



2024 AI+研发数字峰会

AI+ Development Digital summit

AI驱动研发变革 促进企业降本增效

北京站 08/16-17

基于 eBPF 和 Agent 构建 LLM 训练推理优化体系

向阳 云杉网络

科技生态圈峰会 + 深度研习



—1000+ 技术团队的选择



上海站

K+全球软件研发行业创新峰会

时间: 2024.06.21-22



敦煌站

K+思考周®研习社

时间: 2024.10.17-19



香港站

K+思考周®研习社

时间: 2024.11.10-12



K+峰会详情



上海站

Ai+研发数字峰会

时间: 2024.05.17-18



北京站

Ai+研发数字峰会

时间: 2024.08.16-17



深圳站

Ai+研发数字峰会

时间: 2024.11.08-09



AiDD峰会详情



2024 AI+研发数字峰会

AI+ Development Digital summit

深圳站 11/08-09

AI 驱动研发变革 促进企业降本增效

2024深圳站-议题设置

AI+产品线	LLM驱动产品创新	LLM驱动需求与业务分析	AI驱动设计与用户体验
AI+开发线	AI 原生应用开发框架与技术	AI Agents在研发落地实践	LLM驱动编程与单测
AI+测试线	LLM驱动测试分析与设计	基于LLM生成测试脚本与数据	LLM和AI应用的评测
AI+工程线	AI+DevOps 与工具 (LLM 时代的平台工程)	大模型对齐与安全	端侧大模型与云端协同
AI+领域线	领域大模型 SFT 与优化	知识增强与数据智能	大厂专场

扫描右侧二维码
查看更多会议详情



早鸟票限时抢购中 (截止到9月30日)

¥3680

早鸟票

¥2800

学生票



向阳

清华大学博士 / 云杉网络研发 VP

清华大学博士，云杉网络研发 VP，DeepFlow 开源社区负责人。曾在国际顶级学术会议 ACM SIGCOMM、ACM IMC 上发表可观测性方向的学术论文，现负责可观测性产品 DeepFlow，致力于打造一款为云原生和 AI 应用而生的零侵扰可观测性产品。

目录

CONTENTS

1. 背景：训练和推理的效率挑战
2. 现状：传统解决方案和工具的问题
3. 方法：eBPF 构建零侵扰可观测性
4. 实践：PyTorch 全栈剖析和追踪
5. 探索：Agent 自动优化 ML 代码

PART 01

背景：训练和推理的效率挑战

▶ LLM 训练开销大、效率低

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

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

训练时间长：数月

GPU 数量多：数万

模型参数大：万亿

GPU 利用率低：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.

$$148/54 * 365 / 16384 = 6\%$$

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

GPU 年化故障高：
6%~11%

▶ 代码层面训练低效的主要原因

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.](#)

如何知晓你的训练任务是否存在这些问题?

计算效率

```
1 from torch.utils.data import DataLoader
2
3 train_batch_size = 384
4 eval_batch_size = 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).

显存拷贝

```
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.

