Grafana Labs

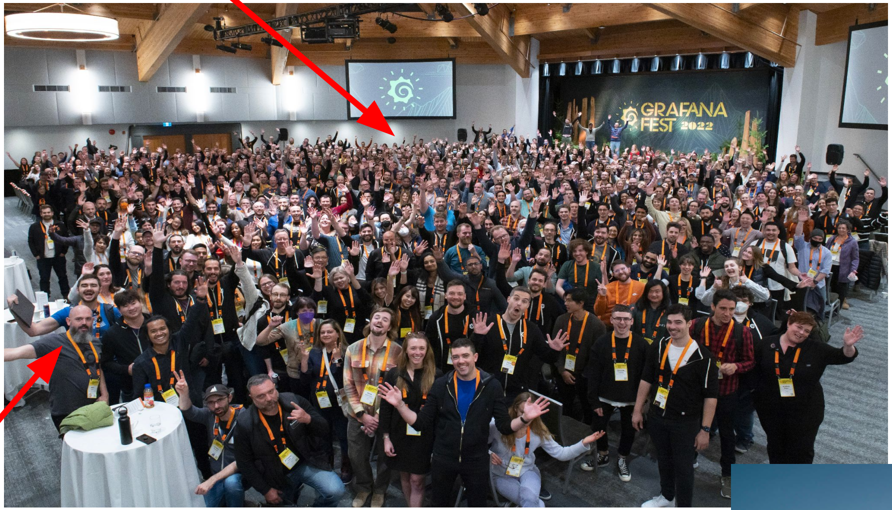
Auto-instrumentation for GPU performance using eBPF

FOSDEM ‘25

Annanay Agarwal
Senior Software Engr.
AI/ML

Marc Tuduri
Staff Software Engr.
Beyla

Nikola Grcevski
Principal Software Engr.
Beyla


GRAFANA
FEST 2022

AGENDA

1. Understanding the problem
    a. Current GPU monitoring solutions
    b. GPU programming model
2. Proposed solution using eBPF and Grafana Beyla
    a. Calls instrumented so far
    b. Future plans
3. Q&A

Hello Randy!

