深度学习框架的思考 袁进辉

【P1】

今天想讨论三个问题。第一个问题杨军其实刚才问了，就是软件和硬件怎么分工，我一会也会讨论一个例子，我们觉得有的事情你让软件做了，芯片可能没有必要那么复杂。

还有一个问题就是希望从我们的角度，来判断一下什么问题是已经解决了的，什么问题是还没解决。如果是已经解决了，基本上就是可用的状态了，如果没解决，也许大家要一起想办法怎么把它解决。

第三个问题就是框架发展了好几年了，像Tensorflow和pytorch这样的，可能很多人都已经觉得写出这种代码的人都是高手，他做的都已经是无懈可击了，或者挑剔不出来毛病了。但是我想讨论几个例子，其实有可能再回头去看的时候会发现做的还蛮糙的，特别是当你有新的视角去看的时候，比如说我原来没考虑大模型的需求的时候，也许有些问题不那么明显，一旦考虑大模型需求，你有一些新的视角进来的时候，就会发现原来很多东西可能都漏掉了，正好涉及到我前一段时间写的几篇博客，蒋老师就让我总结一下，然后正好来作为一个引子来讨论。

【P2】

引子就是训练GPD3这种大模型，英伟达他们做了一些实验，你会看到用了几千张卡，300多台机器，分了64个组，每一组是一个流水线中的stage，所以它是分了64个流水线，每个stage是6台机器， 6台机器之间是data parallelism，就是数据并行，因为机器之间的带宽还是比机器内部的带宽稍微低一些。然后机器内部是 Model parallelism，这其实是做weight的划分的，所以你可以看到神经网络向集群中去映射，是一个蛮复杂的过程，可能换一个模型又换成另一种做法了。这个又回到像上午尹老师提到过的CGRA里面的一些问题，我觉得大家只是在粒度上不太一样，但是本质上是非常相似的，就是逻辑的东西和物理的东西怎么映射，怎么排布，怎么路由等等这样一系列问题，所以这个作为例子就引出了问题来了。

【P3-P5】

刚才提到的复杂，是编程的复杂性，还有一个问题就是运行效率的问题，运行效率的问题在微观层面和宏观层面，我觉得也有很多的对应。比如说以前在芯片里面，大家都知道memory wall，指的是计算的非常快，但是仿存取数据取指令的latency要高的10倍到100倍，那怎么解决呢？以前有cache，乱序流水，各种各样的编译器体系结构，各个层面都做很多的工作。在宏观层面其实也有非常相似的一些问题，现在每个芯片算的是非常快的，每个GPU上面有几千个core，每来一小片数据，它可能是几十毫秒，100毫秒就完成了。同时这些core访问他自己的defase memory的带宽也非常高，就是pbyte的这种带宽，芯片之间马上就降了一个数量级，你用nvlink差不多是一两百G的byte，然后服务器之间如果用RDMA的话又降了一个数量级，当然像刚才提到那个例子，英伟达是两台服务器之间用了8根200gbps的这种高速的传输，所以它能做到机器之间也和nvlink的带宽差不多，但总体上你会看到这里面也有hierarchy的现象，再加上深度学习本身有粒度，像随机梯度下降，这和以前memory reduce，scatter处理的任务有点不太一样，它这里面粒度特别小，可以想象在这样的集群中，每一瞬间都有很多事情在发生，在这里面就有很多的问题，像怎么去overlap那些计算和传输，怎么让计算感觉不到所谓的饥渴，可以每算完一个，然后立刻就能开始下一个等等。我只是抛出来这样一个类比，可以让我们在宏观层面遇到一些问题的时候，能在微观层面找到一些答案。综合到编程的复杂性还有效率的问题来说的话，对用户最友好的一种做法就是用户写程序的时候，基本上都是逻辑层面的，神经网络多少层怎么去连，具体逻辑的网络怎么到集群里面并行，怎么overlap，怎么划分，怎么Placement等等这些问题。如果我们框架的编译器这些都能搞定，打个比方就像硬件的EDA一样，在设计新建的时候，布局连线都可以预先做很多的工作，就可以一定程度上解决这个问题。这个其实就是需要通过编译器来解决的，这个编译器可能是既有宏观层面的，也有微观层面的，来把逻辑的东西和资源的东西生成一个所谓的物理的执行计划的时候，能够做到一方面是自动生成，另一方面是找到尽可能优的方案。

