github.com/eunomia-bpf

Extending eBPF to GPU Device and Driver Contexts

Yusheng Zheng, Tong Yu

eunomia-bpf community

12/11/2025

東京 2025
LINUX PLUMBERS CONFERENCE
TOKYO, JAPAN / DEC. 11-13, 2025

Agenda

Background

- GPU Stack Overview
- Workload Diversity

The Problem

- Static Policies vs Diverse Workloads
- Device Black Boxes
- Existing Solutions & Limitations

Insight

- GPU needs an extensible OS policy interface

Our Exploration

gpu_ext: Extending GPU Driver with eBPF

- Memory & Scheduling struct_ops for resource management

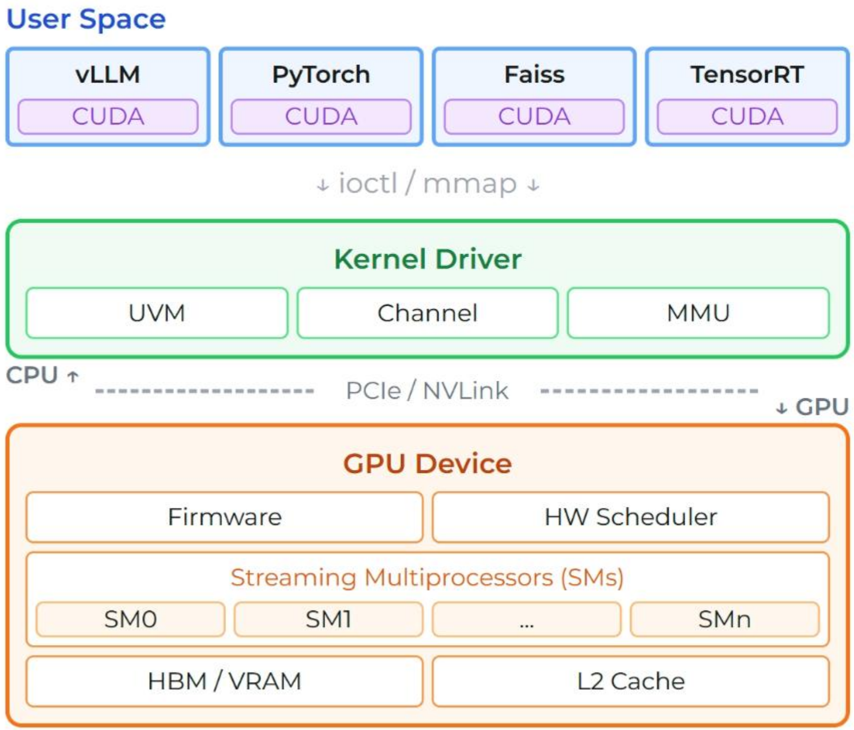
Device eBPF: Offloading eBPF to GPU (bpftime)

- Observability Tools and probes
- Prefetch & Schedule (?)

Cross-layer Coordination

- Cross Device eBPF Maps

Background: GPU Stack Overview



User Space

- Applications: vLLM, PyTorch, Faiss, TensorRT...
- Runtime: CUDA, cuDNN, cuBLAS
- Rich semantic info (model structure, SLOs)

Kernel Driver

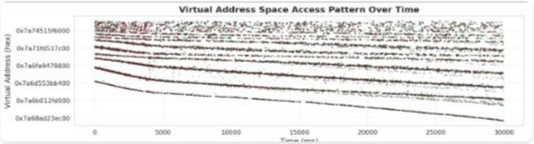
- GPU's "OS component"
- Memory management (UVM, page tables)
- Scheduling (channels, TSG)

GPU Device

- User-defined GPU kernels
- Vendor firmware (proprietary)
- Hardware: SMs, Warps, HBM

