Efficient LLMs

Yatin Nandwani
Research Scientist, IBM Research

Large Language Models: Introduction and Recent Advances

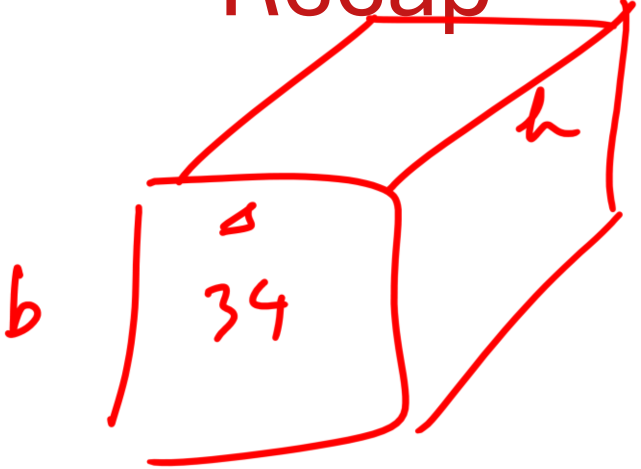
ELL881 · AIL821

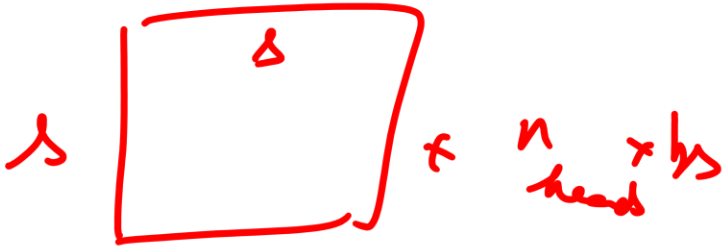
Semester 1,
2025-2026

Recap

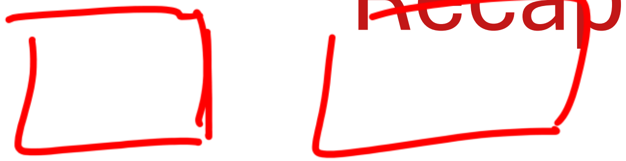
$12h^2 + 13h$

- How to train big models on big data?
- What's in the GPU memory during training?
  i. Model weights ; ii. Param gradients iii. Optim states iv. Activations
- What is the size of params / grads / optim states?
- What is the size of activations?
- How to reduce activation memory?
  - Activation re-computation *aka* Gradient checkpointing
- How to increase batch size?
  - Gradient accumulation (run fwd / bwd *k* times before optim.step())
- Can we parallelize grad. Accumulation? → Data Parallelism
- Can we shard the optim. states, grads, and model params across GPUs? → FSDP
- Still not good for big models & large sequences.





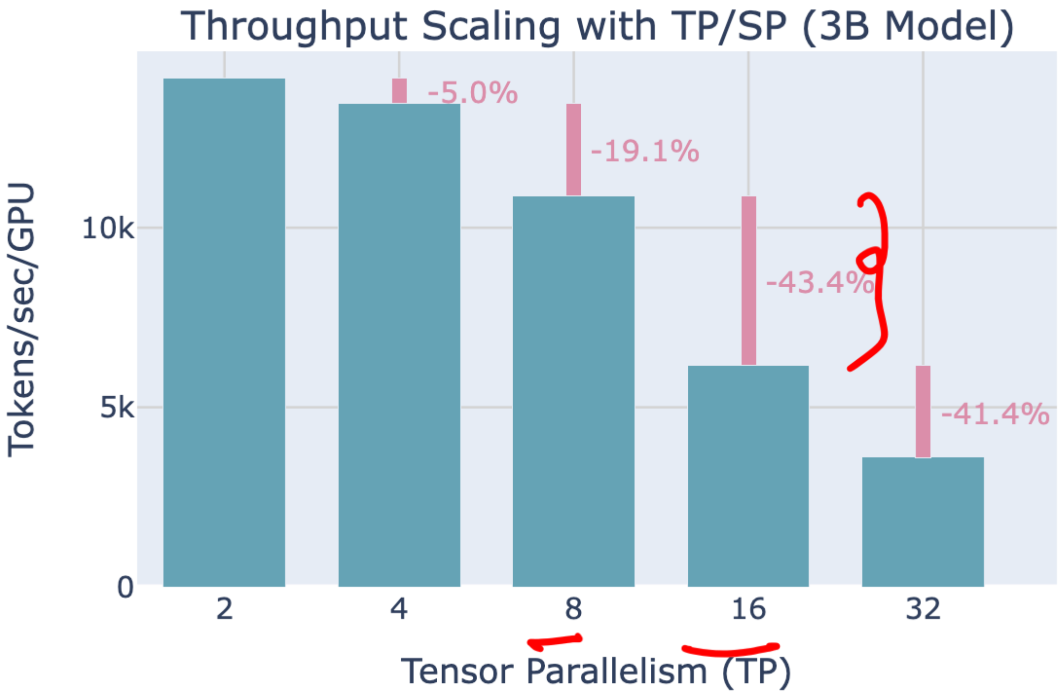
Recap



- Can we split activations of one input across GPUs?
- Split along the hidden dimension -- let's exploit the distributive property of matrix multiplication! → Tensor Parallelism
- TP: both model weights & activations are split across GPUs!
- TP: LayerNorm & Dropout – same computation on same data - wastage of resources :-(
- Combine TP with Sequence Parallel – *reduce & scatter* along seq. length dimension
- How to handle very very long sequences?
- Context Parallel → split sequence into chunks & process each chunk on a different GPU (same weights but different activations on each GPU)
- How to apply attention on a sequence split on multiple GPUs? → Ring Attention
- TP: Communication overhead beyond a node is prohibitive.
- How to handle very large models?

Tensor+Sequence Parallelism - Limitations

1. If we scale the sequence length the activation memory will still blow up in the TP region



Context Parallelism

- TP: Split a model across one node to tame large models
- CP: Tame the activation explosion with long sequences.

- TP: doesn't scale well across nodes

Tensor+Sequence Parallelism - Limitations

1. If we scale the sequence length the activation memory will still blow up in the TP region

Context Parallelism

- TP: Split a model across one node to tame large models
- CP: Tame the activation explosion with long sequences.

- TP: doesn't scale well across nodes
- How about splitting layers across GPUs?

Tensor+Sequence Parallelism - Limitations

1. If we scale the sequence length the activation memory will still blow up in the TP region

2. If the model is too big to fit with TP=8 we will see a massive slowdown due to the inter-node connectivity.

Pipeline Parallelism

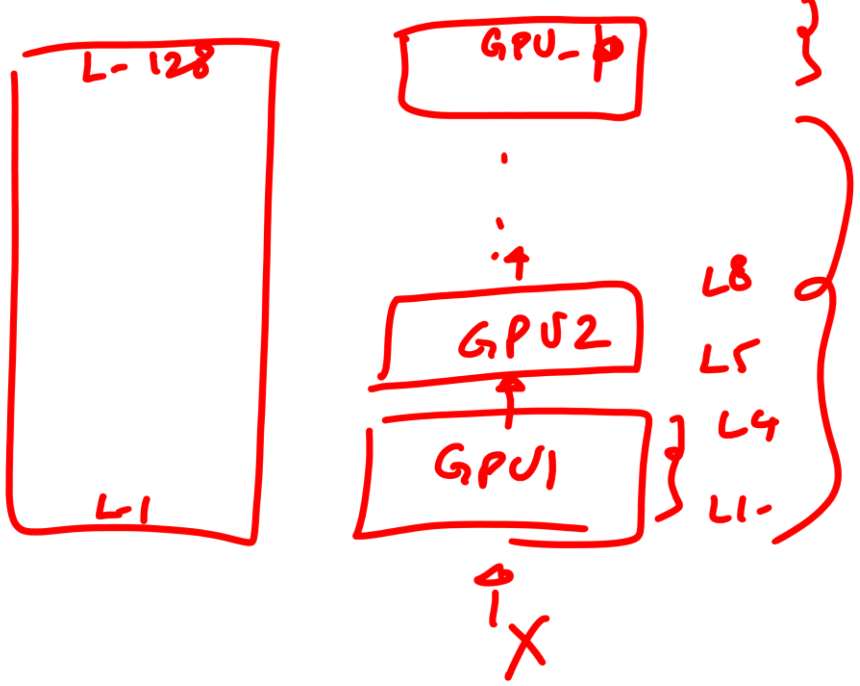
Context Parallelism

- TP: Split a model across one node to tame large models
- CP: Tame the activation explosion with long sequences.

- TP: doesn't scale well across nodes
- How about splitting layers across GPUs?

Pipeline Parallelism

- Split model's layers across multiple GPUs.
- E.g., layers 1-4 on GPU 1, layers 5-8 on GPU 2, and so on.
- Each GPU stores and process a portion of the model's layers, significantly reducing the memory requirements per GPU
- Required interconnect bandwidth stays quite low: send moderate-sized activations at a handful of locations along the model depth
- What is the main issue with this design?

