OSDI 2020 | RAMMER如何进一步"压榨"加速器性能

2020-11-12 | 作者：谢志强

编者按：传统的深度学习框架为了模块化设计，通常使用互不感知的两层调度模型，导致无法充分发挥硬件的计算性能。针对现有深度学习框架的局限，微软亚洲研究院和北京大学、上海科技大学合作提出了 RAMMER：一种可以成倍甚至几十倍地提升深度学习计算速度的编译框架。这套编译框架的背后是微软亚洲研究院打造的深度神经网络编译器 NNFusion，目前已在 GitHub 上开源。

传统深度学习框架由于其自身的局限性，如今还远没有充分发挥出硬件的计算性能。微软亚洲研究院的研究员们在一些测试集上发现，现有的深度学习模型只能用到 GPU 2%到40%的性能。传统的深度学习框架通常把神经网络计算抽象为由算子（operator）与依赖关系构建而成的数据流图（data flow graph, DFG），并按照拓扑序将算子逐个调度给硬件，或像 MXNet 中的依赖引擎（dependency engine）可以将多个算子同时调度到 GPU（算子间并行性，inter operator parallelism）不同的流（stream）执行。而在此之下又存在另一层调度器负责充分发掘算子内并行性（intra operator parallelism）并将计算任务映射给更小粒度的处理单元。

这样的两层调度法尽管为系统设计带来了一些简洁，但在实际的部署中，两层调度器互相不感知会导致几个问题：运行时的调度开销很大，算子间并行性没有被有效利用，以及忽视了算子内和算子间两种并行性的相互影响。

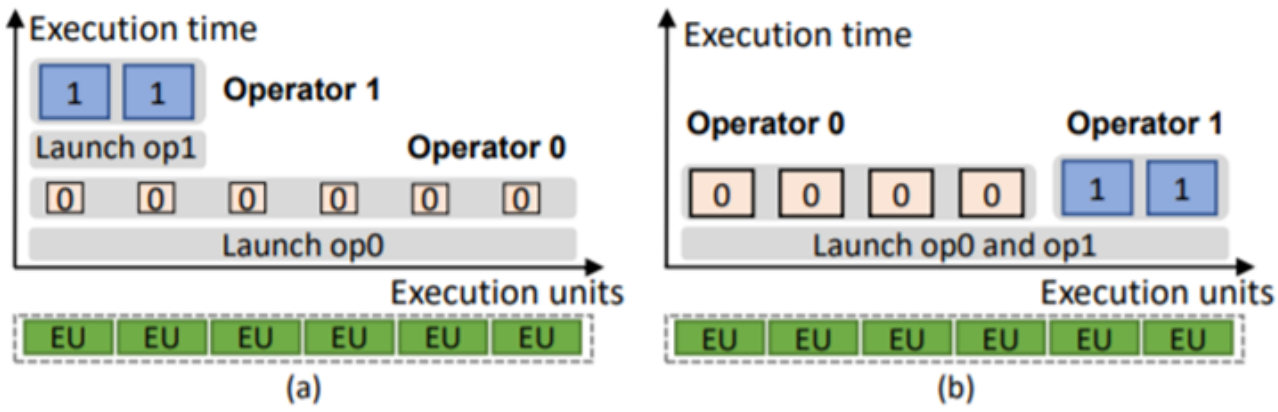

图1：(a) 传统低效率的调度方案，(b) 优化后的调度方案

为了能够打破这种僵局，微软亚洲研究院和北京大学、上海科技大学合作提出了一种可以成倍甚至几十倍地提升深度学习计算速度的编译框架 RAMMER。研究员们将原数据流图中的算子解析为 rOperator 并将其分解为更小的

调度单元 rTask，将底层的硬件抽象为由多个虚拟执行单元（virtualized execution units, vEU）组成的 vDevice。在这套新的抽象下，用户可以通过更细的 rTask 粒度将数据流图调度到多个 vEU 之上，兼顾了计算任务中的两种并行性与底层计算资源的协调。整个调度方案在编译期生成并“静态”映射到硬件计算单元上，因此可以天然地消除掉许多原本存在的调度开销。