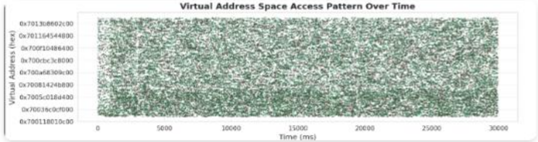
Background: Workload Diversity

Faiss Build



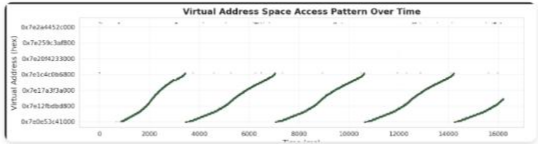
Sequential

Faiss Query



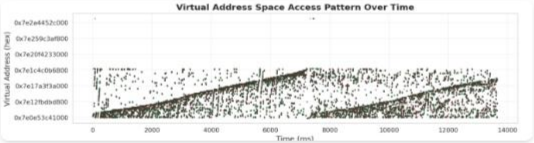
Random

LLM Prefill



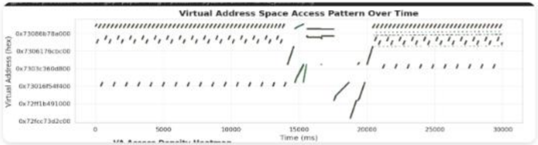
Stride

LLM Decode



Sparse

PyTorch DNN



Periodic

Diverse Resource & Behavior

- **Compute-bound** vs **Memory-bound**
- Different access patterns → different optimal policies

Memory Placement / Offloading

- HBM expensive & limited (RTX 5090: 32GB)
- Models exceed VRAM: MoE, KV-cache in inference / Dataset big in traning

Multi-tenancy Scheduling

- **LC**: LLM inference, needs low P99 latency
- **BE**: Training, needs high throughput
- Conflicts: memory competition, compute interference

The Problem: GPU Software Stack

User-space Runtime (closed-source)

GPU Driver (partially open-source)

- One-Size-Fits-All policies
- Memory: LRU eviction, tree-based prefetch
- Scheduling: Round-robin, fixed timeslice

> Very slow and blackbox policies make people want **kernel bypass** (e.g. UVM offer transparency, but they try to manage memory themselves) like **DPDK**

Vendor Firmware (closed-source, black box)

Applications & Device Code

- Diverse workloads, diverse access patterns

Where can we add extensibility?

- Userspace shim (LD_PRELOAD): change command before they get to driver
- GPU Driver: **policy open-source** after 2022

Existing Solutions For extensibility

User-space Runtimes (vLLM, Sglang, ktransformer) and **Userspace shims** (XSched...)

- Application-bound
- No cross-tenant visibility and control
- Cannot access low level driver mechanisms

Driver Modifications (TimeGraph, Gdev, GPreempt)

- Policies are hard code, hard to maintain and deploy
- Safety risks

Device Profilers (NVBit, Neutrino, CUPTI)

- Design for Read-only
- High overhead

Host eBPF

- GPU device remains a black box
- No programmable hooks in GPU driver for control

Insight: GPU Needs an Extensible OS Policy Interface

GPU Driver is the Right Place

- **Global visibility and control**: coordinate all applications Cross-tenants
- **Privileged access**: controls hardware mechanisms (Replayable Pagefaults, TSG)
- **Transparent**: no app modifications needed

Inspired by **sched_ext/cache_ext**: CPU-side has proven this pattern works

But Host eBPF is Not Enough

- Device side logic is complex
- Device internal execution state invisible
  - Warp divergence, SM load
- Memory sync patterns invisible
- Cannot execute policy logic **inside GPU kernels**

Need to extend eBPF to GPU device contexts

Our Exploration: eBPF for GPU

Part 1: gpu_ext

Extending Linux GPU Driver with eBPF

- Add eBPF attach points to GPU driver
- Memory management hooks in UVM
- Scheduling interface hooks with TSG
- Uses standard eBPF verifier + struct_ops

Part 2: Device eBPF

Running eBPF on GPU Device (bpftime)