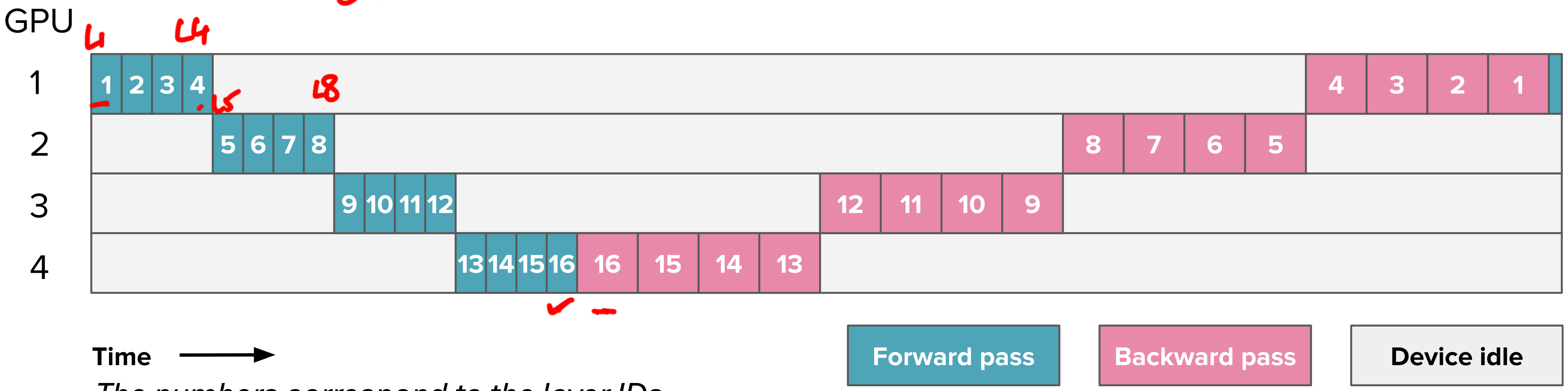


AFAB: All forward, All backward

Idle = $\frac{p.T - T}{T} = p-1$

Ideal

One Batch in this Dia



The numbers correspond to the layer IDs

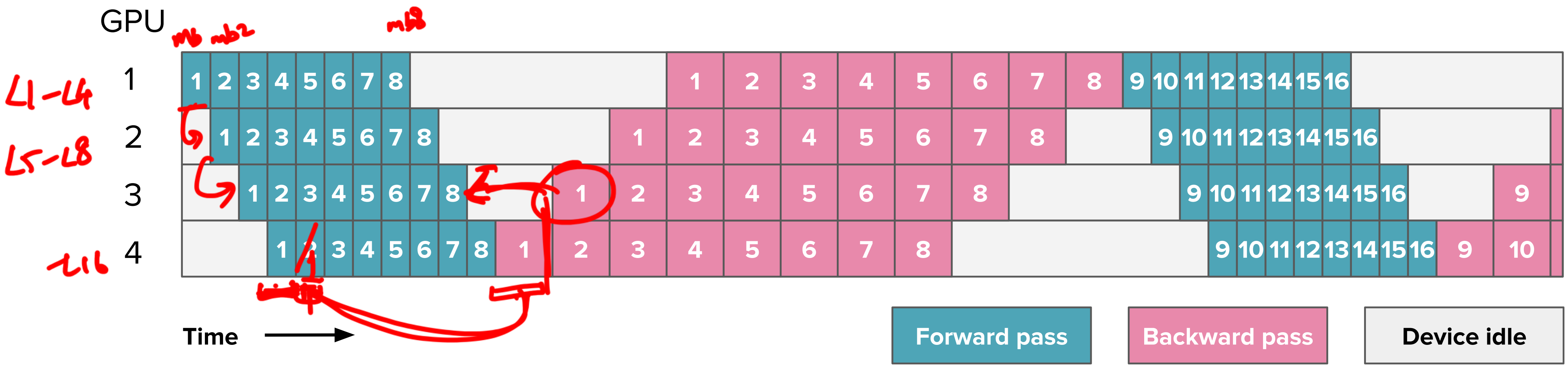
- **Bubble:** GPU idle time (gray color)

Ideal Time t_{id}	= $t_f + t_b$
Additional Time (PP bubble) t_{pb}	= $(p-1) * (t_f + t_b)$ [p: #GPUs]
Ratio r_{bubble}	= $(p-1)$

- Is there a way to reduce the bubble?

L1 – L4

AFAB: All forward, All backward

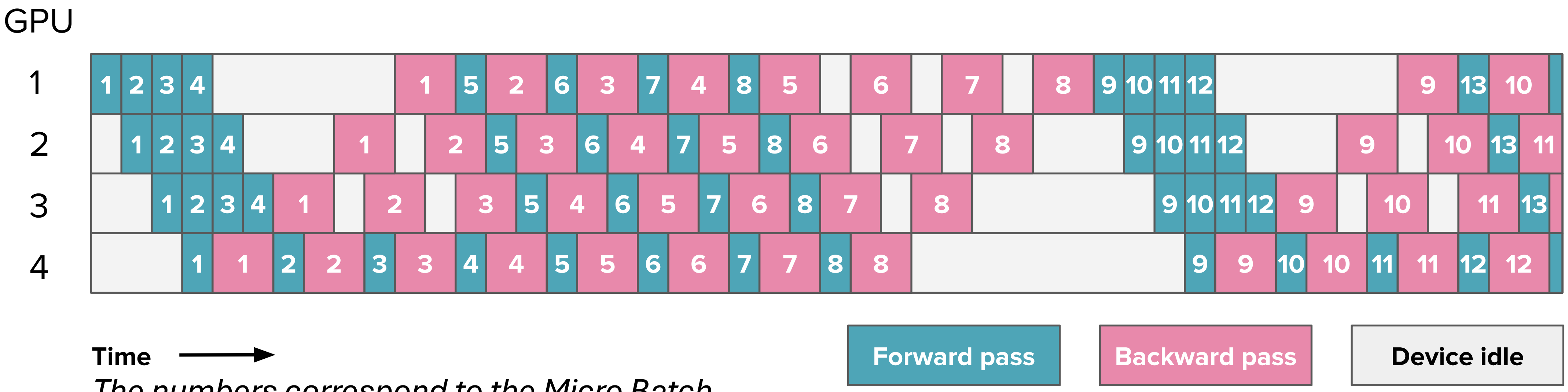


The numbers correspond to the Micro Batch

Ideal Time t_{id}	$= m * (t_f + t_b)$
Additional Time (PP bubble) t_{pb}	$= (p-1) * (t_f + t_b)$ [p: #GPUs]
Ratio r_{bubble}	$= (p-1)/m$

- Can we indefinitely increase m?
- No! Activation memory will explode - need to keep them till bwd pass.
- Is there any alternative to avoid activation storage (and hence increase m)?

1F1B: One forward, One backward

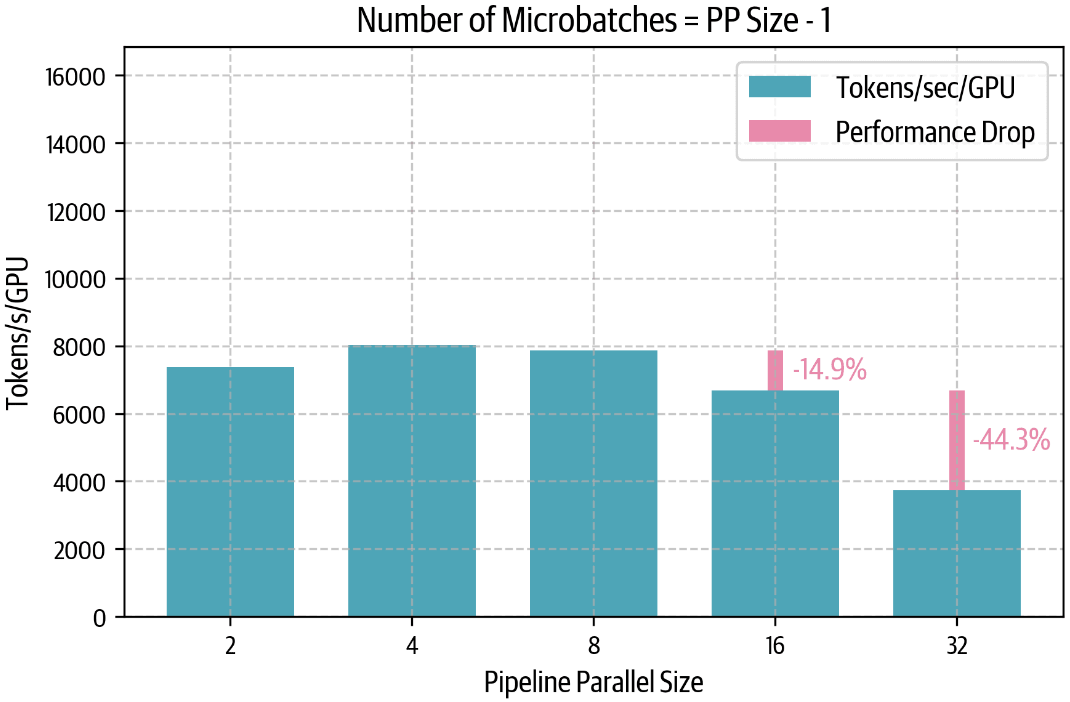


The numbers correspond to the Micro Batch

Ideal Time t_{id}	= $m * (t_f + t_b)$
Additional Time (PP bubble) t_{pb}	= $(p-1) * (t_f + t_b)$ [p: #GPUs]
Ratio r_{bubble}	= $(p-1)/\underline{m}$

1F1B: One forward, One backward

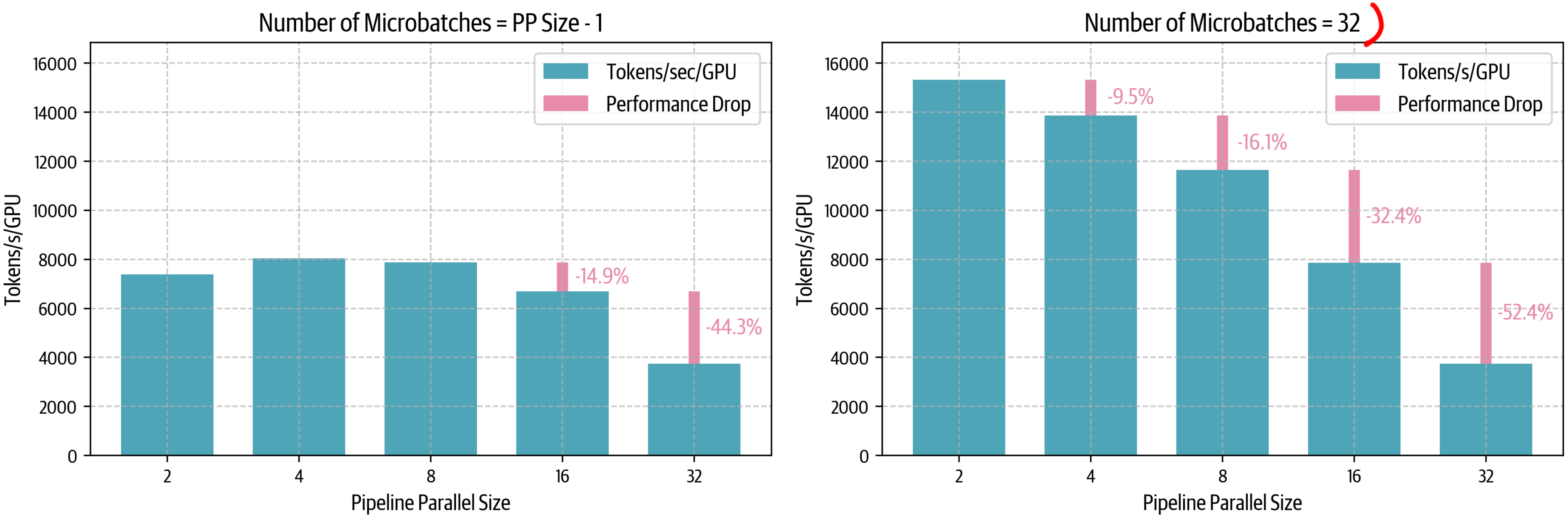
Throughput scaling with Pipeline Parallelism: 1F1B schedule



