

LMTracer: Fine-Grained and Real-Time Performance Profiling for Production LLM Systems

Wei Liu, Yongchao He, Bohan Zhao, Hongyi Wang, Zhenhua Li, Junping Zhao

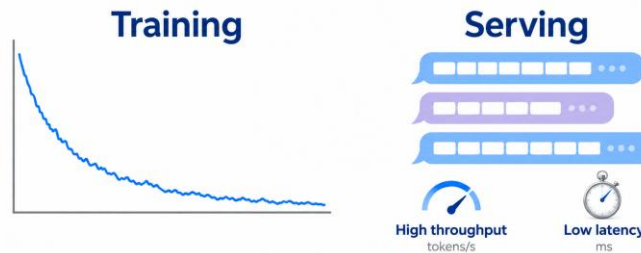
1. Background

□ LLM training and serving are core workloads on the cloud

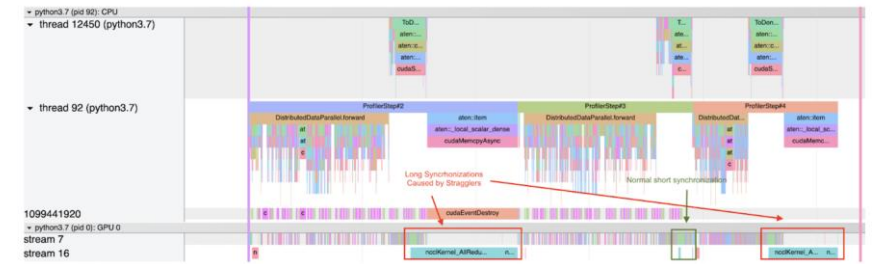
- Training uses **thousands of GPUs** for **weeks or months**
- Serving demands **high throughput** and **low latency**
- Performance bottlenecks **raise costs** and **hurt user experience**



Large-scale GPU infrastructure



Needs for high throughput and low latency



Ensuring QoE through profiling

1. Background

❑ Profiling matters, yet existing tools fall short

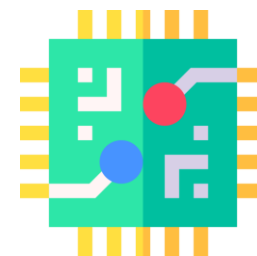
Telemetry-based monitoring



- ❑ Throughput, iteration time, etc.
- ❑ LLM **framework metrics** and **eBPF**
- ❑ Measured mostly on **CPUs**

Lightweight but **coarse-grained**

Kernel-level profiling



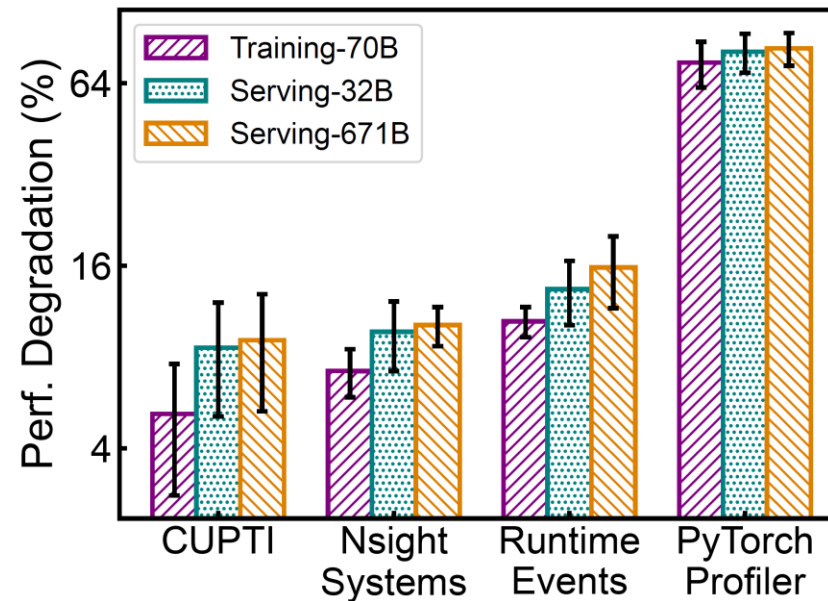
- ❑ Measuring execution times of GPU kernels
- ❑ **Nsight Systems** and **PyTorch Profiler**
- ❑ Measured on **GPUs**

Fine-grained but **intrusive**

2. Motivation

□ Kernel-level profilers impose substantial overhead

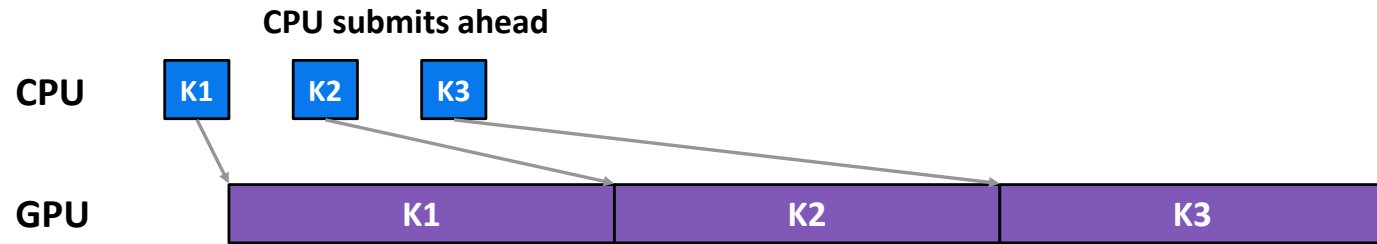
- **Nsight Systems** increases training iteration time by **7.2%**
- **PyTorch Profiler** reduces serving throughput by **83.4%**
- Such overhead limits real-time performance profiling in production



