浅谈AI框架前端对后端的影响

https://zhuanlan.zhihu.com/p/390856360

None Thu Sep, 09 00:42

想写这篇article源于和 @Byron 的一些讨论，感谢百忍同学的输入，有所启发。

很早以前在计算所读书的时候，第一次接触Verilog，用它写一个键盘鼠标的控制器，当时组里一个资深的硬件开发同学看到自己写的第一版代码综合出的网表layout时说了一句话'**garbage in, garbage out**'，提醒自己写硬件描述代码跟软件代码的区别，让自己去follow Verilog的可综合语法规范进行了大范围的重构。

毕业之后，有过几年时间从事EDA编译器的开发，为Verilog开发过FPGA器件上的综合工具，从工具开发者的视角对'**garbage in, garbage out**'的理念有了更深一步的认识。

时隔多年，在AI system行业，从事AI优化工作的时候，会发现上层描述写得放飞一些自我，无论是静态图描述(TF 1.X为代表)，还是动态图描述(PyTorch eager mode)，也会给底层优化工作带来一系列的headache(静态图的零碎算子给策略探索，和图优化带来了一系列负担，动态图过于Pythonic的写法给模型布署带来挑战)。等到自己切换角色，从AI硬件的视角bottom-up来看AI系统全栈优化工作时，对这个问题的体感就更明显了。

总的来说，AI框架前端对后端可能的影响有几点：

1. **框架前端导出静态图的顺利与否，这会直接影响后端优化效果。**

以PyTorch模型为例，在这篇paper里提到了从PyTorch前端导出TorchScript的问题使得FB的一些同学设计了一个看起来----有些'**tricky**'的解决方案。在我们之前的一些实际经验中，Detectron2这样的FB自己开发的模型在导出TorchScript的时候也并不算顺利，需要做不少工作。

而后端，通常来说是期望能够看到一个静态图，因为静态图对优化更为友好，包括计算调度、算子优化、图变换、显存分配，策略探索等等。导出静态图不够smooth，就会影响到后端优化效果。

2.框架前端是否引入比较多的动态shape行为。

在前端能够导出静态图的情况下，如果对相同的静态图，存在高频的dynamic shape行为，也会给后端优化带来麻烦。主要包括：优化耗时、优化占用的存储资源。Torch XLA的项目里有一个相关的notes。特别是对于一些硬件加速器，静态编译的耗时比较大，对这一点会更为敏感。

3.框架前端是否引入了比较多的dynamic control flow的执行行为。

更细节一些，如果这种dynamic control flow是基于AI框架native的API来描述的（比如torch.where这种写法或tf.cond、tf.while_loop的写法），对于后端还是比较友好的，因为这种native API是能够构造出静态图(对于system guy来说，我们是有多么喜欢静态的东西^-^)。

但是，用习惯了eager mode的用户很容易在python代码里写出大量的pythonic的条件控制逻辑，这些基于python语法描述的dynamic control flow行为就未必都那么容易生成静态图了。

这种dynamic control flow容易导致的后果有：

- 以dynamic control flow为边界，图被切割得零碎，引入大量的host侧和加速器的交互开销。
- 如果基于trace的方法，把dynamic control flow给materialize成具体的执行path，就可能触发隐式的dynamic shape，出现2里的问题。

比较理想的是对这种dynamic control flow的建模描述，通过解析Python AST，将其转换到AI框架的静态图表示上。在PyTorch里，这个事情想做到没有多少corner case，恐怕不那么容易。

另一个隐蔽的会加剧dynamic control flow的行为是implicit broadcast。implicit broadcast的麻烦在于，在shape维度引入了运行期的动态性，如果再考虑到rank维度的动态性，故事就更精彩了。这个新加的动态性，对于强假设静态图的后端，要么会引入大量的优化开销，要么为了避免引入过多开销，把图切割得七零八碎带大量的性能penalty。

