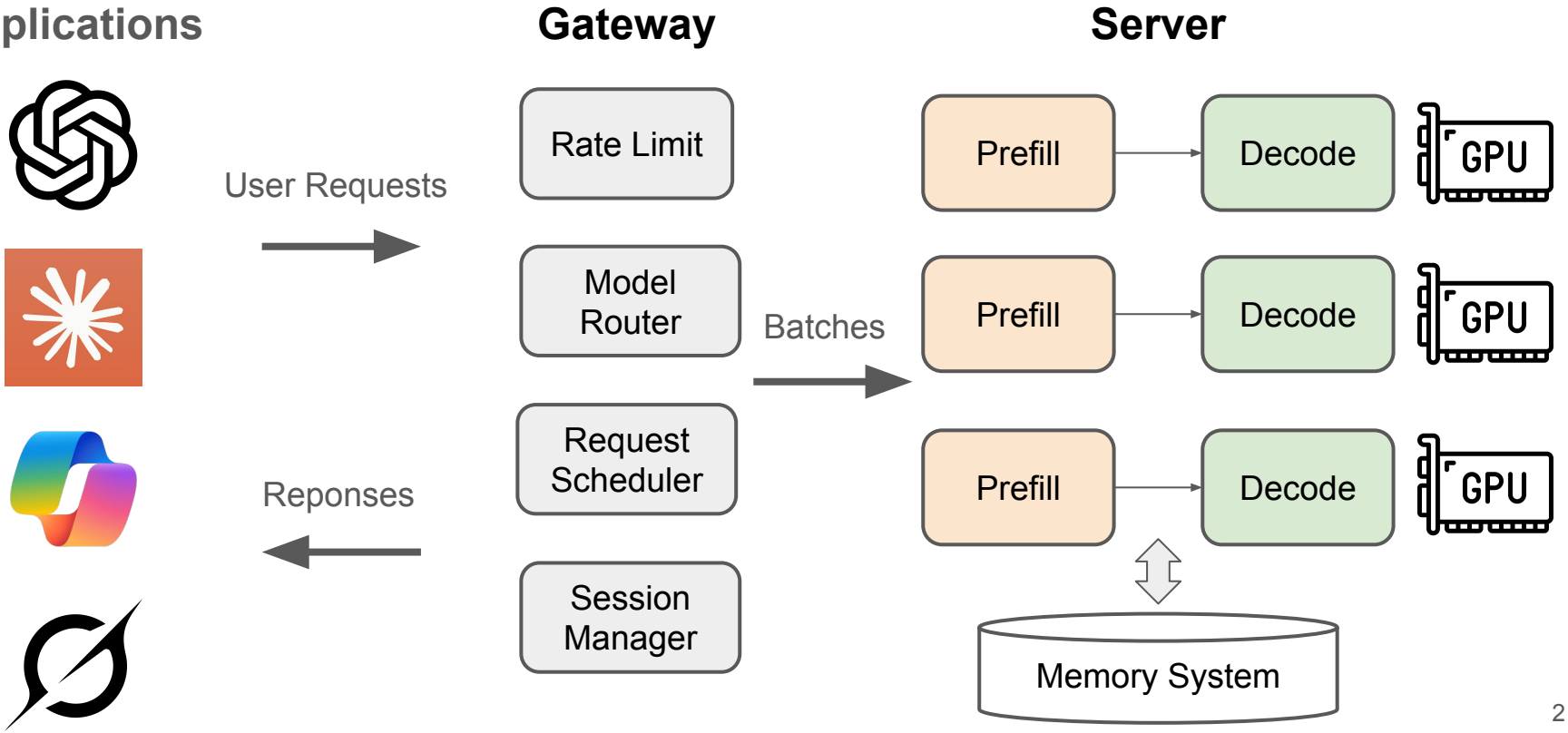
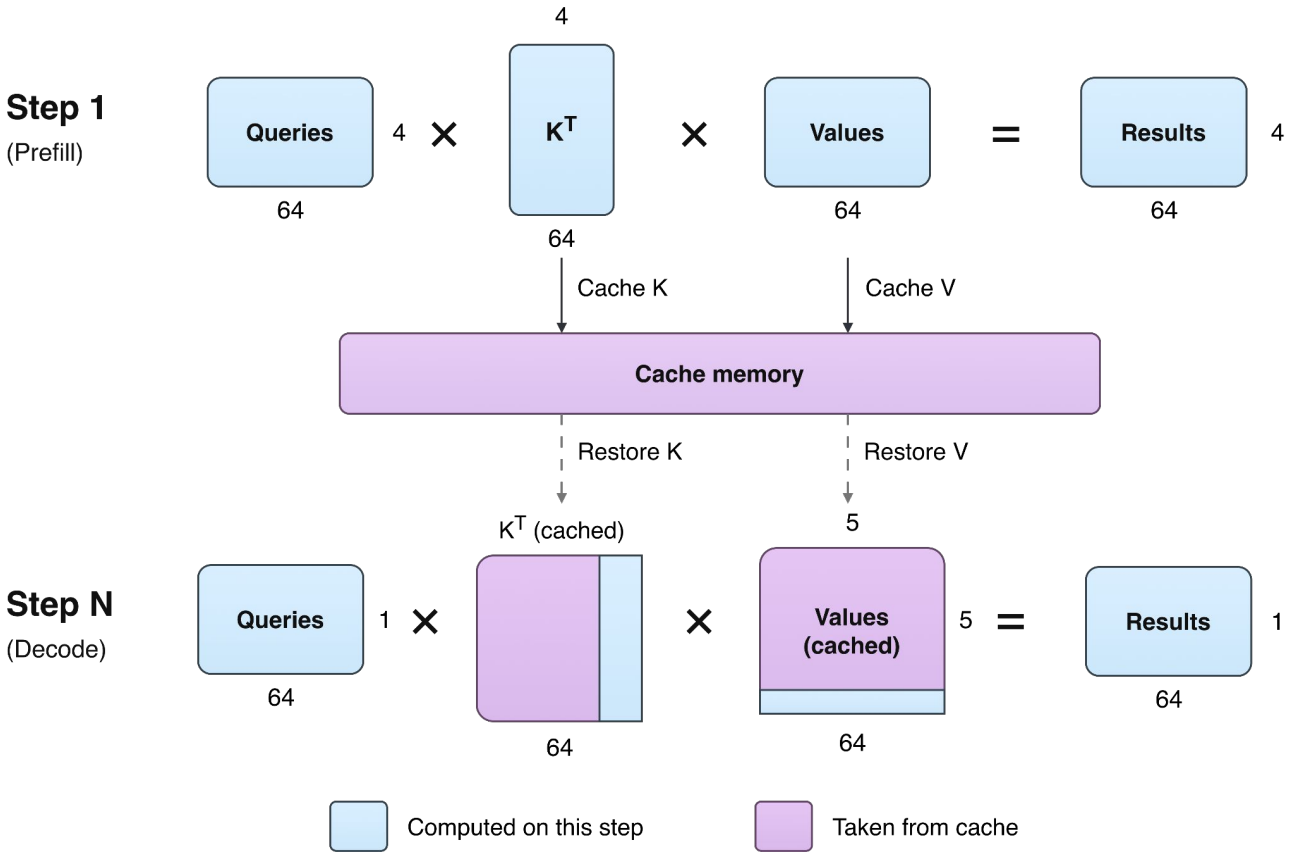
Low Overhead, Continuous Profiling for AI Services

