Tracing [unified] Memory on heterogeneous systems

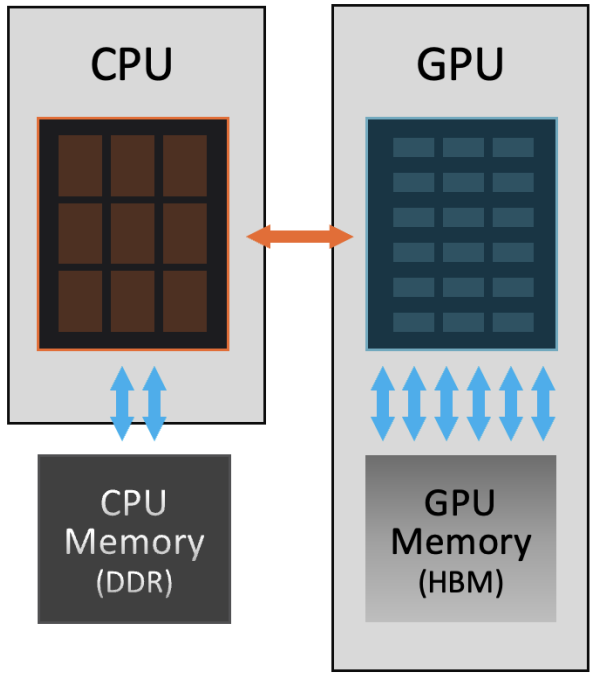
Lancelot Normand

Previously...

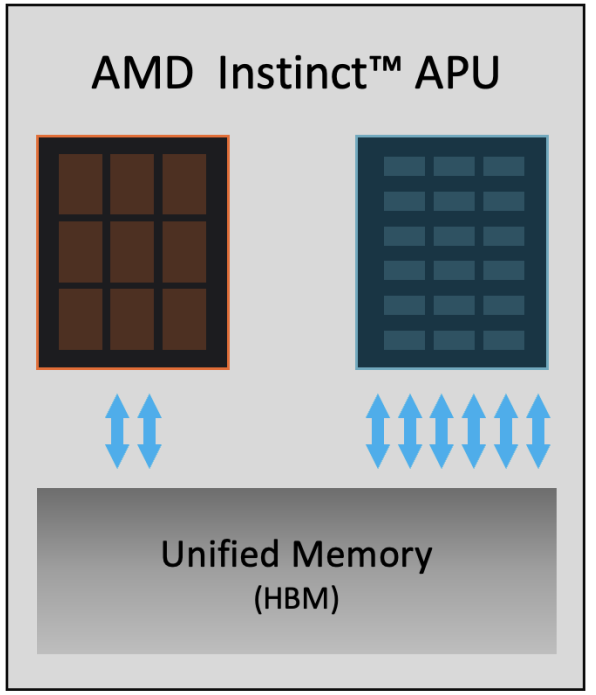
- Started exploring different ways of tracing programs running on GPU for HPC
- Introduction to TC, LTTNG, ROCPROFvX
- Goal : Identify performance bottlenecks on LLM finetuning jobs running on HPC nodes.

Given a machine learning job

- We want to trace the memory movement
  - Alloc/Free on host memory (CPU)
  - Alloc/Free on device memory (GPU)
  - Memory copies between host and devices.
- Why is this interesting in the case of unified memory?
  - Because in unified-memory there should not be any copies between host and device as they share the same memory.



(a) Discrete CPU and GPU.



(b) APU.

https://rocm.blogs.amd.com/software-tools-optimization/mi300a-programming/README.html

Research Questions:

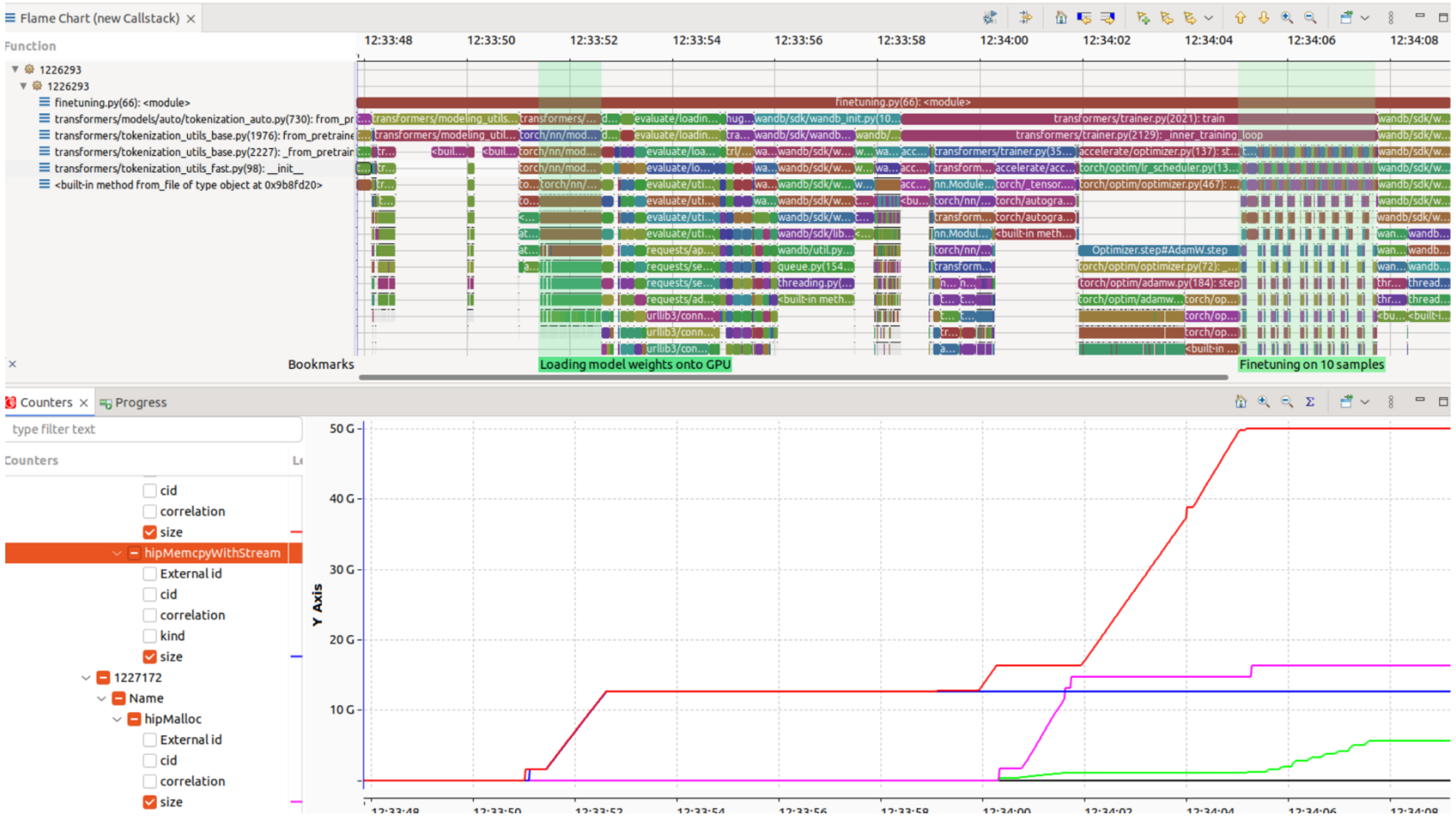
- What are the tools that can be used to trace programs running on both GPU+CPU
- What is the impact of not utilizing unified memory on Pytorch?
- How much memory is duplicated on average?
- How much time is spent in redundant operations?

As of today, none of the machine learning framework support unified-memory.