4. 框架前端设计引入过多host侧和加速器的交互，会潜在影响后端优化效果。

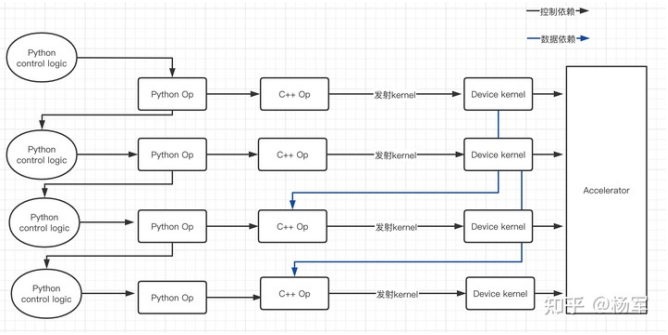
基于Python环境进行AI建模已经是一个de facto的行业习惯了。而为了高性能，通常会用Python来完成控制和胶水逻辑，真正性能消耗的大户还是会通过c++将计算任务dispatch到硬件加速器上完成配合。随着加速器的性能越来越强，Python侧的overhead也就受到了更多关注。理想情况下，Python侧的执行流如果能够和加速器上的执行流尽可能pipelin起来，相当于最终E2E耗时取

$$max(T_{Python}, T_{Accelerator})$$

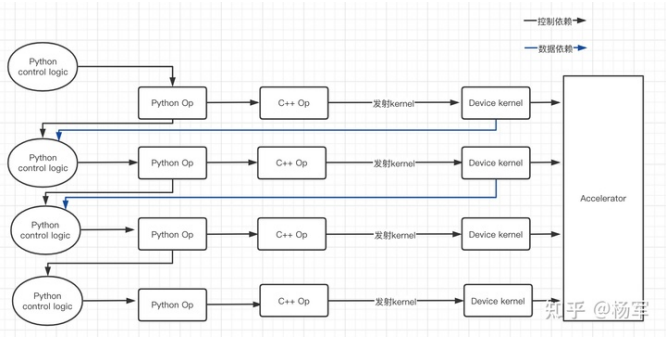
，并且

$$T_{Python} < T_{Accelerator}$$

，这是我们最期望的情况。类似于下面的执行流：



然后，理想很美好，现实很骨感，实际上，很可能会出现这样的行为：



即，**某些Python侧的执行逻辑，需要等待加速器上的device kernel执行返回结果以后，才能继续往下执行，哪怕不存在实质上的data dependency。**这种情况下Python侧的overhead就可能拖慢E2E的执行时间，极端情况下出现*E2E耗时*