Keren Zhou, Tianle Zhong, Hao Wu, HyoukJoong Lee
Jade Nie, Sahil Patel, Ruoming Pang, Philippe Tillet

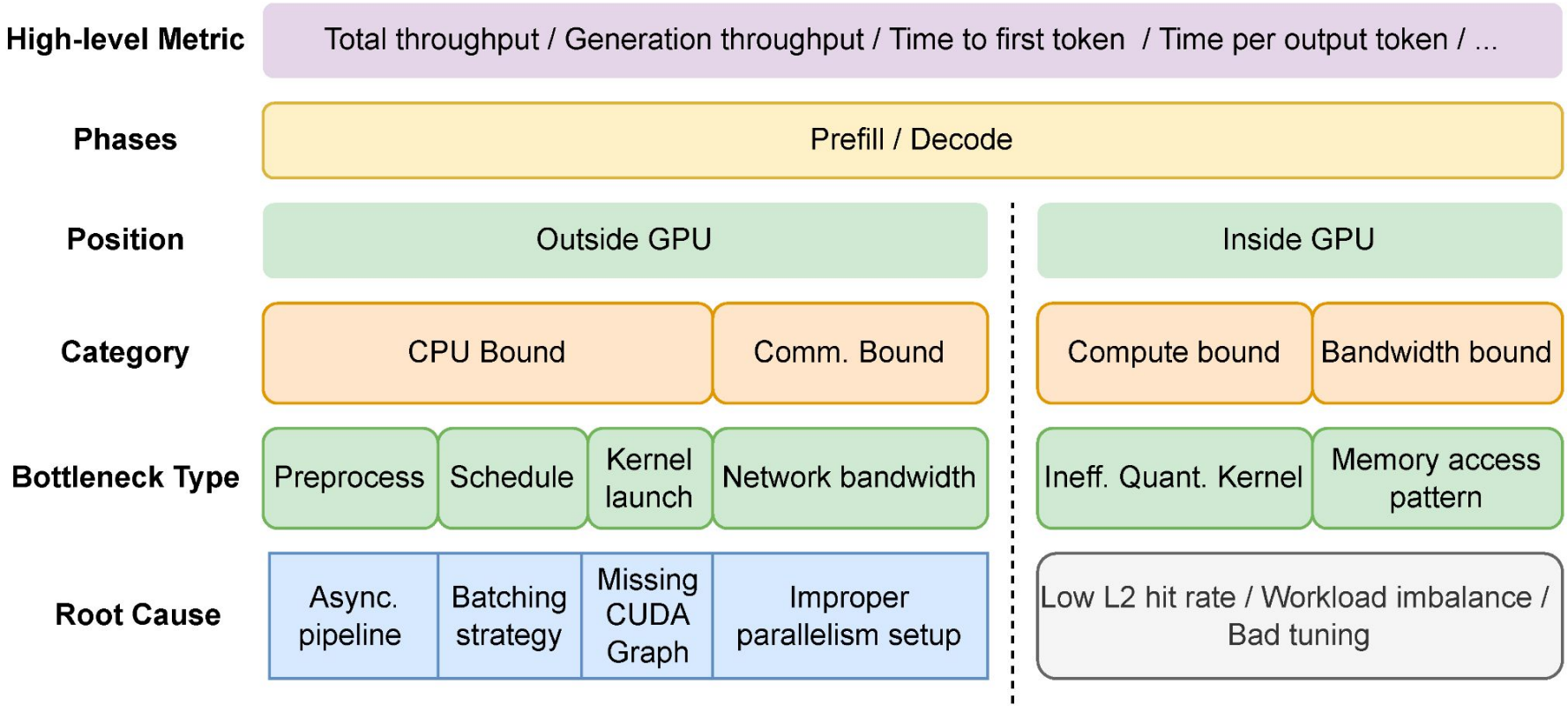
LLM Serving Systems



Prefill *vs* Decode



Decomposition of Performance Challenges



Continuous Profiling

Background

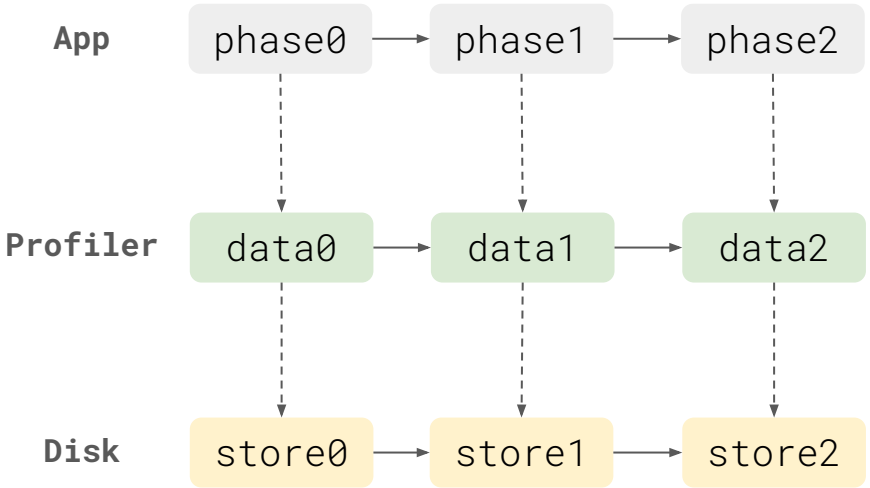
- LLM serving and RL systems heavily rely on GPUs and typically run continuously for days or even months
- Production workloads are inherently complex and dynamic
  - Dynamic shapes lead to varying kernel launch configurations and resource usage
  - Tail latency issues (e.g., *KV cache misses*) introduce unpredictable performance spikes
  - RL pipelines interleave rollout and training, creating varied execution characteristics
  - Multi-GPU behaviors (e.g., *synchronization, imbalance, contention*) are difficult to reproduce in isolated or standalone environments

Existing Tools

- Cluster monitoring tools (e.g., *nvidia-smi*)
  - Sample GPU state to report coarse-grained utilization metrics
  - Very low overhead (<1%)
  - No visibility into per-kernel behavior or proximity to peak FLOPs
- Traditional profilers (e.g., *nsys*)
  - Provide per-kernel execution timelines and detailed traces
  - Non-trivial runtime, memory, and storage overhead (>20%)
  - Not suitable for long-running production workloads (>1 hour)

Periodic Profiling

- Structure execution into explicit application-defined phases
- Aggregate profiling data per phase within the profiler
- Two operating modes
  - Active data fetching
  - Background data flushing