- Compile eBPF to PTX/SPIR-V
- Device-side hooks and helpers
- Inject into GPU kernels via dynamic instrumentation
- Cross-layer eBPF Maps

Part 1: gpu_ext

Extending Linux GPU Driver with eBPF

GPU Scheduling Concepts

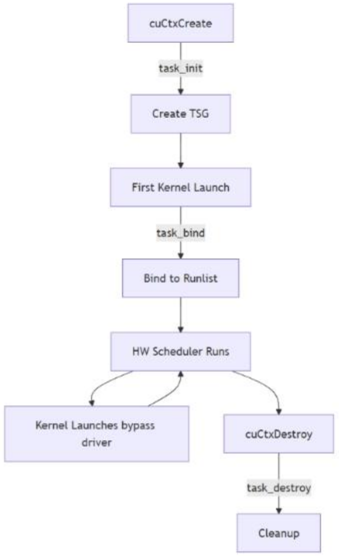
Key Concepts

- **Channel**: Command queue (per CUDA stream)
- **Task Group (TSG)**: Scheduling unit, groups channels
- **Runlist**: HW scheduler's queue of TSGs

Why TSG, Not GPU Kernels?

- **Kernel launch bypasses driver** - userspace writes pushbuffer + doorbell via MMIO
- **Driver only sees TSG lifecycle** - create, bind, destroy

Task Group Lifecycle



Scheduling Parameters

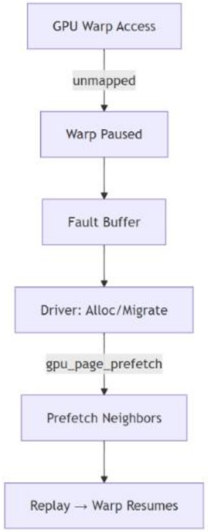
- **Timeslice**: Time before preemption (1s LC / 200μs BE)
- **Interleave Level**: Priority (LOW/MED/HIGH)

GPU Memory Concepts

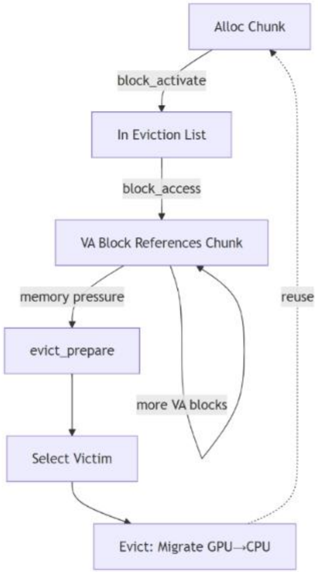
Key Concepts

- **Unified Memory**: CPU & GPU share VA space
- **VA Block**: Virtual address range
- **Chunk**: Physical block (2MB)
- **Replayable Fault**: Warp paused → driver migrates → replay

Page Fault Handling



Chunk-VABlock Lifecycle



Challenge: Expressiveness vs Safety

GPU drivers were **not designed** to expose a programmable interface

- **More Expressiveness** → Expose low-level mechanisms (page tables, command buffers)
  - Risk driver safety and isolation
- **More Safety** → Constrain to high-level abstractions
  - Risk: limits complex memory/scheduling decisions

Our Approach: Narrow, Safe Interface

- Policy **advises**, kernel **decides**
- Expose **structured hooks**, not raw mechanisms; **Bounded operations** via kfuncs
- Implemented as **struct_ops**

Memory Management Interface

```
struct gpu_mem_ops {
  // Eviction hooks (2MB block granularity)
  // Called when block added to eviction list
  // Trigger: first alloc from block, becomes evictable
  int (*gpu_block_activate)(pmm, block, list);
  // Called when any page in block is accessed
  // Trigger: page fault on va_block mapped to this bloc
  int (*gpu_block_access)(pmm, block, list);
  // Called before selecting victim for eviction
  // Trigger: memory pressure, need to free blocks
  // Can: reorder used/unused lists
  int (*gpu_evict_prepare)(pmm, list);
  // Prefetch hooks (page granularity)
  // Called before computing prefetch region
  // Trigger: after page fault handled
  int (*gpu_page_prefetch)(page_index, bitmap_tree,
    max_prefetch_region, result_region);
};
// kfuncs
void bpf_gpu_block_move_head(block, list);
void bpf_gpu_block_move_tail(block, list);
void bpf_gpu_set_prefetch_region(region, first, outer);
```