* this is not an endorsement

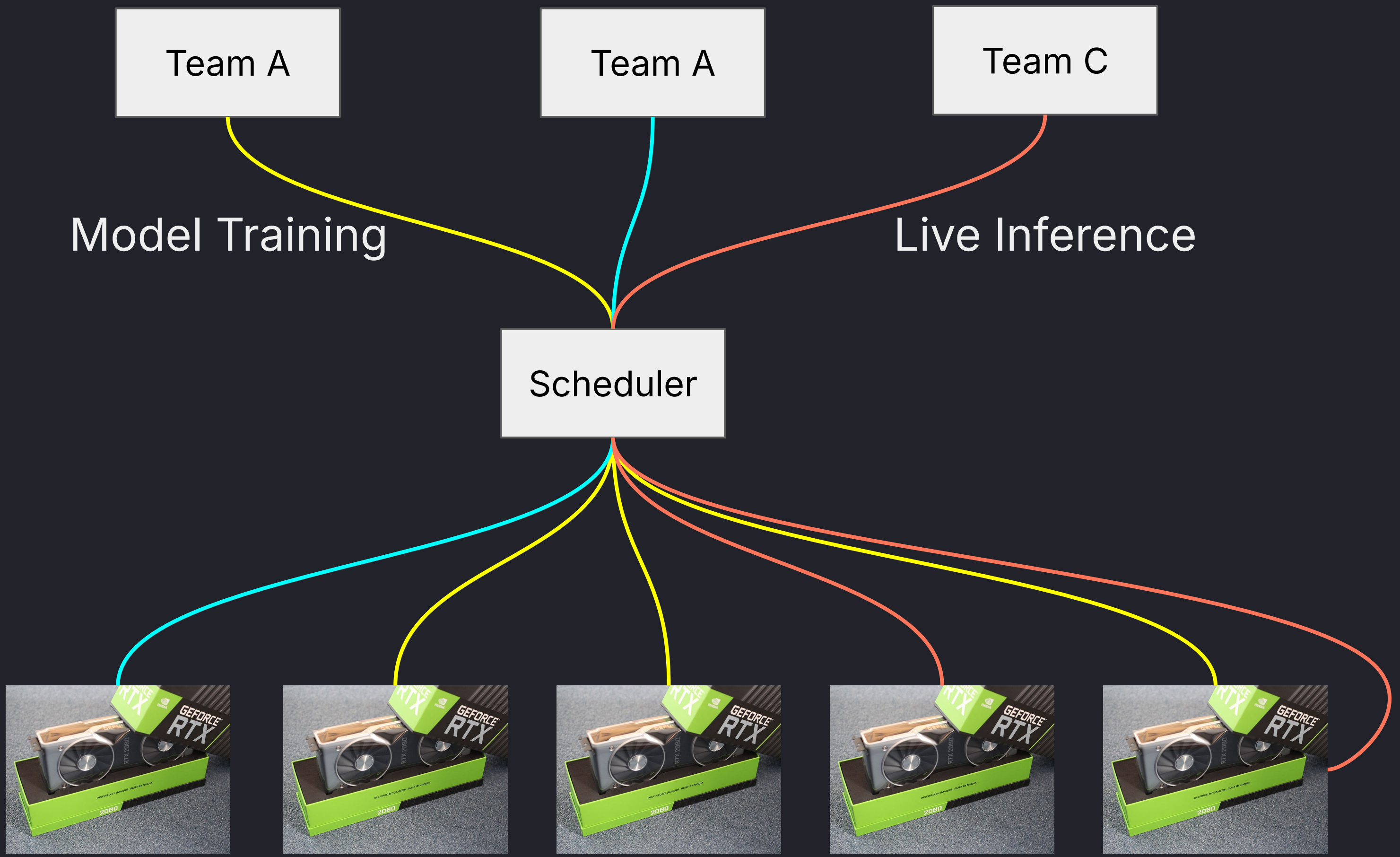

GEFORCE RTX
2080

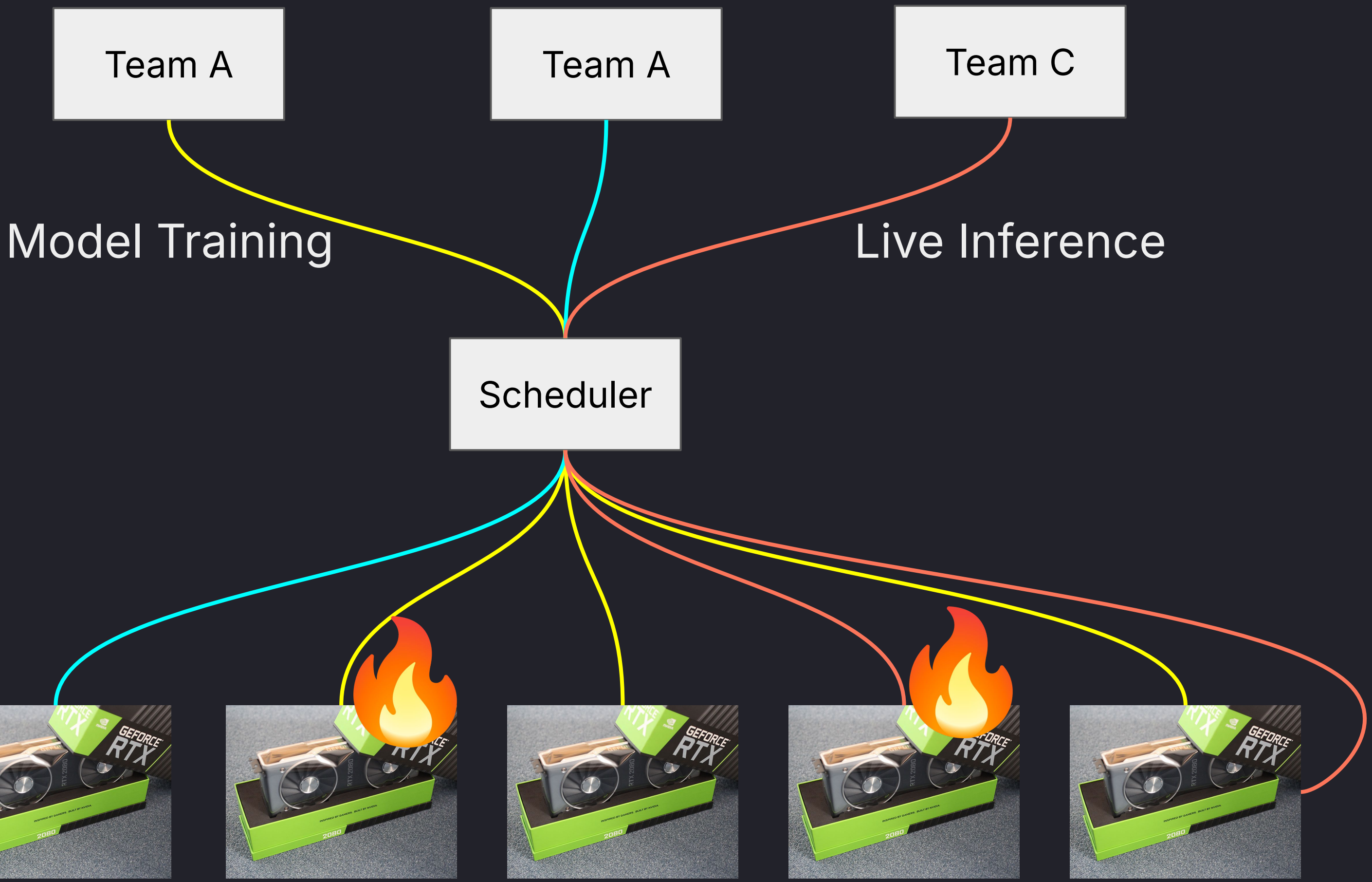

GEFORCE RTX
2080


GEFORCE RTX
2080


GEFORCE RTX
2080


GEFORCE RTX
2080

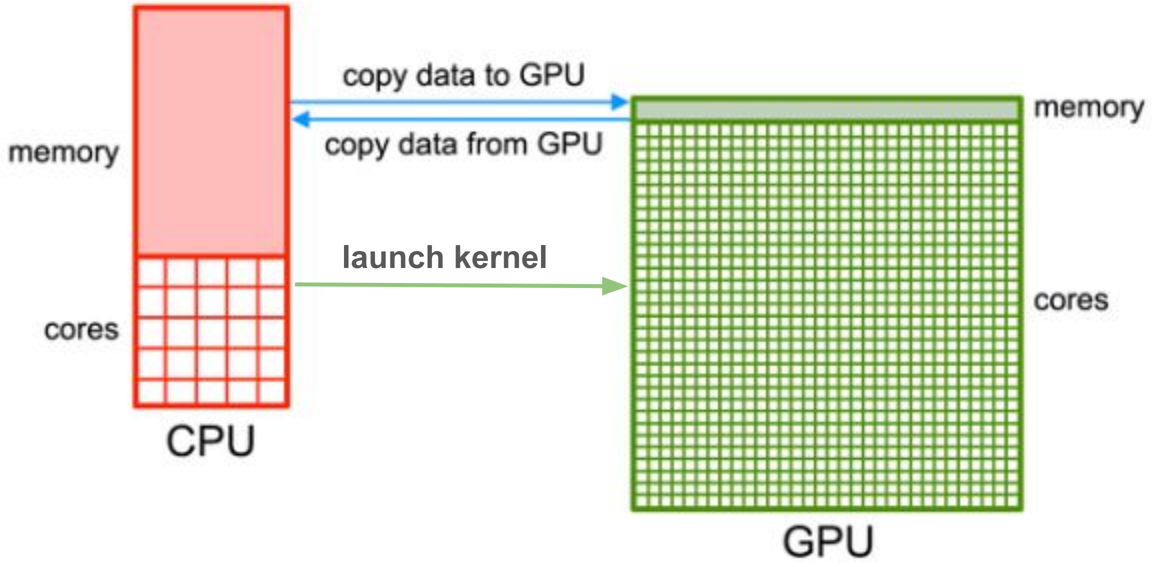

Team A
Team A
Team C
Model Training
Live Inference
Scheduler


Team A
Team A
Team C
Model Training
Live Inference
Scheduler

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.

