**《并行计算》上机报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 姓名： | 王章瀚 | | 学号： | PB18111697 | 日期： | 2021/5/29 |
| 上机题目： | GPU并行编程实验 | | | | | |
| 实验环境：  CPU：8 块 Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz ;  内存: 32GB ;  GPU: GeForce RTX 3070 Laptop ;  操作系统：Windows 10 ;  软件平台：NVIDIA Driver 461.56, CUDA 11.2 ; | | | | | | |
| **一、算法设计与分析：**  题目一:  矩阵乘向量.给定二维数组 A 和一维数组 B, 要求矩阵 A 和向量 B 的内积.  不妨设 , 那么只需要按照矩阵乘法的定义:  即可求出目标结果.  对于布局于 CPU 的算法, 整个过程需要串行化, 其时间复杂度是  而对于布局于 GPU 的算法, 可以将每个元素的计算都并行起来, 计算的时间复杂度是  但对于GPU 运算而言, 需要有数据拷贝与传输的时延, 因此总时间应当是  题目二：  矩阵乘矩阵.给定二维数组 A 和二维数组 B, 要求矩阵 A 和矩阵 B 的内积.  不妨设 , 那么只需要按照矩阵乘法的定义:  即可求出目标结果.  对于布局于 CPU 的算法, 整个过程需要串行化, 其时间复杂度是  而对于布局于 GPU 的算法, 可以将每个元素的计算都并行起来, 计算的时间复杂度是  但对于GPU 运算而言, 需要有数据拷贝与传输的时延, 因此总时间应当是  但对于矩阵乘矩阵, 这种方法还能够有所改进:    通过对矩阵进行分块, 并且使用 GPU 的 shared memory, 我们能够大量减少访存开销, 从而达到更快的速度.  可惜对于个人计算机来说, 不容易测试不同传输速度的结果. 因此后面的测试主要基于调整矩阵大小来体现 GPU 性能优势.  **二、核心代码：**  题目一:  对于 GPU 运算, 其核心就是一个核函数. 通过用特殊方法调用这个函数, CUDA 将分配指定的块和线程去做这个核函数所描述的工作.  其中 blockIdx 和 threadIdx 能够给出块和线程的id. 我们的程序将可以根据这些 id 去分配工作, 从而达成单指令多数据流的高效编程.  对于矩阵乘向量, 为了计算结果向量中每个元素的值, 只需要按照前述算法, 做一个 K 次的循环来累加单元素乘积, 最终即会得到结果.  经历了这个 for 循环后, 再把值赋到结果向量的相应位置, 从而完成计算.    为了使得数据能在 CPU 和 GPU 之间通信流动, 编写 CUDA 代码的时候还需要有这样一些操作:    首先, 需要在 GPU 设备上分配内存, 这可以通过调用 cudaMalloc 来完成. 此后, 由于所要计算的数据原本是在 CPU 上的, 需要拷贝到 GPU 内存里, 这可以通过调用 cudaMemcpy 来完成. 这个拷贝函数要求指定复制的源设备和目标设备.  然后, 为了调用核函数, 我们需要指Block 和 Thread 的个数, 从而让 CUDA Runtime 能够根据这些参数去调度线程来完成工作. 在上述代码里, dimBlock 和 dimGrid 即是定义了相应块和线程的大小.  调用核函数的时候, 用一个 <<< >>> 的符号来指定Block大小与Grid 大小. 其他与普通函数的调用无异.  最后, 需要释放在 GPU 上的内存, 通过调用 cudaFree 即可完成.  题目二：  对于 GPU 运算, 其核心就是一个核函数. 通过用特殊方法调用这个函数, CUDA 将分配指定的块和线程去做这个核函数所描述的工作.  其中 blockIdx 和 threadIdx 能够给出块和线程的id. 我们的程序将可以根据这些 id 去分配工作, 从而达成单指令多数据流的高效编程.  对于矩阵乘矩阵, 为了计算结果矩阵中每个元素的值, 只需要按照前述算法, 做一个 K 次的循环来累加单元素乘积, 最终即会得到结果.  经历了这个 for 循环后, 再把值赋到结果矩阵的相应位置, 从而完成计算.    为了使得数据能在 CPU 和 GPU 之间通信流动, 编写 CUDA 代码的时候还需要有这样一些操作:    首先, 需要在 GPU 设备上分配内存, 这可以通过调用 cudaMalloc 来完成. 