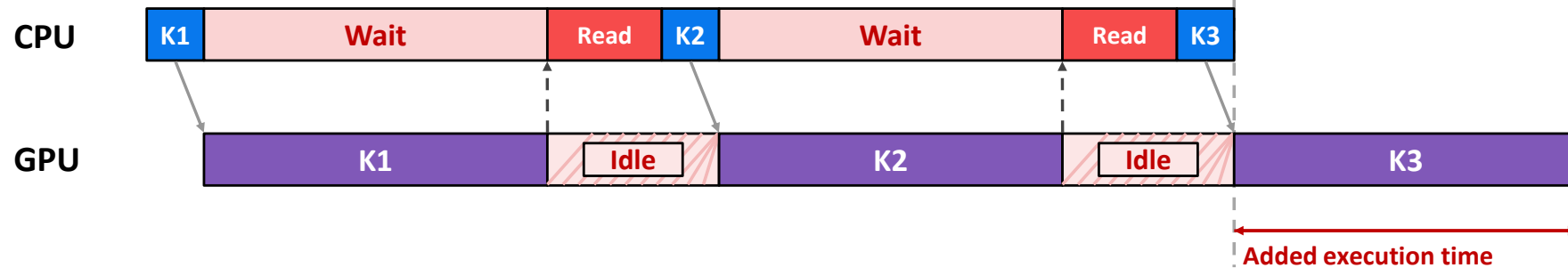
2. Motivation

□ Kernel-level profiling disrupts asynchronous execution

Without profiling



With synchronization-based profiling



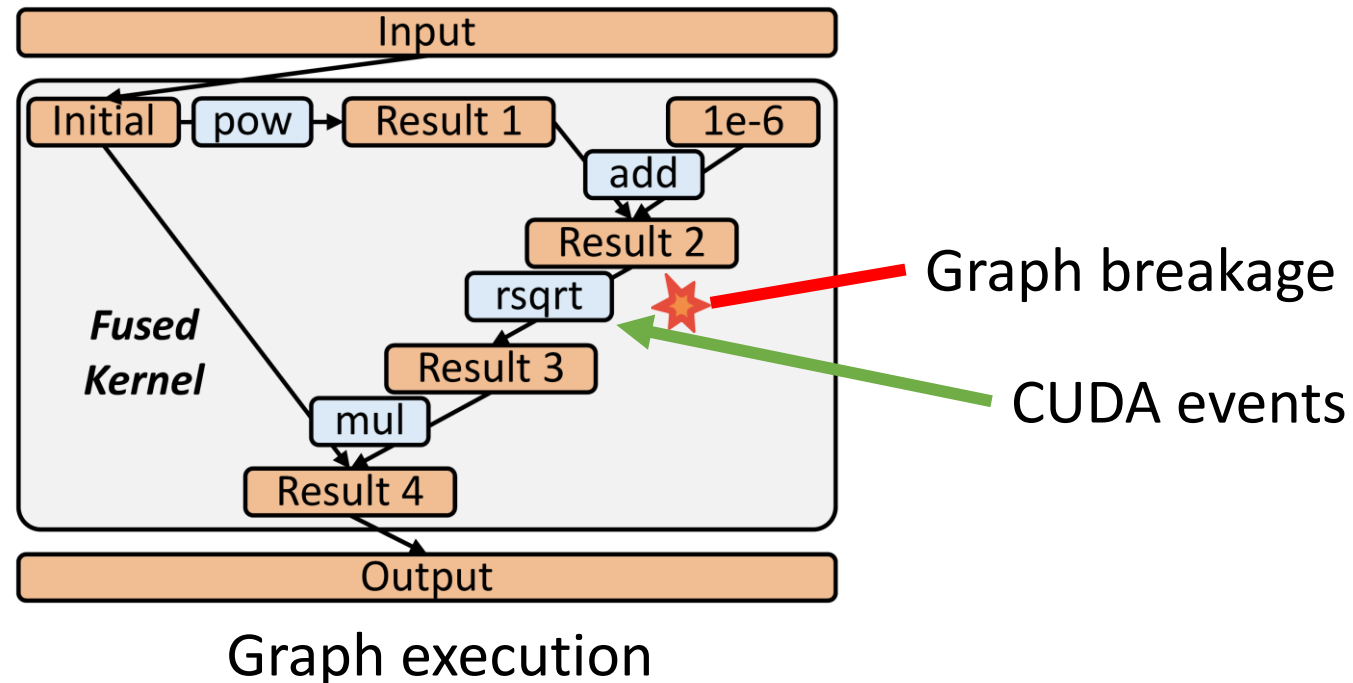
■ CPU blocks: kernel launches

Time

2. Motivation

❑ Kernel-level profilers impose substantial overhead

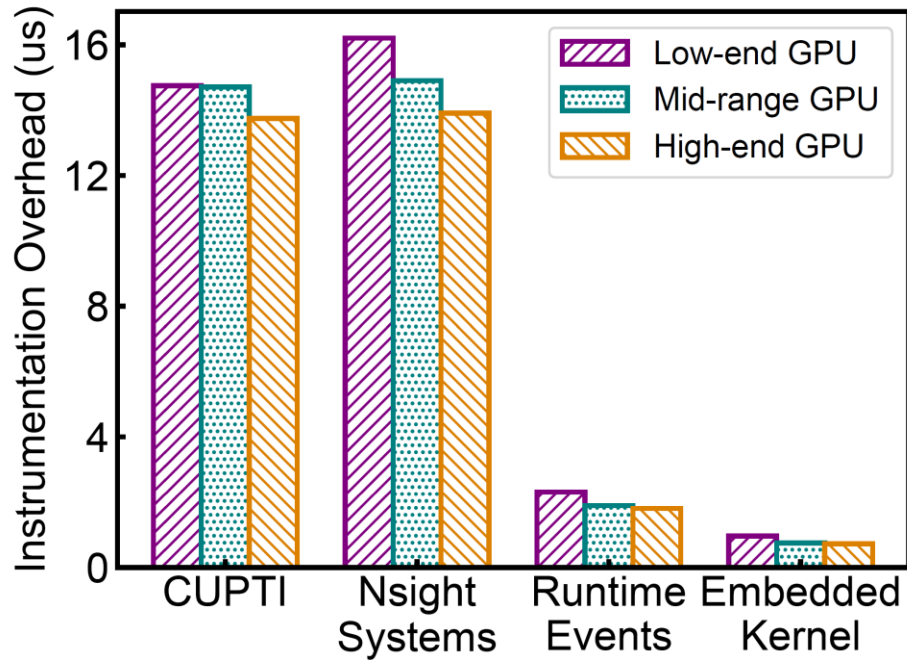
- **CUDA events** can record timestamps asynchronously
- Unsupported profiling operations can **trigger graph breaks**
- Graph breaks **limit kernel fusion** and **introduce extra kernel launches**



2. Motivation

❑ Kernel-level profilers impose substantial overhead

■ Overhead per trace point across profilers



❑ CUPTI and Nsight Systems

~14–16 μs per trace point

❑ Runtime events

Reduce overhead to ~2 μs (but can break graph executions)

❑ Embedded kernels to read GPU clock registers

Achieve ~1 μs or less

2. Motivation

❑ Kernel-level profilers impose substantial overhead

- **Probe kernels embedded in CUDA Graphs** to read GPU clock registers
- Low-overhead because kernels are the basic execution units on GPUs