https://ai.meta.com/research/publications/the-llama-3-herd-of-models/

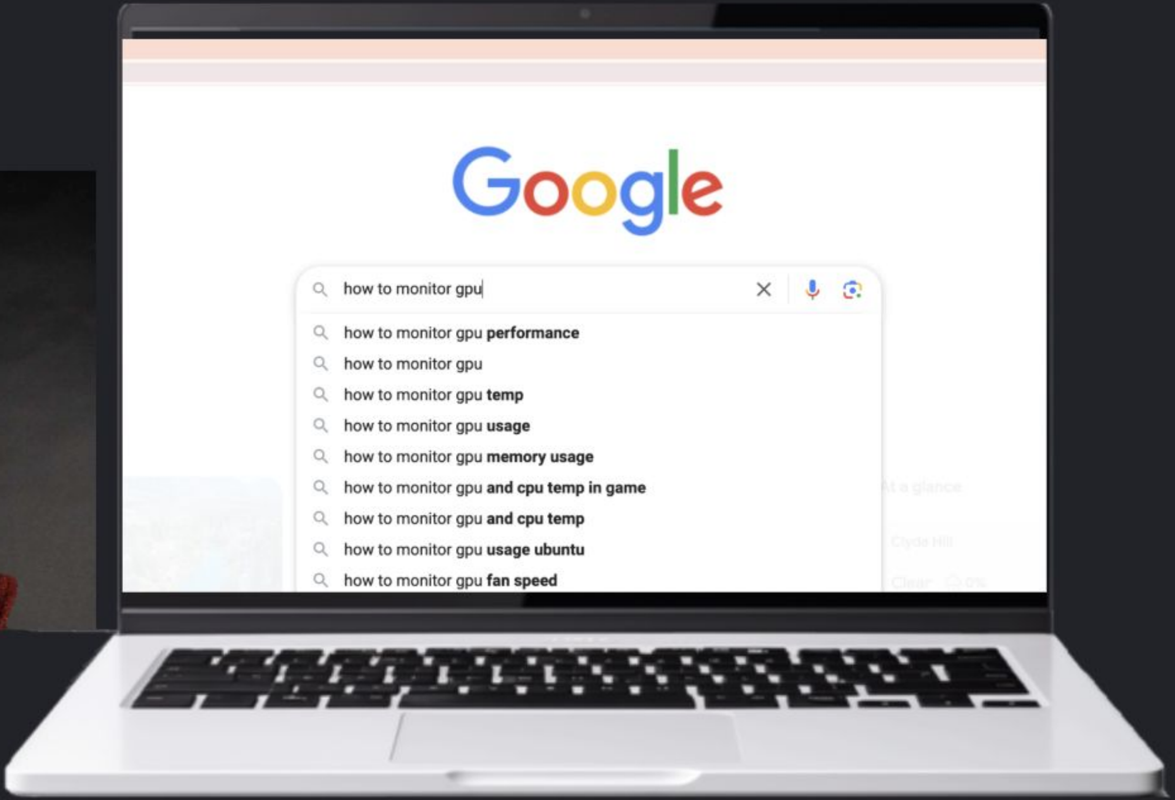


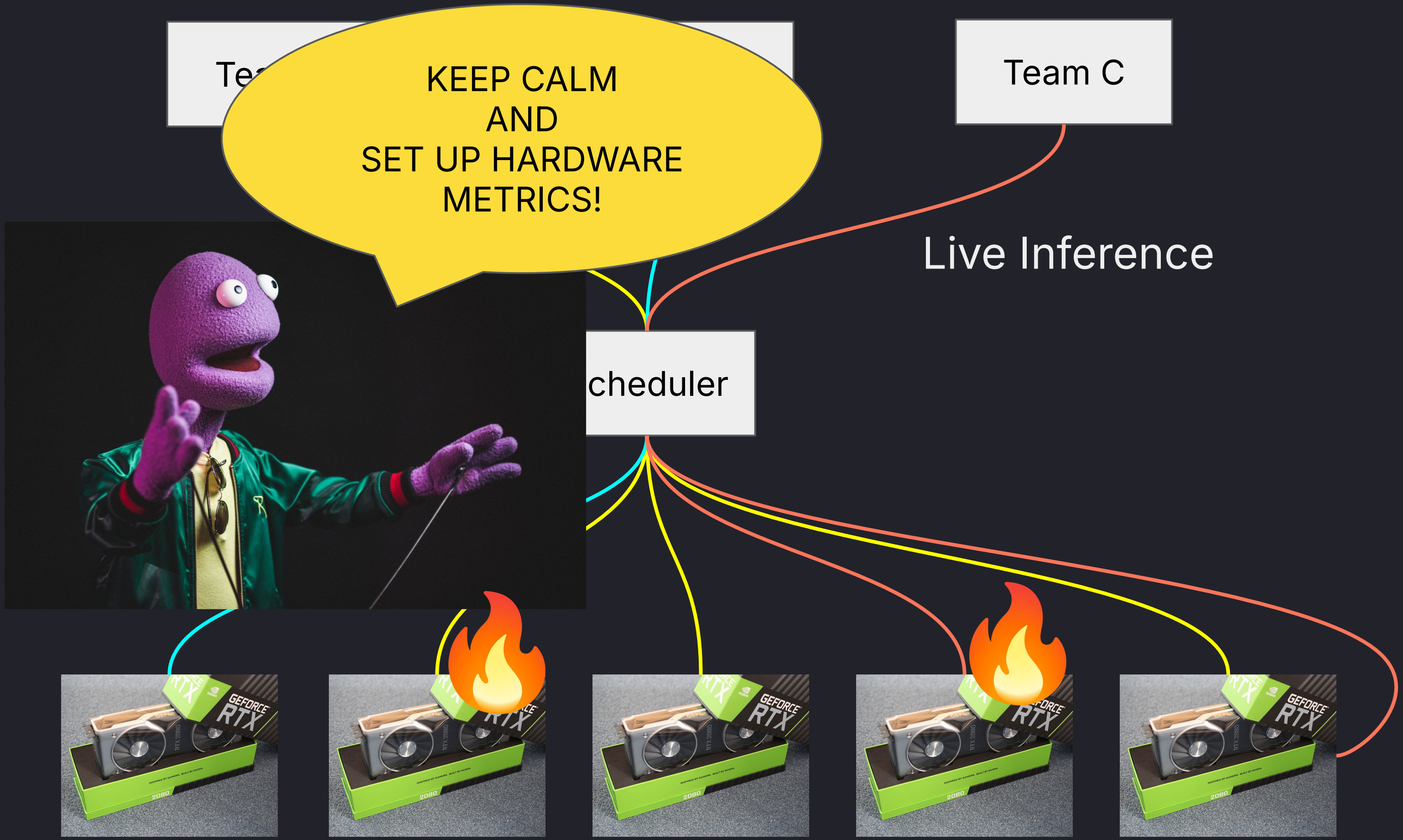

```
data = open("input.dat");          # read the data on the CPU
copyToGPU(data);                   # copy the data to the GPU
matrix_inverse(data.gpu);          # perform a matrix operation on the GPU
copyFromGPU(data);                 # copy the resulting output to the CPU
write(data, "output.dat");         # write the output to file on the CPU
```

https://researchcomputing.princeton.edu/support/knowledge-base/gpu-computing#GPU-Job-Statistics

CPU is the orchestrator for GPU tasks

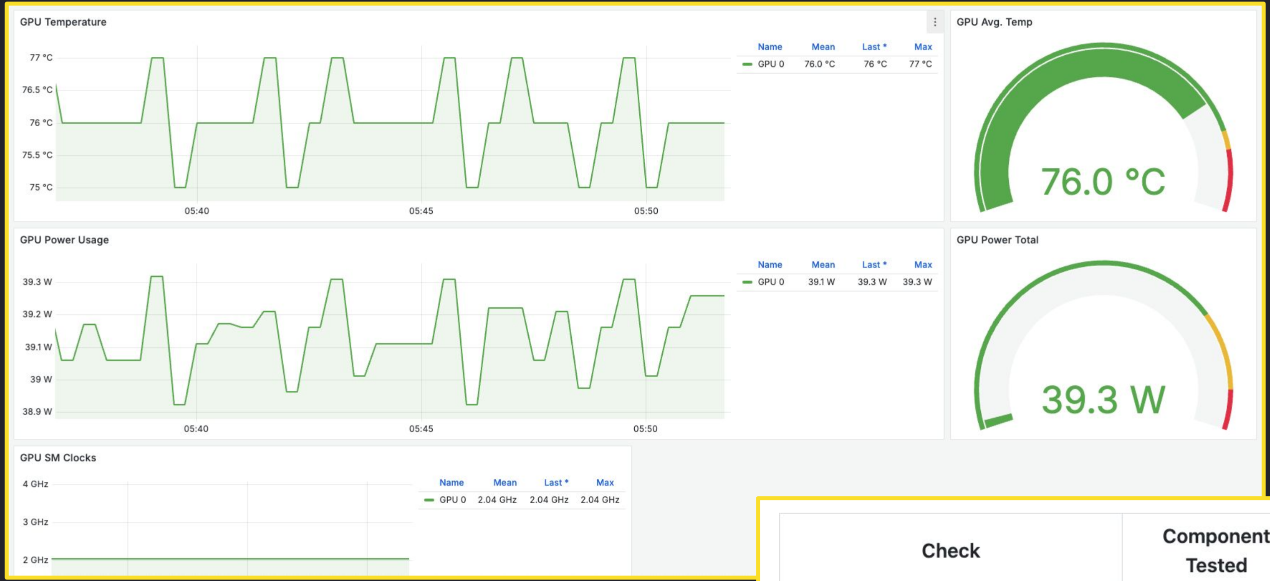
- Kernels are launched from the CPU
  - How many kernels were launched?
  - What are the dependencies between different kernels?
- Memory is allocated and deallocated from the CPU side
  - How much memory was allocated? Was it deallocated?
  - Data transfers are usually done async while other computational tasks are underway.


Google
how to monitor gpu
how to monitor gpu performance
how to monitor gpu
how to monitor gpu temp
how to monitor gpu usage
how to monitor gpu memory usage
how to monitor gpu and cpu temp in game
how to monitor gpu and cpu temp
how to monitor gpu usage ubuntu
how to monitor gpu fan speed


KEEP CALM
AND
SET UP HARDWARE
METRICS!
Team C
Live Inference
cheduler
GEFORCE RTX