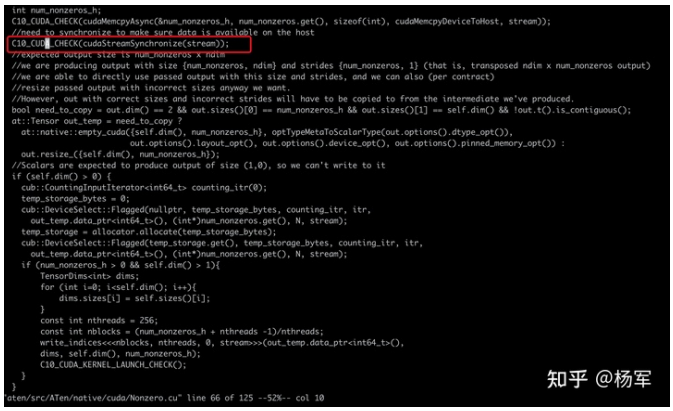
$$\simeq T_{Python} + T_{Accelerator}$$

的情况。

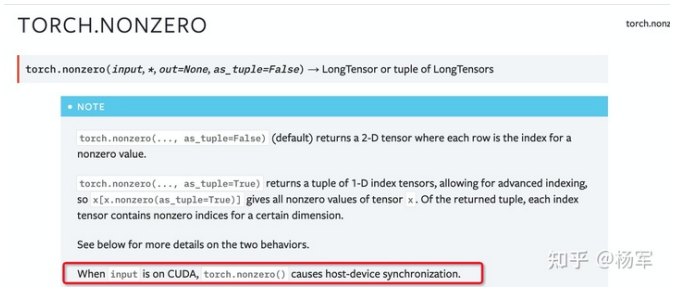
一些导致host侧需要等待加速器上device kernel返回执行结果才可以继续进行后续device kernel发射的例子如下：

```
{
  auto& allocator = *::c10::cuda::CUDACachingAllocator::get();
  auto dataPtr = allocator.allocate(num_matrices*sizeof(int64_t));
  int64_t* mat_el_end_indices_device = static_cast<int64_t*>(dataPtr.get());

  search_end_matrix_indices(mat_el_end_indices_device, num_matrices, indices_dim0);
  AT_CUDA_CHECK(cudaMemcpy(
    mat_el_end_indices_host.get(),
    mat_el_end_indices_device,
    num_matrices*sizeof(int64_t),
    cudaMemcpyDeviceToHost
  ));
}
// Need a pointer to an array to access within a lambda
int64_t* mat_el_end_indices = &mat_el_end_indices_host[0];

Scalar beta = 0;
Scalar alpha = 1;

int64_t mat_el_begin_idx = 0;
size_t workspace_buffer_size = 0;
void* workspace_buffer = nullptr;
auto& allocator = *::c10::cuda::CUDACachingAllocator::get();
::c10::DataPtr dataPtr;

// See Note [Enabling Deterministic Operations]
deterministic = deterministic || globalContext().deterministicAlgorithms();
cusparseSpMMAlg_t mm_alg = deterministic ? CUSPARSE_COOMM_ALG2 : CUSPARSE_COOMM_ALG1;

// Iterate through each set of 2D matrices within the 3D
// tensor inputs, performing a matrix multiply with each
"sparse/cuda/SparseCUDATensorMath.cu" line 904 of 1043 --86%-- col 23
```

和

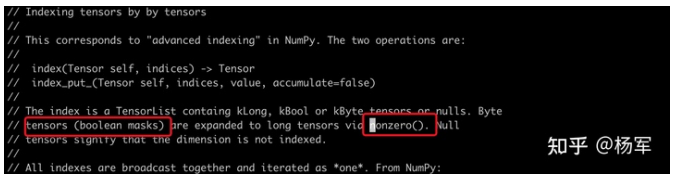


其中，nonzero这个算子带来的麻烦会更多一些，在官方的API描述里有这样一段介绍：



Well, actually we don't want too many host-device synchronizations.....

这个host-device sync操作存在的原因是*nonzero()*计算过程中会需要使用GPU kernel上计算的结果用来参与一些Tensor shape相关的处理逻辑，这个逻辑放在CPU上完成，于是就引入了host-device的同步开销。而把这个事情变得更复杂一些的是用户在写模型的时候，经常使用一些略显花式的indexing操作，这些操作会间接触发nonzero()的操作：



如果做花式indexing的indices正巧对应于一个device memory的Tensor，并且是bool型的mask，那么也会触发nonzero()的操作，引入隐式的device-host synchronization的开销。

5.Eager mode对建模用户更为友好，缩短了建模调试的周期，但同时也为后端优化带来了新的挑战。

新的AI硬件计算力越做越强，但如果对于eager mode灵活性为后端带来的挑战缺失足够关注的话，很可能会发现算力under-utilized的情形。针对单个重点模型，类似NV优化MLPerf那样贴身优化总归可以解决（把模型改写的更适合后端优化假设），但想要达到通用覆盖，特别是当用户不能接受动不动贴身支持改模型的这种合作模式时，就需要在软硬全栈设计的早期阶段对这类问题给予足够的关注了。

发布于 07-19

文章被以下专栏收录