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 parameter⁵ 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.

```
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                fl, pred = evaluator.evaluate(val_loader, model, step)
21                ...
22                if fl > max_f1:
23                    max_f1 = fl
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,
33                    remote_path]).start()
34
35 if __name__ == '__main__':
36     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).

网络传输

LLM 推理开销大、时延高

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

LLM Memory Requirements

Inference Training

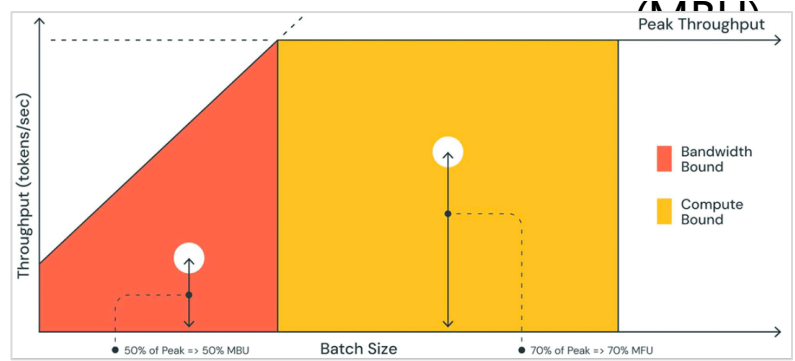
Total Inference Memory: 1.48 TB

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

- [LLM Memory Requirements](#)
- [LLM Inference Performance Engineering: Best Practices](#)

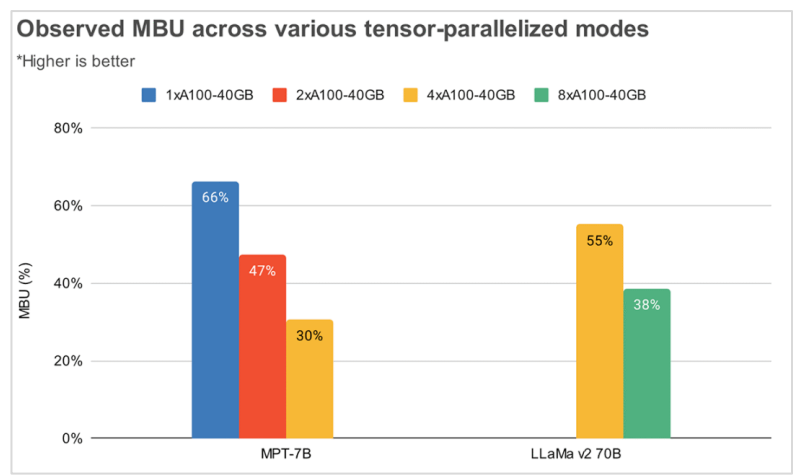
80GB: 1 GPU
 640GB: 1 Node x 8 GPU
 1.28TB: 2 Node x 8 GPU

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



GPU 并非越少越好

GPU 越少，则每个 GPU 需要加载更多的模型参数。




GPU 并非越多越好

GPU 越多，则通信越复杂，内存碎块越多。


没有银弹，唯有持续观测 & 优化。

▶ 排查 LLM 推理显存消耗的挑战

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

Sign in

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.

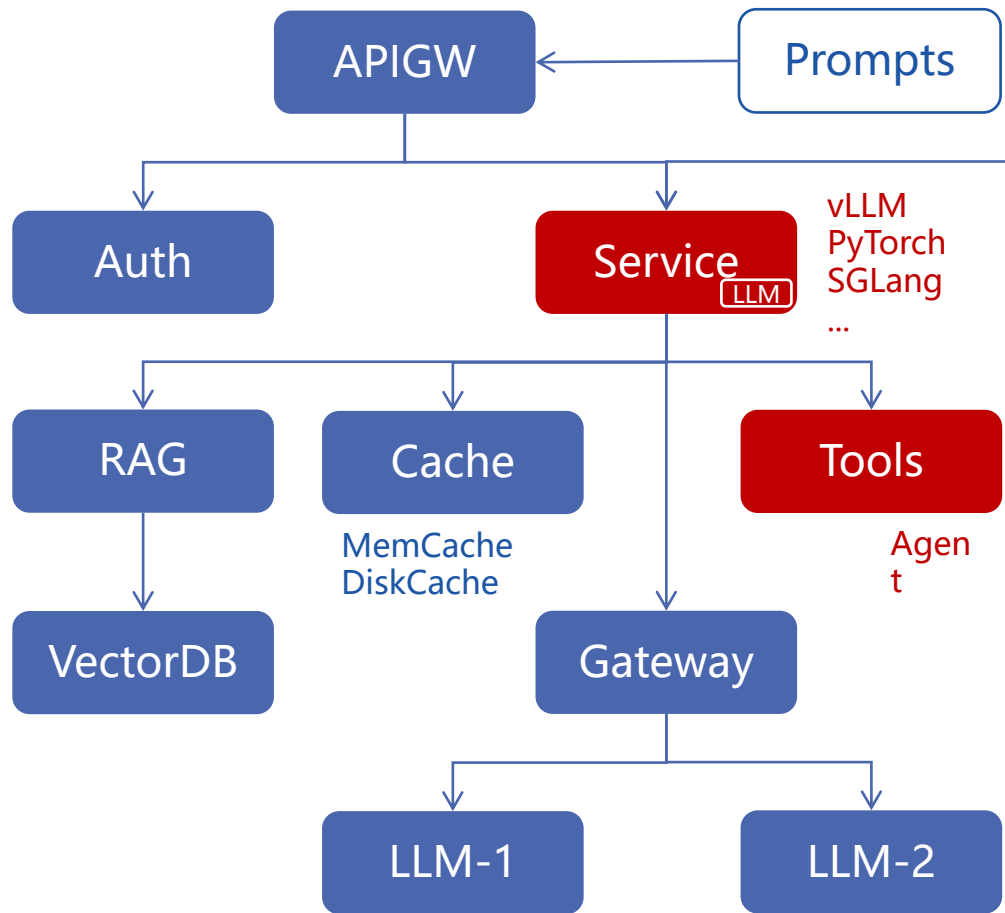
The basic story: vLLM tries to allocate as much memory as possible for KV Cache to accelerate LLM inference. In order to do so, it first profiles the memory usage, guess the maximum size of memory available for KV Cache, and also leave some for storing activation during inference.

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.

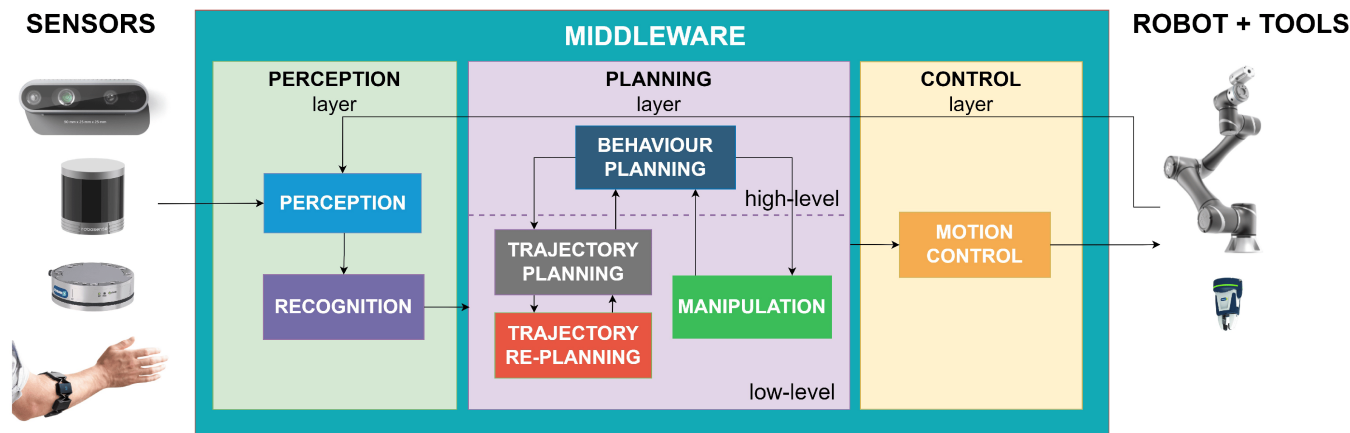
Here are the diagnoses data (produced by `torch.cuda.memory_stats` and `torch.cuda.memory._dump_snapshot`), collected from a server with 2 L4 GPUs:

从推理应用到在线 LLM 推理服务



云 - 在线推理服务是一个复杂的分布式服务
TTFT、TPOT、时延、吞吐

Perception-Planning-Control BEV-OCC-Transformer



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

端 - 自动驾驶、具身智能 (ROS2) 的端到端
低时延和高稳定性要求

从大模型到小模型：消费级 GPU、CPU 协同

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

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

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

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

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.

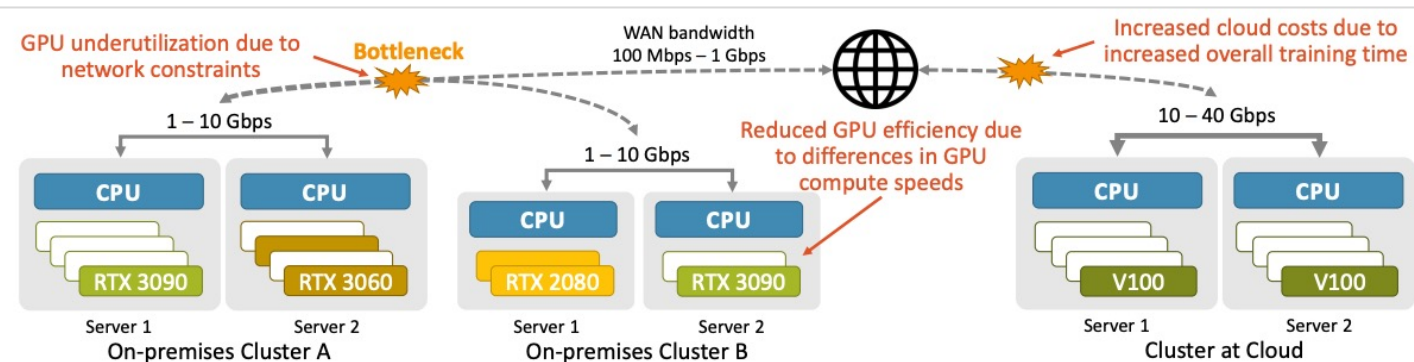
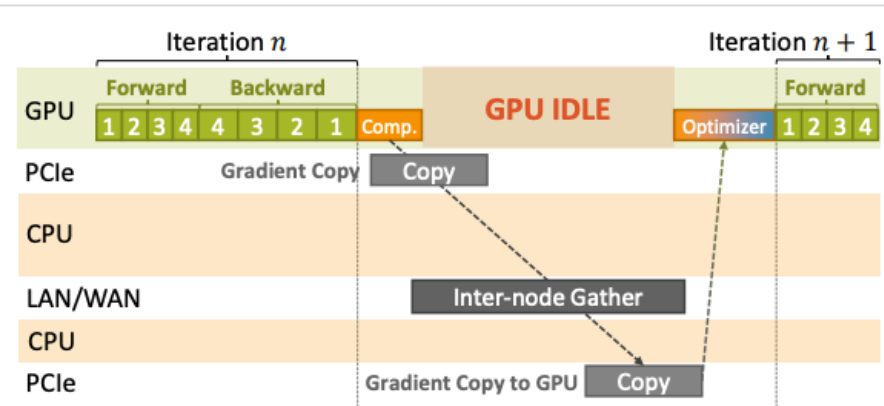
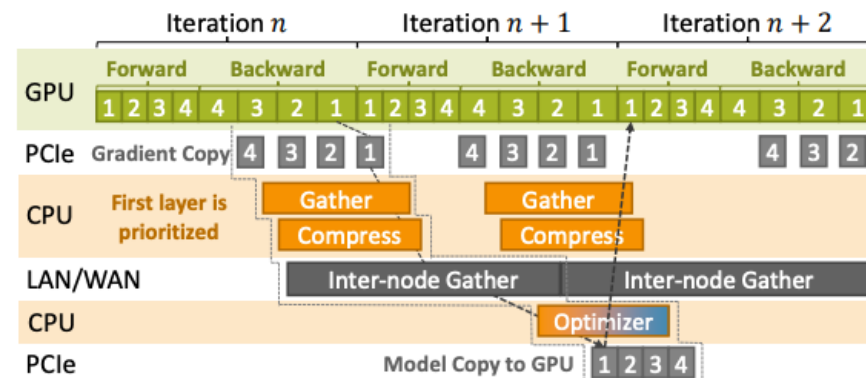


Figure 1: A multi-cluster environment with two on-premises lab clusters and a cloud cluster.



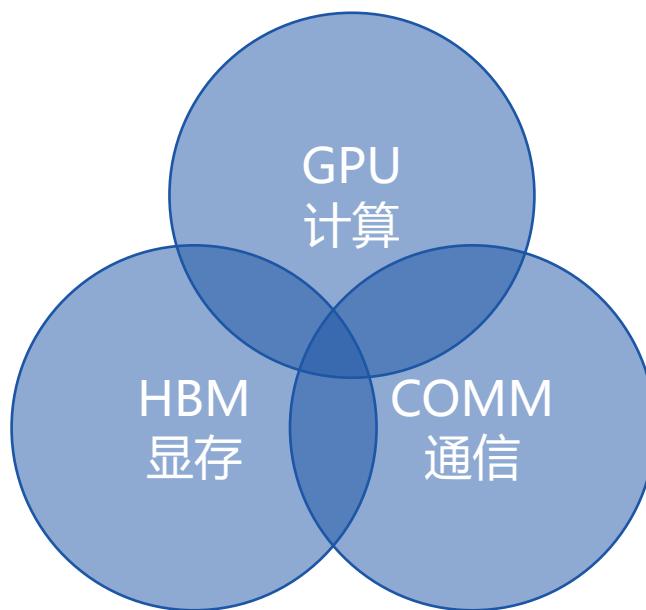
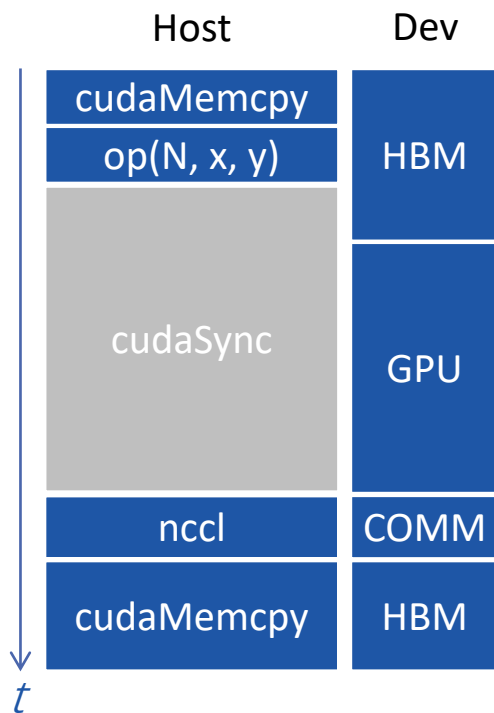
(a) GPU-based Compression (e.g., BytePS-Compress [55])



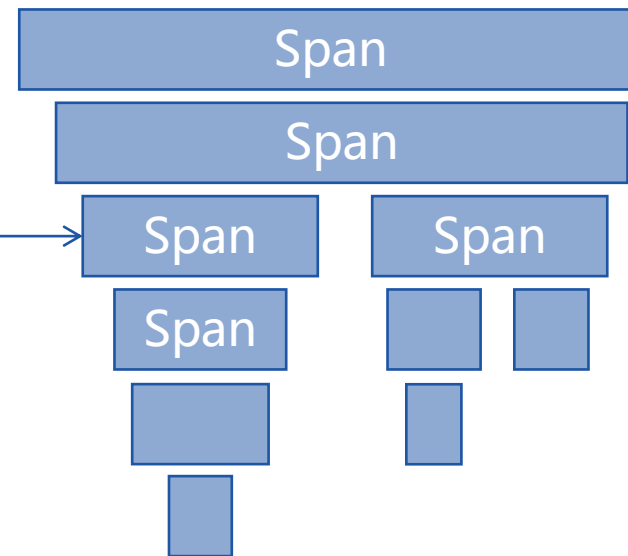
(b) StellaTrain (Ours)

Figure 4: Comparison of training pipelines.

▶ AI 训练和推理的可观测性需求



训练 & 推理



在线、端侧推理服务

- Gateway
- Auth
- 3rd
- ROS2 Nodes
- Gateway
-

PART 02

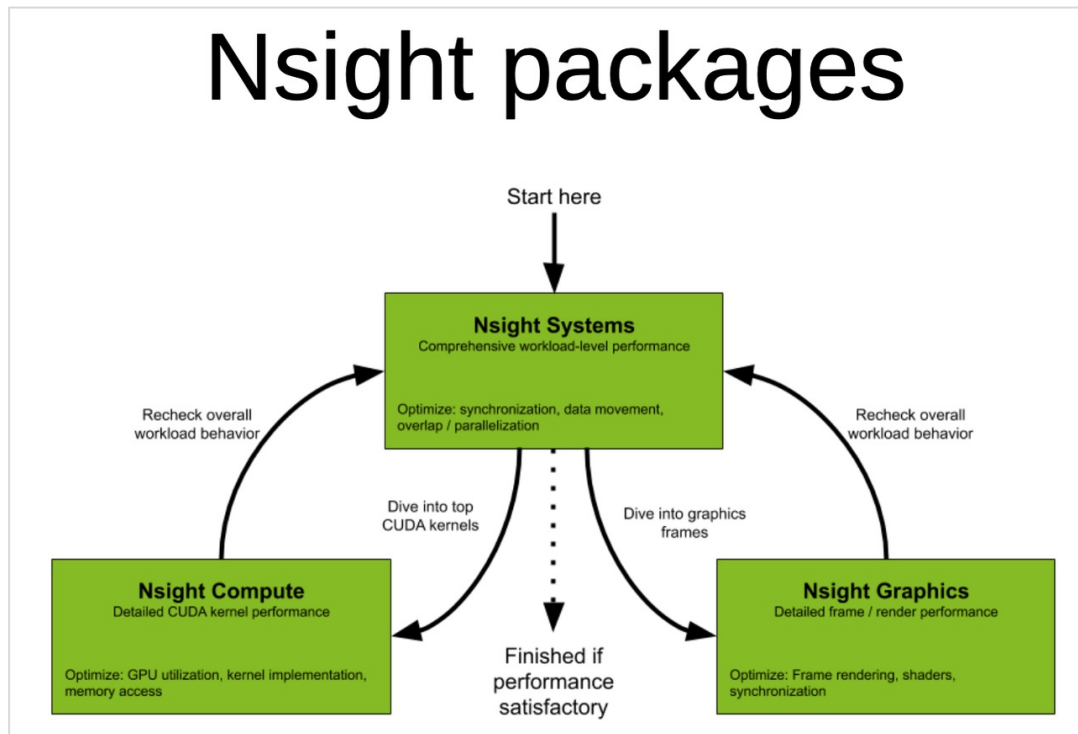
现状：传统解决方案和工具的问题

DCGM Prometheus Exporter



发现故障 ✓
优化性能 ✗

GPU: Nvidia Nsight, PyTorch Profiler



Nsight 的问题:
需要重启进程、缺少 CPU Context。

```
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()
```

需要手工精心打造
插桩、开销

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.

PyTorch Profiler 的问题:
只能用于 PyTorch; 性能影响大; 需要改代码、重启进程。

RDMA 网络：网卡/交换机指标、拨测

RDMA network behind AI Training Cluster

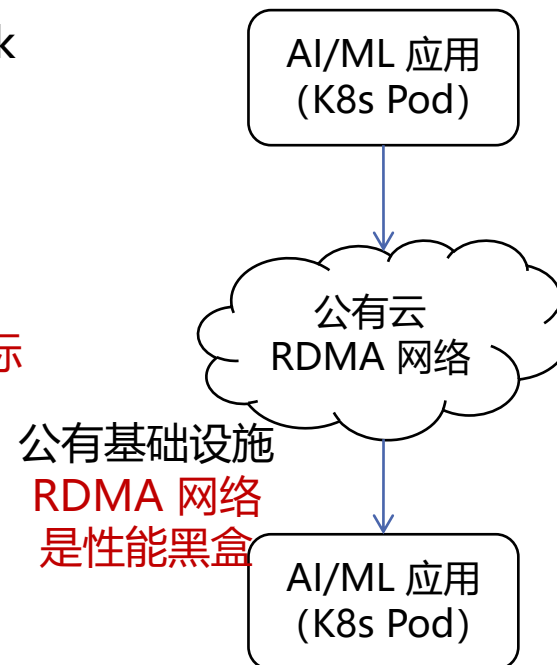
Enables two networked hosts to exchange data in main memory without relying on the processor(CPU)

SCALE AI TRAINING WORKLOADS

THE PICK TO KEEP GPUS BUSY

CAN AFFECT TRAINING EFFICIENCY

Meta: Network Observability for AI/HPC Training Workflows
 私有基础设施
 网卡/交换机指标
 粒度粗



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

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

字节&北邮
 聚焦 RDMA
 网络主动拨测

SIGCOMM

NSDI 2023

▶ 在线推理服务的可观测性：分布式追踪

