CUDA编程方法论之性能优化思路

模拟物理和数值计算做了十多年，从最开始入门的C/C++，到MATLAB，到Python，再到CUDA C，语言学了挺多种。用过许多数值计算库，也手写过许多算法，元胞自动机，蒙特卡洛模拟，数值最优化，矩阵计算，有限元分析，图像处理，图像重建等等，算是也涉猎过不少应用。做了这么多年性能优化，感想很多。算法，或者说算法的某一个程序实现，在能完成它必需的功能以外，最重要的要求就是能在有限的资源支持下，达到足够好的性能。性能优化是程序设计的重要内容，它不仅要求通晓算法功能，编程技能熟练，对环境判断还要准确。一般来讲，具体问题的具体优化方法容易获得，但解决思路和优化方向却往往比较欠缺。具体方法替代性强，有错误好修正，但指导思想错误却会让人把力气使在错误的地方，甚至可能会导致整个实现建立在错误的框架上，积重难返，之后更是事倍而功半。所以呢，还是有必要研究下看起来有些形而上的方法论。

当然，性能优化是大课题，不同的需求，不同的算法，不同的环境，都会导致完全不同的处理方式。具体落实到实现上，不同的语言，不同的硬件，也会有各式各样随机应变的招法。所谓兵无常势，水无常形，能因势利导，方能克敌制胜。就我个人来讲，方向做过不少，但接触面也很有限，经验也很片面。不过呢，不交流也不知道自己是不是片面，所以权当记录下我个人的理解，抛砖引玉，供有兴趣的人参考。另外也希望能以文会友，共同交流进步。因为本人最近几年的性能担当都是用CUDA实现，所以就以CUDA为例，谈一谈自己的理解和感悟。本文不是入门科普文，不系统，也难说客观，全凭感觉。当然，完全形而上也不行，多少会穿插一些具体例子，不然言之无物，显得空洞。

我把对算法实现的性能优化分为几个阶段：

阶段零：合理选择需要性能优化的部分。不要花力气去优化无足轻重的部分，找到性能瓶颈，然后再开始。有人说选择比努力重要，用在这个地方再合适不过了。道理简单，做起来也不算难，无需多讲。

阶段一：对输入需求进行限定和整理。通用性与高性能一般是矛盾的，这

是在挑选算法和实现之前就必须做出的抉择。比如排序，输入数据范围，分布等就会影响算法的选择。再比如，数值计算对精度都有一定容忍度，CUDA里的浮点数精度有单、双和半精度等，峰值性能和内存占用上都差几倍。泛型和模板可以部分解决类型的通用编程问题，但从性能上讲，这个问题几乎无法回避。此外，很多问题存在理论上唯一的最优解，但对于大多数的实际问题，解的最优性并非绝对必要的，能快速得到的准最优解或近似解往往更实用。求最优解和求近似解的算法可能大相径庭，甚至算法框架都不兼容，所以这部分工作一定要在前期就做好。

对需求的限定会影响这部分实现的通用性，不代表通用性就要被放弃。相反，是否能够融入一个通用性强的框架有时候会决定一个实现的生死。通过添加更高一层的外部接口，再加上前期筛选，可以把满足这部分需求的任务导入这个分支，而把其他任务导入另一个通用但性能更差的分支。然后，把最常用分支做快，把不常用的做对，在工作量和总体性能提升之间选取一个合理的平衡，也是通用性和性能之间的平衡。

举两个简单的例子：

例一：许多数学函数求值比如sin(x)，exp(x)等都有Range reduction的操作，为的就是把输入范围缩减到能准确计算的范围，而如果你确认x已经在这个范围内，range reduction就可以省了。如果在精度上可以还容忍一定的损失，则用简单的多项式近似就可以得到不错的精度。注意：一般不用Taylor展开，而是根据取值范围定系数，可以用Minimax optimization甚至最小二乘来求出所需的多项式系数。当然，CUDA里有intrinsic函数__sinf，__expf可以在牺牲一些精度的情况下不做range reduction，但是仍然要调用SFU单元的MUFU指令，延迟是FFMA指令的几倍，因而也要具体根据需要选择。如果是没有相应内置函数的函数求值，则自定义的多项式近似往往具有较好的效果。

例二：整数的除法和取余在CUDA里是比较耗时的操作，通常需要十多条指令。但如果被除数是个编译期常数，则除法和取余都可以用两三条指令完成。也就是说，如果编译Kernel的时候就写死这个被除整数，除法和取余的开销就会很小。通过模板参数还可以支持多个不同的被除数。当然，这些被除数都必须在调用Kernel时在模板参赛中显式指定，要求当前Kernel所有线程的被除数都必须一致。另一个思路是在运行期判断是否是几个预定常数之一（例如用switch case），然后跳转到对应的分

支。这就只要求Warp内所有线程使用相同的被除数，这样才能跳转到同一个分支。与此同时，这几个预定常数在所有可能的被除数中应该占有较大比例，否则这些判断和跳转（以及可能的warp divergence）反而会增加绝大多数非预定数的开销。此外预定常数的个数对性能也有一定影响，所以具体用什么方式实现，取决于输入的分布情况。

