Denzel Farmer
Louis Zheng
Aiyu Kamate

Smart-Pinning CUDA Host Allocator

Applying Transparent eBPF Tracing to CUDA Applications

Core Contributions

1. Designed and implemented a **lightweight tracing framework** for CUDA applications

2. Built a proof-of-concept **smart pinning memory allocator** tailored for I/O bound workloads on top of tracing framework

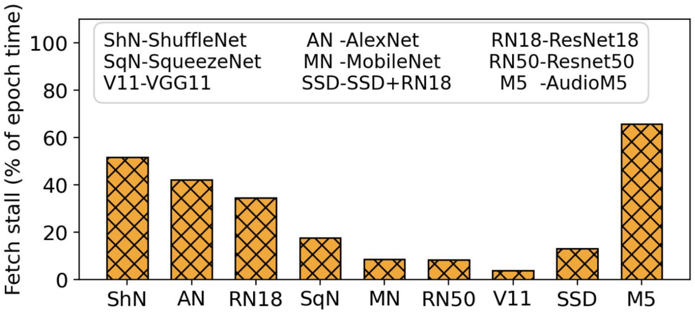
3. Demonstrated **clear performance improvement** in synthetic benchmarks

Motivation

Enhancing Linux-CUDA observability for I/O performance

GPU Workloads Often I/O Bound

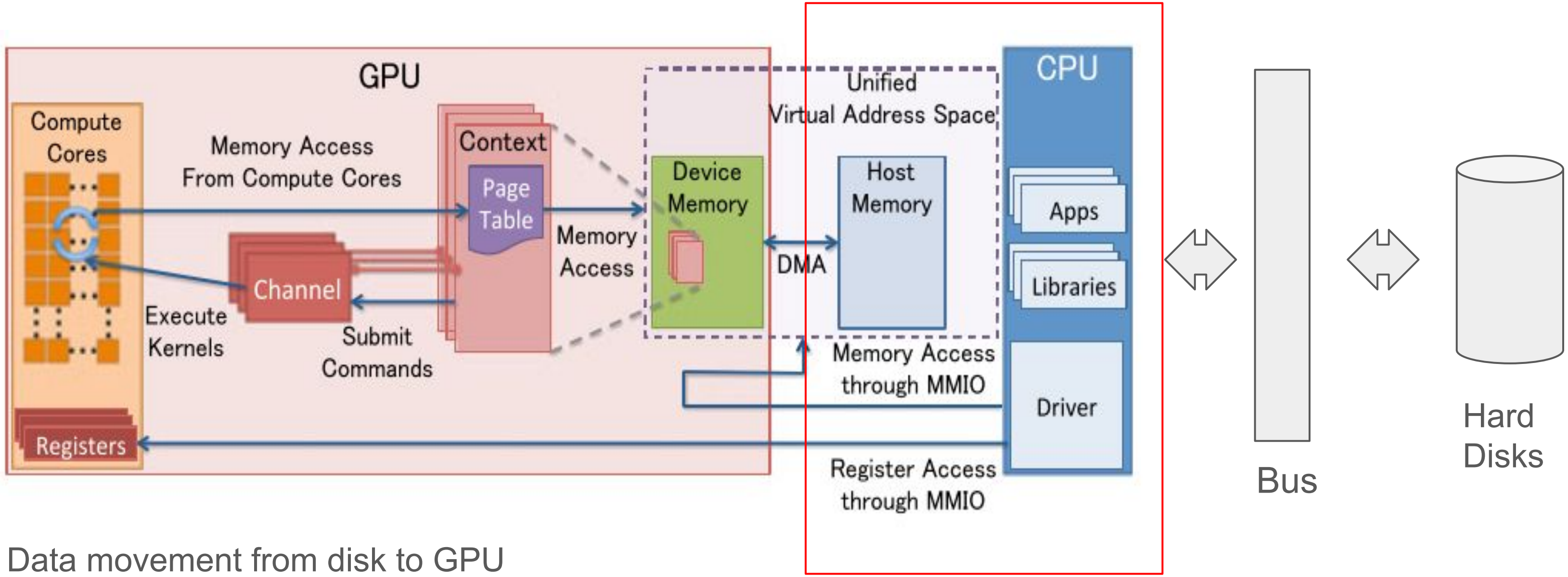
- GPU tasks are **often I/O bound**, most time spent transferring data to/from device
- Optimizing memory transfer and access is a **key problem** in deep learning
- Even under high GPU load, **host-side** (CPU) is **underutilized**, available for to perform optimizations



Training Deep Neural Networks experience significant stalls waiting for I/O - Microsoft

OS Interaction Matters

- Data moves from **high capacity, slow storage** (right) to **low capacity, fast storage** (left)
- Highly parallel GPU cores often starved by **inefficient data transfer pipeline**

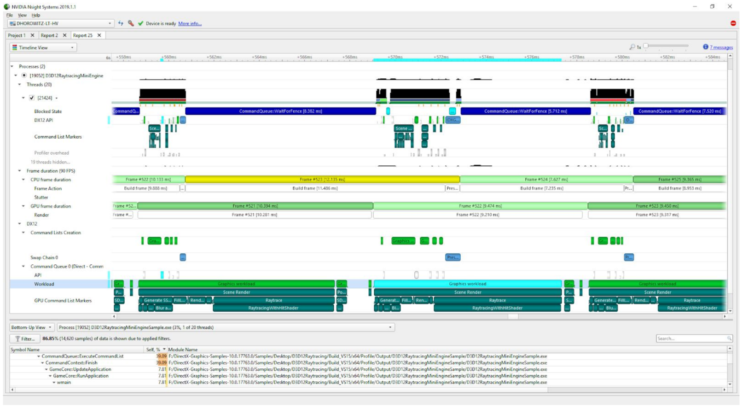


Data movement from disk to GPU

eBPF Tracing Framework

Existing CUDA Profiling

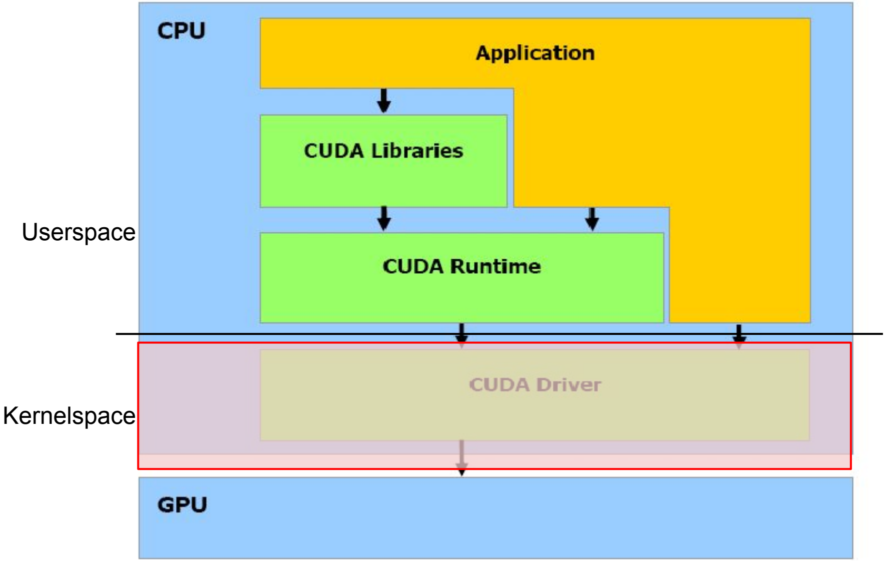
- Application level profiling
  - Ex. Pytorch profiler
- Runtime level profiling
  - Ex. NVCC Profiler
- Very detailed, but some downsides
  - Overhead, lack predictability (not designed for continuous monitoring/advising)
  - Lacking OS context
  - Application/runtime specific



Process [19052] D3D12RaytracingMiniEngineSample.exe (3%, 1 of 20 threads)

86.85% (14,620 samples) of data is shown due to applied filters.

Symbol Name	Self, %	Module Name
CommandQueue::ExecuteCommandList	39.09	F:/DirectX-Graphics-Samples-10.0.17763.0/Samples/Desktop/D3D12Raytracing/Build_VS15/x64/Profile/Output/D3D12RaytracingMiniEngineSample/D3D12RaytracingMiniEngineSample.exe
CommandContext::Finish	39.09	F:/DirectX-Graphics-Samples-10.0.17763.0/Samples/Desktop/D3D12Raytracing/Build_VS15/x64/Profile/Output/D3D12RaytracingMiniEngineSample/D3D12RaytracingMiniEngineSample.exe
GameCore::UpdateApplication	7.81	F:/DirectX-Graphics-Samples-10.0.17763.0/Samples/Desktop/D3D12Raytracing/Build_VS15/x64/Profile/Output/D3D12RaytracingMiniEngineSample/D3D12RaytracingMiniEngineSample.exe
GameCore::RunApplication	7.81	F:/DirectX-Graphics-Samples-10.0.17763.0/Samples/Desktop/D3D12Raytracing/Build_VS15/x64/Profile/Output/D3D12RaytracingMiniEngineSample/D3D12RaytracingMiniEngineSample.exe
wmain	7.81	F:/DirectX-Graphics-Samples-10.0.17763.0/Samples/Desktop/D3D12Raytracing/Build_VS15/x64/Profile/Output/D3D12RaytracingMiniEngineSample/D3D12RaytracingMiniEngineSample.exe



Tracer Principles

- **Linux Context** - profiles in context with OS events and behavior
- **Transparency** - traces unmodified CUDA code, with the option to advise interested components
- **Overhead** - minimizes tracing performance, minimize memory overhead