```
Python TypeScript

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

@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

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?
```

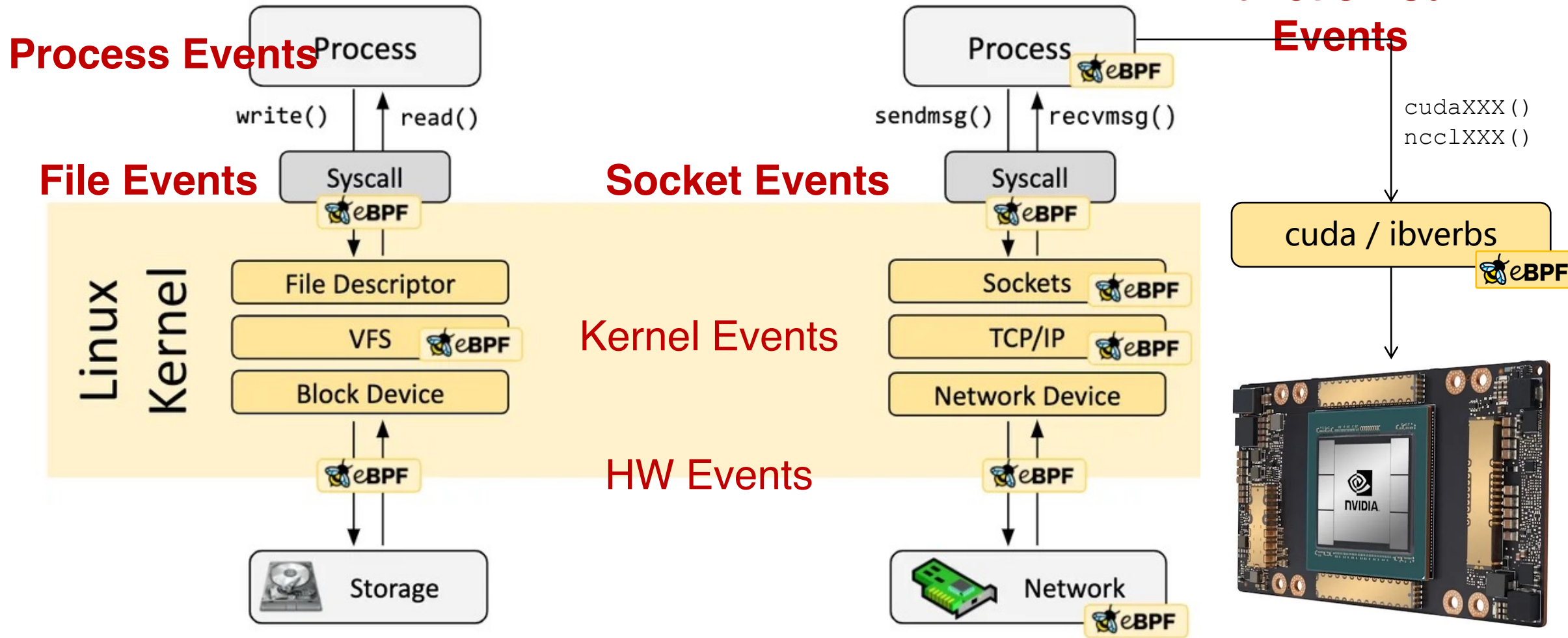
LangSmith

支持的语言有限、需要修改代码

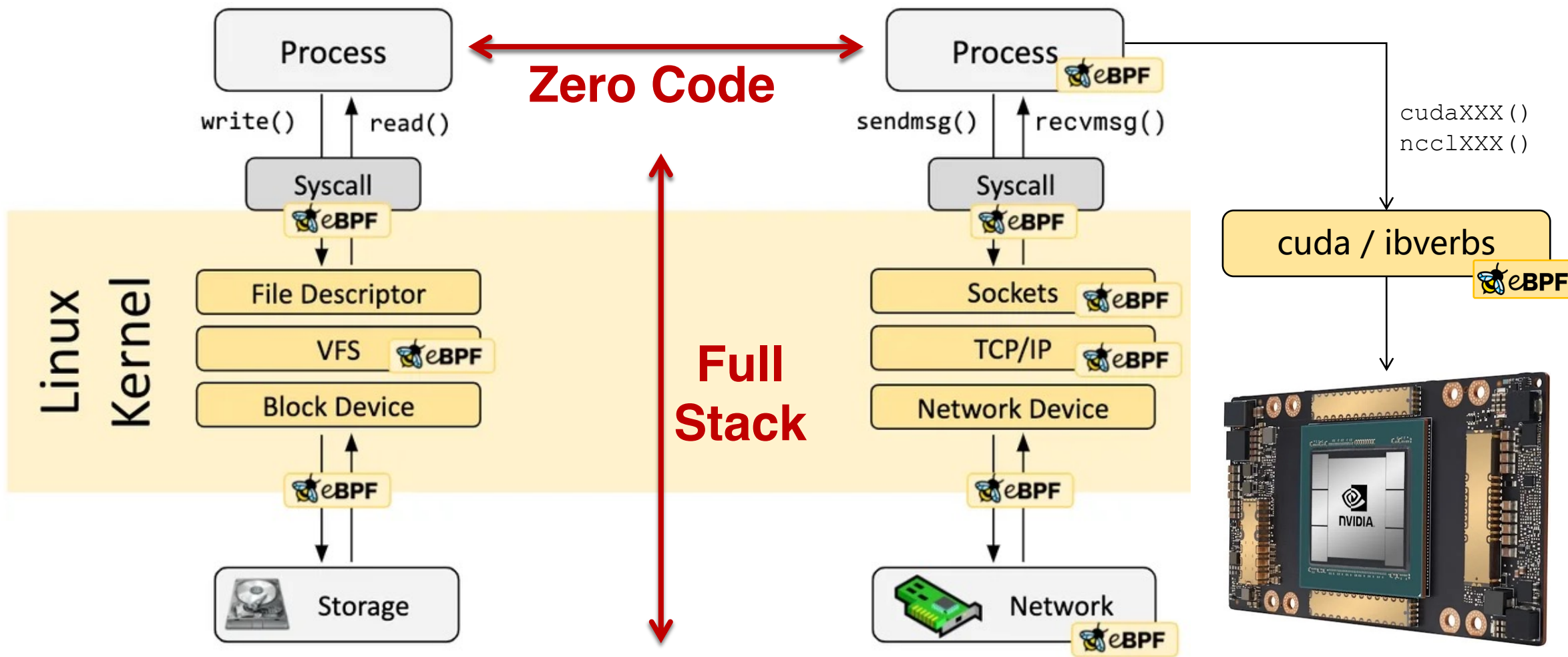
PART 03

方法： eBPF 构建零侵扰可观测性

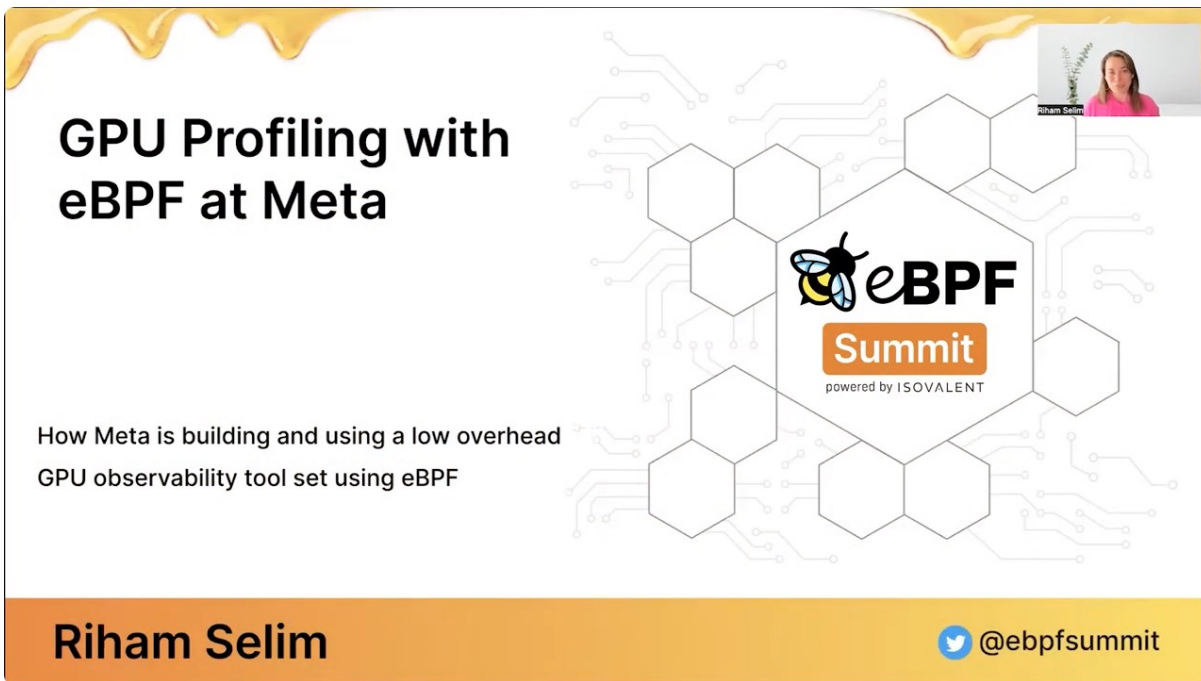
▶ eBPF 的可观测性能力



▶ 使用 eBPF 实现可观测性的优势



▶ 业内探索: eBPF Profiling & Tracing



GPU Profiling with eBPF at Meta

How Meta is building and using a low overhead GPU observability tool set using eBPF

eBPF Summit
powered by ISOVALENT

Riham Selim

@ebpfsummit

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

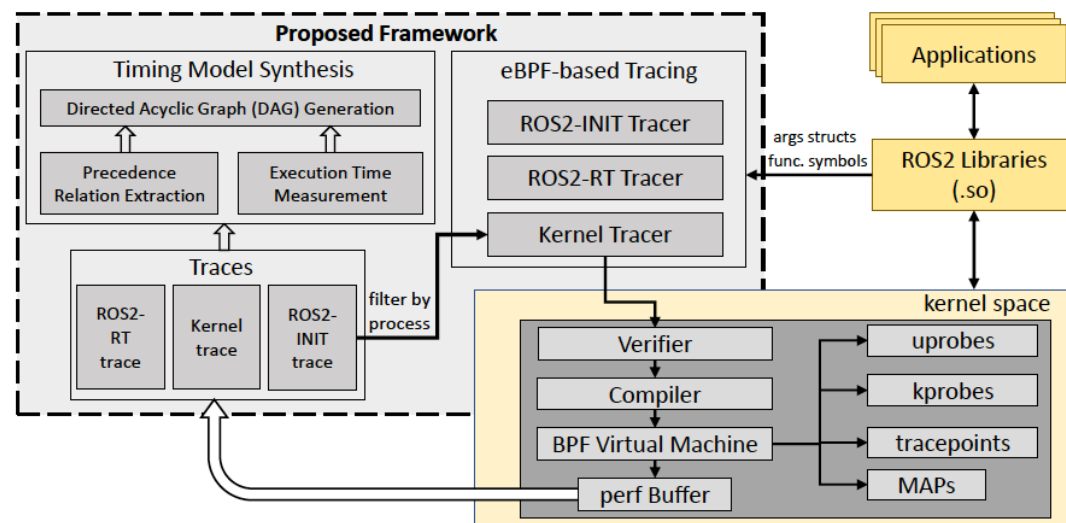


Fig. 1: Proposed trace-enabled timing model synthesis framework.

华为: eBPF Tracing + ROS2

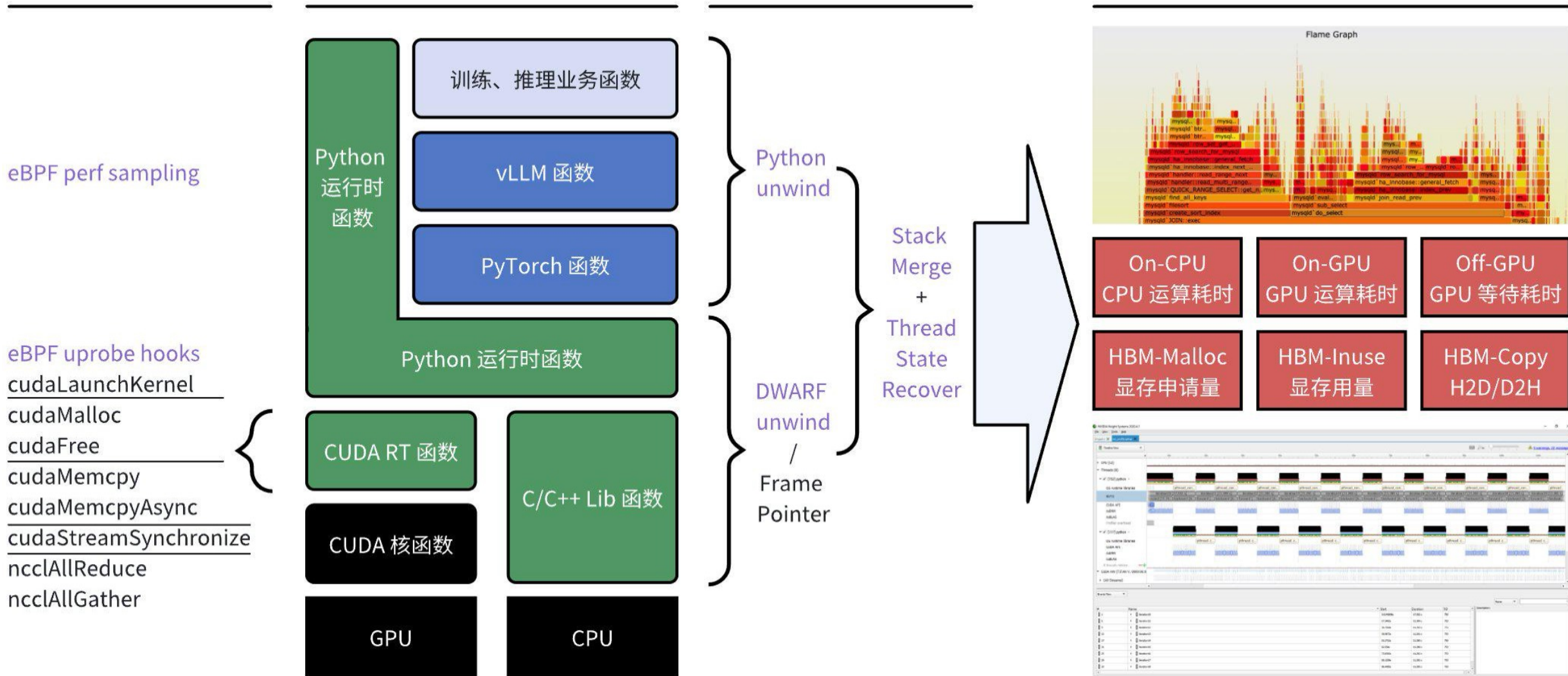
使用 eBPF 实现持续剖析的技术挑战

数据采集

观测对象

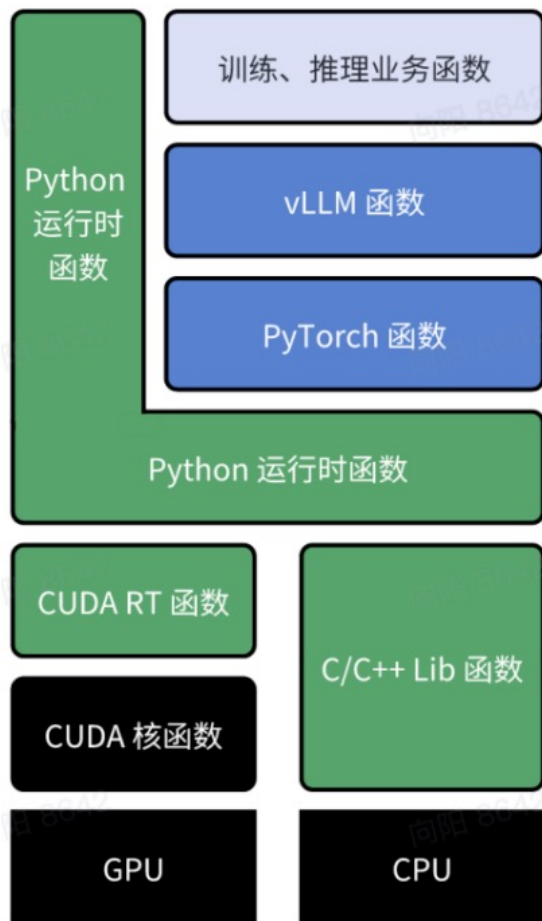
数据关联

数据分析



▶ 如何合并 Python Stack 和 C/C++ Stack

栈合并目的：全栈剖析



