KubeCon

CloudNativeCon

THE LINUX FOUNDATION

OPEN SOURCE SUMMIT

AI_dev

Open Source GenAI & ML Summit

China 2024

KubeCon CloudNativeCon
THE LINUX FOUNDATION OPEN SOURCE SUMMIT
AI_dev Open Source GenAI & ML Summit
China 2024

Unlocking LLM Performance with eBPF: Optimizing Training and Inference Pipelines

Yang Xiang, Yunshan Networks

Outline

1. **Background**: Challenges in Training and Inference Efficiency
2. **Status Quo**: Issues with Traditional Solutions and Tools
3. **Approach**: Building Zero-Code Observability with eBPF
4. **Practical Case**: Full-Stack Profiling and D-Tracing in PyTorch

LLM Training: High Costs and Low Efficiency

	GPT-4	Llama-3.1
Size	1.8T	405B
GPU	25K A100	16K H100
Days	90~100	54
MFU	32%~36%	38%~43%

- Everything We Know About GPT-4 - Klu.ai
- GPT4- All Details Leaked
- The Llama 3 Herd of Models

Training duration: Months

GPU count: >10K

Model size: Trillions of params

GPU MFU: ~40%

Component	Category	Interruption Count	% of Interruptions
Faulty GPU	GPU	148	30.1%
GPU HBM3 Memory	GPU	72	17.2%
Software Bug	Dependency	54	12.9%
Network Switch/Cable	Network	35	8.4%
Host Maintenance	Unplanned Maintenance	32	7.6%
GPU SRAM Memory	GPU	19	4.5%
GPU System Processor	GPU	17	4.1%
NIC	Host	7	1.7%
NCCL Watchdog Timeouts	Unknown	7	1.7%
Silent Data Corruption	GPU	6	1.4%
GPU Thermal Interface + Sensor	GPU	6	1.4%
SSD	Host	3	0.7%
Power Supply	Host	3	0.7%
Server Chassis	Host	2	0.5%
IO Expansion Board	Host	2	0.5%
Dependency	Dependency	2	0.5%
CPU	Host	2	0.5%
System Memory	Host	2	0.5%

Table 5 Root-cause categorization of unexpected interruptions during a 54-day period of Llama 3 405B pre-training. About 78% of unexpected interruptions were attributed to confirmed or suspected hardware issues.

148/54*365/16384 = 6%

(148+72+19+17+6+6)/54*365/16384 = 11%

Annual GPU failure rate: 6%-11%

Causes of Training Inefficiency Beyond Failures

Classification of the 706 low-GPU-utilization issues discovered across 400 deep learning jobs.

Category	Description	No.	Ratio
Interactive Job	The execution of a job entails regular interaction with its owner.	15	2.12%
GPU Oversubscription	A job requests more GPUs than it actually utilizes.	6	0.85%
Unreleased Job	A job does not terminate promptly after completing its computation.	9	1.27%
Non-DL Job	A job is unrelated to deep learning and does not utilize GPUs at all. For example, the job performs data analysis using CPUs solely.	4	0.57%
Improper Batch Size	Improper values of the batch size are used, which decrease the GPU computation of deep learning operators.	181	25.64%
Insufficient GPU Memory	The GPU memory is not sufficient to support more GPU computation.	22	3.12%
Model Checkpointing	A job saves model checkpoints synchronously to the distributed data store.	116	16.43%
Inefficient Host-GPU Data Transfer	The data transfer between main memory and GPU memory is not efficient.	197	27.90%
Data Preprocessing	Raw input data is preprocessed using CPUs before model training.	28	3.97%
Remote Data Read	A job opens and reads input data directly from the distributed data store.	18	2.55%
External Data Usage	A job accesses input data or model files directly from external sites.	18	2.55%
Intermediate Result Upload	A job saves intermediate training results synchronously to the distributed data store.	14	1.98%
Data Exchange	The GPUs of a distributed job continually exchange data, such as gradients and output tensors, among one another.	50	7.08%
Long Library Installation	The installation of dependent libraries takes too much time (at least 10 minutes).	12	1.70%
API Misuse	API usage violates assumptions.	16	2.27%

- Yanjie Gao (Microsoft Research) et al, ACM ICSE 2024, An Empirical Study on Low GPU Utilization of Deep Learning Jobs.
- Yanjie Gao (Microsoft Research) et al, ACM ICSE 2023, An Empirical Study on Quality Issues of Deep Learning Platform.

How can you determine if your training job has these inefficiencies?

GPU Kernels

```
1 from torch.utils.data import DataLoader
2
3 train_batch_size = 32 384
4 eval_batch_size  = 32 448
5 train_loader = DataLoader(dataset = train_data, batch_size =
  ↪  train_batch_size, shuffle = True)
6 eval_loader  = DataLoader(dataset = eval_data, batch_size =
  ↪  eval_batch_size, shuffle = True)
```

Figure 2: A simplified example of Improper Batch Size issues. "train_batch_size" and "eval_batch_size" are specified arguments for model training and evaluation, respectively. The fix is to increase their values independently (lines 3–4).

Memory copy

```
1 import torch
2 from torch.utils.data import DataLoader
3
4 train_loader = DataLoader(train_set,..., num_workers=8,
  ↪  pin_memory=True)
5 eval_loader = DataLoader(eval_set, ..., num_workers=8,
  ↪  pin_memory=True)
6
7 for epoch in range(num_epochs):
8   ...
9   for _, data in enumerate(train_loader, 0):
10    # get the inputs
11    inputs, labels = data
12    inputs = inputs.to(device, non_blocking=True)
13    labels = labels.to(device, non_blocking=True)
```

Figure 4: A simplified example of Inefficient Host-GPU Data Transfer issues. The fix is to enable automatic memory pinning by setting the `pin_memory` parameter to `True` (lines 3–4). We further set the `non_blocking` parameter to `True` (lines 11–12), which tries asynchronous data transfer if possible.

```
1 import torch
2+import threading
3+import shutil
4+import os
5
6 def main():
7   ...
8   for epoch in range(start_epoch, num_epochs):
9     if step > num_training_steps:
10      break
11    ...
12    for i, batch in enumerate(tqdm(train_loader)):
13      ...
14      logits = model(input_data)
15      ...
16      optimizer.zero_grad()
17      loss.backward()
18      optimizer.step()
19      ...
20      f1, pred = evaluator.evaluate(val_loader, model, step)
21      ...
22      if f1 > max_f1:
23        max_f1 = f1
24        torch.save(model, remote_path)
25      + local_path = make_tmp_path(epoch, i)
26      + torch.save(model, local_path)
27
28      + def save_file(local_path, remote_path):
29      +   shutil.copyfile(local_path, remote_path)
30      +   os.remove(local_path)
31
32      + threading.Thread(target = save_file, args = [local_path,
        ↪  remote_path]).start()
33
34 if __name__ == '__main__':
35   main()
```

Figure 3: A simplified example of Model Checkpointing issues. The fix is to save the checkpoint to a local temporary file (lines 25–26), followed by an asynchronous copy to the remote data store in a separate thread (lines 28–32).

In a distributed job, GPUs consistently share data such as gradients and output tensors. This data exchange between GPUs, whether through the network or PCI Express bus, frequently interrupts their designated tasks and causes a sudden drop in GPU utilization to zero. Within the *Data Exchange* category, there are 50 (7.08%) issues. A common fix is to enhance communication efficiency by minimizing the frequency of data exchange (e.g., using large batch sizes) and enabling compressed communication [34, 39]. For users of Horovod [67], enlarging the `backward_passes_per_step` parameter[5] can help accumulate more local gradient updates and transmit them simultaneously. Developers can also leverage Microsoft DeepSpeed's 3D (data, model, and pipeline) parallelism, whose 1-bit Adam and 0/1 Adam [39] optimizers demonstrate significant reductions in communication volume.

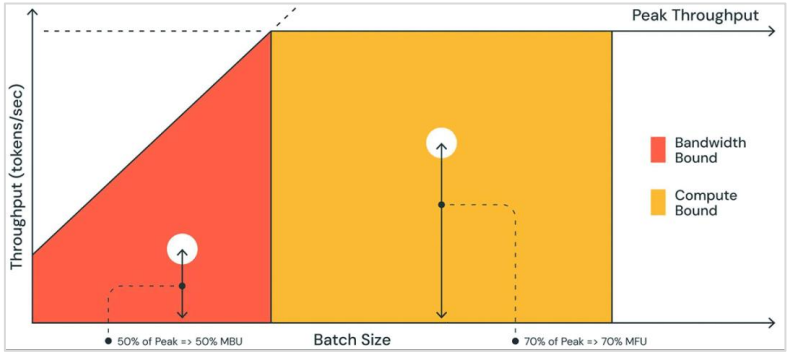
Network transmission

LLM Inference: High Costs and Latency

Llama	8B	70B	405B
FP32	36GB	267GB	**1.48TB**
FP16	20GB	135GB	758GB
INT8	12GB	70GB	382GB
INT4	8GB	37GB	193GB

80GB: 1 GPU
640GB: 1 Node x 8 GPU
1.28TB: 2 Node x 8 GPU

Time To First Token (TTFT)
Time Per Output Token (TPOT)
Model Bandwidth Utilization (MBU)



Fewer GPUs?

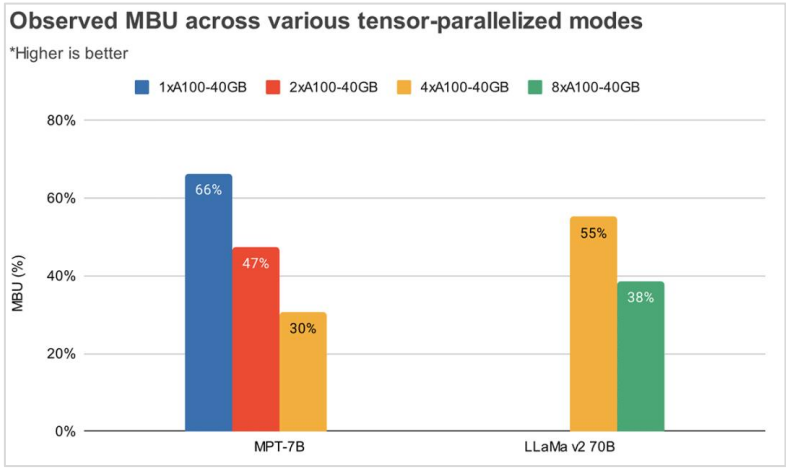
With fewer GPUs, each GPU needs to load more model parameters.

LLM Memory Requirements

Inference Training

Total Inference Memory: 1.48 TB

- **Model Weights:** 1.47 TB
- **KV Cache:** 2.00 GB
- **Activation Memory:** 3.56 GB



More GPUs?

With more GPUs, collective communication becomes more complex, and memory fragmentation increases.

- LLM Memory Requirements
- LLM Inference Performance Engineering: Best Practices

No Silver Bullet — Observability is the Prerequisite for Optimization.

Challenges in Troubleshooting Memory Consumption during LLM Inference

- Biz
- vLLM
- PyTorch
- Python / C++

PyTorch

Sig

Help Needed from vLLM team on profiling pytorch cuda memory

youkaichao1 1 Mar 16

Hi, I'm working on GitHub - vllm-project/vllm: A high-throughput and memory-efficient inference and serving engine for LLMs 7 , and the recently release of pytorch 2.2.0 caused some trouble to me. I came here for help in profiling cuda memory usage.

