
OOM Flight Recorder Ring-Buffer: Deep Dive

Why a ring-buffer flight recorder for GPU OOM fills a gap PyTorch snapshots and NCCL tracing do not — temporal context, automatic dumps, and structured artifacts.

AUTHOR

Prince Agyei Tuffour

PUBLISHED

February 14, 2026

READING TIME

11 min

IN THIS PIECE

- 01 OOM Flight Recorder Ring-Buffer: Deep Dive
- 02 What Is a Flight Recorder Ring-Buffer?
- 03 How Existing Tools Handle OOM Diagnostics
- 04 What Makes Your Implementation Unique
- 05 The "Black Box" Analogy in Depth
- 06 Implementation Considerations
- 07 Why This Matters for Your Roadmap

01

OOM Flight Recorder Ring-Buffer: Deep Dive

02

What Is a Flight Recorder Ring-Buffer?

The concept borrows directly from aviation's Flight Data Recorder (FDR) — the "black box" that continuously records the last N seconds of telemetry so that when a crash occurs, investigators can reconstruct the events leading up to it. In software, the pattern is identical:[\(1\)](#)[\(2\)](#)

- A **ring buffer** (also called a circular buffer) is a fixed-size, FIFO data structure where new data overwrites the oldest data once the buffer is full. It uses two pointers — a write head and a read tail — that wrap around the buffer's capacity, so there's never a need to shift elements or dynamically allocate memory.⁽³⁾⁽⁴⁾
- A **flight recorder** combines this ring buffer with **trigger-based dumping**: the buffer runs continuously with near-zero overhead, and only when a failure event (in your case, OOM) is detected does the system freeze and dump the buffer contents to disk.⁽⁵⁾

The result is that you always have exactly the last N seconds/events of diagnostic context available at the moment of failure, without the cost of writing continuously to disk or the risk of logging growing unboundedly in long-running training jobs.

Why a Ring Buffer Specifically?

- **Bounded memory**: The buffer has a fixed, configurable size. Under a 72-hour training run, it doesn't grow — it always occupies exactly the configured amount.⁽⁴⁾
- **O(1) insert**: Writing a new event is a pointer increment and a memory write. No allocation, no GC pressure, no syscalls.⁽⁶⁾⁽³⁾
- **Overwrite semantics**: When full, the oldest events are silently discarded. This is exactly the desired behavior — you only care about the recent history leading up to the crash, not events from 6 hours ago.⁽⁴⁾
- **No contention on the hot path**: A well-designed ring buffer can be made lock-free for single-producer scenarios, which is typical for per-GPU telemetry streams.⁽⁶⁾

03

How Existing Tools Handle OOM Diagnostics

To understand why your approach is unique, let's map the current landscape:

PyTorch Memory Snapshot (`torch.cuda.memory`)

PyTorch provides a built-in mechanism to capture memory snapshots on OOM:⁽⁷⁾⁽⁸⁾

PYTHON

```
def oom_observer(device, alloc, device_alloc, device_free):  
    snapshot = torch.cuda.memory._snapshot()  
    dump(snapshot, open('oom_snapshot.pickle', 'wb'))  
  
torch._C._cuda_attach_out_of_memory_observer(oom_observer)
```

Limitations:

- This captures a **point-in-time snapshot** of the allocation state at the moment of OOM, not a temporal history of events leading up to it.⁽⁹⁾
- The `_record_memory_history()` API does maintain a bounded event log (`max_entries`), but it must be **explicitly enabled before the run**, and the output is a monolithic pickle file that requires dragging into PyTorch's `memory_viz` web tool.⁽⁸⁾⁽¹⁰⁾
- There's no automatic structured artifact bundle — you get a raw pickle, not a timeline + metadata + environment context.
- The observer fires on *every* OOM attempt, including caught-and-retried OOMs inside the caching allocator, making it noisy without custom filtering logic.⁽⁹⁾

PyTorch Flight Recorder (for Distributed/NCCL)

PyTorch has a separate "Flight Recorder" for debugging **stuck distributed jobs**, not OOM. It records NCCL collective operations into a circular buffer (`TORCH_NCCL_TRACE_BUFFER_SIZE`) and dumps on timeout.⁽¹¹⁾⁽¹²⁾

- It is specifically for collective communication debugging (allreduce, broadcast, etc.).⁽¹¹⁾
- It does not track memory allocation events, tensor lifecycle, or GPU memory pressure.
- The dump trigger is job **timeout**, not OOM.⁽¹³⁾
- Output is NCCL-specific trace data (process group config, collective sequence IDs, stack frames).⁽¹¹⁾

This is the closest analog in the PyTorch ecosystem to what you're building, but it targets a completely different failure mode.

pytorch_memlab

A third-party library that provides line-by-line CUDA memory profiling and a memory reporter. It's useful for interactive debugging but:⁽¹⁴⁾

- No flight recorder / ring buffer pattern.
- No automatic OOM detection or dump.
- Designed for development-time profiling, not production or long-run training monitoring.

