
I Helped Build a GPU Memory Profiler. Then I Had to Learn What GPU Memory Actually Is.

Walking through the Stormlog tutorial after shipping the tool — from PyTorch counters to a deliberate leak, OOM evidence, and the fix.

AUTHOR	PUBLISHED	READING TIME
Prince Agyei Tuffour	March 17, 2026	8 min

IN THIS PIECE

- 01 I Helped Build a GPU Memory Profiler. Then I Had to Learn What GPU Memory Actually Is.
 - 02 How This Started
 - 03 What the Walkthrough Is
 - 04 Step 1: What PyTorch Already Tells You
 - 05 Step 2: The Leak
 - 06 Step 3: Turning the Run Into Evidence
 - 07 Step 4: The Fix
 - 08 What I Actually Learned
 - 09 Run It Yourself
-

01

I Helped Build a GPU Memory Profiler. Then I Had to Learn What GPU Memory Actually Is.

There's a specific kind of embarrassment that comes from building something you don't fully understand.

Not the normal kind, where you're figuring things out as you go. The kind where you've written the code, shipped it to PyPI, and then someone asks you a real question about the domain — and you realize your mental model has holes in it you didn't know were there.

That's where I've been with Stormlog.

• • •

02

How This Started

Silas came to me with a problem he'd been running into repeatedly. Training jobs would die mid-run with cryptic out-of-memory errors. The existing tooling was fragmented — `nvidia-smi` tells you the card is full, PyTorch's native counters tell you what's allocated right now, but nothing gives you a continuous picture of what happened across the whole run. Nothing saves the evidence.

He filed a GitHub issue: *Profile a PyTorch function*. I started working on it.

My first attempt was, in his words, a mess. He's brought it up enough times that I've stopped being defensive about it. I was writing code to solve a problem I understood abstractly but hadn't lived with the way he had. The refactoring that followed taught me more about software design than anything I'd done in the previous six months.

After we sat down together — went through the architecture, did market research on existing profilers, figured out what the actual gaps were — things started clicking into place. I ended up owning the TUI, the visualization exports, the diagnostics view, and several of the CLI commands. Silas drove the core engine. Derrick handled documentation cleanup and the rebrand from "GPU Memory Profiler" to "Stormlog."

We shipped. The tool is real. It's on PyPI. People are using it.

And then Silas published a walkthrough — a complete tutorial taking someone from a clean training run to a deliberate memory leak to an OOM crash to the fix — and I realized I needed to actually go through it myself.

• • •

03

What the Walkthrough Is

The tutorial lives in a [companion report](#). It's a small PyTorch MLP on synthetic data, running on Apple Silicon. The point isn't the model — it's a sequence of scripts that walk you through the full debugging workflow Stormlog was built to support.

You run them in order:

BASH

```
python3 scripts/00_verify_env.py
python3 scripts/01_pytorch_native_debugging.py
python3 scripts/02_train_baseline.py
python3 scripts/03_train_with_leak.py
...
python3 scripts/09_train_fixed.py
python3 scripts/10_compare_runs.py
```

Ten scripts. Each one builds on the last. By the end, you've seen the same training job in three states — healthy, leaking, and fixed — with the evidence to understand exactly what changed between them.

• • •

04

Step 1: What PyTorch Already Tells You

Before Stormlog comes in, the tutorial makes you look at PyTorch's own memory counters. On MPS:

PYTHON

```
torch.mps.current_allocated_memory() # memory tied to live tensors
torch.mps.driver_allocated_memory()  # memory reserved by the backend
```

These two numbers are your starting point. If both stay flat, the run is probably healthy. If they climb together, something is holding memory it shouldn't be.

The limitation — and this is what the tutorial forces you to feel, not just understand intellectually

— is that this is a live-only view. You can watch the numbers while the process runs. When the process exits, so does your evidence. There's no timeline, no classification of what kind of drift you're seeing, nothing to hand to anyone else.

That's the gap. The native pass on the tutorial workload finished with peak allocated memory of 459 MB and a cached leak already visible at 201 MB. Something was drifting. I just couldn't yet explain what.

• • •

05

Step 2: The Leak

The bug is inside a class called `DeviceTensorRetention`. The training loop was caching activations and logits across steps so they could be inspected later:

PYTHON

```
def observe(self, *, hidden, logits, loss, step):
    self.hidden_cache.append(hidden.detach().clone())
    if step % 4 == 0:
        self.logit_cache.append(logits.detach().clone())
```

When I first read this, it looked fine to me. `detach()` breaks autograd history — that's the safe thing to do. `clone()` makes a copy — that seems safe too.

Here's what I was missing: `detach()` breaks the gradient graph, but the tensor is still on the device. `clone()` duplicates it, still on the device. So this code isn't making anything safer. It's creating new full-sized device tensors and appending them to a list that lives for the entire run.