Policies

The default policy is LRU + tree-based prefetching. We impl:

- LFU, MRU, FIFO eviction
- Stride / sequential prefetch
- Per-process memory priority based on PID
- Application-specific...

Safety: Programmable Cache Model

- Policy can **reorder** eviction list, but **cannot remove**
- Kernel picks final victim
- kfuncs only allow **move_head/move_tail** operations
- Prefetch policy sets region, kernel validates bounds

Scheduling Interface

```
struct gpu_sched_ops {
  // Called when task group is created
  // Trigger: cuCtxCreate / cudaSetDevice
  // Can: set timeslice, interleave level
  // Ctx: tsg_id, engine_type, default_timeslice
  int (*task_init)(struct gpu_task_init_ctx *ctx);
  // Called when task group binds to runlist (ONE-TIME)
  // Trigger: first kernel launch activates the TSG
  // Note: subsequent kernel launches bypass driver!
  // Can: admission control (reject bind)
  int (*task_bind)(struct gpu_task_bind_ctx *ctx);
  // Called when task group is destroyed
  // Trigger: cuCtxDestroy / process exit
  // Can: cleanup BPF map state
  int (*task_destroy)(struct gpu_task_ctx *ctx);
};
// kfuncs to set timeslice, interleave level
void bpf_gpu_set_attr(ctx, u64 us);
void bpf_gpu_reject_bind(ctx);
```

Policy Can Set

- Timeslice (1s for LC, 200µs for BE)
- Interleave level (LOW/MED/HIGH priority)
- Accept/reject task binding

Policy

The default is round-robin / FIFO, we can impl:

- LC vs BE differentiation by process name
- Multi-tenant fairness / isolation

Implementation: Extending NVIDIA Open GPU Modules (POC)

Modifications

- UVM module: ~100 lines instrumentation
- Page fault handler hooks
- Prefetch logic hooks
- TSG lifecycle event hooks

Driver Independence

- ~1000 lines eBPF framework integration
- Uses Linux eBPF verifier + GPU-specific struct_ops/kfunc via BTF
- (May be **extracted** as standalone module)

POC Code: github.com/eunomia-bpf/gpu_ext_policy (eBPF policies) | github.com/eunomia-bpf/gpu_ext-kernel-modules (kernel modules)

Use Cases Summary

Single Application

Workload	Policy	Speedup
LLM Expert (llama.cpp)	Sequential prefetch + LFU eviction	**~4x** decode speedup vs default framework offloading
KV-cache (vLLM)	LFU eviction + stride prefetch	**~1.5x** less TTFT vs default framework offloading, close to LMCache

Key: 1) Hardware faster / sofware algorithm old -> Need to do more prefetching 2) Tree-based prefetch not optimal for LLM/ML (ALso tested with GNN / Vector DB)

Multi-Process

Scenario	Policy	Improvement
LC+BE Scheduling	LC 1s / BE 200µs timeslice	**95%** P99 ↓
Memory Priority	HP more prefetch and eviction protection, LP less	**55-92%** time ↓

Key: Default policy does not allow different process has different behavior: we can have priority.

- Compute-bound → Scheduling;
- Memory-bound → Memory policy

Part 2: Device eBPF

Running eBPF on GPU Device (bpftime)

GPU Execution Model Background

What is SIMT?