Active Mode

- The user explicitly controls synchronization and decides when to collect and reset profiling data
- Suitable for fine-grained analysis and controlled experiments

Asynchronous

```
if proton.data.is_phase_complete(session_id, phase_id):
    data_phase = proton.data.get_json(session_id, phase_id)
    proton.data.clear(session_id, phase_id)
```

Synchronous

```
proton.deactivate(session_id, flushing=True)
data = proton.data.get_json(session_id)
```

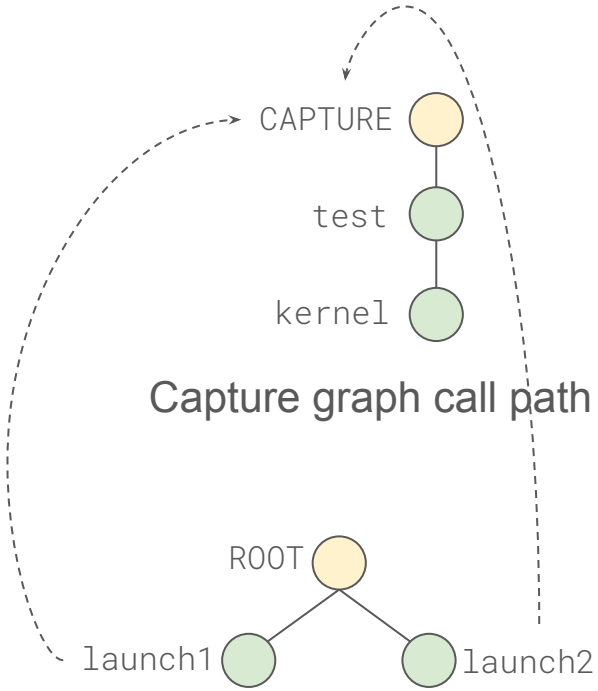
Background Mode

- The profiler autonomously collects, flushes, and recycles profiling data
- Designed for long-running production services with minimal intervention
- Ensures bounded memory usage without manual cleanup
- `proton.start(..., mode="periodic_flushing:format=...")`
  - `"hatchet"`
  - `"hatchet_msgpack"`
  - `"chrome_trace"`

Key Features: Cuda Graph Profiling

- Cuda graphs are widely used in inference (or even training) to reduce CPU API invocation overhead
  - Small GPU kernels: 10us/kernel
  - API invocation: 1us/kernel
- Instead of launching individual kernels, we launch a graph with a single CPU API invocation consisting of thousands of kernels
- How do we attribute each kernel back to the point where are captured?

Key Features: Cuda Graph Profiling




```
proton.start("profile")

def fn():
  with proton.scope("test"):
    kernel()

g = torch.cuda.CUDAGraph()
with torch.cuda.graph(g):
  fn()

with proton.scope("launch1"):
  g.replay()

with proton.scope("launch2"):
  g.replay()
```

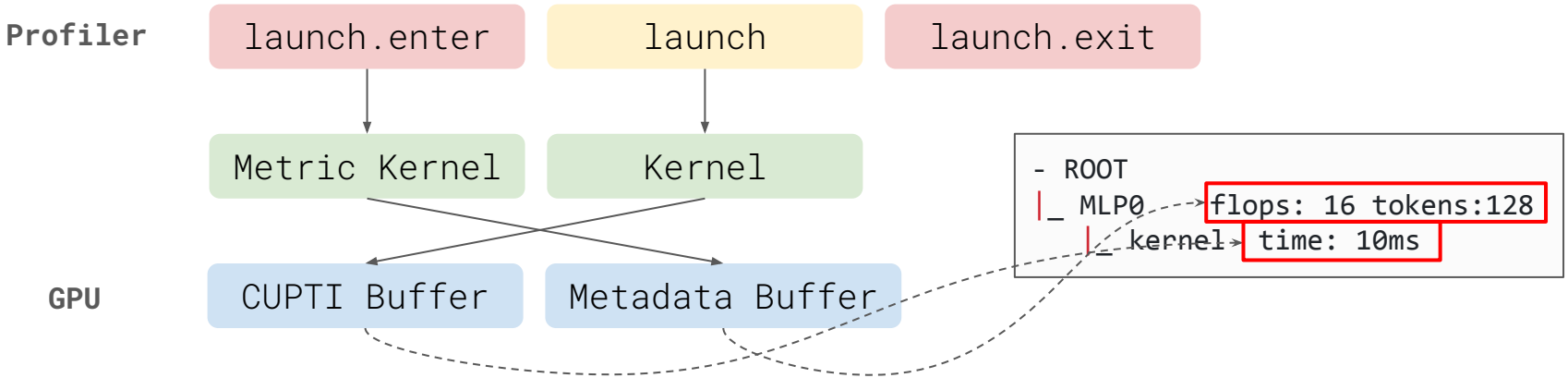
```
- ROOT
|_ launch1
  |_ test
    |_ kernel
|_ launch2
  |_ test
    |_ kernel
```

