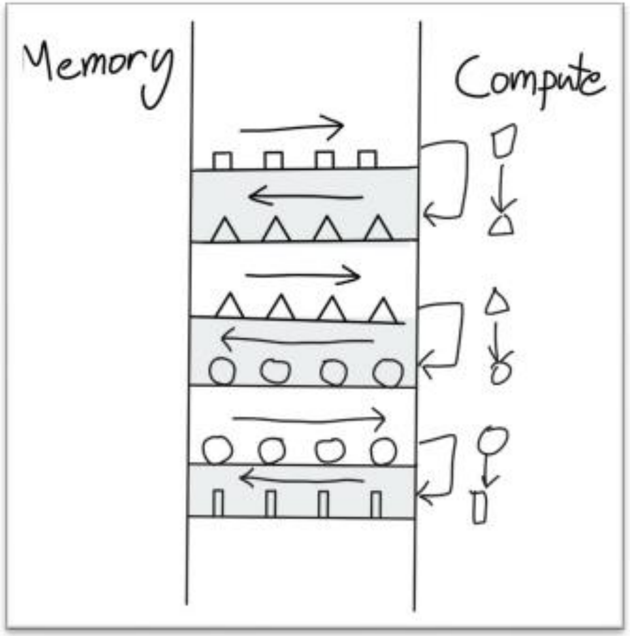
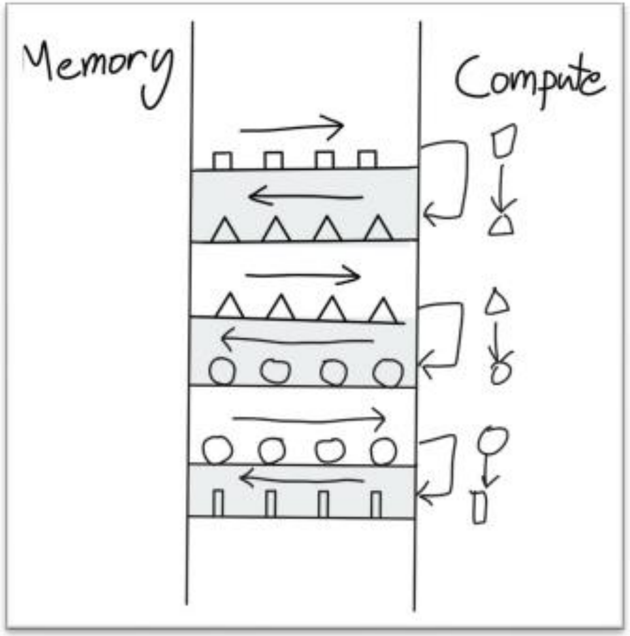


Figure from https://horace.io/brrr_intro.html

Operator Fusion

Version A: Usually, we compute a neural network one operator at a time by moving operation input to GPU SRAM (fast/small), doing some computation, then returning the output to GPU HBM (slow/large)



Version B: Operator fusion instead moves the original input to GPU SRAM (fast/small), does a whole sequence of layer computations without ever touching HBM, and then returns the final layer output to GPU HBM (slow/large)

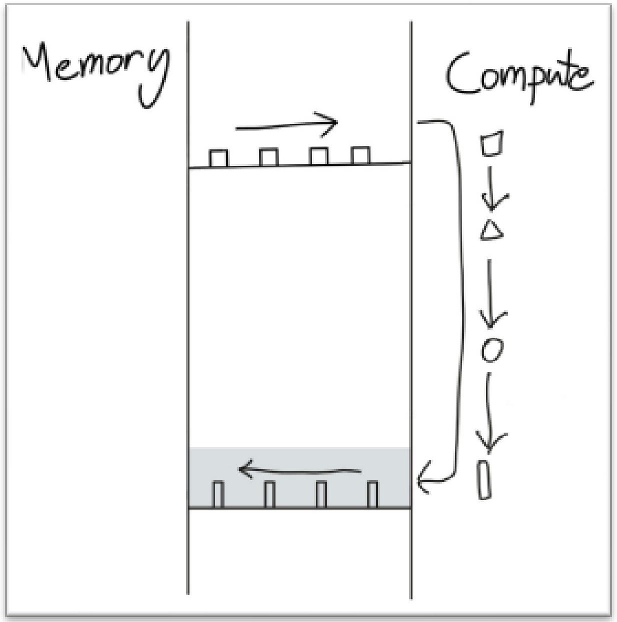


Figure from https://horace.io/brrr_intro.html

Operator Fusion

Version A: Usually, we compute a neural network one operator at a time by moving operation input to GPU SRAM (fast/small), doing some computation, then returning the output to GPU HBM (slow/large)

Version B: Operator fusion instead moves the original input to GPU SRAM (fast/small), does a whole sequence of layer computations without ever touching HBM, and then returns the final layer output to GPU HBM (slow/large)

Version A is how standard attention is implemented

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d},$$

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Operator Fusion

Version A: Usually, we compute a neural network one operator at a time by moving operation input to GPU SRAM (fast/small), doing some computation, then returning the output to GPU HBM (slow/large)

Version B: Operator fusion instead moves the original input to GPU SRAM (fast/small), does a whole sequence of layer computations without ever touching HBM, and then returns the final layer output to GPU HBM (slow/large)

Version B improves performance but requires CUDA code rewriting (or through DL compilers)

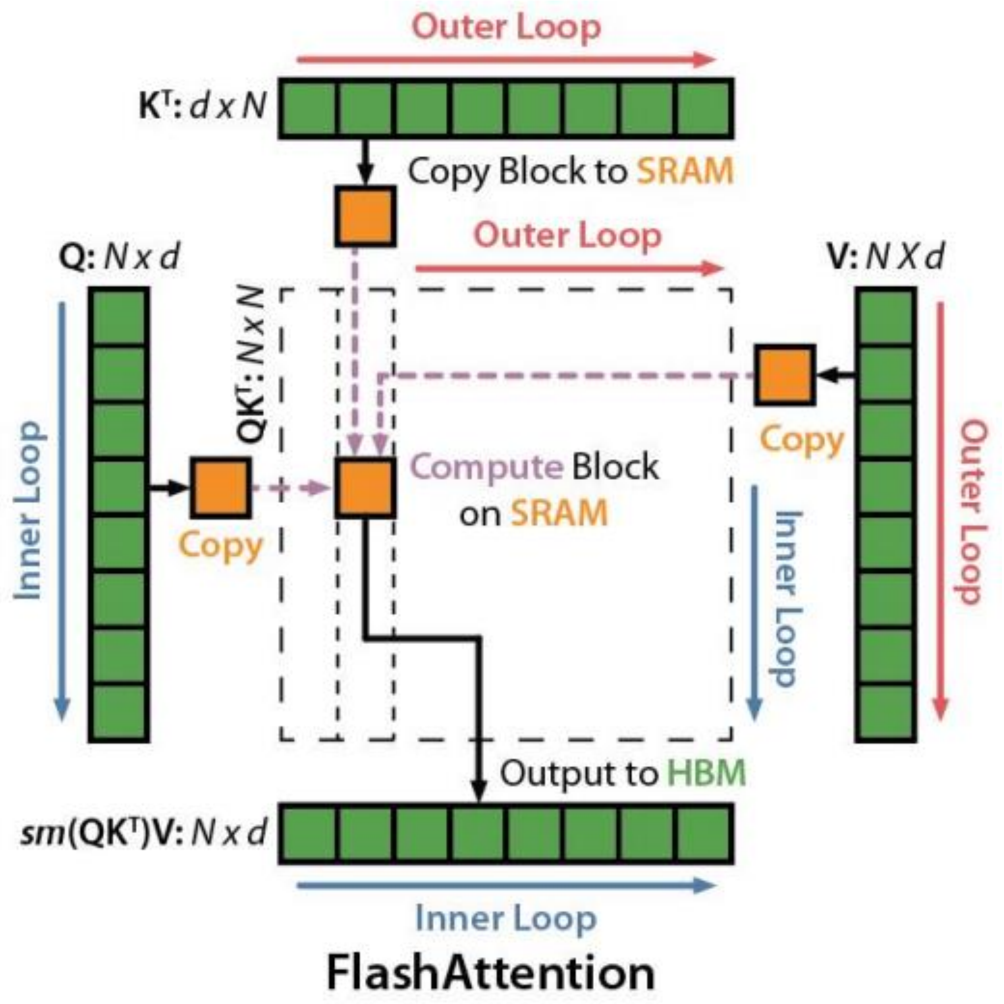
$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d},$$

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$ in HBM.

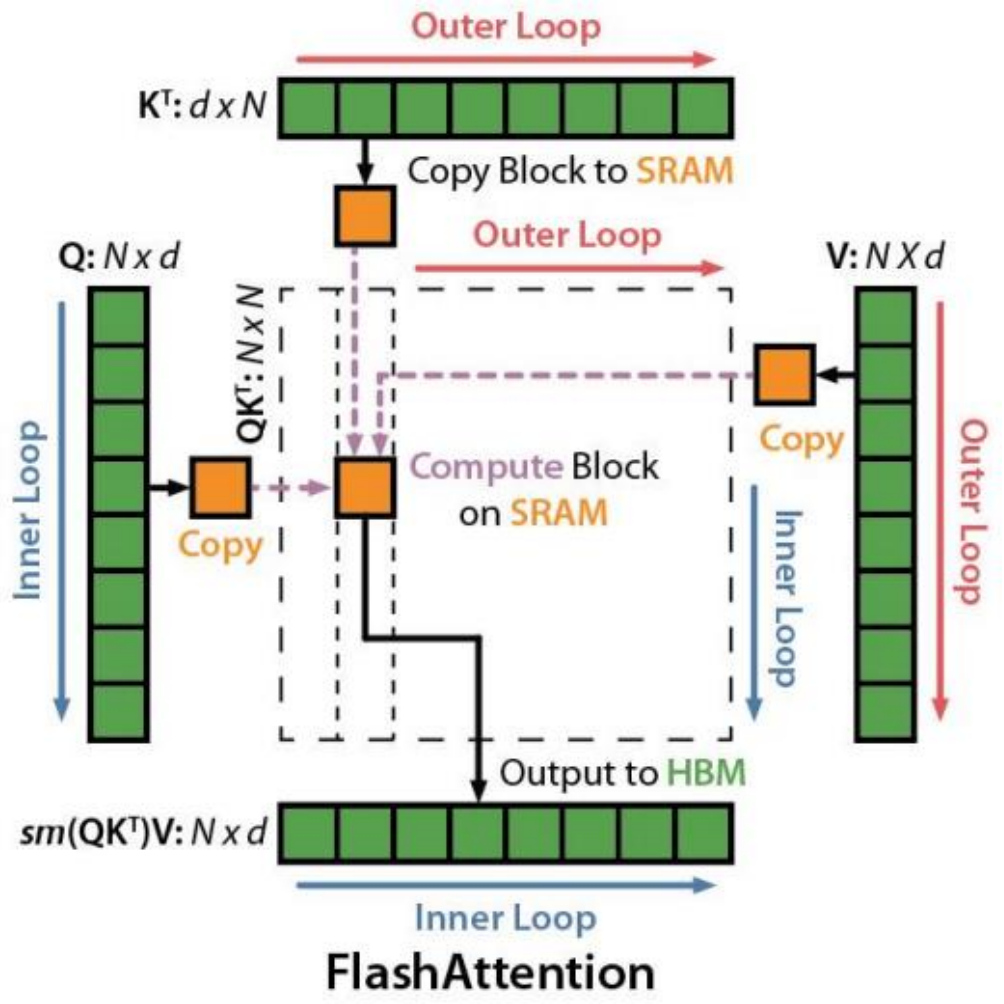
1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Tiling: Decompose Large Attention Calculation into Smaller Blocks



Tiling: Decompose Large Attention Calculation into Smaller Blocks

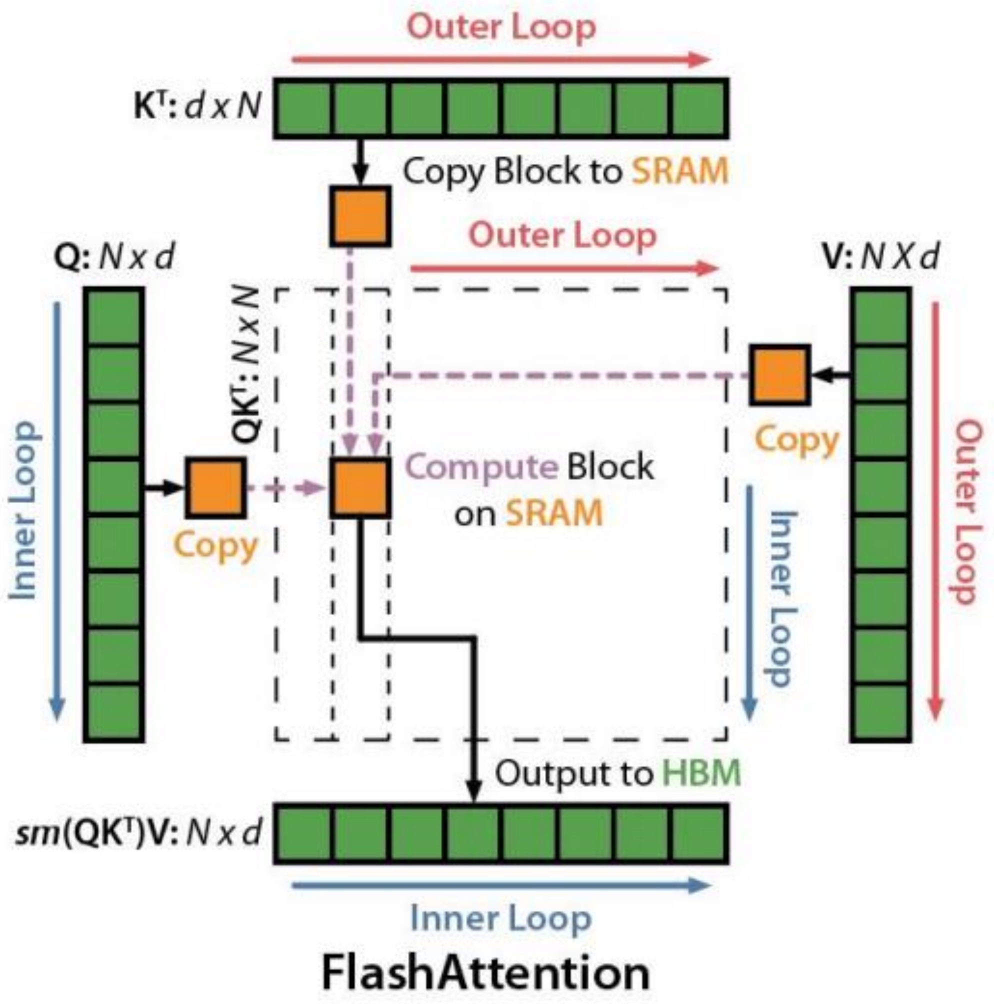
1. Load inputs by blocks from global HBM to SRAM



Tiling: Decompose Large Attention Calculation into Smaller Blocks

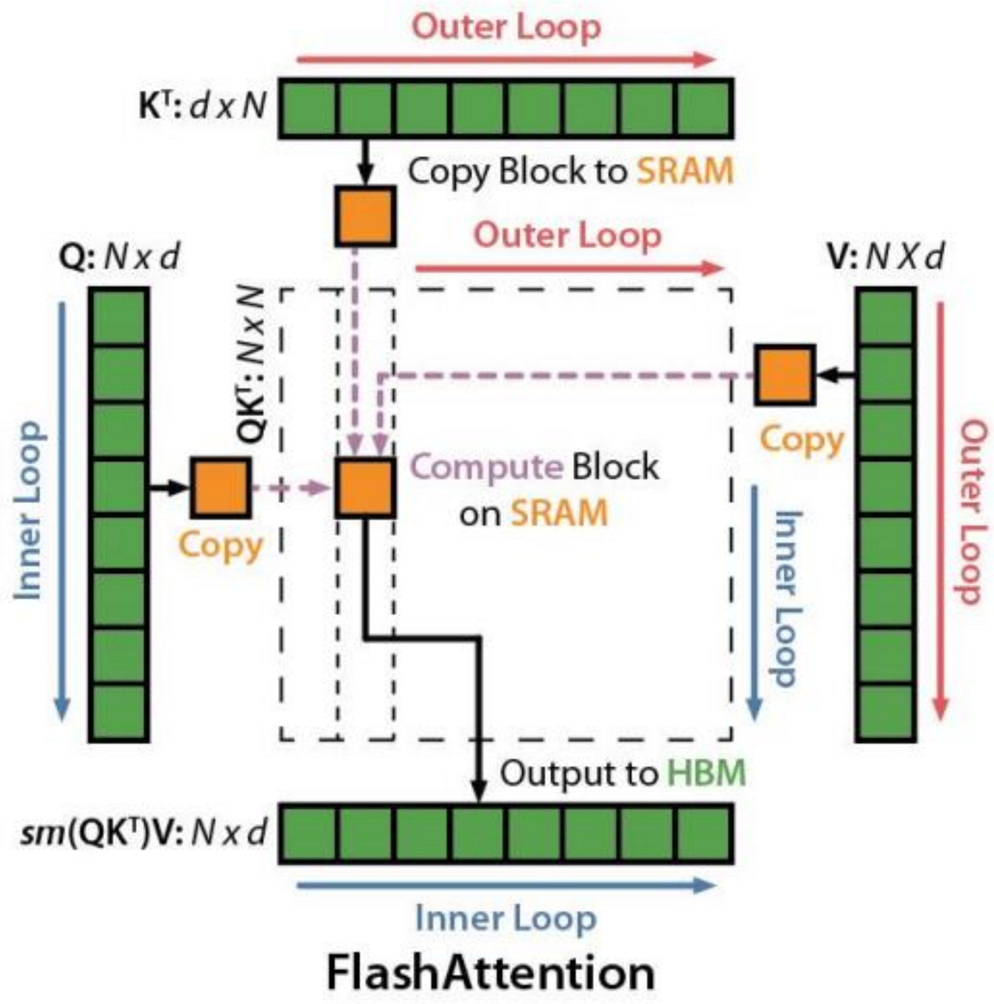
1. Load inputs by blocks from global HBM to SRAM

2. On chip, compute attention output wrt the block



Tiling: Decompose Large Attention Calculation into Smaller Blocks

1. Load inputs by blocks from global HBM to SRAM
2. On chip, compute attention output wrt the block
3. Update output in HBM by blocks



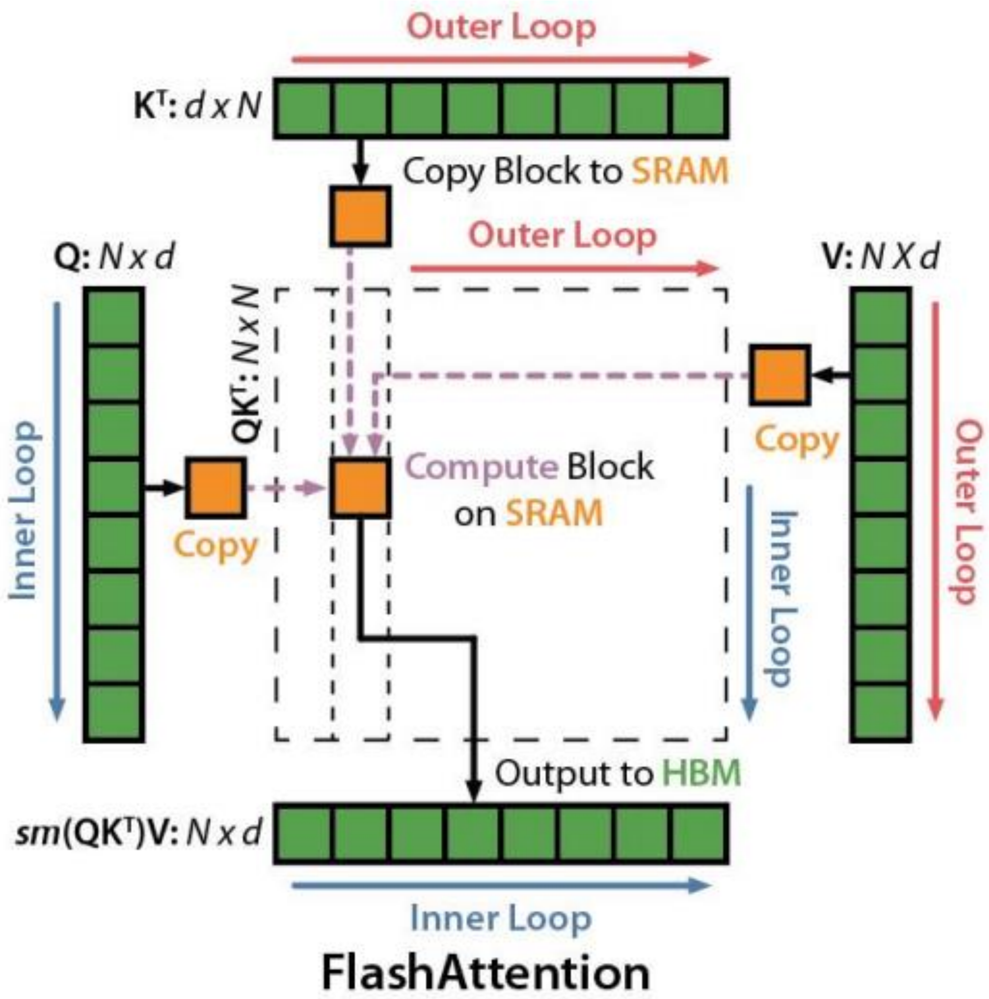
Tiling: Decompose Large Attention Calculation into Smaller Blocks