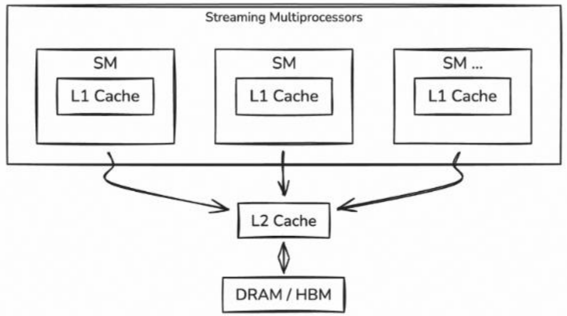
- **S**ingle **I**nstruction **M**ultiple **T**hreads
- Same instruction executes on multiple threads in parallel
- Threads organized into **Warp** (32 threads)
- Same warp threads execute same instruction synchronously
- Different branches → **serialization (Divergence)**

Thread Hierarchy

Thread → Warp (32) → Block → Grid → SM

Feature	CPU	GPU
Thread count	Tens	Tens of thousands
Scheduling unit	Single thread	Warp (32 threads)
Branch handling	Prediction	Serialization
Preemption	Full	Limited

GPU Memory Hierarchy



Memory Levels

Level	Speed	Capacity	Scope
Registers	Fastest	KB	Per-thread
Shared Mem	Fast	48-164KB	Per-block
L1 Cache	Fast	128KB	Per-SM
L2 Cache	Medium	MBs	Global
DRAM/HBM	Slow	GBs	Global

- **Coalesced access**: Consecutive accesses merged into single transaction
- **Bank conflict**: Shared memory contention causes serialization
- **Cache miss**: Determines actual memory latency (L2 miss → HBM access ~400 cycles)

What Can GPU eBPF Do?

Fine-grained Profiling

- Instruction-level observability
- Per-thread/warp/SM metrics
- Memory access pattern detection

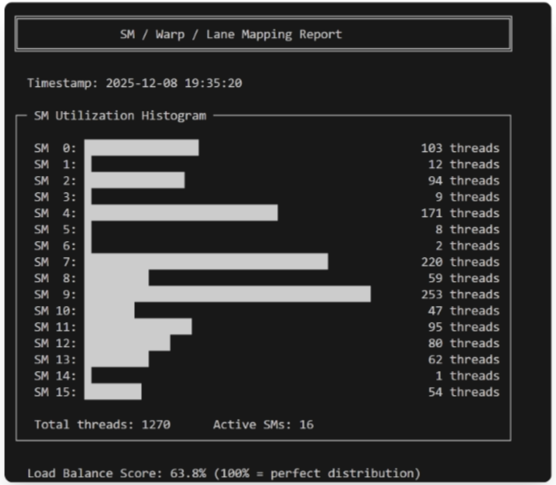
Runtime Adaptation

- Respond to device state
- Safe and Dynamic policy adjustment in GPU kernel

Help Host-side Policies

- Provide device visibility/controlility to host
- Cross-layer coordination

e.g. SM Load Imbalance Trace



127x difference observed between SMs

Traced by bpftime/gpu/threadscheduling

bpftime GPU Support: Maps, Helpers, Attach Types

Attach Types (3)

User can define a compiler pass to define any hook points at instruction level, e.g.:

- `CUDA_PROBE` (entry)
- `CUDA_RETPROBE` (exit)
- `__memcapture` (ld/st)
- `Cluster launch Control Scheduler`

```
__device__ static bool
should_try_steal(State& s,
    int current_block) {
        return true;  // Always try
}
```

GPU Maps (5)