- By reading the **globaltimer** register for NVIDIA GPUs or invoke the **s_memtime** instruction for AMD GPUs

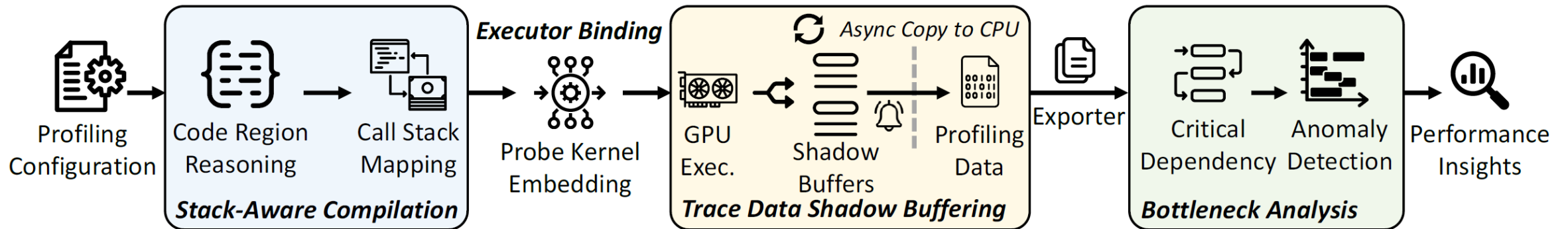
➤ Opportunity

Implement profiling logic as **lightweight GPU kernels**

3. Design of LMTracer

LMTracer: Graph-embedded probing

- Implement profiling logic as **lightweight probe kernels**
- Preserve **kernel fusion** when embedding probes
- Stream trace data to CPUs **asynchronously**



Architectural overview of LMTracer

3. Design of LMTracer

LMTracer: Profiling configuration

```
llm_engine: sglang
probes:
  # LLM modules
  - phase: attention
    module: sglang.srt.models.gpt_oss.GptOssAttention
  # Functions or even code lines
  - phase: norm
    function: sglang.srt.layers.layernorm.RMSNorm.
      forward_cuda
  # Optional: Flexibility for code-line profiling
  line_start: 109
  line_end: 126
exporters:
  # Exporter 1: Save binary trace data to files
  - module: lmtracer.core.exporter.
    FileByteDataExporter
    params:
      output_dir: /tmp/lmtracer_exports
  # Exporter 2: Export metrics to Prometheus
  - module: lmtracer.core.exporter.
    SGLangMetricPrometheusExporter
    params: {}
```