r_{bubble}	$= (p-1)/m = 1$

1F1B: One forward, One backward

Throughput scaling with Pipeline Parallelism: 1F1B schedule

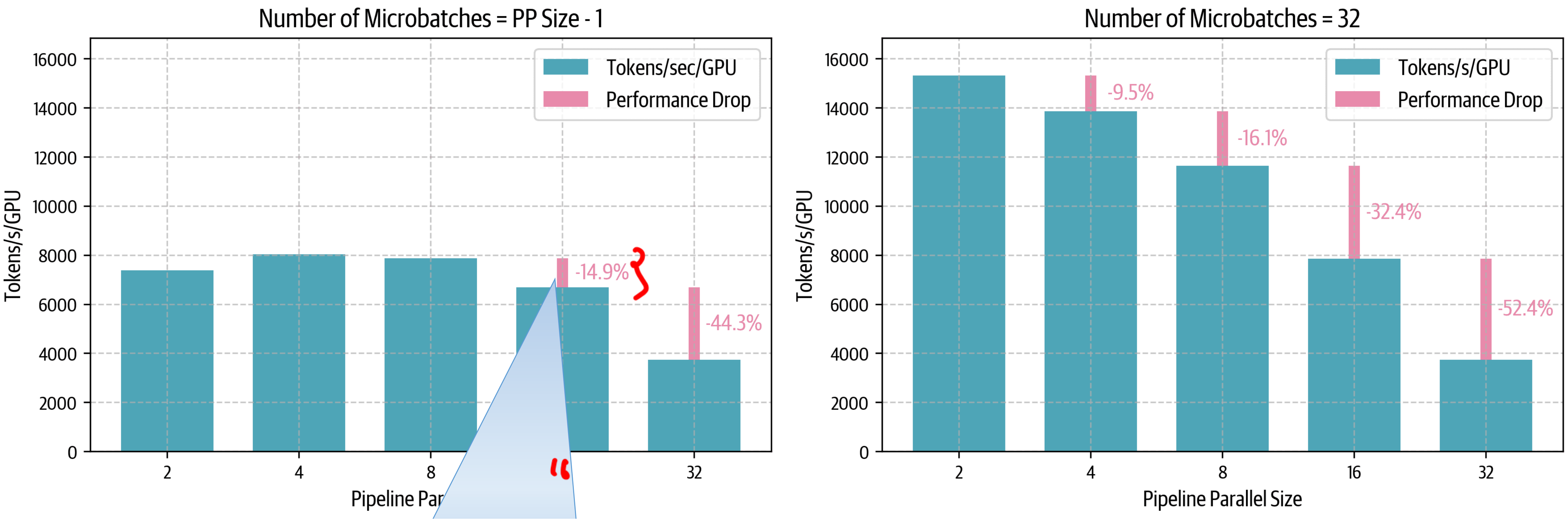


r_{bubble} = $(p-1)/m$ = 1

$1/32$	$3/32$	$7/32$	$15/32$	$31/32$

1F1B: One forward, One backward

Throughput scaling with Pipeline Parallelism: 1F1B schedule



r_{bubble}	$= (p-1)/m = 1$

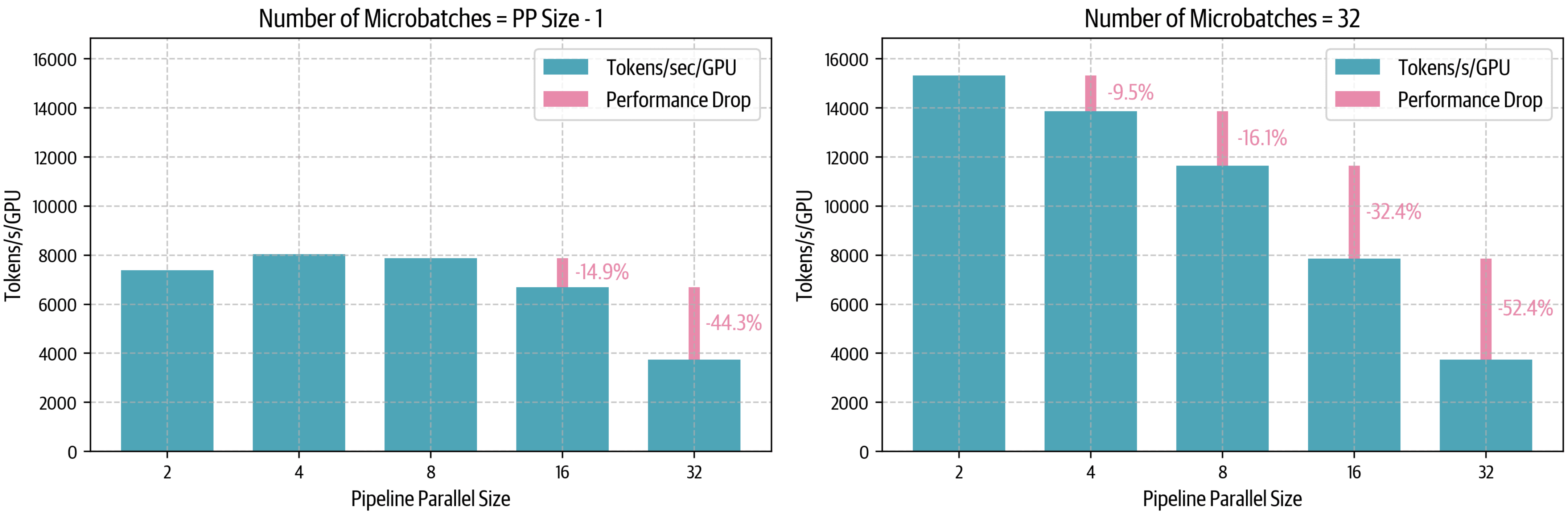
$1/32$	$3/32$	$7/32$	$15/32$	$31/32$

- Only 15% drop in a cross-node scenario
- Much better than 43% in TP

L1 – L4

1F1B: One forward, One backward

Throughput scaling with Pipeline Parallelism: 1F1B schedule

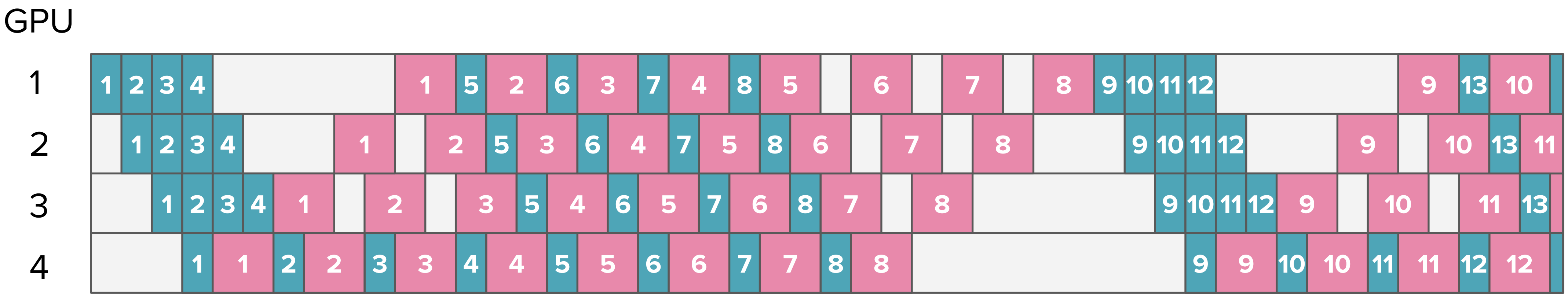


r_{bubble}	$= (p-1)/m$ = 1

1/32	3/32	7/32	15/32	31/32

- 1F1B helps in reducing memory and thus increasing m $(p-1)/m$ ↑
- No effect on the size of the bubble -- numerator is still $(p-1)$
- Can we borrow ideas from Zig-Zag allocation in Ring Attention?

1F1B: One forward, One backward



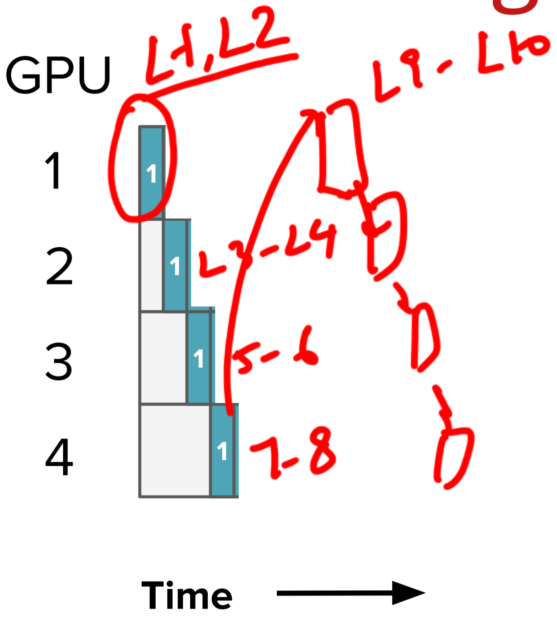
Time →

The numbers correspond to the Micro Batch

Forward pass **Backward pass** **Device idle**

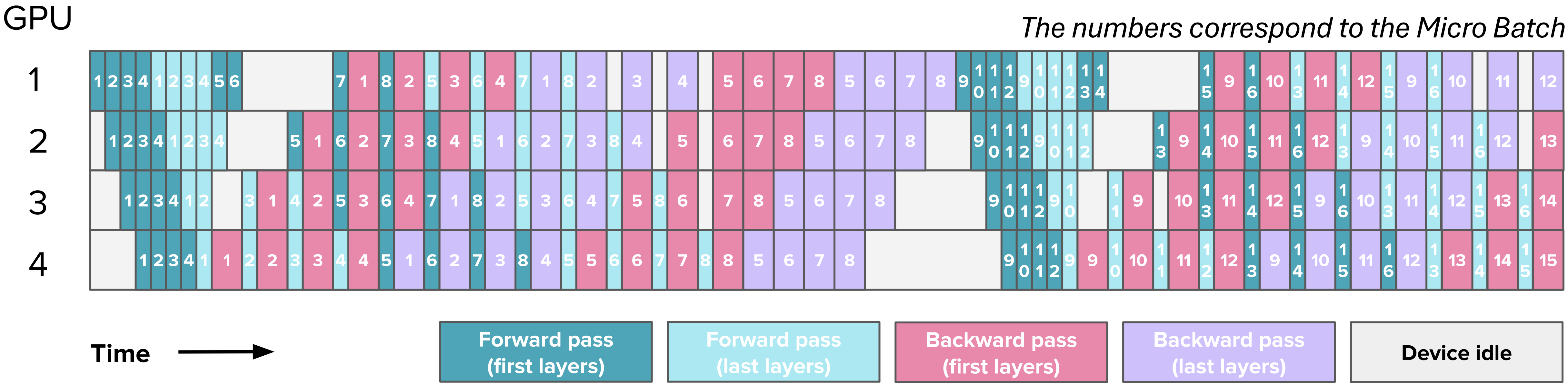
Ideal Time t_{id}	= $m * (t_f + t_b)$
Additional Time (PP bubble) t_{pb}	= $(p - 1) * (t_f + t_b)$ [p: #GPUs]
Ratio r_{bubble}	= $(p - 1)/m$