Tracing Framework Overview

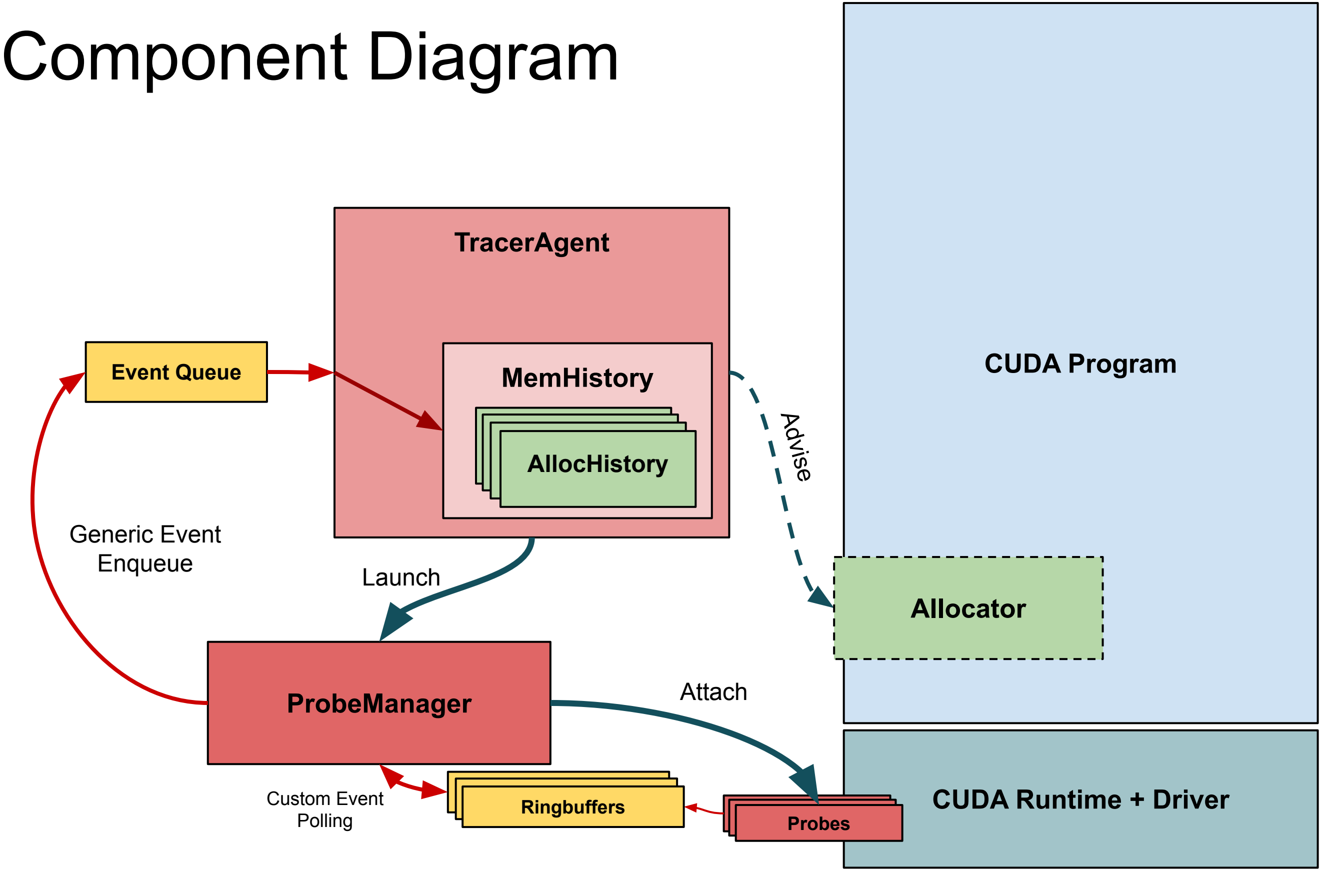
- Library that instruments a target process (or itself) and builds **history of memory events**

- Exposes **API to query event history**, dump event history to database

- Can be **linked in to CUDA applications**, to advise decision making

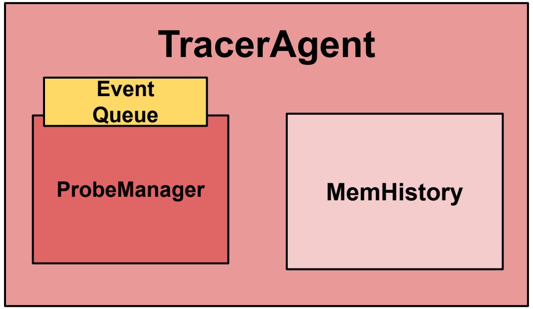
- Or, can build **tracing executable** and manually debug unmodified applications

Tracer Component Diagram

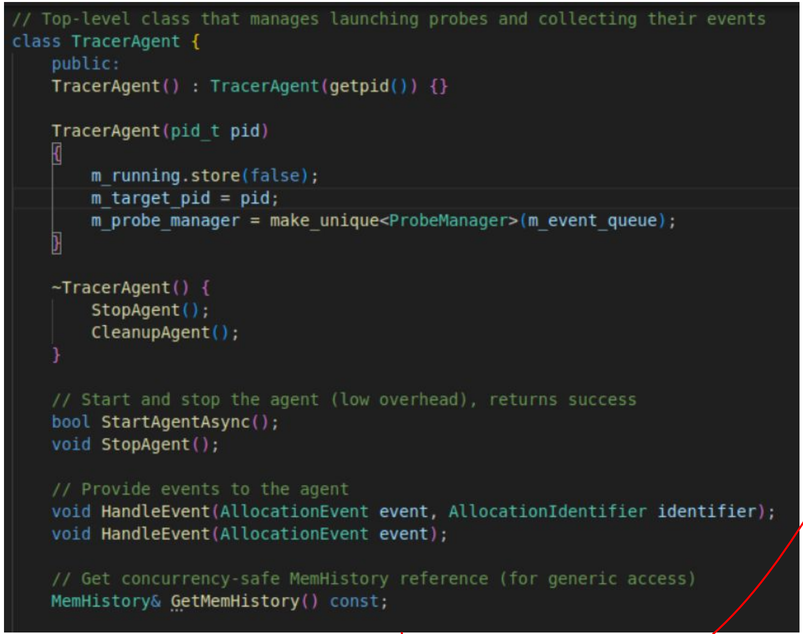


Component: TracerAgent

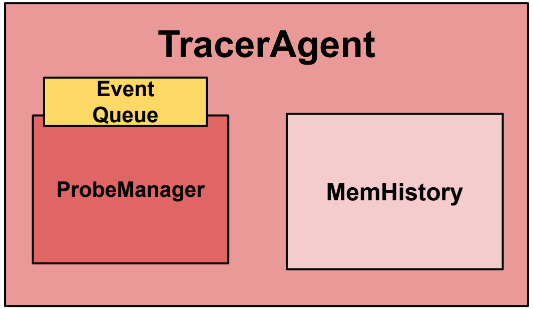
- TracerAgent manages MemHistory and ProbeManager
- Can be built as a standalone executable
    - Allows tracing by PID
- Exposes ‘advisory’ API
    - Applications can fetch hints dynamically
    - Ex. **GetHotspots** and **GetColdspots**
- Dump compacted memory history
    - Accessible via API, dumped to JSON, or binary
    - After each program execution (or, periodically)`

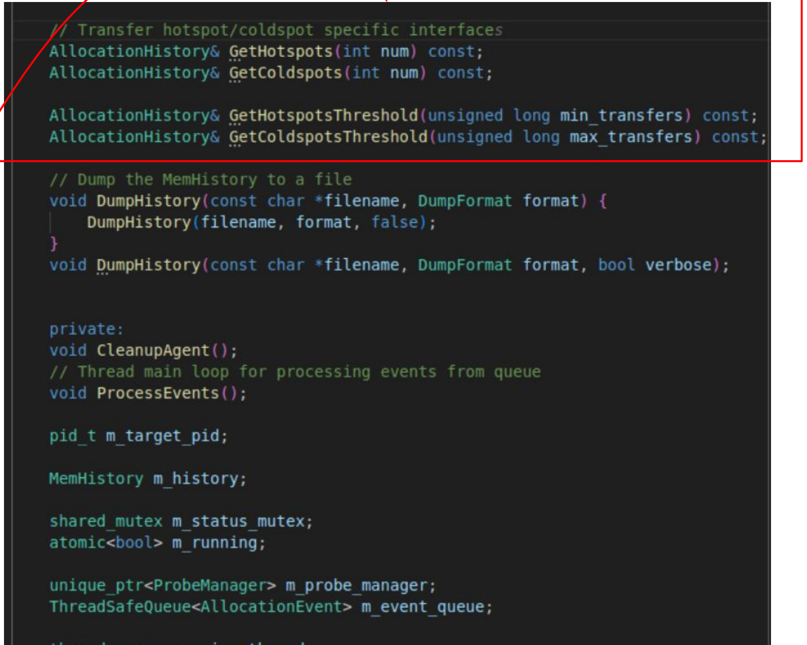


Component: TracerAgent



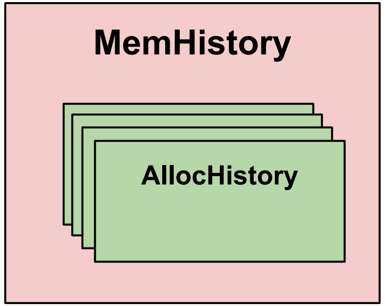
```
// Top-level class that manages launching probes and collecting their events
class TracerAgent {
    public:
    TracerAgent() : TracerAgent(getpid()) {}

    TracerAgent(pid_t pid)
    {
        m_running.store(false);
        m_target_pid = pid;
        m_probe_manager = make_unique<ProbeManager>(m_event_queue);
    }

    ~TracerAgent() {
        StopAgent();
        CleanupAgent();
    }

    // Start and stop the agent (low overhead), returns success
    bool StartAgentAsync();
    void StopAgent();

    // Provide events to the agent
    void HandleEvent(AllocationEvent event, AllocationIdentifier identifier);
    void HandleEvent(AllocationEvent event);

    // Get concurrency-safe MemHistory reference (for generic access)
    MemHistory& GetMemHistory() const;
```





```
// Transfer hotspot/coldspot specific interfaces
AllocationHistory& GetHotspots(int num) const;
AllocationHistory& GetColdspots(int num) const;

AllocationHistory& GetHotspotsThreshold(unsigned long min_transfers) const;
AllocationHistory& GetColdspotsThreshold(unsigned long max_transfers) const;

// Dump the MemHistory to a file
void DumpHistory(const char *filename, DumpFormat format) {
    DumpHistory(filename, format, false);
}
void DumpHistory(const char *filename, DumpFormat format, bool verbose);


private:
void CleanupAgent();
// Thread main loop for processing events from queue
void ProcessEvents();

pid_t m_target_pid;

MemHistory m_history;

shared_mutex m_status_mutex;
atomic<bool> m_running;

unique_ptr<ProbeManager> m_probe_manager;
ThreadSafeQueue<AllocationEvent> m_event_queue;
```

