CUDA并行架构
期末调研报告

计算机科学与技术
1511186
梁宸

February 25, 2020

1 摘要

本次调研目标选择了近几年十分流行的CUDA并行架构, 在硬件基础、编程模型等层面上进行了学习和了解, 使用它编写了测试程序, 和其他并行框架进行比较.

2 CUDA简介

CUDA(Compute Unified Device Architecture)是NVIDIA于2006年推出的并行架构, CUDA的概念包含一个编程模型和一个计算平台, 它提供了一个CUDA指令集架构和并行引擎, 以进行GPU上的通用计算. 由于其硬件基础广泛[1], 编程相对简单, 性能高等特点, 它现在是最流行的并行计算框架之一, 尤其是在GPU平台上.

3 CUDA的背景和优势

在CUDA之前的计算框架基本是CPU计算, 因CPU需要面对更多样、复杂的任务, 处理的指令和数据之间有复杂的逻辑关系, 所以CPU的架构设计也十分复杂, 除计算单元外, 大部分都是缓存以及复杂的控制单元等; CUDA针对的是GPU计算, 而GPU设计的主要目的就是图形计算, 因此它的设计基本上就是一个庞大的计算阵列[1], 通常拥有远强于CPU的浮点运算能力, 这正是人工智能, 数据挖掘等领域中最主要的计算任务.

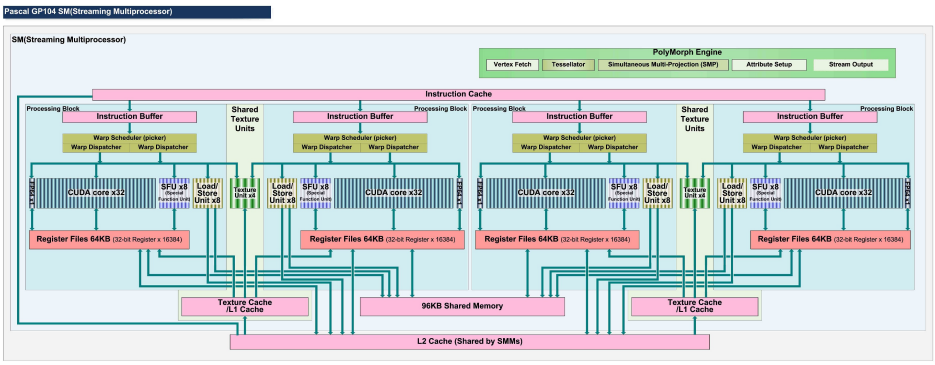
[1] 显卡卖得多

但在CUDA出现之前, 程序员只能只能通过驱动和OS底层提供的API, 来用GPU进行特定的计算, 比如图形渲染, 而没有像CPU的指令集那种通用计算接口. 而CUDA提供了这些, 使得GPU的强大浮点运算能力能够被利用[2].

4 GPU的硬件架构

为了更好了解CUDA中并行模型的设计和使用, 需要大体了解GPU的硬件架构中的并行控制和内存部分.

以Pascal架构为例, 它是NVIDIA目前最新的架构之一, 其最主要的部分是许多Stream Multiprocessor (SM), 它的地位有些像CPU中的core, 其结构简图如下:



其中有许多CUDA core ,也叫做Stream Processor (SP), 它们对应着GPU的物理线程. 图中可以看到每32个core分为一组, 称为一个warp, 它们共享一个调度器和指令缓存, 因此1个warp, 即32个线程, 是GPU调度和执行的最小单位, 这提醒我们最好在编程时也使用warp为单位设置线程, 否则可能会浪费调度资源.

SM每次调度会选择一个warp执行, warp中的线程执行相同的指令, 即SIMT (Single Instr Multiple Thread)模式, 它们以串行的方式执行来保证严格的同步. 但如果遇到控制流, 比如条件跳转, 那么会执行一个分支上的线程, 阻塞另一条, 之后再反过来, 这十分影响性能. 于是在编程的时候要尽量分配同一个warp中的线程执行相同的代码, 比如让它们进入相同的分支, 或者直接避免条件跳转, 尽量不要在线程中执行过大的循环, 而是增加线程数.