Interleaving Stages



The numbers correspond to the Micro Batch

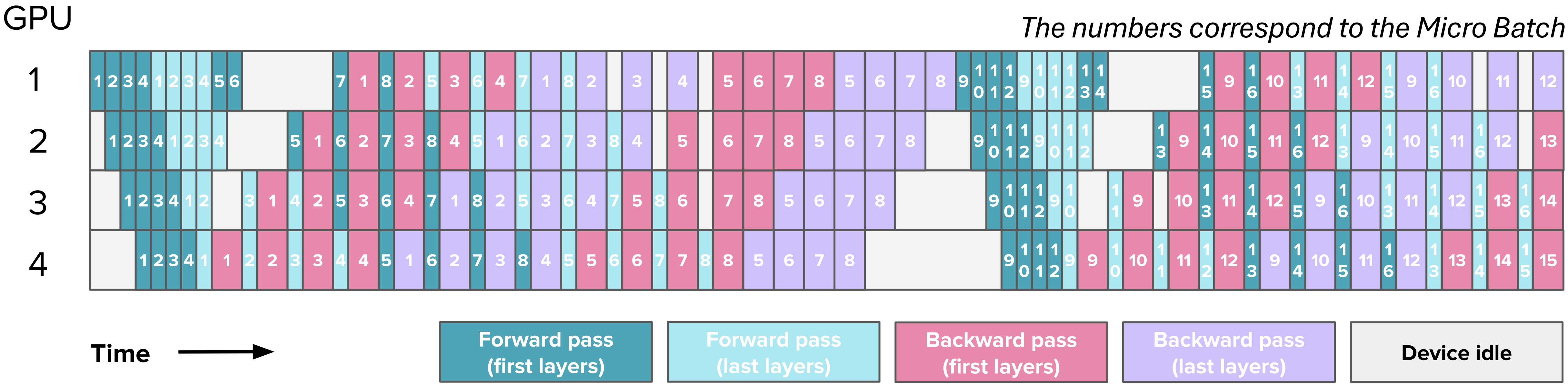
Interleaving Stages


GPU
1
2
3
4
Time
Forward pass (first layers)
Forward pass (last layers)
Backward pass (first layers)
Backward pass (last layers)
Device idle

The numbers correspond to the Micro Batch

- **Looping Pipeline:** micro-batch moves in circles
- **Additional communication:** same GPU visited multiple times.

Interleaving Stages

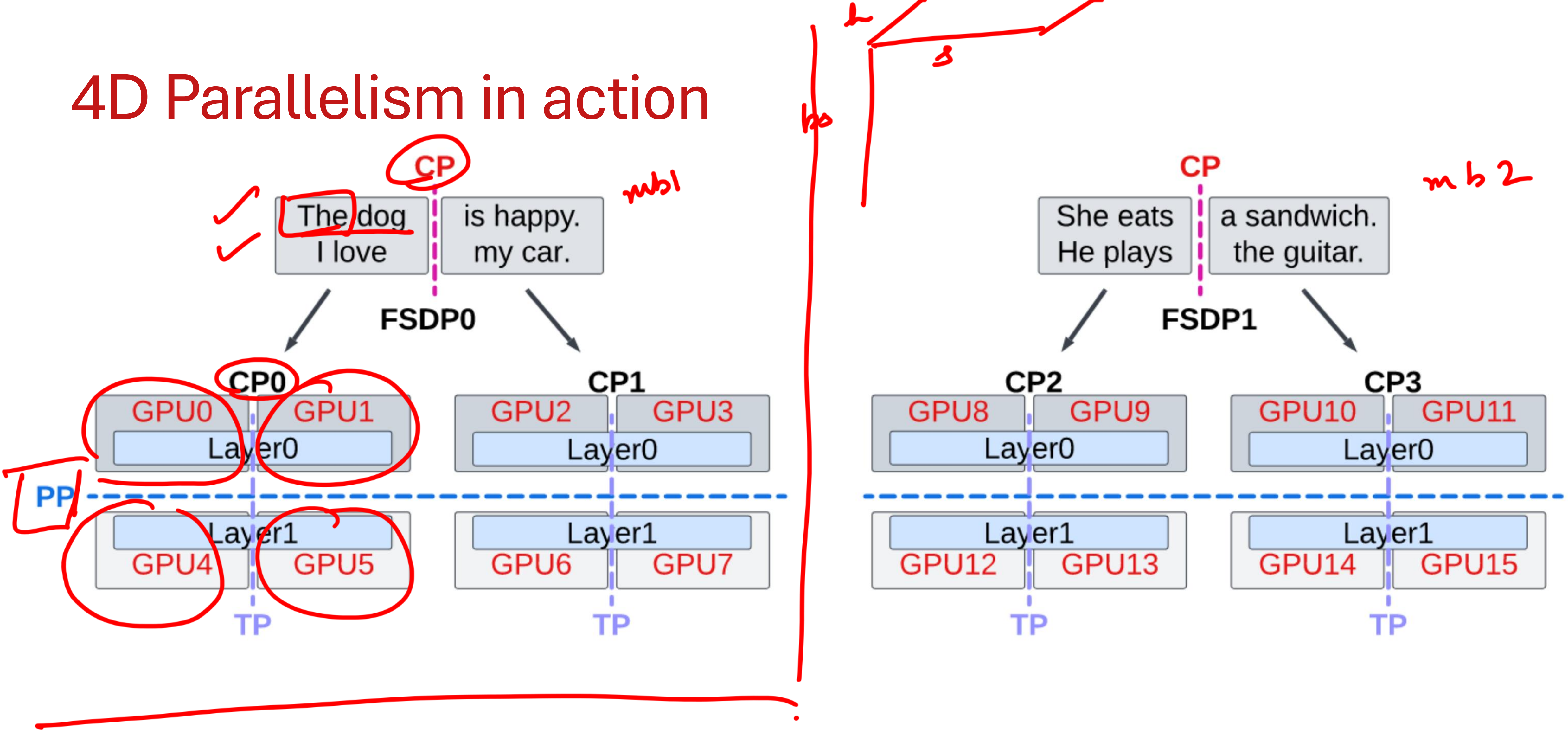


# Stages or model chunks per GPU	v
Ideal Time t_{id}	= $m * (t_f + t_b)$
Additional Time (PP bubble) t_{pb}	= $(p-1) * (t_f + t_b)/v$ [p: #GPUs]
Ratio r_{bubble}	= $(p-1)/(v * m)$

4D Parallelism

1. Data Parallel & ZeRO-1/2/3
2. Tensor Parallel (w/ Sequence Parallel)
3. Context Parallel
4. Pipeline Parallel

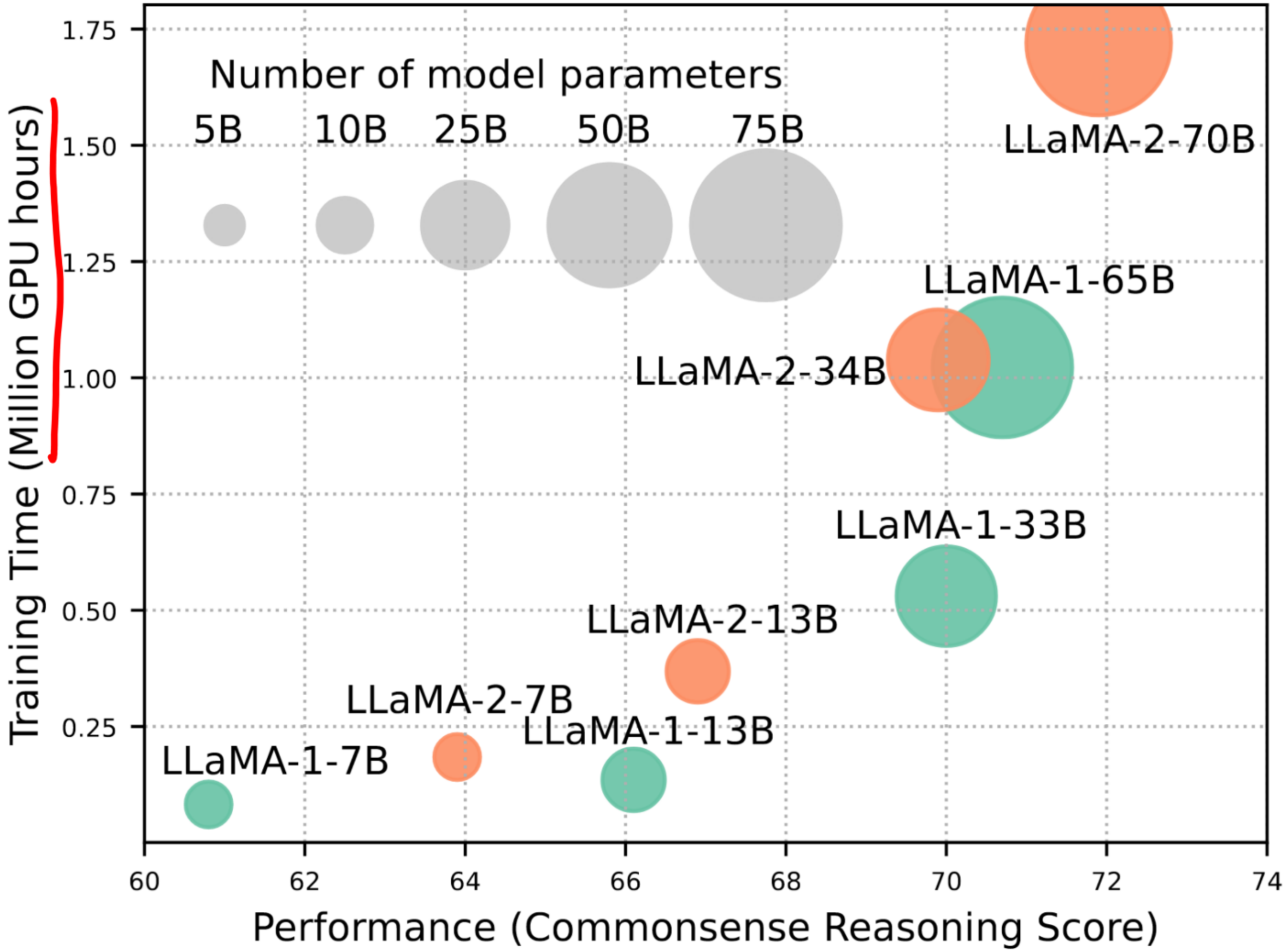
4D Parallelism in action



[Scaling Llama 3 Training with Efficient Parallelism Strategies, Chu et al. 2025]