1. Load inputs by blocks from global HBM to SRAM

2. On chip, compute attention output wrt the block

3. Update output in HBM by blocks

(Everything in a single kernel)



Tiling: Decompose Large Attention Calculation into Smaller Blocks

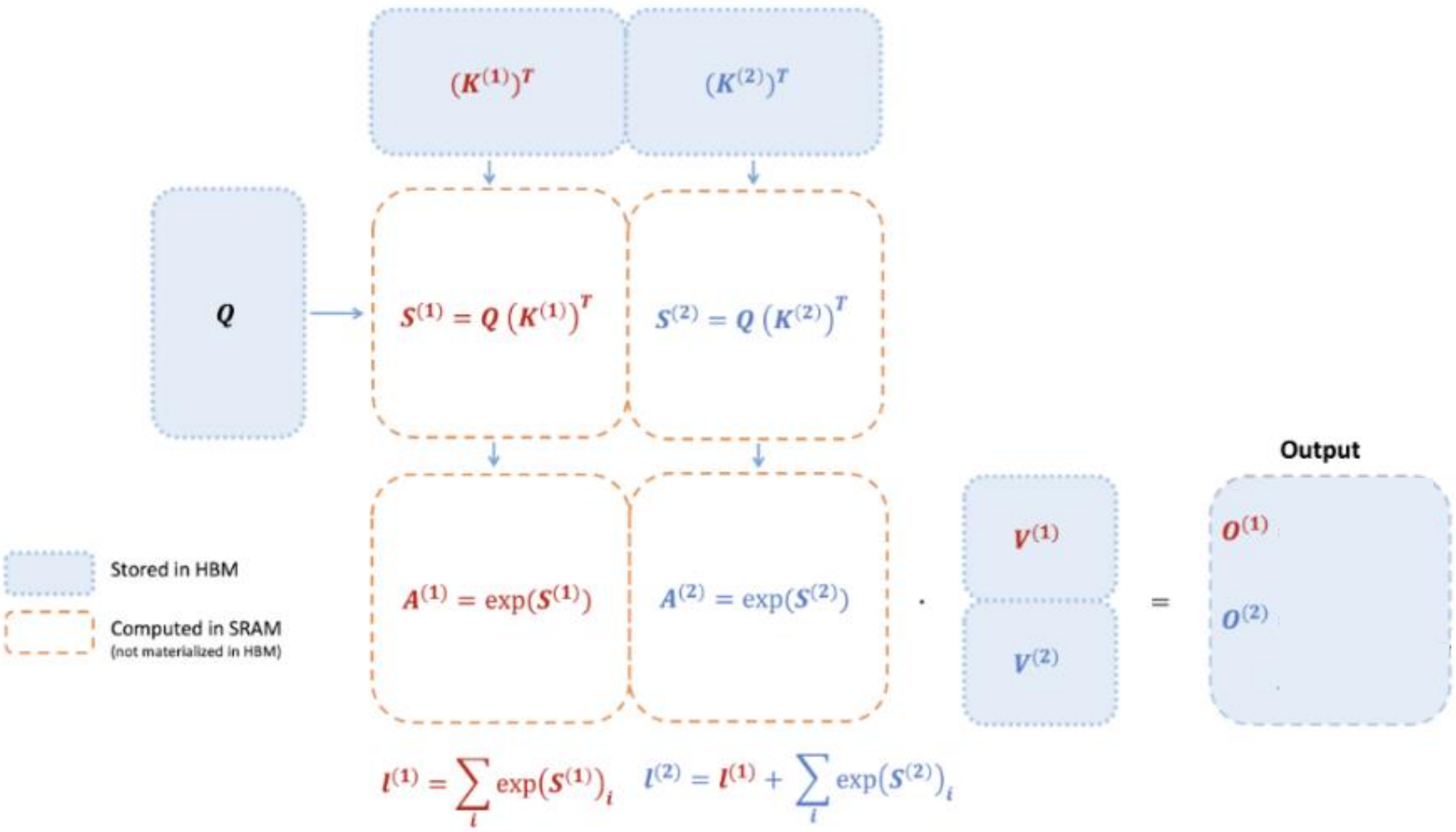
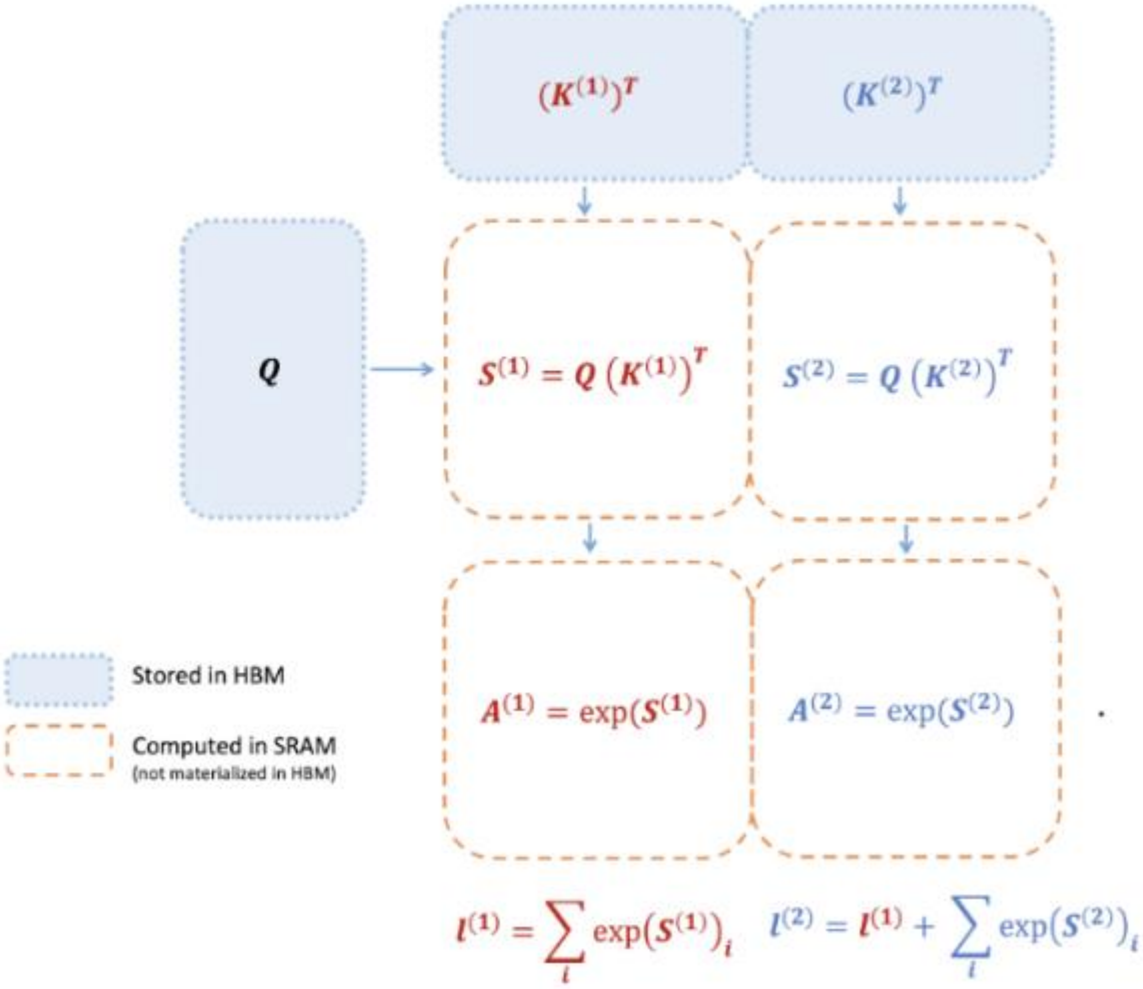
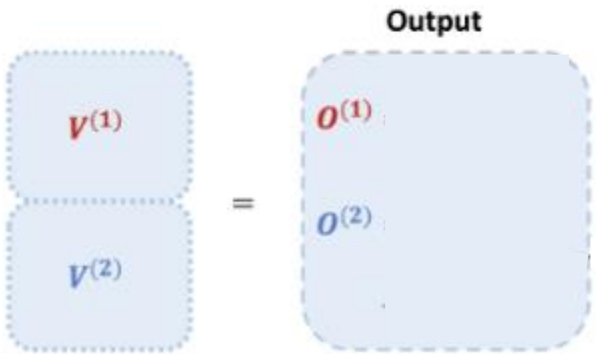


Figure from http://arxiv.org/abs/2307.08691

Tiling: Decompose Large Attention Calculation into Smaller Blocks





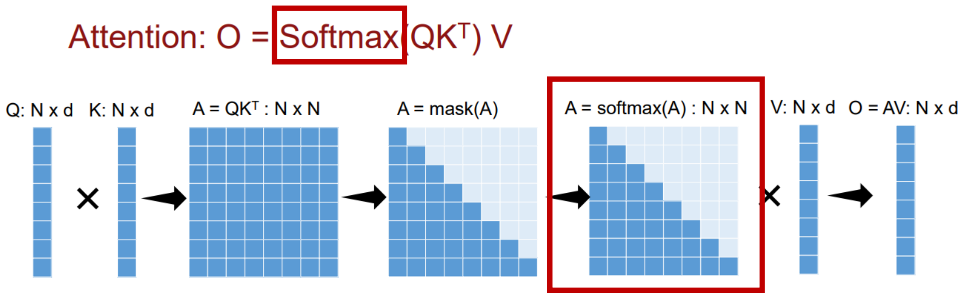
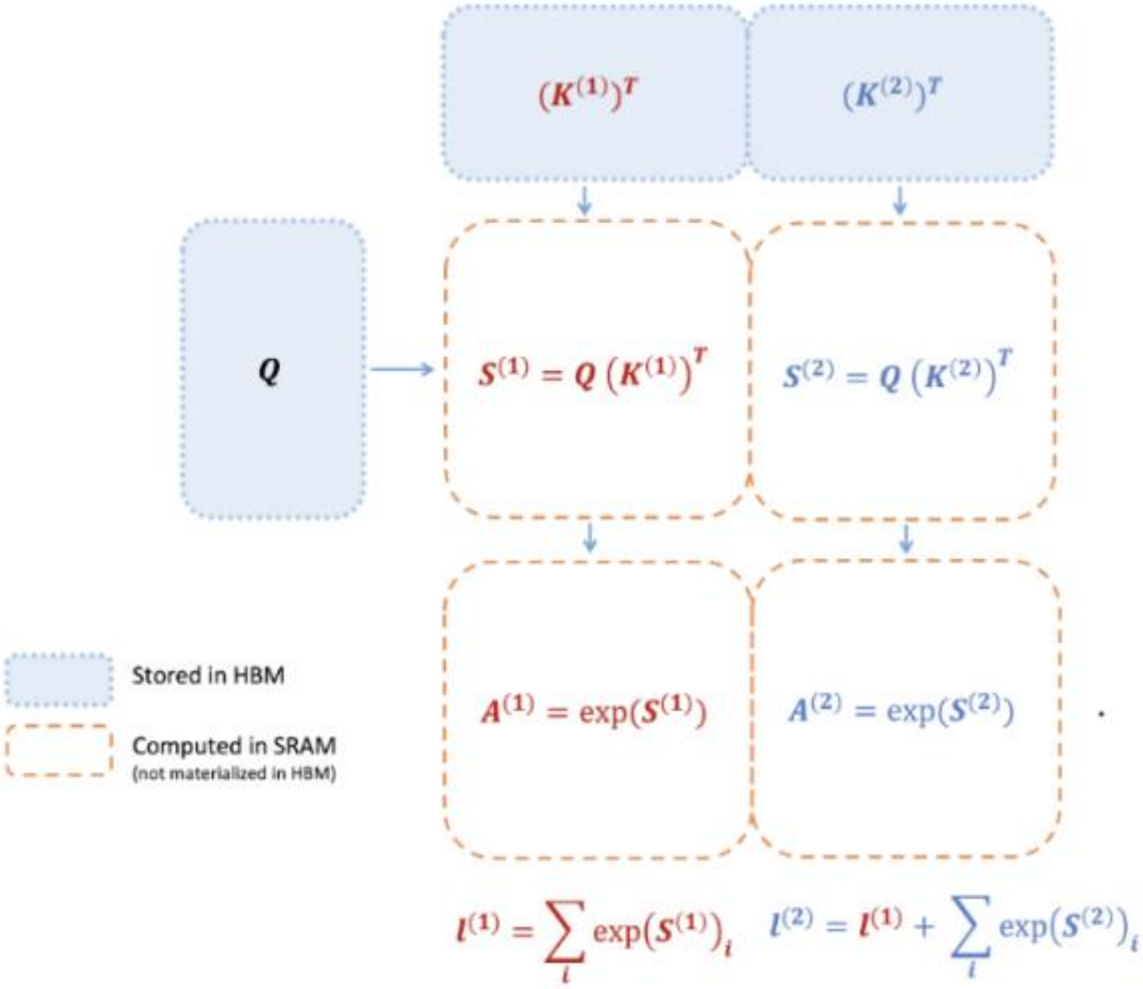
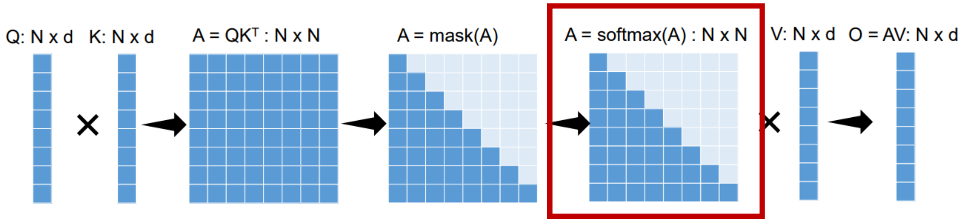


Figure from http://arxiv.org/abs/2307.08691

Tiling: Decompose Large Attention Calculation into Smaller Blocks

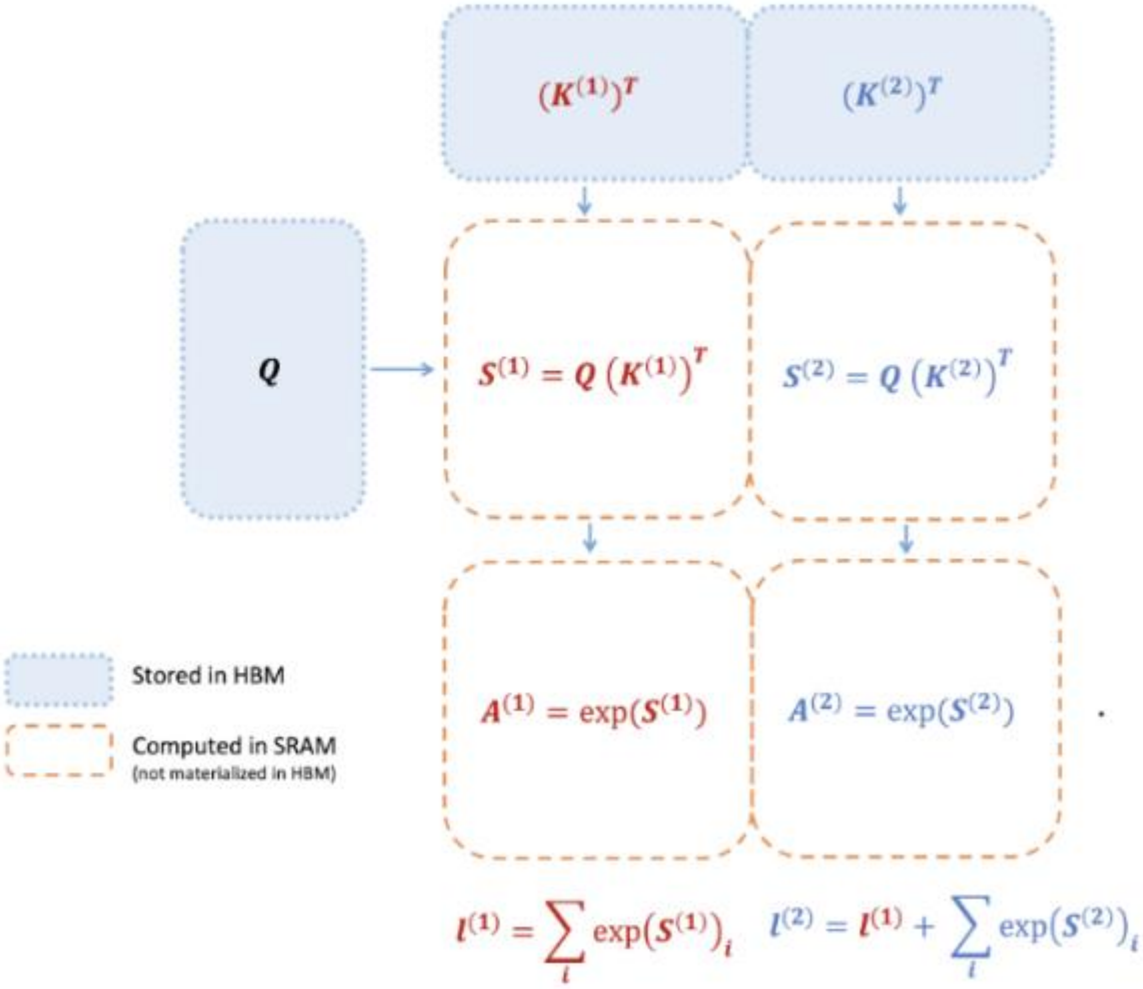




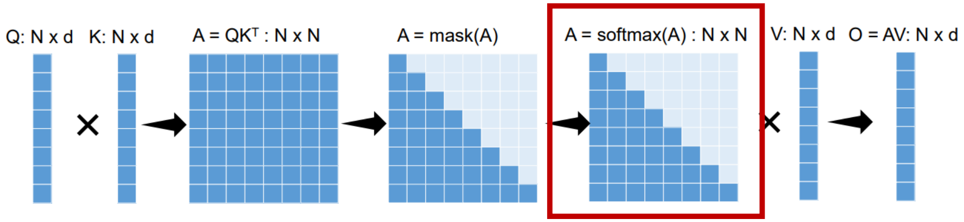
Discussion: How would you solve this problem if you were transported back to 2022?