```
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/deepflo>

▶ 例：剖析显存申请和使用量

① eBPF uprobe Hook cuda_malloc 获取显存申请调用栈

```
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;
}
```

③ eBPF uprobe Hook cuda_free 获取释放的显存地址

```
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;
}
```

④ 计算当前显存消耗

② eBPF uretprobe Hook cuda_malloc 获取申请的显存地址

```
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();

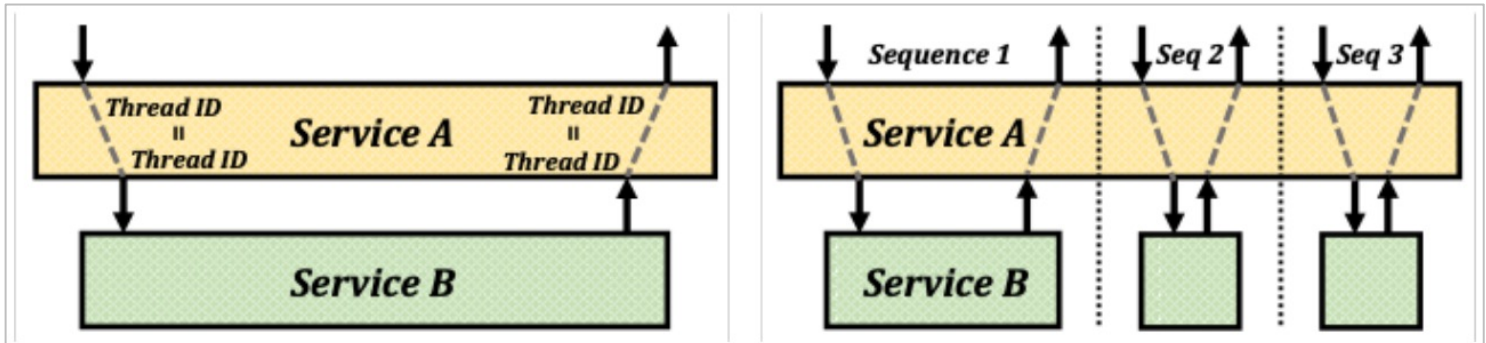
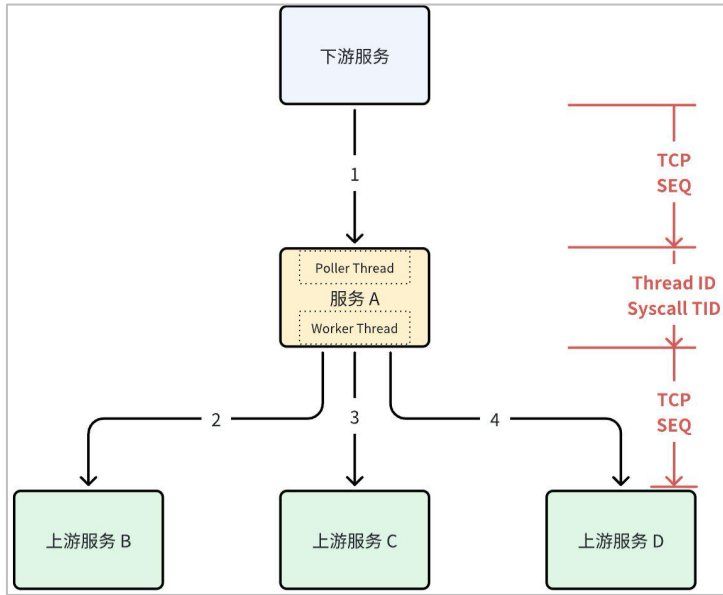
    bpf_probe_read_user(&key->mem_addr, sizeof(__u64), (void *)data->address);
    key->mem_size = (__u64) data->size;

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

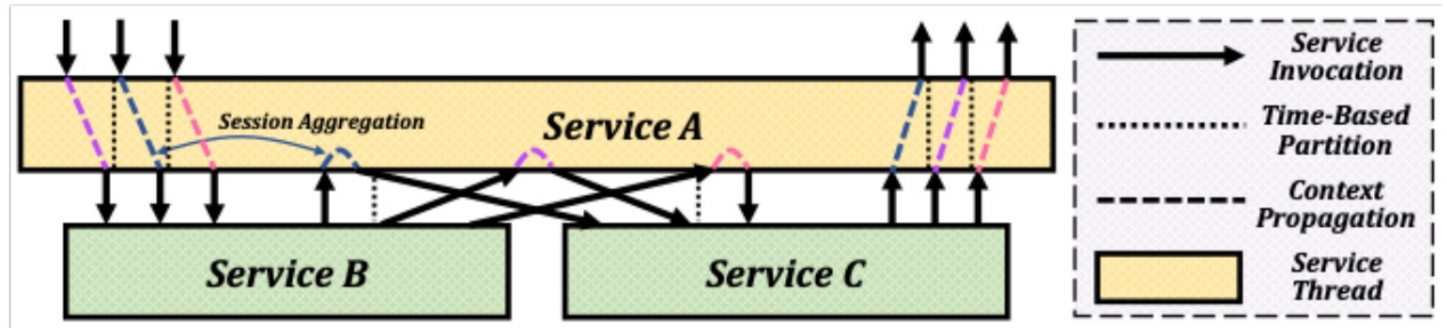
    state->regs.ip = PT_REGS_IP(ctx);
    state->regs.sp = PT_REGS_SP(ctx);
    state->regs.bp = PT_REGS_FP(ctx);
    add_frame(&state->stack, state->regs.ip);
    bpf_tail_call(ctx, &NAME(progs_jmp_uprobe_map), PROG_DWARF_UNWIND_IDX);

    return 0;
}
```

使用 eBPF 实现分布式追踪的技术挑战



(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

 Junxian Shen^{1,2,*}, Han Zhang^{1,4,*}, Yang Xiang^{2,*}, Xingang Shi^{1,4}, Xinrui Li³, Yunxi Shen³, Zijian Zhang^{1,2}, Yongxiang Wu³, Xia Yin^{3,4}, Jilong Wang^{1,4}, Mingwei Xu^{1,4}, Yahui Li¹, Jiping Yin², Jianchang Song², Zhuofeng Li², Runjie Nie²

¹Institute for Network Sciences and Cyberspace, Tsinghua University

²Yunshan Networks

³Department of Computer Science and Technology, Tsinghua University

⁴Zhongguancun Laboratory

 *shenxj22@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 Li¹, Jiping Yin², Jianchang Song², Zhuofeng Li², Runjie Nie². 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>

[Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code](#)