Let’s revisit the motivation …

Training Resources vs Performance

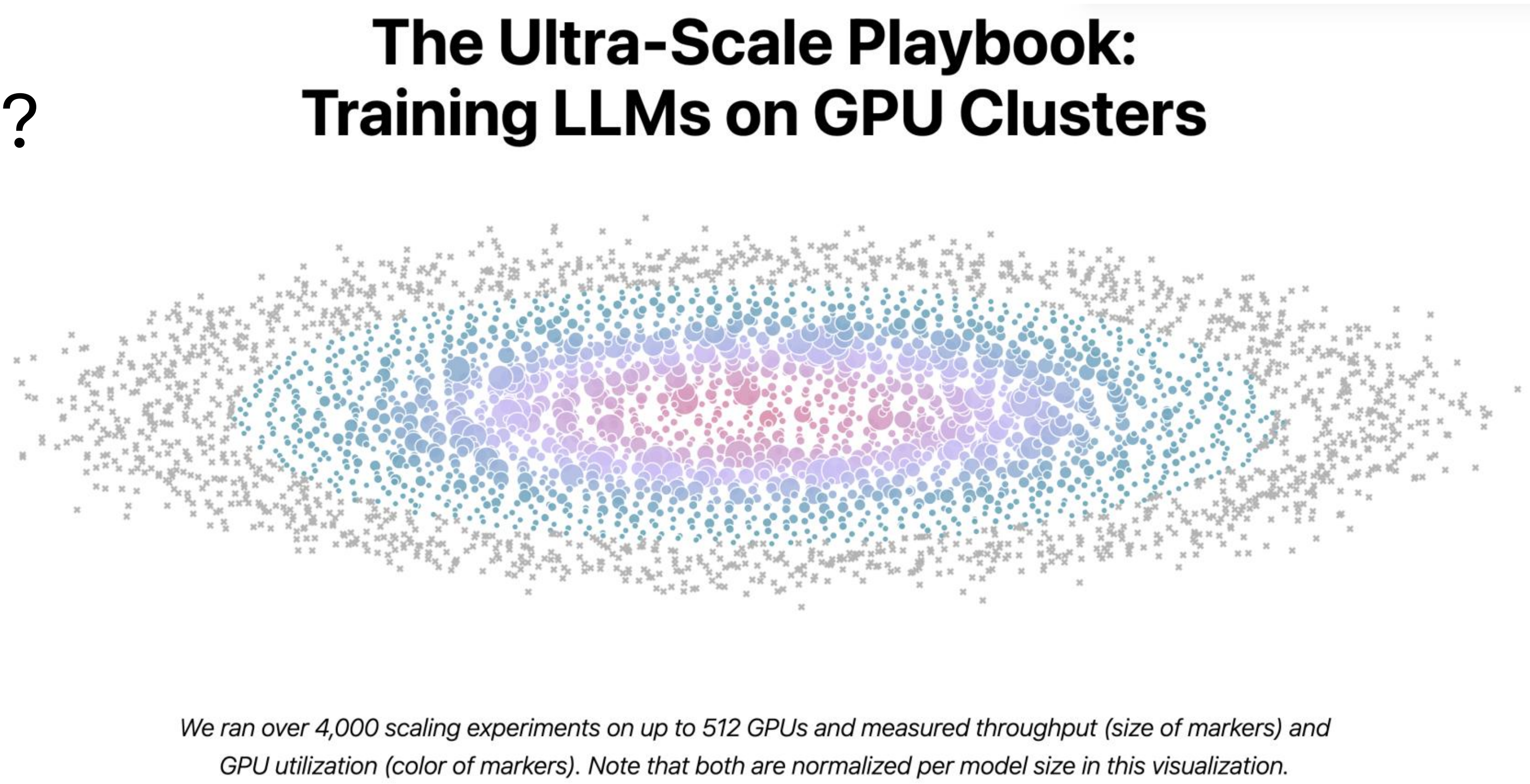


- Based on Nvidia A100 80GB GPU

https://huggingface.co/spaces/optimum/llm-perf-leaderboard

Efficient LLMs

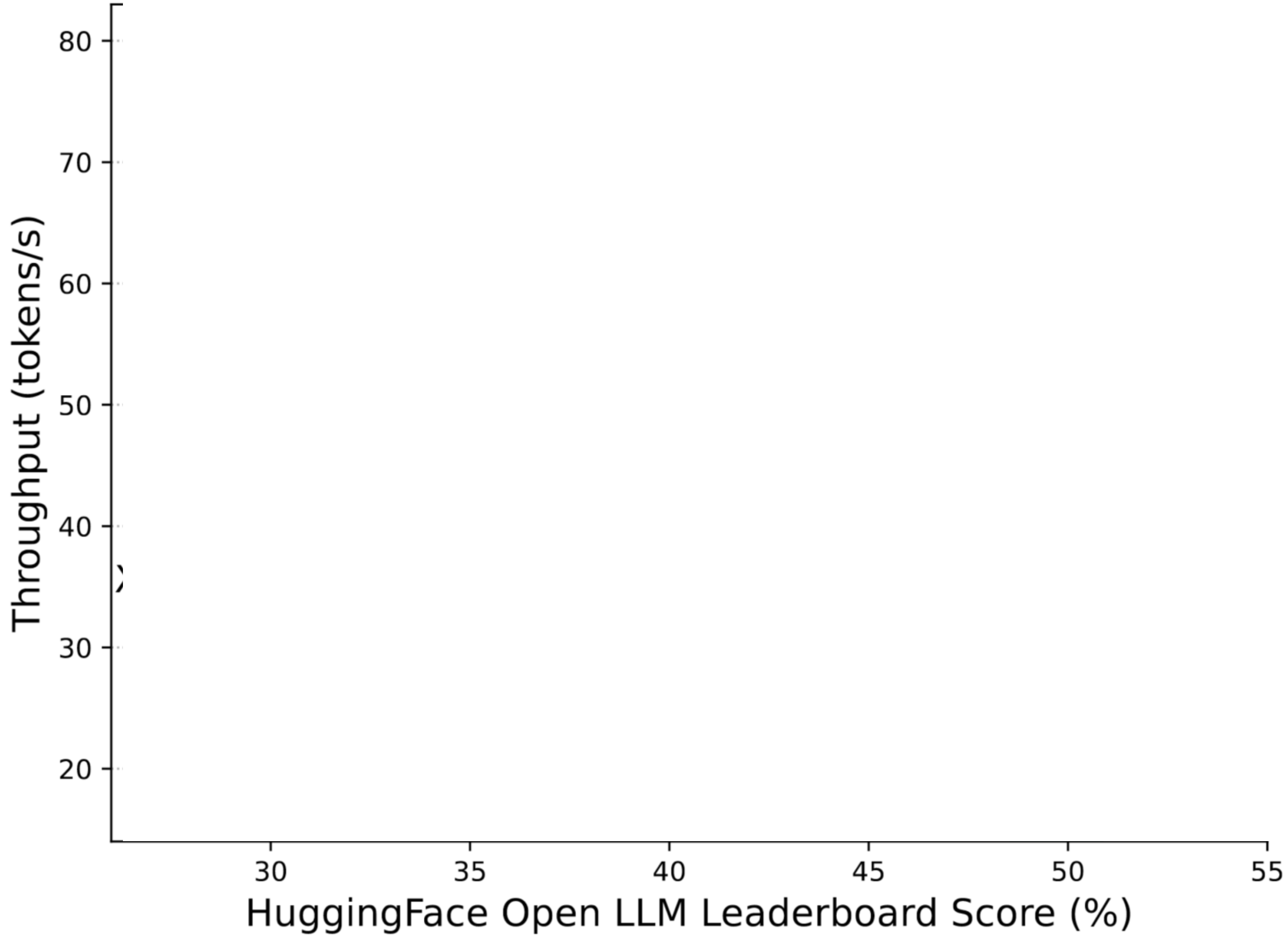
- How to scale training?
  - *Data Parallelism*
  - *Tensor Parallelism*
  - *Context Parallelism*
  - *Pipeline Parallelism*



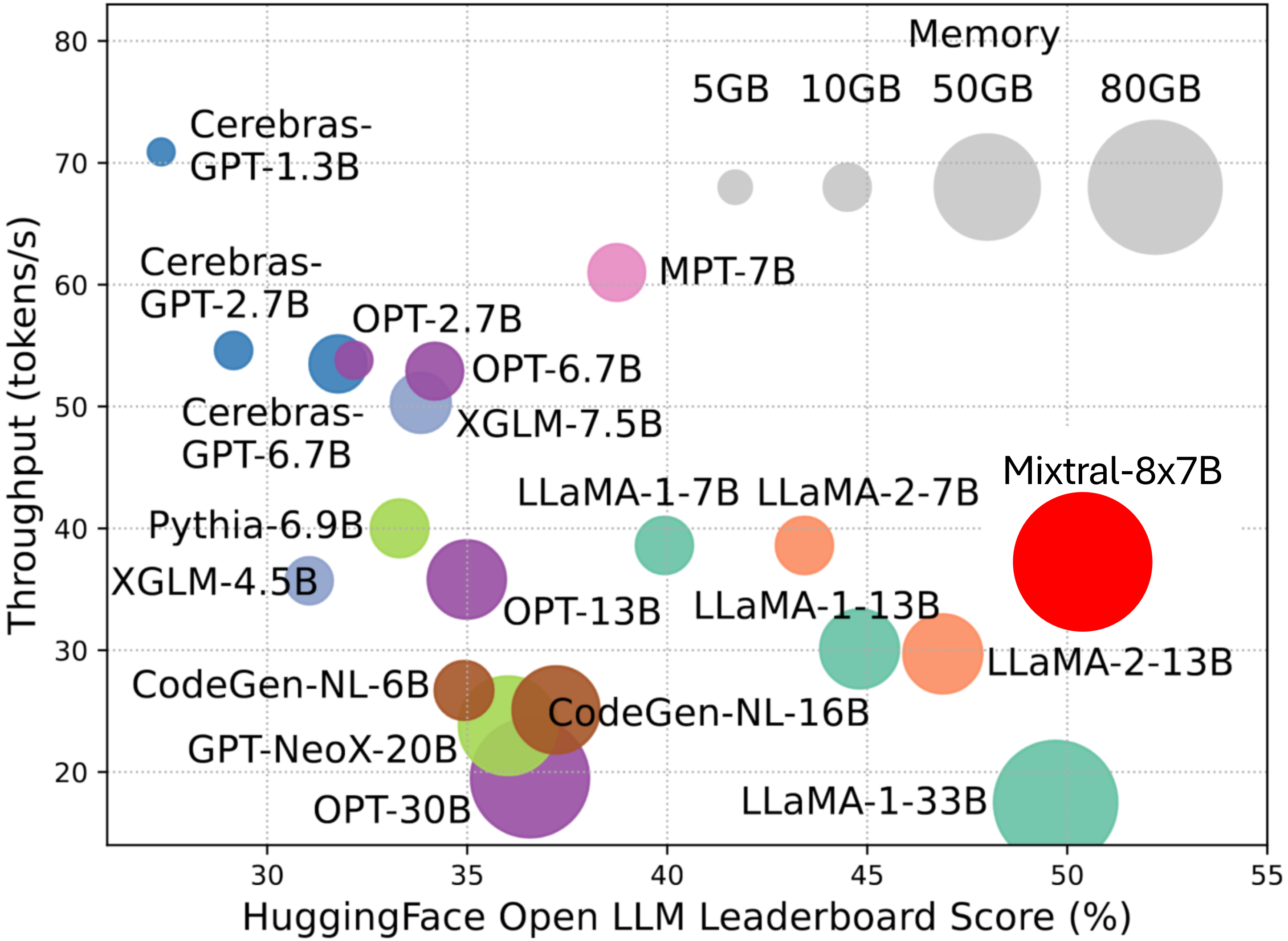
The Ultra-Scale Playbook: Training LLMs on GPU Clusters

We ran over 4,000 scaling experiments on up to 512 GPUs and measured throughput (size of markers) and GPU utilization (color of markers). Note that both are normalized per model size in this visualization.

