8. 深度学习编译器 - 运行时

和编译性语言类似，深度学习模型的开发也可以分为“编译”和“运行”两个阶段。前面介绍的主要是“编译”阶段，这节主要目的是讨论“运行时”需要考虑的一些核心问题。 当然，“编译”和“运行”的划分和具体的实现有关，因此这节讨论的问题未必对所有实现都适用。

对于深度学习模型，“运行时”需要考虑哪些问题呢？对照普通编程语言的运行时，首先需要加载可执行程序，加载完之后从程序入口指令开始执行。如果是多核 CPU，此时需要决定由哪个 CPU 来开始执行。执行过程中需要考虑堆栈等内存资源的管理，还需要考虑函数调用如何传参、返回结果等。 如果将普通可执行程序中的指令换成深度学习中的算子（KERNEL），则深度学习模型运行时和普通可执行程序运行时具备类比性，比如，运行过程需要考虑算子到设备的运行关系、内存/显存资源的管理、KERNEL 调度以及调用子网络的方式等。我们可以粗略的将上述问题分为两类：

1. 硬件资源管理：包括对加速设备、内存和显存的管理等

2. 计算任务调度：包括计算任务的划分、KERNEL 调度、多线程及多模型运行管理等。

8.1 硬件资源管理

运行神经网络模型的物理设备通常由 CPU、GPU、AI 芯片等多种异构计算资源组成。运行时需要具备管理这些硬件资源的能力。包括：

（1) 将某个 KERNEL Launch 到某个计算设备运行

（2) 分配计算芯片内部资源，例如 GPU 的显存， AI 芯片内部的 SRAM

（3）设备之间数据拷贝，例如在用 GPU 计算前，需要将数据从内存拷贝到 GPU 显存；

（4）限制某些资源的使用，比如可以使用的 CPU 资源、GPU 加速设备、内存以及显存资源等。

上述能力依赖于计算设备驱动提供的接口完成。但是在具体实现上，运行时可以选择不同的方法。以显存分配为例，为限制每个模型可以使用的显存，运行时可以分配一块大的显存，然后从这块大的显存上分配计算过程种用到的显存，如果分配不出来，则报 OOM 错误。另一种方案是，运行时直接调用 cudaMalloc 接口分配显存，但维护一个计数器统计显存使用情况，一旦超过规定的限制，则报 OOM 错误。Tensorflow 使用的是第一种方案， 并通过 VirtualDevice 对其进行了封装。还有一种方案是，在编译阶段决定好需要的内存/显存大小，以及运行过程中每个 Tensor 的 OFFSET，然后在运行开始时分配一块或几块显存，后续不再走分配的逻辑，直接根据 OFFSET 访问。TVM 中的 Graph Executor 使用的就是这种方案。

8.2 计算任务调度（KERNEL 调度）

8.2.1 OP 到计算设备的映射

当系统中存在多个计算设备，特别是多种异构计算设备时，比如最常见的，数据中心中服务器中既有 CPU，也有 GPU。 就需要考虑 OP 到计算设备的映射问题，即将每个 OP 分配给哪个设备。如果目标执行环境完全已知，OP 到计算设备的映射可以在编译阶段完成，运行时直接根据映射好的结果执行。当目标执行环境不能提前预知，或者存在动态变化时，可以在运行开始的时候完成 OP 到计算设备的映射（相当于是运行时预编译的过程）。

OP 到计算设备的映射需要考虑可用 KERNEL、 模型运行的整体性能等因素，有多种实现方法。最简单的是基于规则的方法，比如为每种设备分配优先级，当一个 OP 可以在多个设备上运行时，将其分配给优先级最高的设备。其次可以基于 Cost Model 预估将指定 OP 分配到某个设备后，对应 KERNEL 的运行时间以及相关数据的拷贝时间，基于预测结果，以最小化模型运行时间为目标决定 OP 到计算设备的映射结果。除此之外，还可以通过 Auto-Tuning 的方式，生成若干可行的分配结果，通过模拟运行获取其实际运行性能，再从其中选择性能最优的分配方案。Tensorflow 中主要使用的是基于 Cost Model 的方案。同时，Tensorflow 也提供了相关接口供用户指定 OP到设备的映射方案。

8.2.2 KERNEL 运行过程管理

编译过程会为神经网络中的每个 OP 选择对应的 KERNEL，这些 KERNEL 之间存在输入输出的数据依赖。运行时，需要构建 KERNEL 的依赖关系、决定 KERNEL 的执行顺序，并在 KERNEL 运行之前，将输入数据传递给 KERNEL。对于动态神经网络，运行时还需要处理分支、循环等控制逻辑。

运行时 KERNEL 调度的实现方案可以分为两大类：基于图的调度和编译后顺序调度。

基于图的调度

根据组成神经网络的图（可以是拆分后的子图），以及选择好的 KERNEL，构建运行时的图。这张图比神经网络图简单，主要用于维护 KERNEL 之间数据的依赖关系。运行时会维护一个 Ready 队列和一个线程池，在开始的时候选择没有输入依赖的节点加入 Ready 队列，通常为输入节点，或者参数初始化节点。之后，依次从 Ready 队列中取出节点，并交由线程池执行该节点的 KERNEL。执行完成之后，根据运行时图所记录的 KERNEL 依赖关系，将 KERNEL 的计算结果 “传递” 给其他节点作为输入，同时判断接收节点的所有输入是否已经全部就绪，如果是，则将其加入 Ready 队列。依次循环，直到 Ready 队列为空。