当一个warp执行到内存指令时, 它会被阻塞, SM调度执行其他warp以等待数据传输. GPU的这种调度以隐藏内存延迟的方式使得它不像CPU那样依赖缓存.

图的下半部分描述了存储结构, 每个warp共享一套32位寄存器, 每两个warp共享一个L1缓存, 整个MP共享一个共享内存.

[2]CUDA提供的是私有标准, 理论上只支持NVIDIA部分架构的GPU

更宏观地, 一个GPU中有许多SM, 所有SM共享一个L2缓存, 除此之外就是一块全局的内存, 也就是常说的显存. 通常来讲显存的读写要比共享内存慢几百倍. GPU通过PCIE3.0外部连接.

5 CUDA编程模型

CUDA编程模型相对比较简单, 其中最重要的部分就是有关线程的概念. 在CUDA逻辑模型中, 线程被组织为block和grid, 一个block中有多个线程, 而一个grid中有多个block, 都通过最多3维的id来索引. 对应到硬件上, 一个block中的线程一定会被分配到同一个SM上, 它们享有一块高速的共享内存, CUDA因此提供了block内线程同步的函数; 而不同的block不保证分配到同一个SM上, 因此CUDA不提供不同block之间同步的方式[3].这样在CUDA提供了两种粒度的并行, block是粗粒度的并行, block之间没有相互依赖; block内的线程是细粒度的并行, 可以实现同步等通信. 这样的设计使得CUDA在不同数量的SM环境下都可以运行, 具有良好的可伸展性.

CUDA中CPU对GPU的控制被组织为stream, 一个stream包含许多CPU发给GPU要执行的操作, 同一个stream中的操作会被串行执行. 常用的操作有执行核函数, 它们是会被CPU发送到GPU上执行的函数, 通过核函数CUDA提供了控制GPU进行通用计算的接口. 其他常用的操作还有内存拷贝等. 通过stream, 我们可以实现更高层次的任务级别的并行. 而对GPU的stream发出操作后, CPU本身是不一定需要阻塞的, 因此也可以实现CPU和GPU之间的并行.

在上述硬件架构中介绍过, GPU中每个线程要尽量避免循环等控制流. 因此CUDA编程的另一大特点是线程函数尽可能简单, 而以线程数量和id来代替循环等控制流. 此外CUDA的语法能指定变量的存储位置, 比如共享内存, 全局内存等等, 因此编程时对内存的掌控权更大, 而不是像CPU编程时的透明的储存器分层机制.

6 程序设计和实现

为了和之前学习的并行工具做对比, 这里依然选择了矩阵的高斯消元法来进行实践, 并且随着对CUDA学习的深入, 在算法设计和实现上也不断修改. 因为数据依赖性, 算法依然按照轮执行, 每一轮用当前最顶部的行去消去下方的行.

6.1 版本一

最初, 我仍按照CPU编程的思路设计算法. 在第k轮中, 每个线程被静态地分配到一列j上, 它先进行除法阶段, 即a[k][j] = a[k][j] / a[k][k], 而其中a[k][k]是与线程无关的, 于是先让一个线程将它拷贝到高速的共享内存上, 之后其他线程都可以通过共享内存来访问它; 接下来该线程对该列所有元素进行消除, 即对于i ¿ k + 1, 执行a[i][j] = a[i][j] - a[i][k] * a[k][j], 其中a[k][j]是循环不变量, 因此可以在除法

[3]可以使用全局的显存和原子操作来同步, 不过很慢而且不推荐.

阶段就先将它也拷贝到共享内存上, 接下来在线程内部的循环中都通过共享内存访问.