Select the LLM framework/engines:
SGLang, vLLM, Megatron-LM, or others

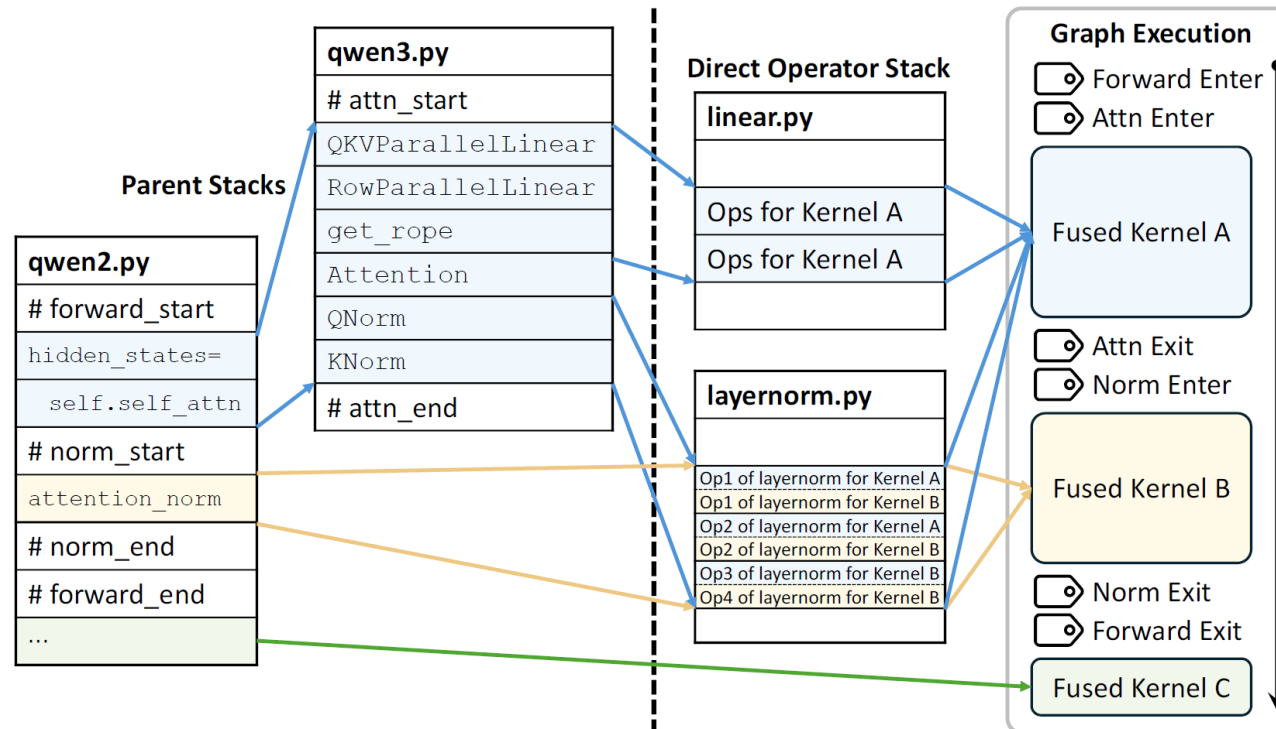
Specify what to profile:
LLM modules/layers, functions, code regions