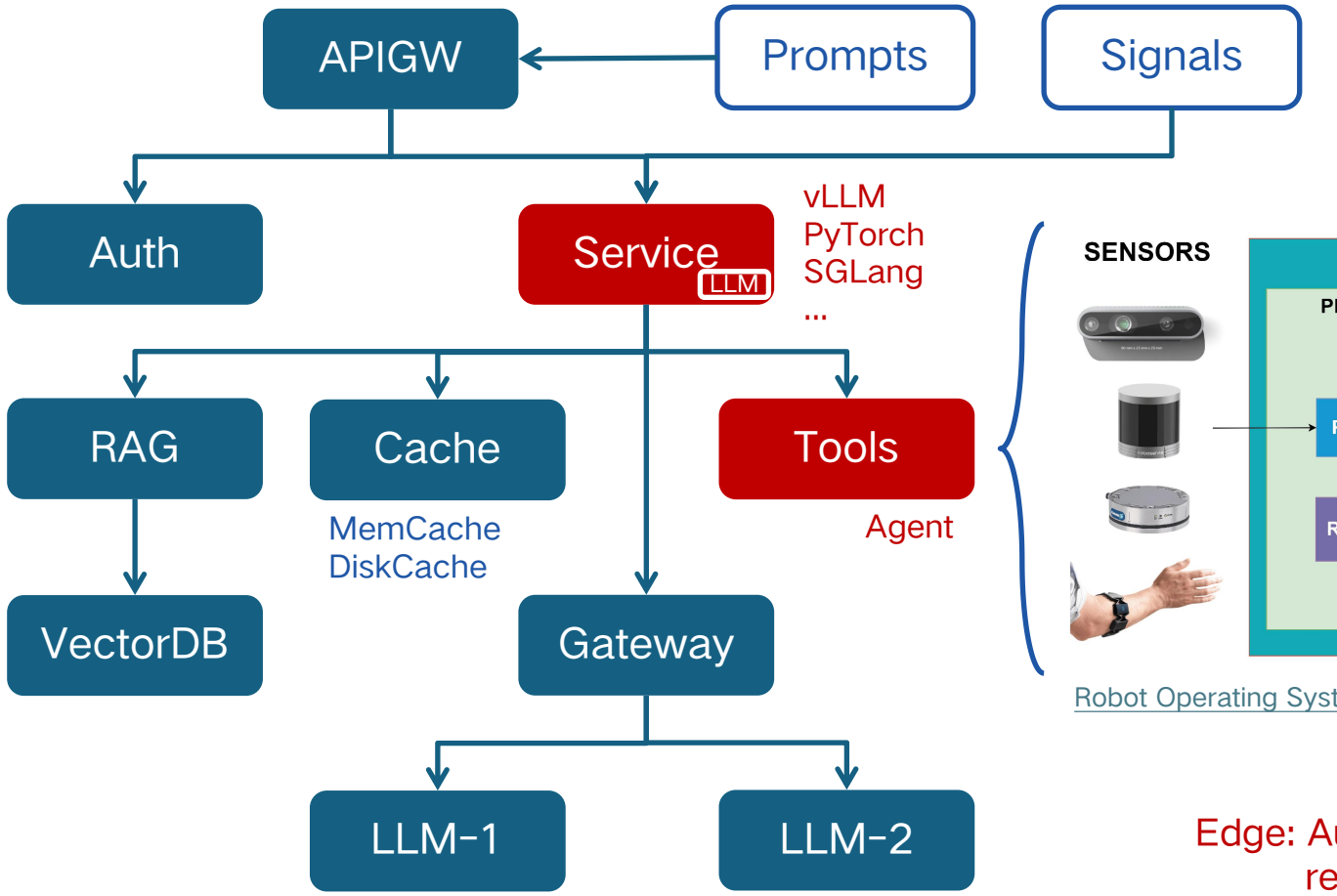
The basic story: vLLM tries to allocate as much memory as possible for KV Cache to accelerate LLM inference. In order to do so, it first profiles the memory usage, guess the maximum size of memory available for KV Cache, and also leave some for storing activation during inference.

What's more, vLLM uses cuda graph to reduce Python overhead.

When PyTorch upgrades from 2.1.2 to 2.2.0 , there seems to be some internal change of memory allocator, and the memory that can be used is decreased. It can cause OOM error during inference.

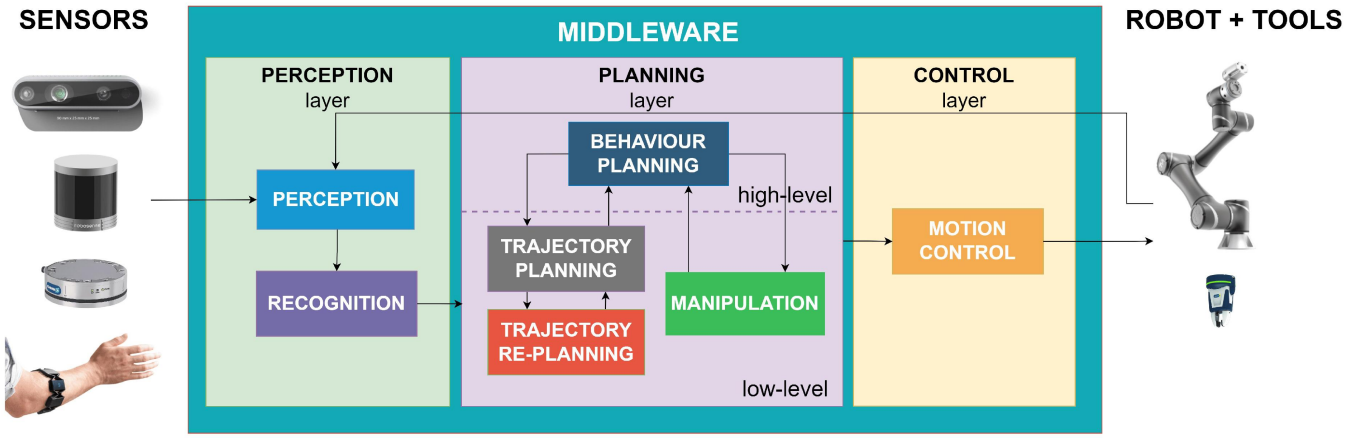
Here are the diagnoses data (produced by `torch.cuda.memory_stats` and `torch.cuda.memory._dump_snapshot`), collected from a server with 2 L4 GPUs:

From Inference Applications to Online LLM Inference Services



Cloud: Online inference is a complex distributed service
TTFT, TPOT, latency, throughput

Perception-Planning-Control
BEV-OCC-Transformer



Robot Operating System 2 (ROS2)-Based Frameworks for Increasing Robot Autonomy: A Survey

Edge: Autonomous driving and embodied intelligence (ROS2)
require end-to-end low latency and high stability

From Large Models to Small Models: Consumer-Grade GPU, CPU Collaboration

Accelerating Model Training in Multi-cluster Environments with Consumer-grade GPUs

Hwijoon Lim
KAIST
Daejeon, Republic of Korea
hwijoon.lim@kaist.ac.kr

Juncheol Ye
KAIST
Daejeon, Republic of Korea
juncheol@kaist.ac.kr

Sangeetha Abdu Jyothi
UC Irvine & VMware Research
Irvine, California, USA
sangeetha.aj@uci.edu

Dongsu Han
KAIST
Daejeon, Republic of Korea
dhan.ee@kaist.ac.kr

Accelerating Model Training in Multi-cluster Environments with Consumer-grade GPUs, SIGCOMM 2024.

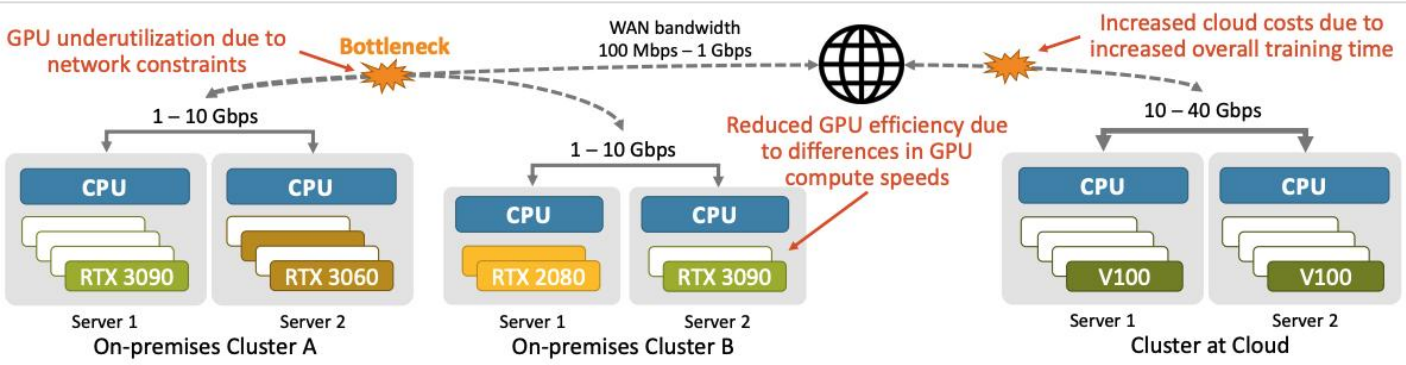


Figure 1: A multi-cluster environment with two on-premises lab clusters and a cloud cluster.

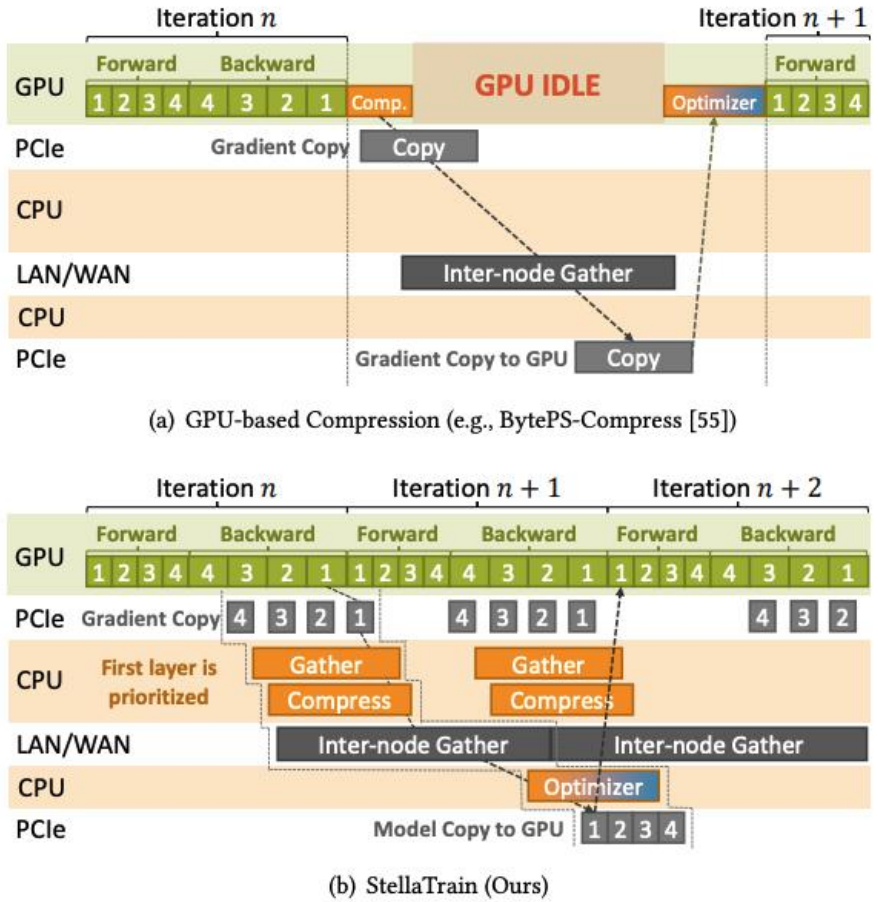
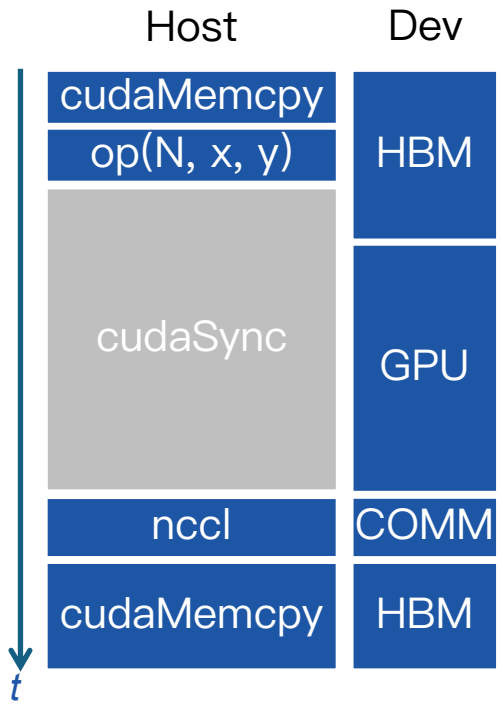
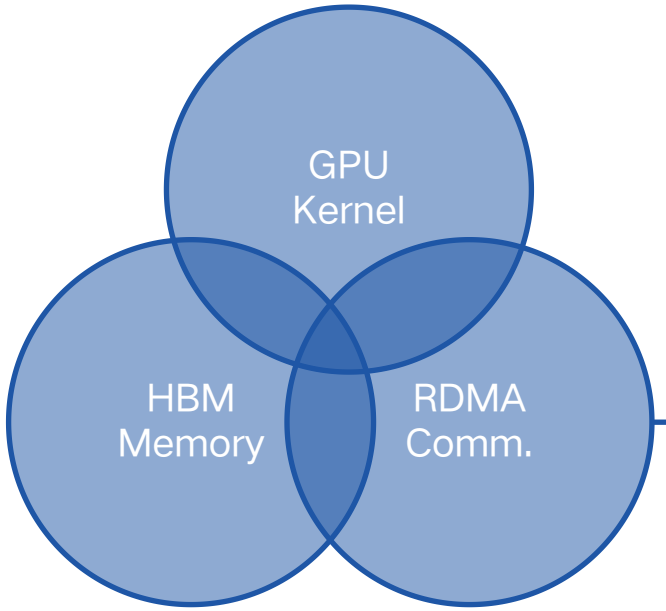


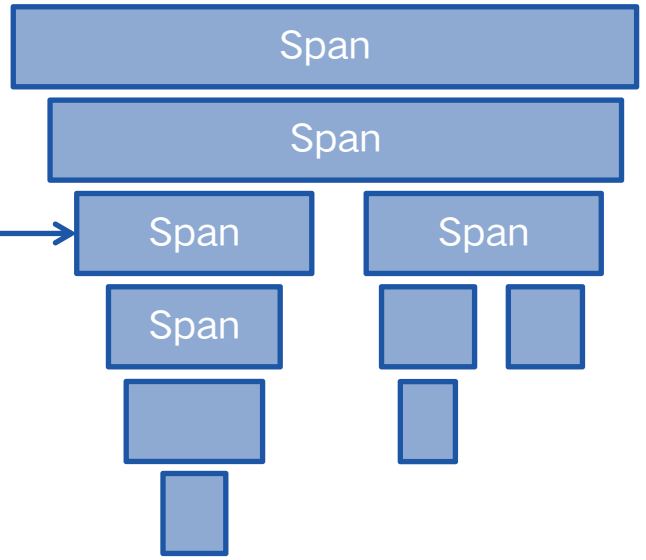
Figure 4: Comparison of training pipelines.

