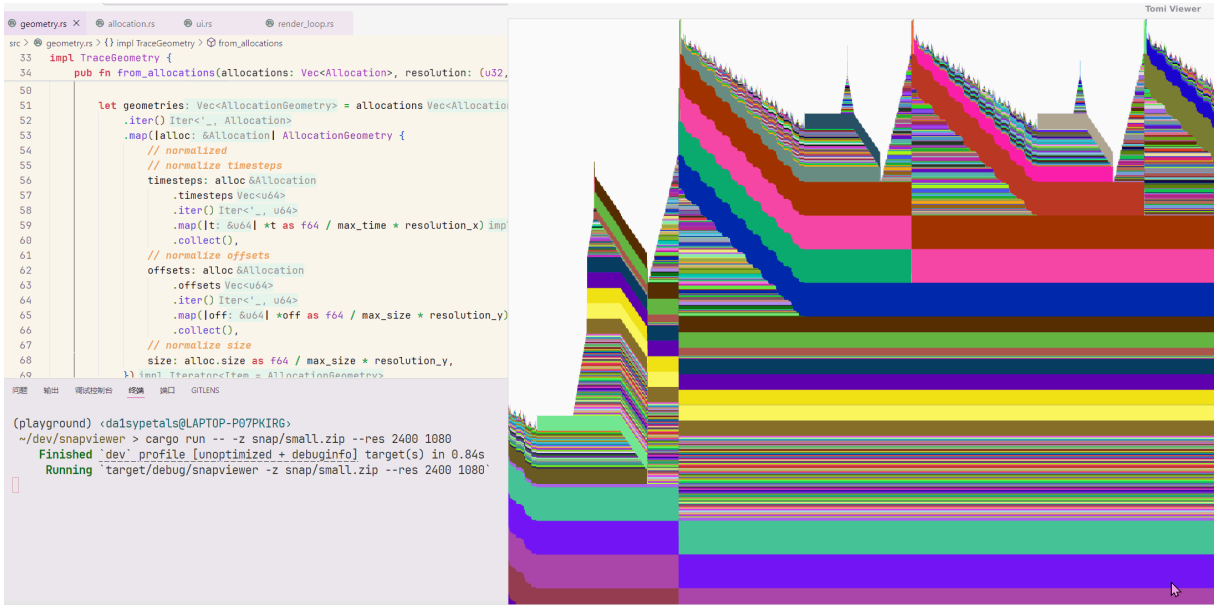
SnapViewer

- PyTorch 在训练模型的时候常常会 OOM，这时候就需要对显存进行优化。当一些简单的方法（降低 batchsize 等）以及行不通的时候，可能就需要对模型本身的显存轨迹进行分析。
- 这时候你会看到这个文档，他会教你如何记录 memory snapshot 并且在 这个网站 上进行可视化。
- 但是有一个很大的问题是：这个网站 太卡了。如果你的模型很小，snapshot 只有几个 MB，流畅度还算能看；如果你的模型比较大，snapshot 达到几十甚至几百 MB，那么这个网站就会变得非常卡，帧率最低可达每分钟两三帧。
- 我去看了这个网站的 js 代码，它主要做了这些事：
  1. 手动加载 python pickle 文件；
  2. 每一帧都重新将原数据解析为图形，然后再每一帧渲染到屏幕上。

  这个解析逻辑是用 js 写的，因此性能嘛…
- 我在对一个几 B 参数量的模型进行 snapshot 的时候发现了这个问题。
  - 为什么需要自己优化，而不是用现成的 LLM 基础设施？长话短说，这个模型是 researcher 自己设计的，里面含有大量的和 LLM 完全不同的模块。现在好像大家默认深度学习

只剩下 LLM 了，以至于甚至有些 tech lead 都认为 LLM 的基础设施可以轻松接到很多其他模型上面…偏题了

- 我原本写了个简单的脚本用来解析 snapshot 里面的内容，尝试借此发现模型里面的显存分配问题；但是在我对着这个模型工作了一个月之后，我终于受不了了。于是有了这个项目：SnapViewer.
- TLDR：将 memory snapshot 的图形解析出来，用一个巨大的 triangle mesh 表示，然后复用渲染库对 mesh 的渲染能力进行渲染。这是一个上百 MB 的 snapshot，在我的集显上跑的还算流畅：



解析 snapshot 格式：

1. 看 pytorch 源码，在 record memory history 函数旁边 docstring 有 snapshot 格式的解释：https://github.com/pytorch/pytorch/blob/main/torch/cuda/memory.py 但

是这个解释是不全的！应该是后续添加功能的时候没有维护 docstring

2. 还是 pytorch 源码，这部分是解析 snapshot 得到的 dict 的：https://github.com/pytorch/pytorch/blob/main/torch/cuda/_memory_viz.py 将 allocator trace 转换成时间线上的显存轨迹，然后送入网页 viewer 的 js 代码将显存轨迹转换成多边形，显示在网页上。
3. 将这部分 js 代码转写为 python 代码。每个多边形代表一次 allocation，对应分配大小，callstack 等信息。功能：将 snapshot dict 保存在 json 文件中。
    - 优化：json 在磁盘上太大了，我们这样做：直接在内存里压缩（python zipfile），然后再写磁盘；在可视化的时候，从磁盘读取 zip （Rust zip） 之后在内存里解压。
        - 这是考虑到实现难度和占用的一个 tradeoff：这样做至多在解析 json 的时候造成一个内存峰值，在 json string 被 deallocate 之后内存就会下降，还可以利用 Rust serde-json 的基础设施（Rust 的 serde-pickle 是个残废的库，不能解析 recursive 的结构）。

渲染和交互

1. 考虑到渲染的分配信息在整个可视化过程中不会变化，将所有的 allocation 组成一个巨大的 mesh，然后直接送到 GPU 上进行渲染。根据用户的行为改变相机的位置，窗口大小等等。
    - 使用 rust 的 three-d 库。对 mesh 有比较好的抽象，而且支持将 mesh 一次性传到 gpu 上，不会每帧做 cpu->gpu transfer；支持鼠标键盘事件。

ii.（世界 - 窗口）坐标转换：先从窗口转换到世界坐标（乘 scale，加上窗口的 center position），再从世界坐标转换到内存位置（预定义的 scale）

1. 制作 UI 和交互，包括：
    - 显存刻度线（考虑现在的屏幕上要显示几条刻度线，精细到什么程度；移动刻度线的世界位置，让刻度线在屏幕的位置保持一致）。
    - 移动和缩放（基于鼠标位置缩放，根据鼠标位置计算新的屏幕中心位置）。
        1. 记录原本的 scale （1/zoom)
        2. 将 zoom 调整为新的 zoom，然后计算 新旧 scale 的比例
        3. 鼠标到新旧屏幕中心的距离之比 = 新旧 scale 之比，根据鼠标世界位置不变的 invariant，更新新的屏幕中心位置。

欢 迎 围 观 & star! → https://github.com/Da1sypetals/SnapViewer ← 如果你也有需要，欢迎试用一下 :)