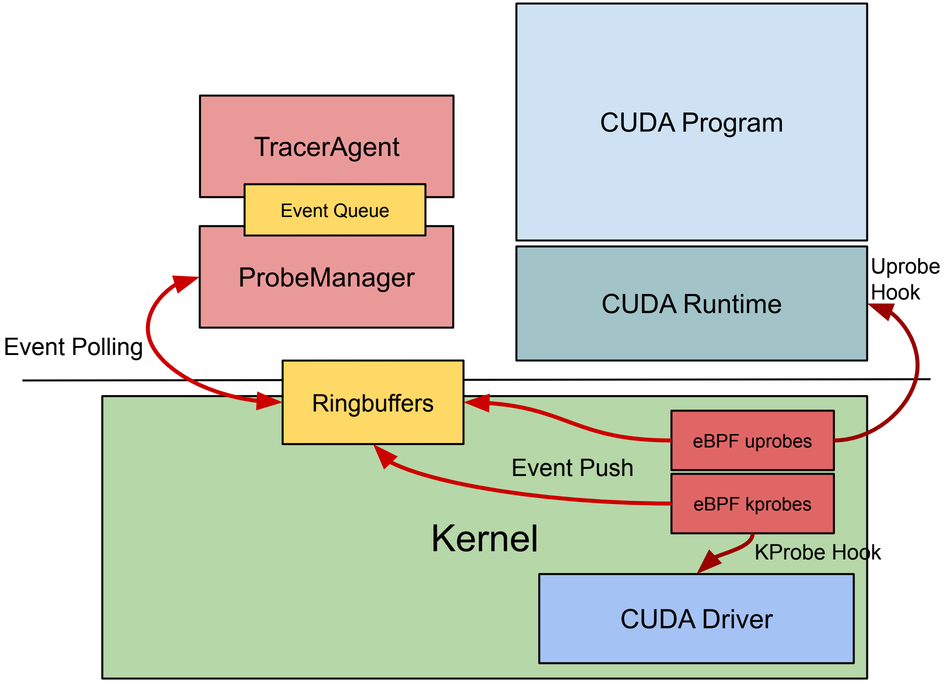
Component: MemHistory



- Tracks various events in the lifetime of an allocation
  - Allocate, page remap, page out, transfer to device, deallocate
- Implemented as sorted set of events, per-allocation
  - Allocations identified by start address, and ‘tag’: return address of allocate call
- Multi-index optimized for access by **transfer count** and **start address**
- Some inefficiencies, but highly optimizable
  - Ex. ‘splitting’ of allocations handled naively–duplicates event chain

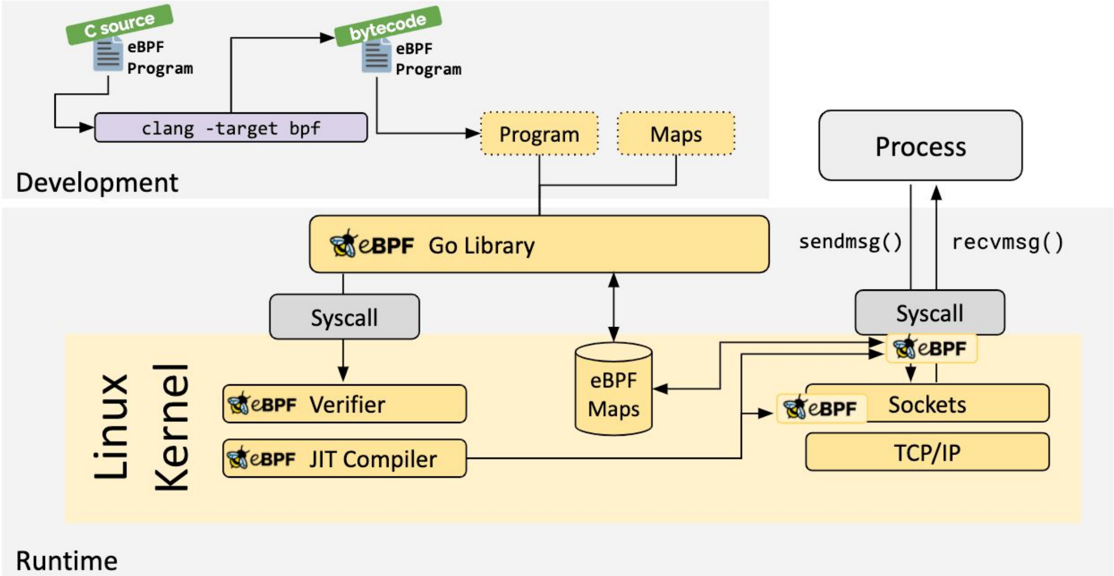
Component: eBPF ProbeManager

- C++ wrapper around eBPF backend
- Creates ringbuffer for each probe
  - Epoll on each ringbuffer
- Merge into generic events
  - Merging drops much information–could expand tracing significantly
- Manage lifetime of probes



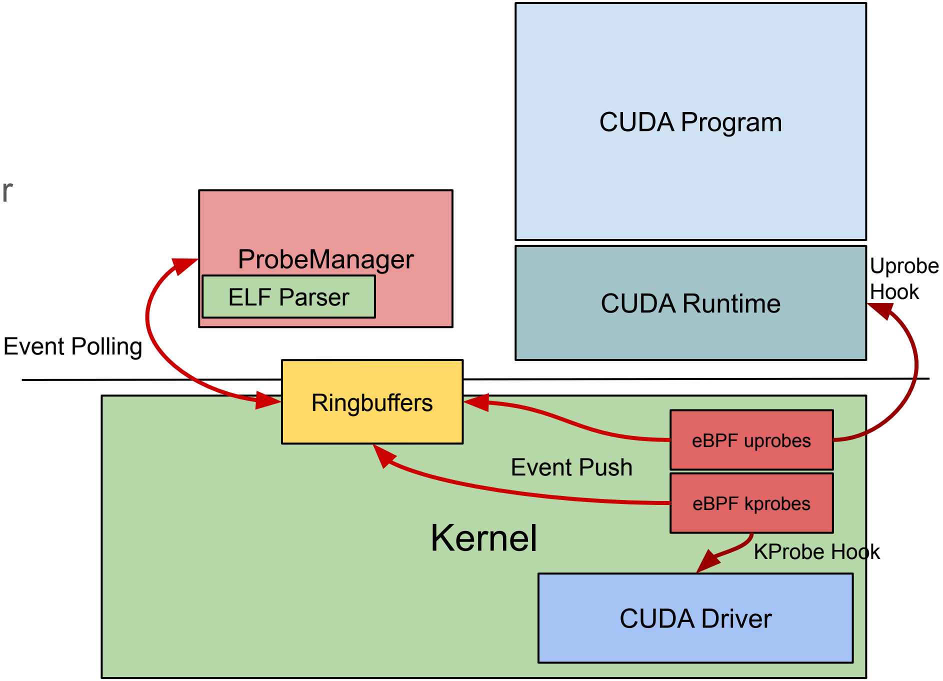
Aside: eBPF for Transparent Profiling

- Recent Linux profiling/control feature (soon Windows)
- Small code fragments dynamically loaded into kernel
  - C (or other) code compiled into bytecode
- Kernel dynamically verifies bytecode safety, attaches to hook point



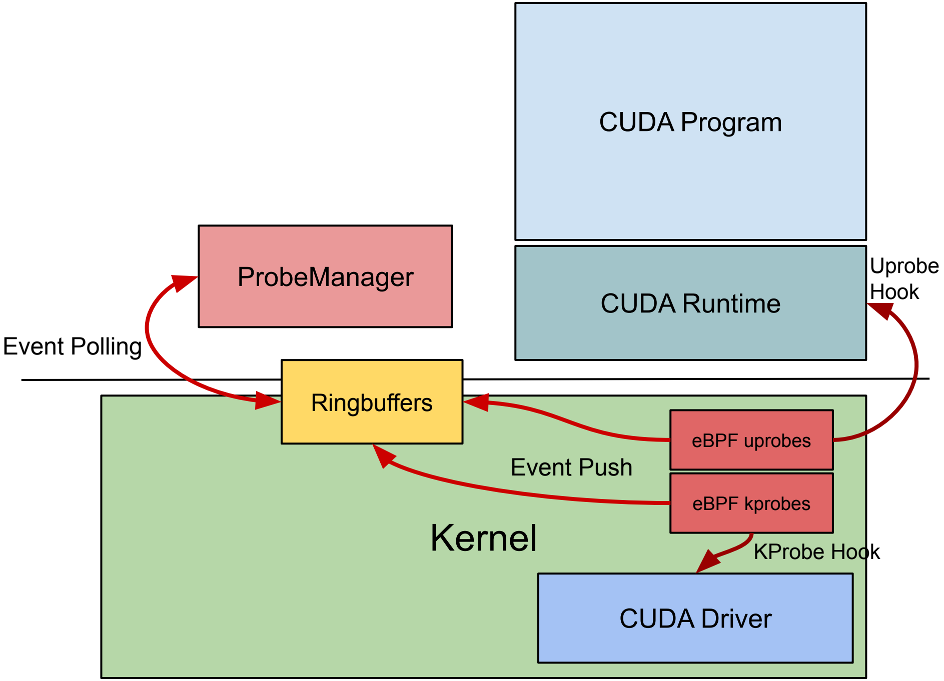
Uprobes for Probing CUDA

- Can hook runtime API functions
  - Ex. cudaMemcpy
- Simple, but has downsides
  - Limited access ‘deeper’ into CUDA stack or kernel (Ex. unified memory)
  - Some performance overhead–each probe requires INT 3/trap
  - Must parse symbol from userspace binary
- Multiple alternatives
  - First-party profilers
  - LD_PRELOAD
  - Function wrappers