Nvidia DCGM exporter
Slurm Job exporter

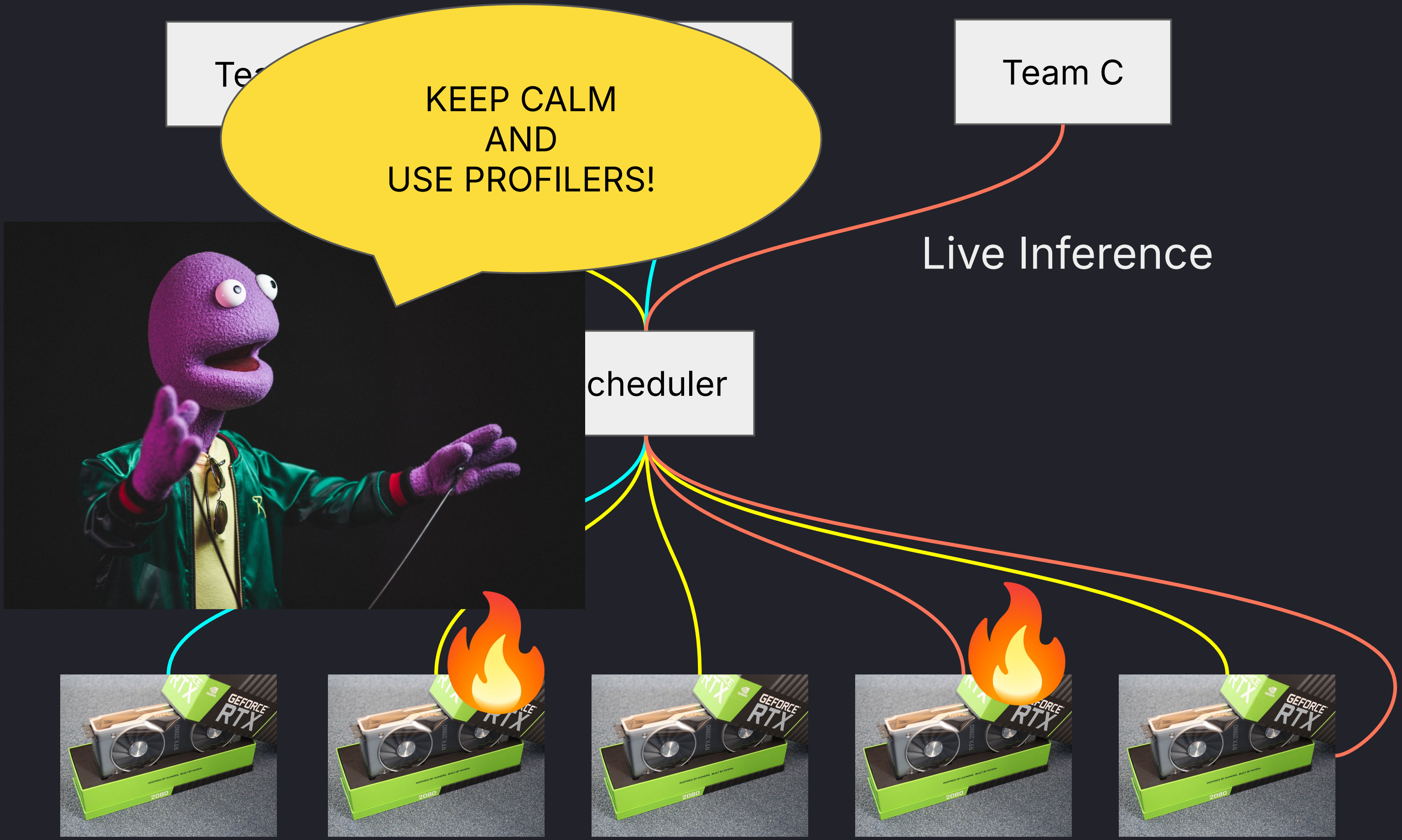
Azure HPC Node Health

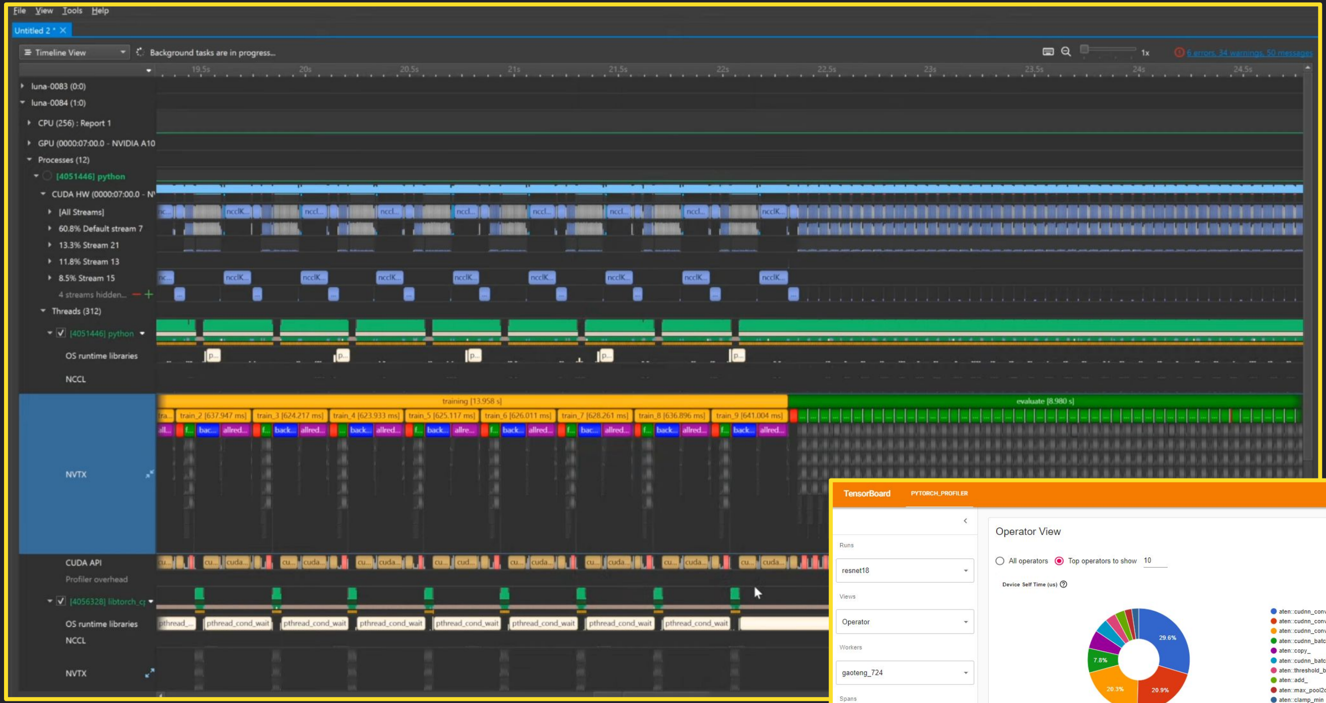


Check	Component Tested	nd96asr_v4 expected	nd96amsr_a100_v4 expected	nd96isr_h1 expect
check_gpu_count	GPU count	8	8	8
check_nvlink_status	NVlink	no inactive links	no inactive links	no inactive
check_gpu_xid	GPU XID errors	not present	not present	not present
check_nvsmi_healthmon	Nvidia-smi GPU health check	pass	pass	pass
check_gpu_bandwidth	GPU DtH/HtD bandwidth	23 GB/s	23 GB/s	52 GB/s
check_gpu_ecc	GPU Mem Errors (ECC)	20000000	20000000	20000000

Problem

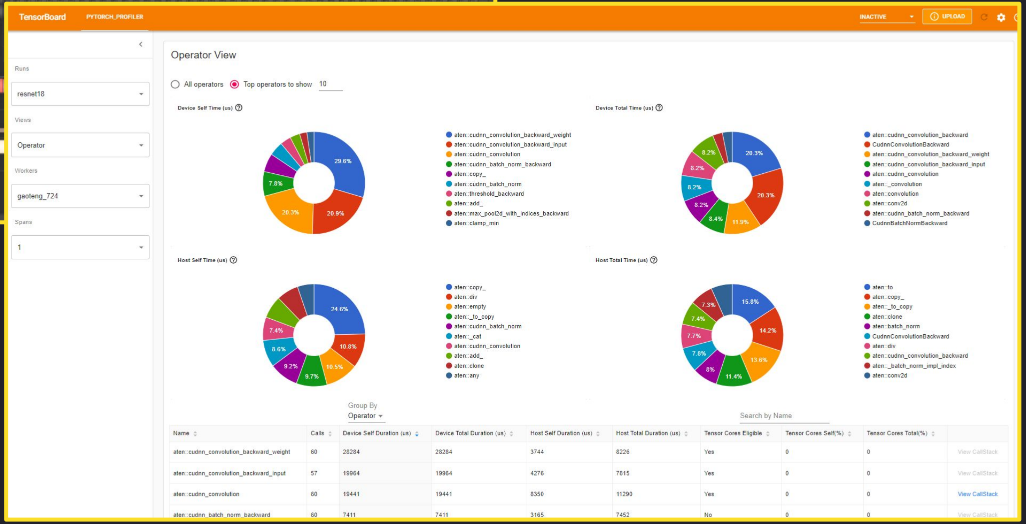
- Hardware metrics are not enough
    - They are helpful to know which GPU / Job failed and the failure states


Tea
KEEP CALM
AND
USE PROFILERS!
Team C
Live Inference
cheduler
GEFORCE RTX
GEFORCE RTX
GEFORCE RTX
GEFORCE RTX



[Nsight systems]()

[Pytorch Profiler]()



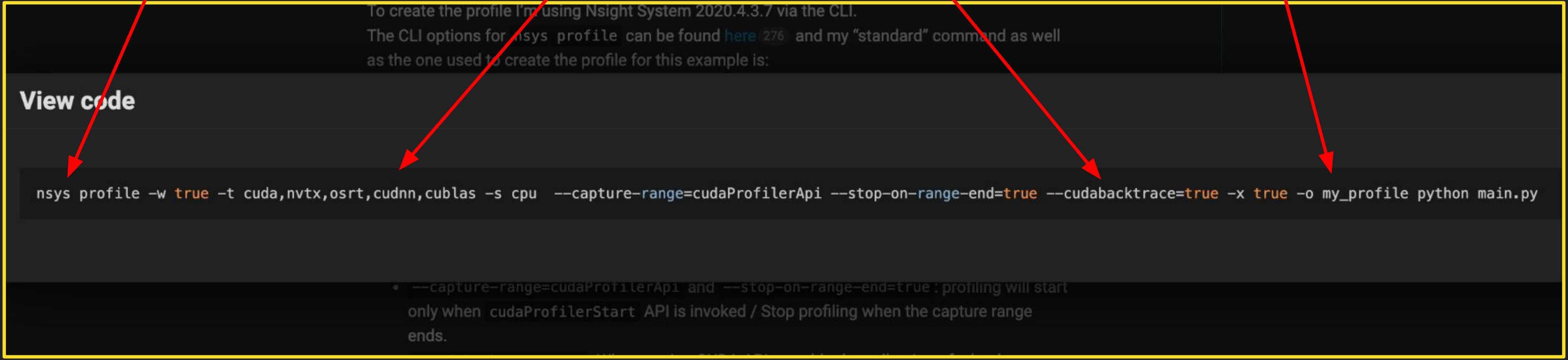
Existing profiling tools

runs in the app process

Libraries to profile

Sample cpu call stack

output on local machine



To create the profile I'm using Nsight System 2020.4.3.7 via the CLI.
The CLI options for nsys profile can be found here 276 and my "standard" command as well as the one used to create the profile for this example is:

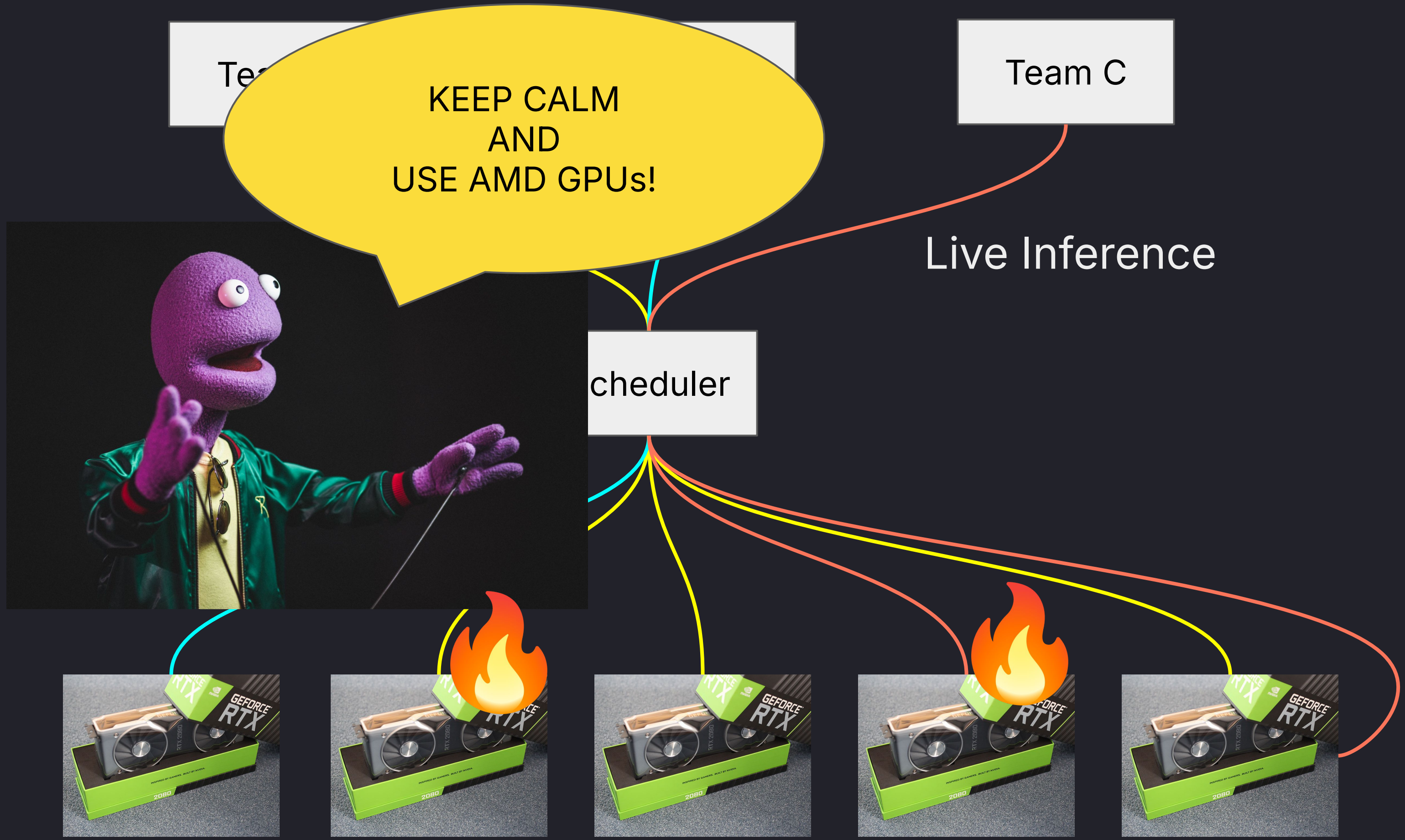
View code

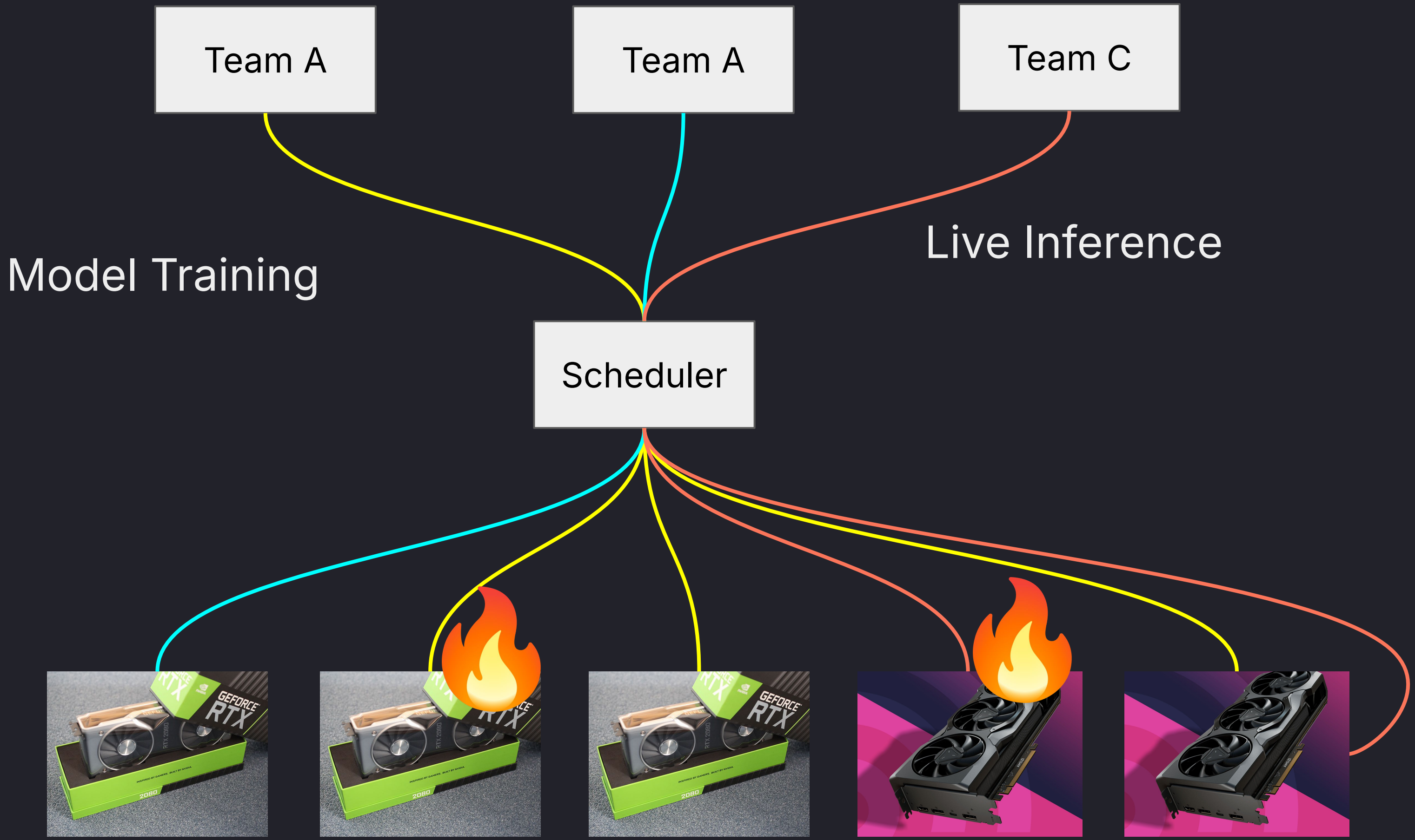
```
nsys profile -w true -t cuda,nvtx,osrt,cudnn,cublas -s cpu  --capture-range=cudaProfilerApi --stop-on-range-end=true --cudabacktrace=true -x true -o my_profile python main.py
```

- --capture-range=cudaProfilerApi and --stop-on-range-end=true : profiling will start only when cudaProfilerStart API is invoked / Stop profiling when the capture range ends.

Problem

- Hardware metrics are not enough
  - They are helpful to know which GPU / Job failed and the failure states
- Limitations with existing GPU profiling
  - Performance overhead
  - Manual instrumentation
  - Lack of CPU context before/after GPU events
  - Big difference in ease of use for GPU vs CPU workloads.