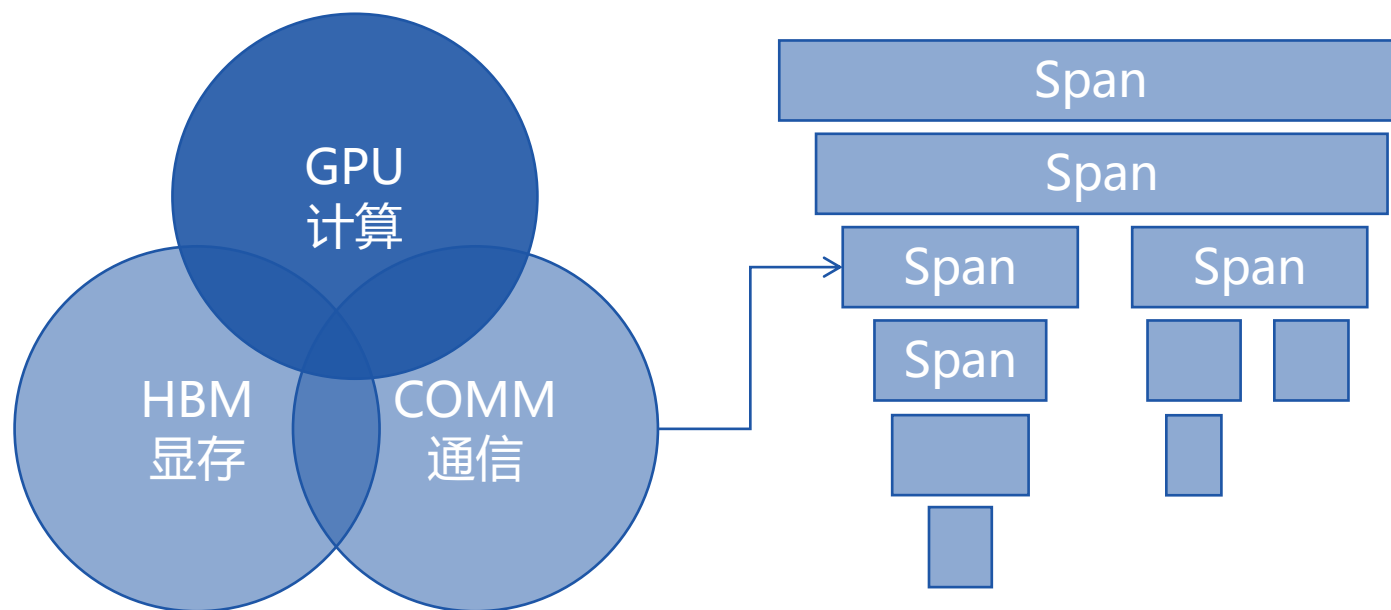
PART 04

实践：PyTorch 全栈剖析和追踪

▶ DeepFlow 中的 eBPF AutoProfiling



▶ 1. Compute Profiling



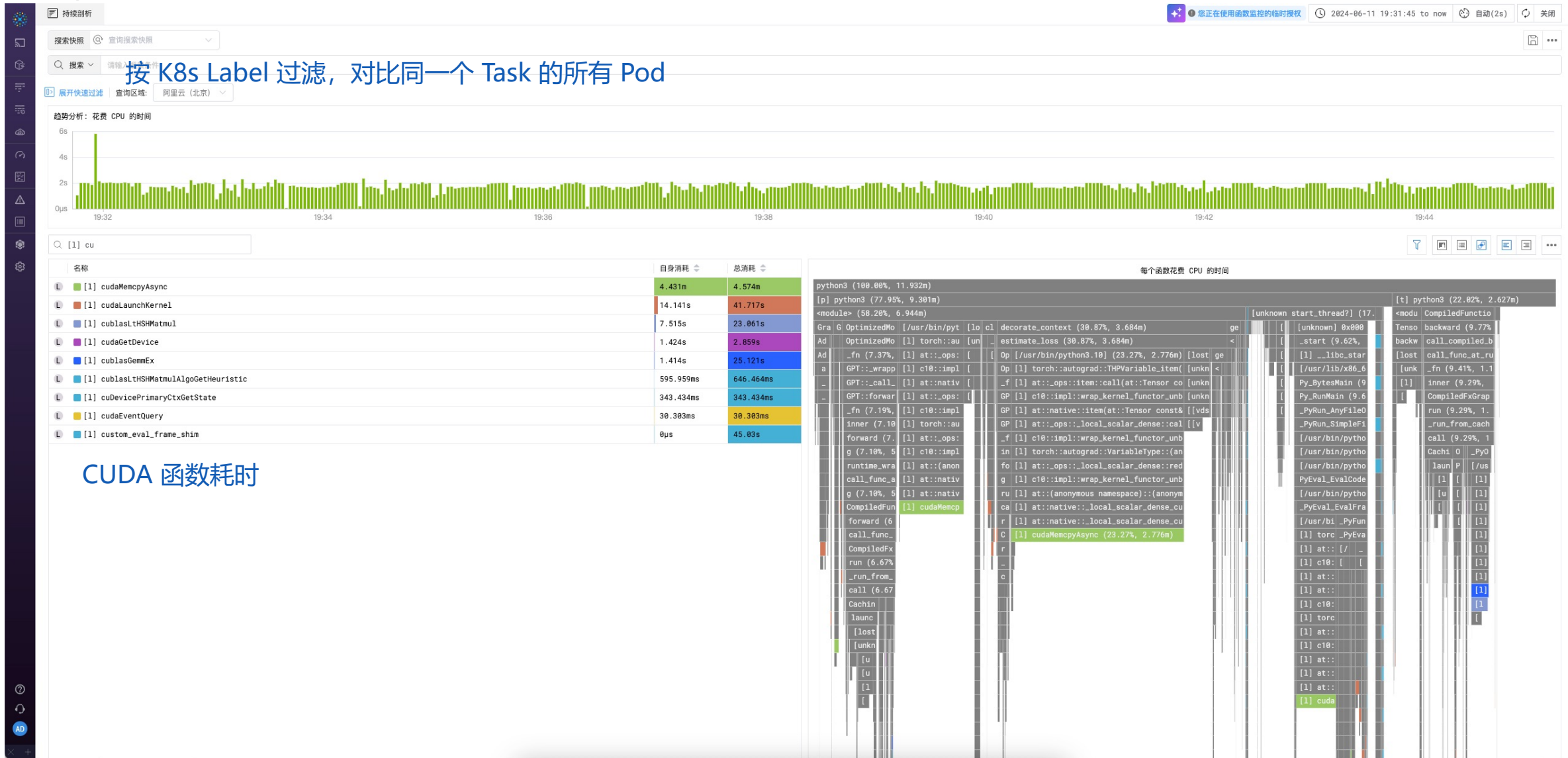
PyTorch + nanoGPT CPU & GPU 火焰图

到底哪个函数调用栈在 CPU、GPU 上花费的时间最长？
到底一个 Task 的哪个 Worker 拖累了整体的速度？
零侵扰、全栈回答上述问题

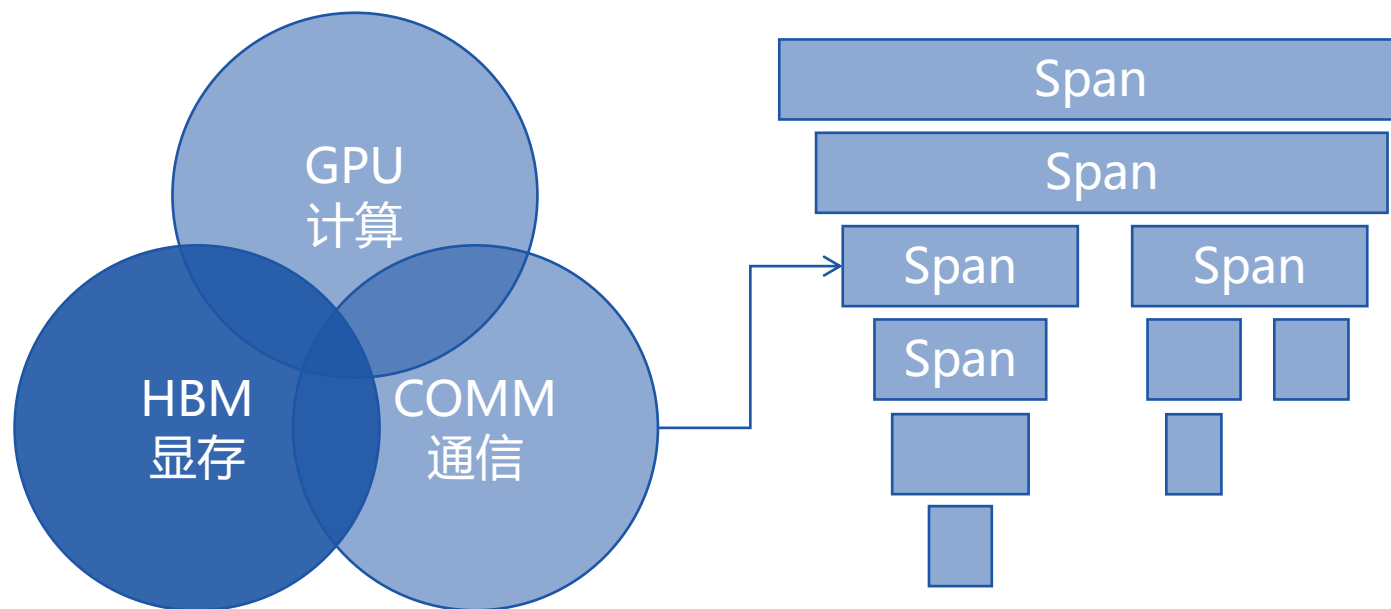


CPU & GPU 全栈剖析:
✓ Python 业务函数
✓ Python vLLM 函数
✓ Python PyTorch 函数
✓ C/C++ Lib 函数
✓ CUDA 入口函数