The model metrics didn't flag this. Validation accuracy was 94.30%, basically identical to the clean baseline. But the memory story was completely different:

- **Clean baseline:** peak allocated 0.075 GB, slope 6.3 MB/s
- **Leaky run:** peak allocated 2.04 GB, slope 637.5 MB/s

That 100x increase in allocated slope is what Stormlog caught while the run was still alive. Without it, the first signal you'd get is the OOM crash — which the tutorial also triggers deliberately, just so you can see what that evidence looks like in the artifacts.

• • •

06

Step 3: Turning the Run Into Evidence

This is the part I built, and seeing it from the user side was genuinely different from building it.

Setting up the tracker takes maybe ten lines:

PYTHON

```
from stormlog import MemoryTracker

tracker = MemoryTracker(
    device="mps",
    sampling_interval=0.5,
    enable_alerts=True,
    enable_oom_flight_recorder=True,
    job_id="stormlog-tutorial",
)
tracker.set_threshold("memory_warning_percent", 70.0)
tracker.set_threshold("memory_critical_percent", 85.0)
tracker.start_tracking()
```

The training loop barely changes. The tracker runs alongside it. When the run finishes, you export:

PYTHON

```
tracker.export_events("artifacts/events.json", format="json")
tracker.export_events("artifacts/events.csv", format="csv")
```

What you get is a full artifact bundle: `events.json`, `events.csv`, `timeline.json`, `stats.json`, `alerts.json`, `timeline.png`, `timeline.html`.

The run is no longer a stream of numbers that disappears. It becomes something you can look at later, load in the TUI, send to a teammate, attach to a triage thread.

I built parts of that export pipeline. I knew it worked. But I hadn't actually run a training job and watched the artifacts appear until this walkthrough. There's a difference between knowing something works and understanding why someone would need it.

• • •

07

Step 4: The Fix

The fix isn't "monitor memory more aggressively." It's "stop retaining device tensors in the first place."

The replacement keeps bounded scalar summaries instead of full tensors. The reductions still happen on the device. What changes is what gets retained:

PYTHON

```
class ScalarSummaryRetention:
    def observe(self, *, hidden, logits, loss, step):
        self.summaries.append({
            "hidden_mean": float(hidden.detach().float().mean().cpu().item()),
            "hidden_std": float(hidden.detach().float().std().cpu().item()),
            "loss": float(loss.detach().cpu().item()),
        })
```

`.cpu().item()` is the move that matters. It copies a scalar result from the device to the host. You keep the information you actually needed — mean, std, loss — without keeping the tensor alive on the device.

After the fix:

- Peak allocated: 0.091 GB (from 2.04 GB)
- Allocated slope: 3.6 MB/s (from 637.5 MB/s)
- Validation accuracy: 94.69% (essentially unchanged)

• • •

08

What I Actually Learned

The tutorial didn't teach me Stormlog's API. I already knew most of it. What it taught me was the *sequence* — the actual experience of moving from "memory is climbing and I don't know why" to "here is the specific tensor, here is when it started, here is the fix."

I'd built parts of the tooling that makes that sequence possible without having lived through the problem that made the tooling necessary. That's a strange position to be in. Useful, in retrospect — it forced me to think about the interfaces carefully because I couldn't rely on intuition about the problem. But it also meant there were gaps in my understanding I didn't know were there.

The walkthrough closed some of those gaps.

The one that stuck most: `detach().clone()` is not a safe pattern for long-running retention. It looks defensive. It reads like someone being careful. But you're duplicating the full tensor and keeping it alive on the device. The right pattern for debugging is to reduce to scalars and move to the host — keep the information, drop the pressure.

I knew that abstractly before the tutorial. Now I've seen the numbers.

• • •

09

Run It Yourself

If you want to see the full sequence:

BASH

```
git clone https://github.com/Silas-Asamoah/stormlog_tutorial
conda env create -f environment.yml
conda activate stormlog-tutorial-mps
bash run_all.sh
```

It runs on CUDA too, not just Apple Silicon. The MPS-specific caveat is that hidden-gap analysis works differently — allocator and device counters are tightly coupled on MPS, so the gap classifier is demonstrated with a saved replay artifact rather than live counters. The walkthrough explains that in context.

Stormlog itself: `pip install stormlog`. Docs at stormlog.readthedocs.io^[2].

• • •

If you want to continue in this GPU-memory thread, the next piece is [What I Learned Profiling PyTorch Memory Leaks Across Two Backends](#)^[3] — the same Stormlog workflow on MPS and CUDA, with reproducible scripts and cross-backend numbers.

• • •

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://github.com/Silas-Asamoah/stormlog_tutorial

[2] <https://stormlog.readthedocs.io>

[3] <https://princeagyeituffour.com/writing/pytorch-memory-leaks-two-backends>

Typeset from the live article at princeagyeituffour.com/writing/stormlog-tutorial-walkthrough. The web version may be newer and carries the interactive parts.