

Low Overhead, Continuous Profiling for AI Services

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



AI System Software Stack



Challenges in Optimizing AI (LLM) Systems

High-level Metric

Total throughput / Generation throughput / Time to first token / Time per output token / ...

Phases

Prefill / Decode

Position

Outside GPU

Inside GPU

Category

CPU Bound

Comm. Bound

Compute bound

Bandwidth bound

Bottleneck Type

Preprocess

Schedule

Kernel
launch

Network bandwidth

Ineff. Quant. Kernel

Memory access
pattern

Root Cause

Async.
pipeline

Batching
strategy

Missing
CUDA
Graph

Improper
parallelism setup

Low L2 hit rate / Workload imbalance /
Bad tuning

Proton

Proton API

- Designed for flexible, low-overhead, and iterative profiling and optimization of GPU kernels, especially Triton
- Proton profiler API
 - `proton.start/proton.finalize`
- Proton instrumentation API
 - `proton.start(backend="instrumentation")`
- Proton PC sampling API
 - `proton.start(backend="cupti", mode="pcsampling")`

Comparison with Nsys/Torch profiler

- More detailed and custom performance metrics
 - Scheduler
 - Kernel invocation context, even inside the CUDAGraph replay
 - (Gluon) Intra-kernel monitoring
- Always-on continuous profiling with negligible performance overhead
 - Suitable for online serving & RL that would run for very long time.
- Less CPU memory overhead and OOM
 - Typical single profile size is < 30MB

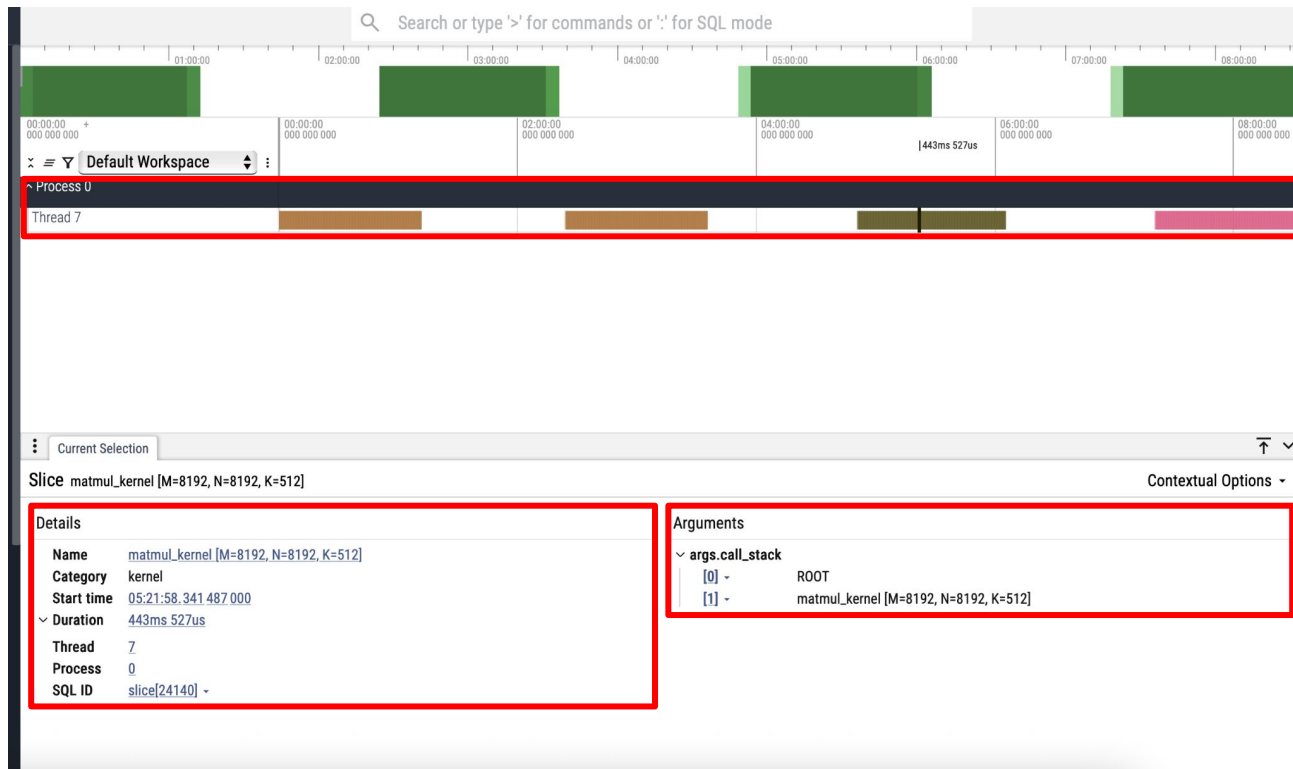
Scope Annotation and Metrics Collection

- Scopes: a user-defined region with semantic information
 - MLP
 - Forward
- User defined metrics will be aggregated into the annotated scope
 - FLOPs
 - #Bytes
 - #Tokens
- API
 - `with proton.scope("name", metrics={name : double | float})`

Call Path Profiling

```
13212533.430 nan ROOT
├─ 11948517.270 2685.285 execute_3_context_0(sq0sk0sqsq0sqsk0)_generation_3(sq3sk7496sqsq3sqsk7496)
│   ├── 2.464 nan _compute_slot_mappings_kernel
│   ├── 3.392 nan _gather_block_tables_kernel
│   └─ 11948502.774 nan cudagraph_replay_full [graph_id=4,num_tokens=4,num_reqs=4]
│       └─ 11948502.774 nan <captured_at>
│           └─ 11948502.774 nan cudagraph_capture_full [graph_id=4,num_tokens=4,num_reqs=4]
│               └─ 122690.232 nan _ZN8cublasLt19splitKreduce_kernelILi32ELi16Eif13__nv_bfloat16fS1_Lb0EfS
amsIT4_EEPKT2_PKT8_PT7_PT3_PKS3_SG_PKT9_S7_PSH_Pv1PS3_PiSM_SM_SG_SG_SG_SG_SG_
│   ├── 12804.043 nan model.layers.0.self_attn.attn.attention
│   ├── 1135.202 nan model.layers.0.self_attn.attn.kv_cache_update
│   ├── 11777.572 nan model.layers.31.self_attn.attn.attention
│   ├── 1025.917 nan model.layers.31.self_attn.attn.kv_cache_update
│   ├── 11854.202 nan model.layers.63.self_attn.attn.attention
│   ├── 1054.371 nan model.layers.63.self_attn.attn.kv_cache_update
│   ├── 1194259.792 nan nvjet_sm90_tst_128x8_64x12_4x1_v_bz_TNT
│   ├── 3791385.795 nan nvjet_sm90_tst_128x8_64x12_4x1_v_bz_splitK_TNT
│   ├── 5677693.696 nan nvjet_sm90_tst_256x8_64x6_2x1_v_bz_TNT
│   ├── 53641.162 nan triton_poi_fused_4
│   ├── 51268.951 nan triton_poi_fused_mul_silu_slice_1
│   ├── 82596.281 nan triton_red_fused_fused_add_rms_norm_0
│   ├── 106807.373 nan triton_red_fused_fused_add_rms_norm_2
│   └─ 2.720 nan mrv2.combine_sampled_and_draft_tokens [num_reqs=3]
└─ 1.216 nan mrv2.prepare_pos_seq_lens [num_reqs=3]
11891280.18 - 13212533.43
9248773.67 - 11891280.18
6606267.16 - 9248773.67
3963760.66 - 6606267.16
1321254.15 - 3963760.66
```