需要强调的是，这些处理的前提是首先要能意识到一些特殊情况可以被加速，其次是能判断这些加速的成本和收益，以及怎样平衡。这些都是实打实的具体知识点，只能靠平时的积累和摸索，并没有什么捷径。

阶段二：精简算法复杂度和计算量。简单说，就是能先算好的就先算好，能少算点就少算点。单从数学角度讲，常用算法的渐近复杂度一般已研究多年，想更进一步并不容易。但是，如果能够灵活运用输入中那些不"通用"的部分，以及对于输出最优性的容忍度，就可以获得比一般实现更大的改进空间。这部分其实是对输入需求进行限定的一个副产品。而从具体程序实现角度讲，有许多公共的计算过程是可以合并的（比如编译优化中常用的公共子表达式消除CSE，把公共变量计算从循环里提到循环外的LICM），这也是减少实际计算量的重要方式。另外，并行算法与串行算法相比，各个计算线程之间可能有很多计算路径是完全一致的，但碍于同步和通信成本比较高，不可避免的会存在一些冗余计算。实现时要尽量去除冗余计算，减少计算资源浪费。常用方法是提取公共计算路径，把结果暂存在可共同访问的区域，用仿存开销代替计算开销。如果是线程内的提取公共表达式，大多数成熟编译器都有类似功能，但把线程间的公共计算提出来，当前还只能自己动手。

阶段三：根据运行环境的性能模型预估性能瓶颈，并做出相应调整。输入限定和精简计算会让你算得少，不浪费，但不代表就一定性能好。性能是实打实运行出来的，对运行环境其实非常敏感。同样的实现和输入，在不同硬件上的性能往往不一样。一部分是因为硬件峰值有高有低，这一般只能用钱解决。另一部分是实现是否能把硬件效能发挥好。即使两块卡有几乎相同的峰值性能，但是架构不一样，就可能存在一些隐藏的瓶颈，性能也可能差上百分之几十，甚至几倍。根据架构优化实现是一件费时费力的事情，不可避免会涉及到架构的许多具体细节，以及各自对算法性能的影响。如果只想了解下大方向，从GPU和CPU架构的区别来看，指导思想还是会简单很多。这里简单展开聊聊我的理解。

CPU的性能模式主要面向的是单个任务。假如任务分为N个阶段执行，CPU会尽量把每个阶段的延迟做到最小。这不仅要求该阶段本身做得够快，还希望轮到该阶段执行时，它所有的准备工作都已完成，不必等待。能由其他单元先算的都先算好备着，即使没用上，白算也在所不惜。所以CPU才有几十级的Pipeline，支持很多复杂的执行路径，比如分支预测，投机执行，乱序执行等等。可以说CPU的性能模式在于最小化单个任务从开始到结束的总延迟。这与CPU线程切换成本比较高有很大关系，毕竟每次切换都需要对上下文进行保存和读取。CPU的ALU占比比较小，很多单元都是在为ALU做准备工作。当然，现代CPU其实也有很强的任务并行能力，多核+超线程现在都很常见，不再是单纯为一个线程服务了。但从设计思想上，最小化当前线程延迟仍然是主要目标。

GPU的性能模式面向的是多个任务。最好多到能保证所有功能单元都不闲着。GPU的ALU占比与CPU相比要高很多，因而为ALU服务的其他单元就不如CPU功能丰富。但GPU有一个很大的优势，就是通用寄存器多。通过把上下文完整的驻留在寄存器里，就可以几乎零成本的切换计算线程。GPU的ALU虽然往往延迟更高，但它可以不断接收新的任务。这样就可以通过在不同任务间切换，来保证ALU一直都有事可做。按照单个任务的时间完成点来看，每个阶段延迟都很长，但只要是ALU一直能得到新任务，始终满负荷运行，它就一定运行在峰值性能模式下。从所有任务的时间消耗来看，相当于就是每个任务都几乎做到最低延迟。

简单的说，CPU的性能体现在低延迟（latency），GPU的性能体现在高吞吐（throughput）。GPU就是删减了CPU中为ALU做各种准备、降低ALU延迟的部分，而大量增加ALU和通用寄存器数目，通过并行任务间的切换（当前颗粒度是warp）来隐藏任务的延迟。CPU的性能优化关键在于不能产生Pipeline断流，当前任务不可以停下等待，必须紧凑的往下推进，这样才能达到峰值性能。而GPU对单个任务的停滞并不关心，只要功能单元始终有新任务输入，那就可以达到峰值性能。

打个不太严谨的比方：某个饭馆只有一个厨师，切菜炒菜摆盘都必须他动手。其他人都在为他洗菜、洗锅、递盘子等。这样一次做一个菜，从点菜到上菜的时间最短。要点在于每次切菜前，菜必须洗好，炒菜前，锅必须洗好，摆盘前，盘子必须准备好。只要保证厨师不闲着，就是峰值性能，其他人只需要保证准备工作做好就行，可以允许有时候闲着。这是CPU模式。另一个饭馆，有几套人马，每个人专门负责一件事，洗菜、切菜、炒