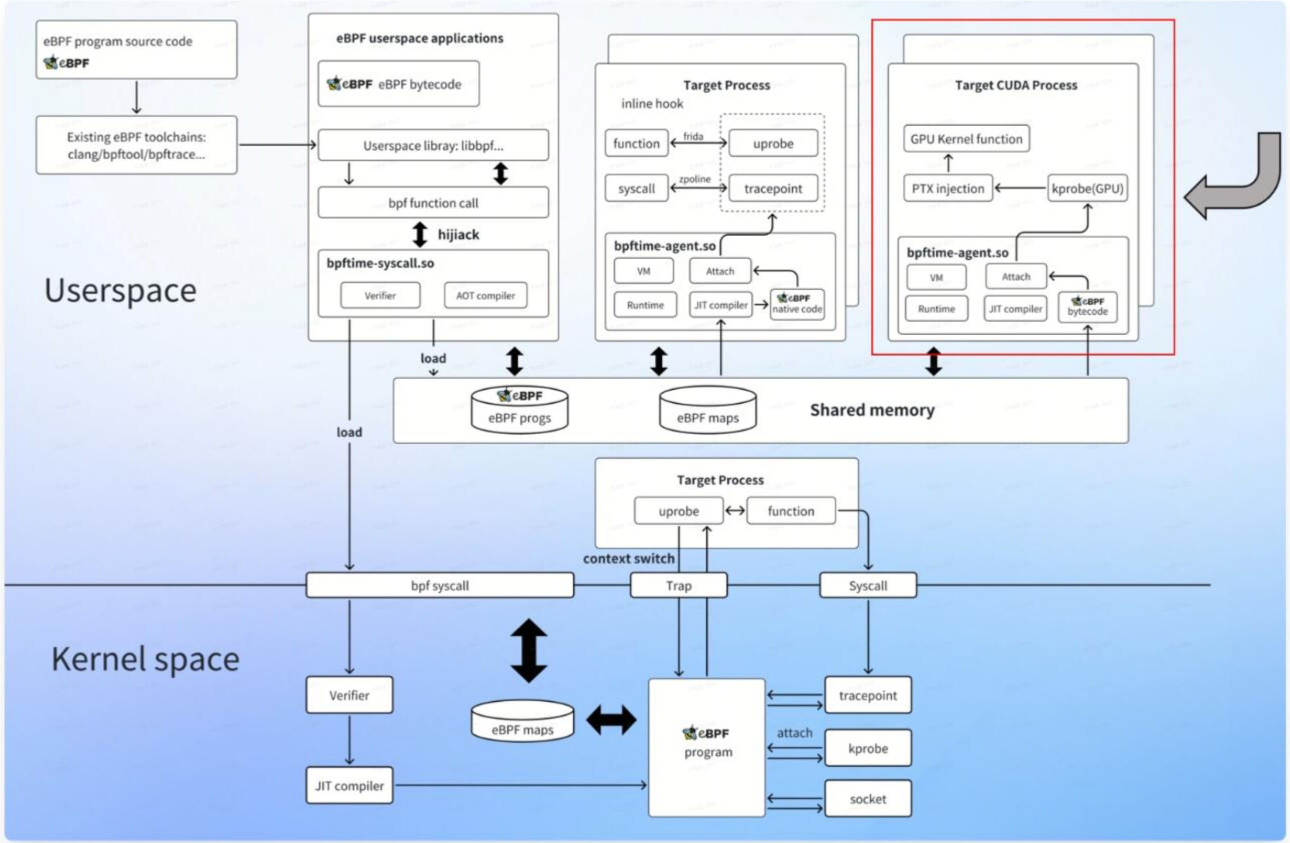
- `PERGPUTD_ARRAY`
- `GPU_ARRAY`
- `GPU_HASH`
- `GPU_RINGBUF`
- `GPU_KERNEL_SHARED`

(Can use all userspace CPU maps with high cost)

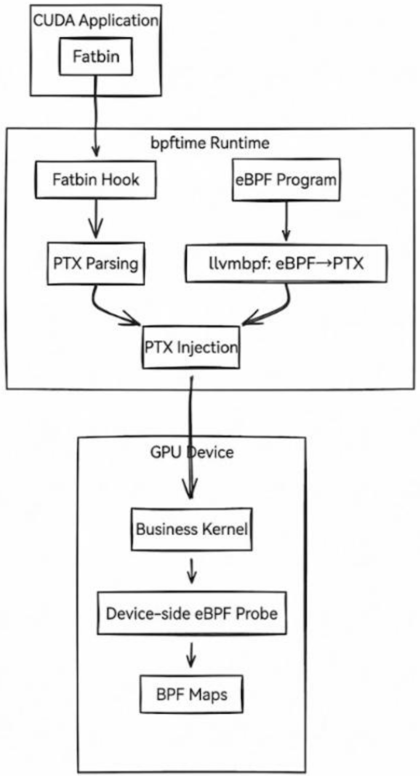
GPU Helpers (15+)