Observability Requirements for AI Training and Inference





Training & Inference



Online and Edge Inference Services

Gateway

Auth

3rd

ROS2 Nodes

Gateway

.....

Outline

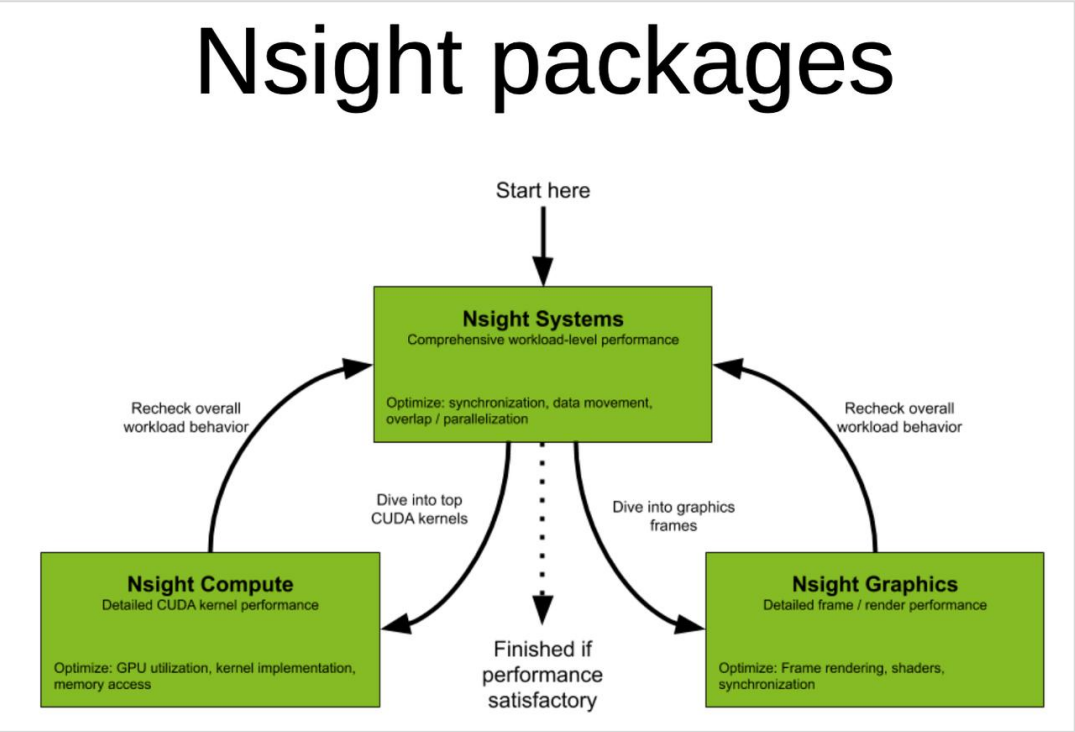
1. **Background**: Challenges in Training and Inference Efficiency
2. **Status Quo**: Issues with Traditional Solutions and Tools
3. **Approach**: Building Zero-Code Observability with eBPF
4. **Practical Case**: Full-Stack Profiling and D-Tracing in PyTorch

DCGM Prometheus Exporter

Detecting Failures √

Optimizing Performance X

GPU：Nvidia Nsight、PyTorch Profiler



Issues with Nsight:

- Requires process restart
- lack of CPU context

```python
def trace_handler(p):
    output = p.key_averages().table(sort_by="self_cuda_time_total", row_limit=10)
    print(output)
    p.export_chrome_trace("/tmp/trace_" + str(p.step_num) + ".json")

with profile(
    activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
    schedule=torch.profiler.schedule(
        wait=1,
        warmup=1,
        active=2),
    on_trace_ready=trace_handler
) as p:
    for idx in range(8):
        model(inputs)
        p.step()
```

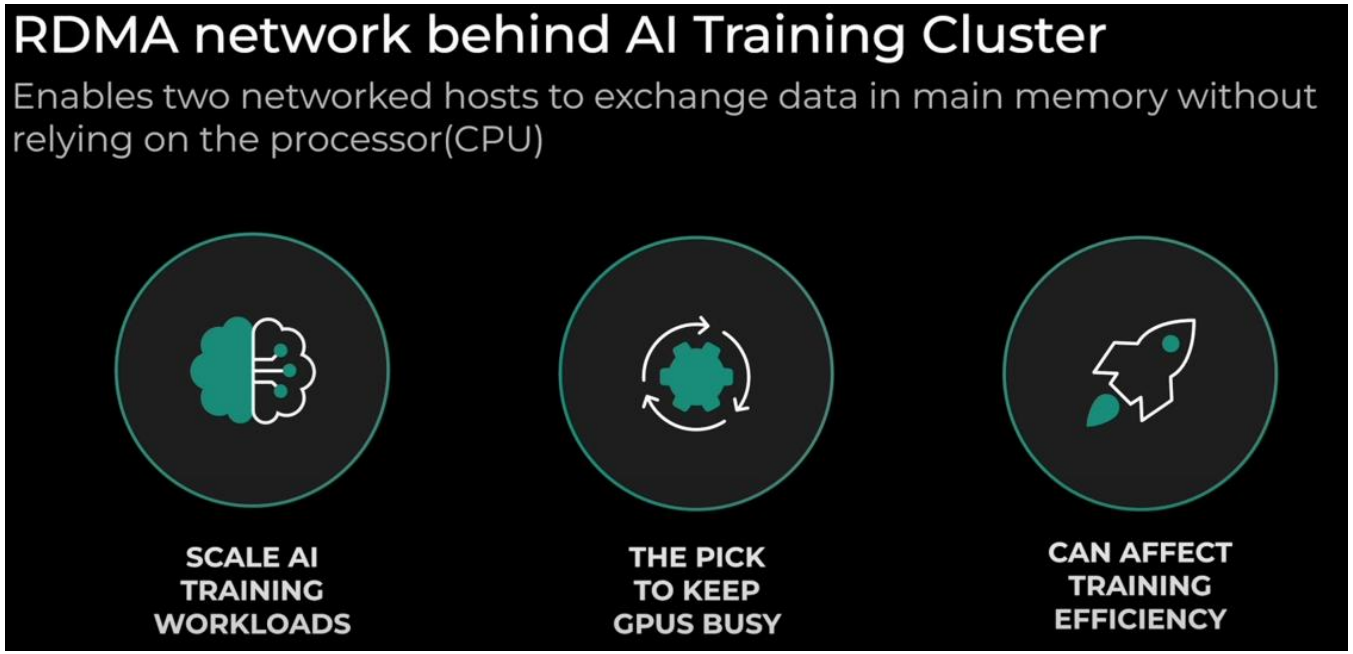
Manual Setup
Instrumentation & Overhead

1. Parameter `skip_first` tells profiler that it should ignore the first 10 steps (default value of `skip_first` is zero);
2. After the first `skip_first` steps, profiler starts executing profiler cycles;
3. Each cycle consists of three phases:
   - idling (`wait=5` steps), during this phase profiler is not active;
   - warming up (`warmup=1` steps), during this phase profiler starts tracing, but the results are discarded; this phase is used to discard the samples obtained by the profiler at the beginning of the trace since they are usually skewed by an extra overhead;
   - active tracing (`active=3` steps), during this phase profiler traces and records data;
4. An optional `repeat` parameter specifies an upper bound on the number of cycles. By default (zero value), profiler will execute cycles as long as the job runs.

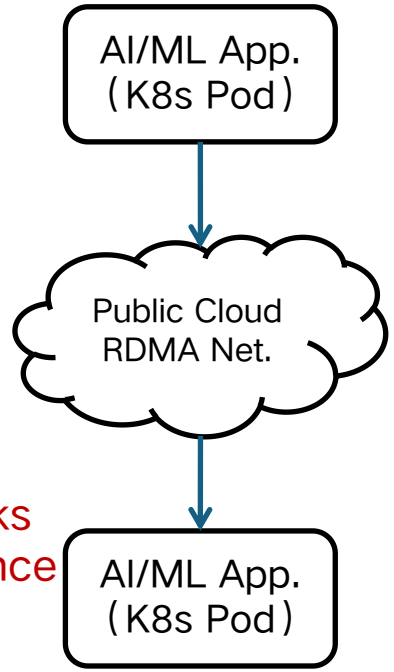
Issues with PyTorch Profiler:

- Limited to PyTorch
- Significant performance impact
- Requires code changes and process restarts

RDMA Network: NIC/Switch Metrics, Probing



Meta: Network Observability for AI/HPC Training Workflows
Private Infra.
Coarse-grained NIC/Switch metrics



Public Infra.
RDMA networks are a performance black box

北京郵電大學 Beijing University of Posts and Telecommunications | ByteDance 字节跳动 | 紫金山实验室 Purple Mountain Laboratories

R-Pingmesh: A Service-Aware RoCE Network Monitoring and Diagnostic System

Kefei Liu, Zhuo Jiang, Jiao Zhang, Shixian Guo, Xuan Zhang, Yangyang Bai, Yongbin Dong, Feng Luo, Zhang Zhang, Lei Wang, Xiang Shi, Haohan Xu, Yang Bai, Dongyang Song, Haoran Wei, Bo Li, Yongchen Pan, Tian Pan, and Tao Huang

SIGCOMM 2024

ByteDance 字节跳动 | 北京郵電大學 Beijing University of Posts and Telecommunications | 紫金山实验室 Purple Mountain Laboratories

Hostping: Diagnosing Intra-host Network Bottlenecks in RDMA Servers

Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong
Lizhuang Tan, Tian Pan and Tao Huang

NSDI 2023

ByteDance & BUPT
Focus on active probing of RDMA networks

Observability for Online Inference Services: Distributed Tracing

Python Typescript

```python
from openai import OpenAI
from traceloop.sdk import Traceloop
from traceloop.sdk.decorators import workflow

Traceloop.init(app_name="joke_generation_service")

@workflow(name="joke_creation")
def create_joke():
  client = OpenAI(api_key=os.environ["OPENAI_API_KEY"])
  completion = client.chat.completions.create(
      model="gpt-3.5-turbo",
      messages=[{"role": "user", "content": "Tell me a joke about opentelemetry"}],
  )

  return completion.choices[0].message.content
```

OpenLLMetry

python

```python
@openlit.trace
def generate_one_liner():
    completion = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[
            {
                "role": "system",
                "content": "Return a one liner from any movie for me to guess",
            }
        ],
    )
```

OpenLIT

Python TypeScript

```python
import openai
from langsmith.wrappers import wrap_openai
from langsmith import traceable

# Auto-trace LLM calls in-context
client = wrap_openai(openai.Client())

@traceable # Auto-trace this function
def pipeline(user_input: str):
    result = client.chat.completions.create(
        messages=[{"role": "user", "content": user_input}],
        model="gpt-3.5-turbo"
    )
    return result.choices[0].message.content

pipeline("Hello, world!")
# Out:  Hello there! How can I assist you today?
```