Choose where to export:
Trace files or monitoring systems

3. Design of LMTracer

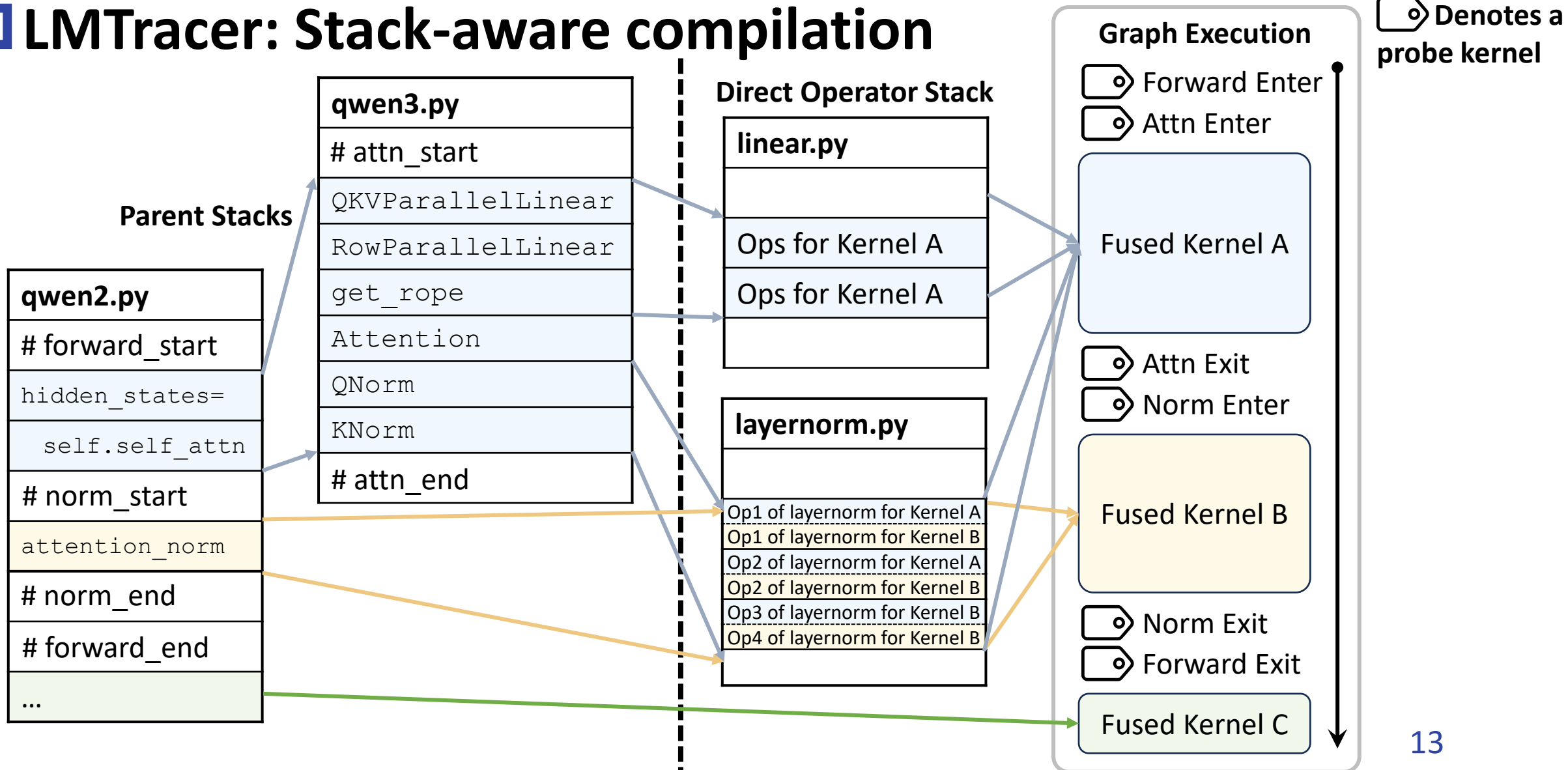
□ LMTracer: Stack-aware compilation

- Mapping user-defined **trace points** to **probe kernels**
- Preserve kernel fusion and graph execution



3. Design of LMTracer

LMTracer: Stack-aware compilation

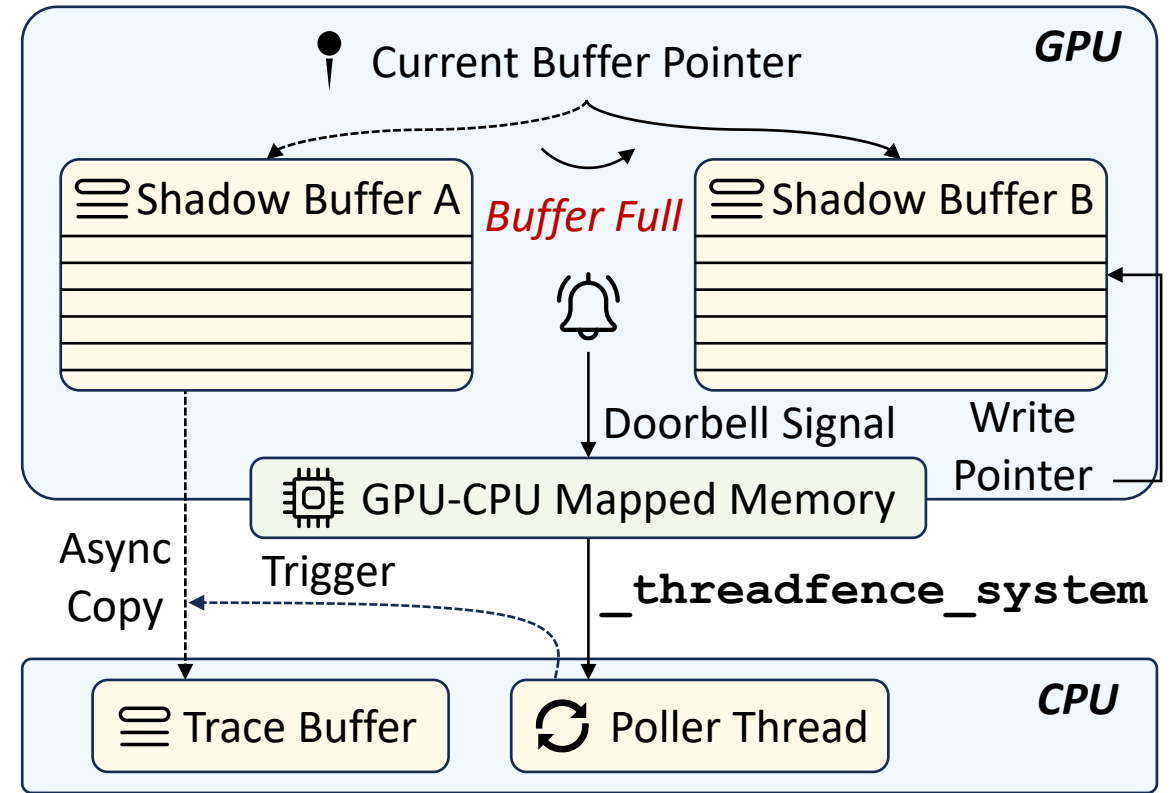


3. Design of LMTracer

LMTracer: Doorbell-assisted shadow buffering

To eliminate GPU-CPU synchronization:

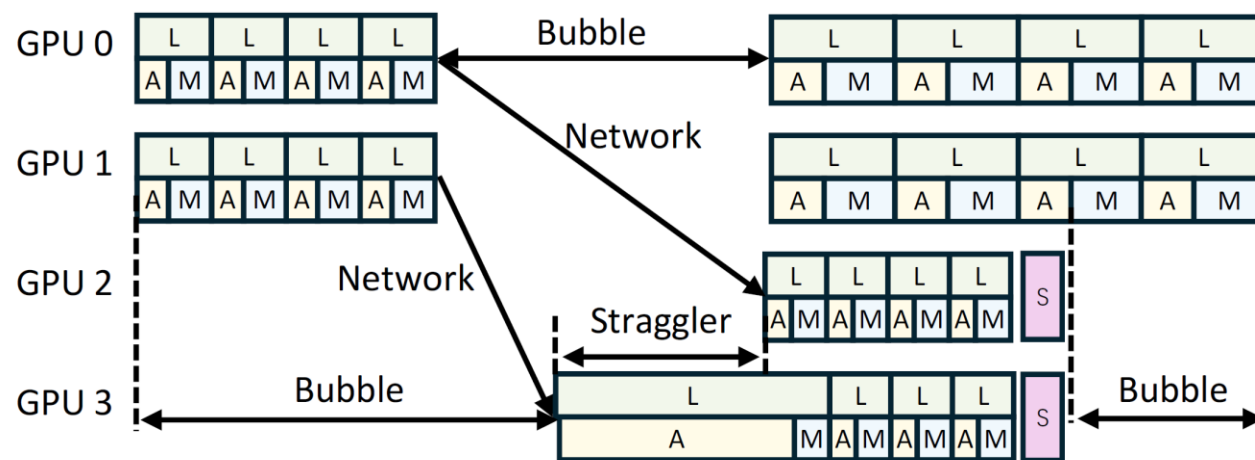
- Probe kernels write traces into **GPU-side shadow buffers**
- A **doorbell** signals the CPU when a buffer is ready
- Asynchronous** transfers overlap with GPU execution



3. Design of LMTracer

LMTracer: Bottleneck analysis

- Reconstruct **execution pipelines** from gathered dependencies
- Compare corresponding operations across GPUs
- Identify **GPU bubbles**, **stragglers**, and **communication delays**



$$\text{BubbleFraction} = 1 - \frac{\sum_{i=1}^N d_i}{t_N - t_1 + d_N}$$

$$\text{StragglingFraction} = \frac{\max_{g \in G} s_g - \min_{g \in G} s_g}{\max_{g \in G} e_g - \min_{g \in G} s_g}$$

4. Evaluation

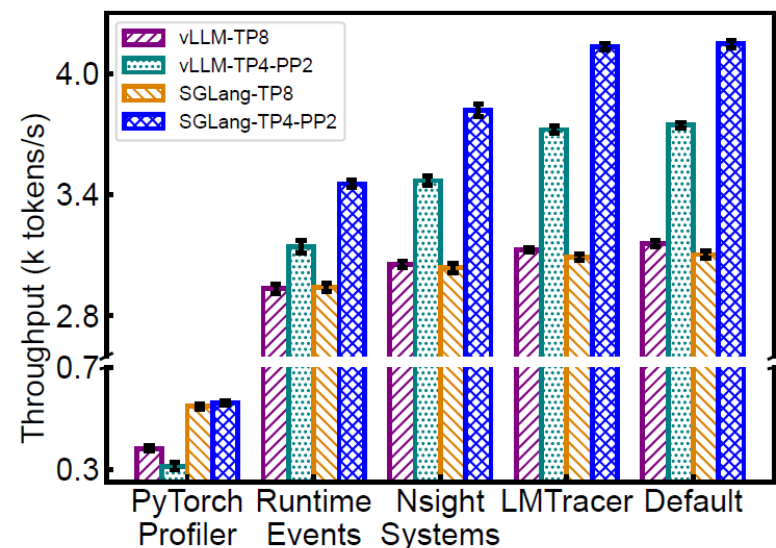
□ Evaluation setup

- Production deployment over **eight months**
- **Serving:** Qwen3-32B, DeepSeek-V3, and GPT-OSS-120B
- **Training:** DeepSeek-V3 **fine-tuning on 256 GPUs**
- **LLM Frameworks:** vLLM, SGLang, and Megatron-LM
- **Baselines:** Nsight Systems, Runtime Events, and PyTorch Profiler