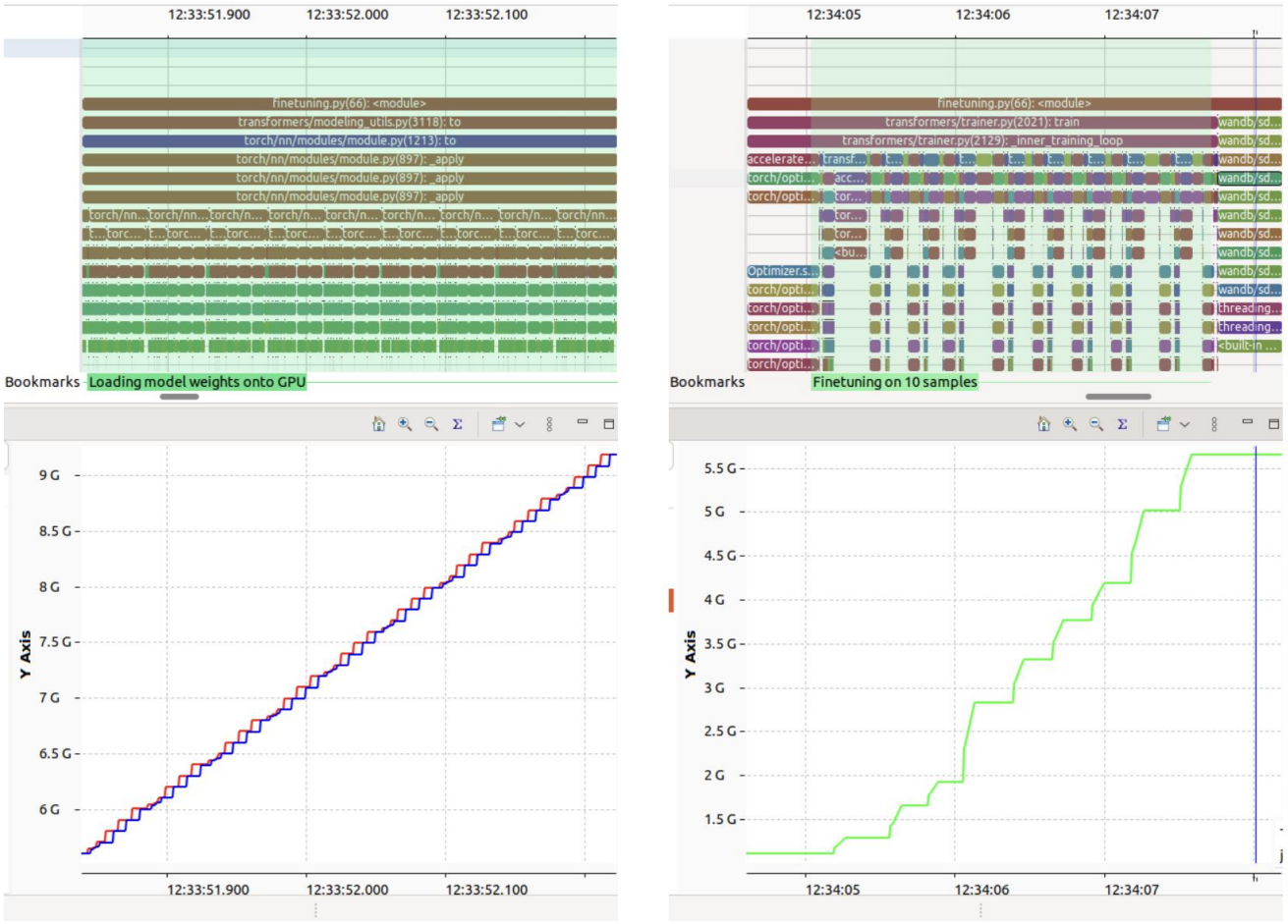
Visualizing memory w/ events

With common existing tools.

Measuring memcpy w/ events



Looking at Pytorch memcpy



Evidence of lack of support for unified-memory

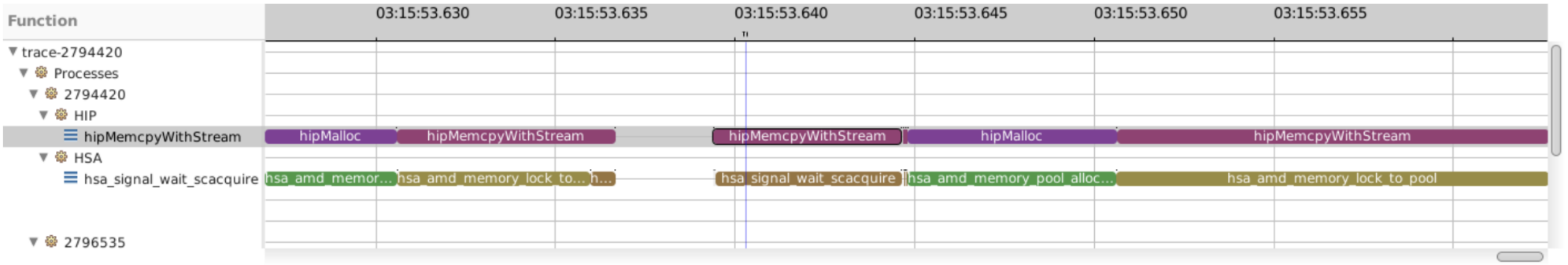


Figure 1.6: Snippet of a Trace of LLM finetuning task written with PyTorch. The presence of `hipMalloc` and `hipMemcpyWithStream` indicates that PyTorch does not consider the unified-memory architecture of the MI300a and treats it as a regular Discrete GPU

1.2.6 Besides Unified memory, what about the performance comparison of matrix multiplication?

Going back to the trivial code defined earlier, I ran the code with matrices of size $N \times N$ where N was ranging from 2 to 16384. This was run on all 4 GPUs.