Timeline Tracing



GPU Streams

Kernel Info

Call Path

Use Case: Persistent matmul

See [09-persistent-matmul.py](#) for more details

```
570.292 9639.901 ROOT
├─ 625.347 1098.902 cublas [M=8192, N=8192, K=512]
│   └─ nan 1098.902 nvjet_hsh_128x256_64x4_2x1_v_bz_coopA_TNN
├─ 504.396 1362.410 matmul_kernel [M=8192, N=8192, K=512]
├─ 568.195 1209.434 matmul_kernel_descriptor_persistent [M=8192, N=8192, K=512]
├─ 581.411 1181.943 matmul_kernel_descriptor_persistent_ws [M=8192, N=8192, K=512]
├─ 545.982 1258.641 matmul_kernel_persistent [M=8192, N=8192, K=512]
├─ 548.539 1252.773 matmul_kernel_tma [M=8192, N=8192, K=512]
├─ 583.887 1176.932 matmul_kernel_tma_persistent [M=8192, N=8192, K=512]
└─ 625.367 1098.867 torch [M=8192, N=8192, K=512]
    └─ nan 1098.867 nvjet_hsh_128x256_64x4_2x1_v_bz_coopA_TNN
```

Legend (Metric: tflop16/s (inc) Min: 504.40 Max: 625.37)



613.27	–	625.37
589.08	–	613.27
564.88	–	589.08
540.69	–	564.88
516.49	–	540.69
504.40	–	516.49

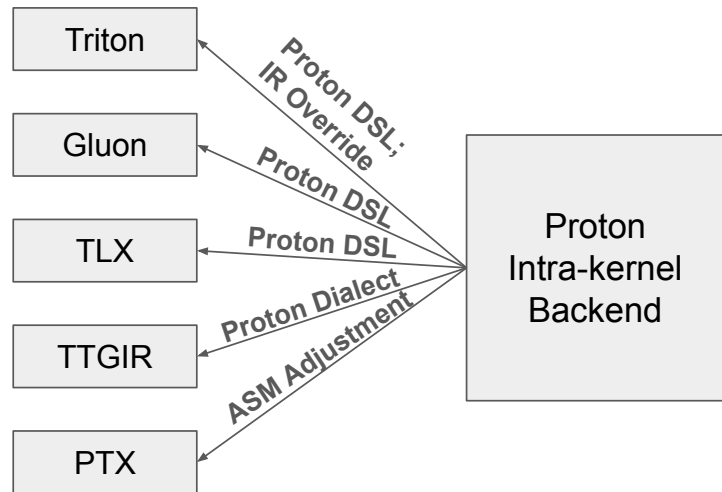
name User code ◀ Only in left graph ▶ Only in right graph

Challenges in Coarse-Grained API

- Desire to understand kernel behavior and GPUs for better optimizations
 - GPU HW extremely complicated
 - Producer-consumer programming
 - Identify critical paths
- Desire to enable fine-grained overlaps
 - Tensor core, Cuda core, SFU, TMA, NVLink
 - Better HW utilization, higher throughput
- Existing profiling tools can't help
 - Sampling-based: NCU, ROCm Compute

Fine-grained APIs

- Flexible user-defined fine-grained profiling
 - Scope-based timeline
 - OP-level latency
- Unlock customized analysis and optimization
 - Compiler integration
 - Dynamic traces



Use Cases

Case 1: Identify Critical Path



Case 3: Operation Latency

Case 2: Detect Resource Contention



OP Name	Latency (cycle)
tanh	800
multiply	550
dot	450
tma_load	1800

Continuous Profiling

Background

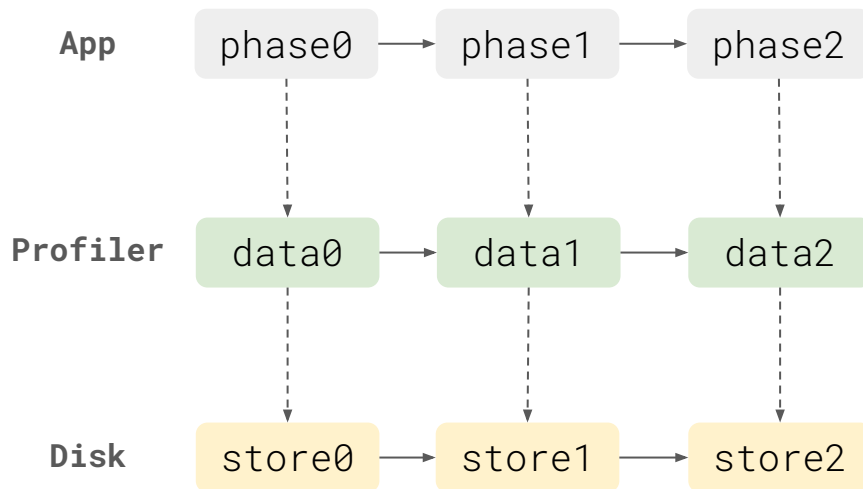
- LLM serving and RL systems heavily rely on GPUs and typically run continuously for days or even months
- Production workloads are inherently complex and dynamic
 - Dynamic shapes lead to varying kernel launch configurations and resource usage
 - Tail latency issues (e.g., *KV cache misses*) introduce 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
 - Significant runtime, memory, and storage overhead
 - 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)
```

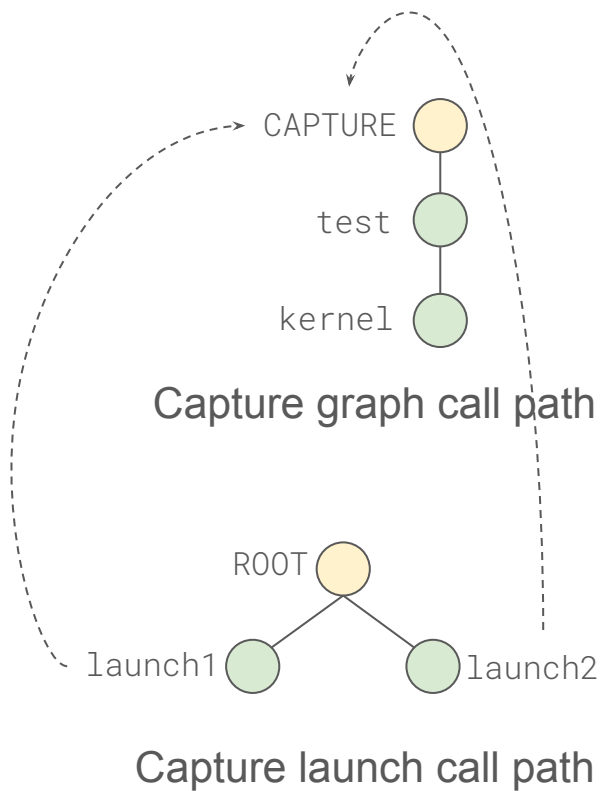
Background Mode

- 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



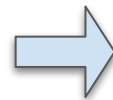
```
proton.start("profile")
```