- `ebpf_puts`
- `get_globaltimer`
- `get_block_idx`
- `get_block_dim`
- `get_thread_idx`
- `exit`
- `get_grid_dim`
- `get_sm_id`
- `get_warp_id`
- `get_lane_id`
- - standard userspace BPF helpers (high cost)

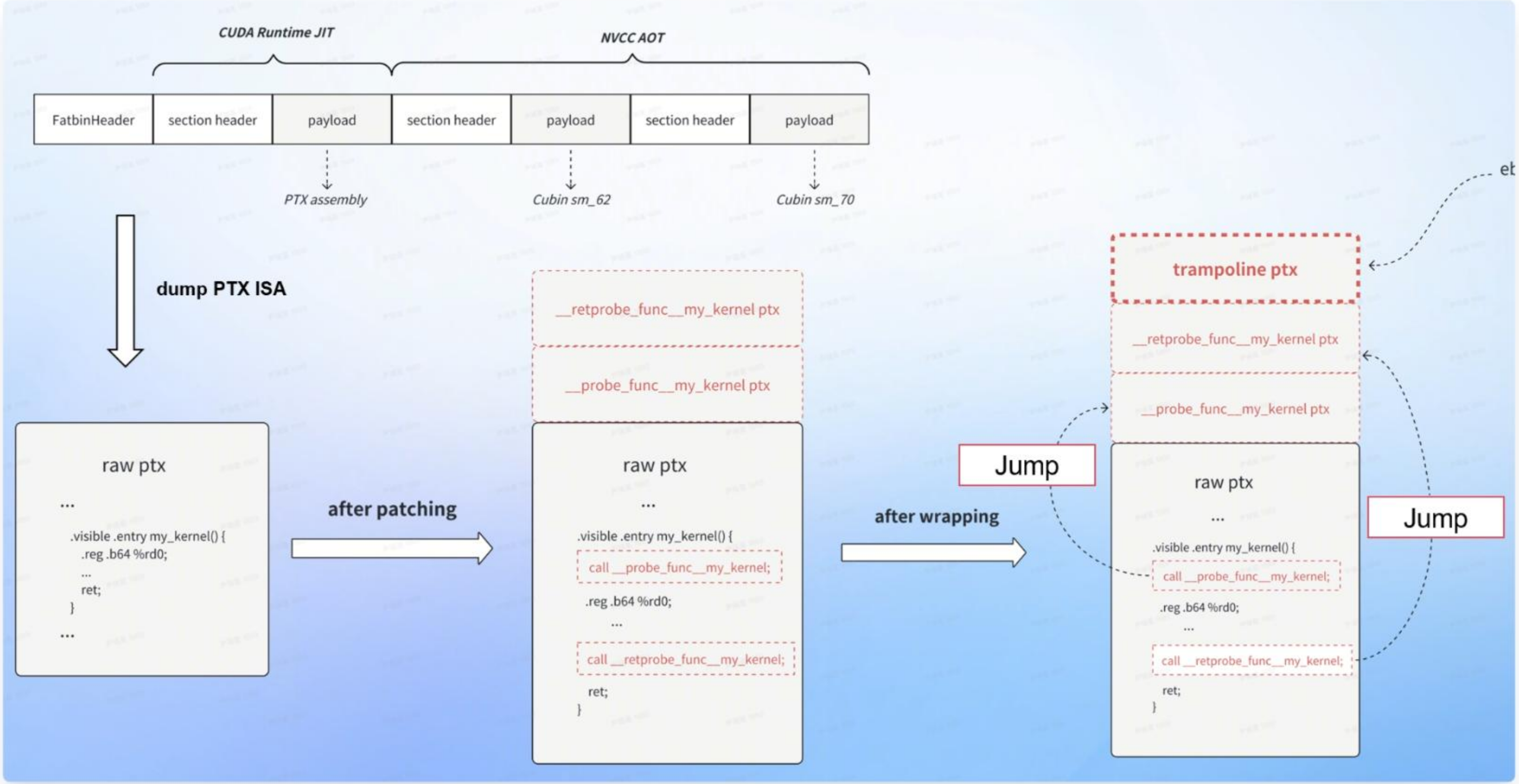
bpftime Architecture (With GPU)