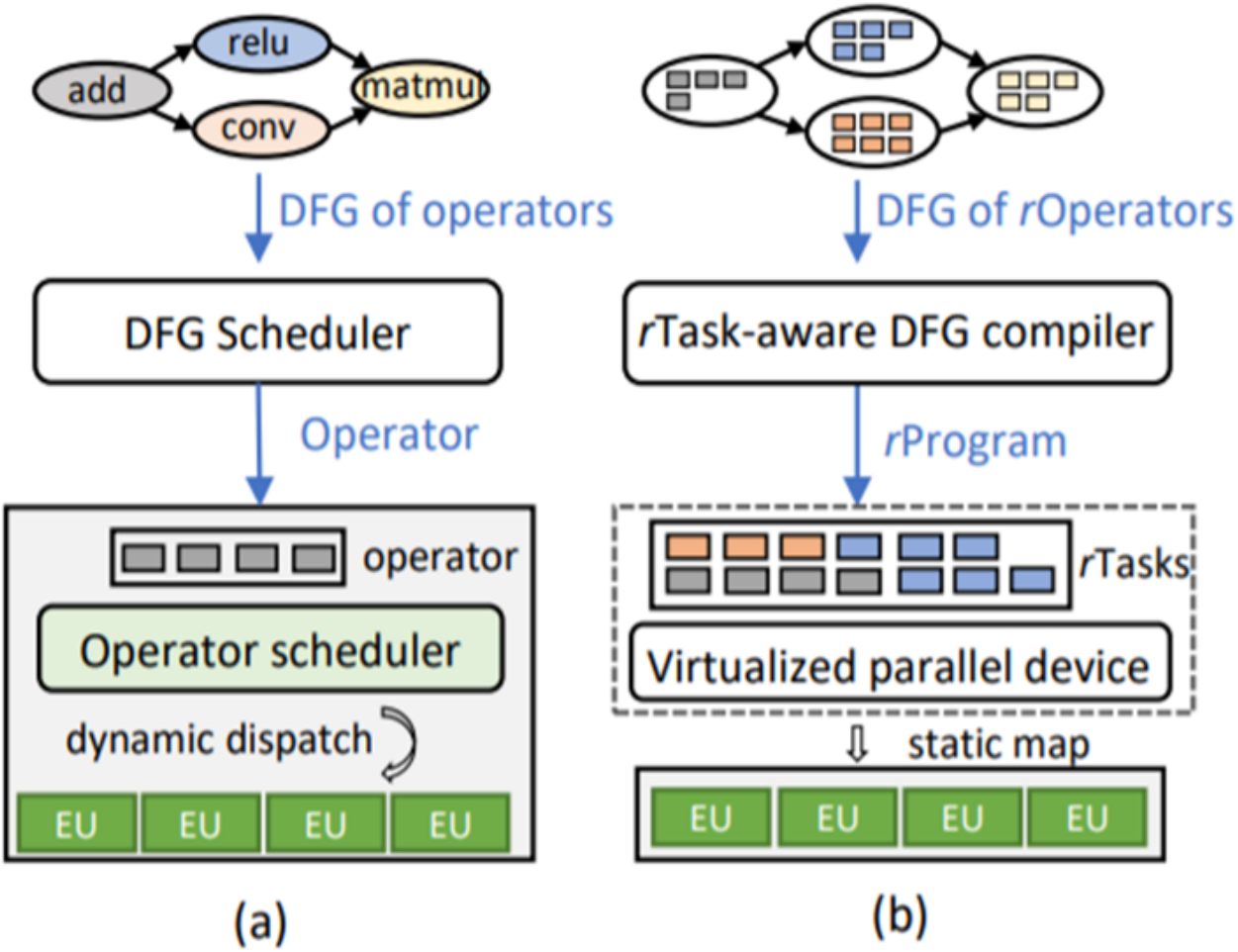

图2：(a) 传统深度学习框架，(b)RAMMER 深度学习编译框架

尽管上述介绍中用了不少 CUDA 的概念，但不难发现 RAMMER 整个设计是硬件通用的。它可以很好地适配到诸如 GPU、IPU 等主流深度学习加速器上。研究员们在 NVIDIA GPU、AMD GPU 和 GraphCore 上都评估了这套编译技术所能取得的性能收益。与 TensorRT 相比，RAMMER 在部分模型上实现了最高3.1倍的性能加速。