Key Features: Metadata Kernel Profiling

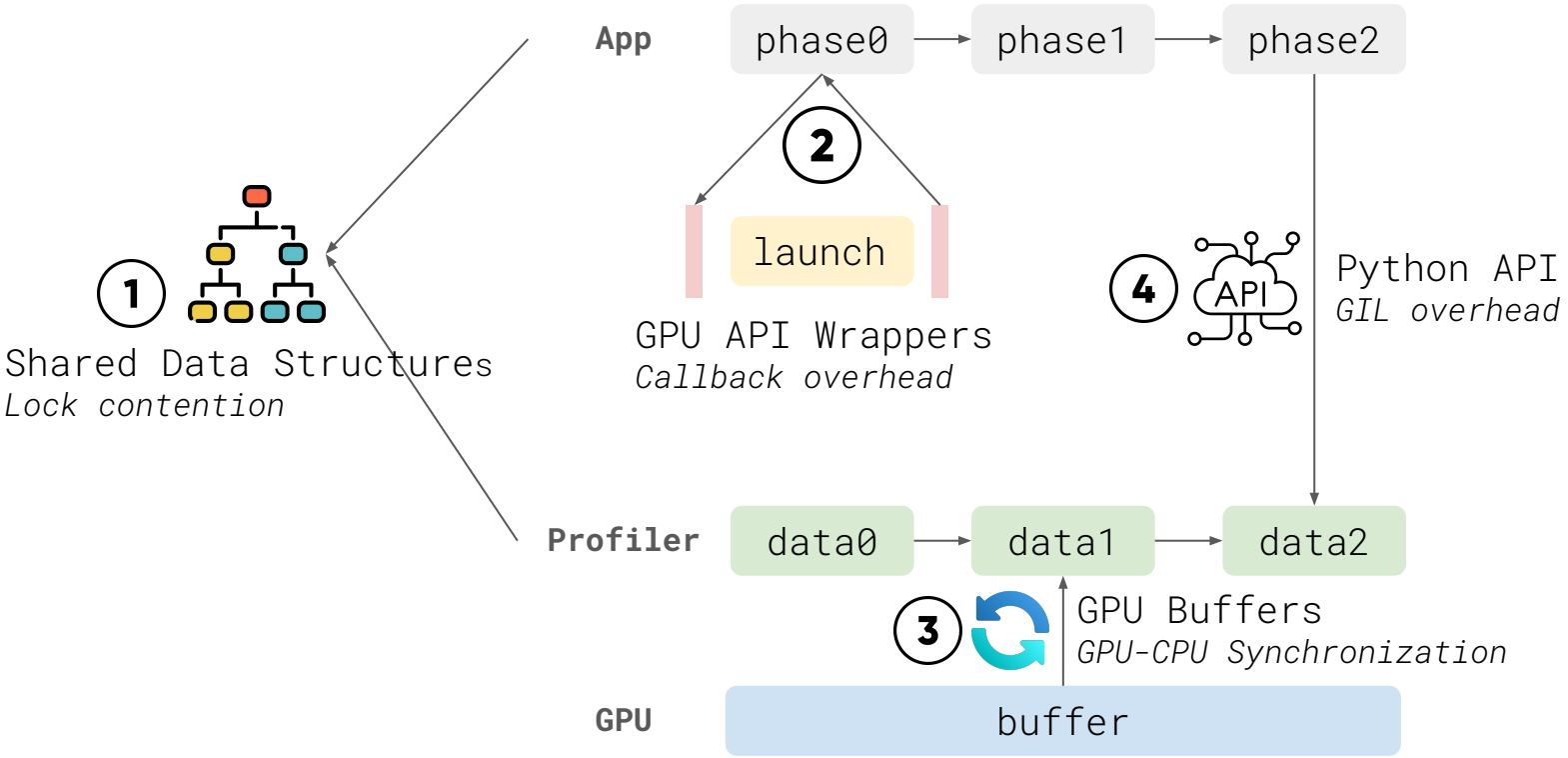
- Metadata (e.g., Tokens) is computed on the CPU
- We cannot copy data from the CPU to the GPU in cudagraph kernels
    - Unless it’s from the pinned memory but not often used
- How do we store metadata on the GPU
    - And keep accumulating metadata without synchronizing the CPU and the GPU?

Key Features: Metadata Kernel Profiling

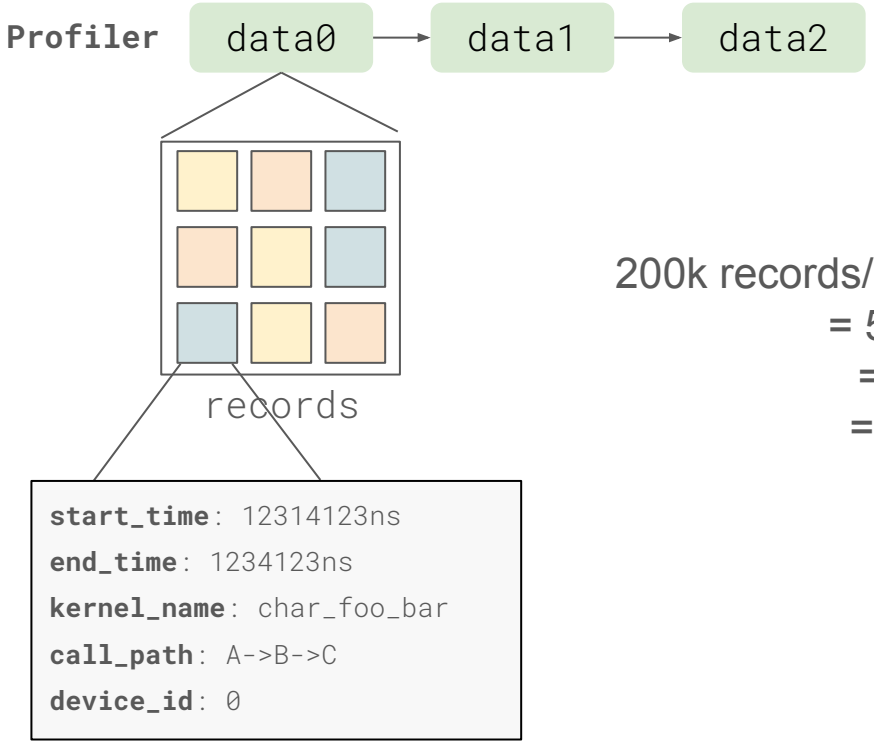
```
def fn():
    with proton.scope("MLP0", metrics={"tokens": 128, "flops": a.sum() * 5}):
        kernel()

with proton.scope("launch"):
    g.replay()
```



Key Challenges: Runtime Overhead



Key Challenges: Memory Overhead



200k records/sec x 256 bytes/record
= 51.2MB/sec
= 3GB/min
= 180GB/hr

Key Challenges: Postmortem Overhead

Disk `store0` → `store1` → `store2`

Parse, analyze, render kernels every minute

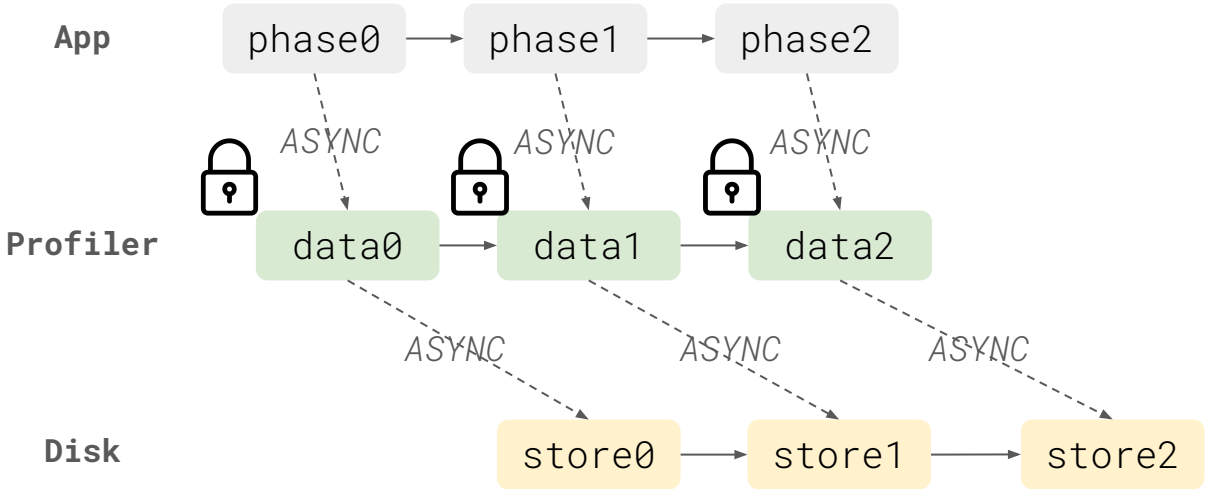
Visualizer

Optimizations

- Runtime
  - Phase separation
  - Synchronization-free buffer copy
  - Launch-order based lock-free metric update
  - Copy-on-write correlation
- Memory
  - Virtual phase
  - User defined records
    - Implemented by NVIDIA collaborators
- Storage
  - Online call path/metric aggregation