Figure from http://arxiv.org/abs/2307.08691

Tiling: Decompose Large Attention Calculation into Smaller Blocks



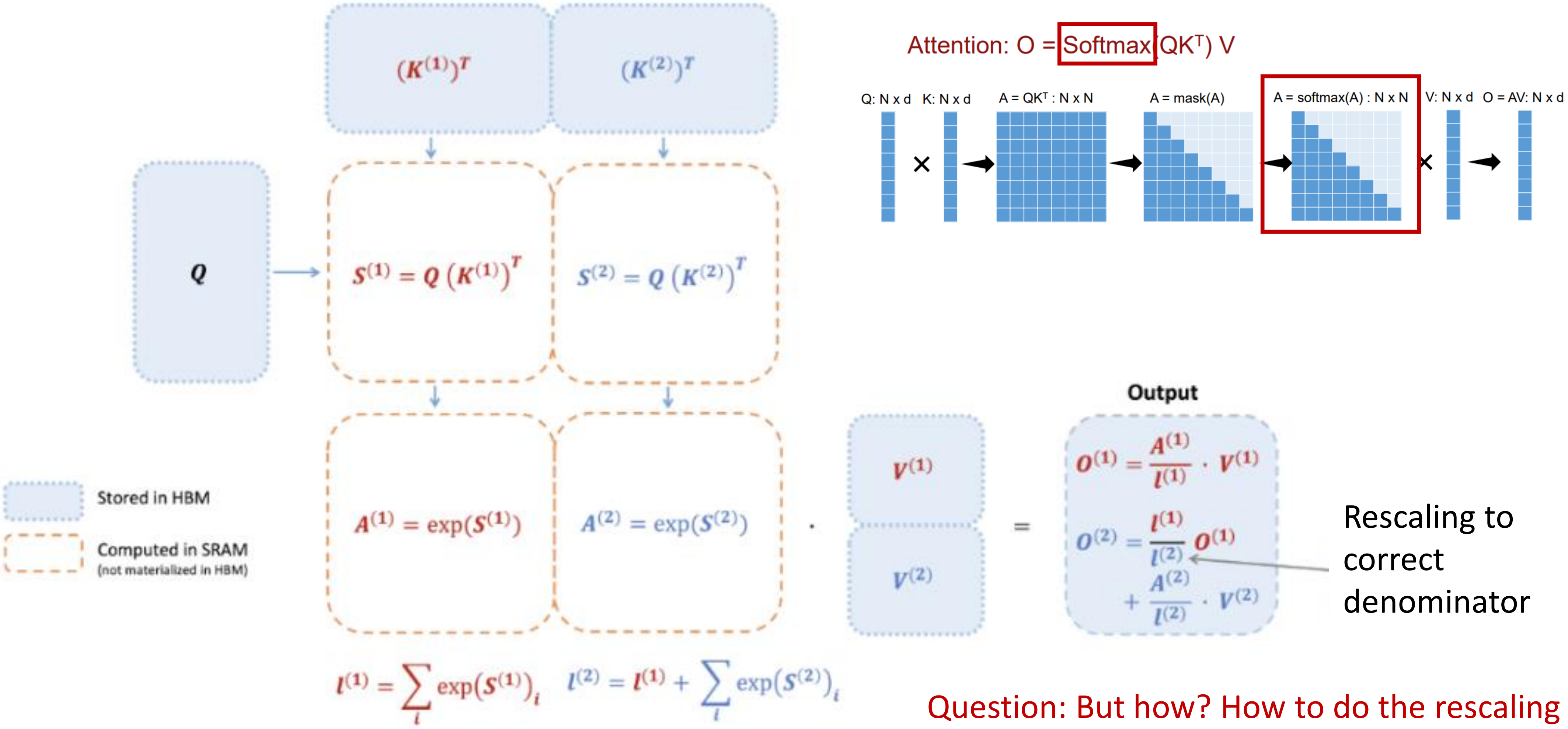
Attention: O = Softmax(QK^T) V



Rescaling to correct denominator

Figure from http://arxiv.org/abs/2307.08691

Tiling: Decompose Large Attention Calculation into Smaller Blocks



Rescaling to correct denominator

Question: But how? How to do the rescaling while retaining the correctness?

Figure from http://arxiv.org/abs/2307.08691

How to Implement Softmax? Standard Softmax

For a vector $x \in \mathbb{R}^B$, softmax is computed as:

$$\mathrm{softmax}(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

How to Implement Softmax? Standard Softmax

For a vector $x \in \mathbb{R}^B$, softmax is computed as:

$$\text{softmax}(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Question: Standard softmax is rarely used in practice, why?

How to Implement Softmax? Standard Softmax

For a vector $x \in \mathbb{R}^B$, softmax is computed as:

$$\text{softmax}(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Problem: Exponentials can explode

How to Implement Softmax? Standard Softmax

For a vector $x \in \mathbb{R}^B$, softmax is computed as:

$$\text{softmax}(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Problem: Exponentials can explode

e^x grows very quickly
Assume x = 1000, $e^{1000} \approx 10^{434}$

Too large to store in FP32, overflow (INF)

Stable Softmax

Subtracting the max value from the input vector before applying the exp function, which helps prevent overflow issue

1. Compute the maximum value in x:

$$m(x) := \max_i x_i$$

Stable Softmax

Subtracting the max value from the input vector before applying the exp function, which helps prevent overflow issue

1. Compute the maximum value in x:

$$m(x) := \max_i x_i$$

2. Compute the adjusted exponentials:

$$f(x) := \left[e^{x_1 - m(x)}, e^{x_2 - m(x)}, \ldots, e^{x_B - m(x)}\right]$$

Stable Softmax

Subtracting the max value from the input vector before applying the exp function, which helps prevent overflow issue

1. Compute the maximum value in x:

$$m(x) := \max_i x_i$$

2. Compute the adjusted exponentials:

$$f(x) := \left[e^{x_1 - m(x)}, e^{x_2 - m(x)}, \ldots, e^{x_B - m(x)}\right]$$

3. Compute the normalization denominator:

$$\ell(x) := \sum_i f(x)_i$$

Stable Softmax

Subtracting the max value from the input vector before applying the exp function, which helps prevent overflow issue

1. Compute the maximum value in x:

$$m(x) := \max_i x_i$$

2. Compute the adjusted exponentials:

$$f(x) := \left[e^{x_1 - m(x)}, e^{x_2 - m(x)}, \ldots, e^{x_B - m(x)}\right]$$

3. Compute the normalization denominator:

$$\ell(x) := \sum_i f(x)_i$$

4. Compute the final softmax:

$$\text{softmax}(x) = \frac{f(x)}{\ell(x)}$$

Stable Softmax

Subtracting the max value from the input vector before applying the exp function, which helps prevent overflow issue

1. Compute the maximum value in x:

$$m(x) := \max_i x_i$$

2. Compute the adjusted exponentials:

$$f(x) := \left[e^{x_1 - m(x)}, e^{x_2 - m(x)}, \ldots, e^{x_B - m(x)}\right]$$

Now the largest exponent is $e^0 = 1$, which is very safe.

All other terms become e^{x-m}, which are less than or equal to 1, avoiding overflow.

3. Compute the normalization denominator:

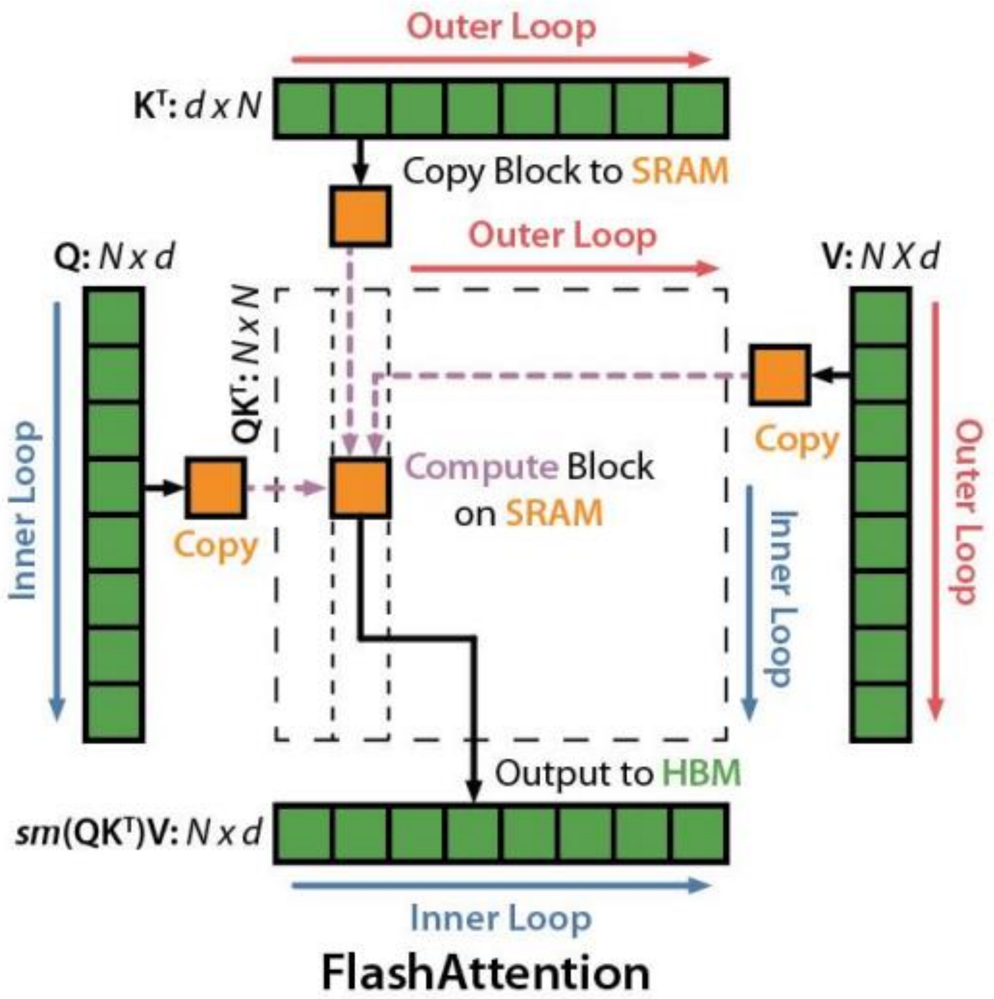
$$\ell(x) := \sum_i f(x)_i$$

4. Compute the final softmax:

$$\text{softmax}(x) = \frac{f(x)}{\ell(x)}$$

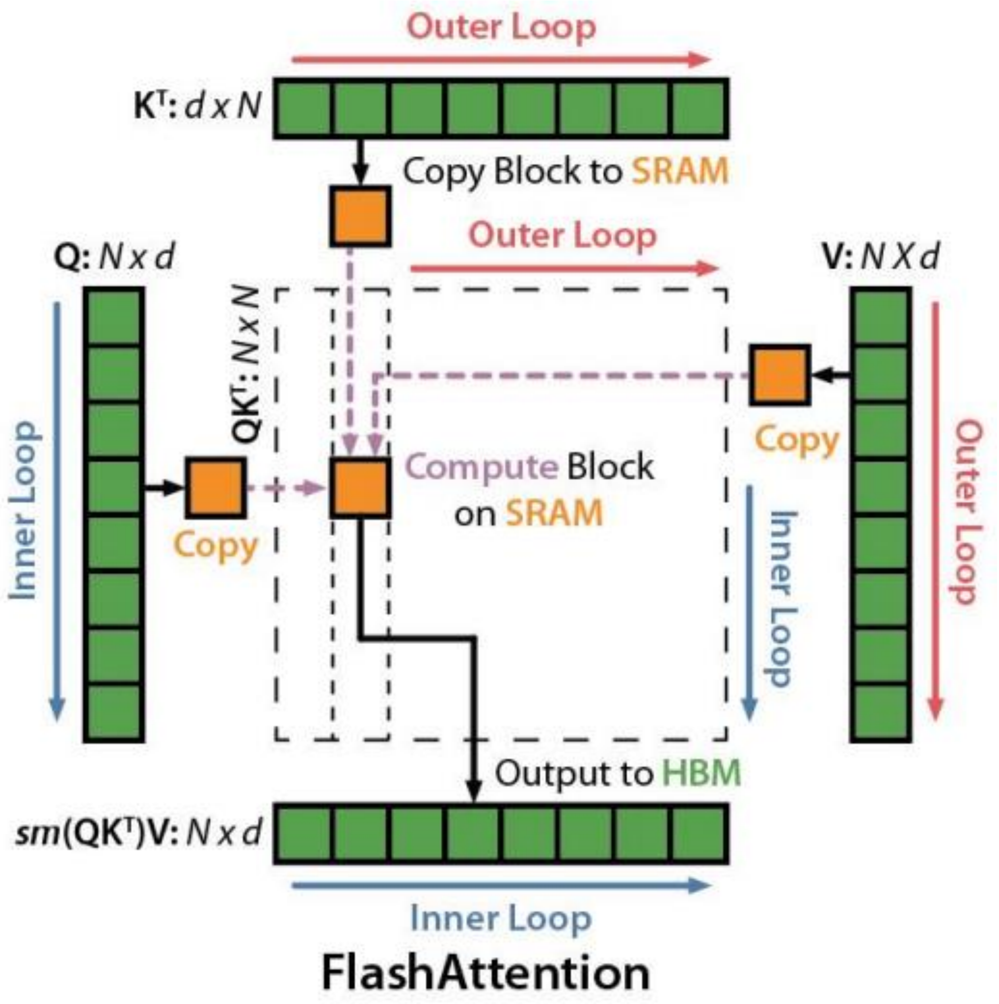
FlashAttention: Tiling + Stable Softmax

1. Load inputs by blocks from global HBM to SRAM
2. On chip, compute attention output wrt the block
3. Update output in HBM by blocks



FlashAttention: Tiling + (Online) Stable Softmax

1. Load inputs by blocks from global HBM to SRAM
2. On chip, compute attention output wrt the block
3. Update output in HBM by online stable softmax



FlashAttention: (Online) Stable Softmax

Let's say we have two blocks $x^{(1)}$ and $x^{(2)}$ each of size B. The concatenated vector is:

$$x = [x^{(1)}, x^{(2)}] \in \mathbb{R}^{2B}$$

FlashAttention: (Online) Stable Softmax

Let's say we have two blocks $x^{(1)}$ and $x^{(2)}$ each of size B. The concatenated vector is:

$$x = [x^{(1)}, x^{(2)}] \in \mathbb{R}^{2B}$$

1. Track the max value across blocks

$$m(x) = \max(m(x^{(1)}), m(x^{(2)}))$$

FlashAttention: (Online) Stable Softmax

Let's say we have two blocks $x^{(1)}$ and $x^{(2)}$ each of size B. The concatenated vector is:

$$x = [x^{(1)}, x^{(2)}] \in \mathbb{R}^{2B}$$

1. Track the max value across blocks

$$m(x) = \max(m(x^{(1)}), m(x^{(2)}))$$

2. Compute adjusted exponentials (rescaling)

$$f(x) = \left[e^{m(x^{(1)})-m(x)} f(x^{(1)}), e^{m(x^{(2)})-m(x)} f(x^{(2)})\right]$$

FlashAttention: (Online) Stable Softmax

Let's say we have two blocks $x^{(1)}$ and $x^{(2)}$ each of size B. The concatenated vector is:

$$x = [x^{(1)}, x^{(2)}] \in \mathbb{R}^{2B}$$

1. Track the max value across blocks

$$m(x) = \max(m(x^{(1)}), m(x^{(2)}))$$

2. Compute adjusted exponentials (rescaling)

$$f(x) = \left[e^{m(x^{(1)})-m(x)} f(x^{(1)}), e^{m(x^{(2)})-m(x)} f(x^{(2)})\right]$$

3. Compute the normalization denominator (incrementally):

$$\ell(x) = e^{m(x^{(1)})-m(x)} \ell(x^{(1)}) + e^{m(x^{(2)})-m(x)} \ell(x^{(2)})$$

FlashAttention: (Online) Stable Softmax

Let's say we have two blocks $x^{(1)}$ and $x^{(2)}$ each of size B. The concatenated vector is:

$$x = [x^{(1)}, x^{(2)}] \in \mathbb{R}^{2B}$$

1. Track the max value across blocks

$$m(x) = \max(m(x^{(1)}), m(x^{(2)}))$$

2. Compute adjusted exponentials (rescaling)

$$f(x) = \left[e^{m(x^{(1)})-m(x)} f(x^{(1)}), e^{m(x^{(2)})-m(x)} f(x^{(2)})\right]$$

3. Compute the normalization denominator (incrementally):

$$\ell(x) = e^{m(x^{(1)})-m(x)} \ell(x^{(1)}) + e^{m(x^{(2)})-m(x)} \ell(x^{(2)})$$

4. Compute the final softmax:

$$\mathrm{softmax}(x) = \frac{f(x)}{\ell(x)}$$

FlashAttention: (Online) Stable Softmax

Let's say we have two blocks $x^{(1)}$ and $x^{(2)}$ each of size B. The concatenated vector is:

$$x = [x^{(1)}, x^{(2)}] \in \mathbb{R}^{2B}$$

1. Track the max value across blocks

$$m(x) = \max(m(x^{(1)}), m(x^{(2)}))$$

2. Compute adjusted exponentials (rescaling)

$$f(x) = \left[e^{m(x^{(1)})-m(x)} f(x^{(1)}), e^{m(x^{(2)})-m(x)} f(x^{(2)})\right]$$

3. Compute the normalization denominator (incrementally):

$$\ell(x) = e^{m(x^{(1)})-m(x)} \ell(x^{(1)}) + e^{m(x^{(2)})-m(x)} \ell(x^{(2)})$$

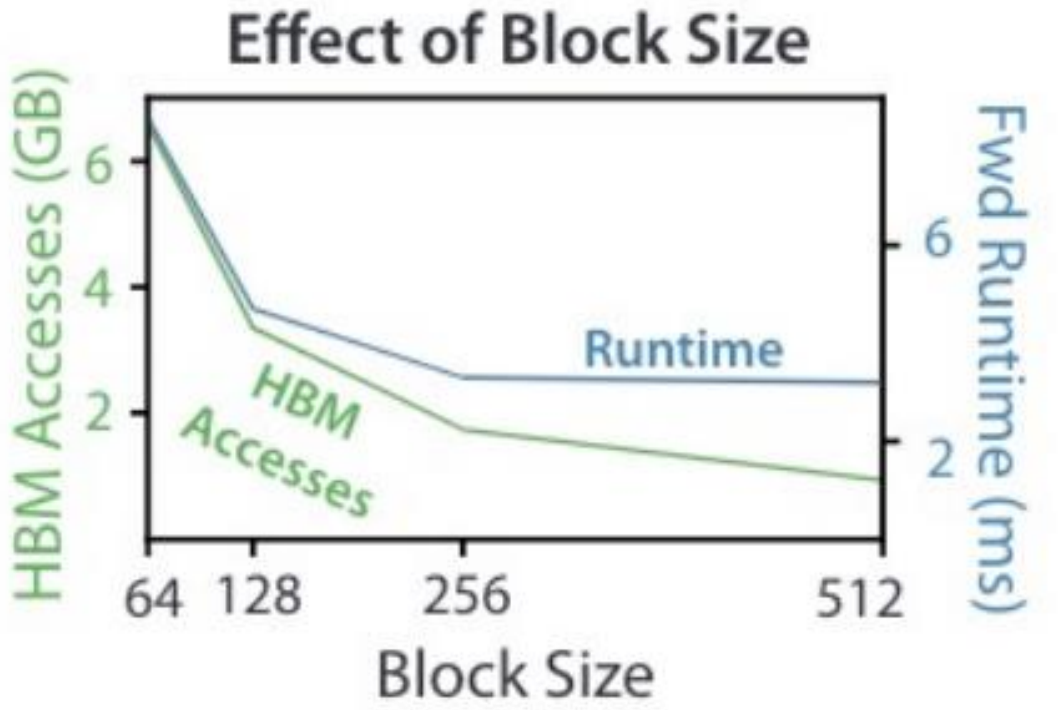
4. Compute the final softmax:

$$\text{softmax}(x) = \frac{f(x)}{\ell(x)}$$

Only need to track intermediate statistics ($m(x^{(i)})$, $l(x^{(i)})$) to compute softmax one block at a time