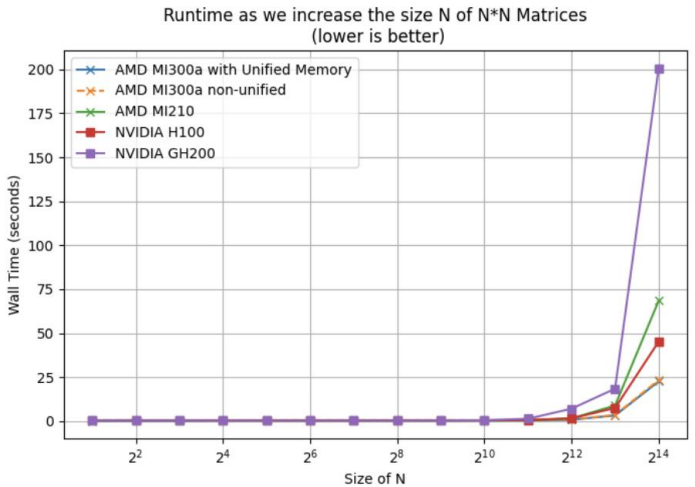


Figure 1.8: Runtime measured after multiplying matrices $A \times B$ of size $(N \times N)$ 10 times on different GPUs

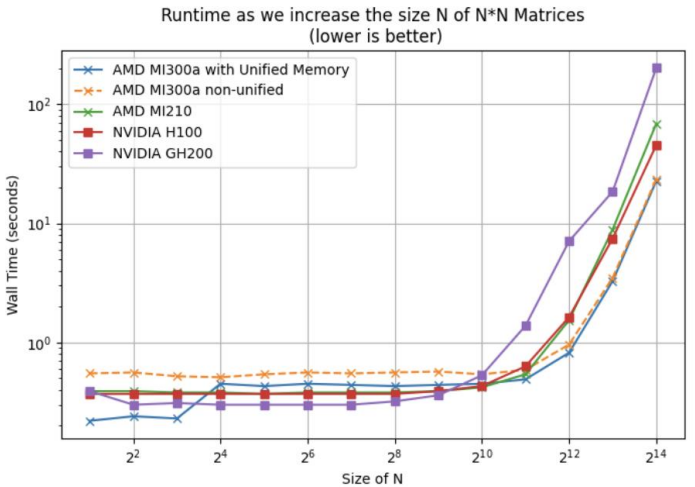


Figure 1.9: Same graph but with logarithmic scale on y axis

Limitations of the current tools for tracing memory movement

- Many tools offer produce some kind of memory trace, but they rarely combine *both host and device memory*.
- While Rocprofiler can trace HipEvents it does not expose pointer addresses of memorycpy.
- Pytorch is heavily instrumented and produces GPU memory traces and complete HIP traces w/ addr but does traces use different clocks that are not trivial to sync. *Bigger traces are also subject to corruption*.
- LTTng ust to trace malloc/free/realloc/calloc…

Limitations of the current tools for tracing memory movement (contd)

- So we have different tools with no common trace format, no common clock.
- Synchronizing them and stitching all that information is labor intensive but it should not be.
- This renders everything very difficult to reproduce and impractical.

A refined tool

To facilitate trace collection of both host and device memory


```
    "bytes": "79691776"
  }
},
{
  "cat": "alloc",
  "pid": 100,
  "tid": 563,
  "ts": 1741050000725115,
  "ph": "E",
  "name": "alloc 76.00 MB @ 0x7ef8c8c00000",
  "args": {
    "addr": "0x7ef8c8c00000",
    "bytes": "79691776"
  }
},
{
  "cat": "alloc",
```

New refined approach

- LD_PRELOAD to overload HIP functions and add custom tracepoint reported to LTTNG-UST
- And that’s it.

Benefits of this approach

- Completely framework agnostic.
- Only need LTTng UST
- A single trace is produced, very easy to reproduce.