【P6-P7】

还是这个例子，比如说像前面有数据运行，后面有模型运行的时候，实际上中间涉及到一套的数据路由。数据路由现在我们已经知道了是有规律的，一个模式到另一个模式切换的时候，中间应该是插入什么运算，前项应该插入什么运算，反向应该插入什么数据路由这些，现在很多人都很清楚了。所以总结来说，如果给定 Placement，就是每个op到底发到哪个设备上去，比如它整个所有的设备可能分成若干组，每个op也许发到所有的设备上去，也许只分到一个子集上去。然后它发到这一组设备上去之后，它又怎么划分，它是切分了数据还是切分了weight，这是parallelism的问题。如果给定前面这两个的话，怎么自动生成执行计划？实际上我觉得这个问题已经解决了，因为无论是从框架层面自动生成中间的路由方案也好，还是用手工的布线或者是手工的组网也好，现在都有例子了。自动的生成，我觉得现在跑的在比较靠前是oneflow和mindspore，然后Tensorflow和pytorch现在还没有比较general的方案出来。Tensorflow在XLA里面是有 TPU上的方案，但在GPU上现在还没有很work的方式出来。我觉得给定config的话，placement和parallelism怎么自动生成真正的执行计划这个问题可能都ok了。

【P8】

第三个就是对于auto-placement和auto-parallelism，现在paper里面我们都看到很多，但是真正的像比如说抛出来GDP3的时候，怎么能完全自动的解决，其实还是很难的。但在芯片设计领域，已经把这个问题变成可work的了，其实这个问题没有想象的那么困难。我们做了一些研究，初步的来说，我们觉得不应该把它作为一个随机的概率，在一个很大的解空间里面不加任何的demand-knowledge去搜索，这个空间实在是太大了。当去做一些理论的分析，比如说流水并行到底给数据并行或者模型并行带来了什么gap，流水平行它本质的问题到底是什么，比如说它的核心其实是load balance的问题等等。做这样一些推导的时候，也许会发现比较好的解。它不是在解空间里面随机的分配的，它应该是占整个解空间很小的一部分，那一小部分就是非常好的解，一旦离开这个很小的一部分，解的效果就立刻下降了，所以在这里面可以加入一些理论分析或者demand knowledge，也许用一些rule的话就能得到很不错的结果了，这是一个抛砖引玉的做法。

还有一个就是单设备的自动代码生成，刚才大家也讨论很多了，这个现在来说是比较难的，也是基本上把它当成搜索的问题，不管是TVM去做，还是poly去做，都是在解空间里面去解，只不过是一个是离散空间去解，一个是说用把它relax一下，转换成连续的再去解，然后再去找到一个整数的解等等。

所以迄今为止我抛出几个观点，一个是什么问题已经解决了，我认为第1个问题已经解决了，第3个第4个现在来说有一部分可工作的解。还有说刚才提到的第一个问题，什么应该是软件做，什么是硬件做，我们的理解是通过oneflow的实践来说，我们觉得在这种芯片协同这种工作就应该软件去做，不要用硬件去做。所以在上午提到的那种超大的芯片，那个芯片上有很多tile，编程的时候既要解决上面tile之间的这种路由，又要解决内部的路由，所以这本质上是在解决分布式的问题。但我们在框架层次实际上是可以解的比较好的，没必要把一个芯片搞得那么大，你完全可以把这种功能非常聚焦的单个的芯片做得非常好，同时他们中间的路由和协调在软件层面就能解的非常好。所以我以 Cerebras为例来说软硬件的分工，软件如果能做得非常好的话，特别是在这种宏观层次，那其实有些问题没必要在芯片层面来解决。涉及到分工的话，宏观的并行分布式，还有微观的单设备的代码生成，大家也可以分工，如果把每个层面的子问题解决的非常好了，其实最后一个整体的解决方案就很不错了。