```
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()
```



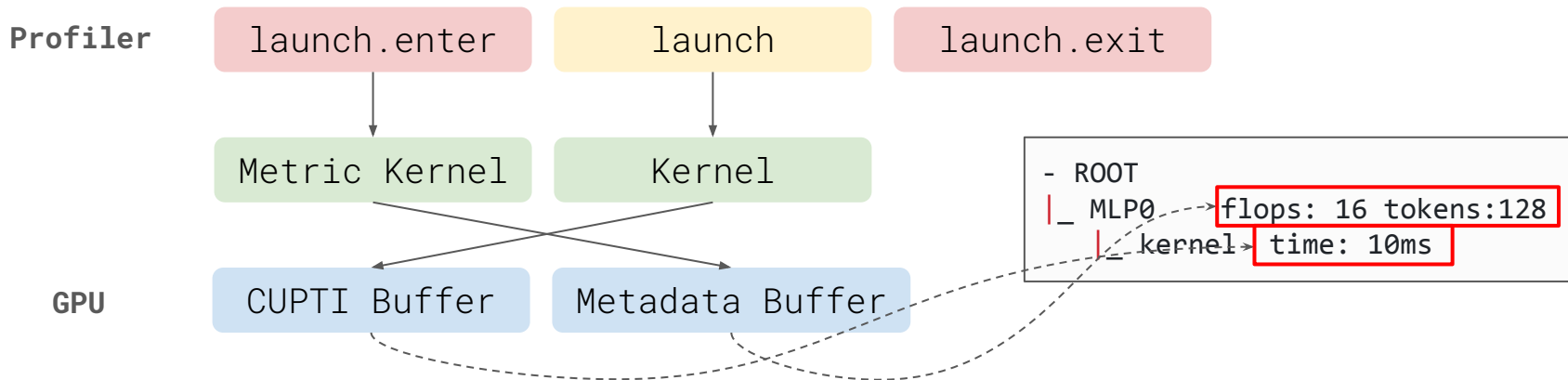
```
- ROOT  
|_ launch1  
   |_ test  
      |_ kernel  
|_ launch2  
   |_ test  
      |_ kernel
```

Key Features: Metadata Kernel Profiling

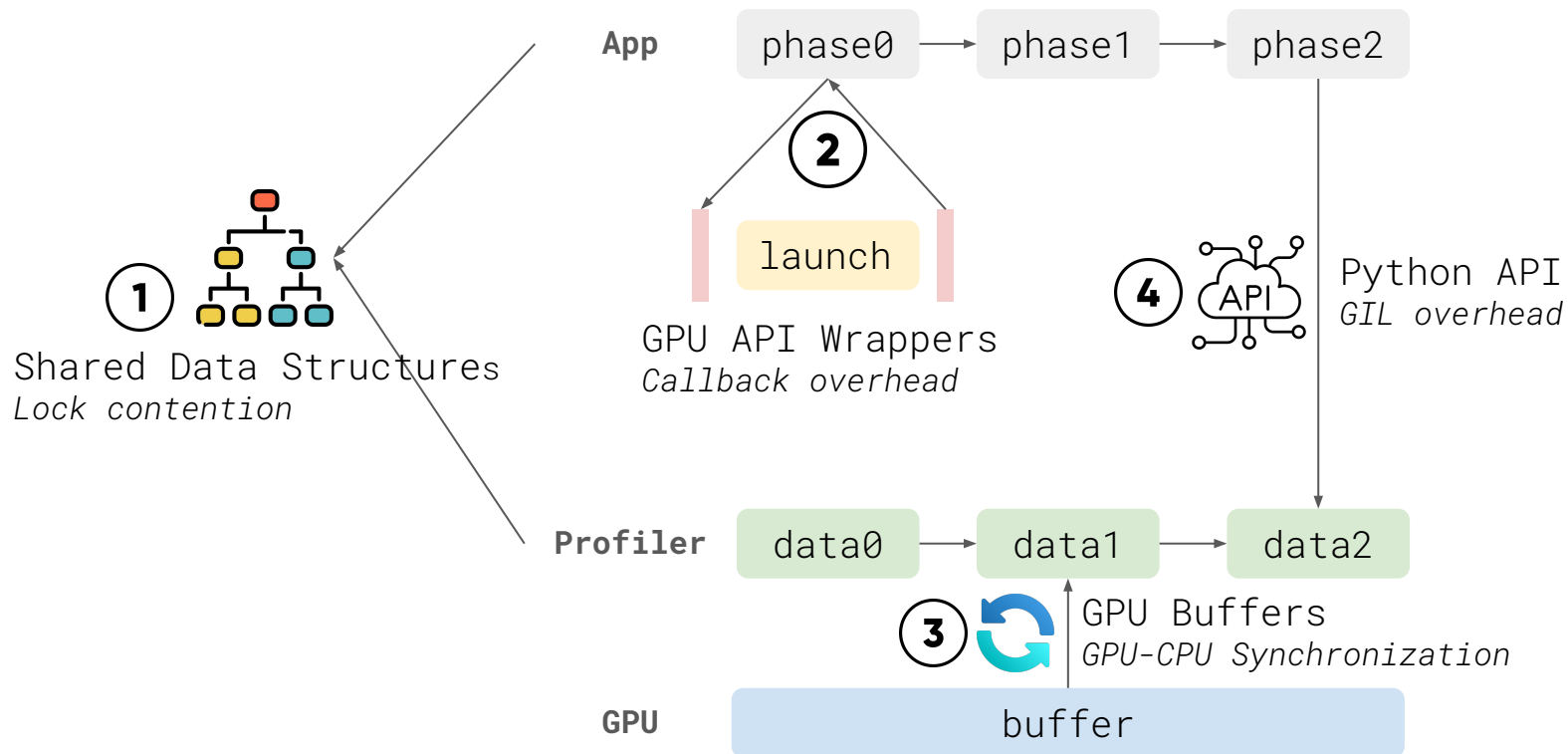
- Metadata (e.g., Tokens) is computed on the CPU
- We cannot copy data from the CPU to the GPU in cudagraph kernels
 - Unless it's from the pinned memory but not often used
- How do we store metadata on the GPU
 - And keep accumulating metadata without synchronizing the CPU and the GPU?

Key Features: Metadata Kernel Profiling

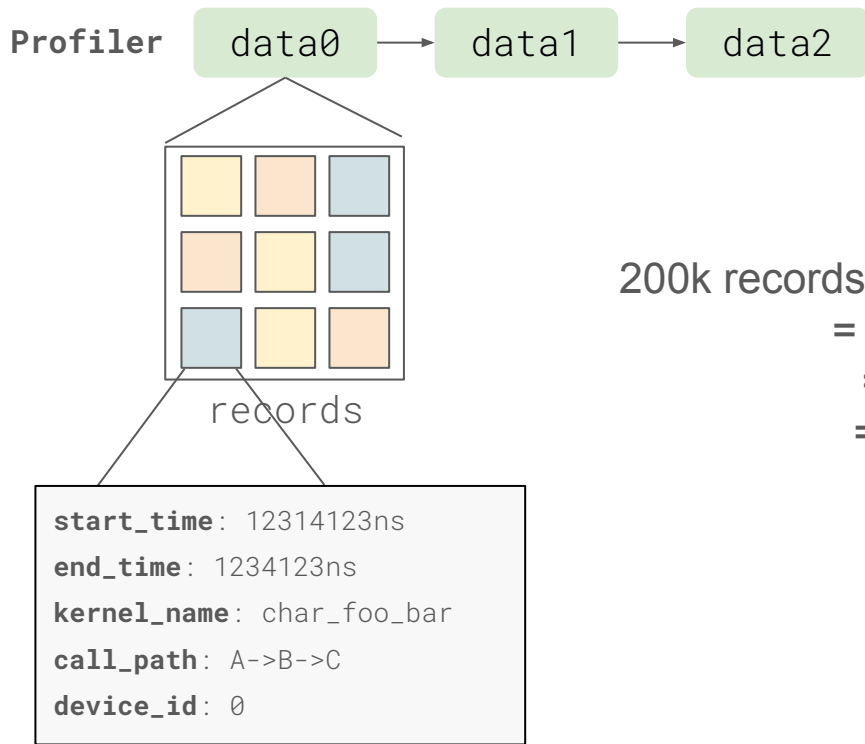
```
def fn():  
    with proton.scope("MLP0", metrics={"tokens": 128, "flops": a.sum() * 5}):  
        kernel()  
  