We define our own tracepoints

memtrace > C hiptrace.h

```c
#undef LTTNG_UST_TRACEPOINT_PROVIDER
#define LTTNG_UST_TRACEPOINT_PROVIDER hiptrace

#undef LTTNG_UST_TRACEPOINT_INCLUDE
#define LTTNG_UST_TRACEPOINT_INCLUDE "./hiptrace.h"

#if !defined(HIPTRACE_H) || defined(LTTNG_UST_TRACEPOINT_HEADER_MULTI_READ)
#define HIPTRACE_H

#include <lttng/tracepoint.h>

LTTNG_UST_TRACEPOINT_EVENT(
    hiptrace,
    hip_malloc,
    LTTNG_UST_TP_ARGS(
        size_t, size,
        void*, ptr,
        int, result
    ),
    LTTNG_UST_TP_FIELDS(
        lttng_ust_field_integer(size_t, size, size)
        lttng_ust_field_integer_hex(void*, ptr, ptr)
        lttng_ust_field_integer(int, result, result)
    )
)

LTTNG_UST_TRACEPOINT_EVENT(
    hiptrace,
    hip_malloc_managed,
    LTTNG_UST_TP_ARGS(
        size_t, size,
        void*, ptr,
        unsigned int, flags,
        int, result
    ),
```

Our library overloading hip functions

```
20  }
21
22  extern "C" hipError_t hipMalloc(void** ptr, size_t size) {
23      using Fn = hipError_t (*)(void**, size_t);
24      static Fn real = load_symbol<Fn>("hipMalloc");
25
26      if (!real) return hipErrorUnknown;
27
28      hipError_t result = real(ptr, size);
29      if (result == hipSuccess) {
30          tracepoint(hiptrace, hip_malloc, size, *ptr, result);
31      } else {
32          tracepoint(hiptrace, hip_malloc, size, nullptr, result);
33      }
34      return result;
35  }
36
37  extern "C" hipError_t hipMallocManaged(void** ptr, size_t size, unsigned int flags) {
38      using Fn = hipError_t (*)(void**, size_t, unsigned int);
39      static Fn real = load_symbol<Fn>("hipMallocManaged");
40
41      if (!real) return hipErrorUnknown;
42
43      hipError_t result = real(ptr, size, flags);
44      if (result == hipSuccess) {
45          tracepoint(hiptrace, hip_malloc_managed, size, *ptr, flags, result);
46      } else {
47          tracepoint(hiptrace, hip_malloc_managed, size, nullptr, flags, result);
48      }
49      return result;
50  }
51
52  extern "C" hipError_t hipMemcpyAsync(void* dst, const void* src, size_t size, hipMemcpyKind kind, hipStream_t stream) {
53      using Fn = hipError_t (*)(void*, const void*, size_t, hipMemcpyKind, hipStream_t);
54      static Fn real = load_symbol<Fn>("hipMemcpyAsync");
55
56      if (!real) return hipErrorUnknown;
```

How we compile, and start tracing

Usage

Compile the hipwrapper.cpp into a library and stores it into `/usr/local/wrap_hip.so` or any preferred directory

```
g++ -fPIC -shared -o /usr/local/wrap_hip.so \
    wrap10.cpp hiptrace.c \
    -ldl -llttng-ust -rdynamic \
    -I. -I/opt/rocm/include \
    -L/usr/local/lib -Wl,-rpath,/usr/local/lib \
    -D__HIP_PLATFORM_AMD__
```

```
LTTNG_UST_CTL_PATH=/home/users/lancend/mestraces lttng-sessiond --daemonize --no-kernel --group=prl_collab -v
lttng create May05-trace
lttng enable-channel --userspace --blocking-timeout=inf blocking-channel
lttng enable-event -c blocking-channel -u lttng_ust_libc*
lttng enable-event -c blocking-channel -u lttng_ust_statedump*
lttng enable-event -c blocking-channel -u hiptrace*
lttng add-context -c blocking-channel -u -t vpid -t vtid
lttng start
LTTNG_UST_ALLOW_BLOCKING=1 LTTNG_UST_APP_PATH="/home/users/lancend/mestraces" LTTNG_UST_DEBUG=1 LTTNG_UST_VERBOSE=1
LD_PRELOAD="/usr/local/lib/liblttng-ust-libc-wrapper.so:/home/users/lancend/code/wrap_hip.so" <YOUR_EXECUTABLE>
```

To export the CTF trace into json call `ctf2ctf <TRACE_DIRECTORY> > trace.json`.