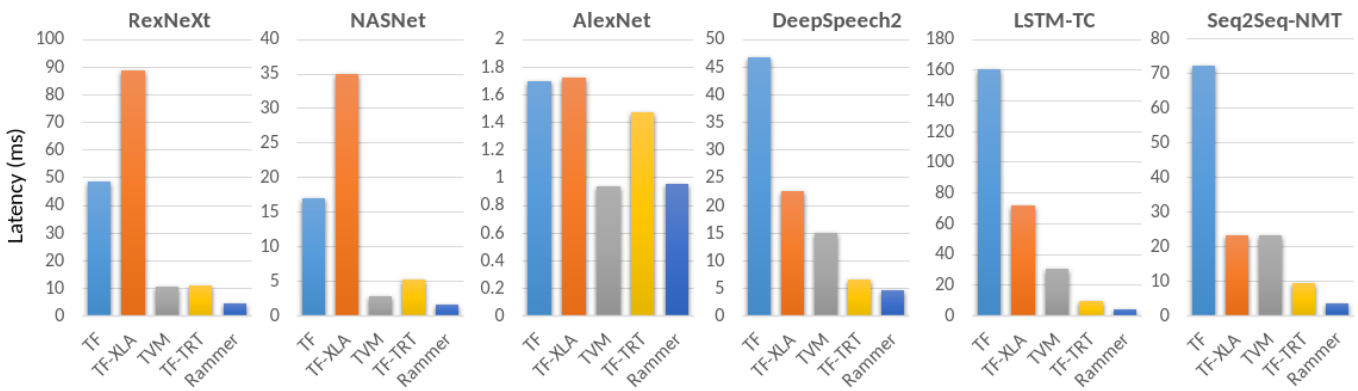

图3：在 NVIDIA V100 GPU 上批次大小设置为1的端到端的模型推理时间对比

RAMMER的设计与实现

RAMMER 背后是微软亚洲研究院在过去一年多时间里打造的一套名为 NNFusion（https://github.com/microsoft/nnfusion）的深度神经网络编译器（DNN compiler）。NNFusion 能够将现有的模型编译为对应设备的可高效运行的源码，同时支持用户自行替换内核实现或自动从外部导入高性能的内核实现。为了方便地与现有的代码库和 GPU 程序设计模型兼容，RAMMER 采用的是源代码转换的方式，而不是像 TVM、Tensor Comprehension [1] 一样定义新的计算抽象需要用户提供算子的计算逻辑。

我们先来了解一下什么是 rTask。rTask 是组成 rOperator 的互相独立的更小的任务单元，也是 RAMMER 抽象中最小的调度单元。在 NVIDIA GPU 上，对于用户提供的内核 CUDA 实现，通过一个小的解析器，可以将每个线程块（thread block）转化为一个 rTask。所以 rTask 可以利用原本内核实现中的语义，虽然这样 rTask 在实现上是与原本的程序设计模型耦合的，但是也大幅度降低了所需的工作负担。

那么要如何创建 vDevice 和 vEU 呢？目前，根据硬件的特性再配合简单的启发式搜索就可以创建 vDevice 和 vEU 了。如果 V100 中有80个 SM，每个 SM 最多能够运行32个线程块，那么就可以创建一个包含有2560个 vEU 的 vDevice，而后根据 rTask 所构成的数据流图与 vDevice 的情况，通过一些简单的策略（譬如直接将 rTask 平铺上 vEU）就能够生成足够高效的调度计划。

由于硬件和编程模型的限制，GPU 运行时的分发器（dispatcher）和调度器（scheduler）并不对用户开放可编程接口，所以研究员们采用了持续线程（persistent thread）[2]的方式，巧妙地以一个相对小的开销将 vEU 与 SM 绑定起来。这样就可以将调度计划“静态”映射给硬件设备了。

RAMMER背后的思考：探索问题的本质

一篇好的系统论文，不仅可以优化性能，更要阐明一个问题。起初，研究员们只是想改善一个具体的神经网络推理时 GPU 利用率偏低的问题（出于对延迟的保障，很多场景下设置小的批次大小其实是标准做法），而除了优化算子实现以外，朴素的想法就是将多个算子一同交给 GPU 设备同时执行，但这并不是一个新的问题。CUDA 很早就引入了流的概念对其提供支持，GPU 社区之前也有一些效果不错的工作（concurrent kernel execution [3]、elastic kernel [4]等），那么在深度神经网络的场景下，为什么大家对这个问题的认知不足？

像上文提到的 MXNet 中有依赖引擎一样，TensorFlow 开发早期也有支持多个流的尝试。但是到后来都接近弃置了，主要原因可能有以下几个方面：

不同的 CUDA 流在运行时采用空间分片（spatial multiplexing）的方法来调度不同流队列（stream queue）上的算子，粒度更粗而彼此之间又极易产生相互干扰影响最终性能 [5]。