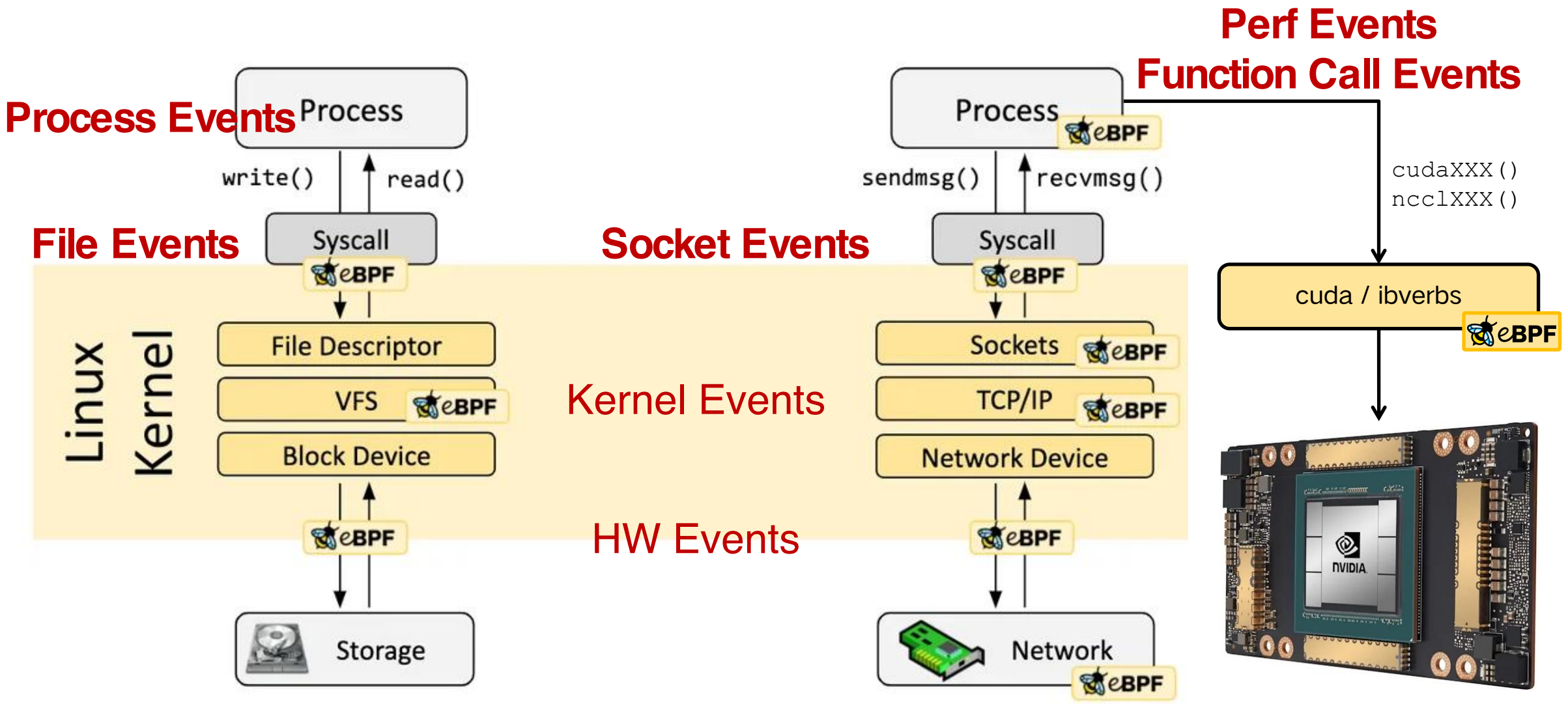
LangSmith

Limited language support, requires code modifications

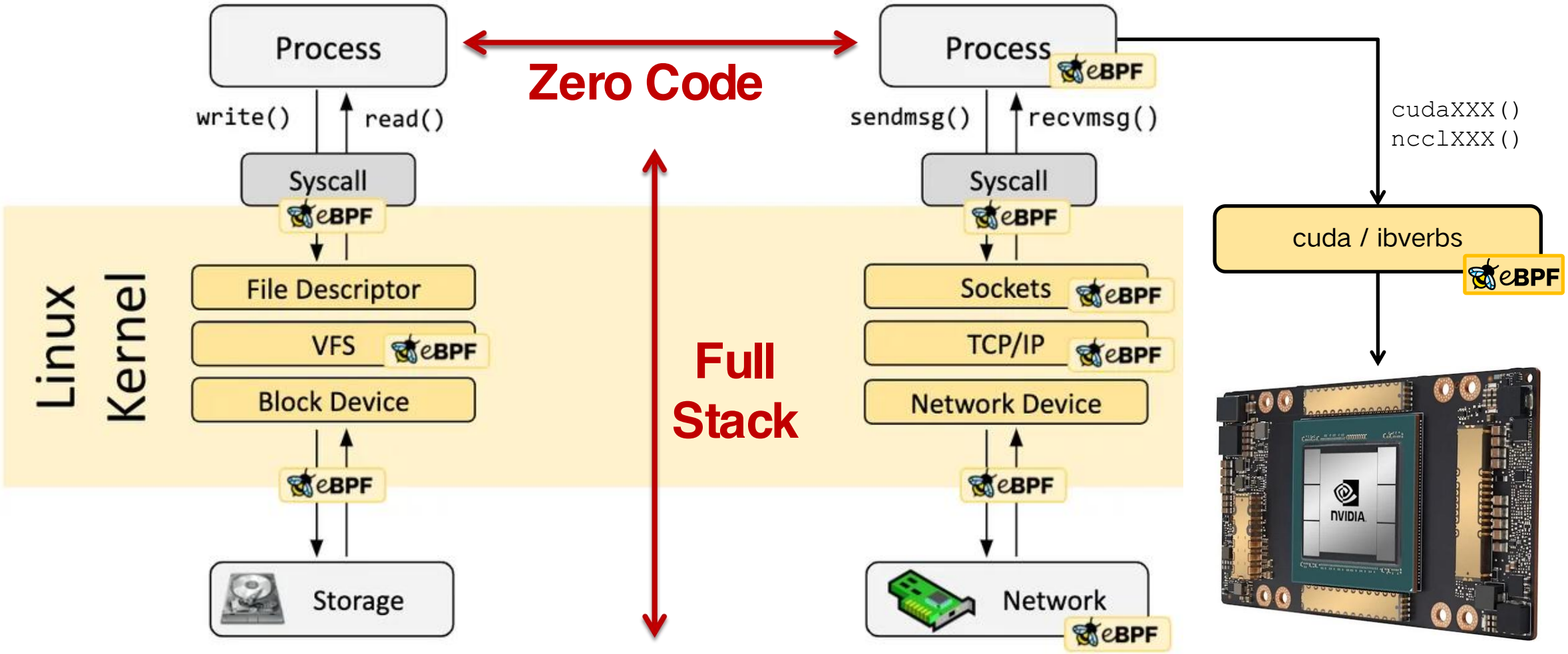
Outline

1. **Background**: Challenges in Training and Inference Efficiency
2. **Status Quo**: Issues with Traditional Solutions and Tools
3. **Approach**: Building Zero-Code Observability with eBPF
4. **Practical Case**: Full-Stack Profiling and D-Tracing in PyTorch

Observability Capabilities of eBPF



Advantages of Using eBPF for Observability



Industry Exploration: eBPF Profiling & Tracing

Meta: eBPF GPU Profiling

Trace-enabled Timing Model Synthesis for ROS2-based Autonomous Applications

Hazem Abaza*†, Debayan Roy*, Shiqing Fan‡, Selma Saidi† and Antonios Motakis*
†*Technische Universität Dortmund*, **Huawei Dresden Research Center*, ‡*Huawei Munich Research Center*
{hazem.abaza, selma.saidi}@tu-dortmund.de, {debayan.roy6, shiqing.fan, antonios.motakis}@huawei.com

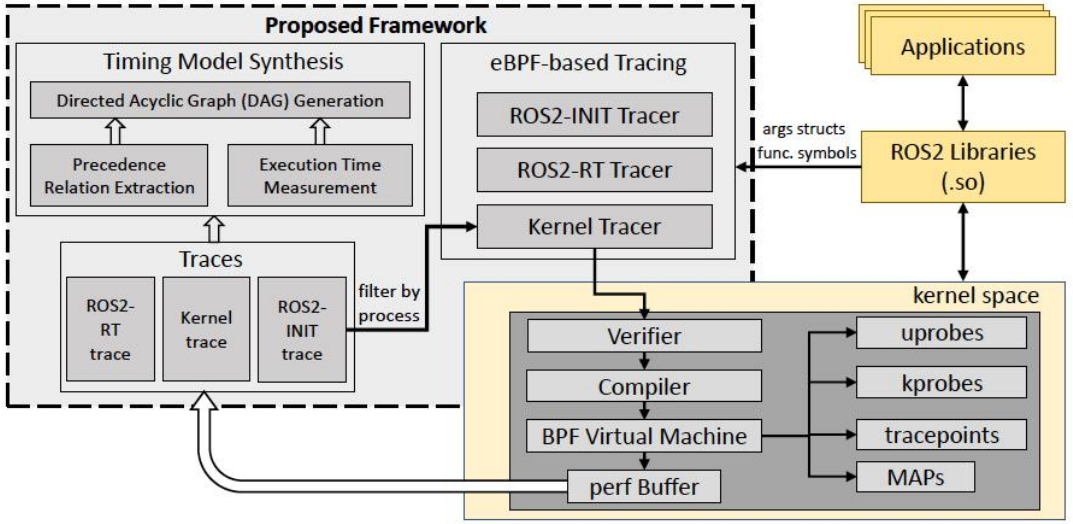
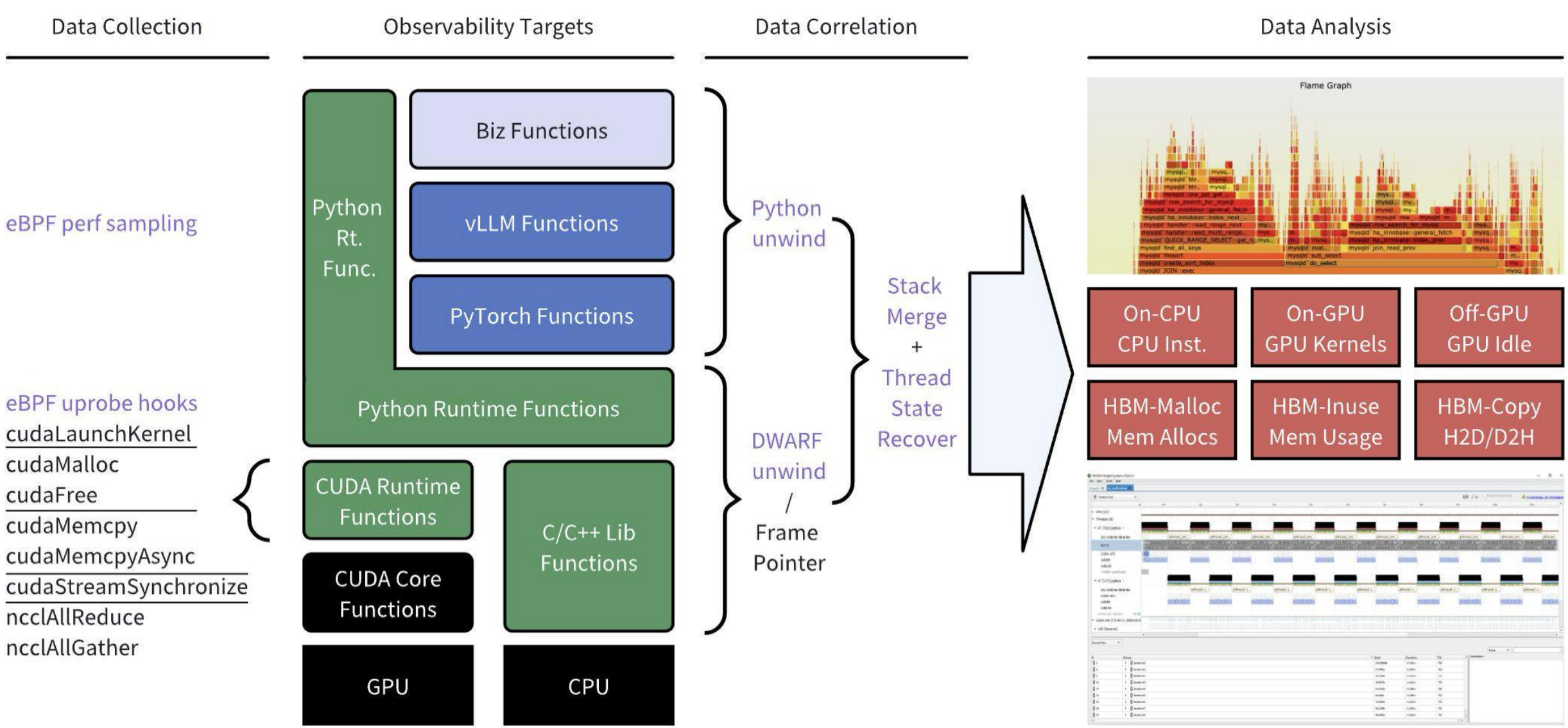


Fig. 1: Proposed trace-enabled timing model synthesis framework.