    with proton.scope("launch"):  
        g.replay()
```



Key Challenges: Runtime Overhead

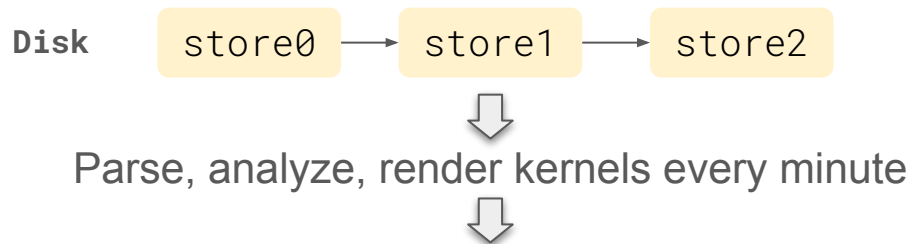


Key Challenges: Memory Overhead

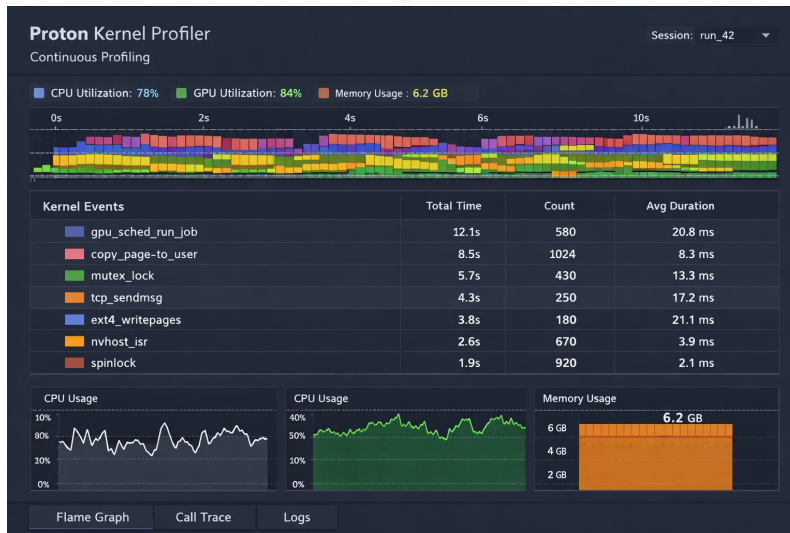


200k records/sec x 256 bytes/record
= 51.2MB/sec
= 3GB/min
= 180GB/hr

Key Challenges: Postmortem Overhead



Visualizer

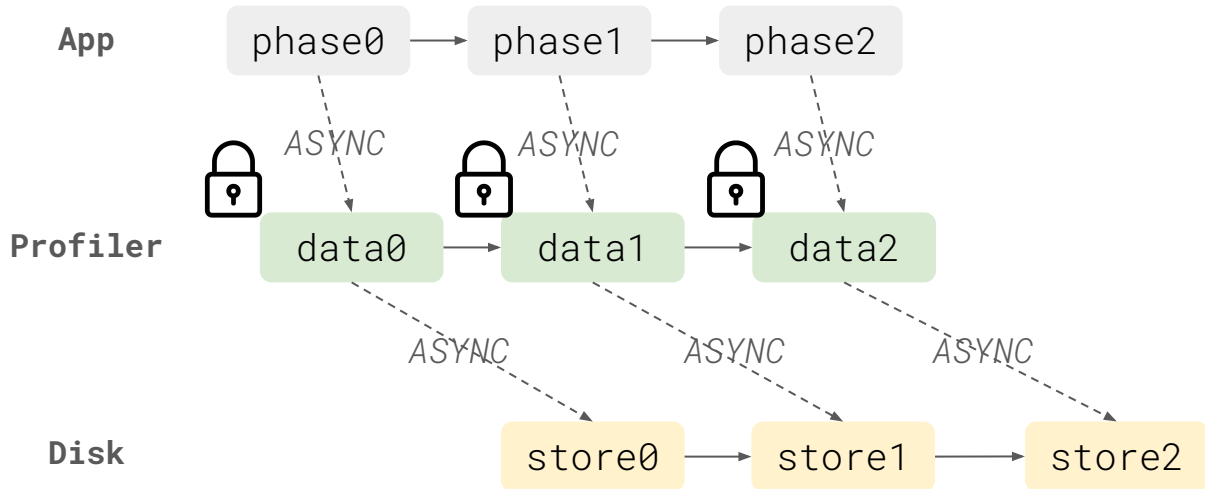


Optimizations

- 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

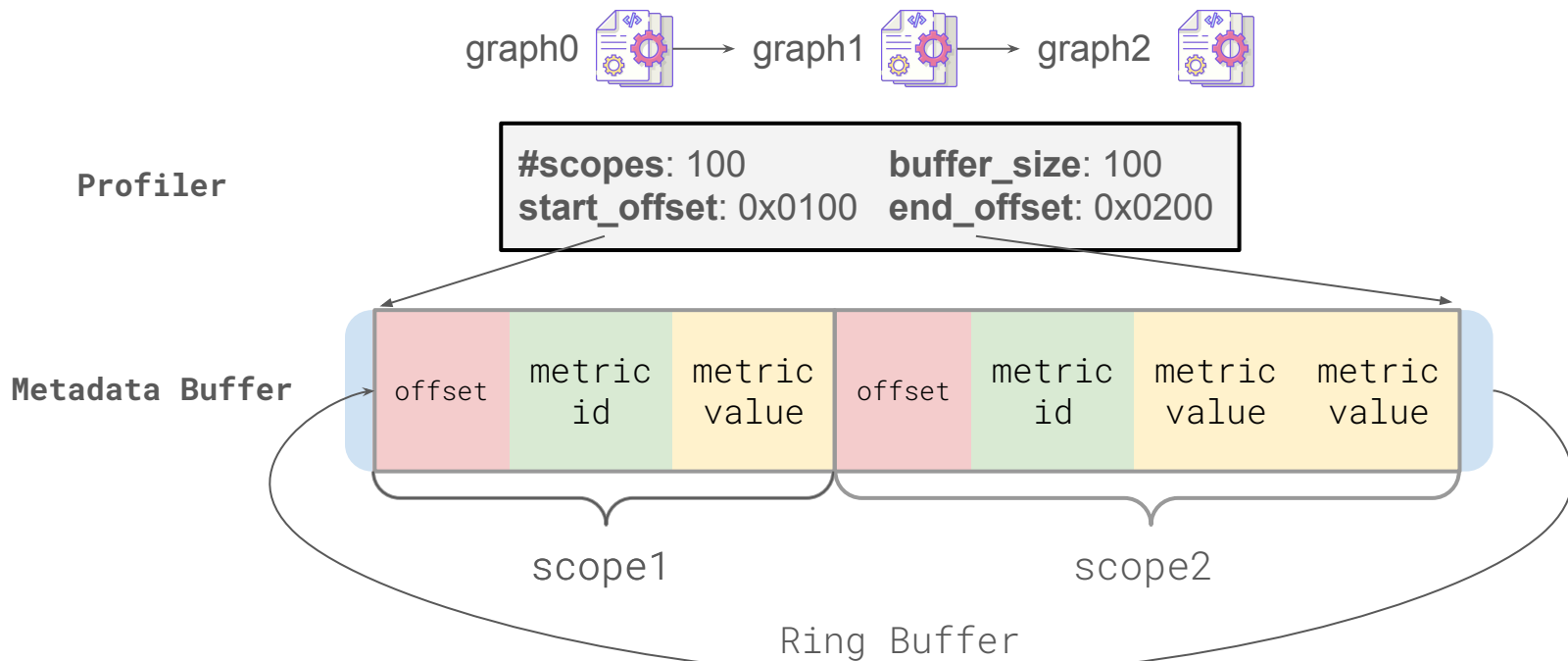
Runtime: Phase Separation