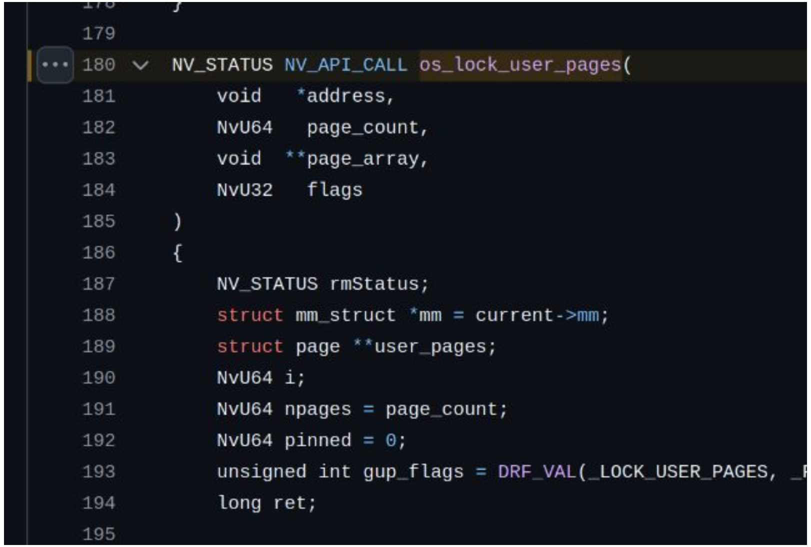
Kprobes for Probing CUDA

- Can hook kernel function
  - Much ‘deeper’–can monitor library-driver and even kernel-device interactions
- Minimal overhead (no trap)
  - Bytecode is JIT-compiled to run on an in-kernel virtual machine
- Plenty of kernel context
  - Memory management details
- Ex: can hook “os_lock_user_pages” driver function
  - Snoop arguments, determine when/how many pages are pinned



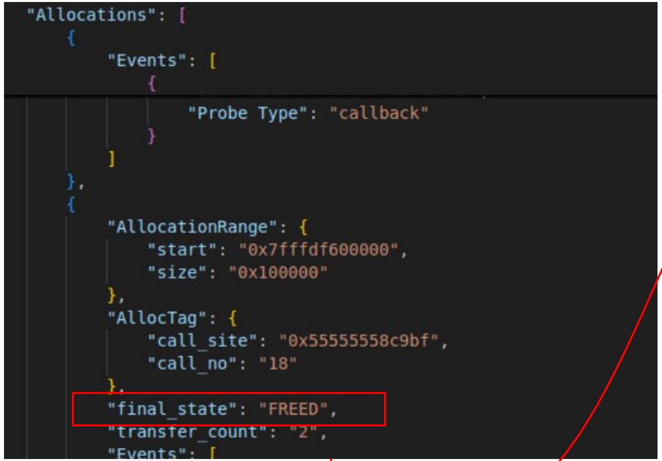
Kprobes for Probing CUDA

- Can hook kernel function
  - Much ‘deeper’–can monitor library-driver and even kernel-device interactions
- Minimal overhead (no trap)
  - Bytecode is JIT-compiled to run on an in-kernel virtual machine
- Plenty of kernel context
  - Memory management details
- Ex: can hook “os_lock_user_pages” driver function
  - Snoop arguments, determine when/how many pages are pinned

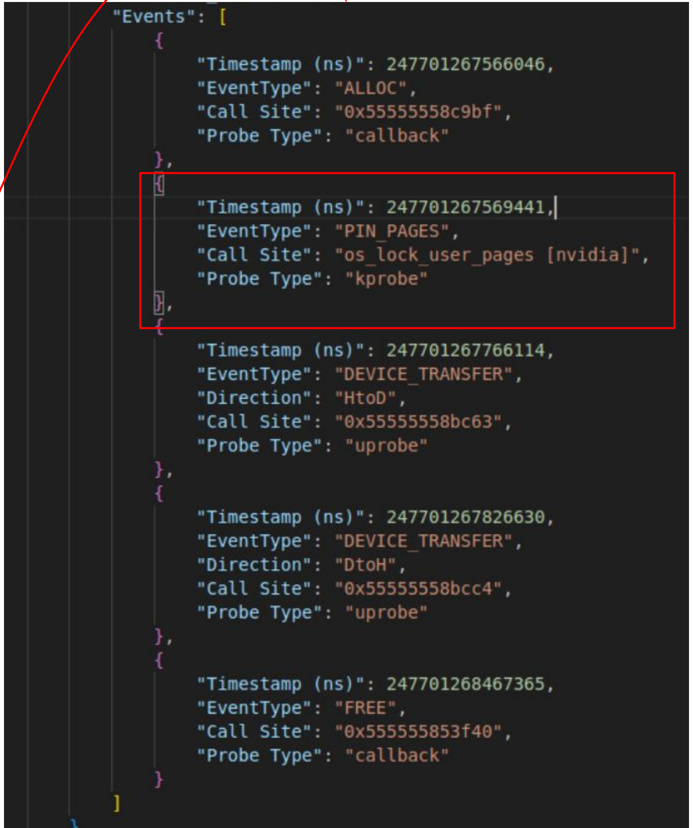


```
}

NV_STATUS NV_API_CALL os_lock_user_pages(
    void    *address,
    NvU64    page_count,
    void   **page_array,
    NvU32    flags
)
{
    NV_STATUS rmStatus;
    struct mm_struct *mm = current->mm;
    struct page **user_pages;
    NvU64 i;
    NvU64 npages = page_count;
    NvU64 pinned = 0;
    unsigned int gup_flags = DRF_VAL(_LOCK_USER_PAGES, _
    long ret;

```

Sample History: Simple CudaMemcpy



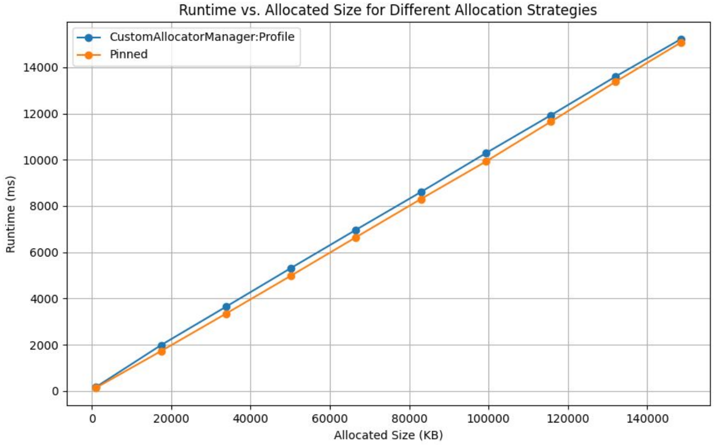
```
"Allocations": [
    {
        "Events": [
            {
                "Probe Type": "callback"
            }
        ]
    },
    {
        "AllocationRange": {
            "start": "0x7fffdf600000",
            "size": "0x100000"
        },
        "AllocTag": {
            "call_site": "0x55555558c9bf",
            "call_no": "18"
        },
        "final_state": "FREED",
        "transfer_count": "2",
        "Events": [
```

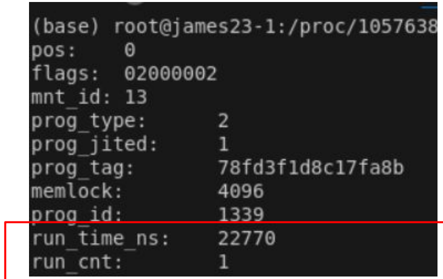


```
"Events": [
    {
        "Timestamp (ns)": 247701267566046,
        "EventType": "ALLOC",
        "Call Site": "0x55555558c9bf",
        "Probe Type": "callback"
    },
    {
        "Timestamp (ns)": 247701267569441,
        "EventType": "PIN_PAGES",
        "Call Site": "os_lock_user_pages [nvidia]",
        "Probe Type": "kprobe"
    },
    {
        "Timestamp (ns)": 247701267766114,
        "EventType": "DEVICE_TRANSFER",
        "Direction": "HtoD",
        "Call Site": "0x55555558bc63",
        "Probe Type": "uprobe"
    },
    {
        "Timestamp (ns)": 247701267826630,
        "EventType": "DEVICE_TRANSFER",
        "Direction": "DtoH",
        "Call Site": "0x55555558bcc4",
        "Probe Type": "uprobe"
    },
    {
        "Timestamp (ns)": 247701268467365,
        "EventType": "FREE",
        "Call Site": "0x555555853f40",
        "Probe Type": "callback"
    }
]
```

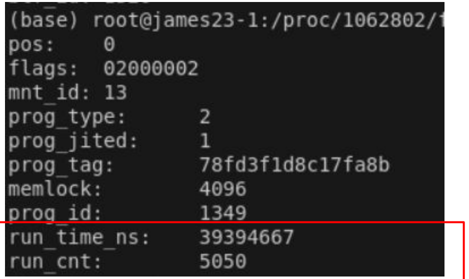
Benchmarking Preview: Tracing Overhead

- Even naive implementation does not slow high-transfer application
- Reasonable memory footprint, with room for optimization
  - Per-allocation memory: ~48 bytes
  - Per-event memory: ~40 bytes
- Per-probe 16 MB ringbuffer overhead
  - Could unify ringbuffers as we add probes
- eBPF probe overhead is negligible
  - First uprobe runtime: ~22 microseconds
  - Avg. uprobe runtime: ~7 microseconds
  - Kprobe runtime: < 1 microsecond





```
(base) root@james23-1:/proc/1057638
pos:    0
flags:  02000002
mnt_id: 13
prog_type:      2
prog_jited:     1
prog_tag:       78fd3f1d8c17fa8b
memlock:        4096
prog_id:        1339
run_time_ns:    22770
run_cnt:        1
```



```
(base) root@james23-1:/proc/1062802/
pos:    0
flags:  02000002
mnt_id: 13
prog_type:      2
prog_jited:     1
prog_tag:       78fd3f1d8c17fa8b
memlock:        4096
prog_id:        1349
run_time_ns:    39394667
run_cnt:        5050
```