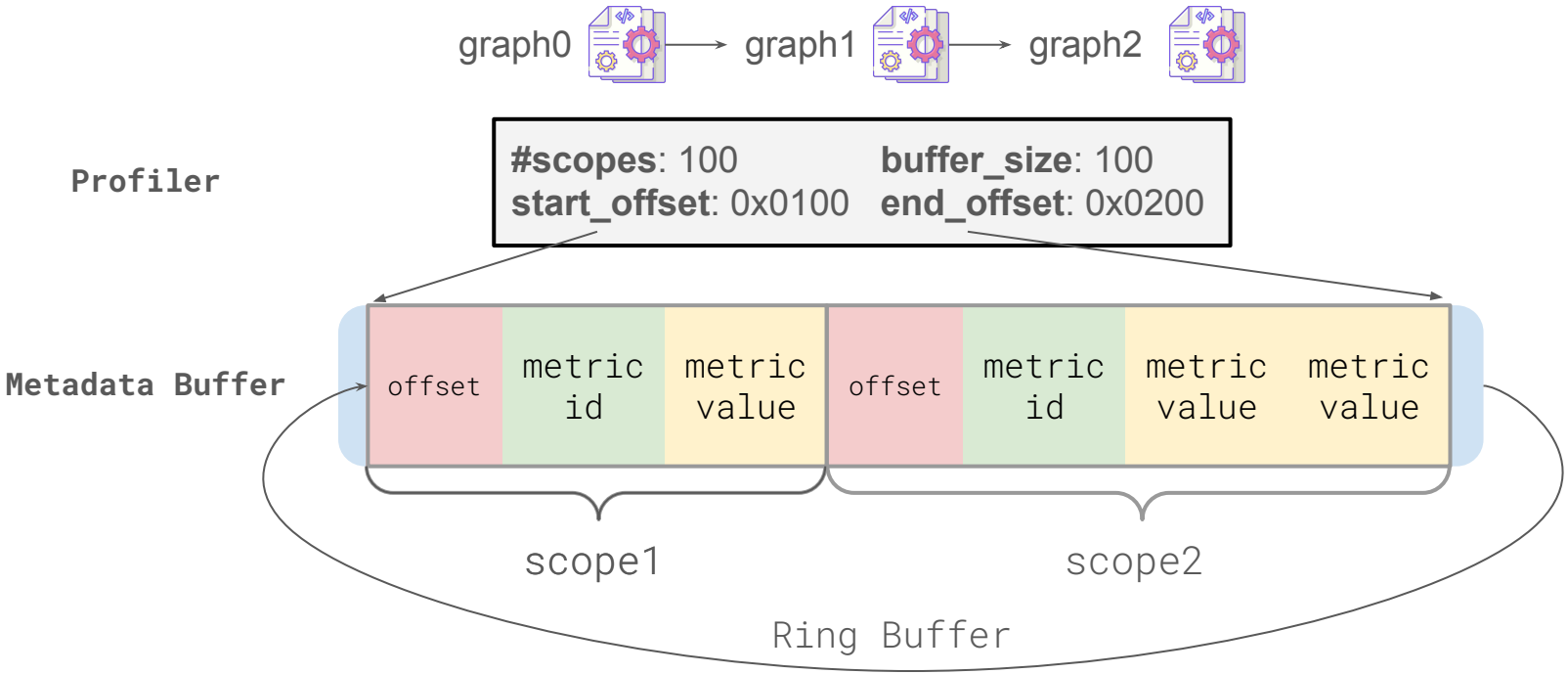
Runtime: Phase Separation and Pipelining

- Asynchronous data collection and disk writing
- Fine-grained locks on each phase
- Avoid disk writing and data collection on the same phase