Te
KEEP CALM
AND
USE AMD GPUs!
Team C
Live Inference
cheduler


Team A
Team A
Team C
Model Training
Live Inference
Scheduler

Problem

- Hardware metrics are not enough
  - They are helpful to know which GPU / Job failed and the failure states
- Limitations with existing GPU profiling
  - Performance overhead
  - Manual instrumentation
  - Lack of CPU context before/after GPU events
  - Big difference in ease of use for GPU vs CPU workloads.
- Observability ecosystem for non nvidia GPUs?

Grafana

Beyla

eBPF-based, zero-code
automatic instrumentation
OpenTelemetry tool

Advantages of eBPF

Zero instrumentation

Framework agnostic

Low overhead

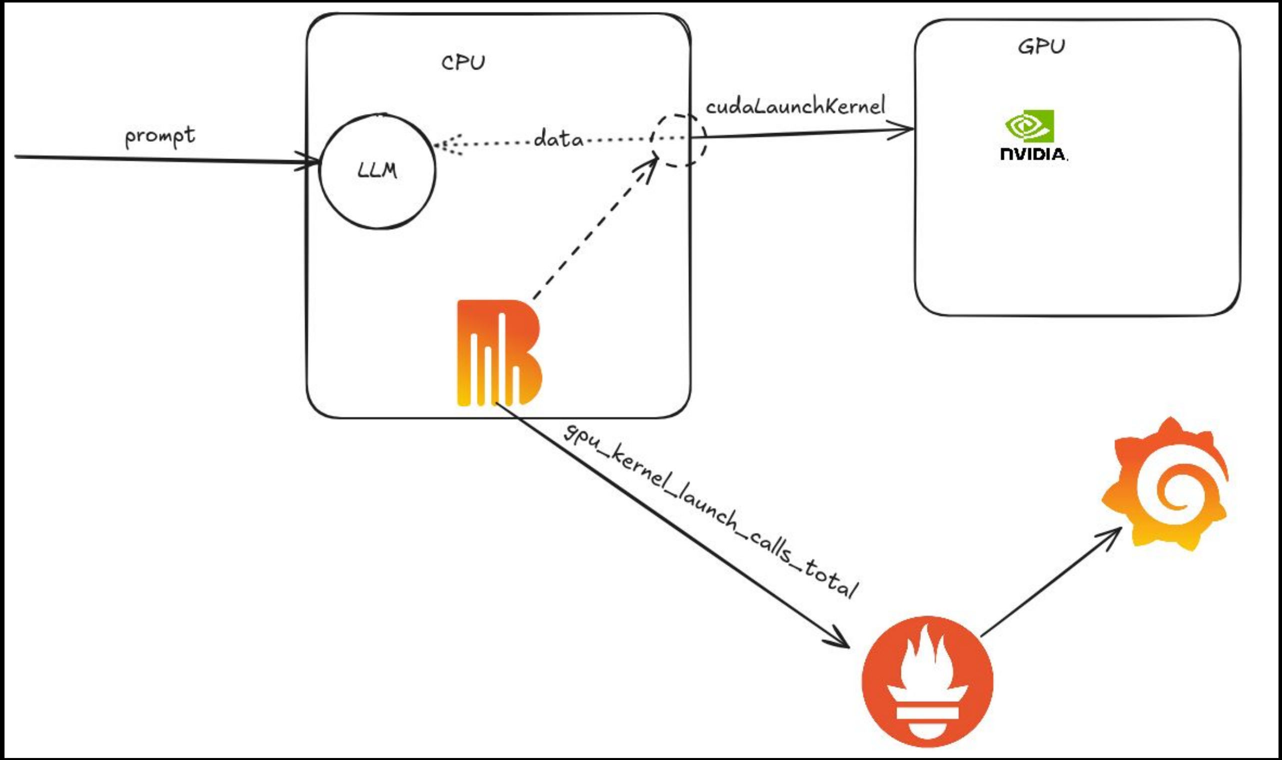
How

EXPERIMENTAL

- Identifying the important cuda calls
- Writing probes and getting data
- Process CUDA libs and module discovery (dynamic linking)
- Access to CPU context before and after GPU calls!

How

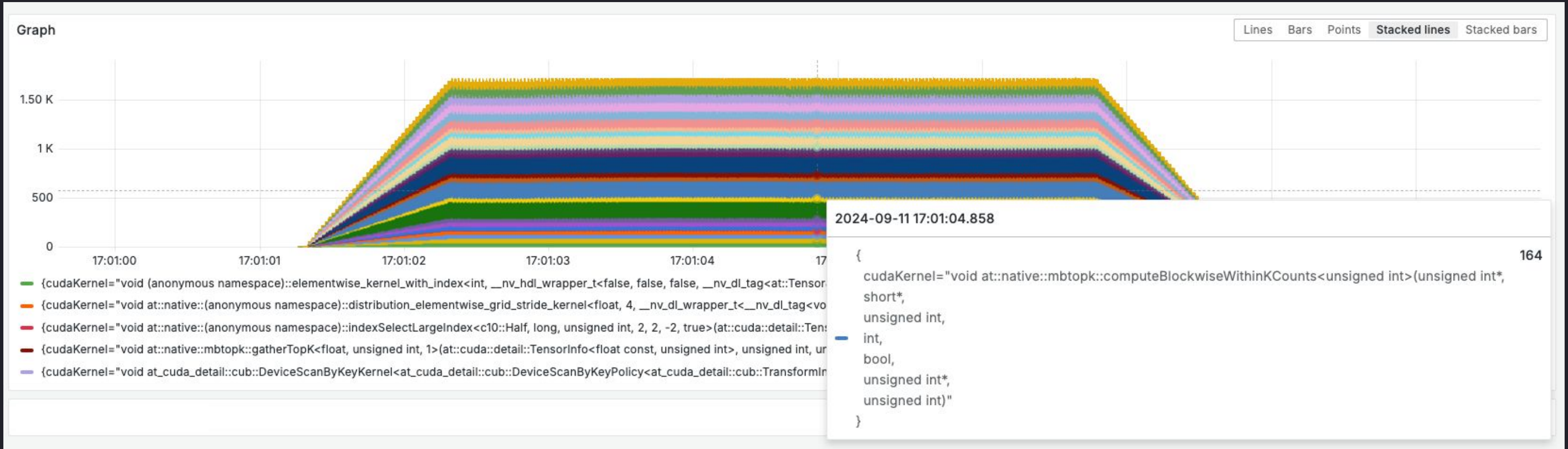
EXPERIMENTAL



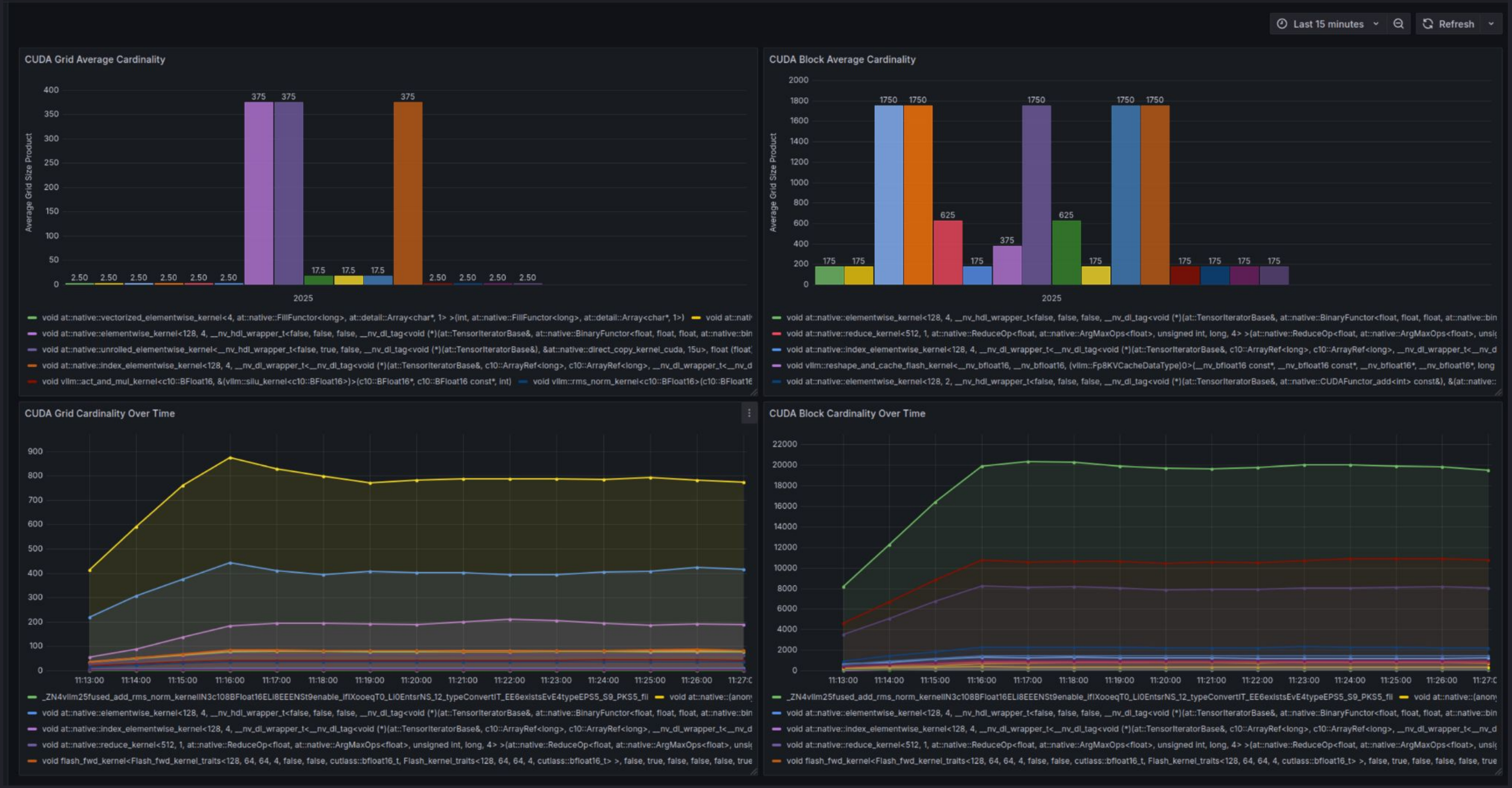
cudaLaunchKernel

```
SEC("uprobe/cudaLaunchKernel")
int BPF_KPROBE(handle_cuda_launch,
        u64 func_off,
        u64 grid_xy,
        u64 grid_z,
        u64 block_xy,
        u64 block_z,
        uintptr_t argv)
```