Potential Applications

- Continuous monitoring
    - Existing (private) projects by Cruise AV, Meta
- OS

- Chose one to implement: pinning allocator

Usecase: Smart Pinning Allocator

Overview

- Built an CUDA host-side allocator **advised** by tracing framework

- Uses **past behavior** to guide pinning behavior

- Improve GPU I/O efficiency and workload performance

Background: Pinned Memory

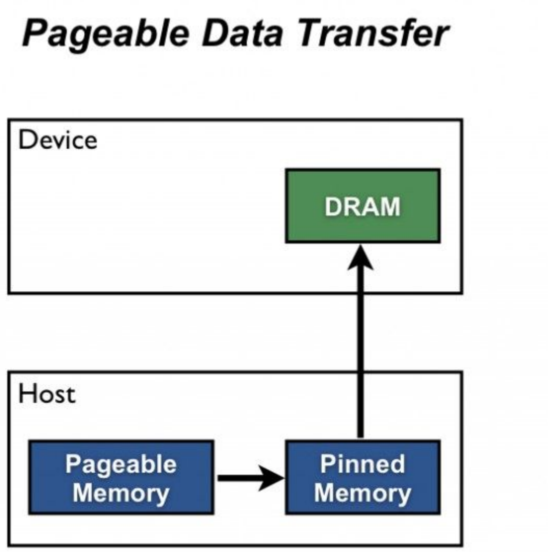
Pageable Data Transfer

Pageable Memory

- **Flexibility**: Supports large & dynamic memory allocations beyond physical limits of RAM due to virtual memory
- **Resource Utilization**: Allows the operating system to optimize physical memory usage by paging out inactive data to disk, freeing up RAM for active processes.

Cons

- Can be swapped in and out by the OS, leading to potential page faults
- Slower data transfer rates due to overhead of pinning and unpinning

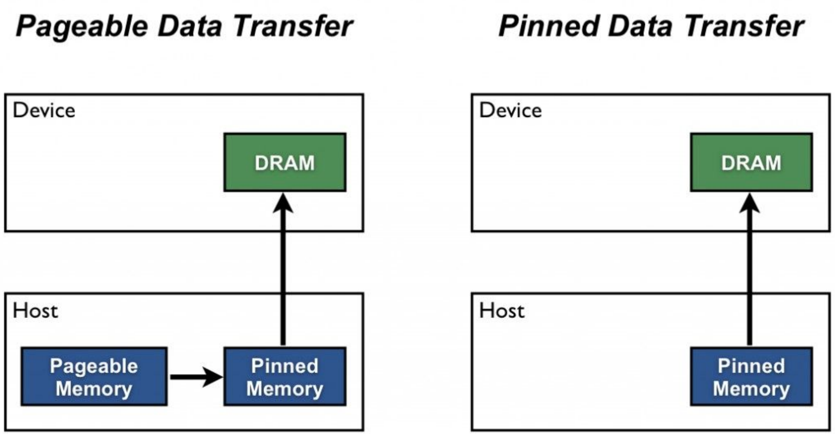


Solution: Pinning Memory

- **Reduced Latency** - eliminates the need for double-copying to device memory

- **Increased Throughput** - enables faster DMA transfers, with less CPU overhead

- **Predictable Performance** - More consistent and predictable memory transfer performance

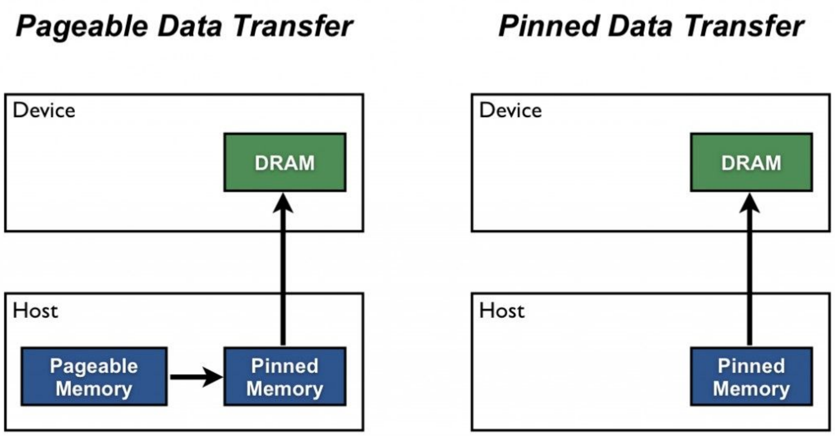


New Problem: Pinning Memory Limits

- **Allocation Limits** - OS imposes hard limits, which can cause abrupt/difficult to debug CUDA crashes

- **Memory pressure -** allocating large amounts of pinned memory reduces available memory for core kernel tasks, can degrade performance

- **Longer allocation times** - higher overhead compared to virtual, pageable memory



Goal: Only Pin What Matters

- Implement an allocator that performs ‘smart pinning’

- Ideally, would pin as many **high-transfer ‘hotspot’** allocations as possible

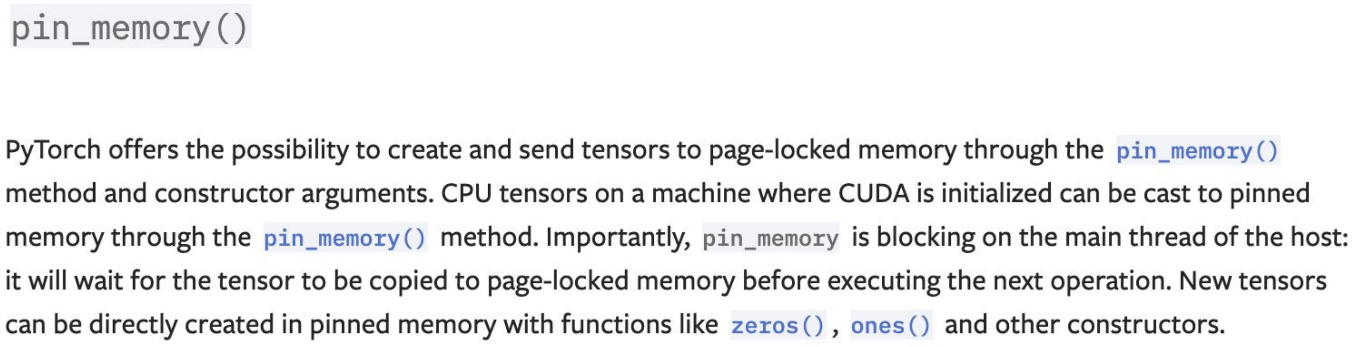
- Would leave **rarely-transferred ‘coldspots’** in pageable memory

- **Key Challenge:** How can we predict which allocations will be ‘hot’ and which will be ‘cold’?

Background: Memory Allocation

PyTorch’s Host Allocator

Let’s take a look at how PyTorch handles host-side memory allocations:



`pin_memory()`

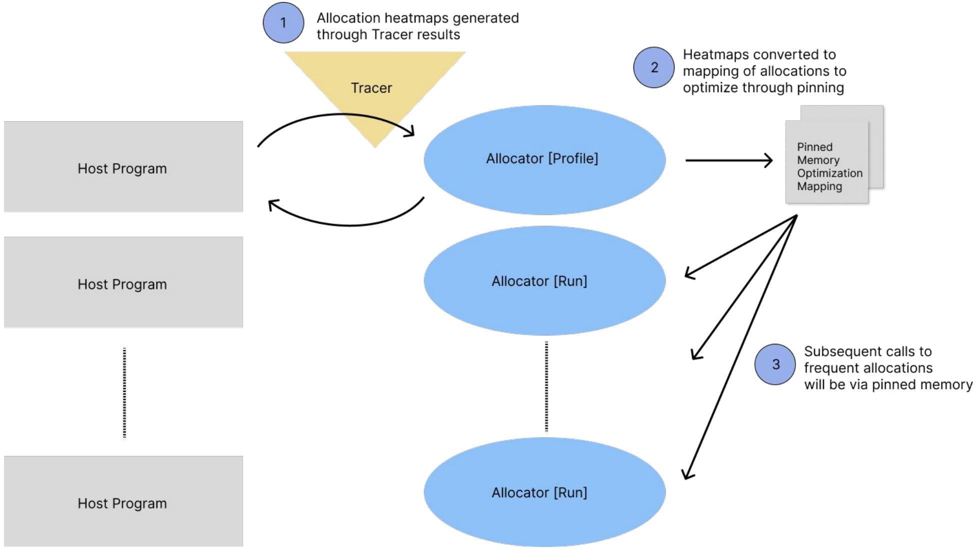
PyTorch offers the possibility to create and send tensors to page-locked memory through the `pin_memory()` method and constructor arguments. CPU tensors on a machine where CUDA is initialized can be cast to pinned memory through the `pin_memory()` method. Importantly, `pin_memory` is blocking on the main thread of the host: it will wait for the tensor to be copied to page-locked memory before executing the next operation. New tensors can be directly created in pinned memory with functions like `zeros()`, `ones()` and other constructors.

The user is given the flexibility to pin/unpin tensors.