Some may point out that this shares a lot of similarities with exa-tracer…

And indeed, it does…

- In fact, exa-tracer implements everything I did but better.
- The good thing about it is that both traces use the same format so one can use them interchangeably in the trace analysis part

The problem of measuring duplicated memory

Limitations

- Duplicated memory is difficult to compute, while at the type of a copy we know for sure memory has been duplicated, it is not always obvious when host memory is rewritten when it is not explicitly freed.
- Copied pointers may not always align with the pointers returned following a malloc. Therefore, we need to maintain an interval of address range so that upon freeing a bigger block subsequent reported duplicated chunks within it are substracted from the current total duplicated bytes

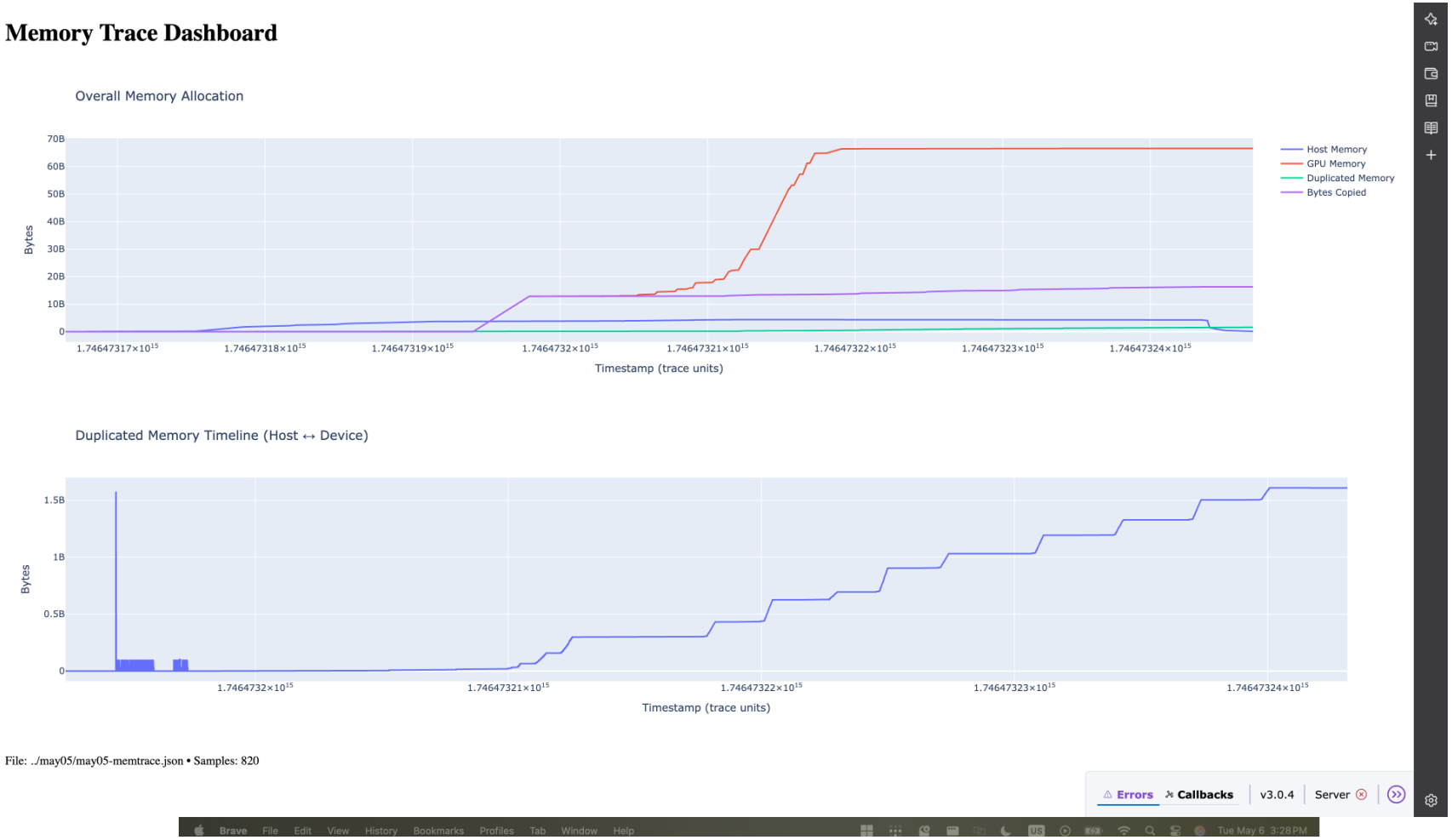
Limitations part two

- Lots of freeing the null ptr which means the information might have been freed before…

What’s next?

- Push further the analysis and vis.
- Make sure the duplicates are accounted correctly

Trace analysis (in progress)



Thank you