Why torch level hijack?

北京大学 尹锦润

北京大学 尹锦润

对于计算劫持

Hijack

Simpler Megatron with trace generation

aicb, echo

MegatronAtten(torch.nn.Module)

```
def cuda_timing_decorator(func):
    def wrapper(*args, **kwargs):

        start_event = torch.cuda.Event(enable_timing=True)
        end_event = torch.cuda.Event(enable_timing=True)

        start_event.record()
        result = func(*args, **kwargs)
        end_event.record()
        torch.cuda.synchronize()

        elapsed_time_ms = start_event.elapsed_time(end_event) * 1000
        return result, elapsed_time_ms
```

Megatron: _apply_QK

torch.bmm(A, B)

Hijack

@cuda_timer

torch._C.bmm(input, mat2, *, out=None) -> Tensor

Aten(Compute Library)

Hijack

intercept cudaLaunchKernel

extensions(torch/csrc/)

autograd

profiler

wrapper/hook/etc

Modify

Hijack

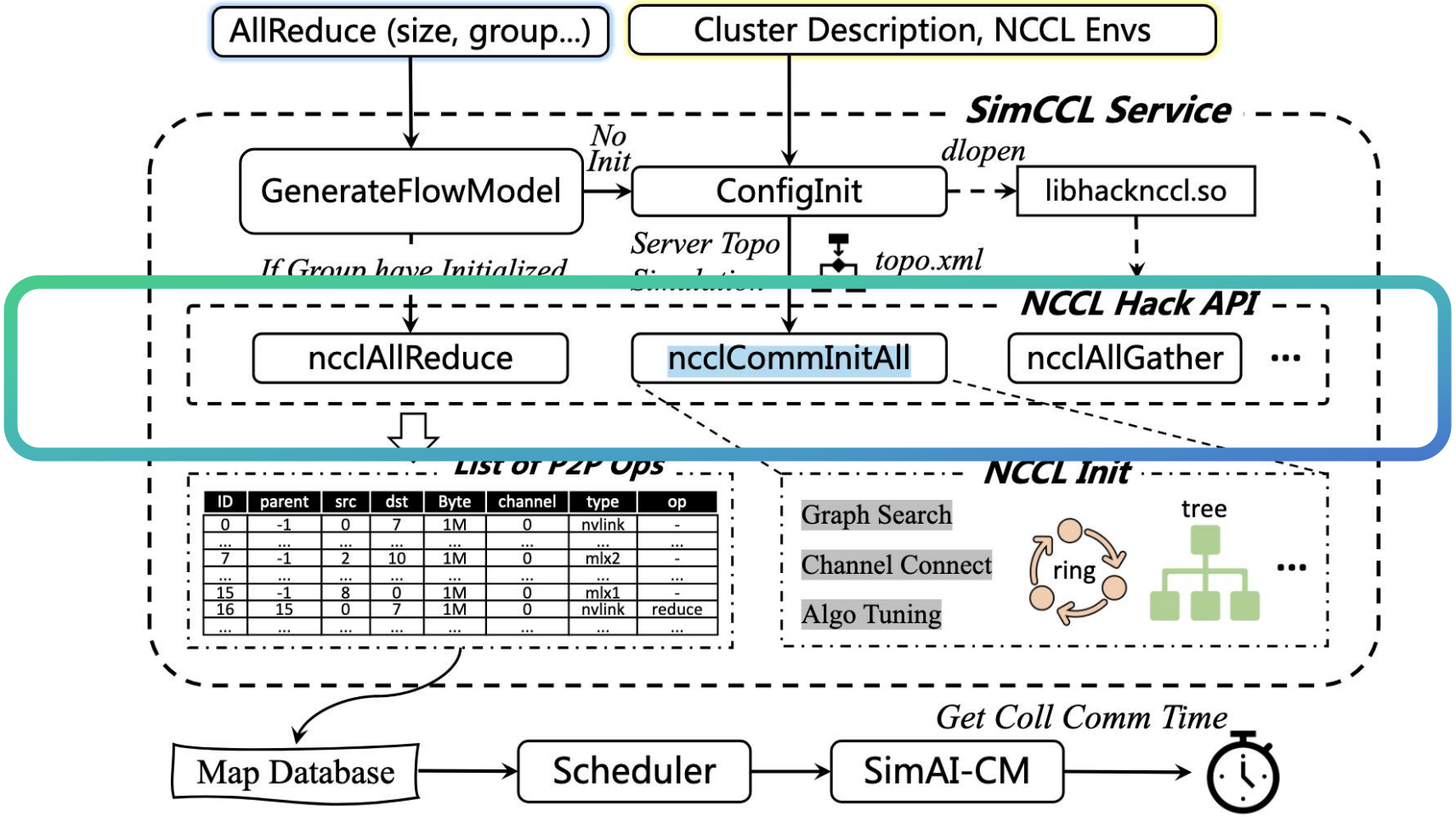
just modify

profiler

c10(Actual implementations of tensor & data store)

对于通信
劫持

- npkit(Echo) : profiling base
- Hack Nccl(Sim AI): 572 lines of Code.