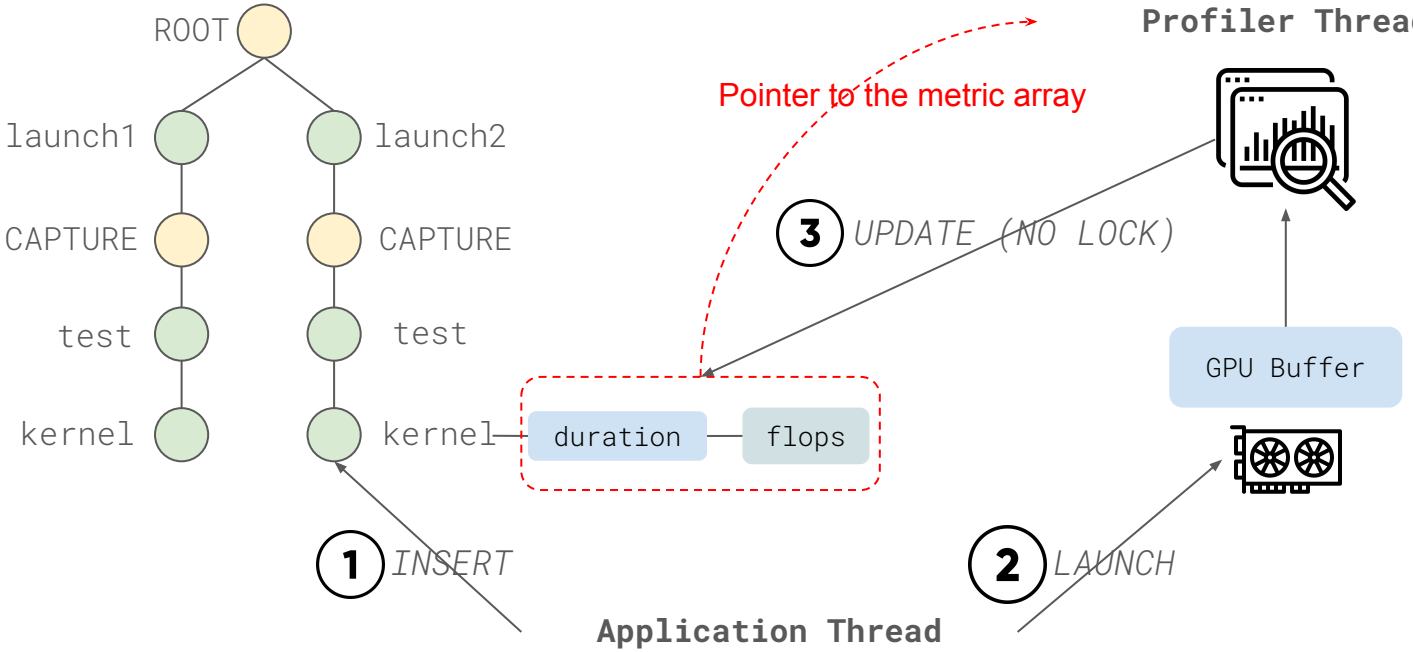


Runtime: Synchronization-free Buffer Copy

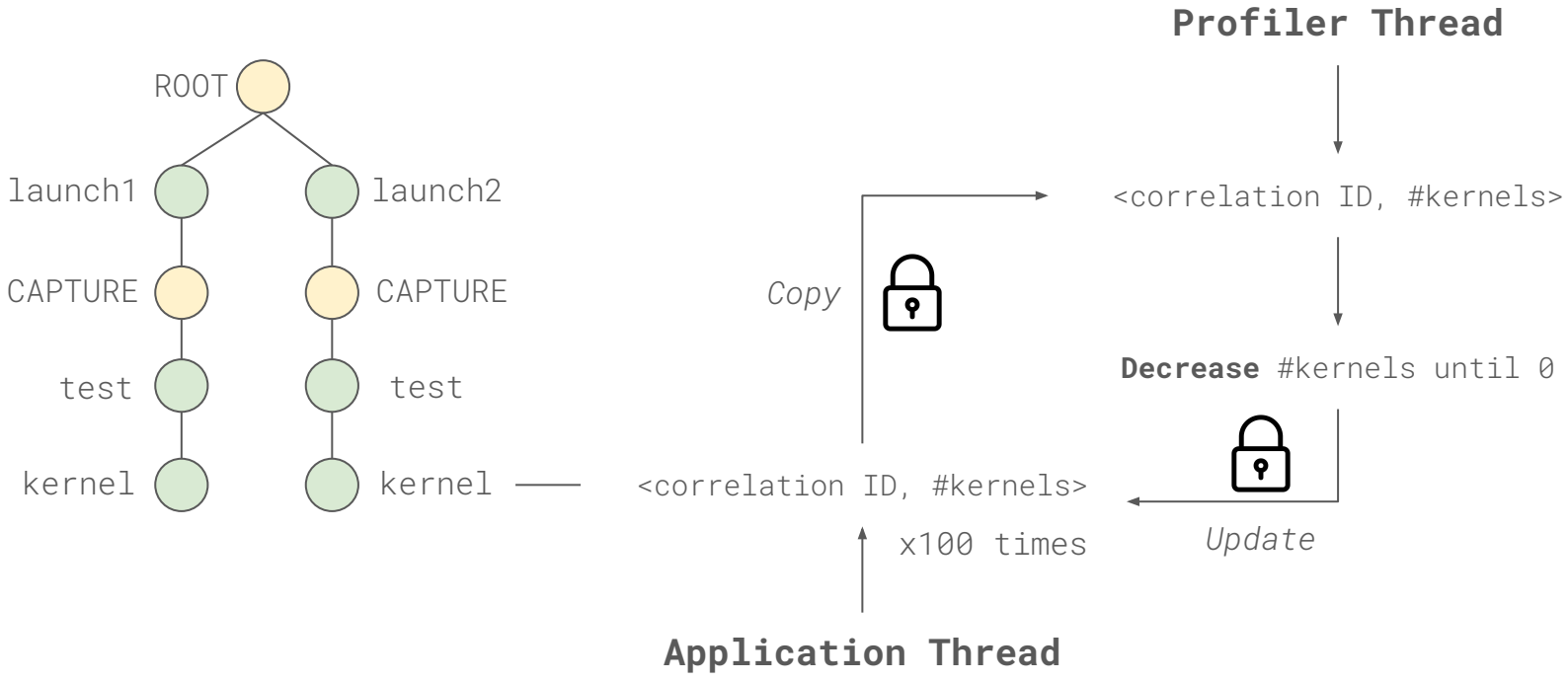
- Collecting metadata without trigger CPU-GPU flushing



Runtime: Launch-order Based Lock-free Metric Update

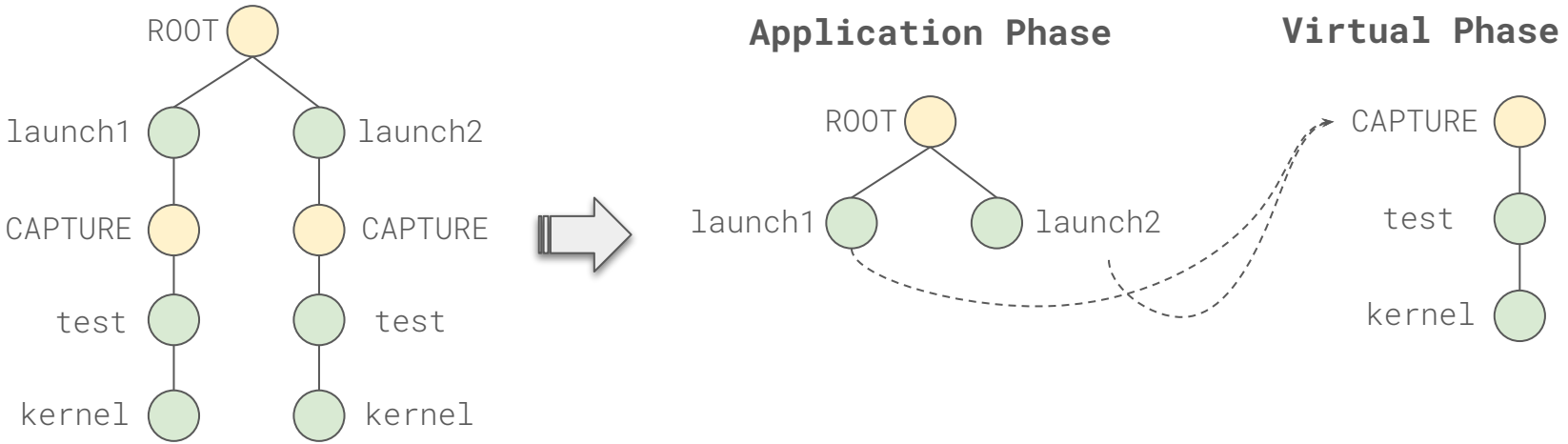


Runtime: Copy-on-Write Correlation



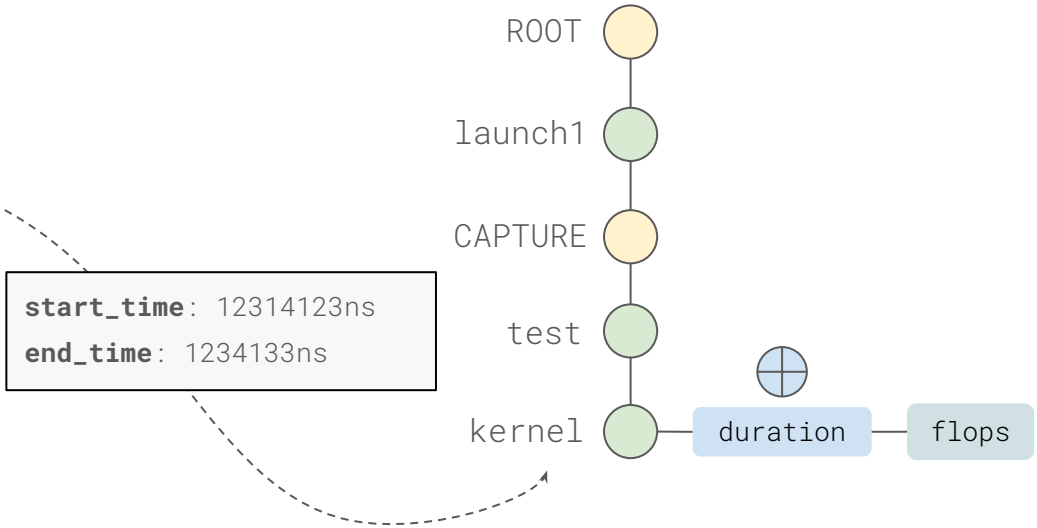
Memory: Virtual Phase

- Naive solution: replicate graph call paths at launch nodes
  - High overhead at each graph launch
- Improved solution: link launch nodes with a virtual phase
  - Reduce memory and launch overhead