PyTorch + nanoGPT CPU & GPU 火焰图

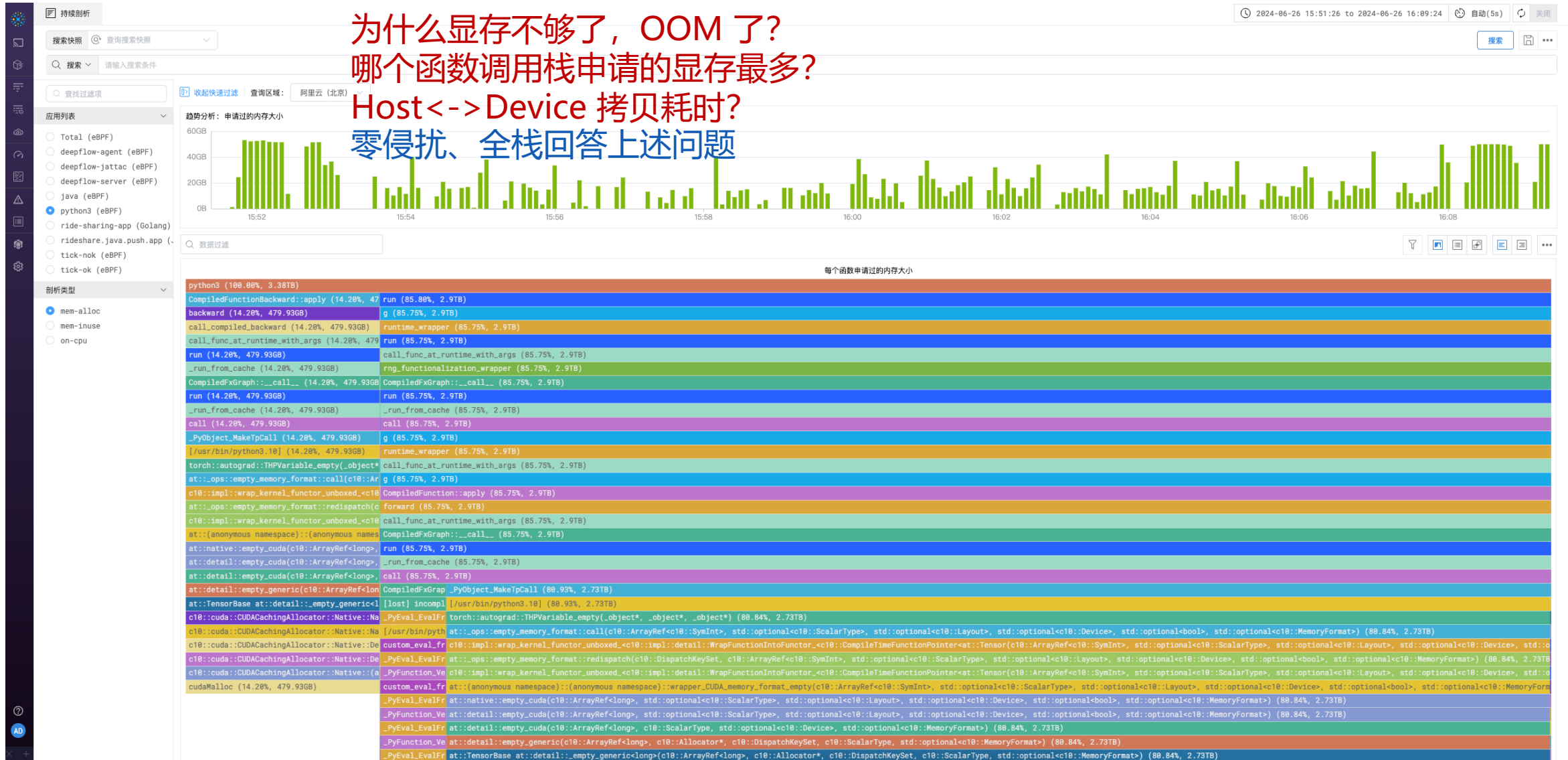


▶ 2. HBM Profiling

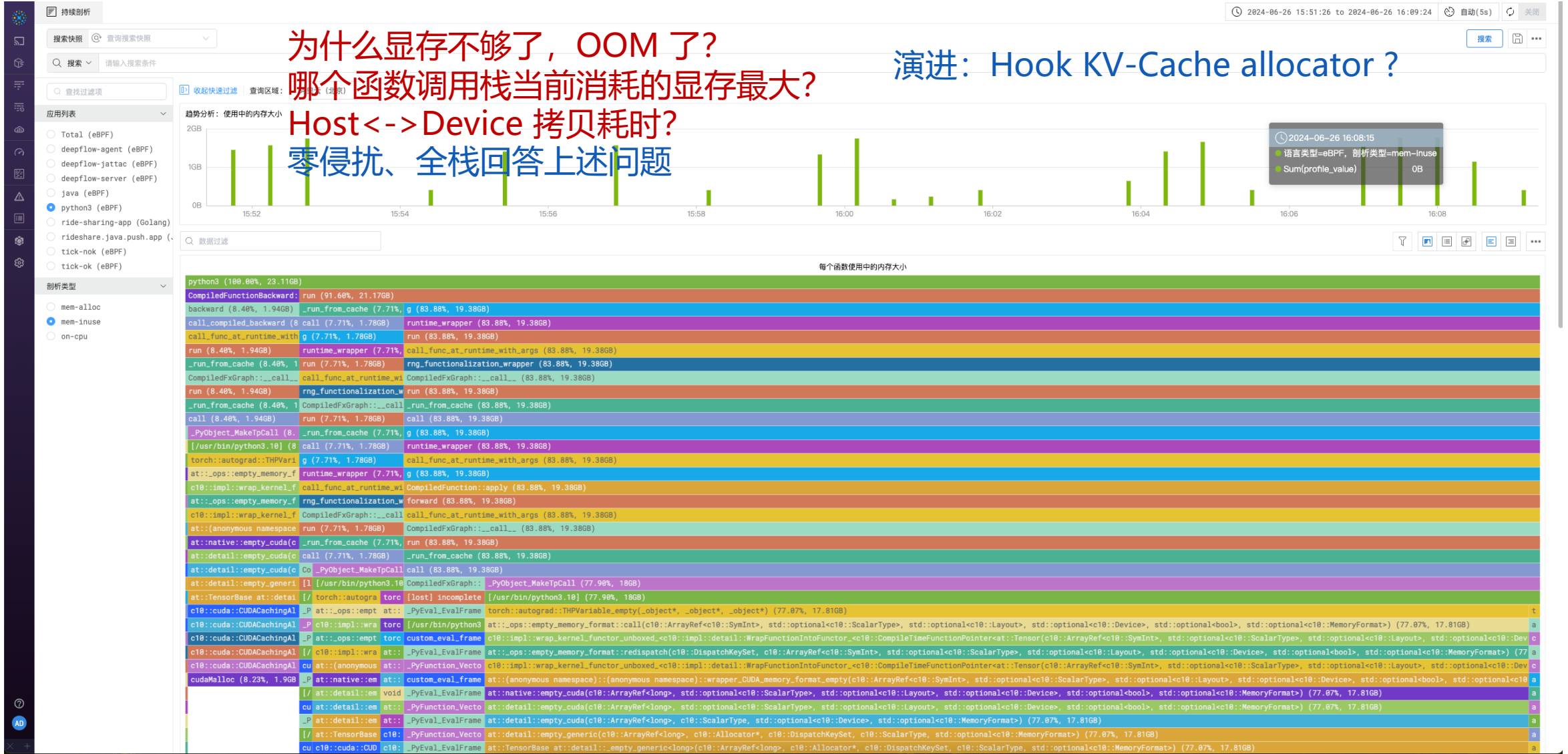


▶ CUDA mem-alloc 显存申请火焰图

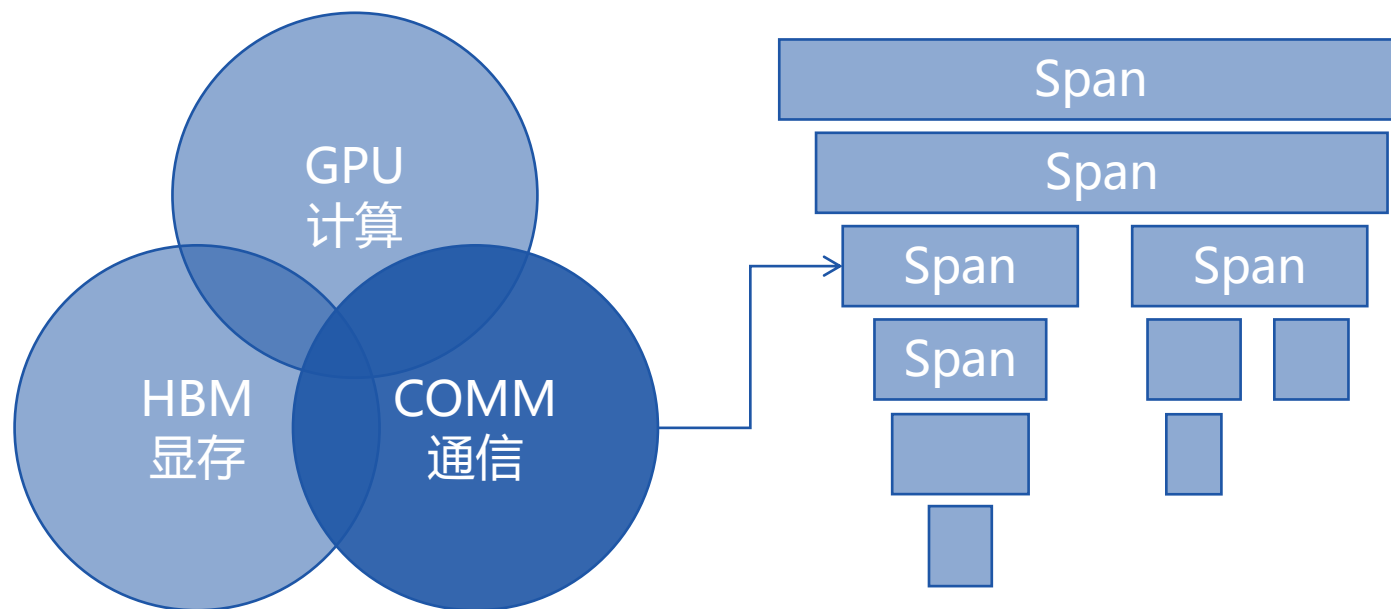
为什么显存不够了, OOM 了?
哪个函数调用栈申请的显存最多?
Host <-> Device 拷贝耗时?
零侵扰、全栈回答上述问题



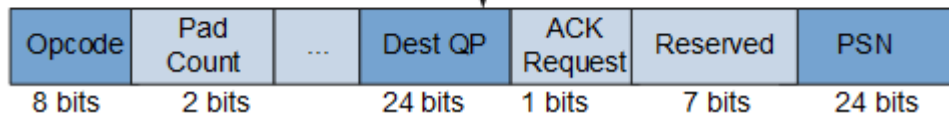
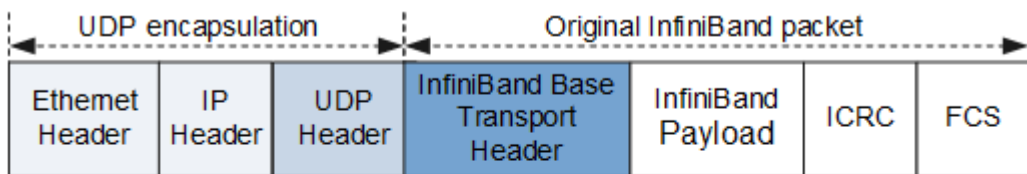
▶ CUDA mem-inuse 显存实时用量火焰图



▶ 3. COMM. Profiling



RDMA 网络性能剖析



RoCEv2 Packet

eBPF Hooks

Function	作用
ibv_modify_qp	Queue Pair 状态修改
ibv_reg_mr	Memory Region 注册
ibv_dereg_mr	Memory Region 注销
ibv_post_send	buffer 发送或 RDMA 读写
ibv_poll_cq	等待 wr 完成

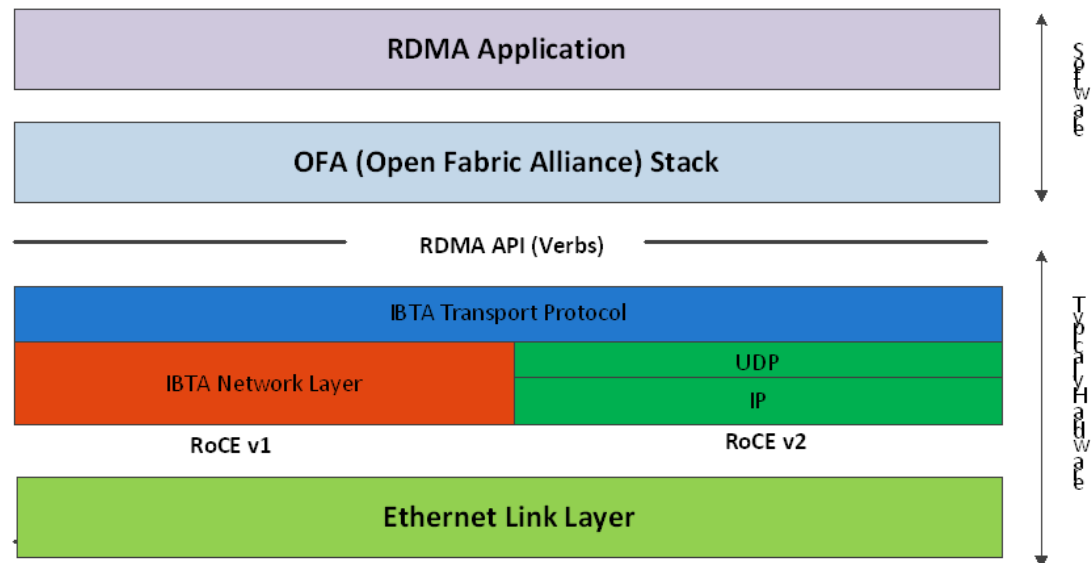
libibverbs Hooks, NCCL Hooks, PyTorch Hooks, ...