NVIDIA Nsight / Nsight Compute

These are low-level kernel profilers. Nsight Compute profiles individual CUDA kernels (warp stalls, memory bandwidth, occupancy). They don't provide memory allocation tracking over time, don't have OOM detection, and require manual invocation with significant overhead.⁽¹⁵⁾

GPUprobe (eBPF-based)

GPUprobe hooks into `cudaMalloc/cudaFree` via eBPF with <4% overhead for continuous monitoring. However:⁽¹⁶⁾⁽¹⁷⁾

- It provides real-time metrics, not a pre-failure context dump.
- No ring buffer with automatic OOM-triggered artifact generation.
- Operates at the CUDA runtime API level, missing higher-level context (which layer, which training step, what the optimizer was doing).

OOMProf (parca-dev)

An eBPF-based tool that captures heap profiles from Go programs just before OOM kill. Interesting parallel to what you're doing, but:⁽¹⁸⁾

- Go-specific (uses Go's memory bucket addresses).
- Captures OS-level OOM killer events, not CUDA OOM.
- No ring buffer of historical telemetry — it captures a single heap snapshot at the moment of the kill signal.

TraceML

A lightweight wrapper that provides real-time GPU memory statistics during training. Focused on live monitoring and suggestions, not post-mortem artifact generation.⁽¹⁹⁾

04

What Makes Your Implementation Unique

Given the landscape above, here's why your OOM flight recorder is genuinely differentiated:

Temporal Context, Not Just a Snapshot

The critical insight is that an OOM doesn't happen in isolation — it's the result of a **sequence of events** (memory allocations, cache fragmentation, gradient accumulation, activation storage) over the preceding seconds or minutes. PyTorch's OOM snapshot tells you "here's what's allocated right now". Your ring buffer tells you "here's the **timeline** of what happened in the N seconds before the crash." This is the difference between a photograph and a security camera replay.⁽¹⁰⁾

Automatic Trigger Without Manual Setup

Existing tools require one of:

- Pre-emptive `_record_memory_history()` calls before training starts⁽⁸⁾
- Environment variable configuration for NCCL flight recorder⁽¹¹⁾
- Manual attachment of profilers

Your design detects OOM at runtime (via `RuntimeError` pattern matching and framework-specific signals) and dumps automatically. The user doesn't need to anticipate the failure — the recorder is always on, and the dump is a zero-intervention response to failure.

Structured Artifact Bundle

No existing tool produces a **structured, self-contained artifact bundle** on OOM. PyTorch gives you a pickle file. NCCL flight recorder gives you per-rank trace files. Your deliverable is:⁽⁸⁾⁽¹¹⁾

- A **timeline** of memory events leading up to OOM
- **Metadata** (model config, batch size, training step, framework version)

- **Environment snapshot** (GPU model, driver version, CUDA version, available memory)
- **Deterministic filename conventions** for automated collection by CI/CD or experiment tracking systems

This is a first-class diagnostic artifact, not a raw dump that requires manual interpretation tooling.

Bounded Overhead for Long Runs

The ring buffer with configurable retention/size controls means your profiler can run for days without memory growth or disk I/O. This is critical for the production training use case. PyTorch's `_record_memory_history(max_entries=100000)` provides some bounding, but it's a separate mechanism from the OOM observer and doesn't produce a unified artifact.⁽⁴⁾⁽⁸⁾

Framework-Agnostic OOM Detection

Your design detects OOM through both `RuntimeError` pattern matching (which works for PyTorch's `torch.cuda.OutOfMemoryError`) and "framework-specific signals," suggesting extensibility to TensorFlow, JAX, or raw CUDA applications. No existing tool spans this boundary.