上述过程是典型的基于 BFS 对图进行遍历的过程，由于 KERNEL 最终的执行顺序和线程个数以及每个 KERNEL 的“执行时间”相关基，因此基于图的调度是一个动态的过程。比如，如果某个 KERNEL 是后加入的，但是它的执行速度很快，依赖它的 KERNEL 也可能被先执行。基于图的调度的优点是不会出现头阻塞的情况，即由于前面 KERNEL 执行很慢，导致后面 KERNEL 不能执行的情况。缺点是运行时需要维护各种依赖关系，开销比较大。

基于图的调度对于分支的处理比较简单，因为不同分支上的计算由不同的节点完成。在调度时，根据分支条件决定将哪个分支的节点加入到 Ready 队列即可。对于循环可以有不同的处理方法，一种是逐轮计算，要求上一轮计算完成之后再开始下一轮的计算，这种实现起来比较简单。另一种是在上一轮尚未完成计算之前，只要满足循环条件，即进入到下一轮循环。由于不

同循环轮次对应的是图中的同一个节点，因此要从 Ready 队列中区分不同轮次的循环，满足 KERNEL 之间数据的依赖关系。

Tensorflow 原生的运行时就是采用了基于图的调度方案。具体的代码在 Executor 中，下面的博客有对相关代码较为详细的介绍:

https://www.cnblogs.com/jicanghai/p/9572213.html,
https://www.cnblogs.com/jicanghai/p/9572217.html

编译后顺序调度

编译后顺序调度是指在编译期间将组成神经网络的图线性化，然后按照线性化后的顺序依次执行。线性化之后每个节点的 KERNEL 可以类比为一条“指令”，KERNEL 的调度因此就类似于指令 Dispatch 的过程，KERNEL 的输入输出数据可以用“寄存器”来抽象，KERNEL 之间的数据通过“寄存器”来传递，KERNEL 执行时只需要从对应寄存器读取数据即可。如果图中不包含分支或循环等动态控制逻辑，线性化只要对图进行前序遍历即可。如果包含控制逻辑，在前序遍历的同时，要考虑节点的执行顺序，在线性化后的同时插入相应的跳转 KERNEL。TVM 中提供了两种运行时的实现：GraphExecutor 和 VM。两种都是编译后顺序调度的方式，GraphExecutor 是针对静态图的，而 VM 则是针对动态图的。

编译后顺序调度的方式相当于在编译期间解决了神经网络中 KERNEL 数据依赖的问题，保证在执行某一个 KERNEL 时，依赖数据均已经 Ready。这种方式的优点是运行时的实现简单，缺点是可能会存在类似头阻塞的问题，例如，KERNEL N + 3 的数据已经 Ready, 但是 KERNEL N + 1 的数据还没有 Ready，因此 KERNEL N + 3 不能被发射执行。该过程本质上和 CPU 中指令发射是同样的过程，因此一种解决方案是模拟 CPU 中的乱序发射，运行时也支持乱序发射。除此之外，由于深度学习中 KERNEL 执行时间通常较长，因此也可以利用异步执行来解决该问题。也即当前 KERNEL 在执行完成之前就会返回，这样可以继续调度下一条指令。异步执行需要保证 KERNEL 在真正执行时，其输入数据已经 Ready 了。有两种方案：一种是类似 CUDA Stream 的方式，保证同一个 Stream 中的计算是按先后顺序执行的，即可以保证数据依赖。另一种方案是用线程间的同步机制，即 KERNEL 在执行之前首先会等待输入数据 Ready。Tensorflow Runtime（GitHub - tensorflow/runtime: A performant and modular runtime for TensorFlow, Tensorflow 重构的运行时）采用的是这种方案。

运行时的线程模型

无论推理还是训练，运行过程都会涉及多线程加速，线程模型即指这些线程的组织和使用方式。运行时的多线程包括三个层面：（1）无相互依赖的 KERNEL 通过多线程并行执行，也即 KERNEL 之间的并行；（2）KERNEL 内部通过多线程加速，也即 KERNEL 内部并行；（3）多个 Batch 的数据通过多线程并行执行，也即数据层面的并行。这三个种并行可以使用多个线程池来实现。例如，KERNEL 之间的并行使用一个线程池；KERNEL 内部并行使用另一个线程池。对于数据并行执行的情况，可以为每个 Batch 的数据单独维护对应的线程池，也可以所有 Batch 的数据共享底层线程池。

上面介绍的 3 种多线程并行主要是对 CPU 设备。对于使用 GPU 加速的情况，上述三种多线程可用不在 CPU 上进行数据准备的部分。对于在 GPU 上运行的计算， KERNEL 之间的并行以及多个 Batch 数据的并行需要通过多个 Stream 来实现；而 KERNEL 内部的并行依赖于 CUDA KERNEL 实现。

Tensorflow 在 config.proto 中提供了若干参数对运行时的线程池进行配置：

配置参数	说明
inter_op_parallelism_threads	Kernel 之间并行处理的线程数，会在创建第一个 Session 时创建
intra_op_parallelism_threads	Kernel 内部并行处理的线程数
use_per_session_threads	设置为 true, 则为每个 session 单独创建 inter_op 线程池，否则所有 session 共用一个线程池。TF 官方表示会在后续版本废弃，建议使用 session_inter_op_thread_pool。
session_inter_op_thread_pool	Vector,为每个 session 指定需要创建的线程池，可以指定多个，每个线程池的数量可以不同

如果设置了 session_inter_op_thread_pool, 且其中包含多个配置，则可以在运行时通过 RunOptions 指定需要使用哪个线程池：

```
message RunOptions {
 …
// The thread pool to use, if session_inter_op_thread_pool is configured
int32_t inter_op_thread_pool = 3;
…
}
```

这些参数的设置对运行时的性能有非常大的影响，论文 Exploiting Parallelism Opportunities with Deep Learning Frameworks 详细分析了其影响和如何优化这些参数的设置，感兴趣的朋友可以参考。