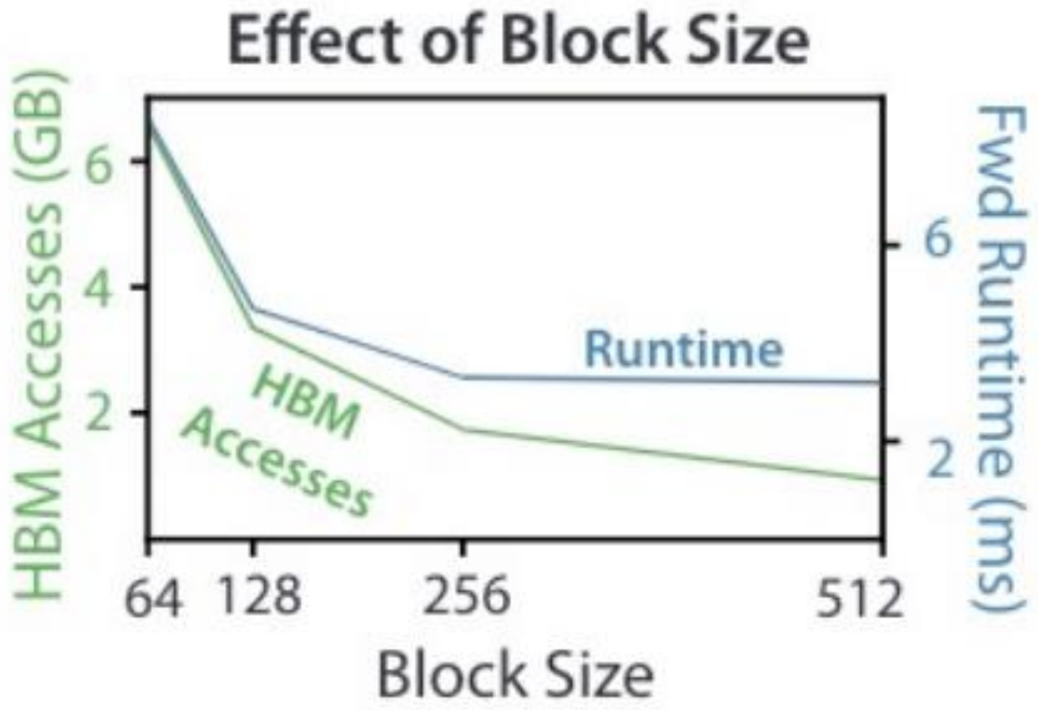
FlashAttention Results

The algorithm is performing exact attention, no reduction in perplexity or quality of the model



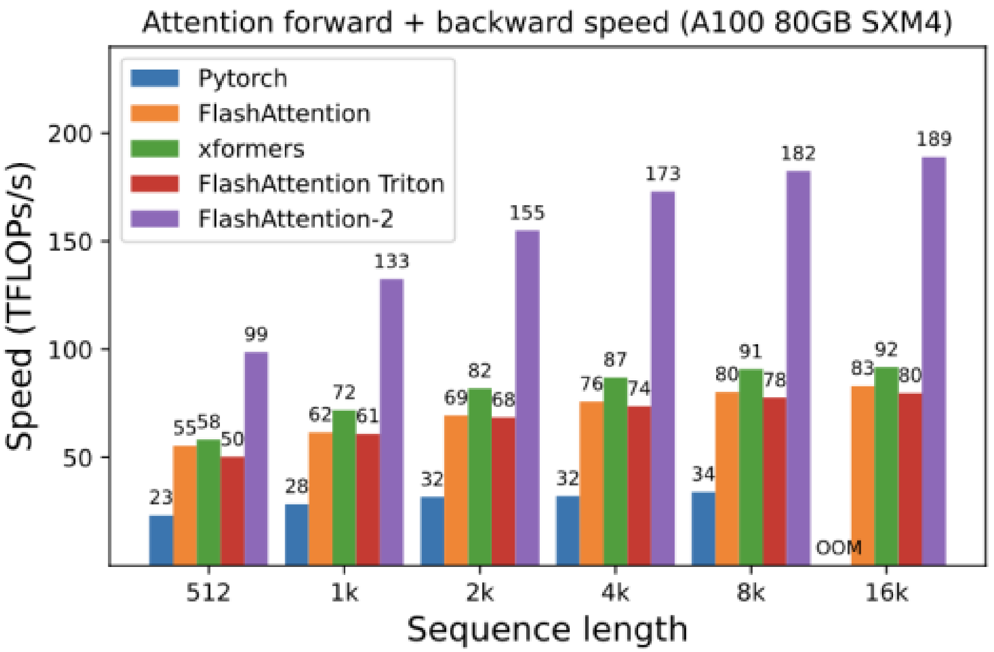
FlashAttention Results

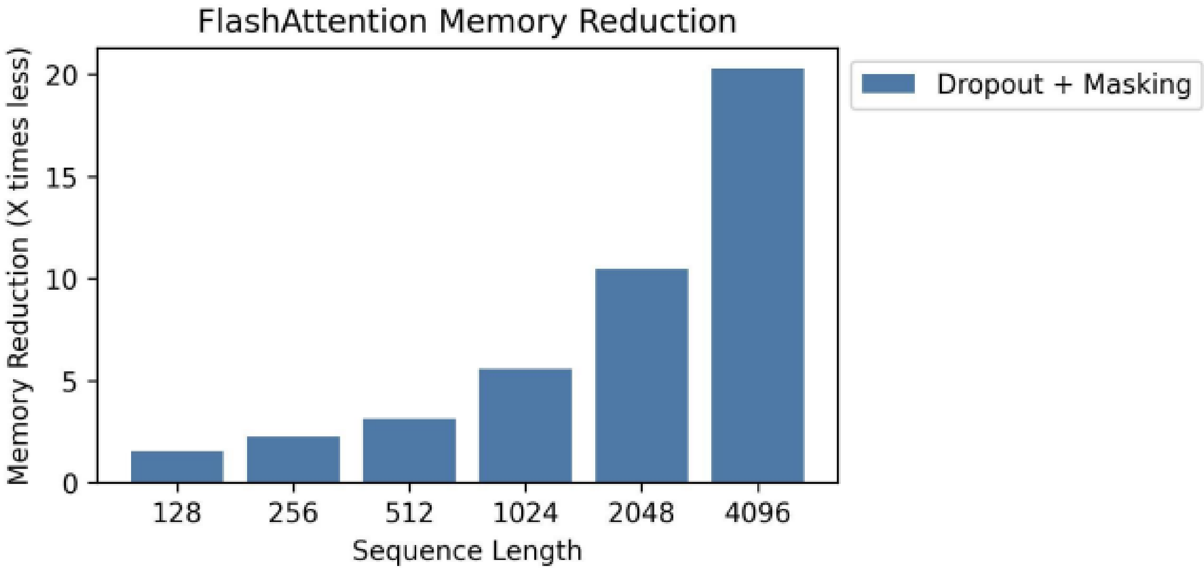
The algorithm is performing exact attention, no reduction in perplexity or quality of the model



Question: What would happen if we further increase the block size?

FlashAttention: 2-4x speedup, 10-20x memory reduction





Memory linear in sequence length

Is FlashAttention Stable?

Is Flash Attention Stable?

Alicia Golden[1,2] Samuel Hsia[1,2] Fei Sun[3] Bilge Acun[1] Basil Hosmer[1] Yejin Lee[1]
Zachary DeVito[1] Jeff Johnson[1] Gu-Yeon Wei[2] David Brooks[2] Carole-Jean Wu[1]

[1]FAIR at Meta [2]Harvard University [3]Meta

***Abstract*—Training large-scale machine learning models poses distinct system challenges, given both the size and complexity of today's workloads. Recently, many organizations training state-of-the-art Generative AI models have reported cases of instability during training, often taking the form of loss spikes. Numeric deviation has emerged as a potential cause of this training instability, although quantifying this is especially challenging given the costly nature of training runs. In this work, we develop a principled approach to understanding the effects of numeric deviation, and construct proxies to put observations into context when downstream effects are difficult to quantify. As a case study, we apply this framework to analyze the widely-adopted Flash Attention optimization. We find that Flash Attention sees roughly an order of magnitude more numeric deviation as compared to Baseline Attention at BF16 when measured during an isolated forward pass. We then use a data-driven analysis based on the Wasserstein Distance to provide upper bounds on how this numeric deviation impacts model weights during training, finding that the numerical deviation present in Flash Attention is 2-5 times less significant than low-precision training.**

***Index Terms*—Generative AI, Numeric Deviation, Training Instability, Attention, Transformers**

One under-explored potential cause of training instability is *numeric deviation*. Numeric deviation between an optimization and its corresponding baseline can lead to the gradual accumulation of errors, which over the course of training have the potential to culminate in loss spikes that require a resetting of the model state [1]. This is challenging to quantify, as training's stochastic nature suggests some level of numeric deviation might be acceptable, yet determining the threshold for when training becomes unstable proves difficult.

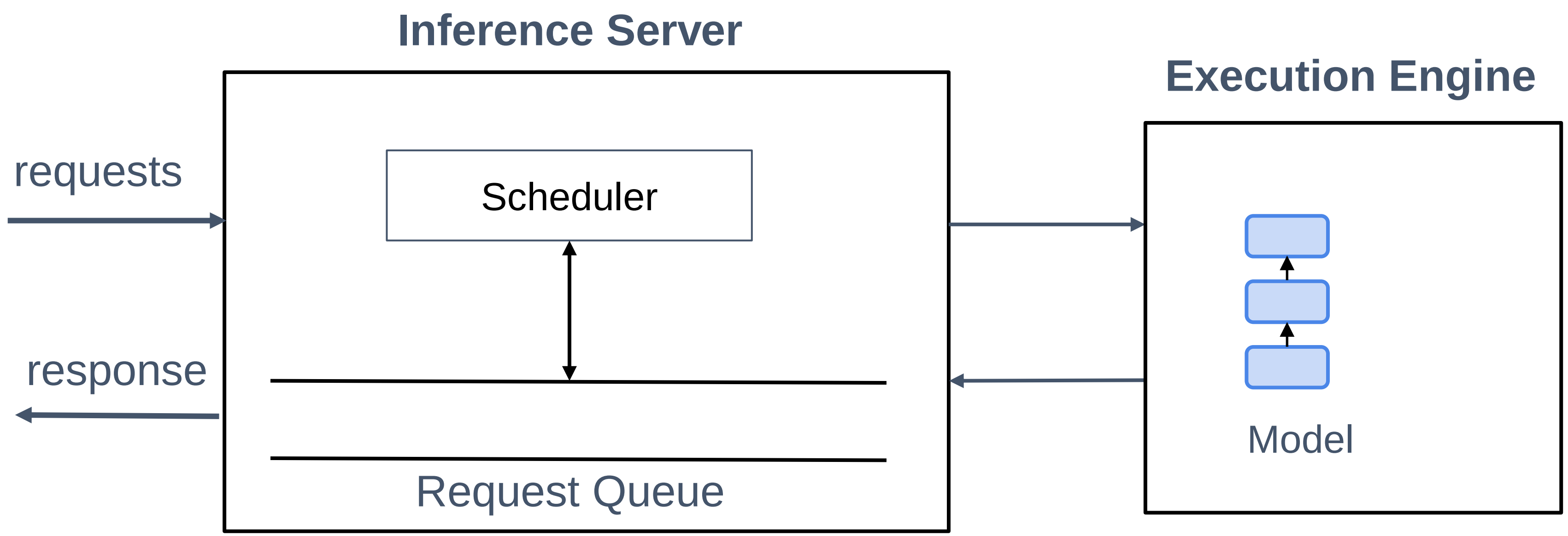
In this work, we develop a principled quantitative approach to understanding numeric deviation in training optimizations. Our approach consists of two phases, including (i) developing a microbenchmark to perturb numeric precision in the given optimization, and (ii) evaluating how numeric deviation translates to changes in model weights through a data-driven analysis based on Wasserstein distance. This ultimately allows us to provide an upper bound on the amount of numeric deviation for a given optimization, and helps to contextualize the improvement within known techniques. We aim to use

Today

DL Inference

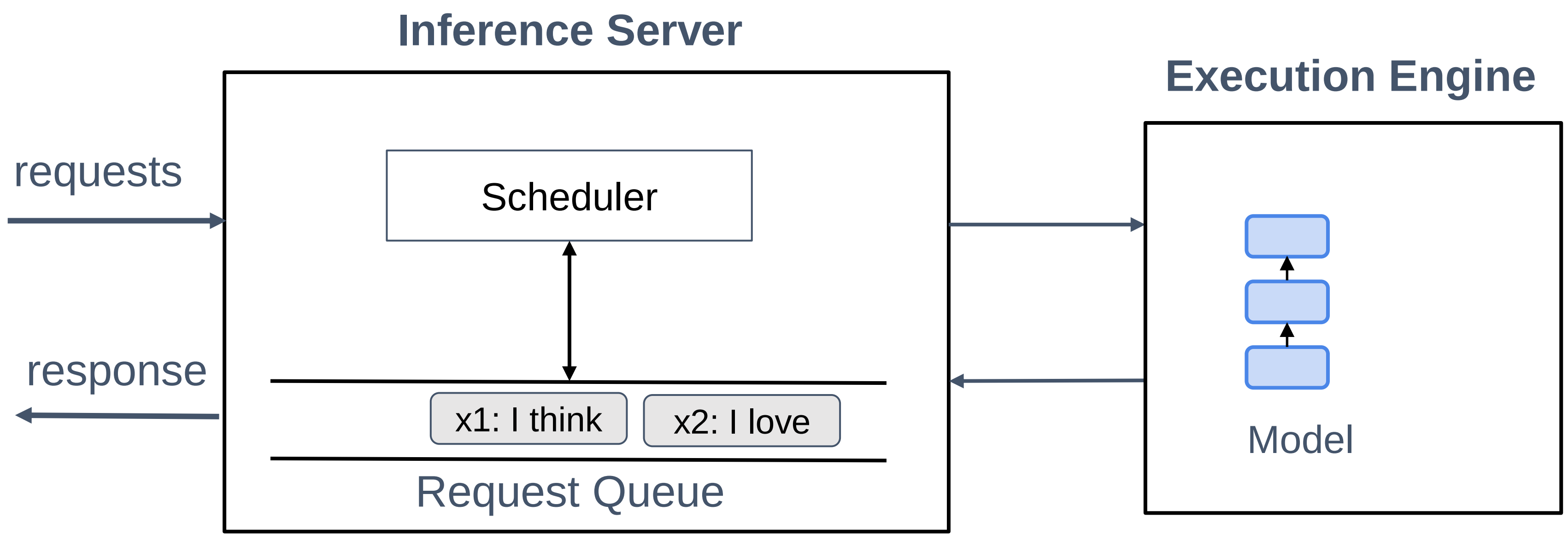
- LLM Inference
- FlashAttention
- Continuous Batching