05

The "Black Box" Analogy in Depth

The aviation flight recorder analogy is worth taking seriously because the engineering constraints are remarkably similar:⁽²⁰⁾

Aviation FDR	Your OOM Flight Recorder
Records last 25 hours of flight parameters	Records last N seconds/samples of GPU telemetry
Fixed-size storage, continuous overwrite	Ring buffer with bounded memory footprint
Crash-survivable (withstands impact, fire, water)	Persists to disk before process termination
Standardized format for investigators	Structured artifact bundle with deterministic naming
Always on, zero pilot intervention	Always on, auto-dump on OOM detection
Enables root-cause analysis post-crash	Enables root-cause analysis of GPU OOM

DTrace in Solaris explicitly uses this analogy — its ring buffer policy is described as "an operating

system analog to the black box flight data recorder present on commercial aircraft". Go 1.25's new Flight Recorder feature follows the same design: a continuous circular buffer of runtime events that can be dumped on demand or on crash, with extremely low overhead.⁽²¹⁾⁽⁵⁾

06

Implementation Considerations

OOM Detection Patterns

For PyTorch, you'll want to handle multiple signals:

1. `torch.cuda.OutOfMemoryError` — the standard Python exception
2. `RuntimeError` with "CUDA out of memory" in the message — for older PyTorch versions
3. `torch._C._cuda_attach_out_of_memory_observer` — the C++-level callback that fires *before* the stack unwinds. This is critical because by the time you catch the Python exception, the caching allocator may have already freed retry memory, distorting the picture.⁽⁹⁾

The observer approach is preferred because it captures the state at the *exact moment* of allocation failure, not after cleanup.⁽⁹⁾

Ring Buffer Design Choices

- **Event granularity:** You need to decide between recording every `cudaMalloc/cudaFree` (high volume, complete picture) vs. sampled/aggregated snapshots (lower overhead, sufficient for most diagnosis). Consider a hybrid: always record allocation/free events but sample stack traces at a configurable rate.
- **Thread safety:** If you're recording from multiple CUDA streams or a DataLoader worker pool, you need either a lock-free SPMC ring buffer or per-stream buffers with merge-on-dump.
- **Timestamp resolution:** Use `torch.cuda.Event` timestamps for GPU-side timing and `time.monotonic_ns()` for CPU-side, so the timeline can correlate GPU and CPU events.

Artifact Bundle Format

Consider a directory or tar bundle with:

TEXT

```
oom_dump_20260214_115530/
├── timeline.jsonl      # Ring buffer events in chronological order
├── metadata.json      # Training step, model config, batch size
├── environment.json   # GPU info, driver, CUDA version, memory capacity
├── snapshot.pickle    # Optional: PyTorch memory snapshot at OOM time
└── README.md         # Human-readable summary
```

Using JSONL for the timeline enables streaming analysis and `grep`-based debugging without specialized tooling.

07

Why This Matters for Your Roadmap

Your core product promise is **fast root-cause diagnosis after failure**. Every existing tool in the ecosystem either:

- Requires manual setup before the failure occurs (PyTorch memory snapshot)
- Targets a different failure mode (PyTorch NCCL flight recorder -> stuck jobs, not OOM)
- Provides real-time monitoring but no post-mortem artifacts (TraceML, GPUprobe, DCGM)
- Operates at the wrong abstraction level (Nsight -> kernel performance, not memory lifecycle)

Your OOM flight recorder fills a genuine gap: **zero-configuration, always-on, automatic pre-failure context capture with structured artifact output**. This is the feature that turns your profiler from a "run it when you have a problem" tool into an "always-on safety net" — which is a fundamentally different value proposition and a strong differentiator in the GPU tooling space.

• • •

NOTES

- (1) [In plane view](#)^[1] — An airplane's digital flight-data recorder, or “black box,” holds massive amounts of data, documenti...
- (2) [How black boxes became key to solving airplane crashes](#)^[2] — Flight data recorders and cockpit voice recorders, called “black boxes” are nearly indestructible.
- (3) [Ring Buffer Basics - Embedded](#)^[3] — The ring buffer's first-in first-out data structure is useful tool for transmitting data between asy...
- (4) [Circular buffer](#)^[4] — A circular buffer, circular queue, cyclic buffer or ring buffer is a data structure that uses a sing...