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

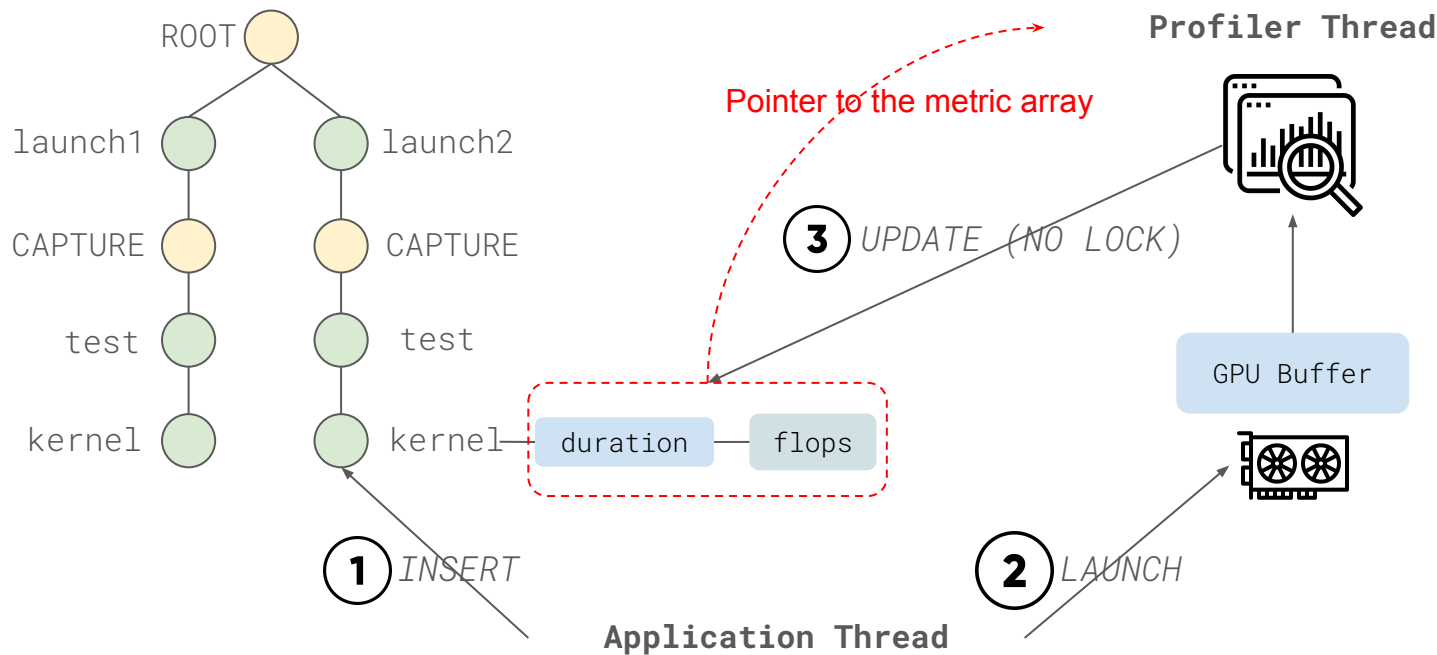


Runtime: Synchronization-free Buffer Copy

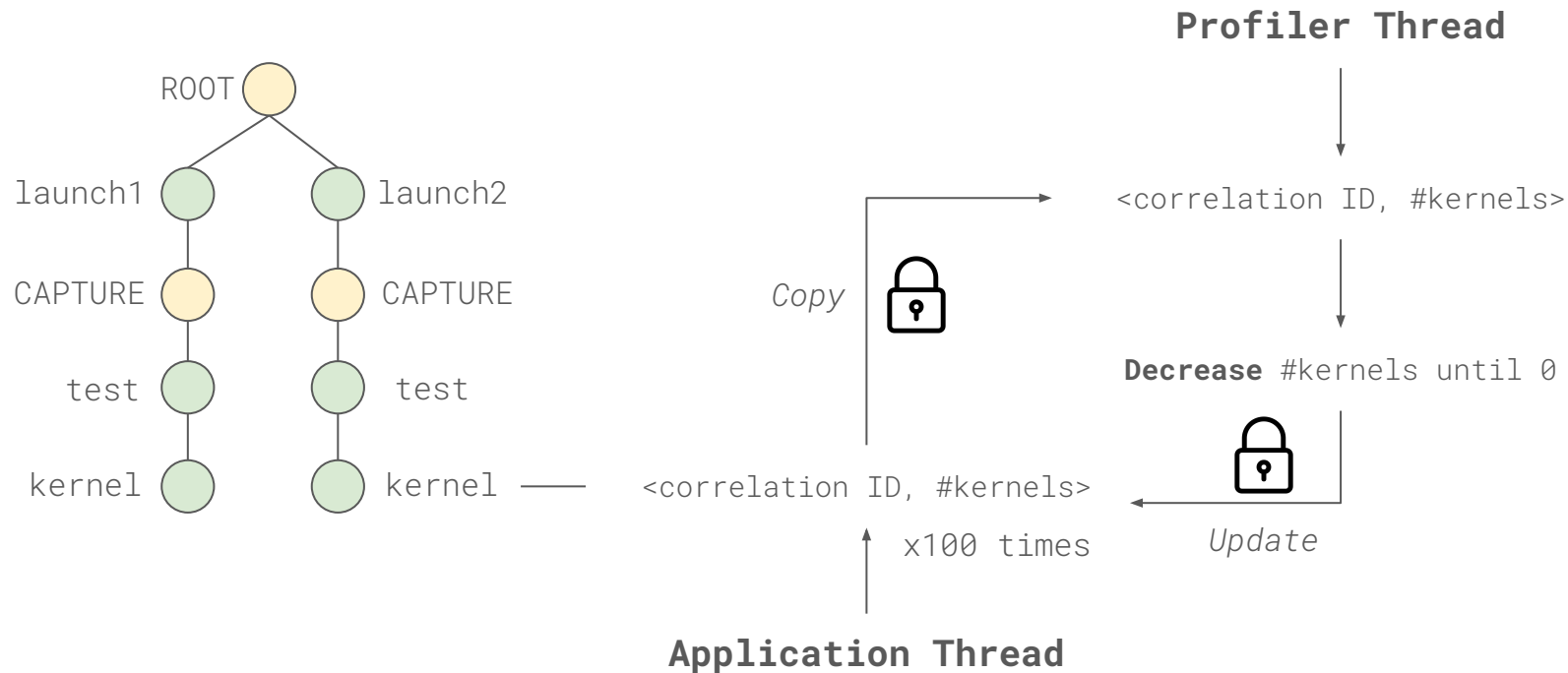
- Collecting metadata without trigger CPU-GPU flushing



Runtime: Launch-order Based Lock-free Metric Update

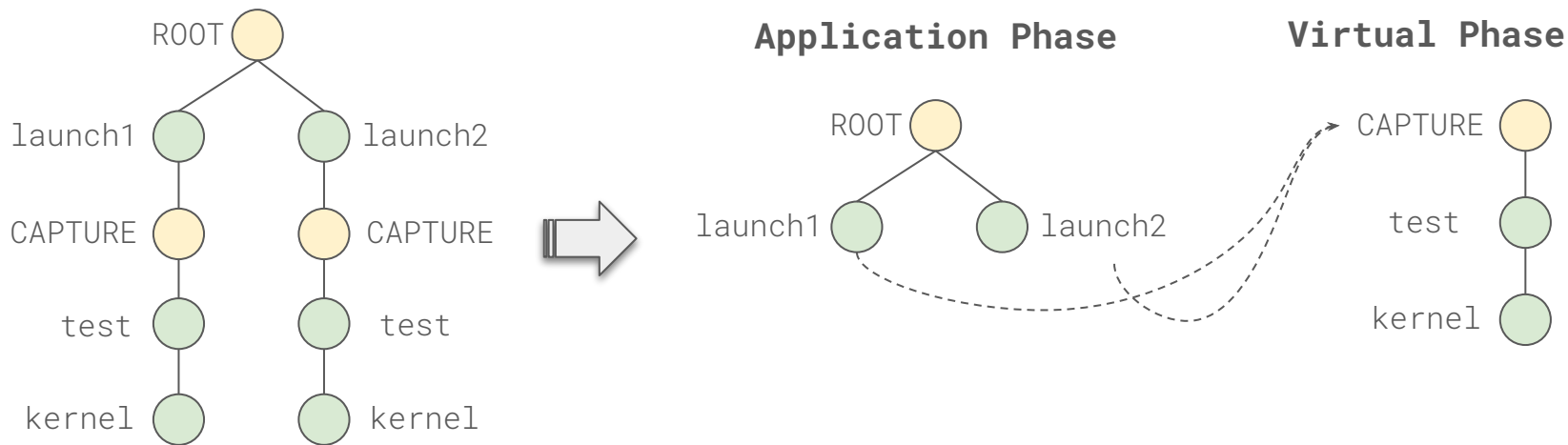


Runtime: Copy-on-Write Correlation



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

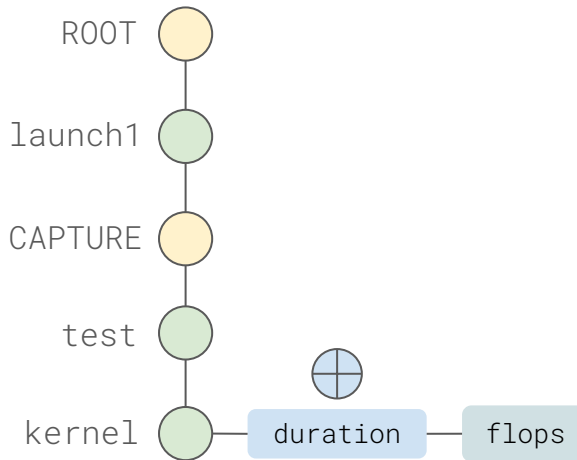
```
proton.start("profile")