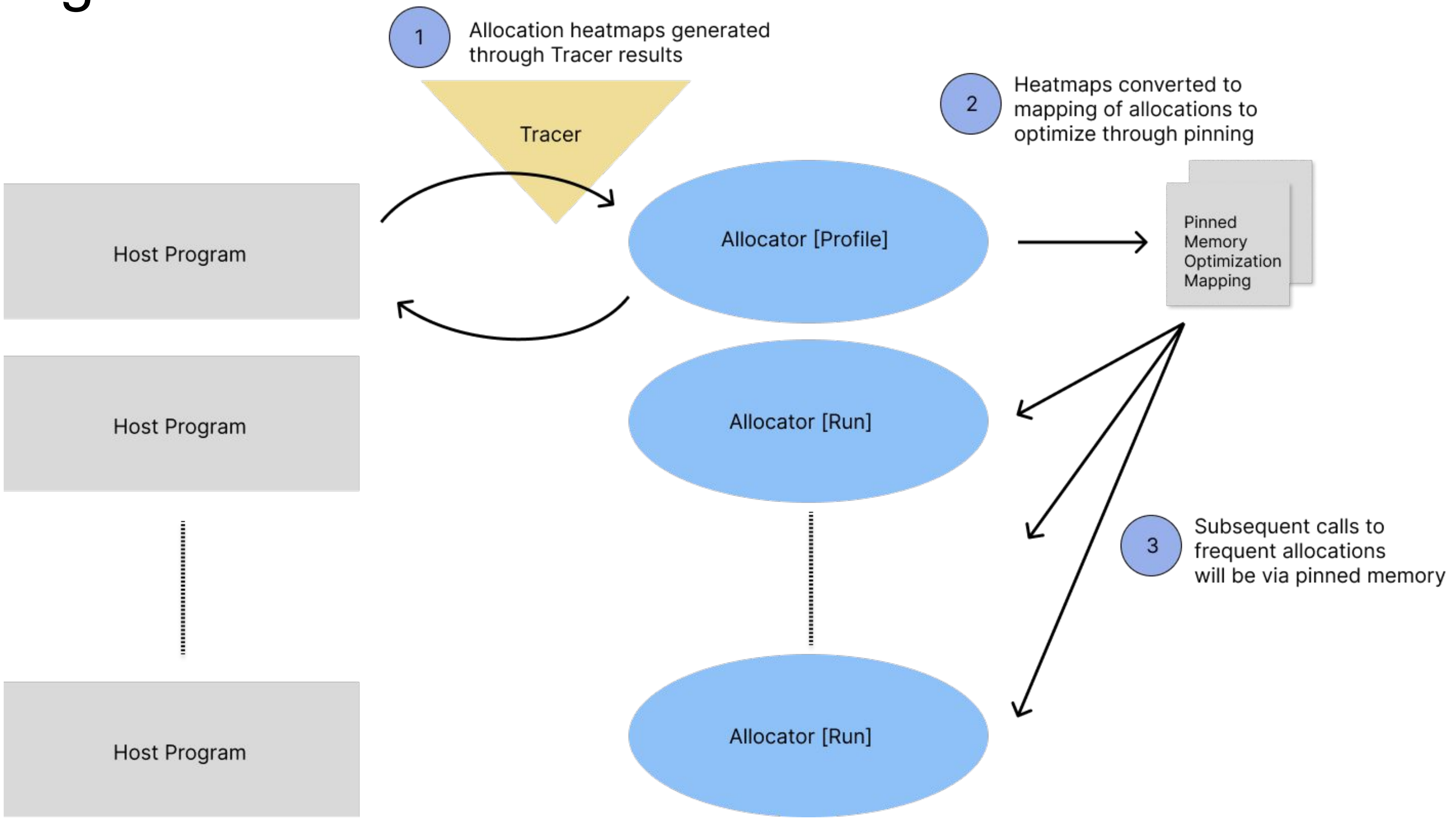
Design

Design Overview

- **Lightweight tracer** observes/records device memory transfers
- **Custom allocator** queries tracer to make allocation decisions
- When the program repeats, utilize history to guess which allocations to pin
    - i.e. after a full run, or after some predictable event like a batch


1
Allocation heatmaps generated through Tracer results
Tracer
2
Heatmaps converted to mapping of allocations to optimize through pinning
Host Program
Allocator [Profile]
Pinned Memory Optimization Mapping
Host Program
Allocator [Run]
3
Subsequent calls to frequent allocations will be via pinned memory
Host Program
Allocator [Run]

The Program



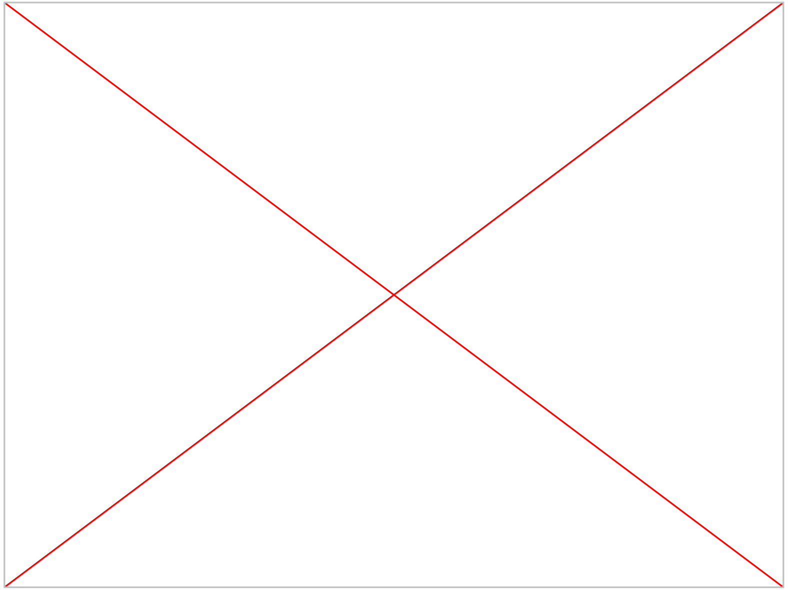
1 Allocation heatmaps generated through Tracer results

2 Heatmaps converted to mapping of allocations to optimize through pinning

3 Subsequent calls to frequent allocations will be via pinned memory

Demo

- Example on comparing performance of pinned, non-pinned, and our custom allocator



Implementation: Allocator

Allocator

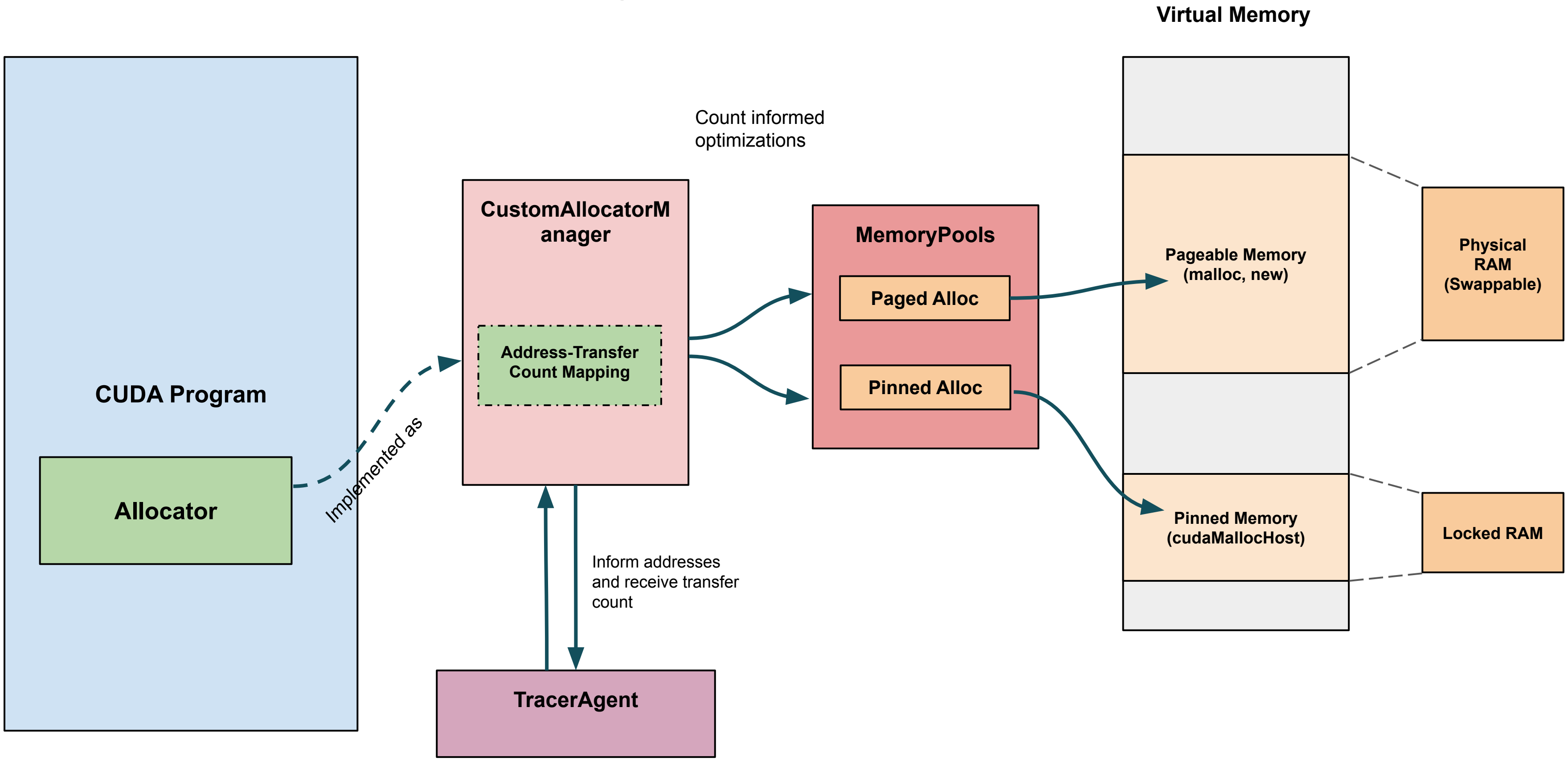
Design

- **Tracer-informed decision making** for pinned/page memory allocation

Components:

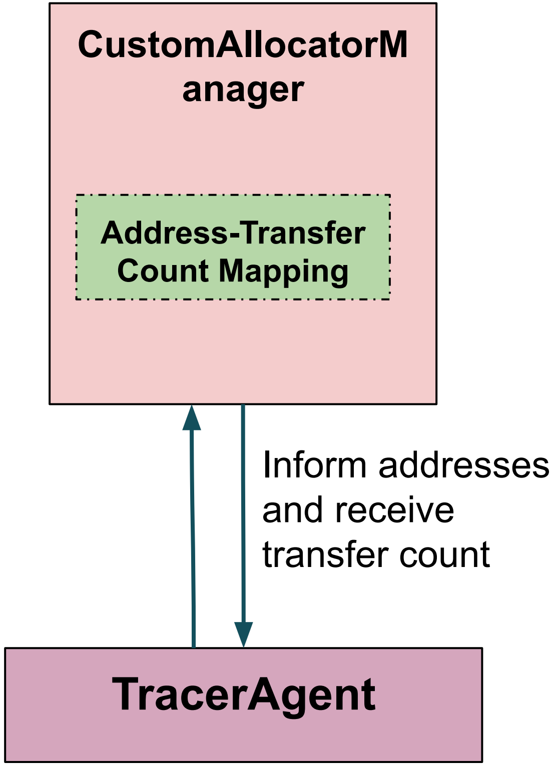
- CustomAllocatorManager (Modes: [Profile], [Use])
- MemoryPools: handles malloc and cudaMallocHost