GPU 的 SM 在不断增加。现在 Ampere GA100 中有128个 SM，但几年前 Kepler GK180 中仅有15个 SM，所以在早期，无论是 GPU 社区还是 DNN 的框架开发，在现有的 GPU 编程模型下都已经形成了硬件对于算子间并行性并没有太多加速潜力的印象。

早期的神经网络结构比较简单，如 AlexNet 等本身在算子间并行性上也没有更多发挥的空间。但随着 AutoML 的出现，网络结构趋于复杂，此外也有ResNext [6]、ResNeSt [7] 等工作引入了新的神经网络设计模式，这个问题正

变得更重要。

只是将算子间并行性挖掘起来会是一个好的性能优化，但不足以成为一个好的系统工作。在之前的工作中，微软亚洲研究院的研究员们已经完成了初步的实现并且在一些模型上获得了较好的加速效果，但是当时还没有完全对问题进行清楚地定义，而且因为没有 NNFusion 代码库的支持，实验相对简陋，没有取得很好的反馈。

重新定义一个问题和定位一个工作并不是在用不同的写法来写“茴”字。之前的工作只是在做一个广义上的内核融合，并没有设立起 rTask 和 vEU 的抽象。而此次确定本质的问题在于原本系统中两层调度的差距以后，新的抽象很快探明了更大的优化空间：首先是将原本的通过成本模型来选择子图进行融合的问题，转变为了以更细粒度下的调度和资源分配问题。而得益于绝大部分情况下，神经网络计算的特征（DFG, 算子和张量）在编译时间是已知的，因此可以将调度的开销移交给编译器，这既提升了搜索的效率也简化了系统设计。

更重要的是，让算子间并行性与算子内并行性相互影响这个问题走进研究员们的视野。举个例子，如果对于同一个算子有两种内核实现，其中一个比另一个多消耗三倍的资源（如 CUDA Cores、Shared Memory 等），但是只取得两倍的加速，这在并行计算中是很常见的一个现象。而在此前单个算子独占整个硬件的情况下，毫无疑问会选择更快的实现。而研究员们的实验表明，在算子间和算子内两种并行性协同调度的情况下，选择资源“性价比”最高的实现而非“最快”往往是更优的选择。这其实挑战了之前许多生成高性能算子的工作如 AutoTVM [8] 等的一个基本假设，单个算子独占整个硬件表现出的计算性能是否真的是性能调优的金标准？显然，子图替换TASO [9] 加上高性能内核实现 TVM 两个“最优”相结合，并没有带来真的最优。

研究员们基于新的抽象，仅尝试了简单的策略就在一些场景下获得了超过现有 SOTA 的性能。在此，欢迎大家基于这个抽象进一步尝试更多调度策略，来探索对于一个数据流图（或者其子图）搜索算子间和算子内并行性相互影响下的更高性能的整体实现。

NNFusion 现已在 GitHub 开源：https://github.com/microsoft/nnfusion。目前已经发布了0.1 版本, 该版本支持 TensorFlow 和 ONNX 在内的主流模型格式以及 CUDA GPU 等设备，并提供了丰富的性能优化策略，支持端到端的模型到源代码的 AOT 编译来消除运行时的开销，同时还消除了对第三方库或框架的依赖。如果你有更深入的研发需求，可以直接修改 NNFusion 生成的代码来进行模型的定制化优化。

欢迎前往体验 NNFusion，也期待你可以在 NNFusion 中贡献真知灼见，一起“压榨”加速器的性能！

参考文献

[1]Tensor Comprehensions https://arxiv.org/abs/1802.04730
[2] Persistent Thread https://ieeexplore.ieee.org/document/6339596
[3] Concurrent Kernel Execution https://ieeexplore.ieee.org/document/5999803
[4] Elastic Kernel http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.649.8875&rep=rep1&type=pdf
[5] Dynamic Space-Time Scheduling for GPU Inferencehttps://arxiv.org/pdf/1901.00041.pdf
[6] ResNext https://arxiv.org/abs/.054311611
[7] ResNeSt https://arxiv.org/abs/2004.08955
[8] AutoTVM https://arxiv.org/pdf/1805.08166.pdf
[9] TASO https://github.com/jiazhihao/TASO