整个算法在一个核函数调用中完成, 也就是说GPU一次性地完成所有工作. 在每一轮结束后, 需要所有的线程进行同步, 但CUDA只提供同一block内的线程同步, 因此这样只能将所有线程分配到一个block中. 而由在的介绍中提到, 一个block只会分配到一个SM上, 也就是说程序只利用到了一个SM, GPU利用率很低. 并且一个block中线程数目上限并不大, 因此每个线程要做多个列, 这样在核函数中需要三层循环, 最外层循环迭代轮, 第二层循环迭代列j, 最内层循环迭代行i, 之前提到过warp内部是串行来严格同步执行, 这样较为复杂和庞大的控制流会严重影响性能. 事实上随后的测试显示, 这种实现性能甚至低于单线程的AVX版本, 在2000*2000规模的矩阵输入下, 运行时间已经难以承受.

6.2 版本二

在了解了GPU的基础硬件架构后, 发现应将SM尽可能的全部利用, 并且减少核函数内循环层数, 以增加并行度. 因此将核函数内最外层循环提出到CPU函数中, 这样每次GPU只会执行一轮算法, 同时也做到了线程同步. 并改变线程的组织形式, 分为多个block, 每个block对应矩阵中连续的几列, block中所有线程的工作范围就是这几列.

第k轮的除法阶段中, 先由一个线程将a[k][k]放入共享内存中. 再按顺序给每个线程分配一个顶部元素a[k][j], 执行a[k][j] = a[k][j] / a[k][k], 随后将a[k][i]放入共享内存; 在消除阶段, 每个线程按顺序被分配到行i上, 对该行的所有元素a[i][j], 执行a[i][j] = a[i][j] - a[k][j] * a[i][k]. 其中a[k][j]在除法阶段中被放到了块的共享内存中, 而注意到a[i][k]对同一行是不变量, 所以也可以先放到高速的共享内存中[4]

这样的设计和实现使得版本二相对一比, 利用到了更多的SM, 并且降低了线程函数的复杂度, 大幅提高了程序的并行度和性能.

6.3 版本三

在版本二的算法中, 核函数内部依然存在两层循环, 并且每个线程要参与除法和消除两个阶段, 仍然过于复杂. 在学习了GPU架构中的warp调度等知识后, 认识到应使用grid和block组织线程索引, 代替循环, 而不用过于担心block数量过多, 因为GPU的硬件提供了高效的线程调度. 于是设计block的尺寸为固定参数b*b的正方形, 在第k轮中, 设置grid尺寸为g*g, g为刚好能使g*g个block覆盖矩阵的数目. 这样每个线程会对应一个矩阵元素a[i][j], 其中i和j可以根据其所在grid和block的索引计算出. 省略除法阶段, 核函数中只执行a[i][j] = a[i][j] - a[k][j] * a[i][k] / a[k][k].

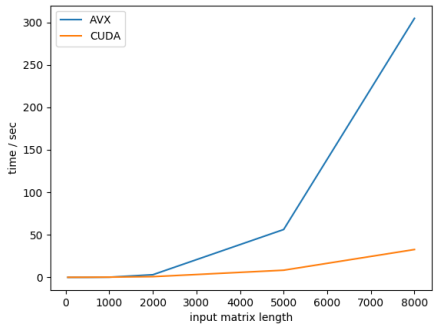
[4]虽然nvcc很有可能会将它分配到寄存器上

上式中可以看出, 每个block内, 不同的线程会重复访问相同的a[k][k], a[i][k]和a[k][j], 而这些元素的大小和block的长宽之和成正比, 因此设置block为正方形会增大这些元素进入共享内存或寄存器的概率. 再考虑到本机GPU中CUDA core的数目和warp的尺寸, 因此设置block尺寸为32*32.