关键标签:

- Client 及其关联的 K8s Pod、标签
- Server 及其关联的 K8s Pod、标签
- Client Queue Pair
- Server Queue Pair

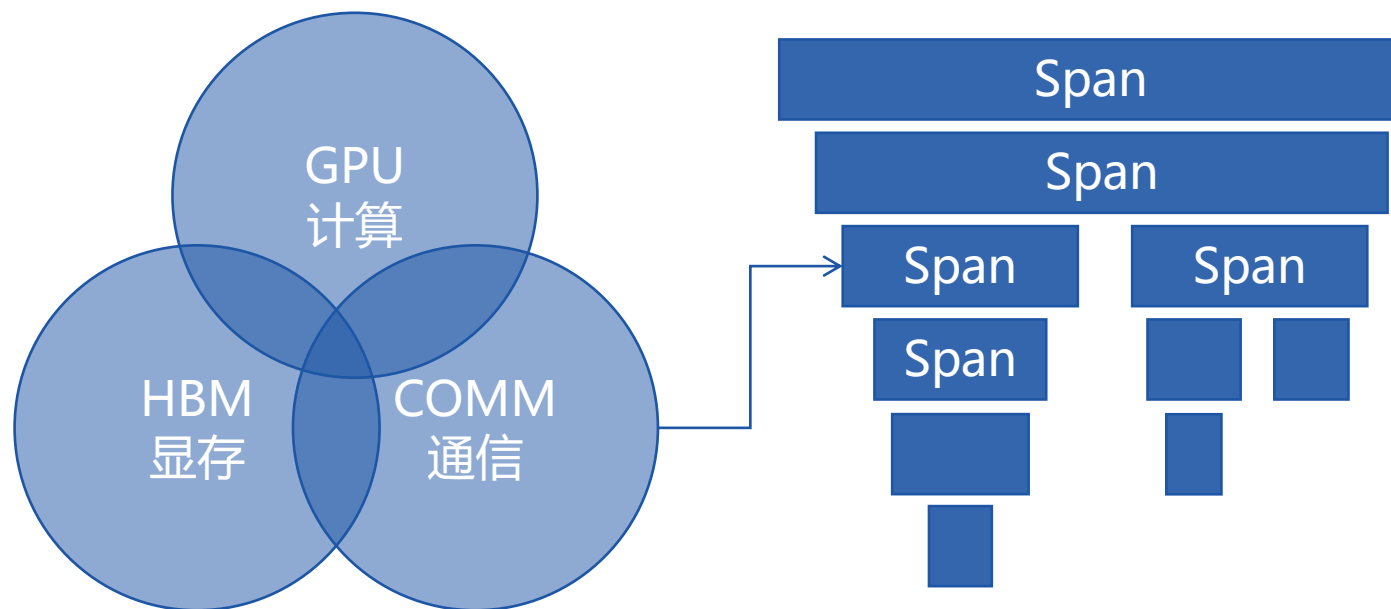
关键指标:

- 丢包率: 即 NACK 的比例
- 时延: ACK 的时延
- 吞吐: 通信对的 bps、pps 等

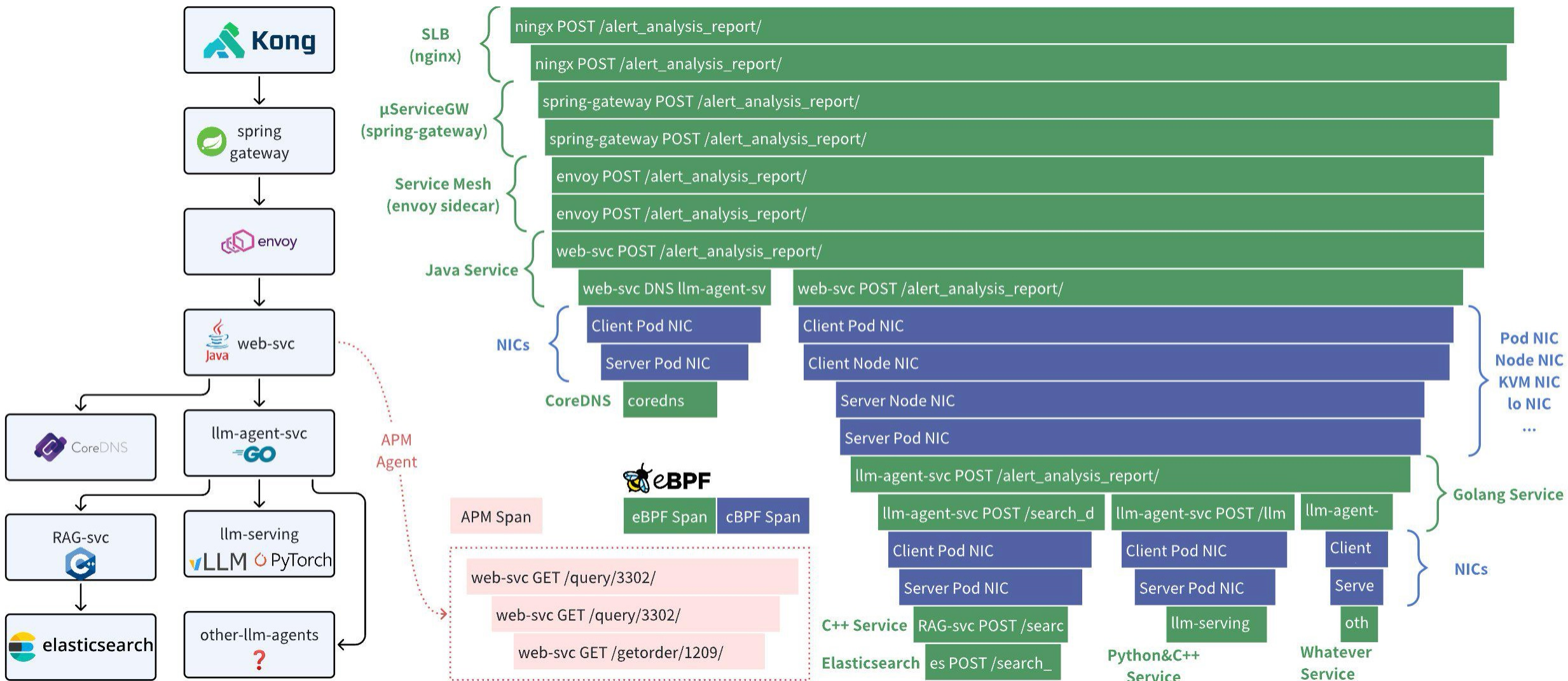


使用 eBPF Hook RDMA API

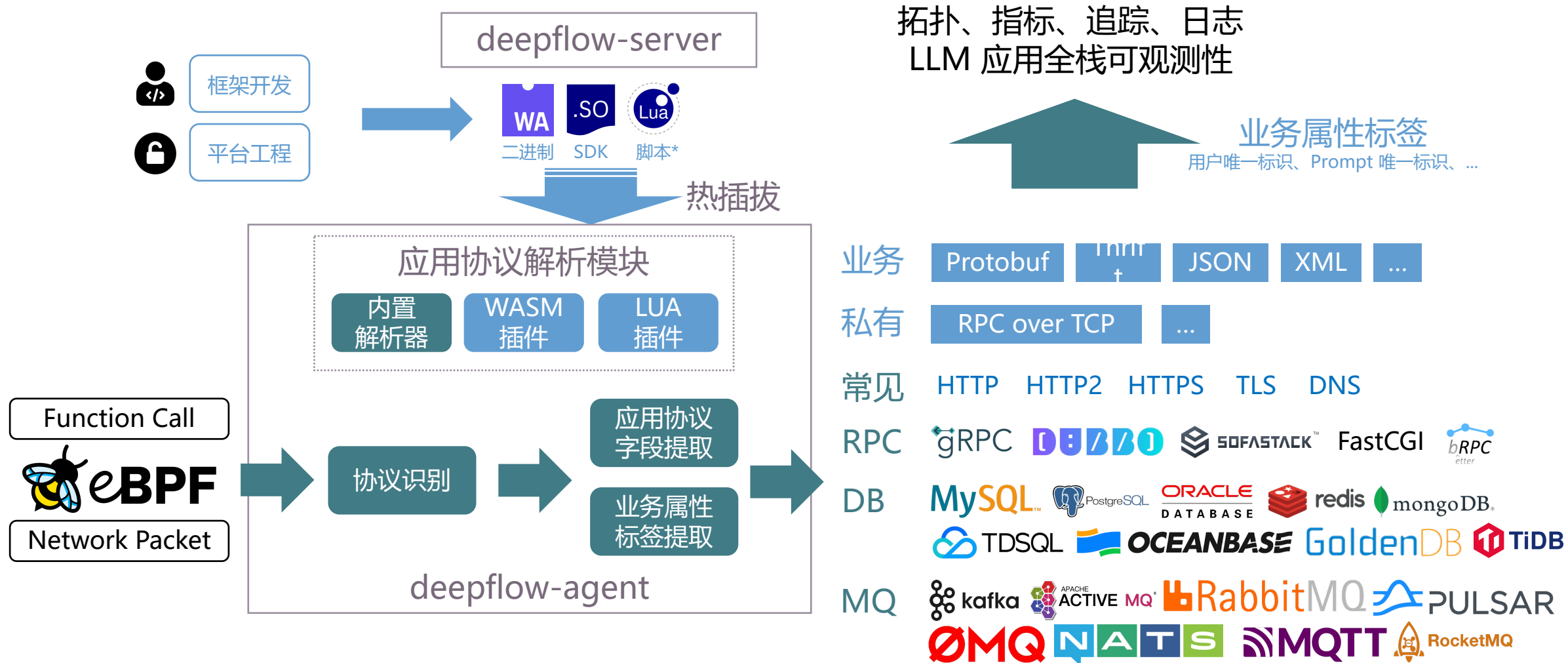
▶ 4. Distributed Tracing



▶ 在线推理服务、端侧 ROS2 推理服务



▶ 内置协议识别能力 + 可编程协议识别能力



PART 05

探索：Agent 自动优化 ML 代码

▶ 如何快速高效的理理解全栈函数



Business Code

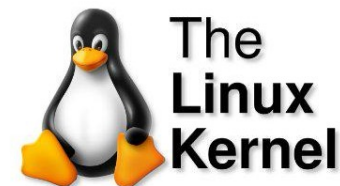
Prompt Engineering
tag.commit_id



RAG
Docs / Comments

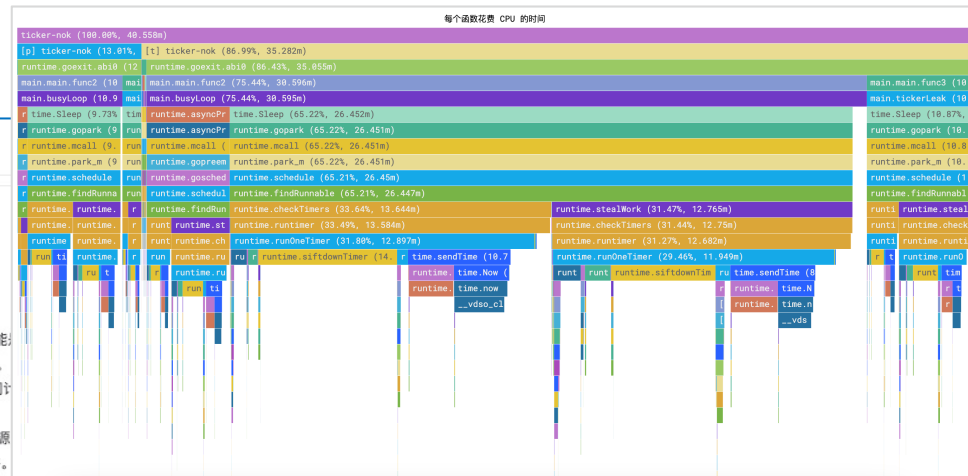


LLM / 微调



LLM / 微调

▶ 利用 LLM Agent 自动优化 ML 代码



持续剖析智能分析

性能分析总的情况

- 总的调用路径: 1312条

重要节点概述

- runtime.siftdownTimer: 自身耗时高达8803秒, 总耗时8986秒, 表明定时器操作占用了大量的CPU资源, 可能是并发同步上存在问题。
- runtime.chansend: 自身耗时4063秒, 总耗时4136秒, 频繁的channel发送操作, 可能是并发同步上存在问题。
- __vdso_clock_gettime: 自身耗时2880秒, 总耗时2927秒, 表示获取时间操作较为频繁, 可能与定时器和时间片有关。
- runtime.runOneTimer: 自身耗时1595秒, 总耗时22434秒, 与定时器执行相关的函数, 是CPU消耗较大的点。
- runtime.runtimer: 自身耗时1106秒, 总耗时23719秒, 另一个与定时器相关的函数, 同样耗费了大量的CPU资源。
- main.busyLoop: 自身耗时416秒, 总耗时21161秒, 代表业务层面的一个忙循环, 可能在等待某些资源或者条件。