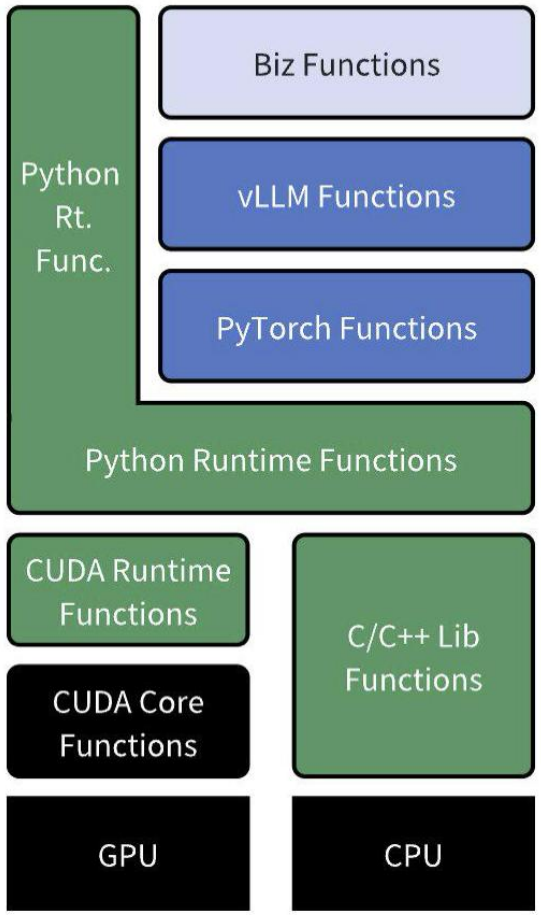
Huawei: eBPF Tracing + ROS2

Technical Challenges of Implementing Continuous Profiling with eBPF



How to Merge Python and C/C++ Stacks

Purpose of Stack Merging:
Full-Stack Profiling




```python
def baz():
    time.sleep(100)

def bar():
    baz()

def foo():
    bar()

def main():
    foo()

main()
```

```
#0  __select_nocancel ()
#1  pysleep
#2  time_sleep
#3  call_function
#4  PyEval_EvalFrameEx
#5  fast_function
#6  call_function
#7  PyEval_EvalFrameEx
#8  fast_function
#9  call_function
#10 PyEval_EvalFrameEx
#11 fast_function
#12 call_function
#13 PyEval_EvalFrameEx
#14 fast_function
#15 call_function
#16 PyEval_EvalFrameEx
#17 _PyEval_EvalCodeWithName
#18 PyEval_EvalCodeEx
#19 PyEval_EvalCode
```

https://github.com/deepflwio/deepflow

Example: Profiling Memory Allocation and Usage

① eBPF uprobe hooks cuda_malloc to capture the memory allocation call stack

```c
SEC("uprobe/cuda_malloc")
int uprobe_cuda_malloc(struct pt_regs *ctx) {
    __u64 id = bpf_get_current_pid_tgid();
    __u32 tgid = id >> 32;

    void *address = (void *) PT_REGS_PARM1(ctx);
    __u64 size = (__u64) PT_REGS_PARM2(ctx);
    malloc_data_t *data = cuda_malloc_info__lookup(&tgid);
    __u64 call_time = bpf_ktime_get_ns();

    if (data) {
        data->address = address;
        data->size = size;
        data->call_time = call_time;
        data->rip = PT_REGS_IP(ctx);
    } else {
        malloc_data_t newdata = { .address = address, .size = size, .call_time = call_time, .rip = PT_REGS_IP(ctx) };
        cuda_malloc_info__update(&tgid, &newdata);
    }

    return 0;
}
```

② eBPF uretprobe hooks cuda_malloc to capture the allocated memory address

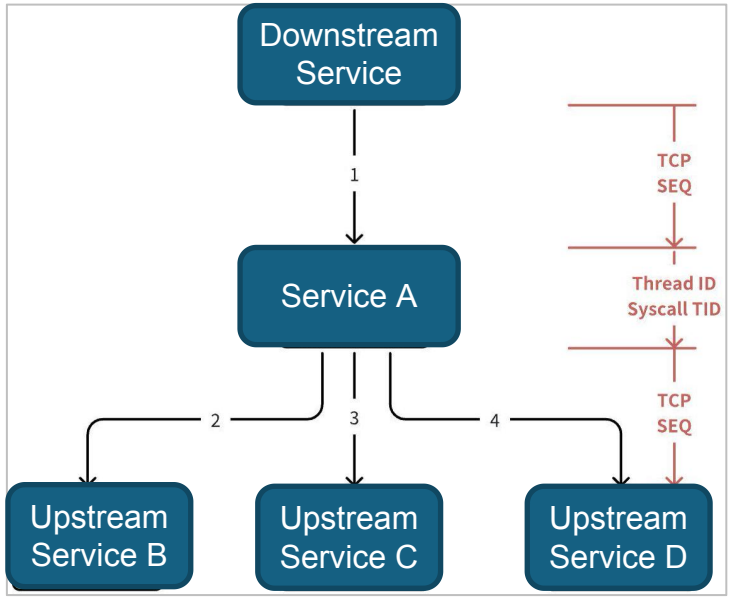
```c
SEC("uretprobe/cuda_malloc")
int uretprobe_cuda_malloc(struct pt_regs *ctx)
{
    __u64 id = bpf_get_current_pid_tgid();
    __u32 tgid = id >> 32;

    long ret = PT_REGS_RC(ctx);
    if (ret != 0) {
        return 0;
    }

    malloc_data_t *data = cuda_malloc_info__lookup(&tgid);
    if (data == NULL) {
        return 0;
    }

    cuda_malloc_info__delete(&tgid);

        __u32 zero = 0;
        unwind_state_t *state = heap__lookup(&zero);
        if (state == NULL) {
                return 0;
        }
        __builtin_memset(state, 0, sizeof(unwind_state_t));

        struct stack_trace_key_t *key = &state->key;
        key->tgid = id >> 32;
        key->pid = (__u32) id;

        /*
         * CPU idle stacks will not be collected.
         */
        if (key->tgid == key->pid && key->pid == 0) {
                return 0;
        }

        key->cpu = bpf_get_smp_processor_id();
        bpf_get_current_comm(&key->comm, sizeof(key->comm));
        key->timestamp = bpf_ktime_get_ns();

        bpf_probe_read_user(&key->mem_addr, sizeof(__u64), (void *)data->address);
        key->mem_size = (__u64) data->size;

    // add one frame for cudaMalloc
    // native unwinding start from uretprobe, which has the reg state after cudaMalloc return
    add_frame(&state->stack, data->rip);

        state->regs.ip = PT_REGS_IP(ctx);
        state->regs.sp = PT_REGS_SP(ctx);
        state->regs.bp = PT_REGS_FP(ctx);
    add_frame(&state->stack, state->regs.ip);
    bpf_tail_call(ctx, &NAME(progs_jmp_uprobe_map), PROG_DWARF_UNWIND_IDX);

    return 0;
}
```

③ eBPF uprobe hooks cuda_free to capture the freed memory address

④ Calculate current memory usage

```c
SEC("uprobe/cuda_free")
int uprobe_cuda_free(struct pt_regs *ctx) {
    __u64 id = bpf_get_current_pid_tgid();

        void *addr = (void *) PT_REGS_PARM1(ctx);

        __u32 zero = 0;
        unwind_state_t *state = heap__lookup(&zero);
        if (state == NULL) {
                return 0;
        }
        __builtin_memset(state, 0, sizeof(unwind_state_t));

        struct stack_trace_key_t *key = &state->key;
        key->tgid = id >> 32;
        key->pid = (__u32) id;

        /*
         * CPU idle stacks will not be collected.
         */
        if (key->tgid == key->pid && key->pid == 0) {
                return 0;
        }

        key->cpu = bpf_get_smp_processor_id();
        bpf_get_current_comm(&key->comm, sizeof(key->comm));
        key->timestamp = bpf_ktime_get_ns();

        key->mem_addr = (__u64) addr;

        bpf_perf_event_output(ctx, &NAME(cuda_memory_output), BPF_F_CURRENT_CPU, &state->key, sizeof(state->key));

    return 0;
}
```

Technical Challenges of Implementing Distributed Tracing with eBPF



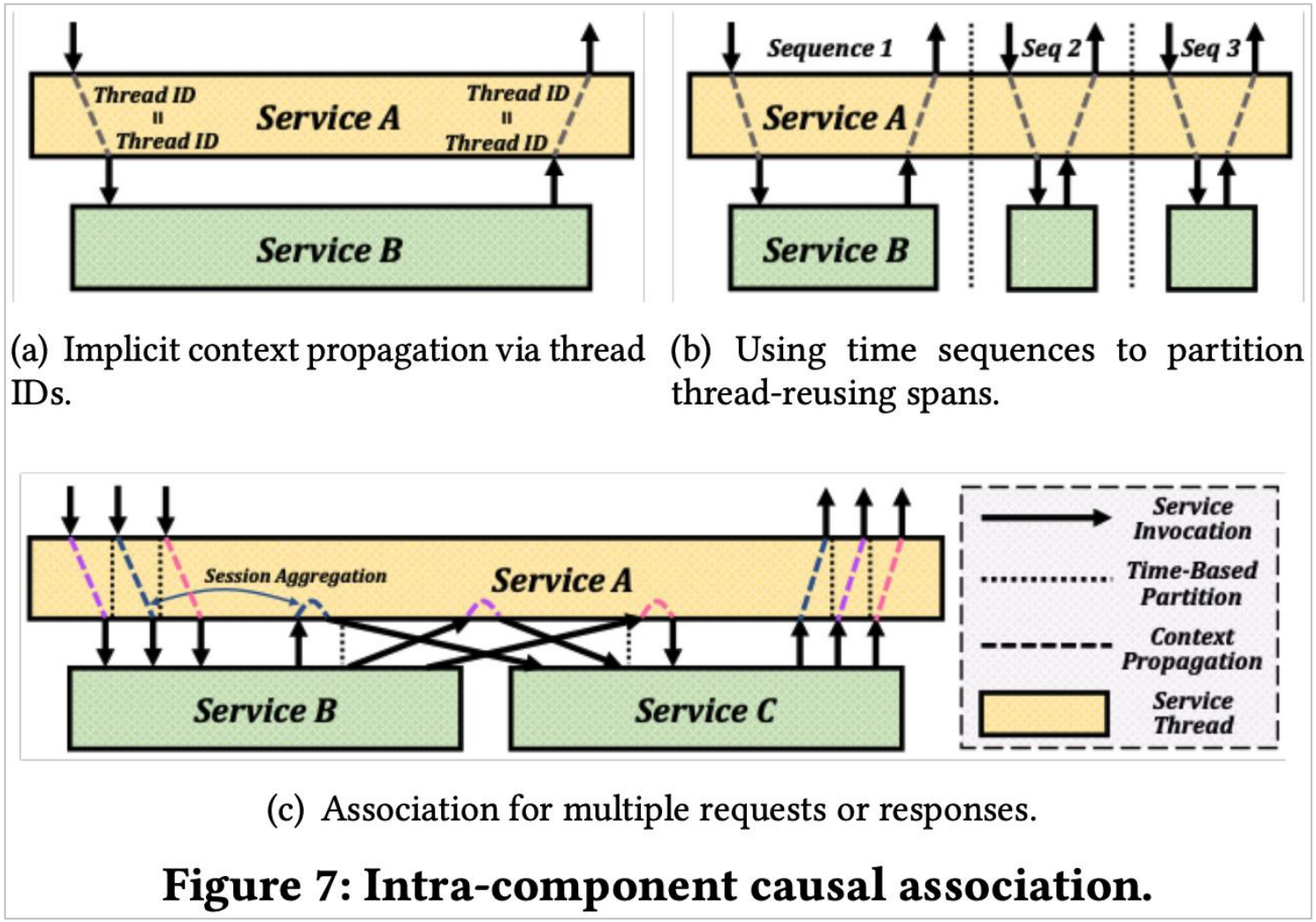
Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code

Junxian Shen[1 2 °], Han Zhang[1 4 •], Yang Xiang[2 ⋆], Xingang Shi[1 4], Xinrui Li[3], Yunxi Shen[3], Zijian Zhang[1 2], Yongxiang Wu[3], Xia Yin[3 4], Jilong Wang[1 4], Mingwei Xu[1 4], Yahui Li[1], Jiping Yin[2], Jianchang Song[2], Zhuofeng Li[2], Runjie Nie[2]
[1]Institute for Network Sciences and Cyberspace, Tsinghua University
[2]Yunshan Networks
[3]Department of Computer Science and Technology, Tsinghua University
[4]Zhongguancun Laboratory
°shenjx22@mails.tsinghua.edu.cn •zhhan@tsinghua.edu.cn ⋆xiangyang@yunshan.net

ABSTRACT

Microservices are becoming more complicated, posing new challenges for traditional performance monitoring solutions. On the one hand, the rapid evolution of microservices places a significant

Mingwei Xu[1 4], Yahui Li[1], Jiping Yin[2], Jianchang Song[2], Zhuofeng Li[2], Runjie Nie[2]. 2023. Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3603269.3604823



(a) Implicit context propagation via thread IDs.