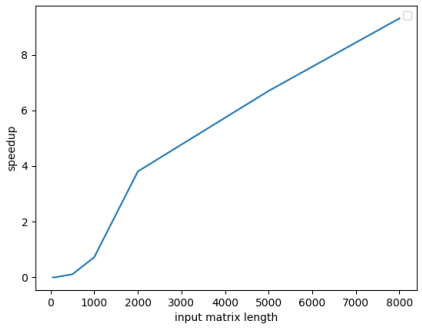
在算法执行结束后, 需要将输入矩阵从显存拷贝回内存. 此内存操作的耗时并不小, 在测试2000*2000的输入下, nsight profiling工具显示它接近算法执行时间的五分之一. 考虑在算法的每轮结束后, 最顶部的行结果已确定, 因此可以直接开始拷贝回内存. 因此使用stream来进行异步的计算和数据传输, 一个stream进行计算, 即核函数, 另一个stream负责传输结果, 二者用CUDA的API进行同步. 此外在核函数中, 省略了每行的除法工作, 这里可以每次传输回一行的结果后, 由CPU负责该行的除法, 这样就做到了CPU与GPU的并行.

7 测试结果

测试的环境为Windows 10 64bit, Intel Core I7, NVIDIA GTX1070. 设置不同的输入矩阵规模, 出于执行时间考量, 选用AVX版本的程序做串行的对比, 测试AVX版本与CUDA版本三的程序执行时间, 结果如图:

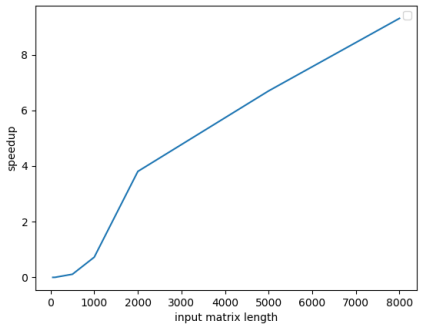


可以看到CUDA版本性能较高, 且其执行时间增长低于串行最优算法O(N^3)ﬂ$B.\Lambda v\emptyset$:



可以看到即使在最大输入8000*8000的矩阵下, 加速比接近10, 且依然是增长趋势. 此前作业中最好的结果是PThread实现的多线程AVX版本, 它在5000*5000的输入下加速比稍高于3, 而此版本在相同的输入规模下加速比为7左右, 可见CUDA的性能之高.

利用整合了Nsight工具的Visual Studio, 可以对CUDA程序进行profiling, 在8000*8000的最大输入, 考察异步计算和数据传输带来的优化效果:



	Source	Destination	Queue Time (µs)	Submit Time (µs)	Start Time (µs)	Duration (µs)	Size (bytes)	Rate (MB/s)
4	Device	Host Unpinned			282,996.781	4.032	31996	7567.9
5	Device	Host Unpinned			294,447.139	4.000	31992	7627.5
6	Device	Host Unpinned			305,917.529	4.064	31988	7506.4
7	Device	Host Unpinned			317,438.064	3.040	31984	10033.7
8	Device	Host Unpinned			328,944.647	4.032	31980	7564.1
9	Device	Host Unpinned			340,875.200	4.064	31976	7503.6
10	Device	Host Unpinned			352,349.655	4.064	31972	7502.7
11	Device	Host Unpinned			364,411.506	4.064	31968	7501.7

上图是程序运行时调用内存拷贝以及核函数的情况, 由二者的起始时间可以证实计算与数据传输的并行, 而由传输速度估算, 如果不进行延迟隐藏, 那么在计算结束后需要大概3s的时间来传输结果, 大约为执行时间的十分之一.

8 总结

CUDA框架提供了使用NVIDIA GPU进行通用计算的途径, 其浮点运算能力强大, 硬件层面并行度高, 且并行开销小. 相较Pthread, OpenMP等工具, 个人认为模型抽象度更高. 与CPU上的并行设计不同, 它利用硬件的物理线程调度等优势, 要尽量避免复杂的控制逻辑, 这也使得CUDA程序通常十分简洁. 但也有一些缺点, 比如GPU上的环境与CPU-OS整体隔离性有点强, 更像黑盒, 调试更为困难.

References

https://blog.csdn.net/kkk584520/article/details/53814067

https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/4

https://www.cnblogs.com/tibetanmastiff/p/4620803.html

https://docs.nvidia.com/cuda/

https://blog.csdn.net/jiangbo1017/article/details/53940428

https://www.cnblogs.com/1024incn/p/4539754.html