LLM Serving System



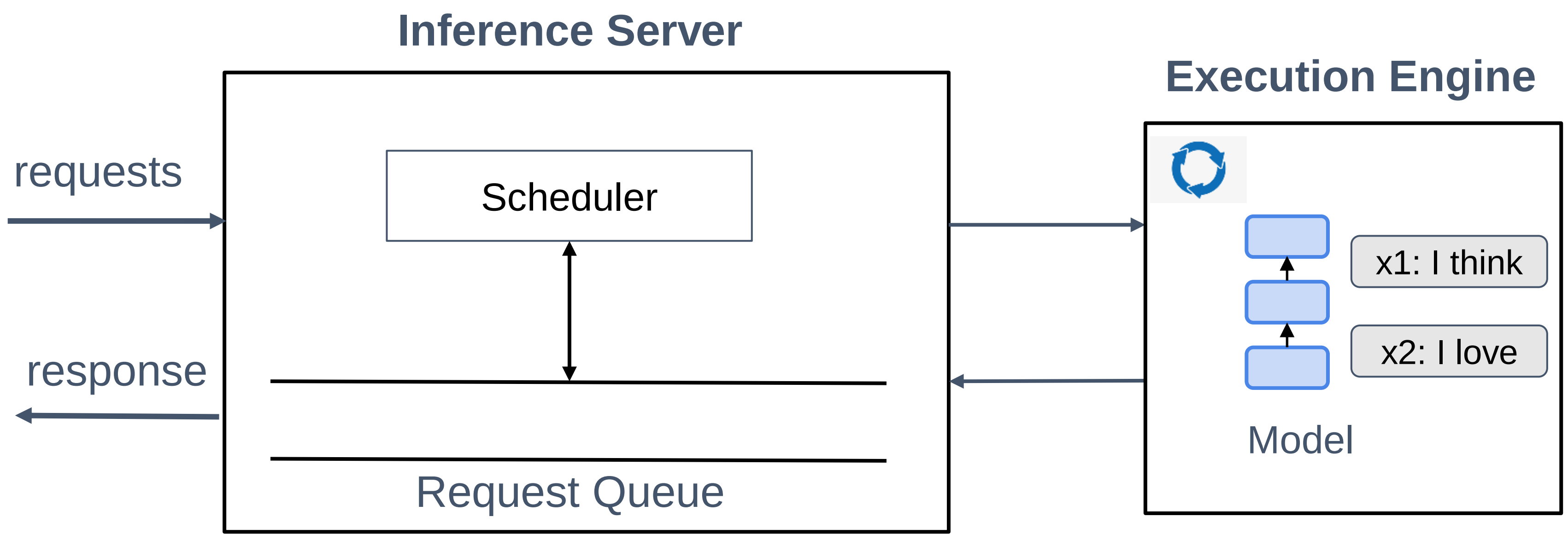
Maximum Batch Size = 3

LLM Serving System



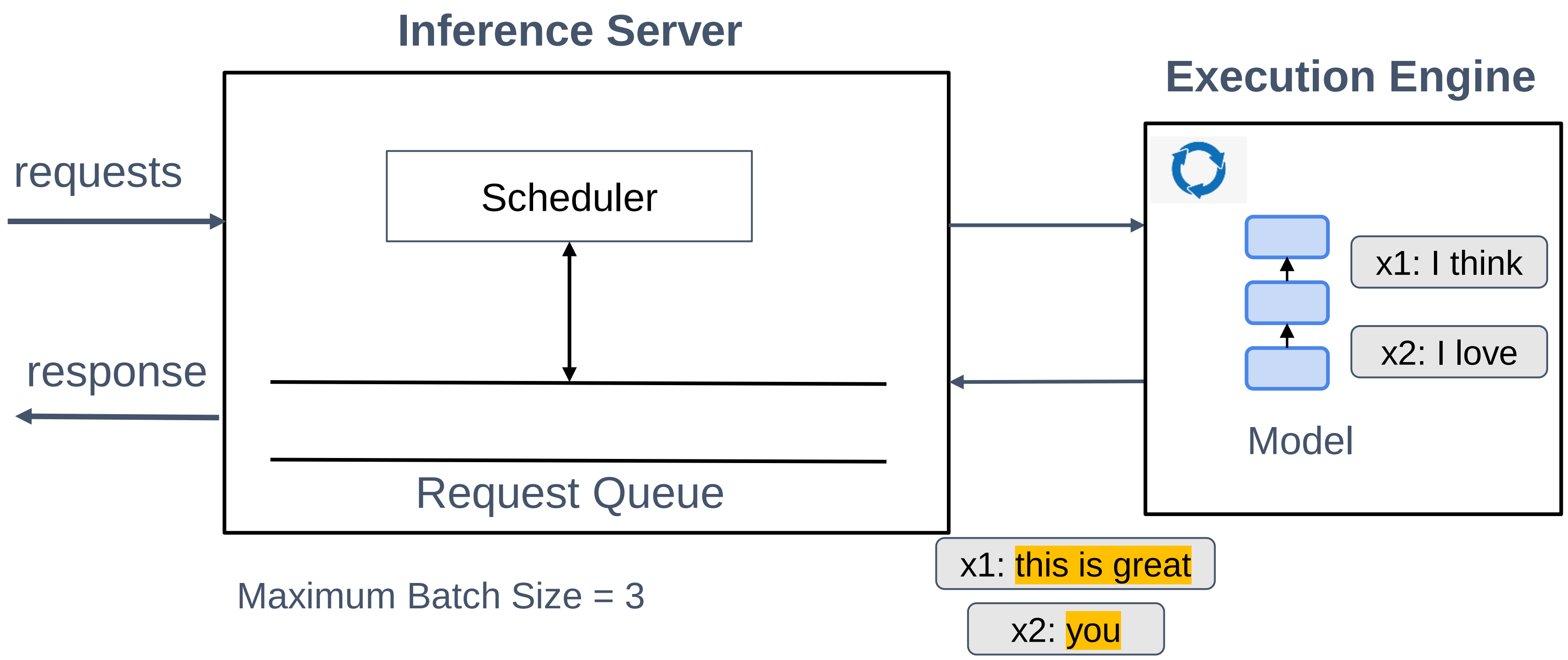
Maximum Batch Size = 3

LLM Serving System

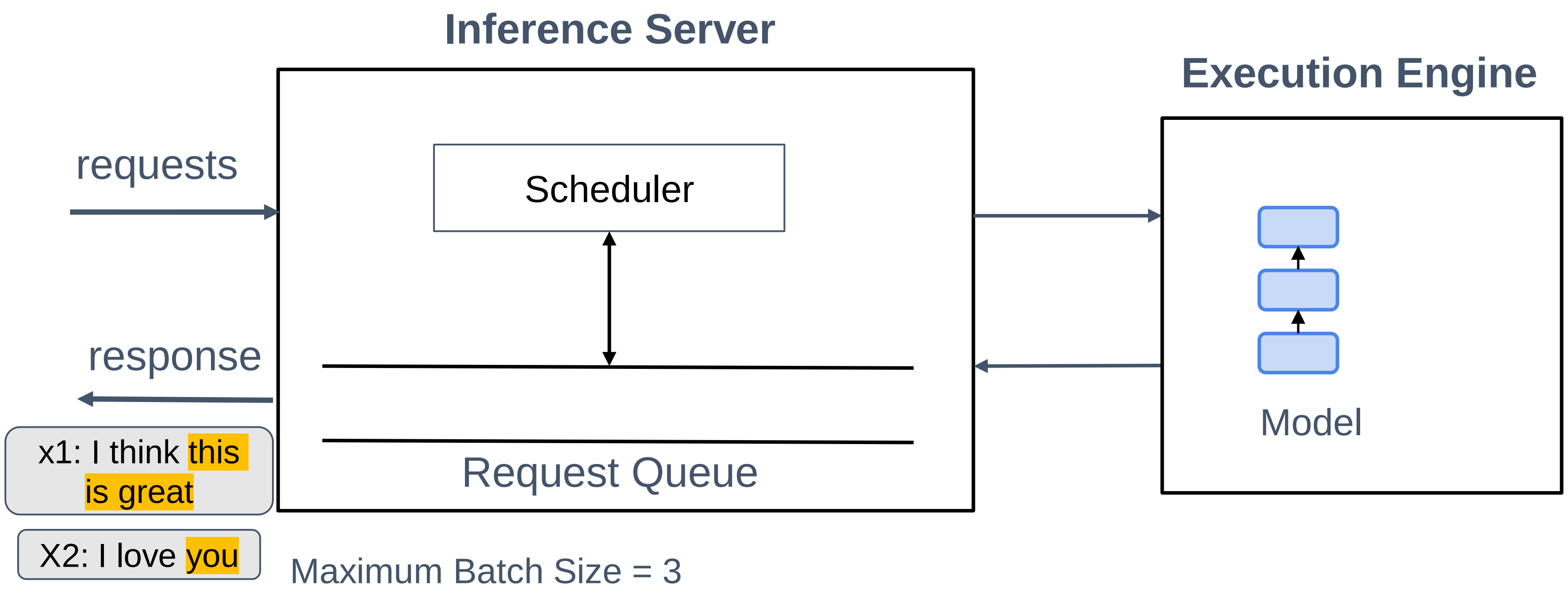


Maximum Batch Size = 3

LLM Serving System

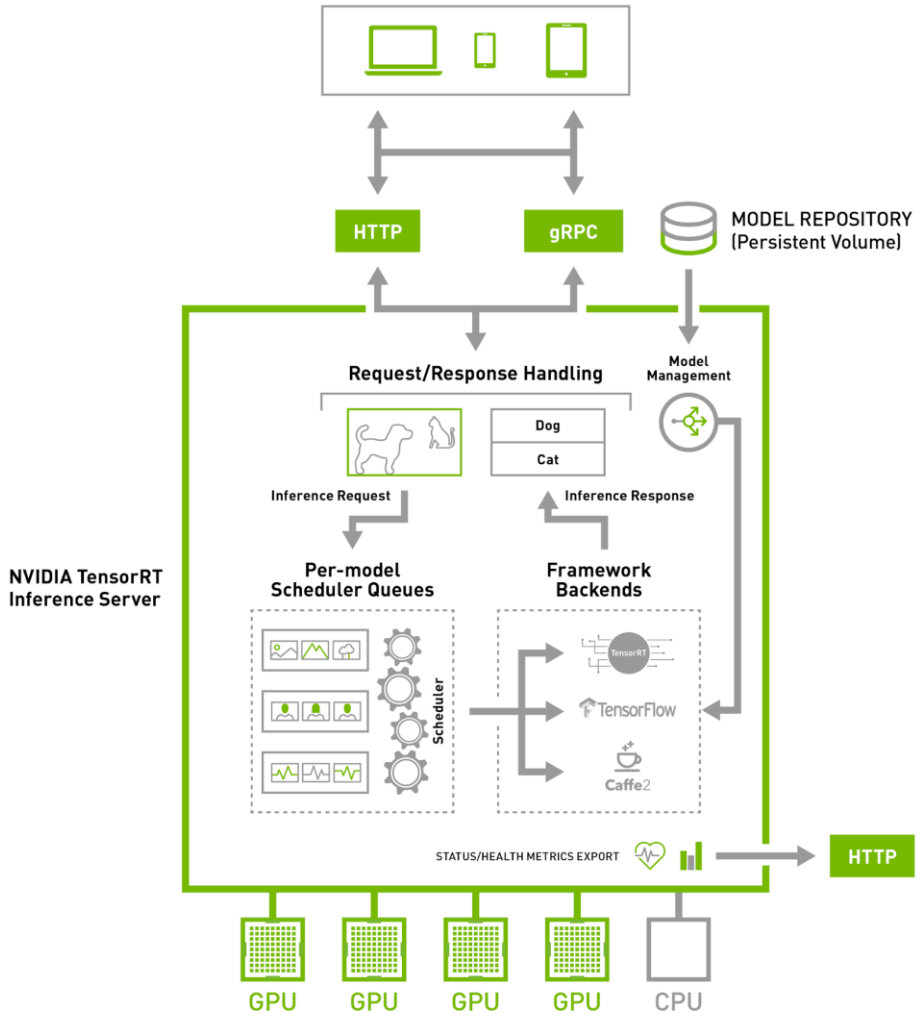


LLM Serving System



Example: TensorRT Inference Server

- Separates implementation of serving layer and execution layer
- Implements scheduling and batching algorithms
  - Dynamic Batching
  - Sequence Batching
  - Continuous Batching
- Allows multiple models to concurrently execute
- Supports multiple frameworks
  - vLLM backend
  - TensorFlow
  - PyTorch
  - ONNX



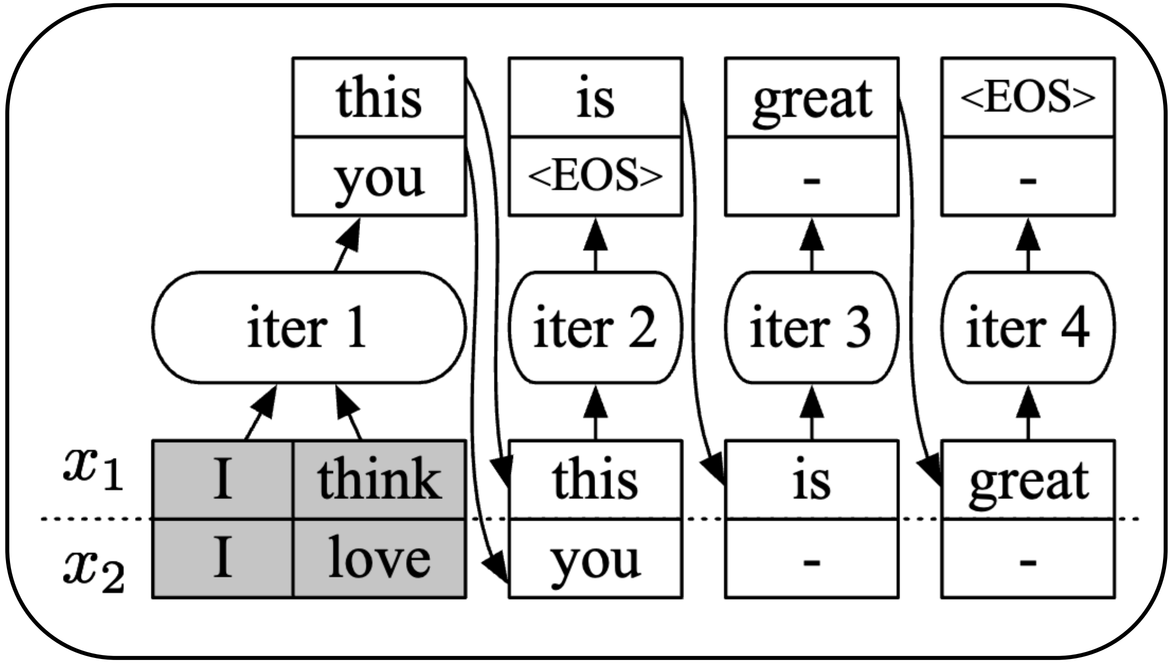
Today

DL Inference

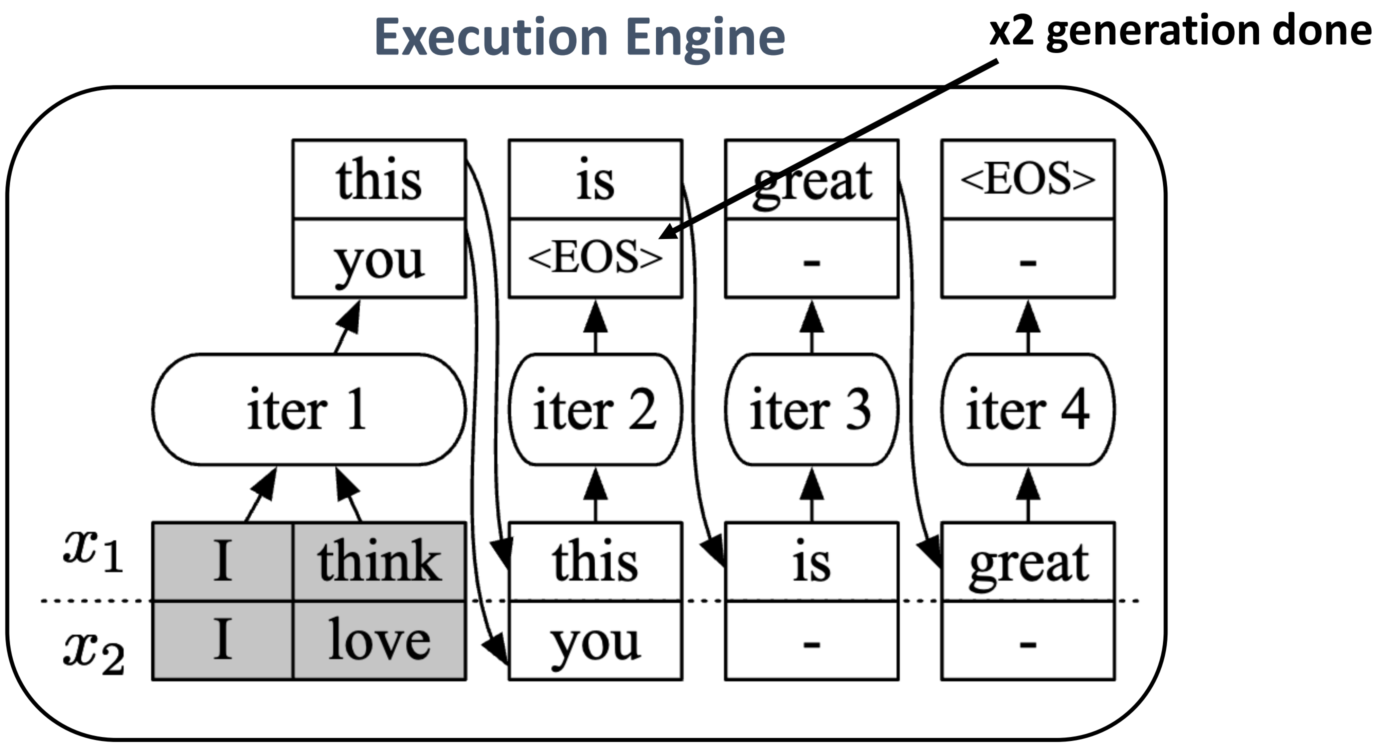
- FlashAttention
- LLM Inference
- Continuous Batching