Storage: Online Aggregation

```
proton.start("profile")

def fn():
  with proton.scope("test"):
    for i in range(100):
      kernel()

g = torch.cuda.CUDAGraph()
with torch.cuda.graph(g):
  fn()

with proton.scope("launch1"):
  g.replay()
```



Experiments

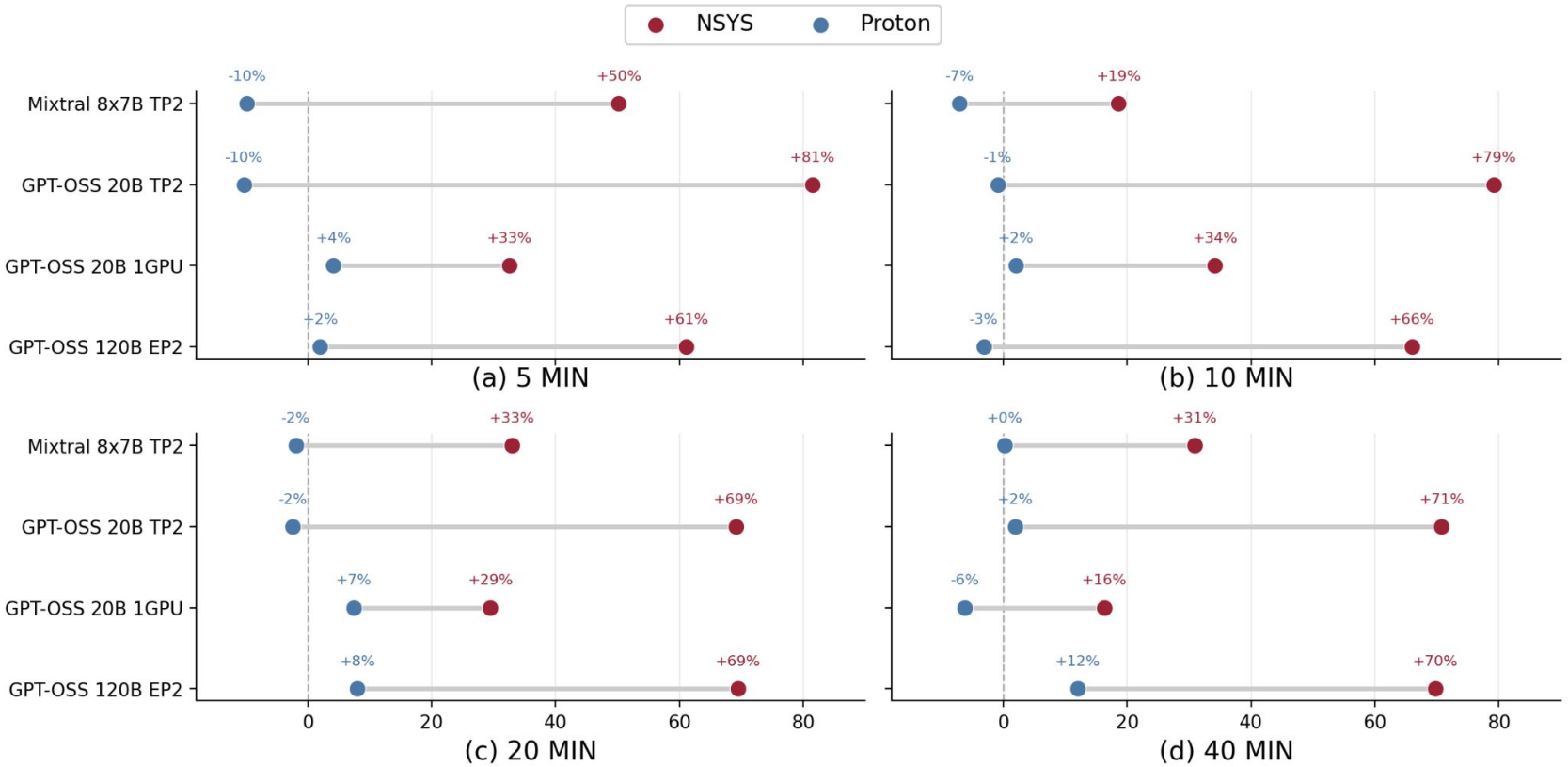
Environments

- Hardware
  - GPU: NVIDIA B200, 180 GB HBM
  - CPU: 2× Intel Xeon Platinum 8570, 112 cores (224 threads)
  - Memory: 2.0 TiB DDR
- OS
  - Red Hat Enterprise Linux 10.0 (Coughlan)
- Software
  - vllm: 0.1.dev14510+g7d9714cbf
  - triton: 3.7.1