Inference Throughput vs Performance

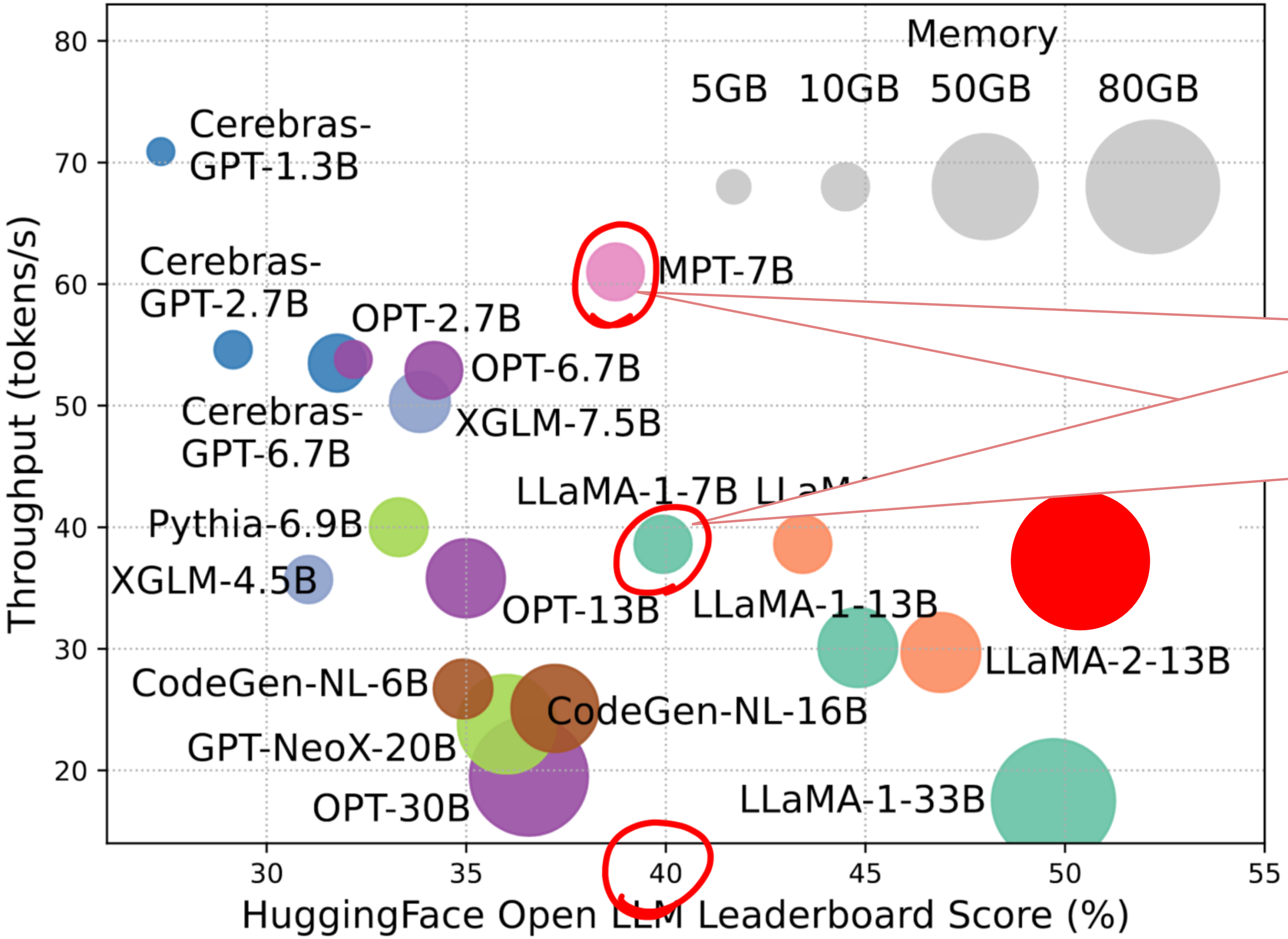


Inference Throughput vs Performance



- On Nvidia A100 80GB GPU;
- 16-bit quantized
- Batch Size - 1
- Prompt size of 256
- Generating 1000 tokens

Inference Throughput vs Performance



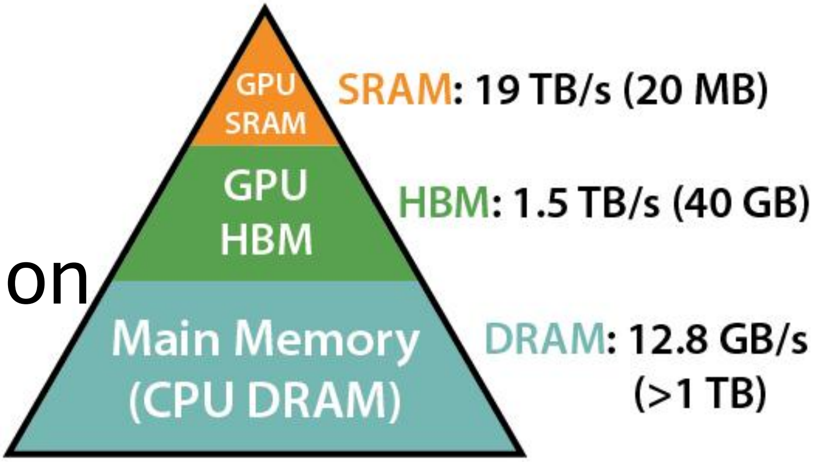
- Similar performance, different throughput! How?
- Efficient implementation –
  - Fused kernel for attention

Efficient LLMs

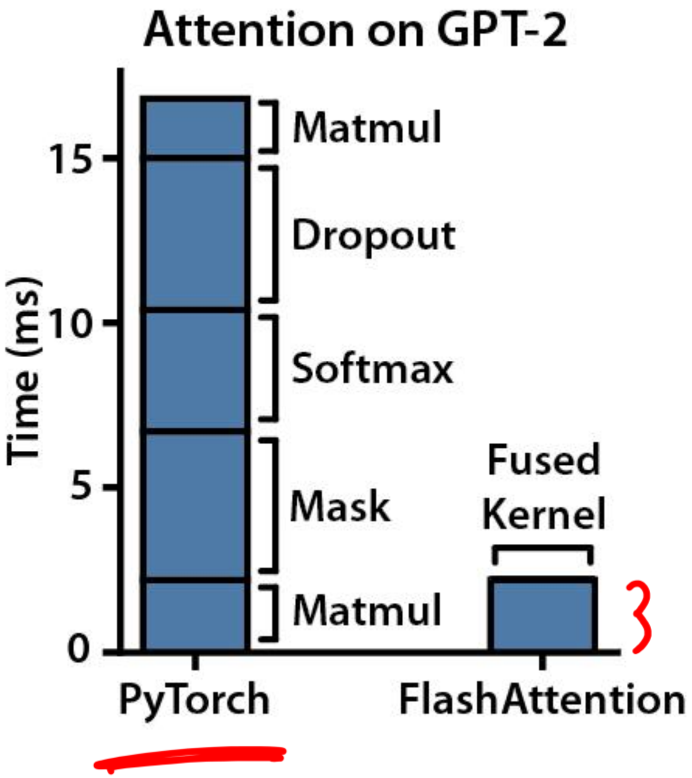
- How to scale training?

  Parallelism …

- Efficient implementation
  - Flash Attention
  - Paged Attention



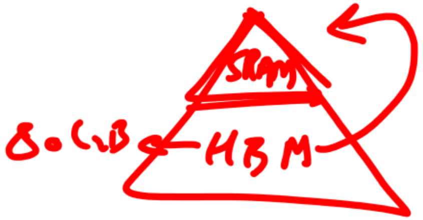
Memory Hierarchy with Bandwidth & Memory Size



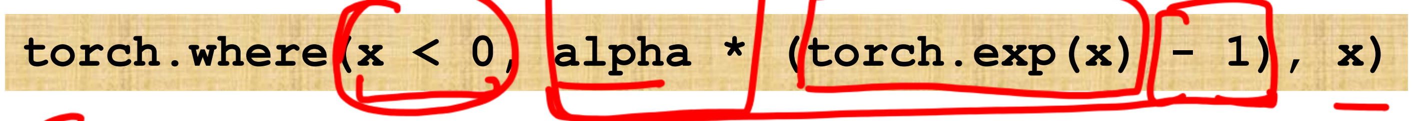
Attention on GPT-2

Efficient Implementation of Attention

GPU Basics



#1 #4 #2 #3



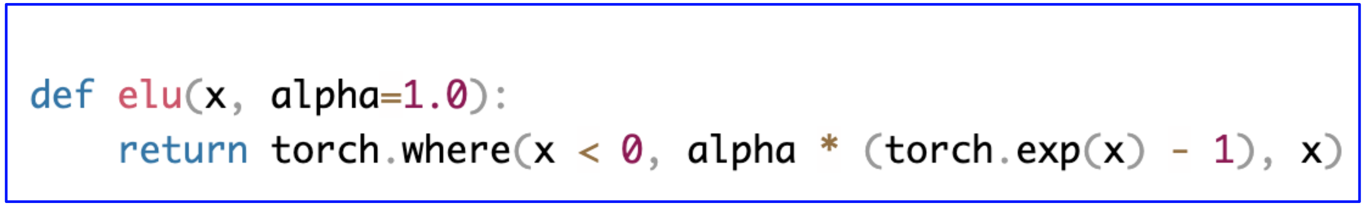
```
torch.where(x < 0, alpha * (torch.exp(x) - 1), x)
```

What is a kernel?

- A piece of code running on a core of the GPU
- Implements basic operations – vector addition, elementwise multiplication, matrix multiplication etc.
- Written in CUDA or Triton, compiled to low level assembly

1. **lt_kernel** → produces mask from x < 0
2. **exp_kernel** → computes *exp(x)*
3. **sub_kernel** → computes *exp(x) - 1*
4. **mul_kernel** → computes *alpha * (…)*
5. **where_kernel** → chooses between *alpha*(exp(x)-1)* and *x*

All tensor manipulations are converted to a series of kernel calls.



```
def elu(x, alpha=1.0):
    return torch.where(x < 0, alpha * (torch.exp(x) - 1), x)
```

Can we create a custom kernel to replace a series of kernel calls that we use repeatedly?

Yes! That's called a fused kernel.

GPU Basics

What is a kernel?

- A piece of code running on a core of the GPU
- Implements basic operations – vector addition, elementwise multiplication, matrix multiplication etc.
- Written in CUDA or Triton, compiled to low level assembly

All tensor manipulations are converted to a series of kernel calls.