(b) Using time sequences to partition thread-reusing spans.

(c) Association for multiple requests or responses.

Figure 7: Intra-component causal association.

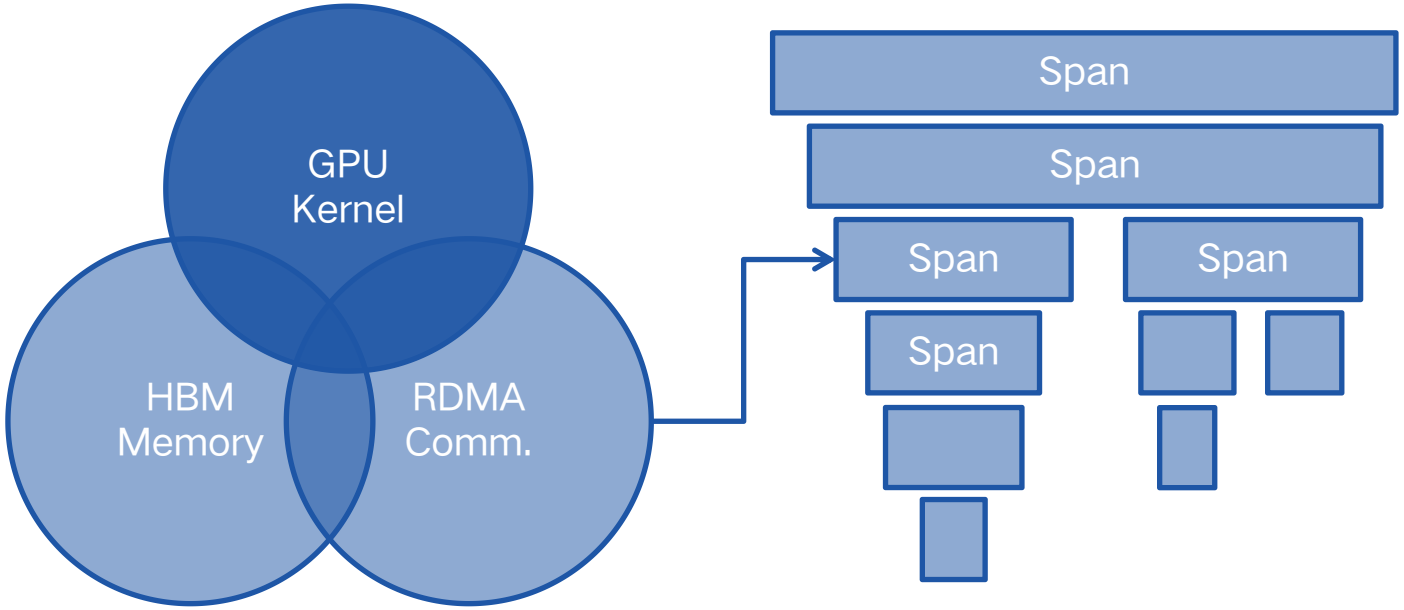
Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code

Outline

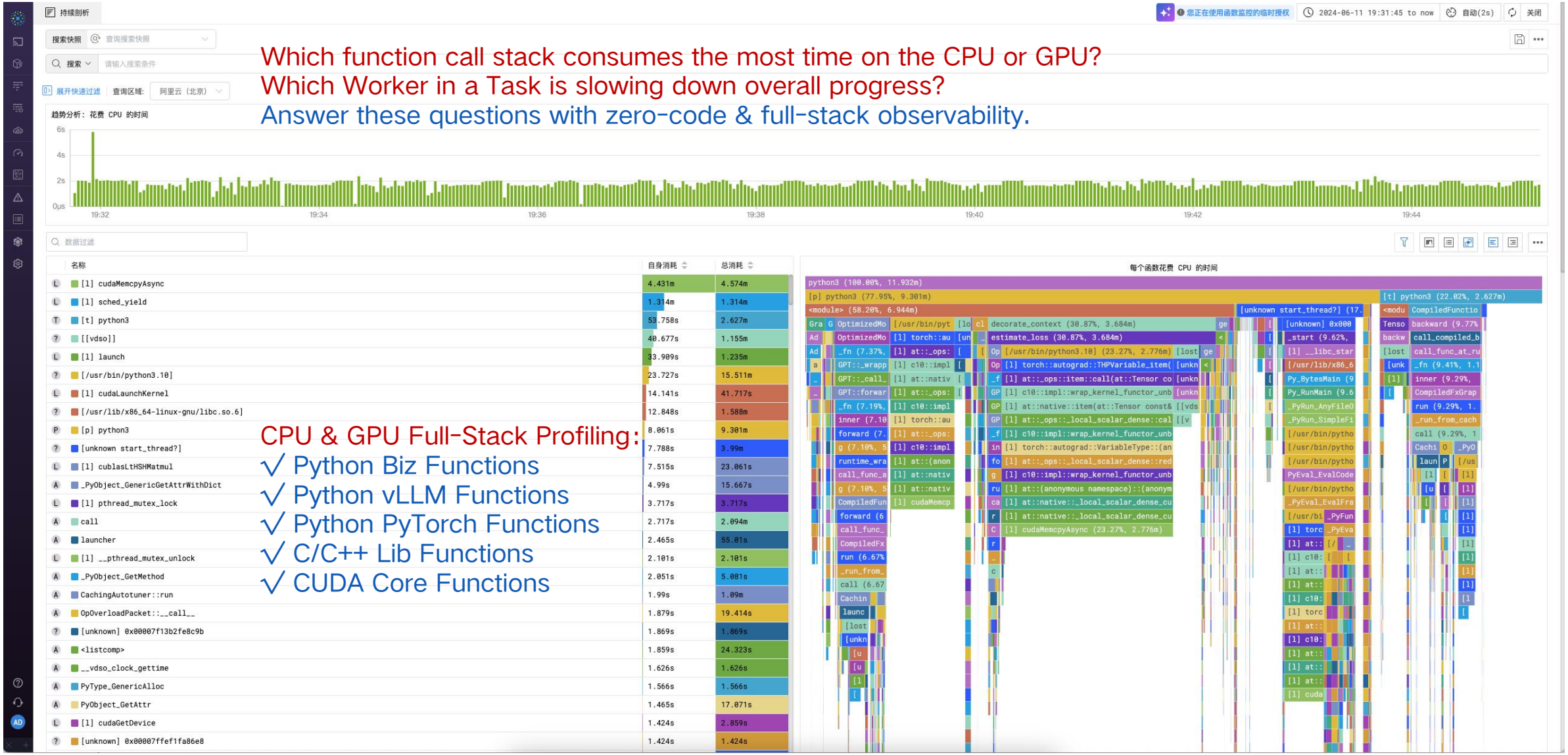
1. **Background**: Challenges in Training and Inference Efficiency
2. **Status Quo**: Issues with Traditional Solutions and Tools
3. **Approach**: Building Zero-Code Observability with eBPF
4. **Practical Case**: Full-Stack Profiling and D-Tracing in PyTorch

eBPF AutoProfiling in DeepFlow

1. GPU Profiling



PyTorch + nanoGPT: Full-Stack (CPU & GPU) Flame Graph



CPU & GPU Full-Stack Profiling:

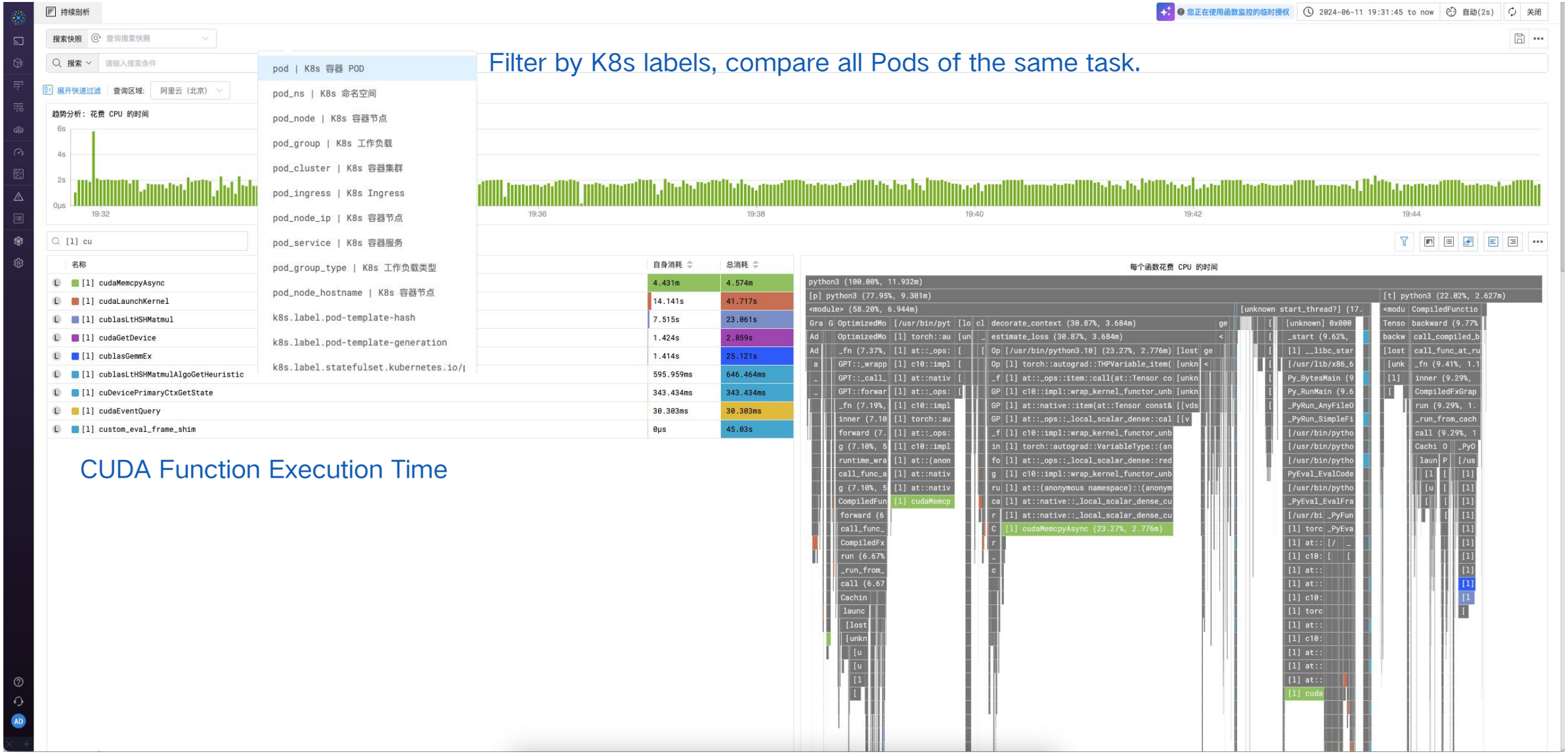
- √ Python Biz Functions
- √ Python vLLM Functions
- √ Python PyTorch Functions
- √ C/C++ Lib Functions
- √ CUDA Core Functions

Which function call stack consumes the most time on the CPU or GPU?

Which Worker in a Task is slowing down overall progress?

Answer these questions with zero-code & full-stack observability.