Instrumentation: Fatbin Hook & PTX Injection



PTX Injection: Patching & Wrapping



Example: launchlate - Kernel Launch Latency Profiler

```
BPF_MAP_DEF(BPF_MAP_TYPE_ARRAY, launch_time);

// CPU-side uprobe captures launch time
SEC("uprobe/app:cudaLaunchKernel")
int uprobe_launch(struct pt_regs *ctx) {
    u64 ts_cpu = bpf_ktime_get_ns();
    bpf_map_update_elem(&launch_time, &key, &ts_cpu, BPF_A
}

// GPU-side kprobe captures execution start
SEC("kprobe/_Z9vectorAddPKfS0_Pf")
int kprobe_exec() {
    u64 ts_gpu = bpf_get_globaltimer();
    u64 *ts_cpu = bpf_map_lookup_elem(&launch_time, &key);
    u64 latency = ts_gpu - *ts_cpu;
    // Update histogram...
}
```

Problem

CUPTI shows kernel "started" quickly, but it's slow. Why?

Hidden issue: Thread blocks competing for SMs with other kernels (multi-process, multi-stream)

- **CUPTI sees**: Kernel start/end time (looks fine)
- **Reality**: Many blocks waiting for SM resources
- **bpftime**: Per-thread block/warp scheduling timestamp inside kernel

How It Works

1. **CPU uprobe**: Record T1 at `cudaLaunchKernel()`
2. **GPU kprobe**: Record T2 **per-thread block** at kernel entry
3. See **when each thread block gets scheduled**

Optimizations

Warp-level Execution

Problem: Per-thread eBPF causes warp divergence & bandwidth waste

Solution: Execute eBPF **once per warp** (32 threads), not per thread

- Warp leader executes, broadcasts result / updates maps
- Reduces overhead by **60-81%** vs naive injection
- Avoids divergence and deadlock risks

Hierarchical Map Placement

Problem: PCIe latency ~40µs vs GPU local ~100ns (**400-1000x difference**)

Solution: Logically Verify once, place at runtime

Data Type	Placement
Hot state (frequent)	GPU local, batch sync
Cold config	Host DRAM
Bidirectional	Hierarchical shards

- Relaxed consistency: staleness affects optimality, not correctness

Performance: Observability Tools Overhead

Tested on a P40 GPU with llama.cpp 1B inference.

Tool	LOC	bpftime	NVBit
kernelretsnoop	153	**8%**	85%
threadhist	89	**3%**	87%
launchlate	347	**14%**	93%

Key: Warp-uniform execution achieves **3-14%** overhead vs NVBit's **85-93%**

Problems & Next Steps

Why not extend HMM or DRM?

- Nvidia cuda computing is bypass the DRM.
- HMM is like a interface, mechaism is still in driver.

The design is portable:

- POC in SPIR-v
- ARM also has similar feature set.

More standard API for all GPU drivers?

Cgroups?

Thanks & Questions

POC Code

github.com/eunomia-bpf/gpu_ext_policy | github.com/eunomia-bpf/gpu_ext-kernel-modules

GPU eBPF (bpftime)

github.com/eunomia-bpf/bpftime

Arxiv will be released soon.