【P10】

我再提第三个问题，也许现在的框架看上去已经非常蓬勃了，但是实际上中间还是有很多case没有考虑到的，特别是有大规模分布式的时候。举一个talli的例子，这个例子其实在TensorFlow里面有时候会发生，这个例子简单来说就是有两个op,op1和op2，他们分别需要三份检测和两份检测，同时有数据搬运操作给他服务，数据搬运操作也是需要消耗显存的。那调度的时候现在的框架的问题出现在哪里呢？就是在于判断 op是不是满足发射的条件，只看输入是不是available。其实现在所有框架都是这么做的，两个输入一旦 ready了，立刻就把op发出去了，发射出去之后，以TensorFlow为例，会在compute里面再去malloc output的空间，大多数时候是能申请到的，但是如果申请不到的话，那什么时候可以继续往下走是不确定的。这里举了一个能跑下去的例子，但是还有的跑不下去的，比如说到了第二步之后，o2先执行的话，它一定是能先执行下去的，因为o2只需要两个，o2成功执行之后又释放出了更多显存，然后o1就可以执行了，但是o1和o2在2的时候实际上都是ready的状态，如果这个调度器先调了o1的话，就会出现一些不可预测的情况。有的时候 o1就在当前的schedule的线程上在做，就是inline那种情况，它就把这个线程阻塞住了，阻塞之后意味着也行别的ready的op，包括o2也没法调度了。如果做得好的话o1是可以到另一个线程去调度的，这样它就可以继续执行下去，但到底要取多少个线程也是一个比较复杂的问题。这其实是一种dynamic，就是说在运行之前没有做类似于死锁的分析，整个任务能不能跑下去一定程度上是寄希望于运气的。如果提前做一些静态的分析，流水线的pipline分析等等，是有可能把这个问题解决掉的，比如说可以在o1，o2之间加一个dependency。

【P11】

还有一个问题就是计算和搬运在现在的框架里面没有统一的表达，计算有计算图，但是搬运的时候实际上搬运的操作没有放在图里面的。这个是TensorFlow的paper，可以看到在搬运的时候会插入send、receive，但实际上只是在逻辑层面插入了send、receive，但比如说跨了机器跨了设备的时候，底下真正的CUDA copy不是send和receive来完成的，是在send和receive下面又触发了一系列的call back，而call back不在于它 graph engine的管理范围之内。

【P12】

这里面其实会有一些比较tricky的问题，比如说刚才我们提到了为了解决死锁要额外加入dependency，而这时候加入一些dependency就会使得一些op有了多重的依赖。问题就是call back函数在框架里面它只是解决一个依赖的问题，但是如果下面的运算有两个依赖，前面的一个条件只解决了一个dependency的问题，另一个dependency没解决，这时候实际上需要再到 graph scheduler那里去查询另外的依赖，这也会使得整个问题会变得比较复杂。TensorFlow里有一种可能的解决方法，就是在call back里面再去查询 scheduler的各种计数器的状态，也许会返回说你可以做了，当然就可以用call back顺利执行下来，如果scheduler返回不能做，实际上意味着这个call back需要再包一次，再挂到 schedule的某一个waiting list里面去。当然这是一个fix，打个patch，但实际上在现在的框架里面还没有这么做，也就是说它没考虑中间的那些数据搬运，是否有多重的依赖，是否想控制它的顺序等等这样一系列的问题。

【P12】

第三个问题就是动态确定线程池，在很多分布式系统里面都有这个问题，大家会经常用线程池来解决问题，但是 memory pool size一般都是heuristic确定的，但这个的时候也会出现一些问题，比如size确定了之后，也可能会触发死锁，因为任务的并行力度可能超过了thread的个数，我前一段时间写了个博客来分析这个问题。还有一些情况就是，如果thread pool是可以动态调整，但可能你不知道调整到多少。这个图比较抽象，但现在的框架的scheduler大部分都是使用的最后一个,因为通常一个线程管理 graph的state，counter以及launch这些kernel，但是一个线程管理很多设备的时候，每个device都要set device call next，基本上大家不采用这种办法。一般来说会让每个device都有一个单独的线程去调度它，这个时候又会引入一些问题，这个细节我就跳过，我觉得经过各种讨论之后，最后就会到最后一个状态，就像 pytorch这种例子一样，每个device它有专门的线程为他服务，而且线程既管理 scheduler的counter，也管理launch kernel这些操作。这里面核心的就是要确定 thread pool size，而难题在于现在的框架没有保证kernel launch是不会阻塞的，如果没有这个保证的话，你就不知道到底要申请多少个线程是安全的。但是如果我们能够在框架层面去确保每个kernel发射出去之后，它是调度中不可剥夺的，它一定可以从头到尾成功执行下来，那线程的数量就等价于device的数量。这实际上就引入一个新的问题，就说runtime的调度粒度，不能是以前那种thread的那种粒度，像操作系统OS thread的力度，它需要的是类似于user level thread的这种每个op都可以是一个user thread，这个时候无论有多少op其实kernel thread大小是差不多的。

我大概是抛一些砖，可能我们很多人会觉得一些东西已经无懈可击了，是非常完美的了，但实际上细想的话还是有很多问题的。好，我的分享就到这了，谢谢。