Allocator Component Diagram


Virtual Memory
Count informed optimizations
CUDA Program
Allocator
Implemented as
CustomAllocatorManager
Address-Transfer Count Mapping
MemoryPools
Paged Alloc
Pinned Alloc
Pageable Memory (malloc, new)
Physical RAM (Swappable)
Pinned Memory (cudaMallocHost)
Locked RAM
Inform addresses and receive transfer count
TracerAgent

Component: CustomAllocatorManager

- 2 Modes: [Profile] (1st run) and [Use] (subsequent optimizations)
- Has a Tracer Pointer internally
- “Profile” mode pins everything, and:
  - passes allocation information to Tracer
  - outputs Tracer Info about device transfers as JSON in destructor
- “Use” mode pins smartly based on the device transfer information of previous run, given by Tracer


CustomAllocatorManager
Address-Transfer Count Mapping
Inform addresses and receive transfer count
TracerAgent

Evaluation: Benchmarking

Measurements

We set out to measure the performance of our allocator by doing the following benchmarks

- CustomAllocatorManager (Smart Pinning) v. Simple Malloc (No Pinning)
- CustomAllocatorManager (Smart Pinning) v. CudaMalloc (Full Pinning)
- CustomAllocatorManager uses CudaMalloc when deemed necessary and Malloc otherwise.

Experiment

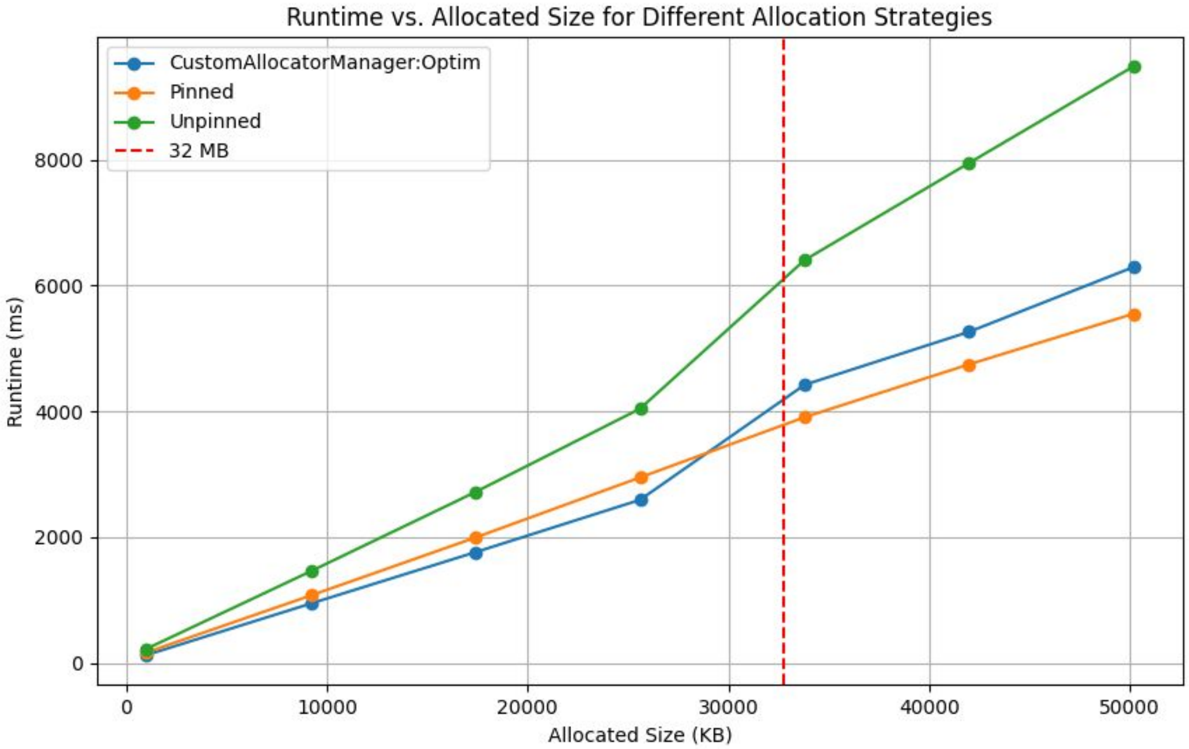
In the following measurement, we have two types of calls:

- frequently transferred between host and device (using cudaMemcpy)
- infrequently transferred between host and device (using cudaMemcpy)
- CustomAllocatorManager
  - pins all memory in “Profile” mode and saves Tracer information
  - only pins the frequent memories in “Use” mode
- compare runtime of allocation + transfer + deallocation

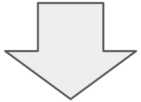
CustomAllocatorManager performs the best until allocated size is very large

full pinning performs worse for very small size due to overhead. it performs the best for very large size

Malloc might be performing worse after size is larger than 32MB.