菜、摆盘等等。你只要保证点足够多的菜，让每个人每时每刻都不闲着，那就是峰值性能。至于具体某个菜从点菜到上菜过了多久，中间是不是有很长时间没人过问，并不重要。最后所有的菜一起上的时候，其实没人关心这个菜是不是一气呵成，只要足够短的时间内能把所有菜上齐就行。这是GPU模式。

GPU的运行模式决定了它与CPU不一样的优化思路。为了能有效隐藏延迟，GPU的架构也在不断进化，现在至少都支持线程并行（Thread Level Parallelism, TLP）和指令并行（Instruction Level Parallelism, ILP）两种方式。TLP就是多个线程同时运行，这是GPU基本功能。ILP则稍微有点复杂。一般说来，如果没有指令并行，每个指令的延迟会决定该线程多久后可以被重新选中执行。这期间的空缺必须通过执行其他warp来填充（也就是TLP）。指令延迟越短，同一个SM运行的warp越多，延迟就越容易被隐藏。这也是Occupancy对性能的影响所在。CUDA现在支持线程内指令并行，也就是说同一个线程的上一个指令没完成也可以执行下一个指令，除非是下一个指令依赖之前某个未完成指令的输出。每种指令的延迟有长有短，以Turing为例，短的FFMA、LOP3等只有4个周期，MUFU大约十几，DP单元指令一般几十，Global Memory的load在cache miss时可能几百。但是，只要是延迟能被充分隐藏（也就是功能单元一直在忙），那总体算起来就相当于几乎没有延迟，这些指令的实际开销，也就是几乎一样的（指令吞吐会不一样）。当然，延迟越长的指令越难隐藏，线程指令依赖度越高越无法指令并行，这也是优化中需要充分考虑的。

这里需要点名一个实战中常遇到的误区。很多人认为，由于global memory的延迟很高，一些计算量不是特别大的量，即使会被很多线程反复用到，也宁愿每次重新计算，而不是从global memory中读取。其实，如果是内存读写单元不是很忙，又有足够的并行任务，很少的global memory的访问延迟是很容易被隐藏的。CUDA编译器一般会把global load指令LDG尽量提前，这样可以充分利用后面的无依赖指令，再加上切换其他warp来隐藏这部分延迟。这样LDG的净效果几乎就相当于开销最小的FFMA。如果变量之间有较强的locality，那通过cache还可以大大减小这个延迟，使其更容易隐藏。如果LD/ST单元很忙，导致带宽紧缺，或是occupancy很小导致延迟很难被隐藏，那就需要仔细分析。以我的经验，几条指令能算出来的量可以不用存，需要更大开销的量很多还是存下来性能更好。当然，如果需要存的变量特别多导致

内存占用紧张，或是由于依赖性需要改变整体算法流程的，另当别论。

了解运行模型不仅能提供性能优化的手段，同时也可以用来判断性能瓶颈。算法定下来后可以分析带宽占用和SM使用率的大致关系，与实际profiler结果比较，就可以知道是带宽还是ALU使用上存在瓶颈。比方说设备峰值是4TFlops，400G/s。而一个算法处理1GB global memory里的数据要2G Flop，实测带宽是峰值50%=200G/s，浮点性能是峰值10%=400GFlops，说明带宽是瓶颈。另外，带宽虽是瓶颈，但离100%较远，说明有些地方的延迟隐藏得不够好，导致单元有时处于闲置状态。如果是带宽占用25%=100G/s，浮点性能50%=2000GFlops，计算比带宽与算法相比高很多，说明浮点计算有较大浪费，应该想办法精简计算量。当然，有些数学上Flop的计算与具体指令不对应，需要自行判别，典型的是浮点除法有时也算一个Flop，但实际会用到很多指令。如果是带宽80%，320G/s，浮点性能10% 400GFlops，那说明有些数据可能被多次获取而且没能cache，或者是内存访问合并较差，带宽浪费很多。这些判断要反馈回需求限定和精简计算量的步骤，进行相应调整。比如带宽瓶颈就把一些简单的计算现算，计算瓶颈就把一些能存下来的存下来。当然，这只是一个方面，具体性能指标和反馈方式还有很多，必须全盘考虑才能得到比较准确的分析。一个简单的判断是，如果只有一个指标接近峰值，其他都离峰值很远，那优化空间一般很大。如果大多数性能指标都能到峰值的50%~80%，那一般空间就比较有限了，想再优化就需要非常全面具体的微调。如果所有指标都离峰值比较远，说明存在严重的延迟隐藏不足，或是运行参数不合理，需要重新安排程序结构。

完美的实现是不存在的，总有可以优化的地方。但是，如果一个实现把输入能做的简化也考虑到了，计算上也没什么的浪费，实测也没什么特别严重的瓶颈，那就基本上可以满意了。能做到极致的实现确实是可遇不可求，有时候宁愿把这个时间用来推广和泛化当前实现，让它可及性更好。毕竟过了几年，程序设计的思路和硬件架构也许都大不一样了，不能指望它能流传给下一代。