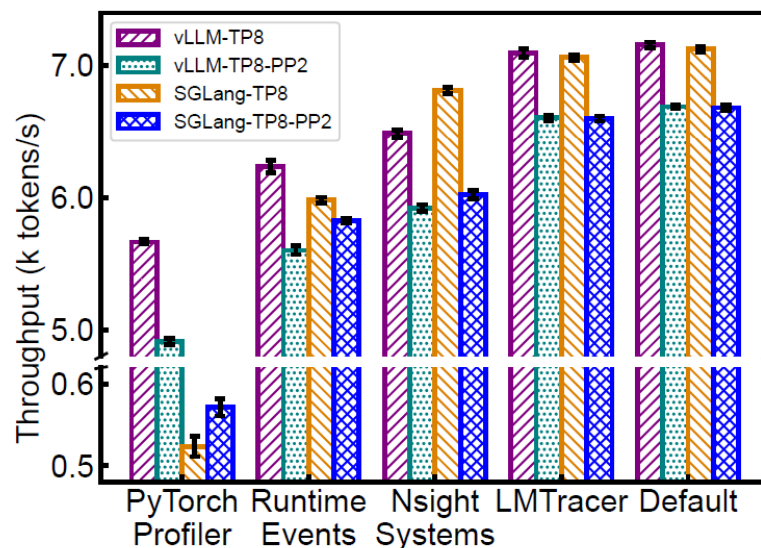
4. Evaluation

LMTracer introduces negligible profiling overhead

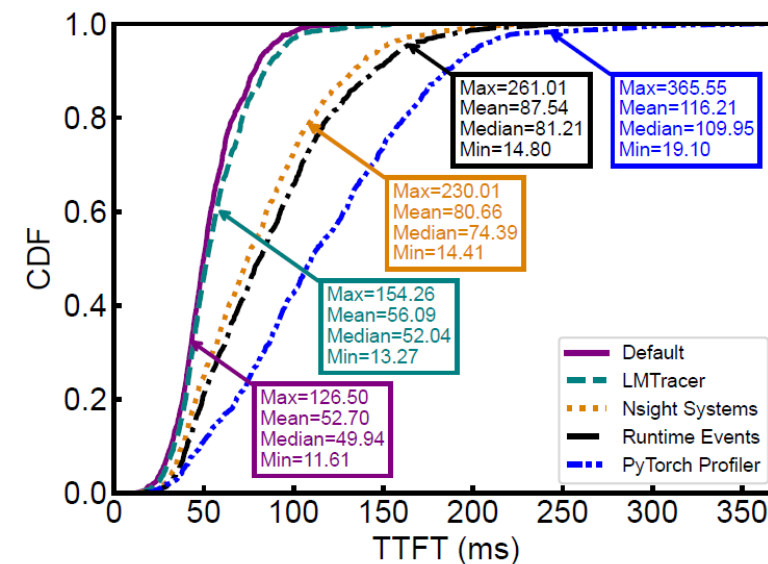
- Only **0.9%~1.5%** throughput degradation across tested configurations
- Consistently **lower overhead** than existing profilers



Qwen3-32B Inference



DeepSeek V3 Inference

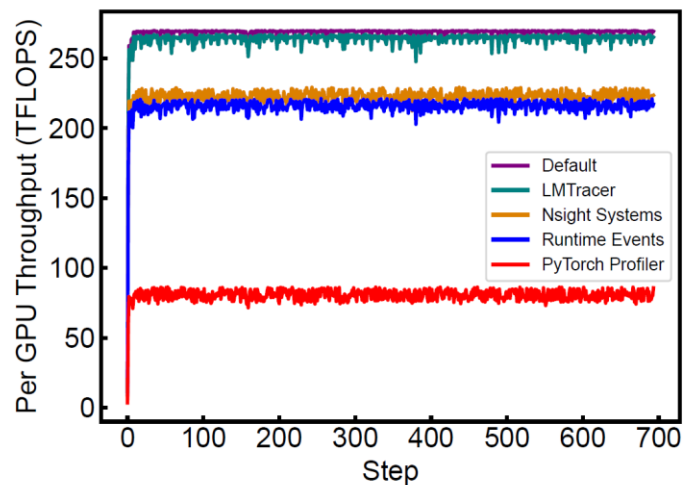


GPT-OSS-120B Inference

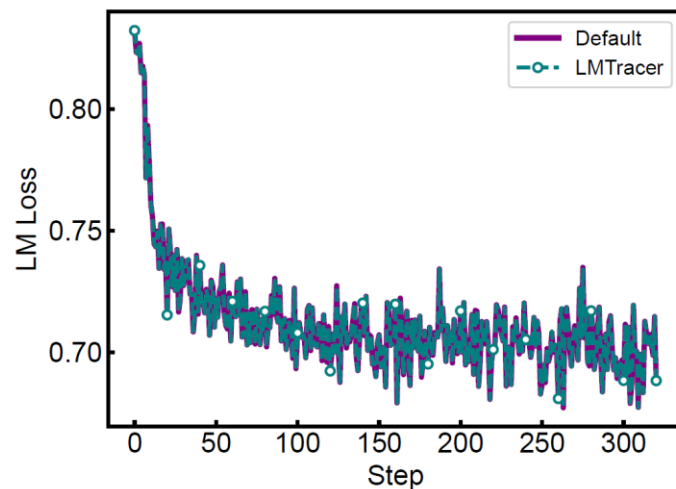
4. Evaluation

□ LMTracer introduces negligible profiling overhead

- Only **0.9%~1.5%** throughput degradation across tested configurations
- Consistently **lower overhead** than existing profilers