最热调用路径概述

- 调用链1: ticker-nok -> [t] ticker-nok -> runtime.goexit.abi0 -> main.main.func2 -> main.busyLoop
 - 出现次数588次, 结合 main.busyLoop 节点的高耗时, 重点关注业务层函数的循环逻辑及其优化。
- 调用链2: ticker-nok -> [t] ticker-nok -> runtime.goexit.abi0 -> main.main.func2 -> main.busyLoop -> time.Sleep -> runtime.gopark -> runtime.mcall
 - 出现次数426次, 含有 time.Sleep, 表明程序有意识的休眠, 可能是为了控制循环速度或等待条件。
- 调用链3: ticker-nok -> [t] ticker-nok -> runtime.goexit.abi0 -> main.main.func2 -> main.busyLoop -> time.Sleep -> runtime.gopark -> runtime.mcall -> runtime.park_m -> runtime.schedule -> runtime.findRunnable -> runtime.checkTimers
 - 出现次数209次, 链条较长且涉及多个调度和定时器相关函数, 说明在等待和定时器检查上消耗了大量的CPU。

总体概述

整个性能分析显示, 程序存在大量与定时器相关的操作, 这些操作占据了很大一部分的CPU资源, 特别是 runtime.siftdownTimer 和 runtime.runOneTimer。同时, main.busyLoop 表明业务逻辑中存在大量的忙等, 可能需要优化逻辑以减少CPU消耗。调用链分析显示, time.Sleep 和 runtime.gopark 的使用表明在循环中有意识的休眠或等待, 可能需要重新评估这部分逻辑的必要性和效率。需要特别注意的是, 频繁的定时器操作和channel通信可能是导致性能问题的关键所在, 应当作为优化的重点。

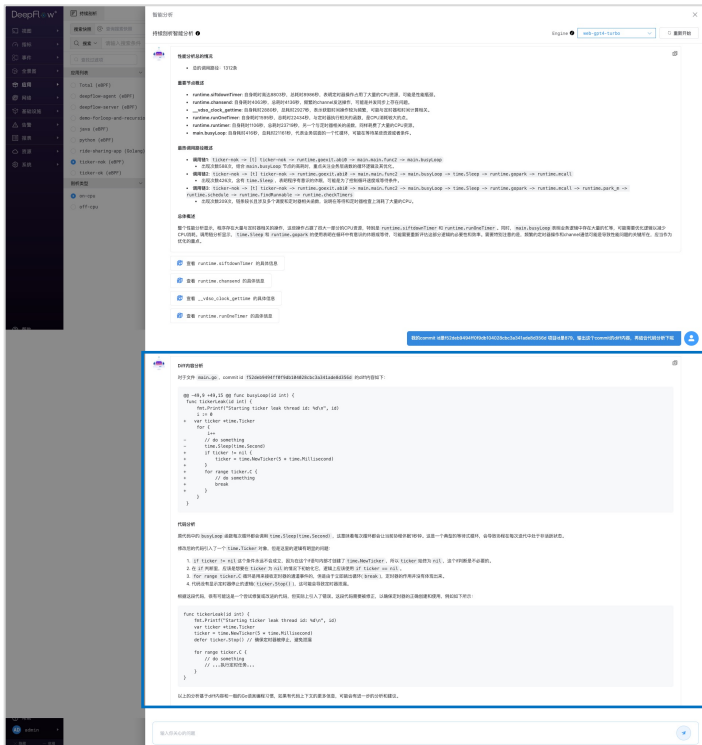
查看 runtime.siftdownTimer 的具体信息

查看 runtime.chansend 的具体信息

查看 __vdso_clock_gettime 的具体信息

查看 runtime.runOneTimer 的具体信息

利用 LLM Agent 自动优化 ML 代码 / 2



Diff内容分析

对于文件 main.go, commit id f52deb9494ff0f9db104028cbc3a341ade8d356d 的 diff内容如下:

```
@@ -49,9 +49,15 @@ func busyLoop(id int) {
func tickerLeak(id int) {
    fmt.Printf("Starting ticker leak thread id: %d\n", id)
    i := 0
+   var ticker *time.Ticker
    for {
        i++
        // do something
        time.Sleep(time.Second)
+       if ticker != nil {
+           ticker = time.NewTicker(5 * time.Millisecond)
+       }
+       for range ticker.C {
+           // do something
+           break
+       }
    }
}
```

代码分析

原代码中的 busyLoop 函数每次循环都会调用 time.Sleep(time.Second), 这意味着每次循环都会让当前协程休眠1秒钟。这是一个典型的等待式循环, 会导致协程在每次迭代中处于非活跃状态。

修改后的代码引入了一个 time.Ticker 对象, 但是这里的逻辑有明显的问题:

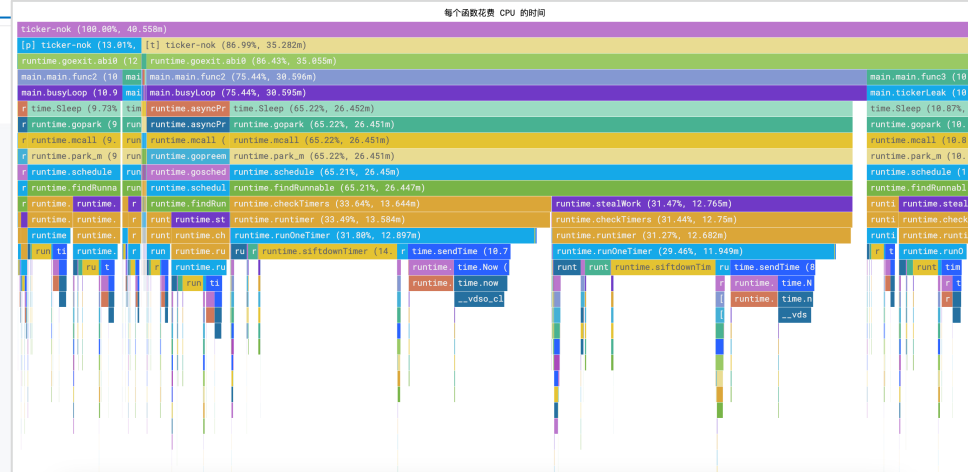
1. if ticker != nil 这个条件永远不会成立, 因为在这个if语句内部才创建了 time.NewTicker, 所以 ticker 始终为 nil, 这个if判断是不必要的。
2. 在 if 判断里, 应该是想要在 ticker 为 nil 的情况下初始化它, 逻辑上应该使用 if ticker == nil。
3. for range ticker.C 循环是用来接收定时器的通道事件的, 但是由于立即跳出循环(break), 定时器的作用并没有体现出来。
4. 代码没有显示定时器停止的逻辑(ticker.Stop()), 这可能会导致定时器泄漏。

根据这段代码, 很有可能这是一个尝试修复或改进的代码, 但实际上引入了错误。这段代码需要被修正, 以确保定时器的正确创建和使用, 例如如下所示:

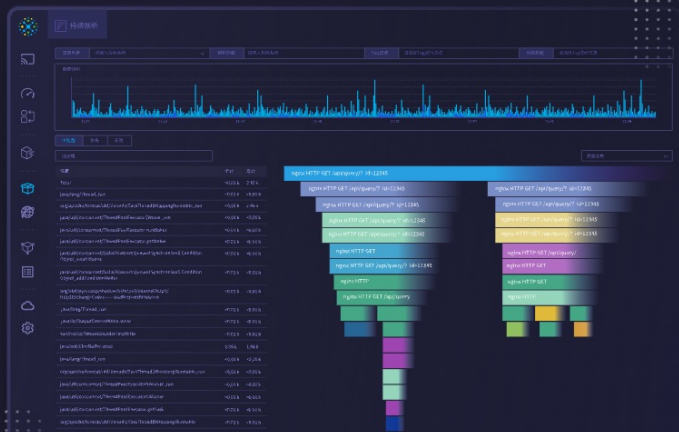
```
func tickerLeak(id int) {
    fmt.Printf("Starting ticker leak thread id: %d\n", id)
    var ticker *time.Ticker
    ticker = time.NewTicker(5 * time.Millisecond)
    defer ticker.Stop() // 确保定时器被停止, 避免泄漏

    for range ticker.C {
        // do something
        // ...执行定时任务...
    }
}
```

以上的分析基于diff内容和一般的Go语言编程习惯, 如果有代码上下文的更多信息, 可能会有进一步的分析和建议。



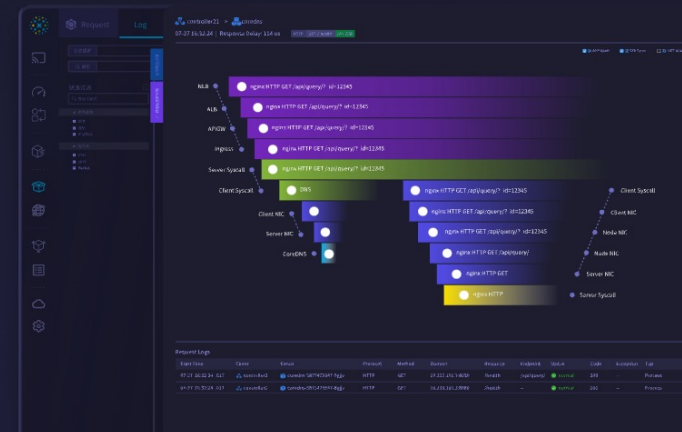
DeepFlow: 零侵扰实现 AI 应用的全栈可观测性



CPU+GPU 全栈持续剖析



集合通信网络性能监控



在线推理服务分布式追踪



无需修改代码, 无需重启进程

科技生态圈峰会 + 深度研习



—1000+ 技术团队的选择



上海站

K+全球软件研发行业创新峰会

时间: 2024.06.21-22



敦煌站

K+思考周®研习社

时间: 2024.10.17-19



香港站

K+思考周®研习社

时间: 2024.11.10-12



K+峰会详情



上海站

Ai+研发数字峰会

时间: 2024.05.17-18



北京站

Ai+研发数字峰会

时间: 2024.08.16-17



深圳站

Ai+研发数字峰会

时间: 2024.11.08-09



AiDD峰会详情



2024 AI+研发数字峰会

AI+ Development Digital summit

深圳站 11/08-09

AI 驱动研发变革 促进企业降本增效

2024深圳站-议题设置

AI+产品线	LLM驱动产品创新	LLM驱动需求与业务分析	AI驱动设计与用户体验
AI+开发线	AI 原生应用开发框架与技术	AI Agents在研发落地实践	LLM驱动编程与单测
AI+测试线	LLM驱动测试分析与设计	基于LLM生成测试脚本与数据	LLM和AI应用的评测
AI+工程线	AI+DevOps 与工具 (LLM 时代的平台工程)	大模型对齐与安全	端侧大模型与云端协同
AI+领域线	领域大模型 SFT 与优化	知识增强与数据智能	大厂专场

扫描右侧二维码
查看更多会议详情



早鸟票限时抢购中 (截止到9月30日)

¥ 3680

早鸟票

¥ 2800

学生票

THANKS