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()
```

start_time: 12314123ns
end_time: 1234133ns

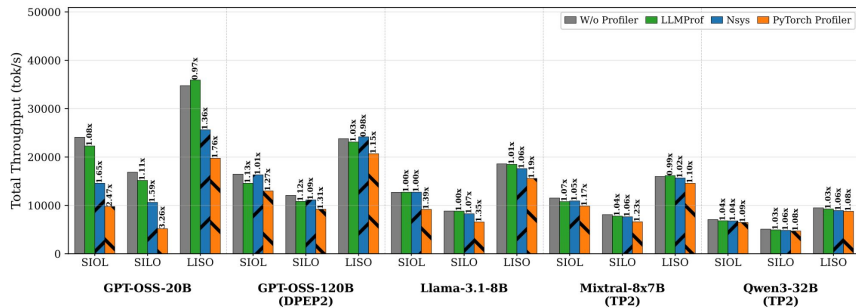


Experiments

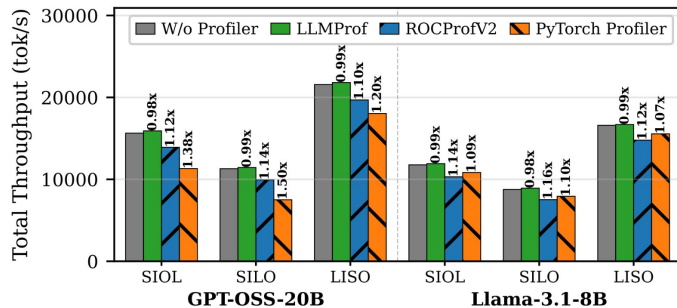
Environments

- Hardware
 - GPU: 8x NVIDIA H100 SXM
 - CPU: 128 vCPU
 - Memory: 1.6 TiB DDR
- OS
 - Ubuntu 22.04
- Software
 - vllm: v0.26.0rc1
 - PyTorch: 2.11+cu130

Negligible Runtime Performance Overhead



(a) NVIDIA H100



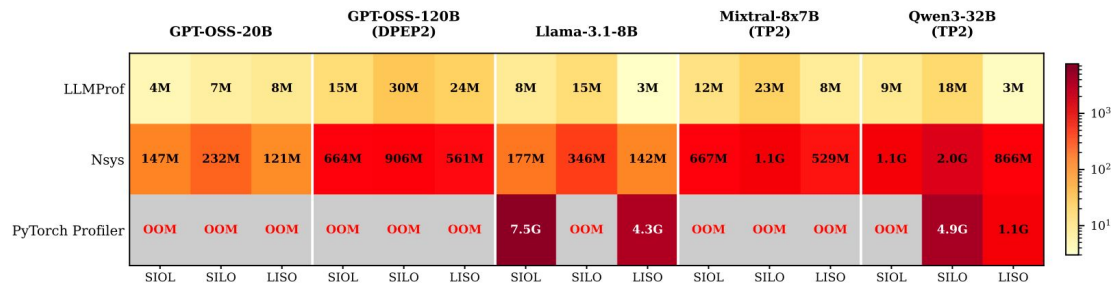
(b) AMD MI300X

TABLE III: Average profiling overhead ($\times$ slowdown over unprofiled baseline). Throughput metrics are inverted so that higher values consistently indicate more overhead.

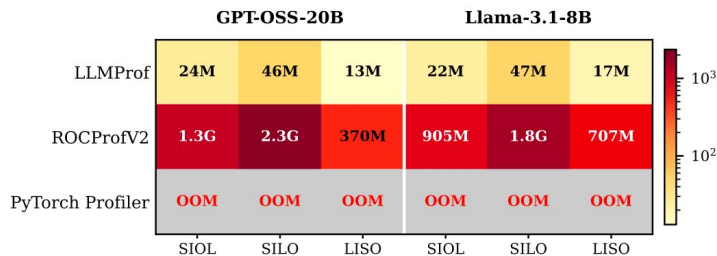