DeepSeek-V3 Fine Tuning (Training)



Consistent Language Model Loss

4. Evaluation

□ Ablation study: contribution of each design

- Each design choice contributes to low profiling overhead

Throughput degradation (%)

Configuration	Serving	Training
LMTracer	0.9%	1.5%
Without probe merging	2.9%	3.0%
Without dual buffers	12.8%	5.7%
Without doorbell signals	16.2%	18.2%
Without asynchronous copy	13.3%	8.7%

Serving: Qwen3-32B on low-end GPUs

Training: DeepSeek-V3 SFT on high-end GPUs

4. Evaluation

□ Profiling fidelity: linking kernels to user code

- Map compiled GPU kernels **back to user-defined code regions**
- Distinguish model operations hidden behind **generic kernel names**

Baseline: compiled kernel names

triton_red_fused__to_copy_add_
mean_mul_pow_sqrt_0

ampere_bf16_s1688gemm_bf16_
128x64_sliced1x2_ldg8_f2f_tn

triton_poi_fused_mul_silu_1

ampere_bf16_s1688gemm_bf16_
128x64_sliced1x2_ldg8_f2f_tn

LMTracer: code locations and semantics

RMS normalization

layernorm.py:127-148 (forward_native)

MLP up projection

utils.py:70 (default_unquantized_gemm)

SiLU activation

activation.py:77 (forward_native)

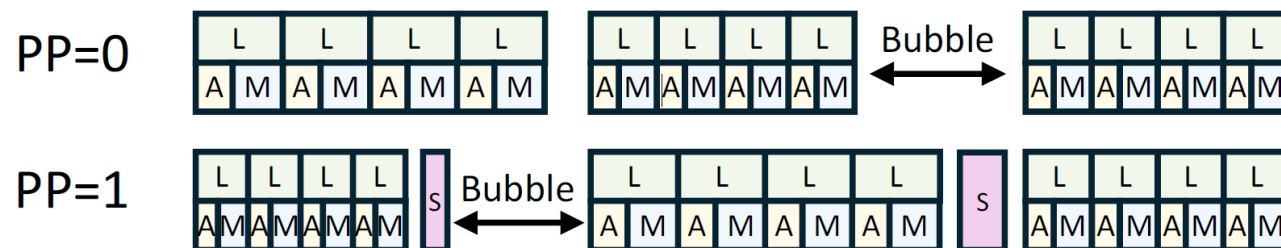
MLP down projection

utils.py:70 (default_unquantized_gemm)

4. Evaluation

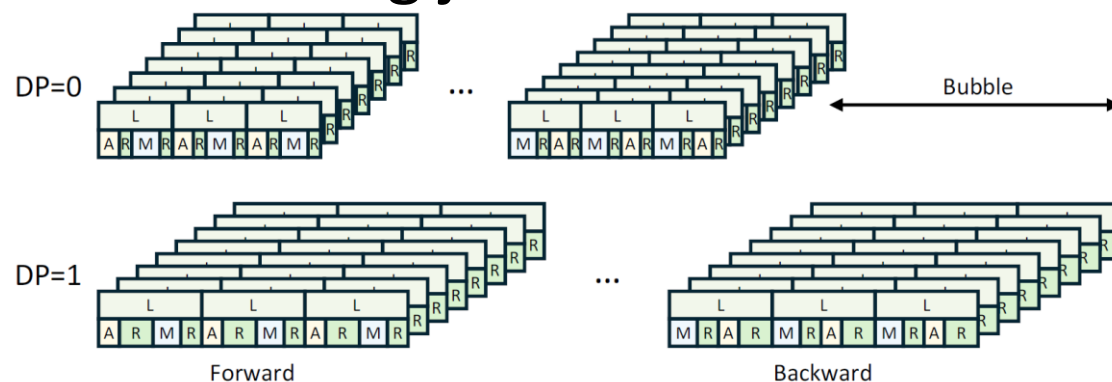
□ Case studies: diagnosing production bottlenecks

- Unbalanced scheduling in serving: uneven request execution times create pipeline bubbles



High bubble fraction

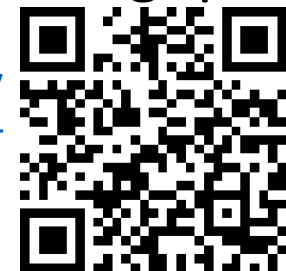
- NVLink issues in training: slow AllReduce operations in DP 1 stall the entire training job



High straggling fraction

5. Conclusion

- We identify the challenges of fine-grained, real-time profiling in production LLM systems, where existing tools either provide **limited execution details** or incur **substantial overhead** from CPU–GPU synchronization.
- We present LMTracer, a performance profiling framework that combines ***graph-embedded probing*** with doorbell-assisted shadow buffering to capture GPU execution at user-defined code regions with minimal overhead.
- We have deployed LMTracer in production for eight months, **identifying 15 performance issues across LLM training and serving**. After fixing these issues, we observed 21.2% lower serving latency and 24.5% higher throughput.
- Code and data are released at <https://llm-profiling.github.io/>



Thanks!
Q & A