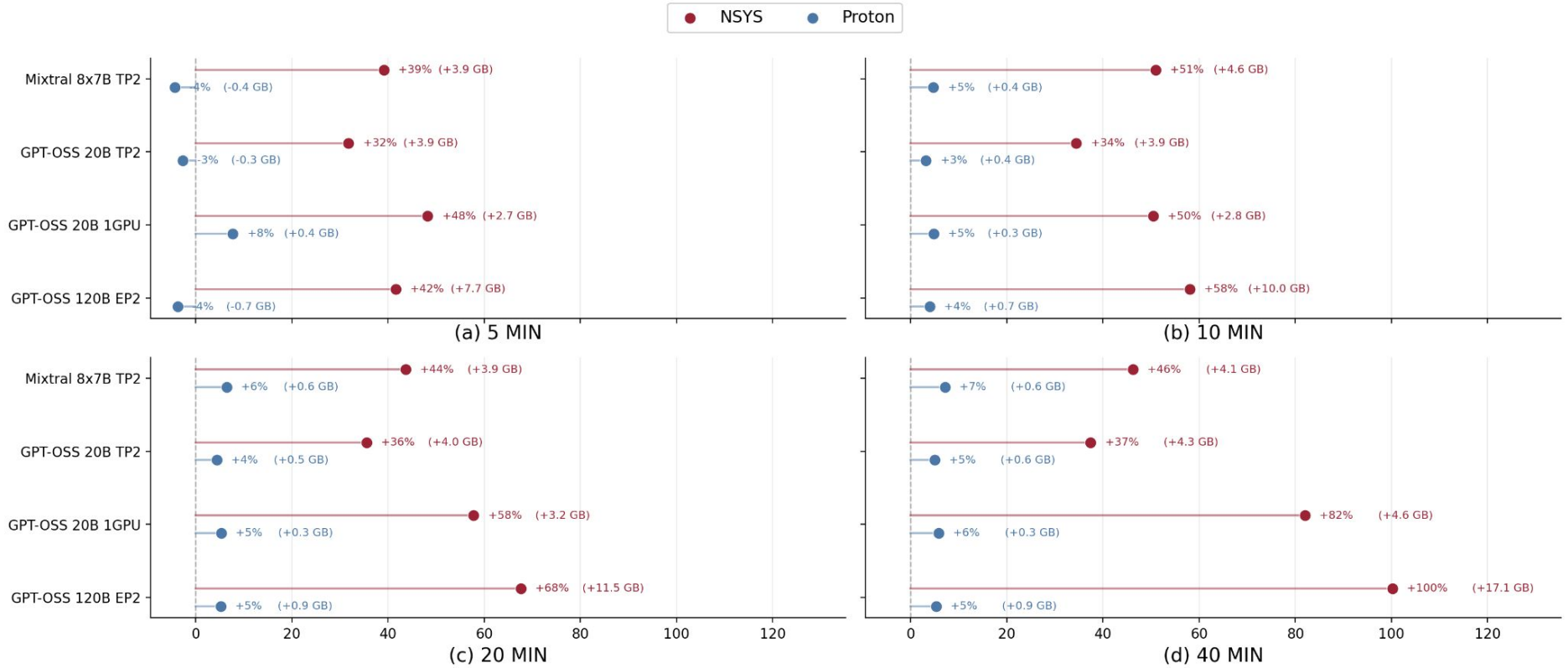
Setup

- Four models
  - GPT-OSS-20B
  - GPT-OSS-20B-TP2
  - GPT-OSS-120B-EP2
  - Mixtral-8-7B-EP2
- Two profilers
  - nsys
  - proton (periodic profiling mode)
- vLLM workloads
  - bench serve mode
  - random prompts (in = 500 tokens, out = 2000 tokens)
  - batches of 50 prompts

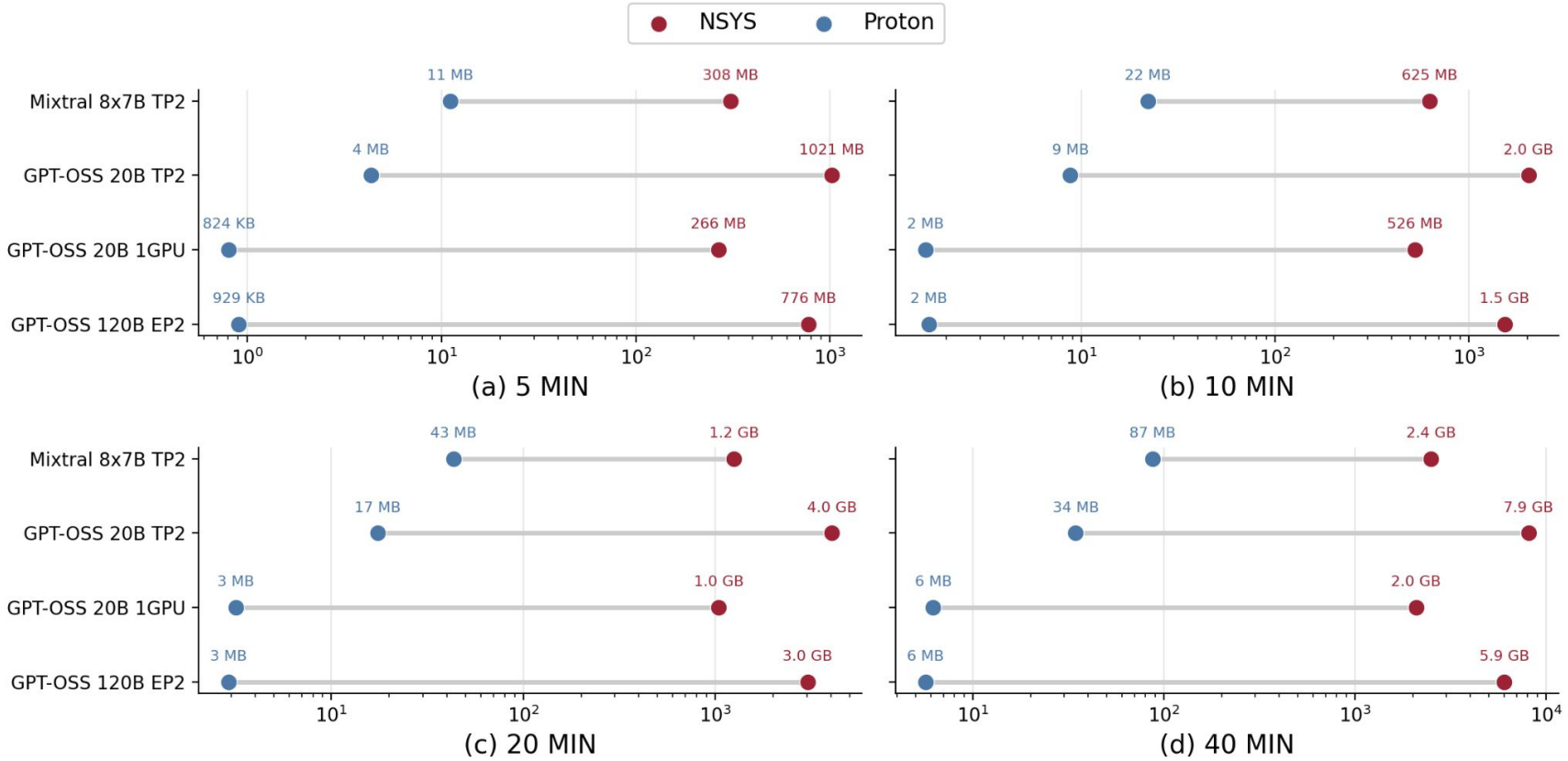
Runtime Overhead



CPU Memory Overhead



Storage Overhead



Conclusions

Related Work

- Our paper
  - [SC'26] *LLMPROF: Identifying Performance Bottlenecks in LLM Serving Systems with Top-Down Profiling*
  - Integrated into PyTorch/Tokenspeed/vLLM (in progress)
- Papers from Chinese companies
  - [Tencent] *ARGUS: Production-Scale Tracing and Performance Diagnosis for over 10,000-GPU Clusters*
  - [Alibaba][OSDI'26] *StriaTrace: Efficient Tracing and Diagnosis for Online LLM Inference (Operational Systems)*

Takeaways

- Proton provides fine-grained kernel-level performance insights while remaining efficient for long-running production deployments
    - How to profile MegaKernels?
- Co-designing the profiling framework and LLM services by exposing multi-layer, multi-level profiling APIs is the best practice
    - There is no single profiling tool that fits all frameworks