Platform	Tool	Total Tput	Gen Tput	TTFT	TPOT
H100	LLMPROF	1.04	1.04	1.04	1.06
	Nsys	1.14	1.13	1.18	1.14
	PyTorch Prof.	1.46	1.41	1.32	1.43
MI300X	LLMPROF	1.00	1.00	1.00	1.00
	ROCProfV2	1.13	1.05	1.11	1.13
	PyTorch Prof.	1.22	1.22	1.20	1.23

Fig. 15: Total throughput. Higher the better.

Small Profile Size



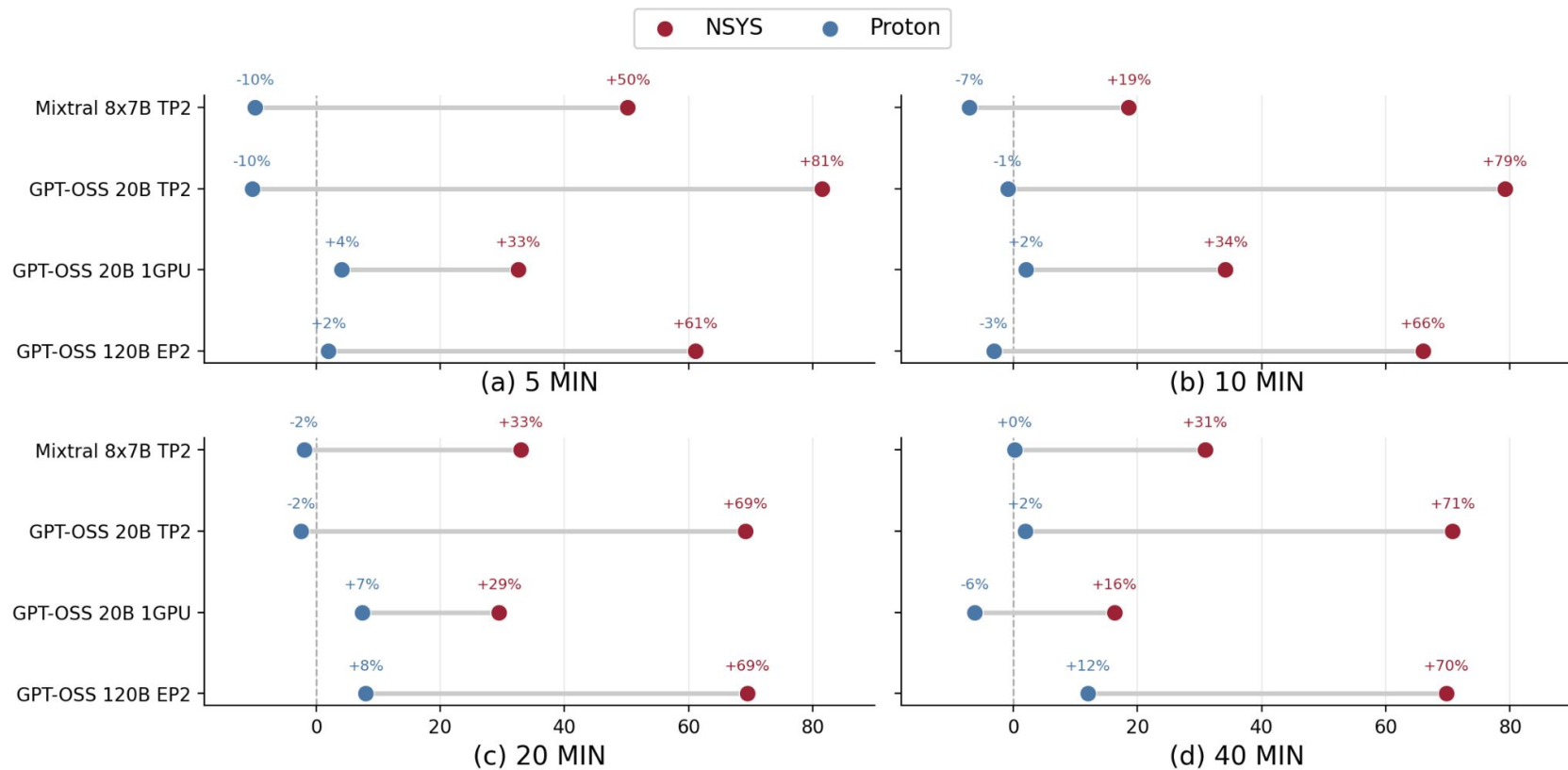
(a) NVIDIA H100



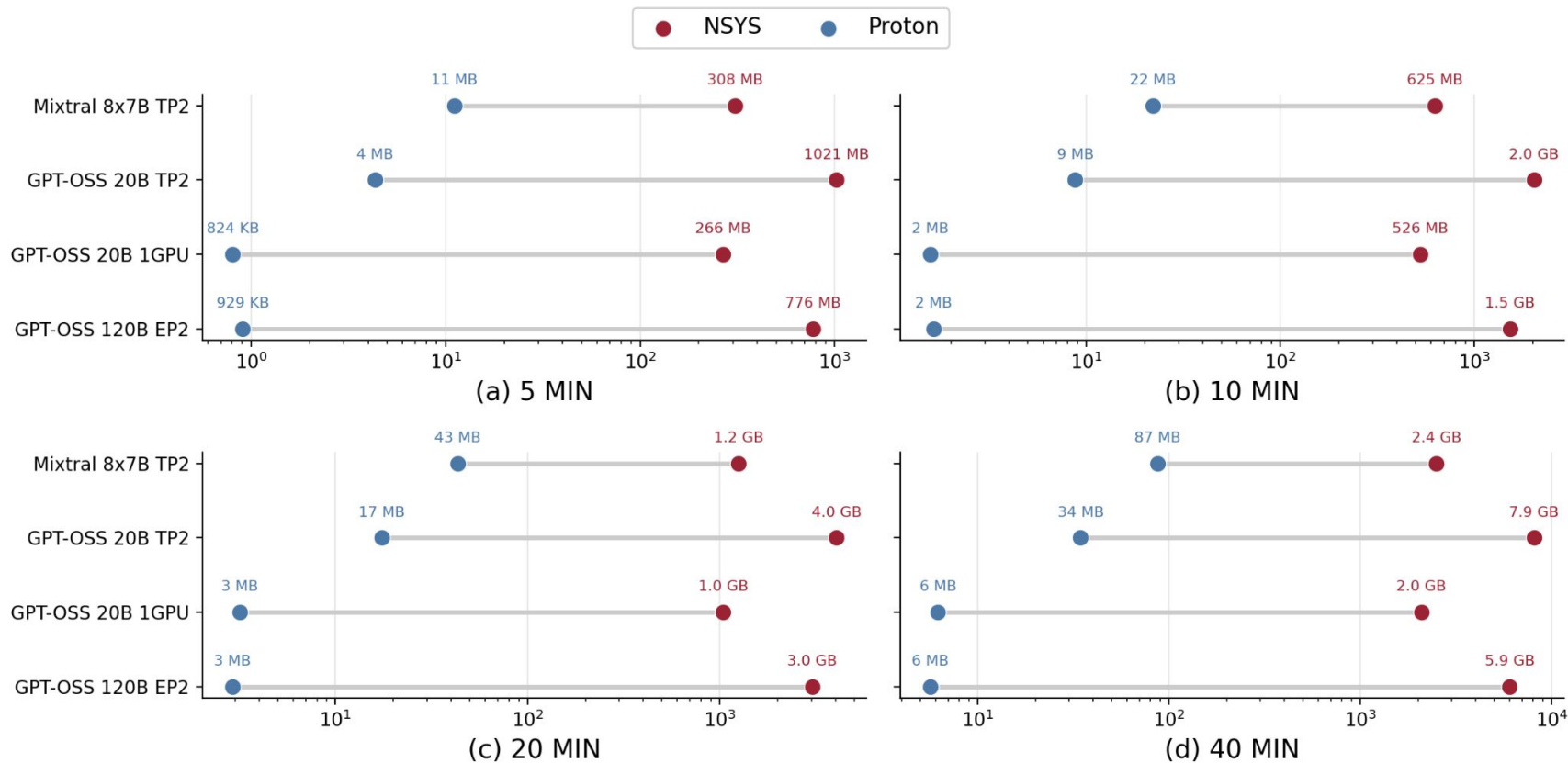
(b) AMD MI300X

Fig. 16: Profile size heatmap. Lower the better.

Continuous Profiling Overhead



Continuous Profiling Storage



Case Studies

Post-mortem Analysis with proton-viewer

For proton hatchet profiles, we can inspect them from different perspectives with proton-viewer. For example:

Check a specific model layer:

```
proton-viewer -m time/us,count -i '.*model[.]layers[.]0.*' profile.hatchet
```

Check a specific model forward:

```
proton-viewer -m time/us,count -i '.*execute_4096_context_3.*' profile.hatchet
```

Check diff between two profiles:

```
proton-viewer -m time/us,count --diff-profile run_1.hatchet run_2.hatchet
```

vLLM Profiling Results ver. Before

- **DeepEP high-throughput mode (vllm#24588)**
 - The only mode available for GPT-OSS. Low-latency mode is not compatible
 - No support for cudagraph -> High kernel launch & CPU-GPU synchronization overheads

```
12:02:04 (vllm) root@dccc-pcde2-2101-0-55bd-e90e-a23f vllm ±|tianle/accel_hackathon x|
→ proton-viewer -m time/ms,cpu_time/ms -t 100 proton_profile_43284_1761033764078415.hatchet
5620.921 nan ROOT
├─ 114.579 nan _ZN7deep_ep9intranode21cached_notify_combineILi8EEEvPPvPiiiiPS4_i
├─ 5402.243 15562.854 _model_forward
│   ├── 125.945 2169.415 OAITritonExperts-make_routing_data
│   ├── 655.040 4268.918 OAITritonExperts-triton_kernel_fused_experts
│   │   ├── 406.154 2009.787 matmul_ogs-w1
│   │   │   ├── 272.820 nan _p_matmul_ogs_NNT_bf16xbf16xmxfp4_16x256x128x1_swiglu
│   │   │   └─ 248.886 2126.647 matmul_ogs-w2
│   │   │       └─ 156.018 nan _p_matmul_ogs_NNT_bf16xbf16xmxfp4_16x256x128x1
│   ├── 639.829 nan _ZN7deep_ep9intranode15notify_dispatchILi8EEEvPKiPiS3_S4_iiiPKbS4_S4_iiPPvPS
│   ├── 3145.280 nan _ZN7deep_ep9intranode21cached_notify_combineILi8EEEvPPvPiiiiPS4_i
│   ├── 270.848 nan _ZN7deep_ep9intranode7combineI13_nv_bfloat16Li8ELi768ELi4096EEEvPT_PfPKS3_F
│   └─ 212.743 nan _ZN7deep_ep9intranode8dispatchILi8ELi768ELi8192EEEvP4int4PfPiPLS4_S5_S5_PKS2
```

Legend (Metric: time/ms (inc) Min: 114.58 Max: 5620.92)

5070.29 - 5620.92
3969.02 - 5070.29
2867.75 - 3969.02
1766.48 - 2867.75
665.21 - 1766.48
114.58 - 665.21

name User code ◀ Only in left graph ▶ Only in right graph

Setup

- 2xB200 GPU
- input/output len: 1000
- num prompt: 2048
- max concurrency: inf

gpt-oss-20b	DP=2 (eager)
Successful requests	2048
Benchmark duration (s)	149.55
Total input tokens	2045956
Total generated tokens	2048000
Request throughput (req/s)	13.69
Output token throughput (tok/s)	13694.81
Peak output token throughput (tok/s)	22178
Peak concurrent requests	2048
Total Token throughput (tok/s)	27375.96
-----Time to First Token-----	
Mean TTFT (ms)	13601.12
Median TTFT (ms)	13265.34
P99 TTFT (ms)	26072.52
-----Time per Output Token (excl. 1st token)-----	
Mean TPOT (ms)	111.4
Median TPOT (ms)	112.32
P99 TPOT (ms)	118.76
-----Inter-token Latency-----	
Mean ITL (ms)	111.53
Median ITL (ms)	99.31
P99 ITL (ms)	382.69

Integration Plan

To integrate, we can reuse a lot of existing things in vLLM with minor modifications. We have merged ([#48789](#)) a minimal eager mode-only integration.

- Reuse: the shared profiler trigger in inference frameworks
 - `/start_profile` & `/end_profile` endpoint in vLLM
- Reuse: the existing NVTX scope
- Modification: Activate proton scope during CUDAGraph capture
- Modification: More annotations around kernel invocations for custom metrics

Takeaways

- Proton provides fine-grained kernel-level performance insights while remaining efficient for long-running production deployments
- 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