Problem 1: Request Level Scheduling

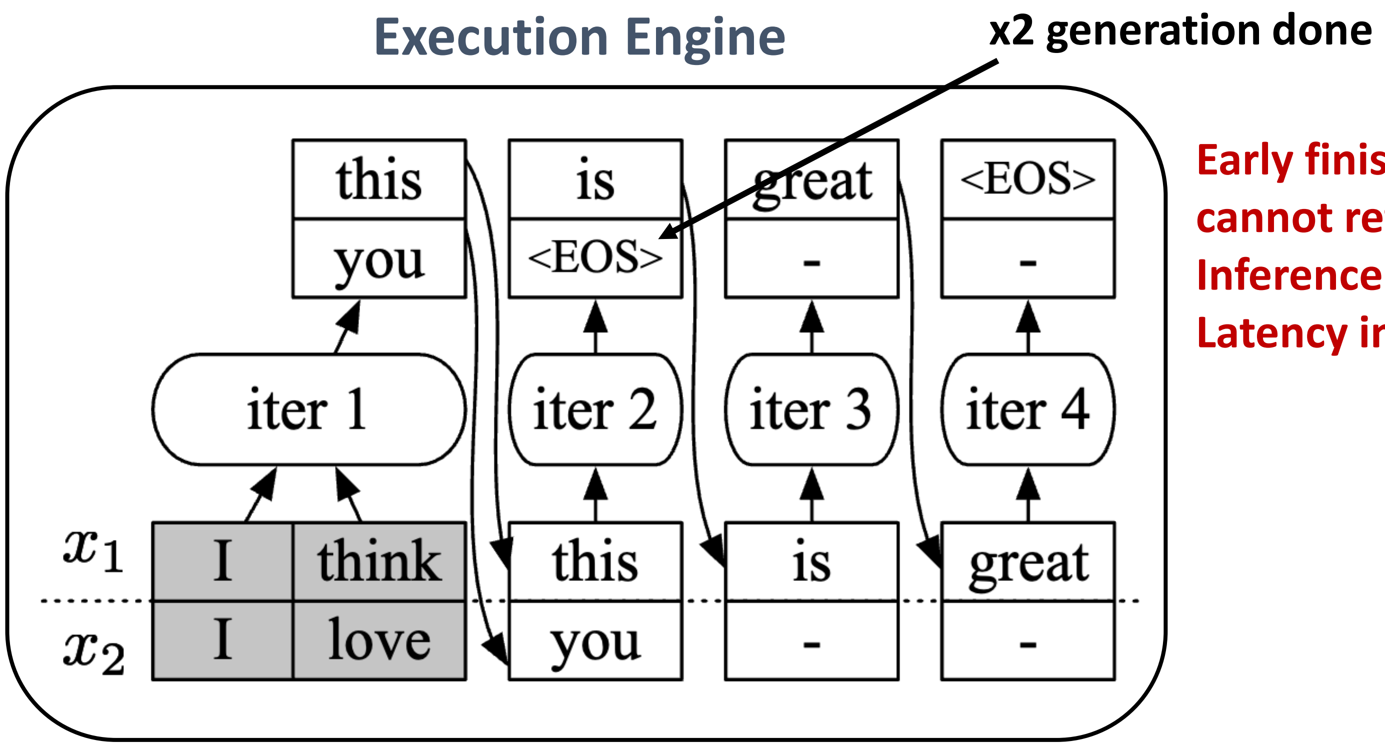
Execution Engine



Problem 1: Request Level Scheduling

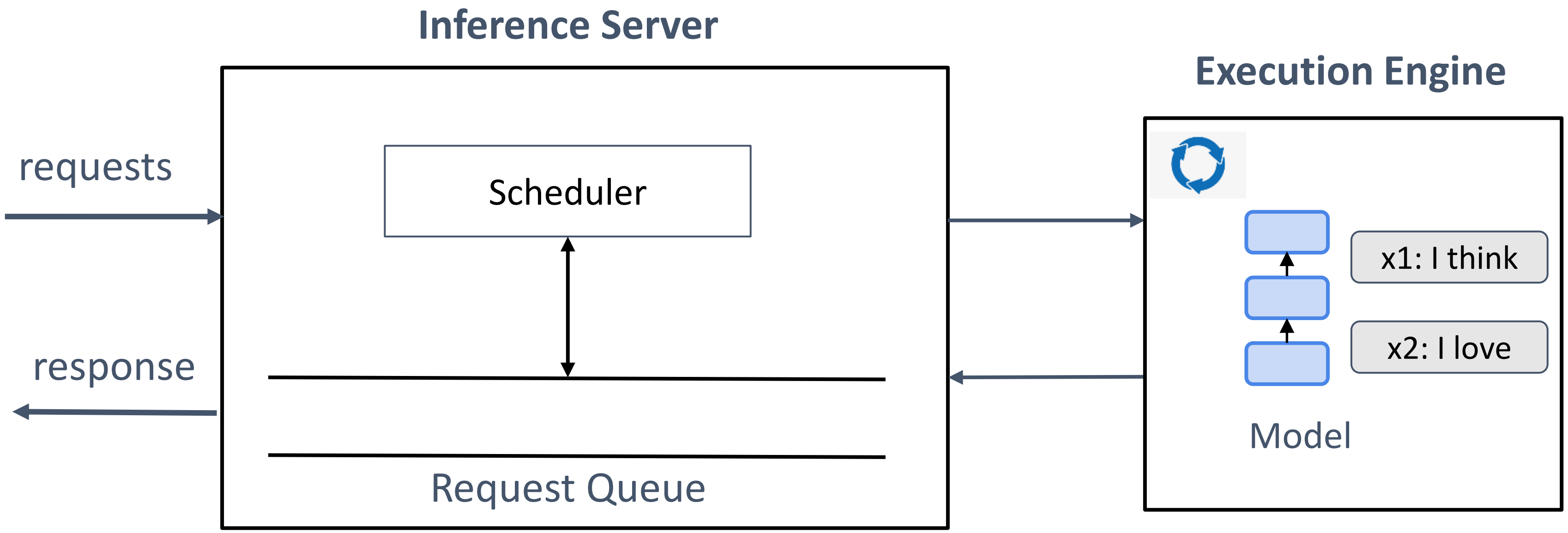


Problem 1: Request Level Scheduling



Early finished requests cannot return to the Inference Server → Latency increase

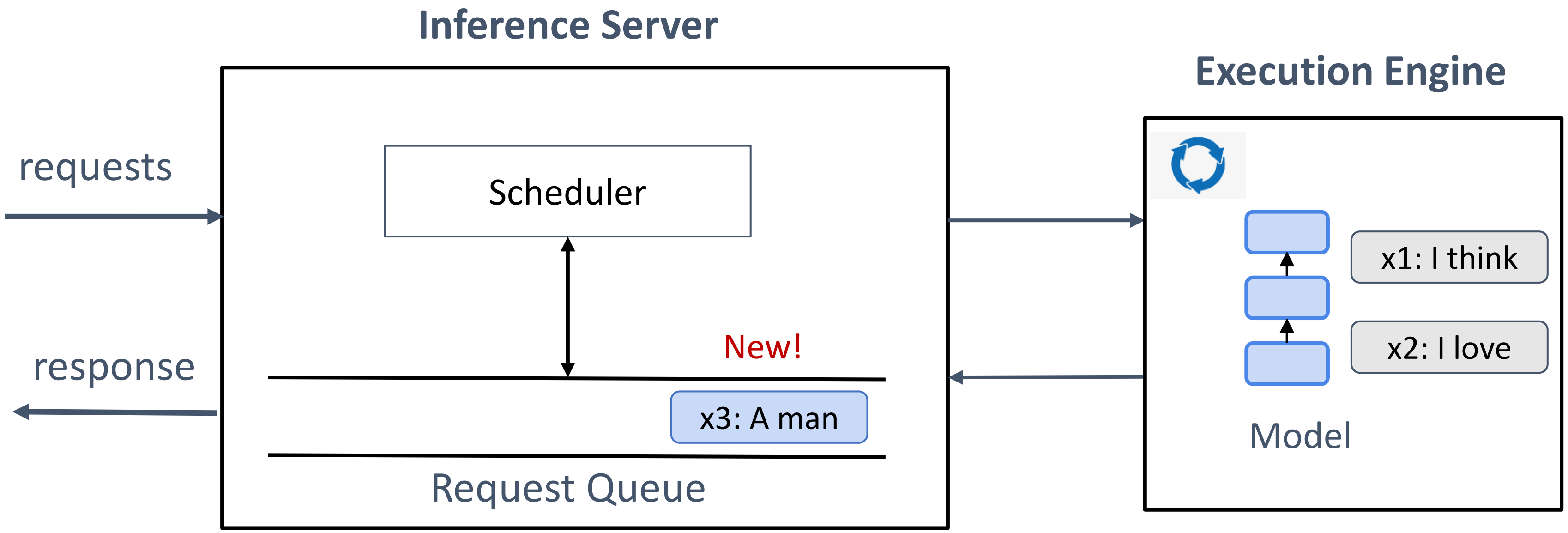
Problem 1: Request Level Scheduling



Maximum Batch Size = 3

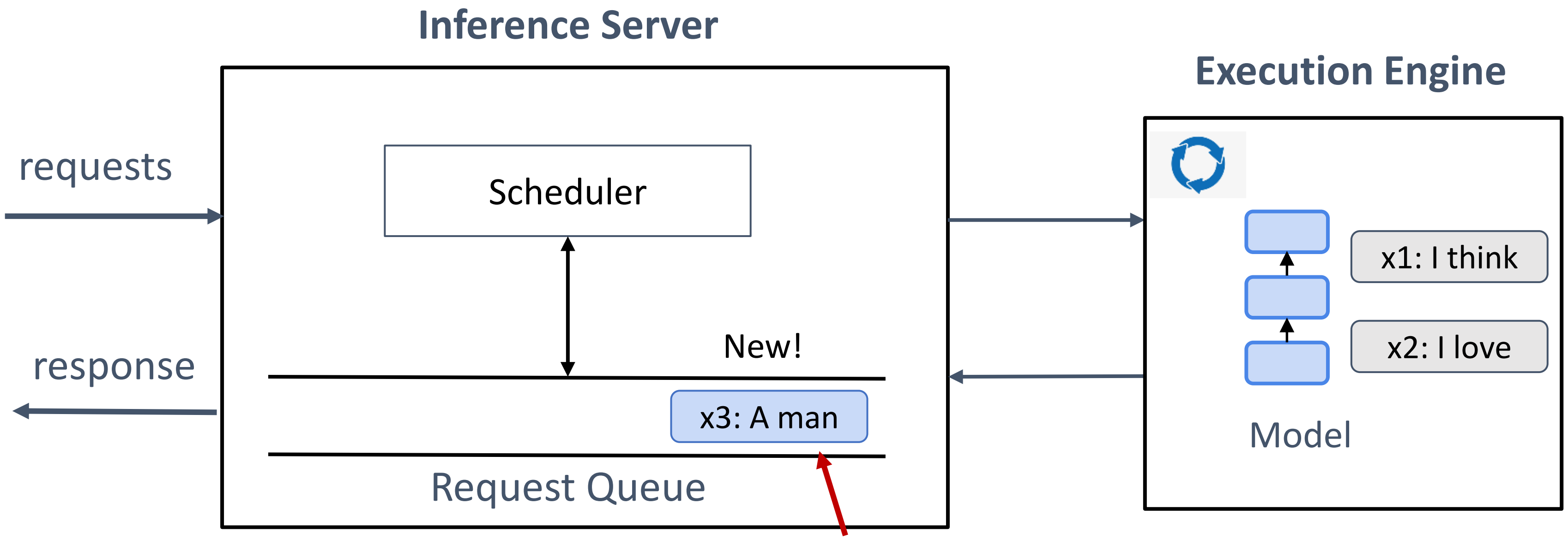
Started processing x1 and x2

Problem 1: Request Level Scheduling



Maximum Batch Size = 3

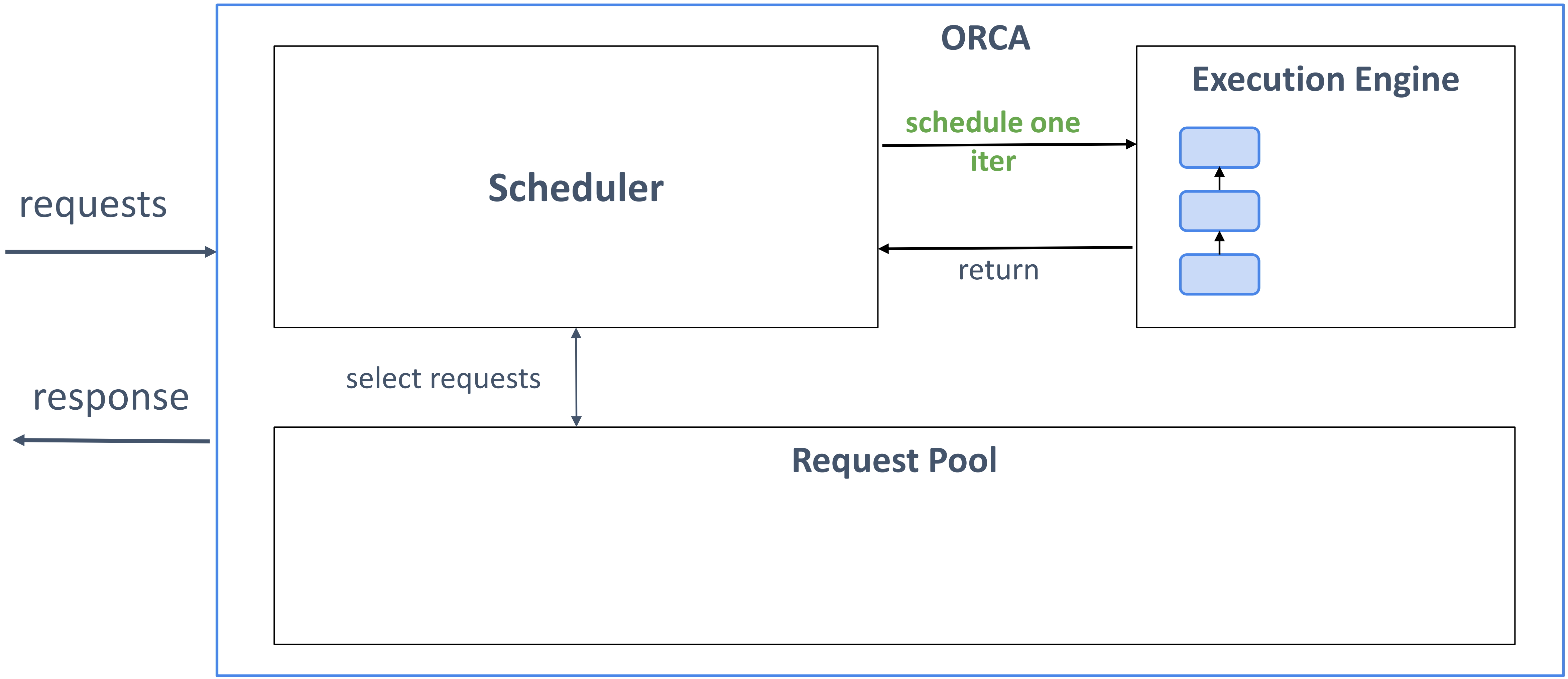
Problem 1: Request Level Scheduling



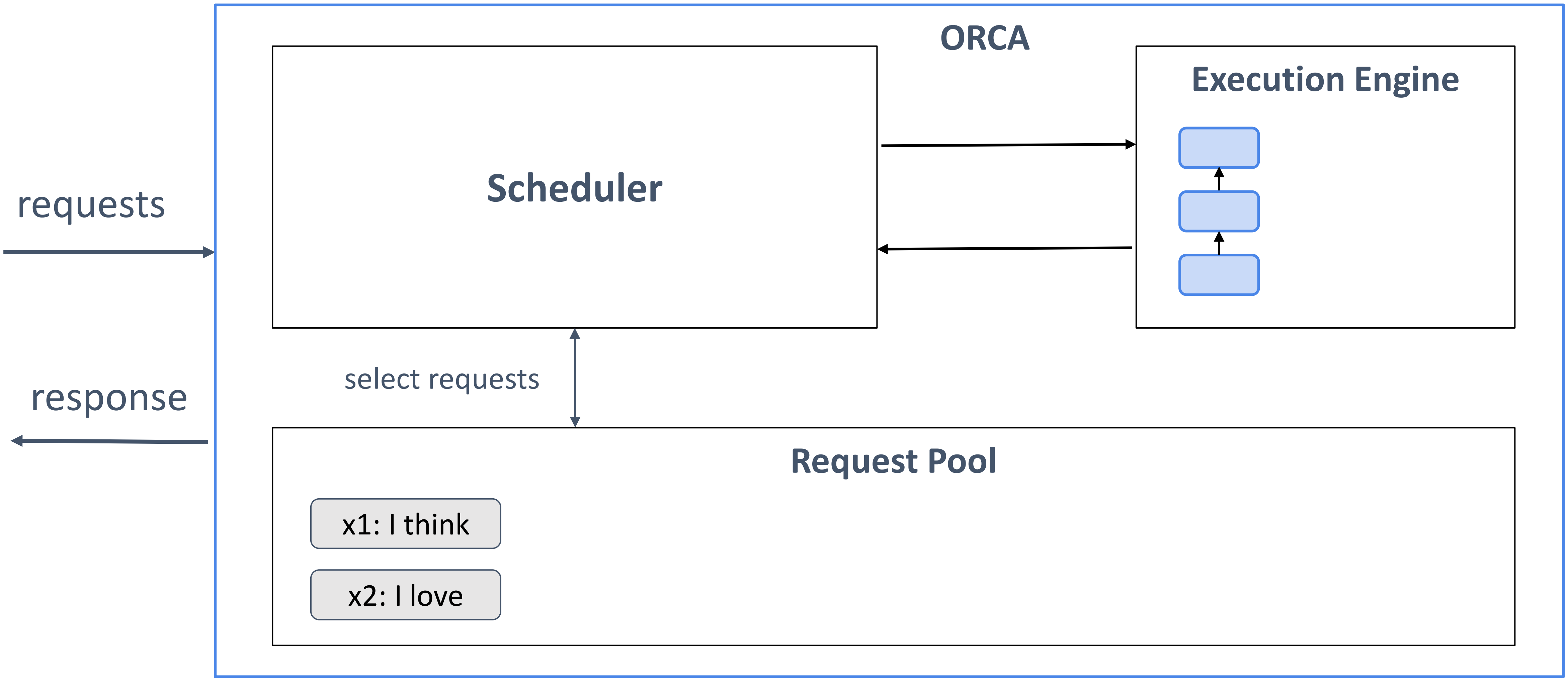
Maximum Batch Size = 3

Late join requests need to wait until engine finishes execution
→ Latency Increase