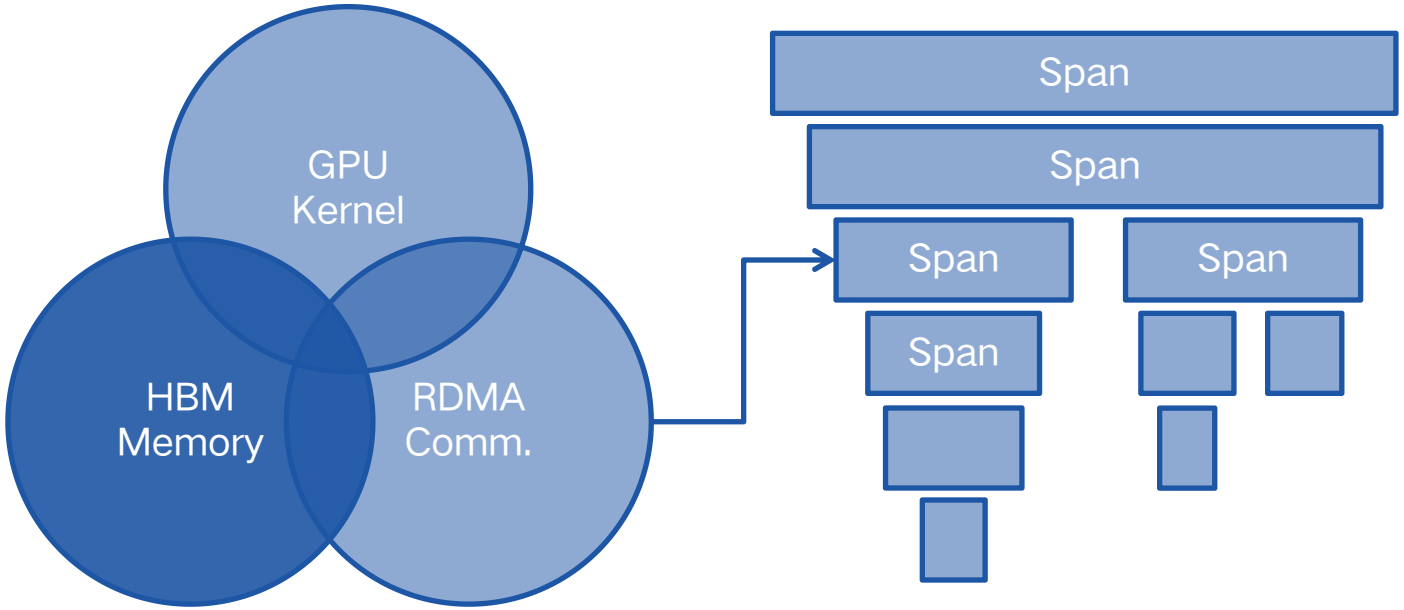
PyTorch + nanoGPT: Full-Stack (CPU & GPU) Flame Graph



Filter by K8s labels, compare all Pods of the same task.

CUDA Function Execution Time

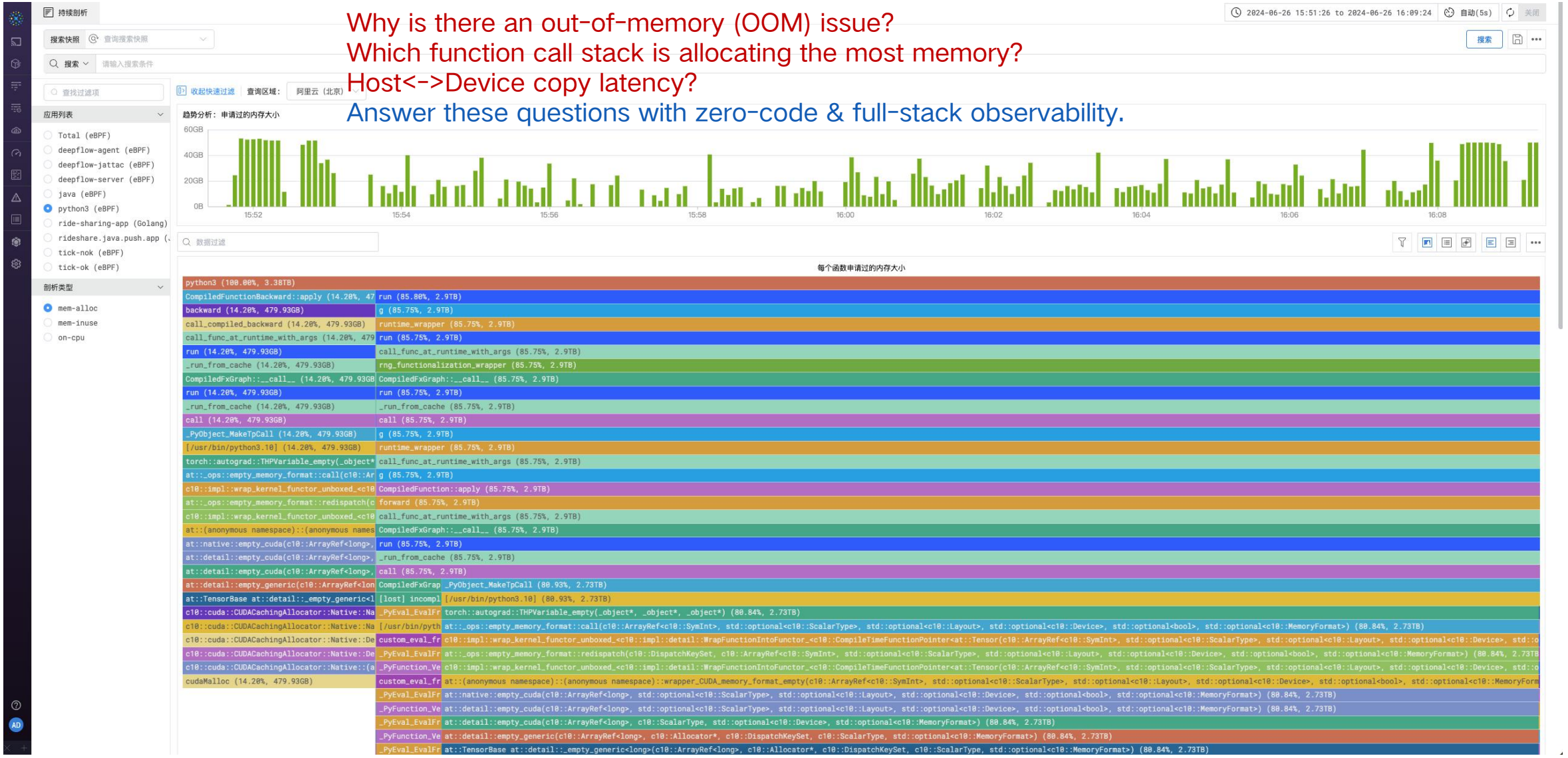
2. HBM Profiling



CUDA mem-alloc: Memory Allocation Flame Graph

KubeCon CloudNativeCon China 2024 | THE LINUX FOUNDATION OPEN SOURCE SUMMIT | AI_dev

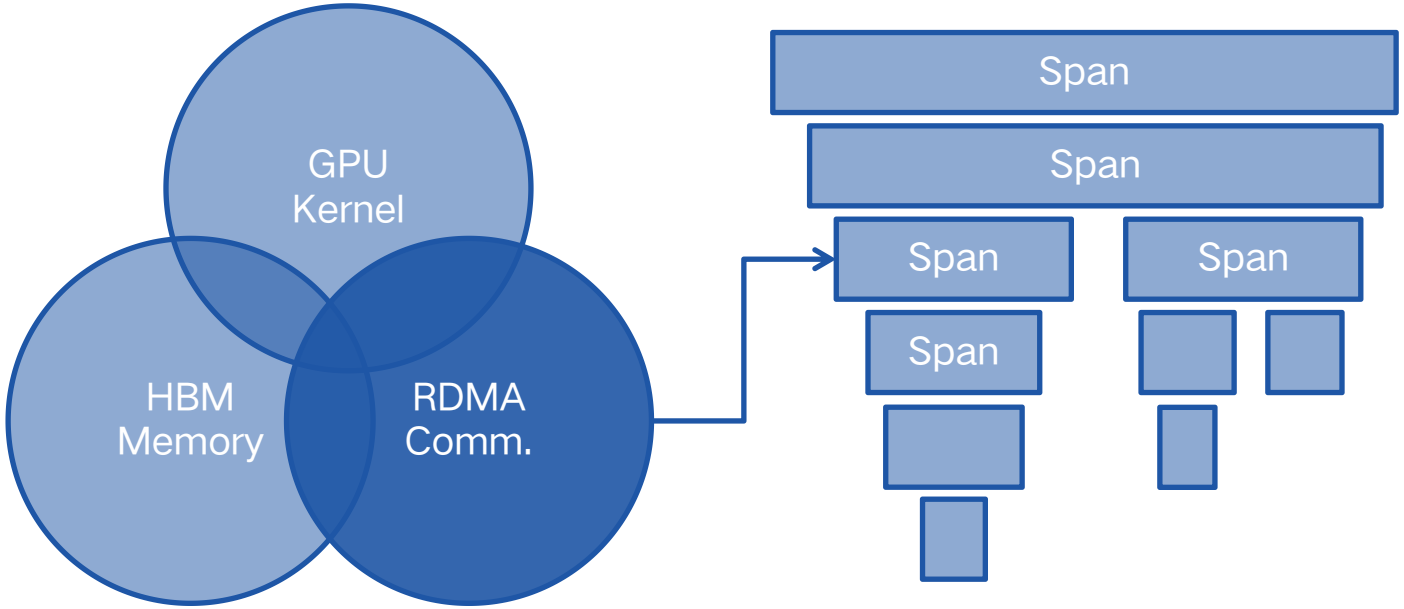
Why is there an out-of-memory (OOM) issue?
Which function call stack is allocating the most memory?
Host<->Device copy latency?
Answer these questions with zero-code & full-stack observability.



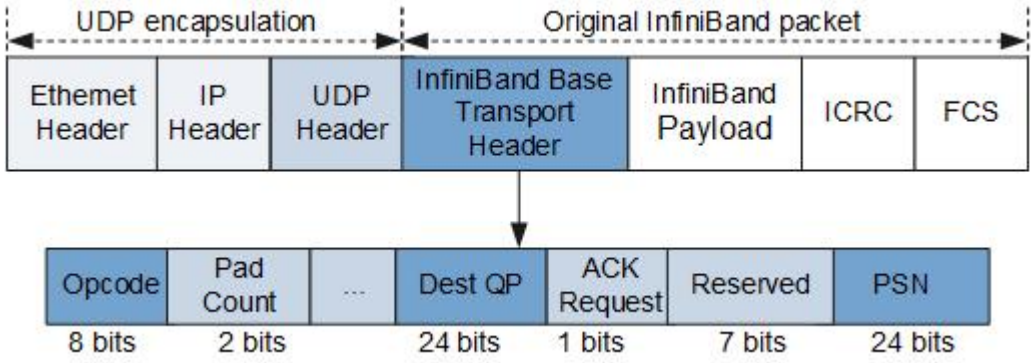
CUDA mem-inuse: Memory Usage Flame Graph

Why is there an out-of-memory (OOM) issue?
Which function call stack is allocating the most memory?
Host<->Device copy latency?
Answer these questions with zero-code & full-stack observability.