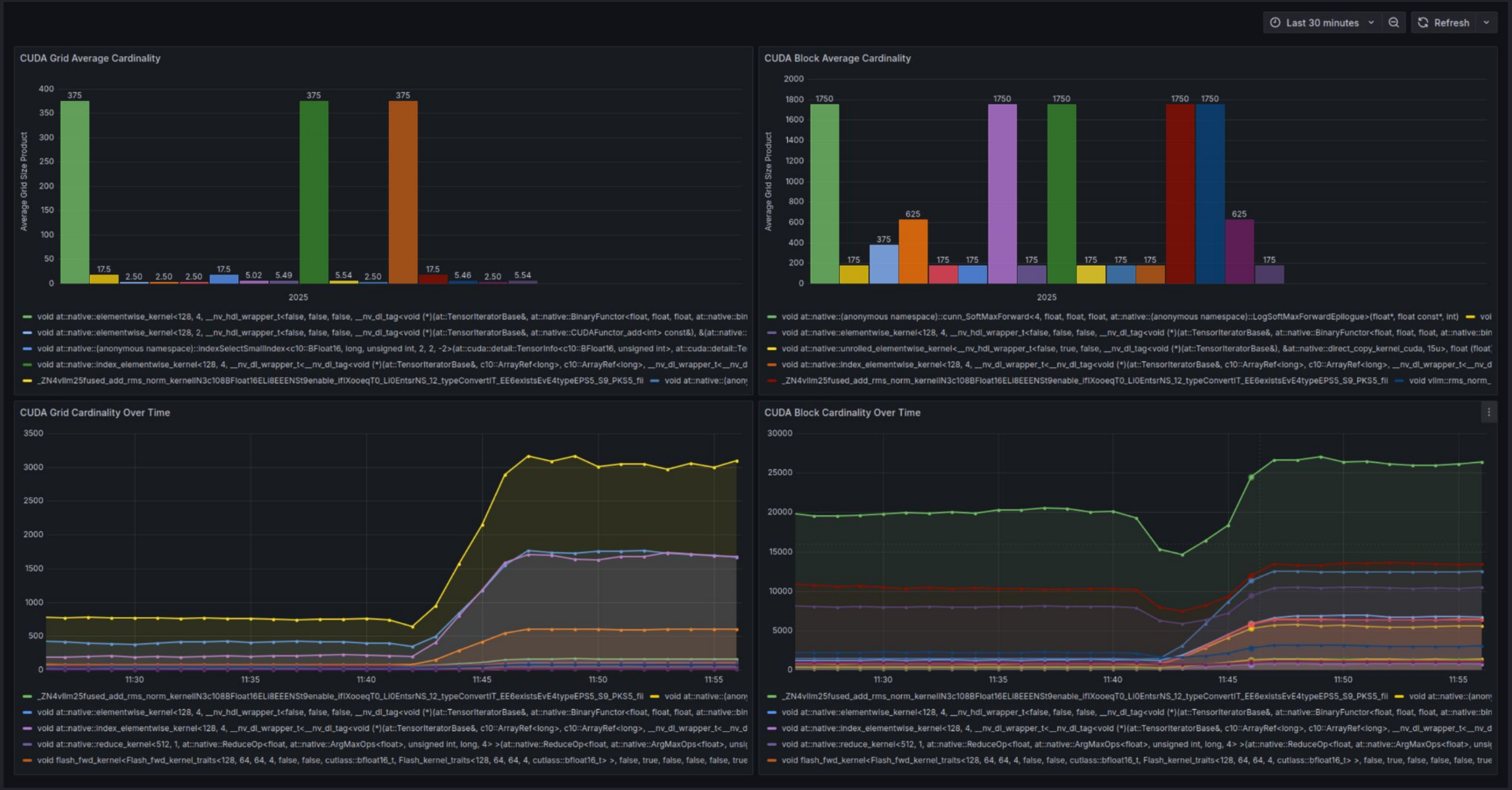
cudaLaunchKernel



cudaLaunchKernel - dimensions



cudaLaunchKernel - dimensions



cudaMalloc

```
SEC("uprobe/cudaMalloc")
int BPF_KPROBE(handle_cuda_malloc, void **devPtr, size_t size)
```

cudaMemCpy (Host to Device, Device to Host)

```
SEC("uprobe/cudaMemcpyAsync")

int BPF_KPROBE(handle_cuda_memcpy, void *dst, void *src, size_t size, u8 kind)
```

cudaMemCpy (Host to Device, Device to Host)

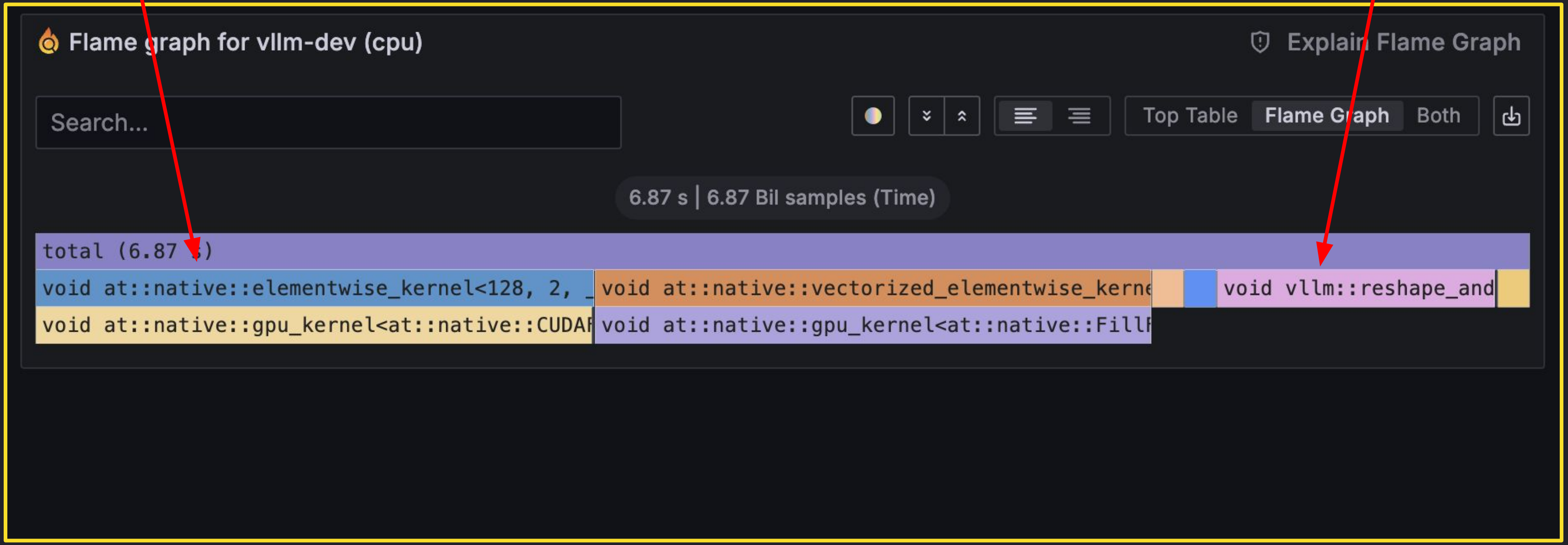
Profiling

```
if (prog_cfg.capture_stack) {
    // Read the Cuda Kernel Launch Stack
    e->ustack_sz =
    bpf_get_stack(ctx, e->ustack, sizeof(e->ustack), BPF_F_USER_STACK) / sizeof(uint64_t);
}
```

Profiling

Pytorch kernels!

vLLM kernels!