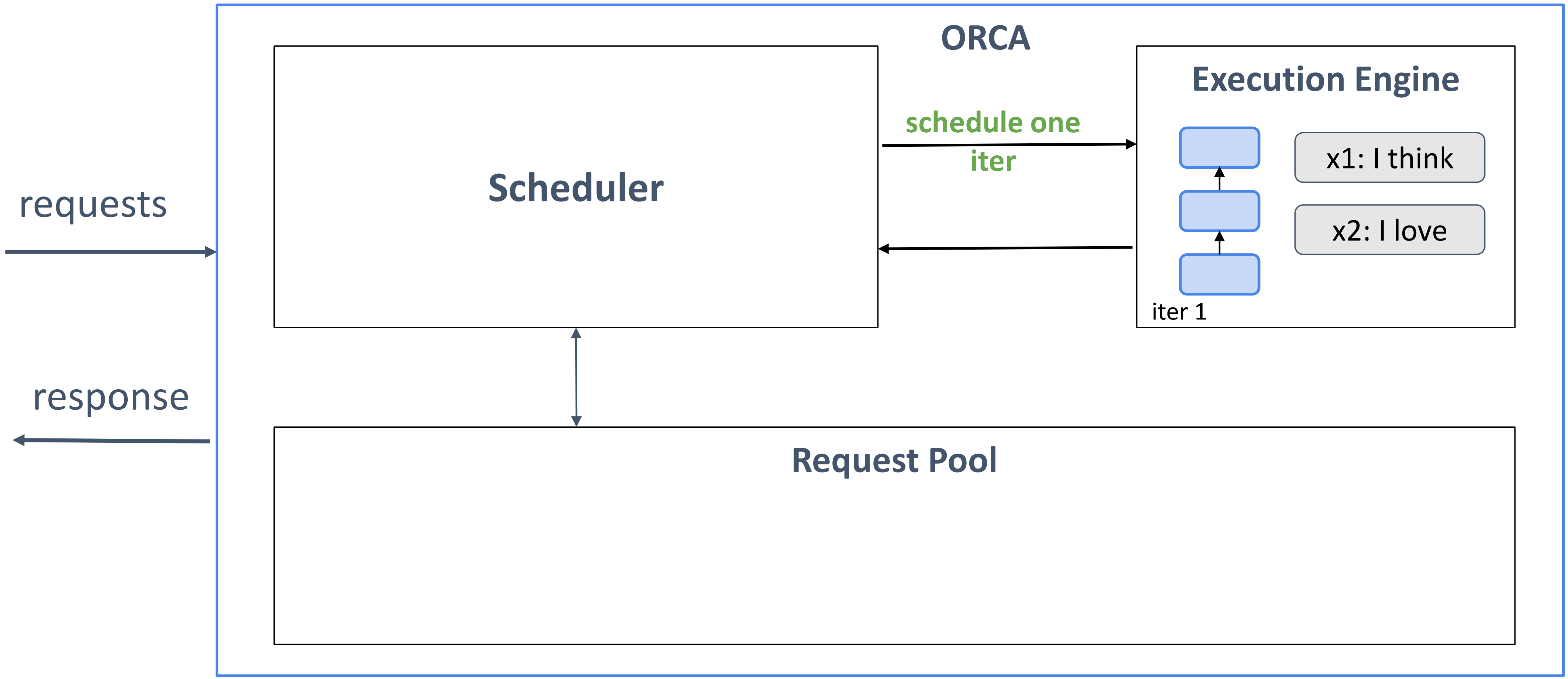
Solution 1: Iteration Level Scheduling



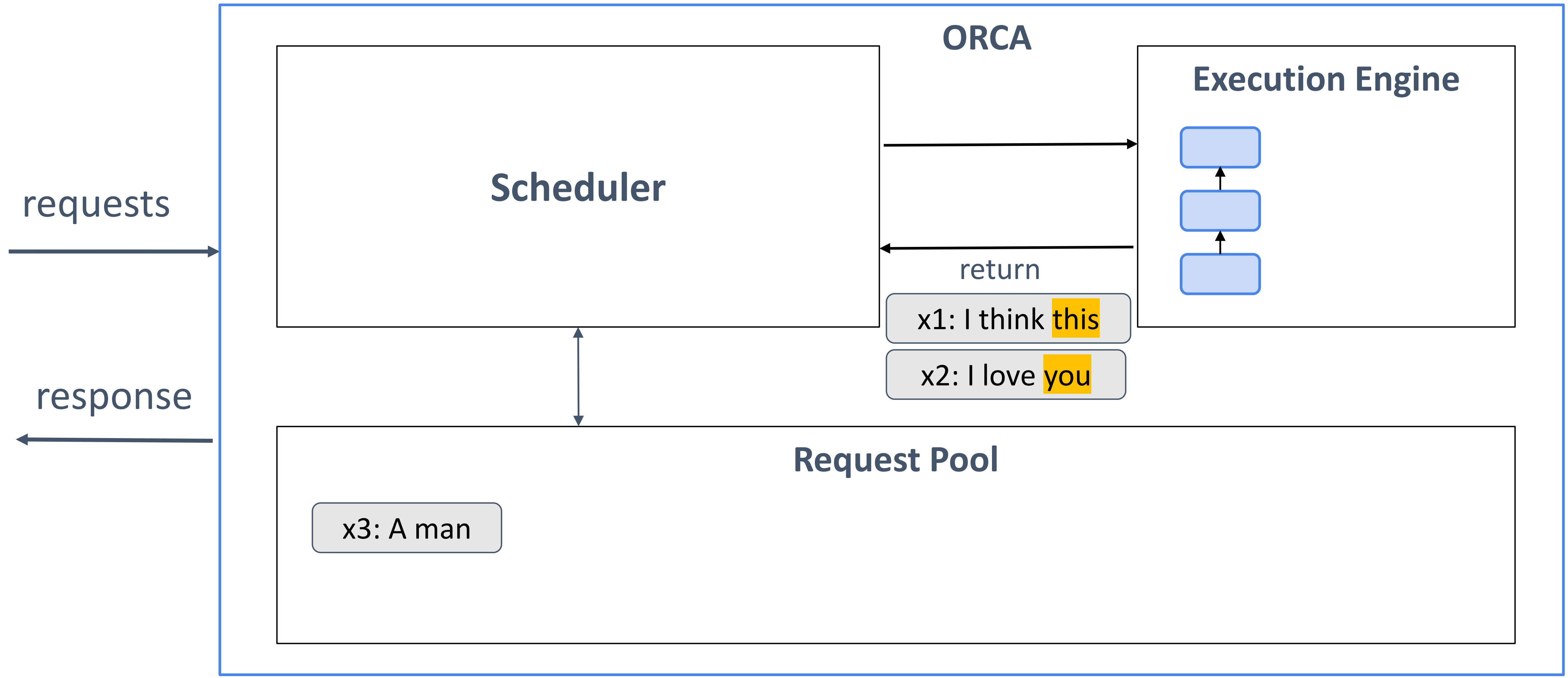
Solution 1: Iteration Level Scheduling



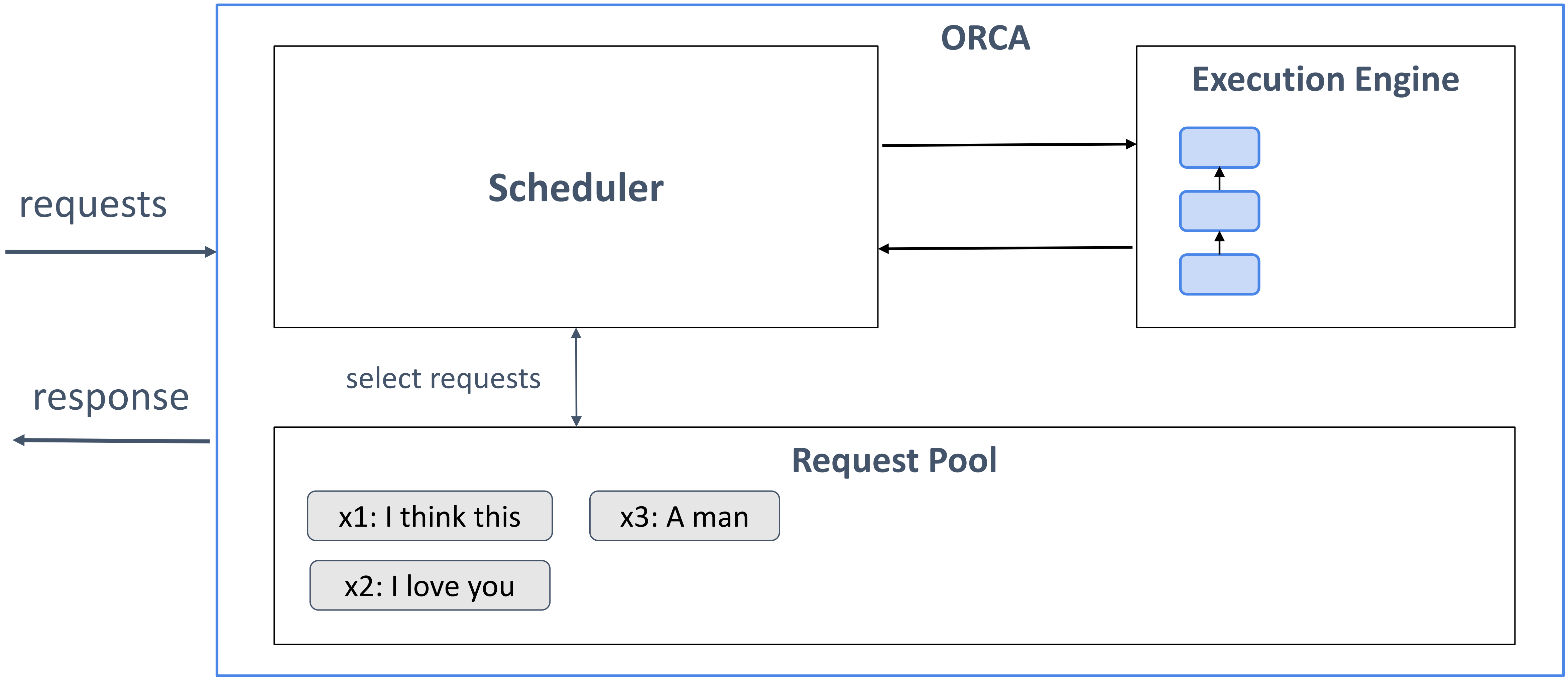
Solution 1: Iteration Level Scheduling



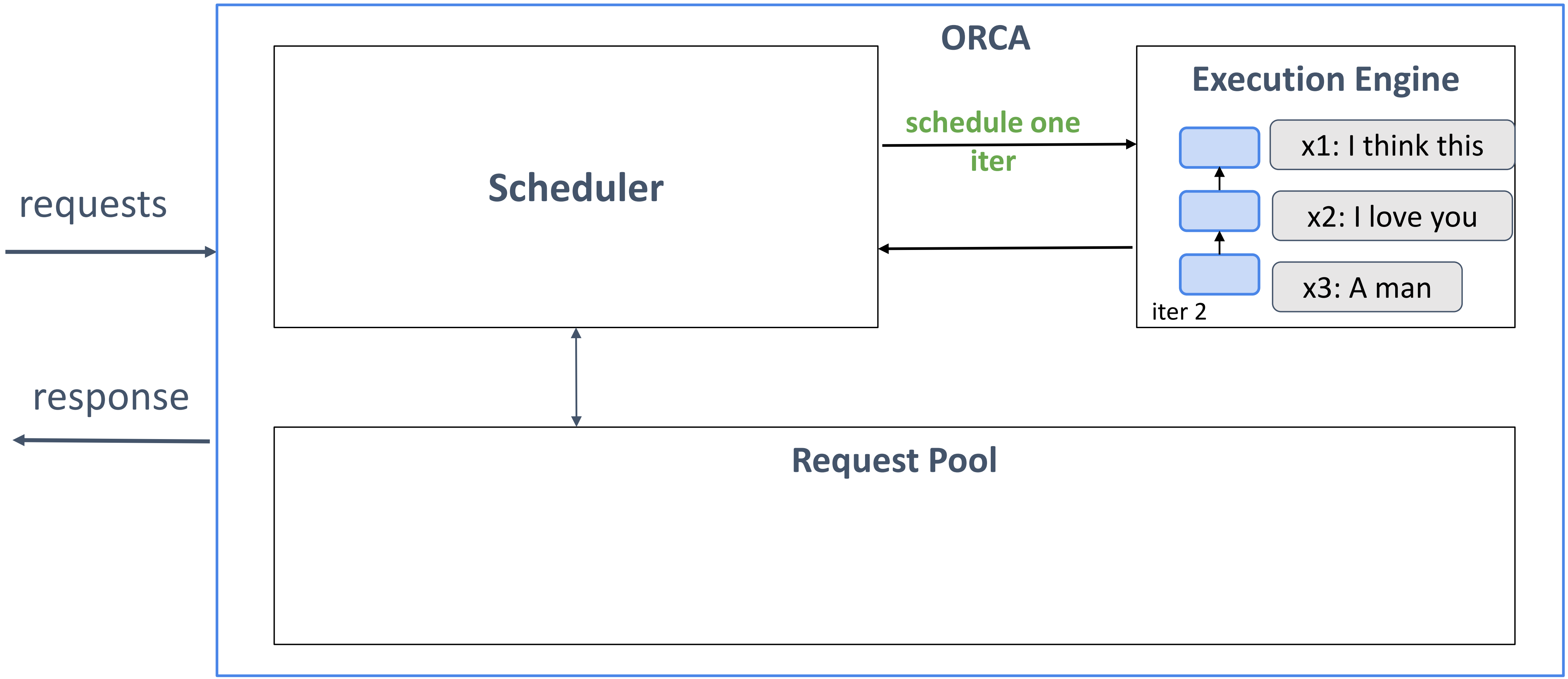
Solution 1: Iteration Level Scheduling



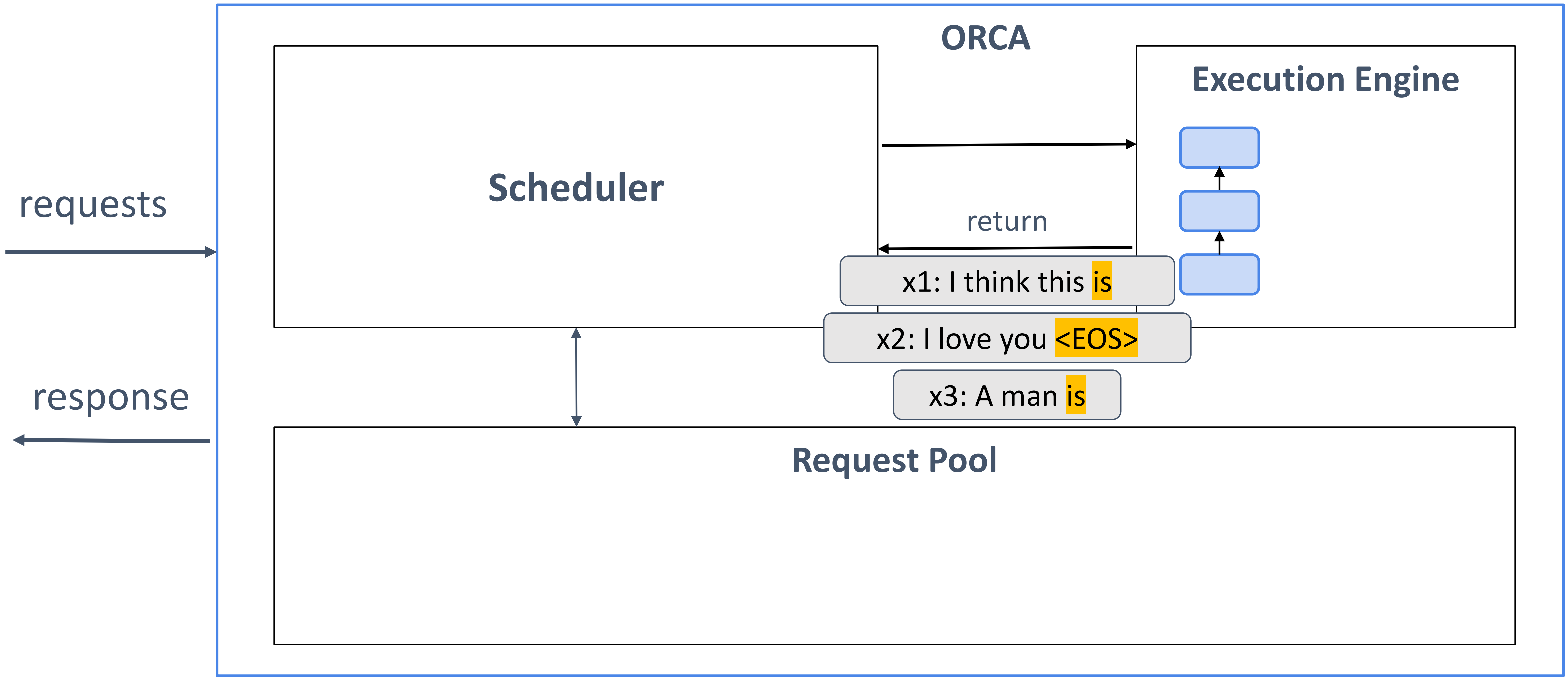
Solution 1: Iteration Level Scheduling



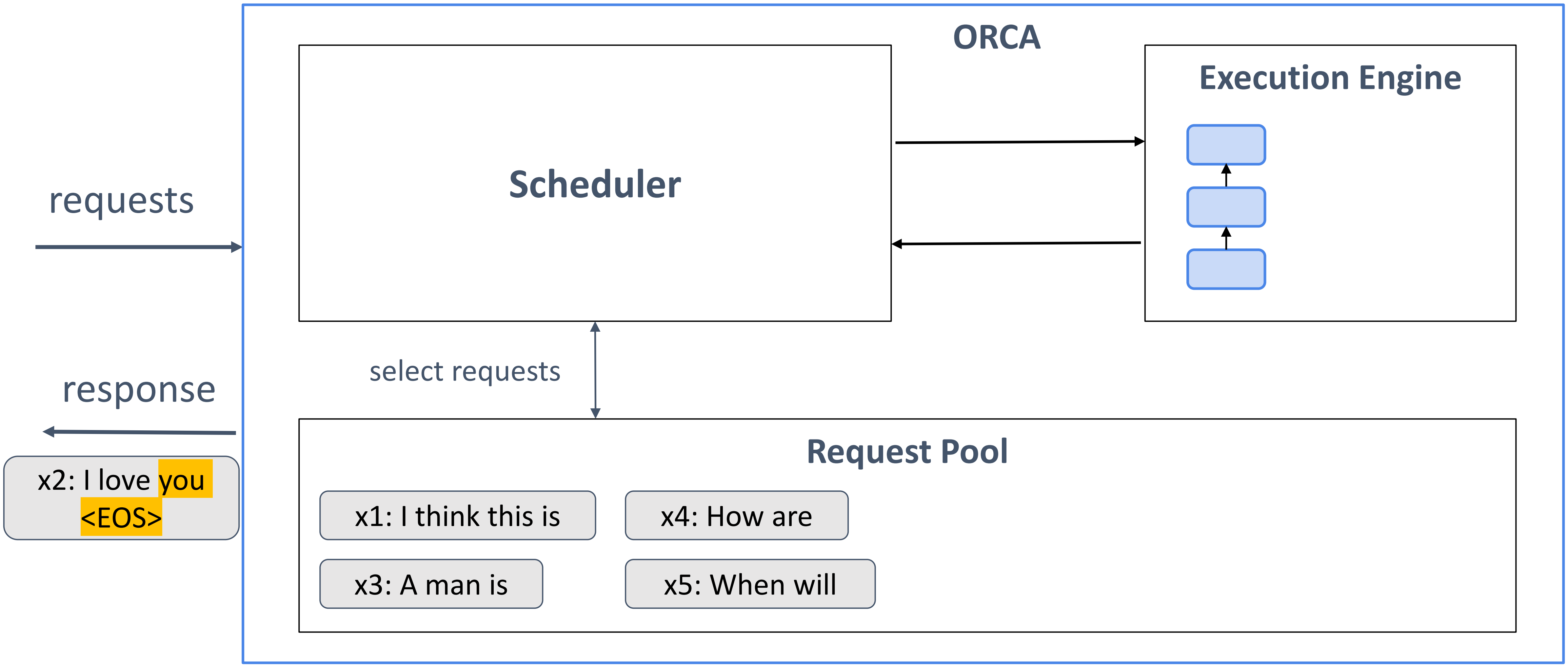
Solution 1: Iteration Level Scheduling



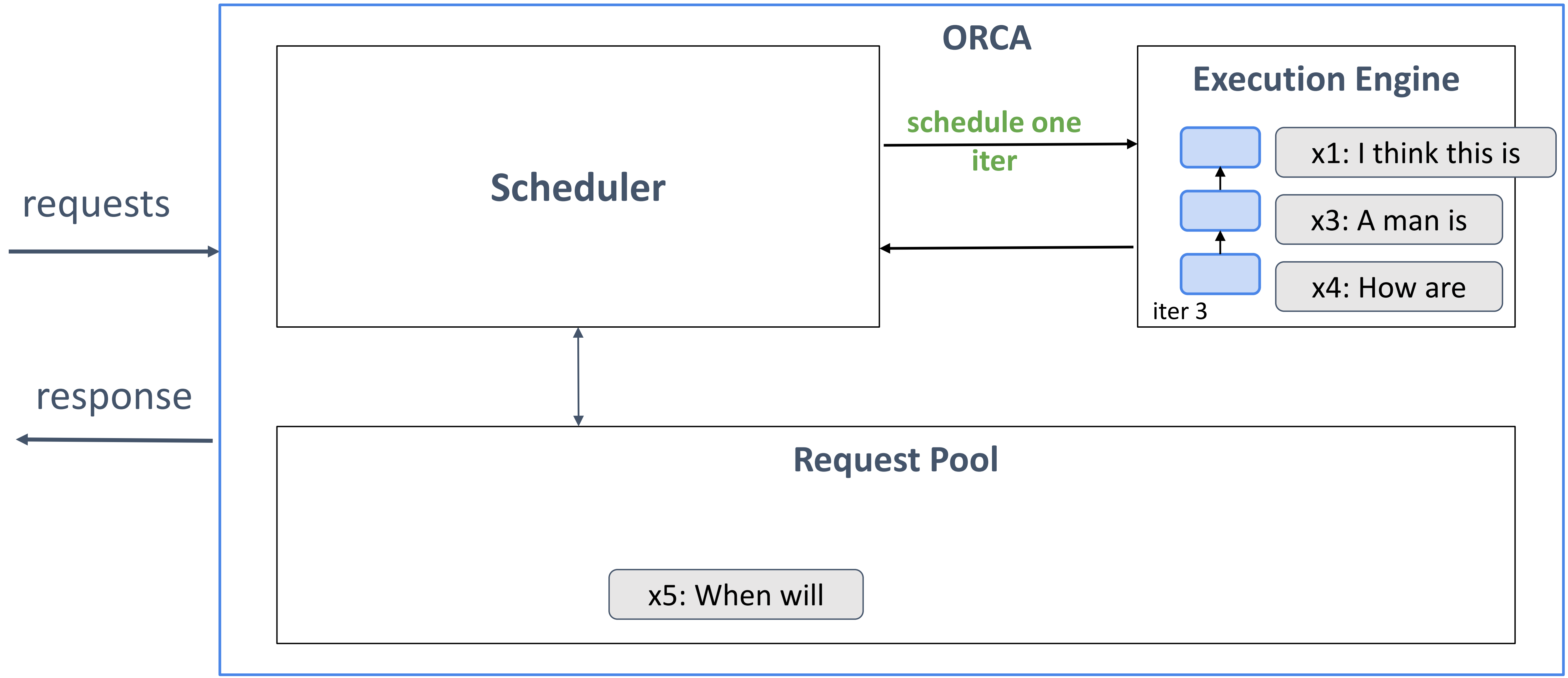
Solution 1: Iteration Level Scheduling



Solution 1: Iteration Level Scheduling



Solution 1: Iteration Level Scheduling



Solution 1: Iteration Level Scheduling

Iteration Level Scheduling handles early finished requests and late joining requests

Problem 2: How to Batch Requests?

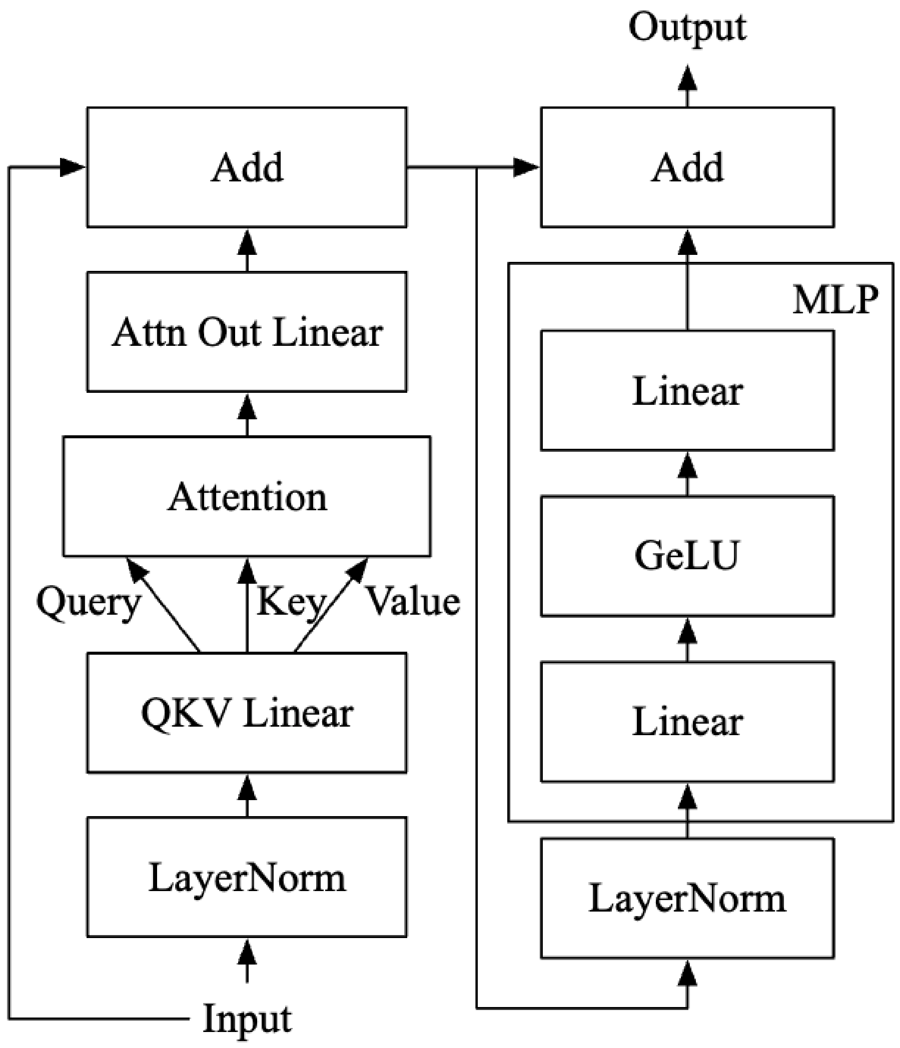
Let's assume Batch Size B = 1

Input Dimension: [L x H] (L=sequence length, H=hidden dim.)

Attention Operation:

1. **QK^T** : [LxH] x [HxL] → [L x L]
2. **P = softmax(QK^T)** : [L x L]
3. **O = PV** : [LxL] x [LxH] → [L x H]

With Batch Size B, **QK^T** will be **[B x L x L]**



Problem 2: How to Batch Requests?

Let's assume Batch Size B = 1

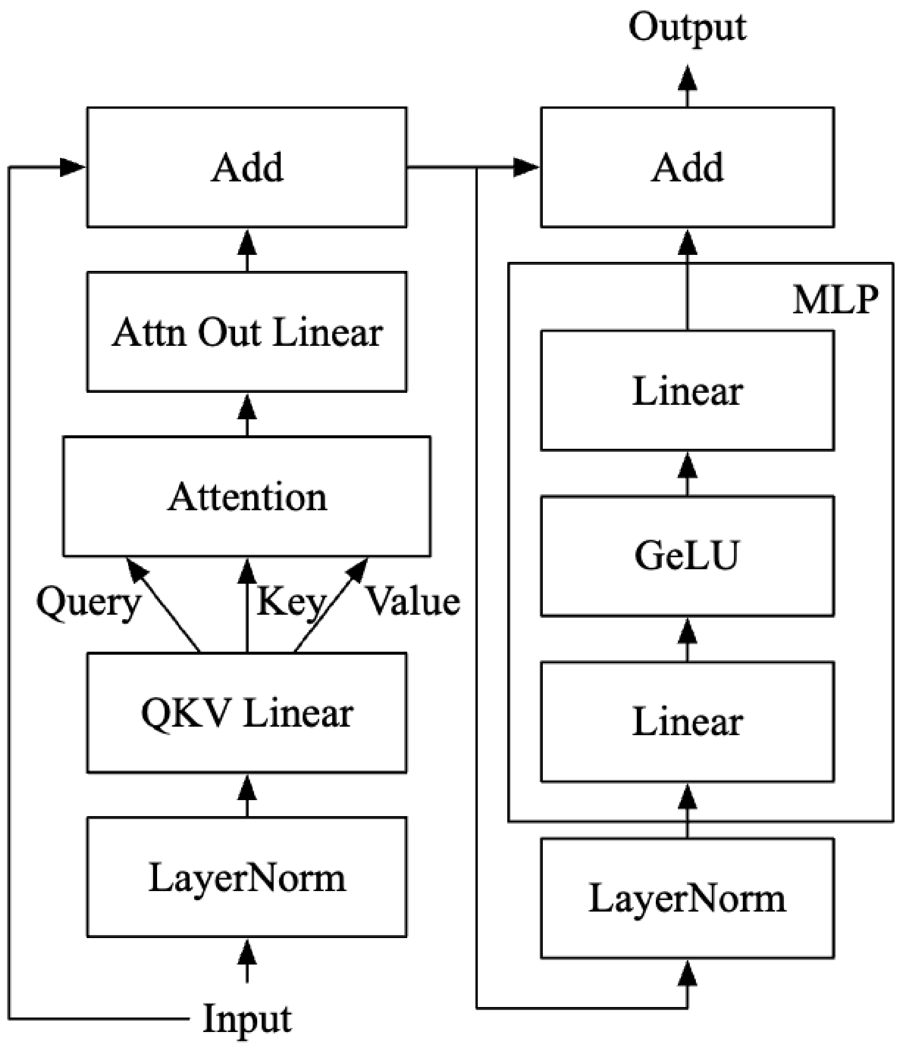
Input Dimension: [L x H] (L=sequence length, H=hidden dim.)