3. COMM. Profiling



RDMA Network Performance Profiling



RoCEv2 Packet

Key labels:

- Client and its associated K8s Pod and labels
- Server and its associated K8s Pod and labels
- Client Queue Pair
- Server Queue Pair

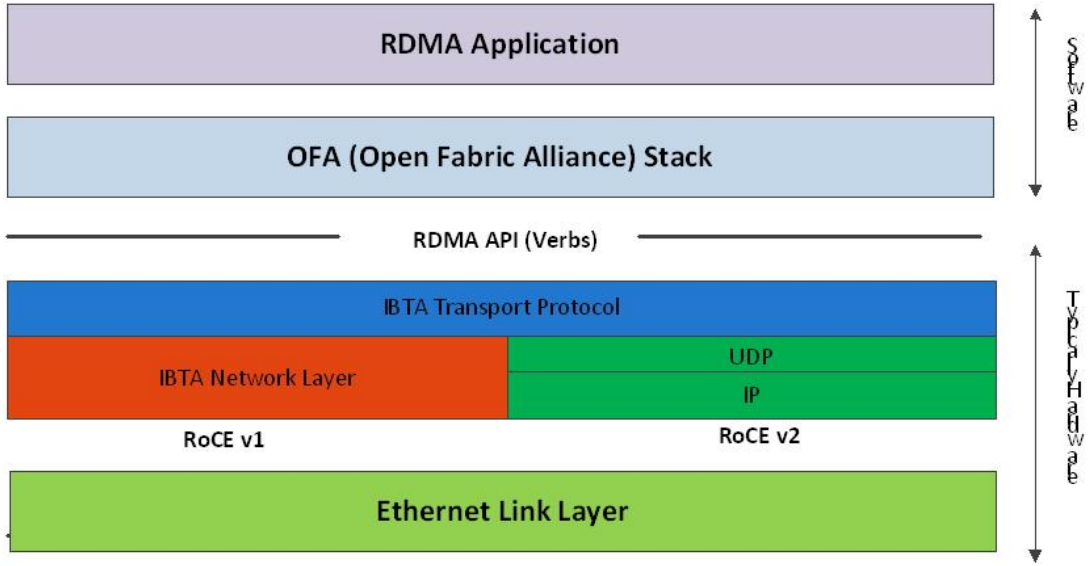
Key metrics:

- Packet loss rate: Proportion of NACKs
- Latency: ACK latency
- Throughput: bps and pps of communication pairs

eBPF Hooks

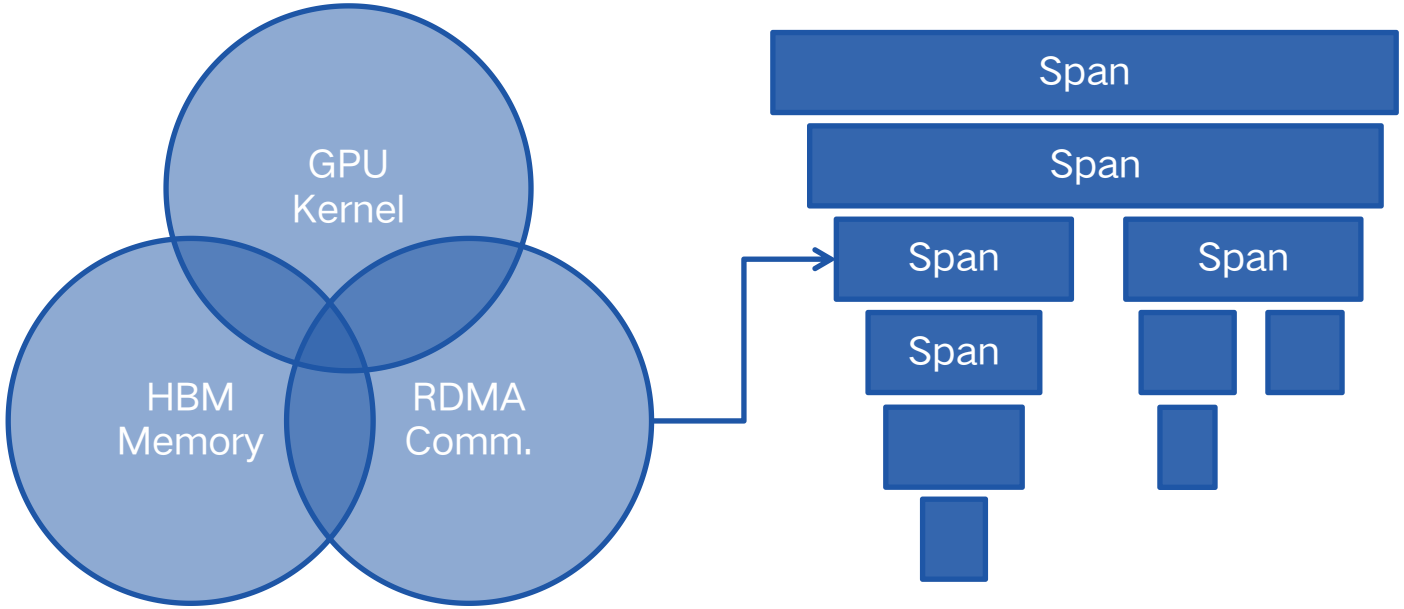
Function	Explanation
ibv_modify_qp	Queue Pair state modify
ibv_reg_mr	Memory Region registration
ibv_dereg_mr	Memory Region deregistration
ibv_post_send	Buffer send or RDMA read/write
ibv_poll_cq	Waiting for wr completion

libibverbs Hooks, NCCL Hooks, PyTorch Hooks, ...

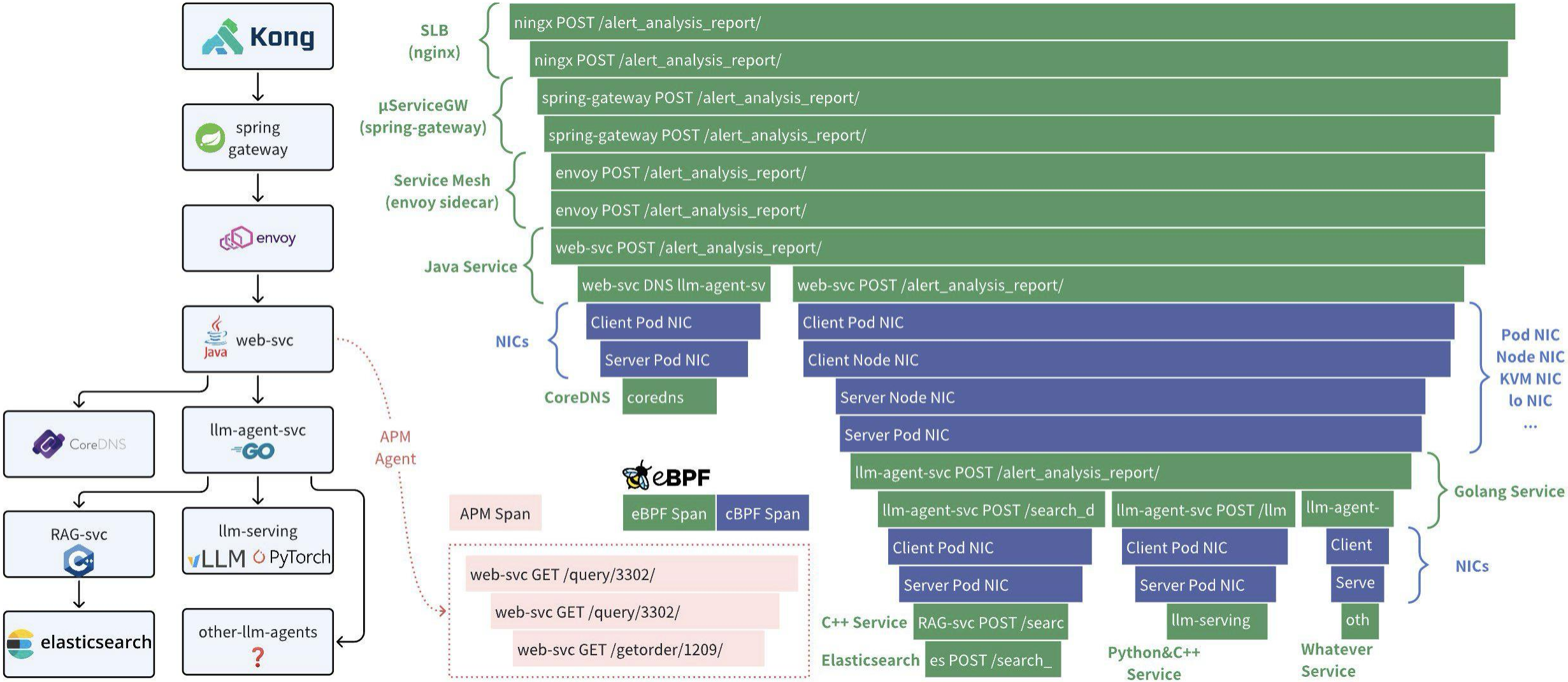


Use eBPF to hook RDMA APIs

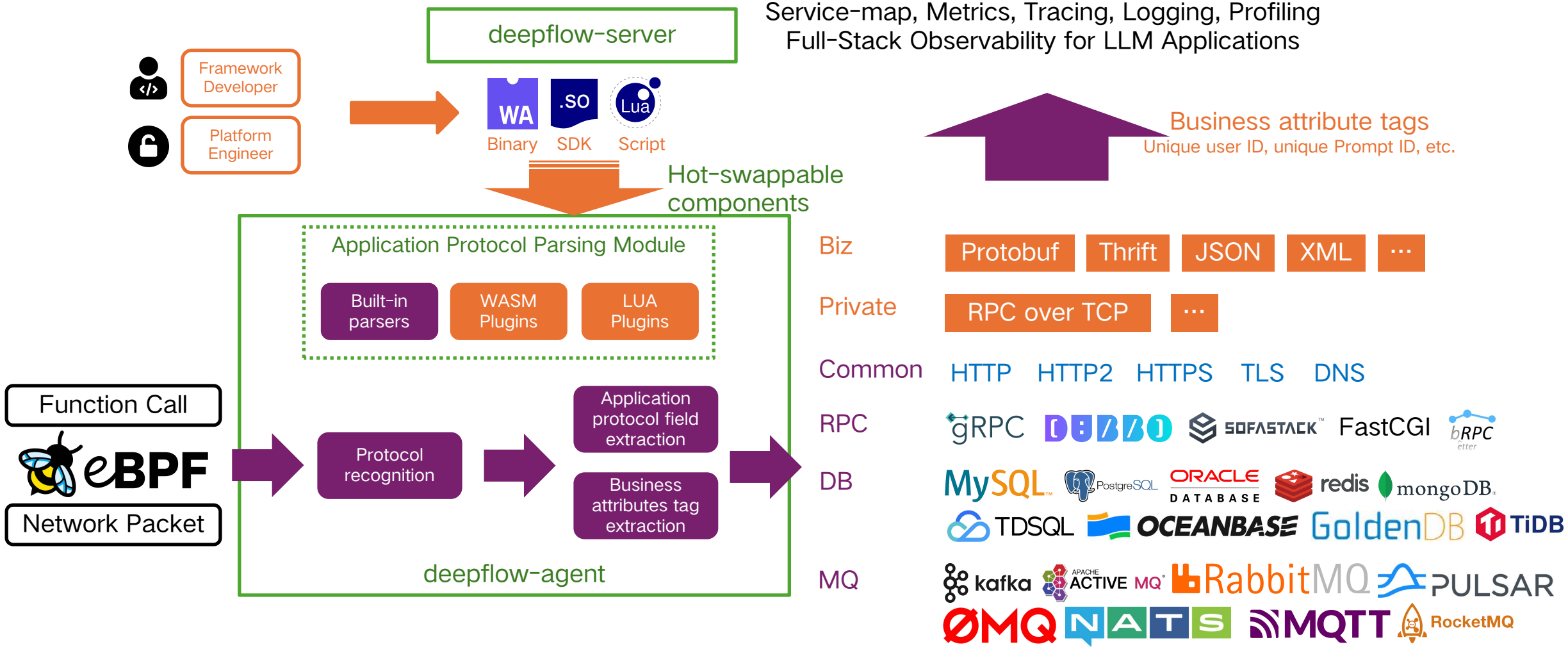
4. Distributed Tracing



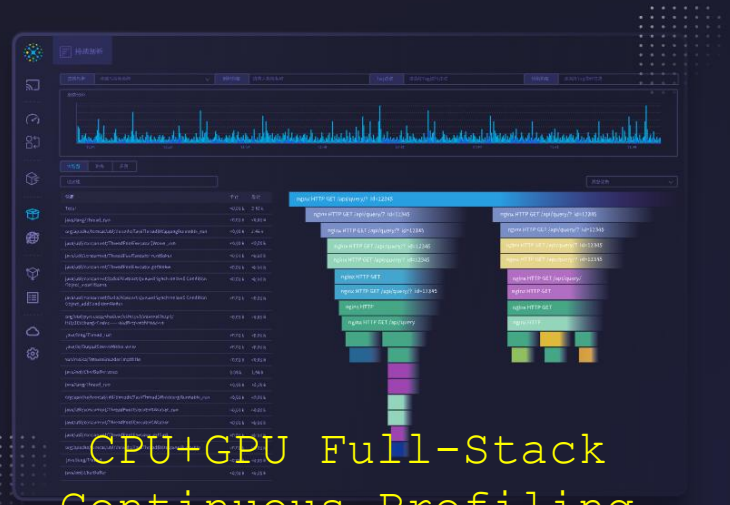
Cloud/Edge Online Inference Services



Built-In and Programmable Protocol Recognition Capabilities



DeepFlow: Zero-Code & Full-Stack Observability for AI Infra & Applications



CPU+GPU Full-Stack
Continuous Profiling

Collective Communication
Network Performance Monitoring

Distributed Tracing for
Online Inference Services

Prometheus
Open Telemetry
SKYwalking
Pyroscope
VECTOR BY DATADOG

AS SINK

DeepFlow

https://deepflow.io

AS SOURCE

Grafana
Prometheus
Open Telemetry
SKYwalking
kafka

eBPF
Wasm

No Code Changes, No Process Restarts Required