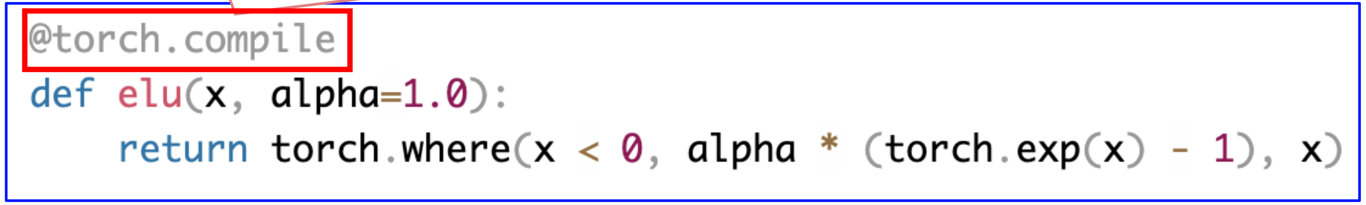
Can we create a custom kernel to replace a series of kernel calls that we use repeatedly?

Yes! That's called a fused kernel.

```
torch.where(x < 0, alpha * (torch.exp(x) - 1), x)
```

1. **lt_kernel** → produces mask from x < 0
2. **exp_kernel** → computes *exp(x)*
3. **sub_kernel** → computes *exp(x) - 1*
4. **mul_kernel** → computes *alpha * (...)*
5. **where_kernel** → chooses between *alpha*(exp(x)-1)* and *x*

Decorator to dynamically compile fn into a kernel



```
@torch.compile
def elu(x, alpha=1.0):
    return torch.where(x < 0, alpha * (torch.exp(x) - 1), x)
```

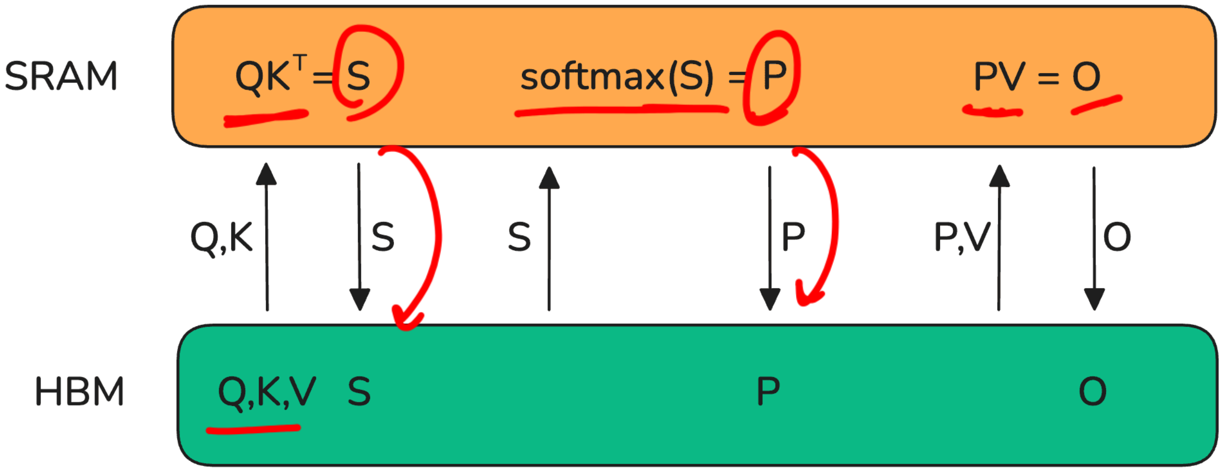
GPU Basics - Memory Hierarchy

When a kernel runs:

- Tensors are first moved to SRAM from HBM
- Computation happens
- Results written back to HBM