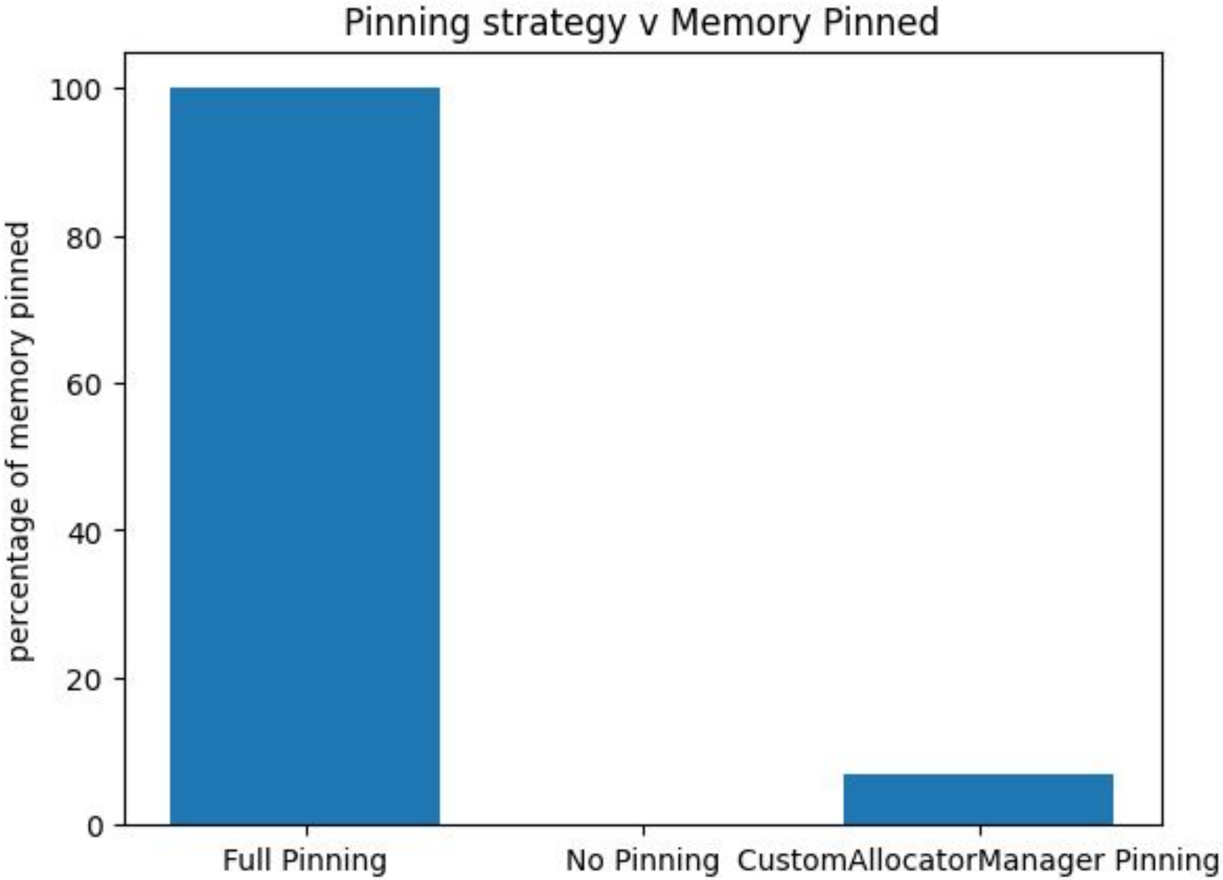


Smart Pinning only pins a small percentage of memory

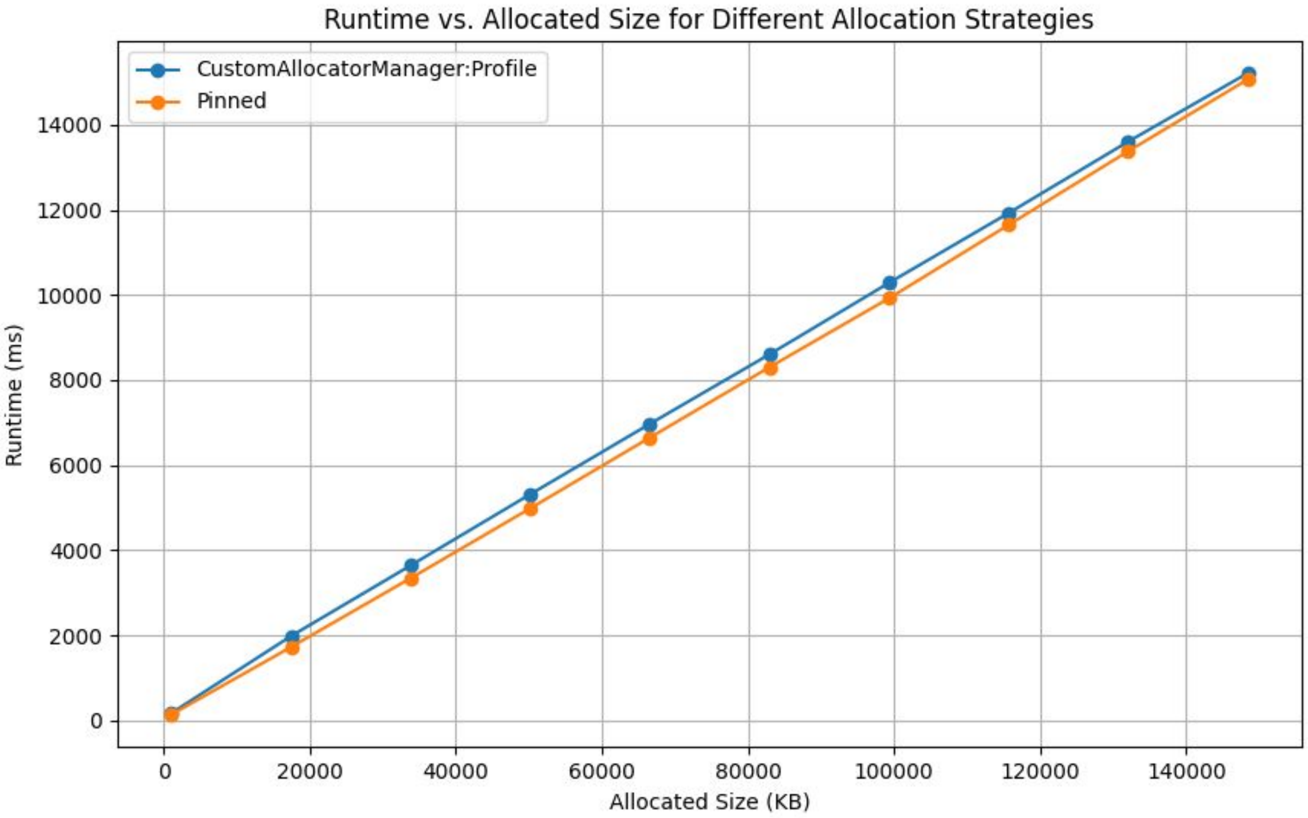


Performs much better than no pinning

Performs better / slightly worse than full pinning



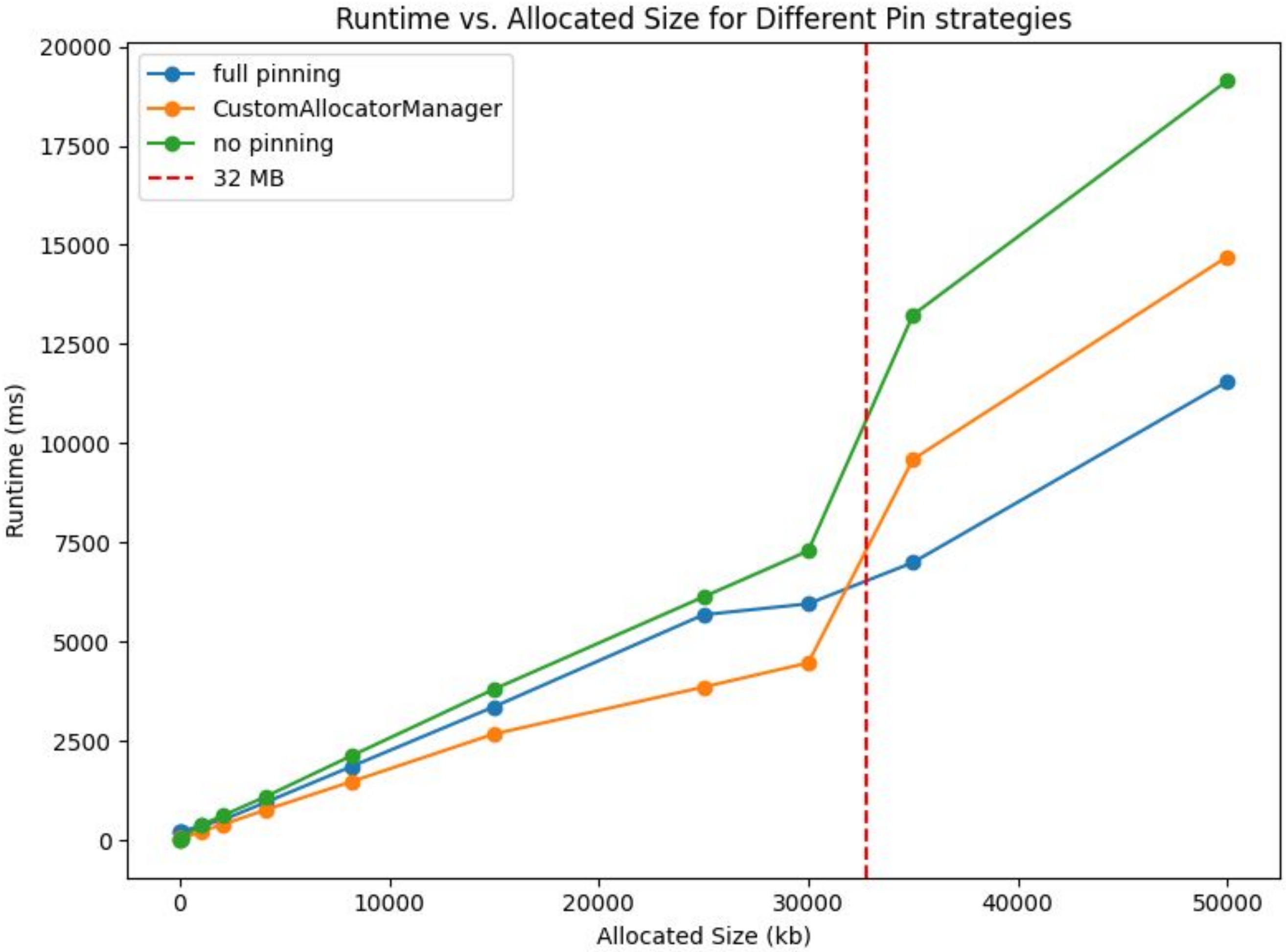
Very little overhead from the Profile mode, which uses the Tracer



CustomAllocatorManager performs the best until allocated size is very large

full pinning performs the worst for very small size due to overhead. it performs the best for very large size

Malloc might be performing worse after size is larger than 32MB.



Challenges

Tracer Challenges

- Difficult to manage complex toolchain
  - Building eBPF probes and eBPF based application
- Complexity of GPU driver, and CUDA functionality
  - Focus on limited set of scenarios for allocation, transfer, deallocation
- eBPF probes can be cumbersome to develop
  - Hard to debug, and easy to ‘anger’ the verifier
  - Design brings as much processing out of probes as possible

Allocator Challenges

- Different runs may produce slightly different control flow
  - Must decide what it means for an allocation to be the same
- Used a combination of call site + iteration count as unique allocation identifier
  - Complicated by ASLR and PIE code
- Allocator must be robust to misaligned advice

Future Work

Expand Benchmarking

- Test on actual deep learning tasks such as
  - Large-scale matrix multiplication or convolution
  - Deep learning training (backpropagation) and inference (forward propagation)
- Measure tracing overhead in various scenarios
  - Find worst-case scenarios for tracing, and mitigate
- Further explore behavior of unmodified samples
  - Particularly, those that use newer CUDA memory features like unified memory

Performance Improvement

- Based on **profiling overhead**, optimize tracer and allocator implementation

- Improve **allocator algorithm** and **advising API**, to utilize more history
  - Build a ‘pattern history table’ to recognize variable-length patterns (like branch predictors)

- Utilize eBPF kernel access to gain more **memory management context**

More, and More Complex, Probes

- Current probes quite simple–snoop function call arguments
- Further probes could observe/optimize other kernel components
  - Observe (or inform) page cache to improve data load performance
  - Shortcut network stack for distributed training by intercepting data at TC/XDP networking hooks

The Goal

Given the shortcomings of existing allocators in addressing the unique demands of ML training, there exists a substantial opportunity to **design a memory allocator that is finely tuned to the specific allocation and deallocation patterns** of deep learning workloads. Our goal is to create a **CUDA host memory allocator** that:

1. **Optimizes for Large Contiguous Allocations:**
   a. ML training often involves allocating large blocks of memory for model parameters, gradients, and activations. Efficiently handling these large, contiguous memory regions can significantly reduce allocation overheads.
2. **Enhances Data Transfer Efficiency:**
   a. By leveraging **pinned (page-locked) memory**, our allocator can facilitate faster and more efficient host-device data transfers. Pinned memory allows for direct memory access (DMA) transfers, minimizing latency and maximizing throughput.
3. **Implements Advanced Memory Pooling:**
   a. Incorporating sophisticated memory pooling strategies to **reuse memory blocks** effectively aligns with the repetitive allocation patterns observed in ML training, thereby reducing fragmentation and allocation times.

Overview

- Motivation
    - Problem tackled : Crude implementation of allocator in ml frameworks
    - Pinned Memory as an option for speedup
- Background + the How/Implementation
    - eBPF
    - Pinned memory vs paged memory
- Overview of Program
    -
- Measurements and Benchmarks
    - Comparison between pure malloc w/ our custom allocator
        - What parameters do we want to test for (Dig into understanding how ML data transfers look like)
    - Comparison under workloads similar to real life ML training data transfers
    -
- Tutorial
    - Demonstration of code in action
    - How to use
- Design (if we have time)
    - Modern CPP used?

The How

On a high level, our allocator utilizes the following processes to leverage pinned memory transfers.

1. The allocator monitors allocation activity of the program and determines the most frequent patterns that could potentially benefit from switching to pinned
    a. Monitoring is achieved through **eBPF** probing
    b. A frequency heatmap from probing is used to determine following rounds of program calls allocation methods
    c. This utilizes the hardware configuration of GPUs to achieve faster transfer