- (5) [Oracle® Solaris 11.3 DTrace \(Dynamic Tracing\) Guide](#)^[5] — This chapter describes the DTrace facilities for postmortem extraction and processing of the in-kern...
- (6) [Microcontrollers: Interrupt-safe ring buffers](#)^[6] — An interrupt-safe circular buffer. This lesson is dedicated to understanding this very important dat...
- (7) [Visualizing Traces](#)^[7] — Technical guides for various topics related to machine learning and CUDA programming.
- (8) [Understanding CUDA Memory Usage](#)^[8] — To debug CUDA memory use, PyTorch provides a way to generate memory snapshots that record the state ...
- (9) [How to do CUDA traces callback only for uncaught OOM errors? - C++](#)^[9] — From this blog post by @zdevito, we see that we can add a callback every time it OOMs but this callb...
- (10) [Understanding GPU Memory 1: Visualizing All Allocations ...](#)^[10] — The Memory Profiler is an added feature of the PyTorch Profiler that categorizes memory usage over t...
- (11) [\(prototype\) Flight Recorder for Debugging Stuck Jobs](#)^[11]
- (12) [Flight Recorder for Debugging Stuck Jobs](#)^[12]
- (13) [torchTitan/docs/debugging.md at main · pytorch/torchTitan](#)^[13] — A PyTorch native library for large model training. Contribute to pytorch/torchTitan development by c...
- (14) [GitHub - Stonesjtu/pytorch_memlab: Profiling and inspecting memory in pytorch](#)^[14] — Profiling and inspecting memory in pytorch. Contribute to Stonesjtu/pytorch_memlab development by cr...
- (15) [Fix GPU Bottlenecks: PyTorch Profiler + Nsight](#)^[15] — Nsight Compute and PyTorch Profiler are the fastest way to turn a sluggish GPU training run into a m...
- (16) [The Accelerator Toolkit: A Review of Profiling and Tracing for ...](#)^[16]
- (17) [The Accelerator Toolkit: A Review of Profiling and Tracing for GPUs ...](#)^[17] — Unlock the potential of eBPF
- (18) [parca-dev/oomprof: eBPF OOM Memory Profiler](#)^[18] — OOMProf is an eBPF-based process monitor that automatically captures heap profiles from Go programs ...
- (19) [TraceML: A lightweight tool to see GPU memory + efficiency issues in real time during training](#)^[19] — TraceML: A lightweight tool to see GPU memory + efficiency issues in real time during training
- (20) [Investigating Software Failures With a](#)^[20]
- (21) [Go 1.25 Flight Recorder: Unmasking Production Heisenbugs](#)^[21] — Flight Recorder captures a rich set of runtime events, providing granular insights into the Go sched...

• • •

That's the piece. If it earned a margin note, I'd love to hear it.

prince.agyei.tuffour@gmail.com

LINKS, FOR WHEN YOU'RE READING ON PAPER

- [1] <https://news.mit.edu/2011/black-box-analysis-0912>
- [2] <https://www.cnbc.com/2025/11/30/black-box-airplane-crashes.html>
- [3] <https://www.embedded.com/ring-buffer-basics/>
- [4] https://en.wikipedia.org/wiki/Circular_buffer
- [5] https://docs.oracle.com/cd/E53394_01/html/E53395/gkyer.html

- [6] <https://www.downtowndougbrown.com/2013/01/microcontrollers-interrupt-safe-ring-buffers/>
- [7] <https://zdevito.github.io/2022/12/09/memory-traces.html>
- [8] https://docs.pytorch.org/docs/stable/torch_cuda_memory.html
- [9] <https://discuss.pytorch.org/t/how-to-do-cuda-traces-callback-only-for-uncaught-oom-errors/183983>
- [10] <https://pytorch.org/blog/understanding-gpu-memory-1/>
- [11] https://pytorch.org/tutorials/prototype/flight_recorder_tutorial.html
- [12] https://docs.pytorch.org/tutorials/unstable/flight_recorder_tutorial.html
- [13] <https://github.com/pytorch/torchtitian/blob/main/docs/debugging.md>
- [14] https://github.com/Stonesjtu/pytorch_memlab
- [15] <https://acecloud.ai/blog/gpu-bottlenecks-pytorch-profiler-nsight/>
- [16] <https://www.yunwei37.com/blog/gpu-profile-tools-analysis>
- [17] <https://eunomia.dev/blog/2025/04/11/the-accelerator-toolkit-a-review-of-profiling-and-tracing-for-gpus-and-other-co-processor/>
- [18] <https://github.com/parca-dev/oomprof>
- [19] https://www.reddit.com/r/LocalLLaMA/comments/1nuds4m/traceml_a_lightweight_tool_to_see_gpu_memory/
- [20] <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=9f8a74c9ca75ff36106b84016a9db0646b259563>
- [21] <https://allur.co/en/blog/go-125-flight-recorder-unmasking-production-heisenbugs>

Typeset from the live article at princeagyeituffour.com/writing/oom-flight-recorder-deep-dive. The web version may be newer and carries the interactive parts.