Attention Operation:

1. **QK^T** : [LxH] x [HxL] → [L x L]
2. **P = softmax(QK^T)** : [L x L]
3. **O = PV** : [LxL] x [LxH] → [L x H]

With Batch Size B, **QK^T** will be **[B x L x L]**

With different sequence lengths, QK^T cannot be computed



Solution 2: Selective Batching

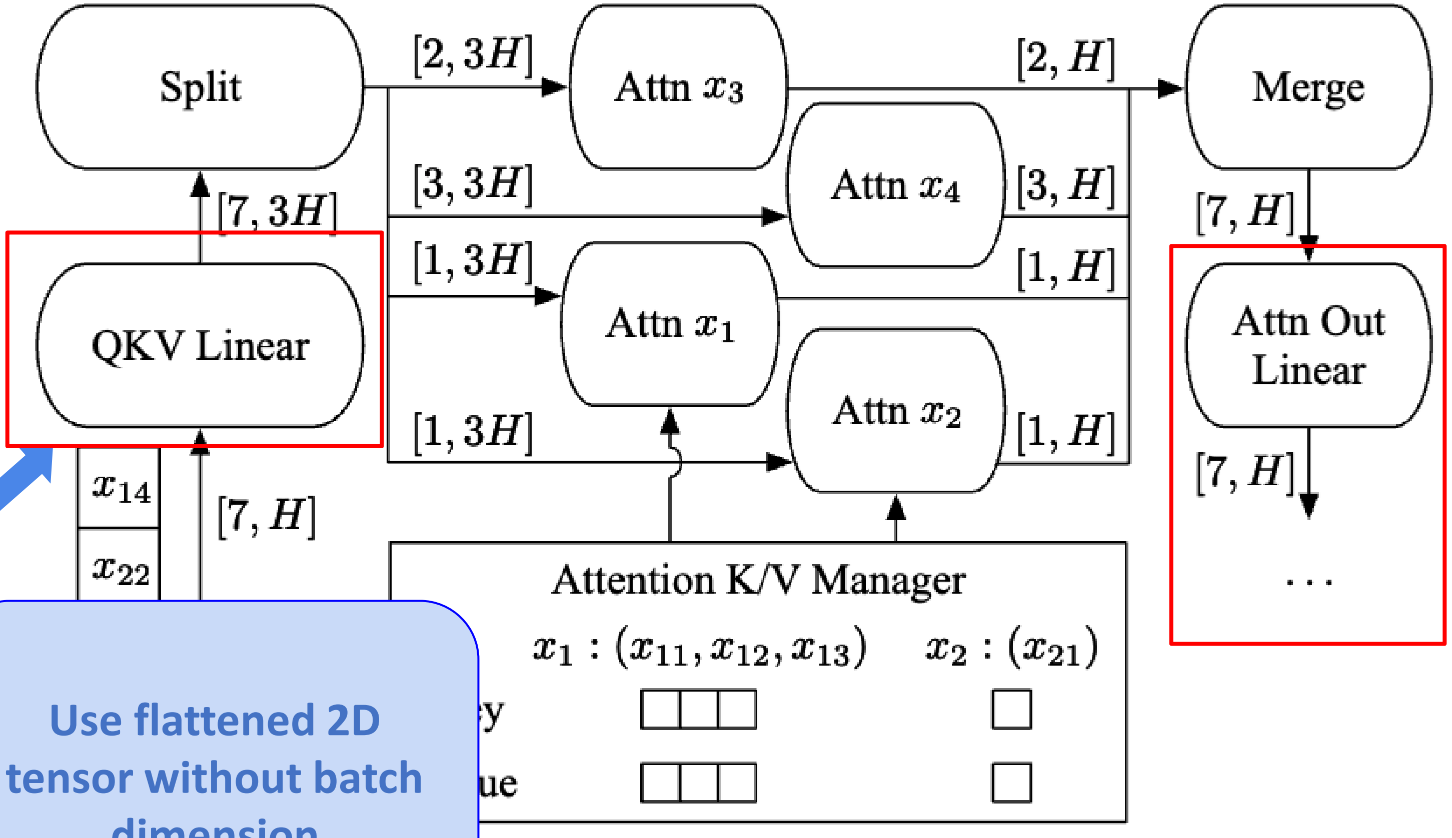
Only **Attention operation** does not work with batching tensors with diff. L_i

Batch for other ops. (Layer Norm, GeLU, etc.)

Coalesce $[L_i, H]$ tensor to $[\Sigma L_i, H]$ for batching

x1: [1,H]
x2: [1,H]
x3: [2,H]
x4: [3,H]

→ **[7,H] tensor**



Use flattened 2D tensor without batch dimension

Solution 2: Selective Batching

Only **Attention operation** does not work with batching tensors with diff. L_i

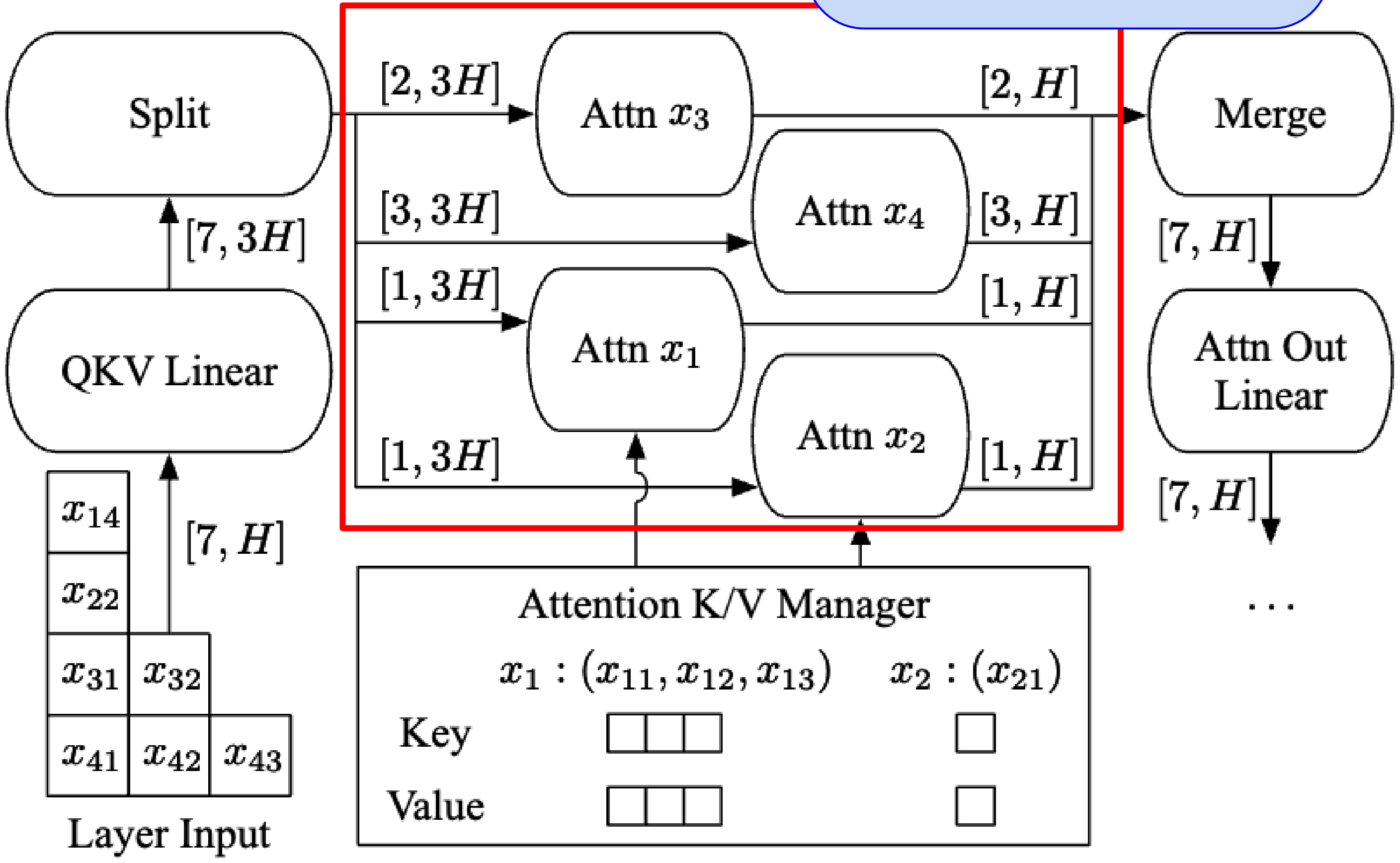
Batch for other ops. (Layer Norm, GeLU, etc.)

Coalesce $[L_i, H]$ tensor to $[\Sigma L_i, H]$ for batching

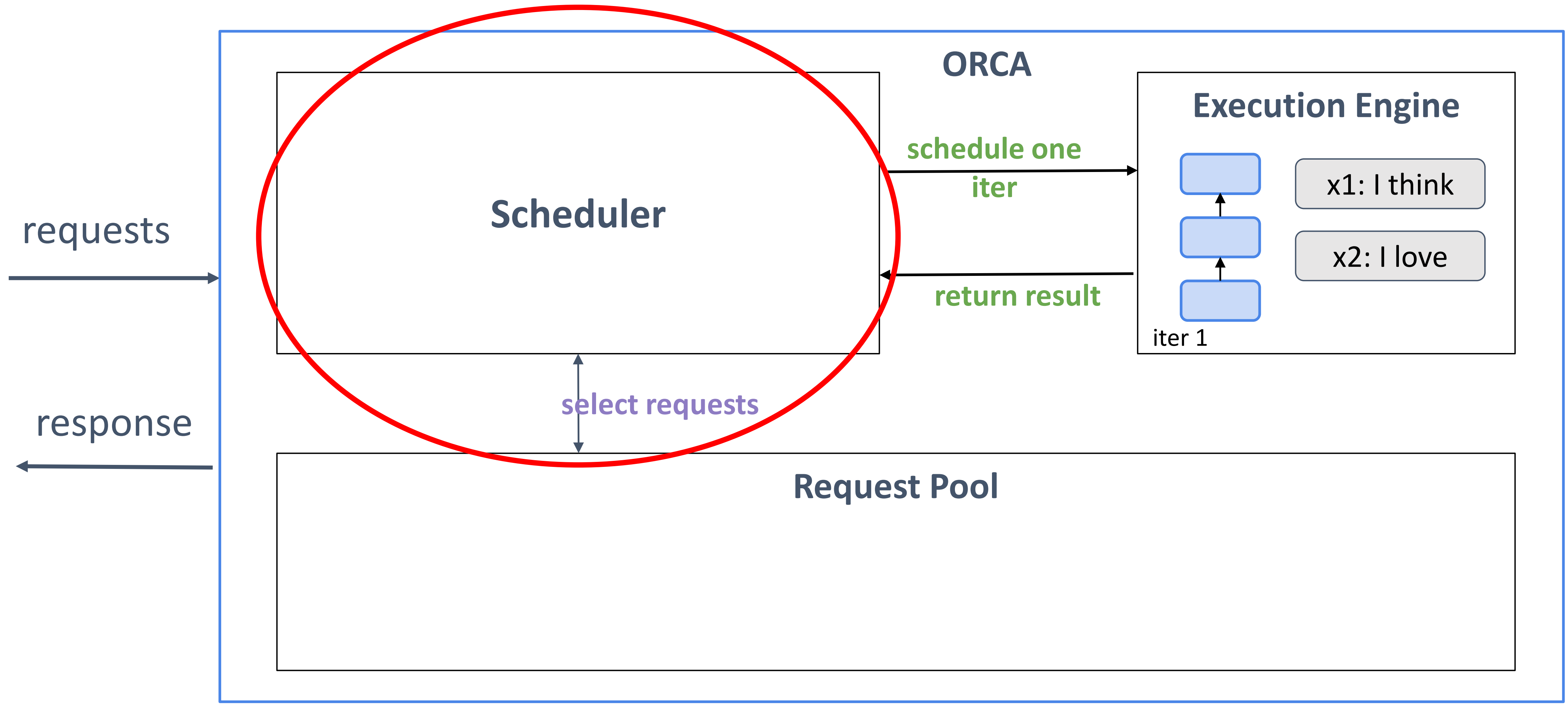
x1: [1,H]
x2: [1,H]
x3: [2,H]
x4: [3,H]

→ **[7,H] tensor**

Split, process each request and merge tensors



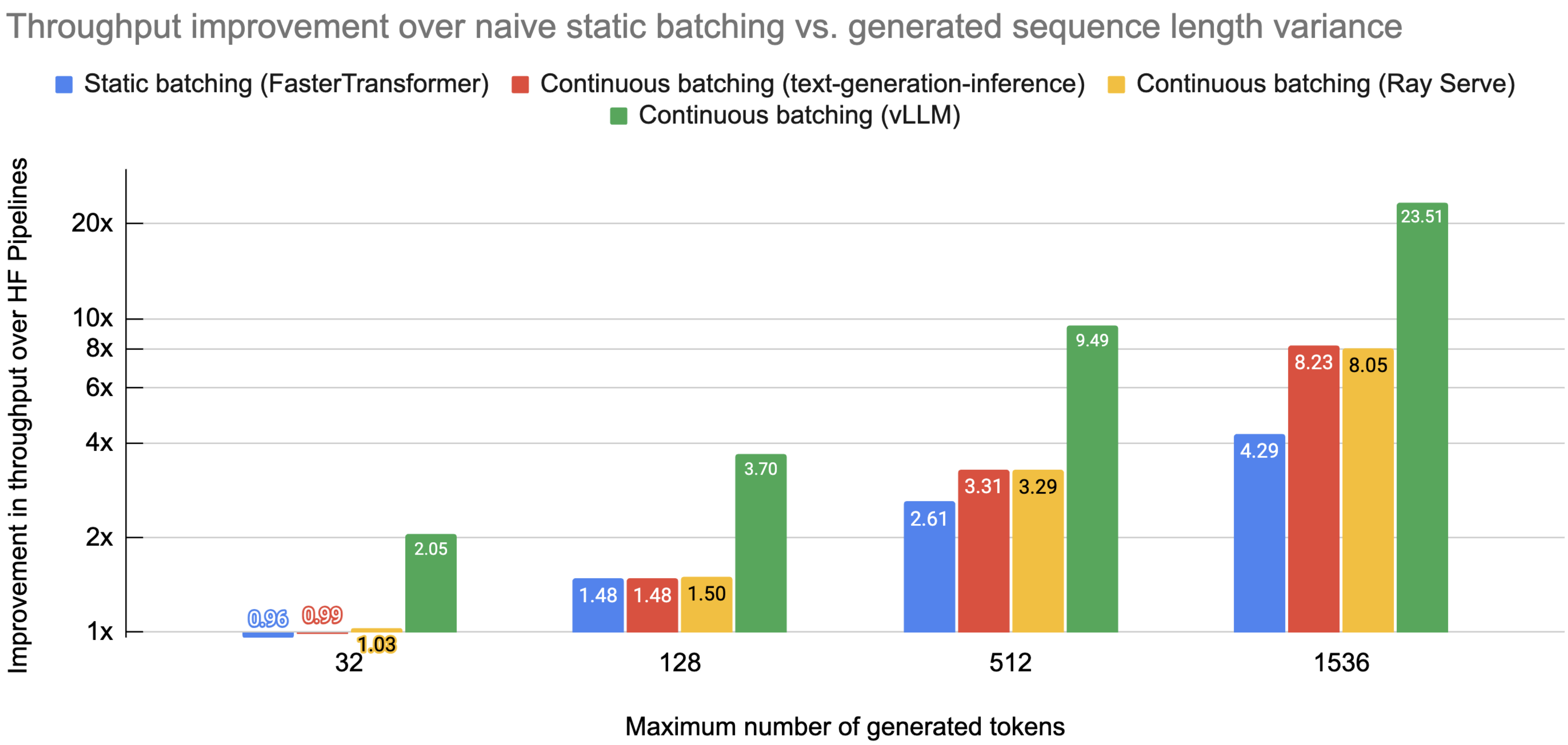
LLM Inference Scheduler



LLM Inference Scheduler

- Enforces iteration-level **first-come-first-served (FCFS)** property
- Maximum batch size → Throughput vs. Latency control knob
- Keep track of number of reserved slots to avoid deadlock
  - Slot := memory required for storing an Attention key and value for a single token
- Reserves max_tokens memory slots per request

Throughput Improvement from Continuous Batching



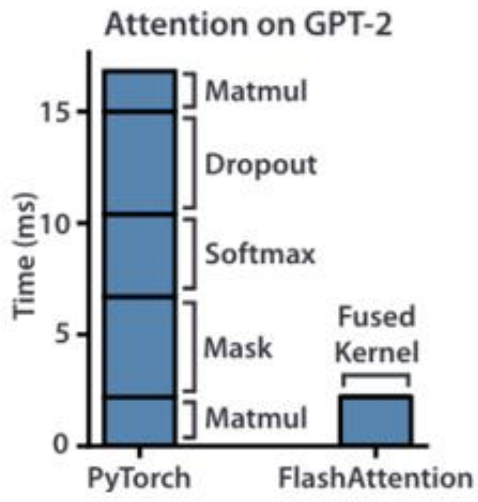
https://www.anyscale.com/blog/continuous-batching-llm-inference

Continuous Batching Step-by-Step

- Handle early-finished and late-arrived requests more efficiently
- Higher GPU utilization

Questions?

FlashAttention



FlashAttention

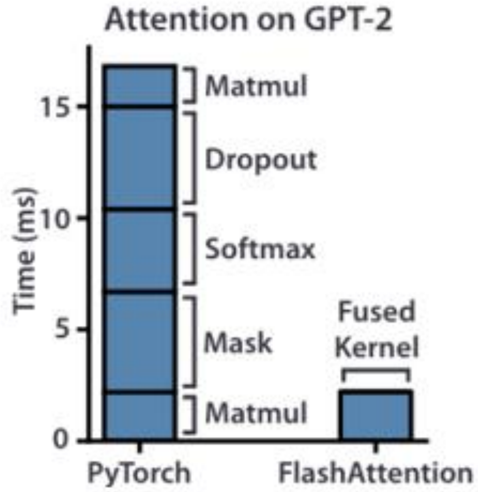


Table 1. Proportions for operator classes in PyTorch.

Operator class	% flop	% Runtime
△ Tensor contraction	99.80	61.0
□ Stat. normalization	0.17	25.5
○ Element-wise	0.03	13.5

- Matrix multiplication takes up 99% of the FLOPS
- But only takes up 61% of the runtime

Question: Why do other operators take so much time?

FlashAttention

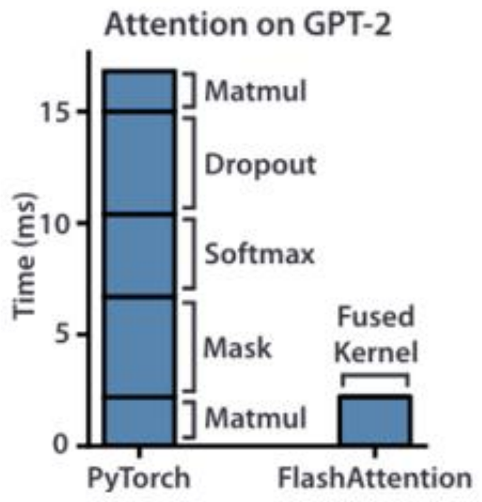


Table 1. Proportions for operator classes in PyTorch.

Operator class	% flop	% Runtime
△ Tensor contraction	99.80	61.0
□ Stat. normalization	0.17	25.5
○ Element-wise	0.03	13.5

- Matrix multiplication takes up 99% of the FLOPS
- But only takes up 61% of the runtime

Question: Why do other operators take so much time?

Inference is usually memory-bound

- Lots of time is wasted moving data around on the GPU instead of doing computation

Generative LLM Inference: Autoregressive Decoding

- **Pre-filling phase** (1-th iteration):
  - Process **all** input tokens at once
- **Decoding phase** (all other iterations):
  - Process a **single** token generated from previous iteration
  - Use attention keys & values of all previous tokens

- Key-value cache:
  - Save attention keys and values for the following iterations to avoid recomputation

Generative LLM Inference: KV Cache