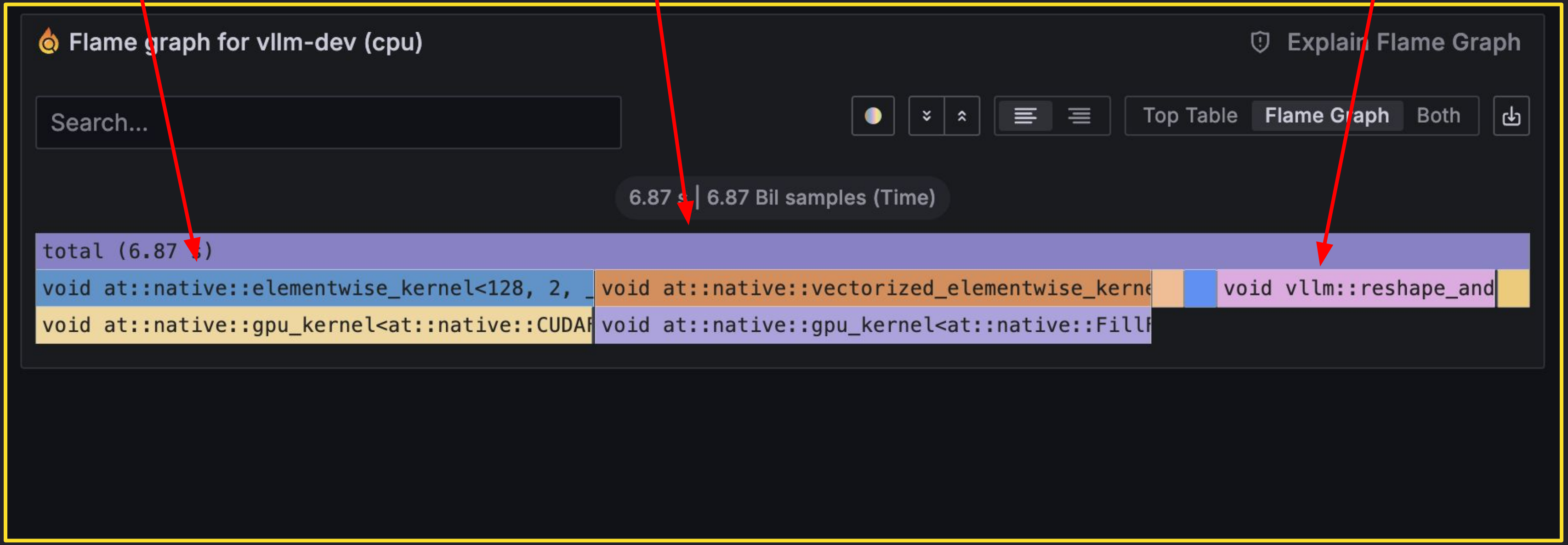


Profiling

Pytorch kernels!

Rest of the stack?

vLLM kernels!



Profiling (Future)

```
#0  0x000079c0d1c75044 in cudaLaunchKernel () from /home/nino/.local/lib/python3.10/site-packages/torch/lib/../../nvidia/cuda_runtime/lib/libcudart.so.12
#1  0x000079c08835d407 in at::native::(anonymous namespace)::index_select_out_cuda_impl<c10::BFloat16>(at::Tensor&, at::Tensor const&, unsigned long, at::Tensor const&)::{la
mbda()#1}::operator()() const::{lambda()#2}::operator()() const () from /home/nino/.local/lib/python3.10/site-packages/torch/lib/libtorch_cuda.so
#2  0x000079c0884411e2 in void at::native::(anonymous namespace)::index_select_out_cuda_impl<c10::BFloat16>(at::Tensor&, at::Tensor const&, unsigned long, at::Tensor const&)
 () from /home/nino/.local/lib/python3.10/site-packages/torch/lib/libtorch_cuda.so
#3  0x000079c08820fcf5 in at::native::index_select_out_cuda(at::Tensor const&, long, at::Tensor const&, at::Tensor&) ()
   from /home/nino/.local/lib/python3.10/site-packages/torch/lib/libtorch_cuda.so
#4  0x000079c08820fffa in at::native::index_select_cuda(at::Tensor const&, long, at::Tensor const&) ()
   from /home/nino/.local/lib/python3.10/site-packages/torch/lib/libtorch_cuda.so
#5  0x000079c089747e97 in at::(anonymous namespace)::(anonymous namespace)::wrapper_CUDA__index_select(at::Tensor const&, long, at::Tensor const&) ()
   from /home/nino/.local/lib/python3.10/site-packages/torch/lib/libtorch_cuda.so
#6  0x000079c089747f63 in c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor (at::Tensor const&,
long, at::Tensor const&), &at::(anonymous namespace)::(anonymous namespace)::wrapper_CUDA__index_select>, at::Tensor, c10::guts::typelist::typelist<at::Tensor const&, long,
at::Tensor const&> >, at::Tensor (at::Tensor const&, long, at::Tensor const&)>::call(c10::OperatorKernel*, c10::DispatchKeySet, at::Tensor const&, long, at::Tensor const&)
    () from /home/nino/.local/lib/python3.10/site-packages/torch/lib/libtorch_cuda.so
#7  0x000079c0bd77ba36 in at::_ops::index_select::call(at::Tensor const&, long, at::Tensor const&) ()
   from /home/nino/.local/lib/python3.10/site-packages/torch/lib/libtorch_cpu.so
#8  0x000079c0bceb58f6 in at::native::embedding_symint(at::Tensor const&, at::Tensor const&, c10::SymInt, bool, bool) ()
   from /home/nino/.local/lib/python3.10/site-packages/torch/lib/libtorch_cpu.so
#9  0x000079c0be006f84 in at::(anonymous namespace)::(anonymous namespace)::wrapper_CompositeExplicitAutograd__embedding(at::Tensor const&, at::Tensor const&, c10::SymInt, b
ool, bool) () from /home/nino/.local/lib/python3.10/site-packages/torch/lib/libtorch_cpu.so
#10 0x000079c0be00da25 in c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor (at::Tensor const&,
at::Tensor const&, c10::SymInt, bool, bool), &at::(anonymous namespace)::(anonymous namespace)::wrapper_CompositeExplicitAutograd__embedding>, at::Tensor, c10::guts::typelis
t::typelist<at::Tensor const&, at::Tensor const&, c10::SymInt, bool, bool> >, at::Tensor (at::Tensor const&, at::Tensor const&, c10::SymInt, bool, bool)>::call(c10::Operator
Kernel*, c10::DispatchKeySet, at::Tensor const&, at::Tensor const&, c10::SymInt, bool, bool) ()
   from /home/nino/.local/lib/python3.10/site-packages/torch/lib/libtorch_cpu.so
#11 0x000079c0bdd80155 in at::_ops::embedding::call(at::Tensor const&, at::Tensor const&, c10::SymInt, bool, bool) ()
   from /home/nino/.local/lib/python3.10/site-packages/torch/lib/libtorch_cpu.so
#12 0x000079c0d0660bee in torch::autograd::THPVariable_embedding(_object*, _object*, _object*) ()
   from /home/nino/.local/lib/python3.10/site-packages/torch/lib/libtorch_python.so
```

Limitations with eBPF approach

- No information available on kernel execution time
- No access to GPU hardware APIs to measure Temperature, SM utilization, etc

Recap

- Close the gap between traditional GPU monitoring and modern monitoring solutions
- More architectures? Don’t want to instrument every LLM and every framework
- Capture context before/after GPU call
- Instrument more CUDA operations

Thank you!

Q&A

Have more questions?

Grafana Community Slack: slack.grafana.com

Beyla project: github.com/grafana/beyla

Acknowledgements: GPU profiling at Meta, Nvidia Developer Tools