Torch 层面-目的

- 完全单机，初始化多机多卡环境后，串行生成 0 ~ N-1 卡的 trace。
- 对于通信：
  - 采用 SimAI 的 Hack Nccl。
  - 用单机将 all2all/send-recv/allgather 操作劫持，记录，不进行任何操作，时间 = 0，数据 = random
- 对于计算：
  - Mock & Half-trace。
  - 调用 kernel（torch._C 下的，非常细粒度的 kernel）：
    - 前 500 次（所有卡上累计）：正常运行，生成 trace
    - 之后：不运行，填充 random，生成 trace，时间为之前的记录平均

Why not Mock(like Echo & SimAI)

- 无法完全模拟真实运行情况-粒度不细，环境不真
  - 对于 CUDA 操作，会有一次 launch，之后在某个时间才会 synchronize——有多个 kernel 同时运行，多个 cuda stream。
  - Echo 对于 Kernel Overlap 使用了 XGBoost Model 进行修正，but accuracy... 61.48% achieve 5% error with 3090.
  - SimAI - @cuda_timer 实现中显式同步。
  - Torchlevel - 前面时间基于运行实测，非常精确。
- 拓展性：0
  - 对于即有框架都要用工程量进行 Mock 来换取准确性。
  - 用户自定义：手动，改完 Megatron 再在模拟器上改一遍。。。
  - Torchlevel：只要是 torch 算子，都可以。
- 通用性 max，集成现有 torch.profiler, chakra 等 workload 生成工具。
  - deepseek 的训练&推理 trace 就是用 torch.profiler 生成的

Why not full-trace(like chakra)

- 耗时太长
  - 重复操作太多，而大部分操作其实连 tensor size 都是一样的
  - 从系统底层进行省略，节省大部分操作

Why not kernel-hijack(just intercept cudaLaunchKernel)

- Why chakra / profiler works:
  - dependency is important!
  - only kernel’s benchmark result is not enough!

Why several times?

- 第一次运行，kernel 会选择参数，进行微调，选择最佳的执行方式运行。
- 多次保证稳定性。

How can hack-nccl work?

- 一台机子，我怎么知道多线程代码同步到了哪里？
  - dependency is important!
  - 所有网络操作可以拆解为 send/recv，这也是 astra-sim 模拟处理的问题。我们只要把这种操作写入单机 trace，剩下的...
  - $Send_A$(B, tensor), $Recv_B$(A, tensor) -> dependency

How to build dependency?

- 通信的 dependency：我们只需要记录，astra-sim 会处理。
- 计算的 dependency：
  - dependency 分为两种，数据流和控制流
  - (Kineto) profiler 会记录操作需要数据位置，astra-sim 会处理数据流。
  - (Execution) profiler 会记录调用栈，cpu -> gpu，可以处理控制流
  - btw, torch.profiler 有两种， chakra = (two traces -> chakra trace)

Method evaluation?

- 通用性 max，除非用 tensorflow
- 单机完成
- 高精度：真实的 trace
- 高效拓展规模：
  - kernel 并不会随着卡数增长而增长。
  - 卡数增加只会增加 workload 生成逻辑上的规模，运行时候大部分都在 random。

But at what cost?

- 工程量不大，难度很大
- torch.profiler 就是这个逻辑，函数调用栈上增加记时操作，我们只需要理解其 C++ backend 咋写的就行了
- 然后在关键的地方增加几行 & 写一些辅助函数
- 和 torch.profiler 不同的是：
  - torch.profiler 实际上是写了个 decorator
  - 我们不仅仅是 decorator，还要修改调用的函数（省略调用）
- 理解 profiler -> 几乎理解大部分 torch C++ 代码实现逻辑
- 在 python 层面写少量内容。

目前进展

对于调用的栈帧

执行对应的栈递归:

根据栈帧获取对应代码情况（？）

不知道在干啥（？）

按是否起始栈帧进行时间戳加入:

```
void PythonTracer::recordPyCall(
    ThreadLocalResults& tls,
    PyFrameObject* frame,
    bool is_startup_frame) {
  static constexpr auto E = EventType::PyCall;
  const auto key = [&]() -> TraceKey {
    auto code = THPCodeObjectPtr(PyFrame_GetCode(frame));
    if (code.get() == module_call_code_) {
      auto locals = THPObjectPtr(PyFrame_GetLocals(frame));
      auto self = THPObjectPtr(PyDict_GetItemString(locals, "self"));
      Py_INCREF(self.get());
      auto back = THPFrameObjectPtr(PyFrame_GetBack(frame));
      TORCH_INTERNAL_ASSERT(back != nullptr);
      return tls.intern<CallType::PyModuleCall, E>(
          frame, self.get(), back.get());
    } else if (code.get() == optimizer_hook_) {
      auto locals = THPObjectPtr(PyFrame_GetLocals(frame));
      auto self = THPObjectPtr(PyDict_GetItemString(locals, "self"));
      Py_INCREF(self.get());
      auto back = THPFrameObjectPtr(PyFrame_GetBack(frame));
      TORCH_INTERNAL_ASSERT(back != nullptr);
      return tls.intern<CallType::PyOptimizerCall, E>(
          frame, self.get(), back.get());
    } else {
      auto back = THPFrameObjectPtr(PyFrame_GetBack(frame));
      auto f_back = (back.get() != nullptr) ? back.get() : frame;
      return tls.intern<CallType::PyCall, E>(no_ephemeral_t(), frame, f_back);
    }
  }();
  const auto time = c10::getApproximateTime();
  is_startup_frame ? start_frames_.push_back({key, time})
                   : queue_->getSubqueue()->emplace_py_call(key, time);
}
```