100

- A lot of transfer b/w memory & workers
- Bottleneck: Lower bandwidth in HBM

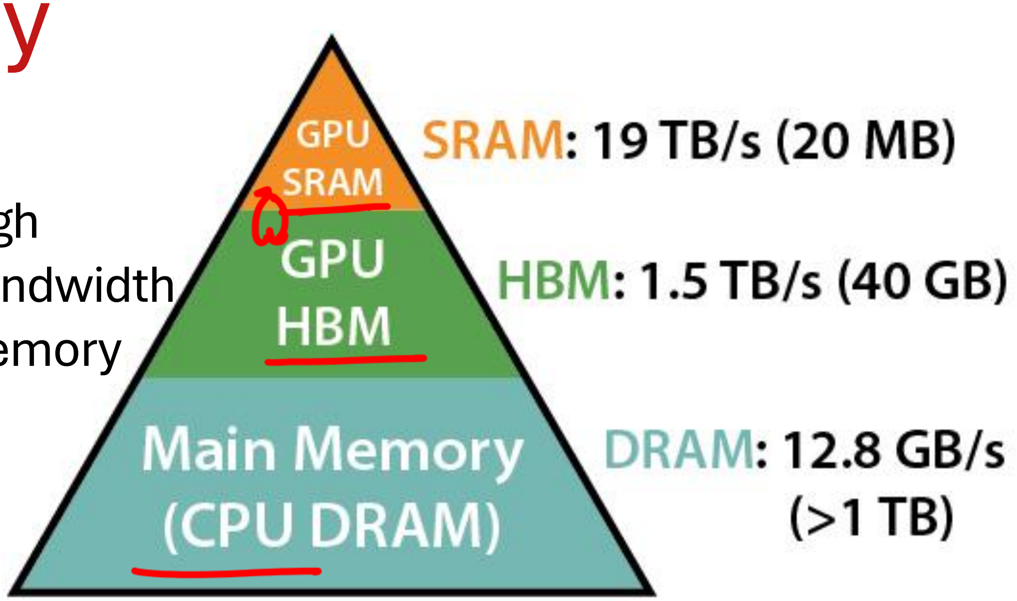


SRAM: $QK^T = S$ softmax(S) = P PV = O

Q,K ↑ S ↓ S ↑ P ↓ P,V ↑ O ↓

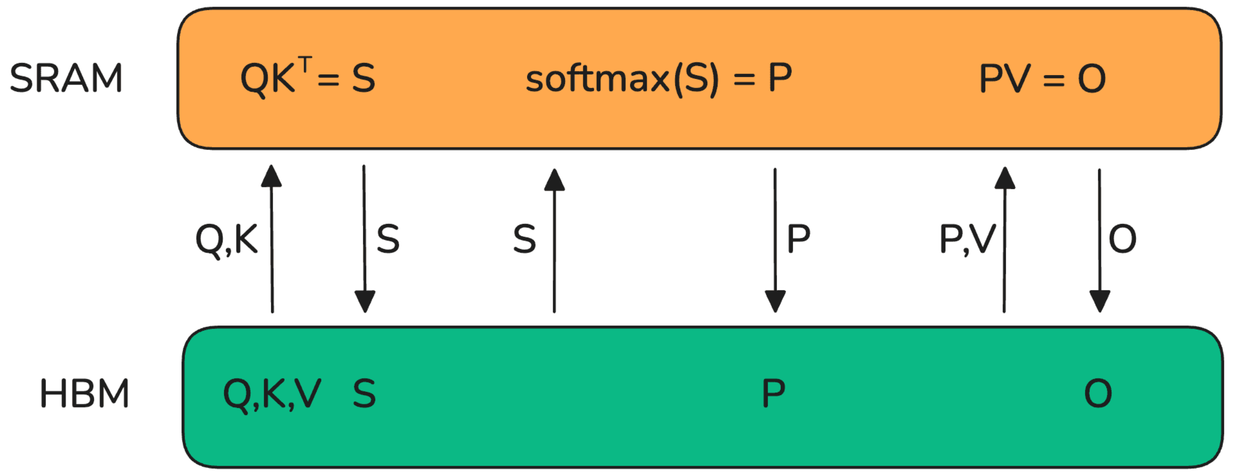
HBM: Q,K,V S P O

High Bandwidth Memory



Memory Hierarchy with Bandwidth & Memory Size

Flash Attention



SRAM

$QK^T = S$ softmax(S) = P PV = O

Q,K S S P P,V O

HBM

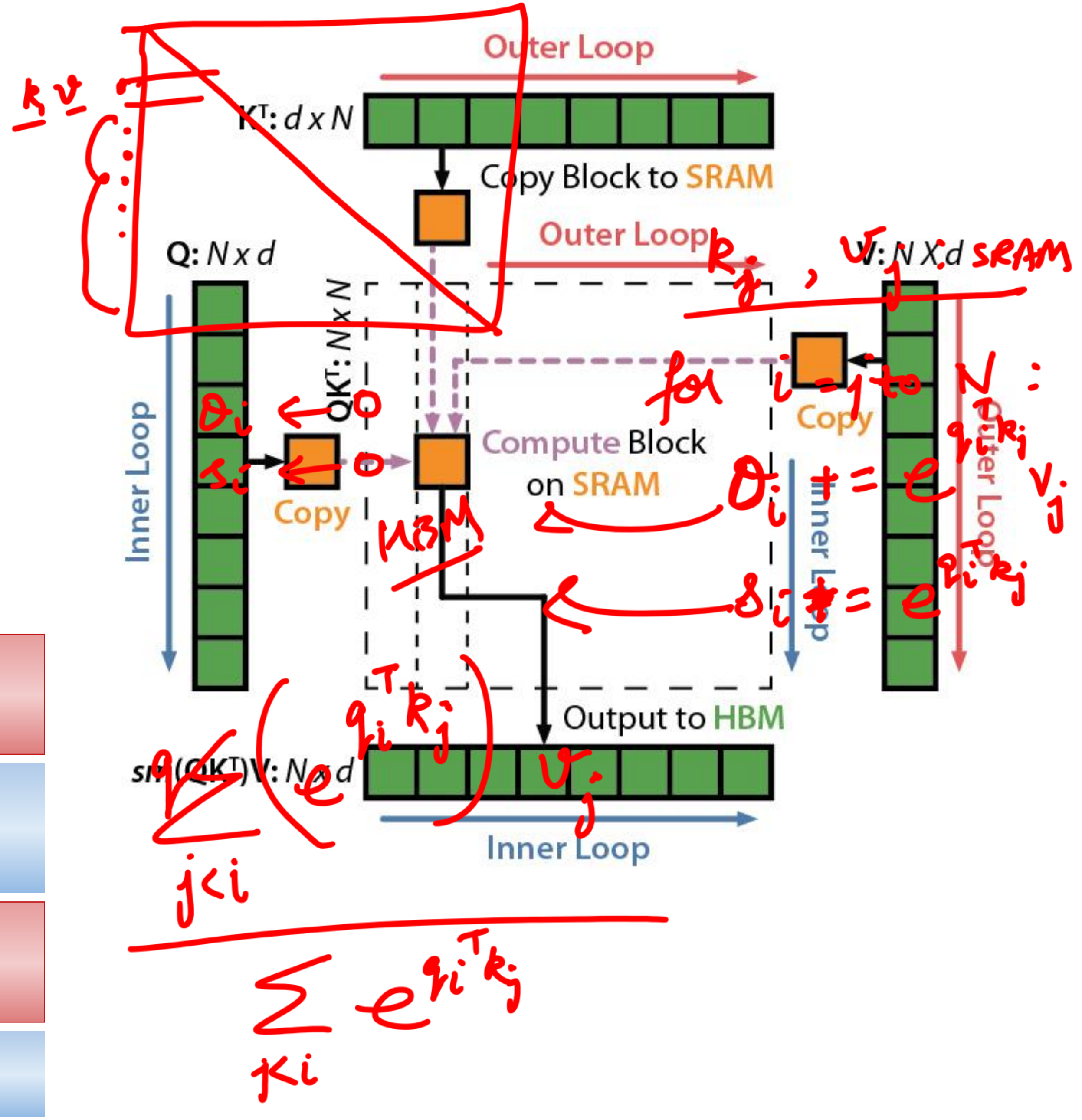
Q,K,V S P O

- A lot of transfer b/w memory & workers
- Bottleneck: Lower bandwidth in HBM

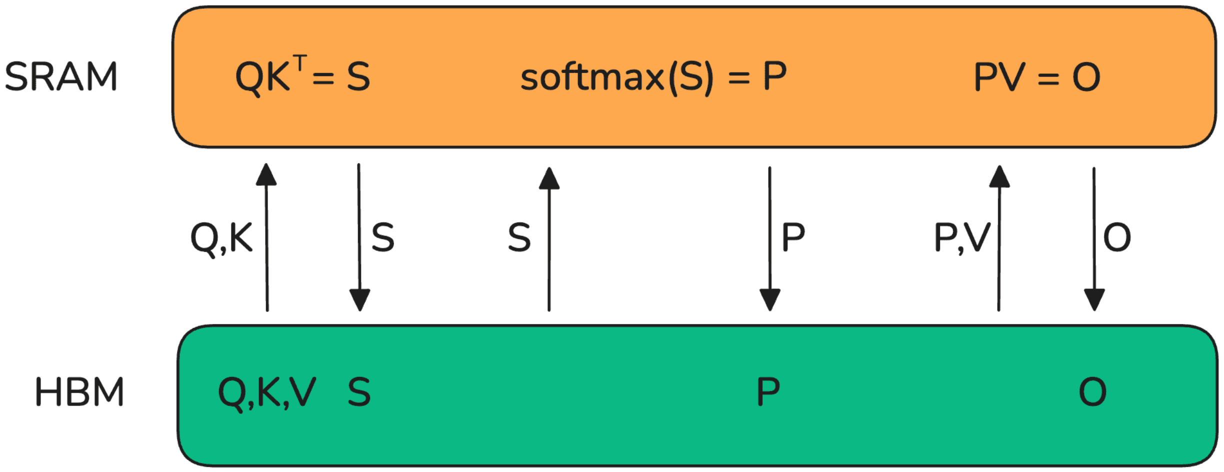
Let's write a fused kernel for attn that avoids back & forth b/w HBM and SRAM

But SRAM is limited ☹
Can we get away with S matrix?

Does Ring Attention rings a bell?



Flash Attention



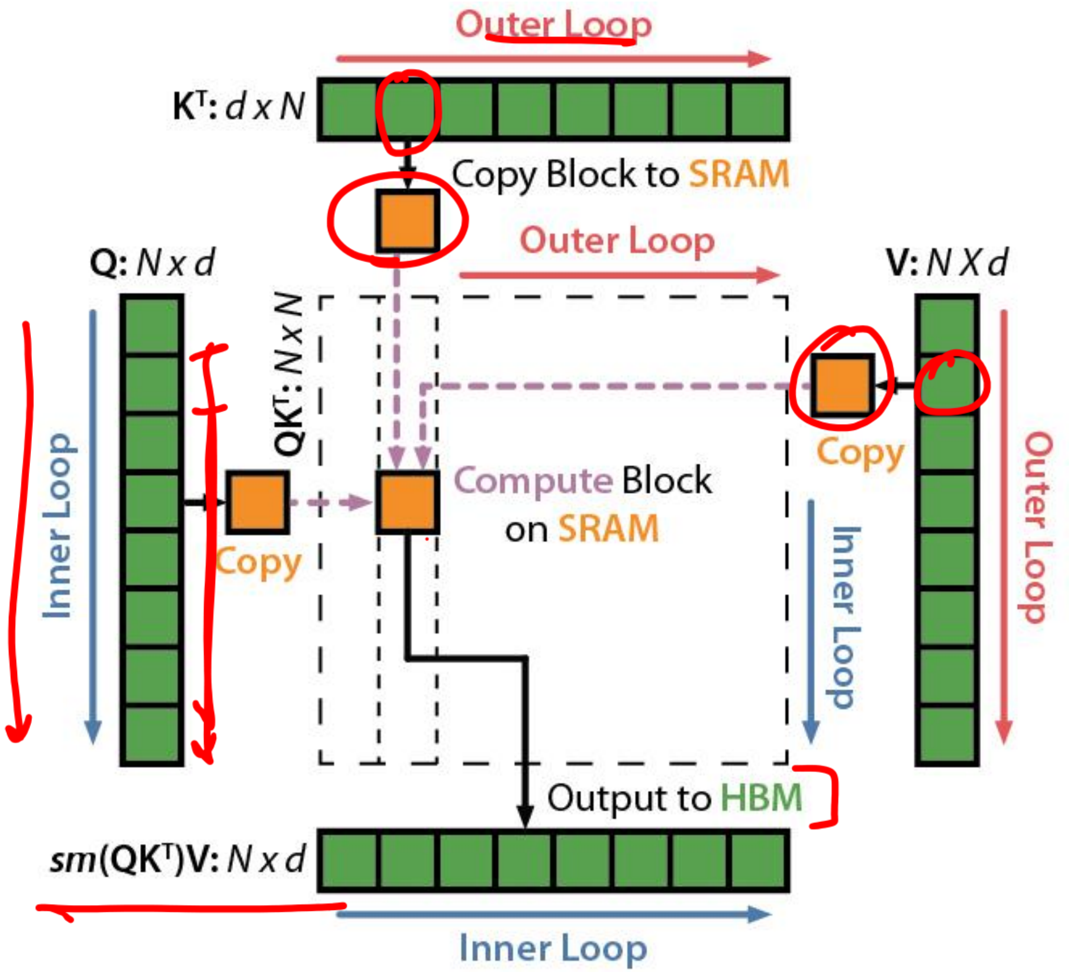
SRAM

$QK^T = S$ softmax(S) = P PV = O

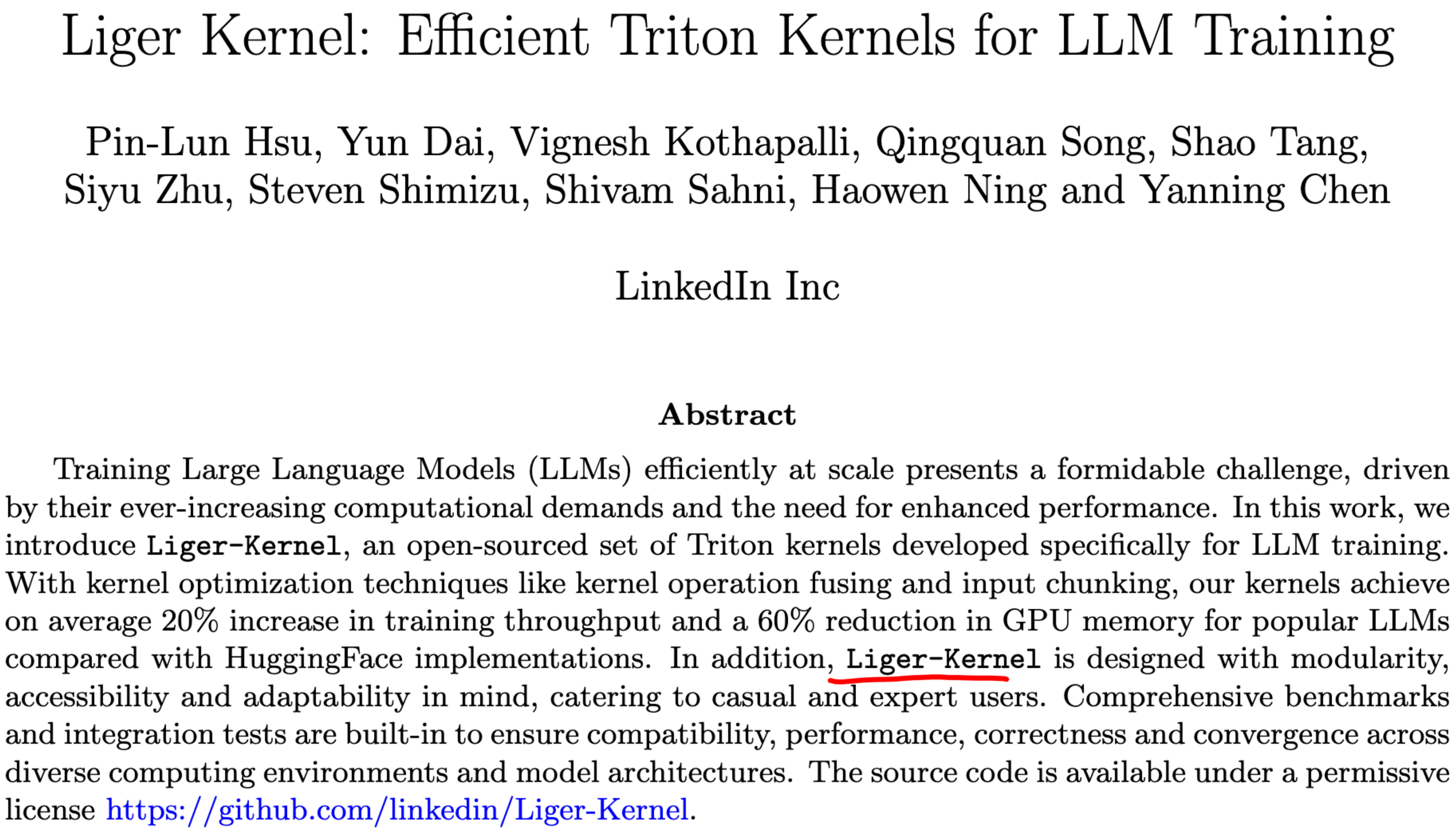
Q,K S S P P,V O

HBM

Q,K,V S P O



What else can we replace by a fused kernel?



Liger Kernel: Efficient Triton Kernels for LLM Training

Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang,
Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning and Yanning Chen

LinkedIn Inc

Abstract

Training Large Language Models (LLMs) efficiently at scale presents a formidable challenge, driven by their ever-increasing computational demands and the need for enhanced performance. In this work, we introduce **Liger-Kernel**, an open-sourced set of Triton kernels developed specifically for LLM training. With kernel optimization techniques like kernel operation fusing and input chunking, our kernels achieve on average 20% increase in training throughput and a 60% reduction in GPU memory for popular LLMs compared with HuggingFace implementations. In addition, **Liger-Kernel** is designed with modularity, accessibility and adaptability in mind, catering to casual and expert users. Comprehensive benchmarks and integration tests are built-in to ensure compatibility, performance, correctness and convergence across diverse computing environments and model architectures. The source code is available under a permissive license https://github.com/linkedin/Liger-Kernel.