此后, 由于所要计算的数据原本是在 CPU 上的, 需要拷贝到 GPU 内存里, 这可以通过调用 cudaMemcpy 来完成. 这个拷贝函数要求指定复制的源设备和目标设备.  然后, 为了调用核函数, 我们需要指Block 和 Thread 的个数, 从而让 CUDA Runtime 能够根据这些参数去调度线程来完成工作. 在上述代码里, dimBlock 和 dimGrid 即是定义了相应块和线程的大小.  调用核函数的时候, 用一个 <<< >>> 的符号来指定Block大小与Grid 大小. 其他与普通函数的调用无异.  最后, 需要释放在 GPU 上的内存, 通过调用 cudaFree 即可完成.  而对于采用矩阵分块的算法, 并利用共享内存的方法而言, 其核心代码如下:    这里最大一层循环是大小为WIDTH/TILE\_WIDTH的循环, 该循环中计算一个矩阵乘一个矩阵的结果, 每个线程负责将块内数据拷贝进共享内存, 并通过一个简单的循环来累积结果的值.  **三、结果与分析：**  **这里主要通过考虑不同数据大小来探究 GPU 运算的性能究竟如何. 此外还会分析共享内存能带来多少性能提升.**  题目一:  对于矩阵乘向量, 我们可以测试一下它随数据量增大, 性能表现有何差异.  但对于数据量非常小的情况来说, 使用 GPU 毫无意义, 因此起始数据量比较大.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 矩阵大小 | 1024\*1024 | 2048\*2048 | 4096\*4096 | 8192\*8192 | 16384\*16384 | | CPU(us) | 2239 | 10986 | 44837 | 157256 | 557032 | | GPU(us) | 92318 | 95560 | 117725 | 209020 | 520497 | | 加速比 | 0.024253 | 0.114964 | 0.380862 | 0.752349 | 1.070193 |   可视化结果如下    可以看到, 对于矩阵乘向量这一非常简单的操作, GPU 需要在数据非常大的时候才能超越CPU的性能.  题目二：  对于矩阵乘矩阵, 我们考虑方阵情况, 测试不同数据大小的时候性能表现有何差异.  但对于数据量非常小的情况来说, 使用 GPU 毫无意义, 因此起始数据量比较大.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 矩阵大小 | 128\*128 | 256\*256 | 512\*512 | 1024\*1024 | 2048\*2048 | | CPU(us) | 4323 | 48267 | 389107 | 3599929 | 97400035 | | GPU(us) | 90087 | 97075 | 129704 | 378168 | 3635020 | | 加速比 | 0.047987 | 0.497213 | 2.999961 | 9.519391 | 26.794910 |   可视化结果如下    可以看到, 对于矩阵乘乘法这一操作, 由于其复杂度比较高, 用串行的算法就远不如 GPU 性能强大.  除此之外, 还可以试一下采用共享内存的方式会有怎样的加速比:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 矩阵大小 | 128\*128 | 256\*256 | 512\*512 | 1024\*1024 | 2048\*2048 | | CPU(us) | 5929 | 59769 | 459336 | 3665205 | 130733491 | | GPU用共享内存  (us) | 87976 | 92704 | 101089 | 111836 | 1892106 |   使用共享内存的结果中, 当矩阵大小为 2048\*2048的时候甚至可以达到60几倍. 这是常规硬件结构无法达到的性能表现.  与前面的情况对比如下:    **四、备注（\* 可选）：**  有可能影响结论的因素：  GPU 和CPU的匹配程度决定了加速比. 显然用一个很差的CPU和一个很好的GPU的时候, 如果达到非常大的加速比. | | | | | | |
| **总结：**  本实验完成了 CUDA 的矩阵乘法, 令我对CUDA的编程有了一个基础认识. 同时也认识到了GPU这种专用硬件的强大. | | | | | | |
| 附录（源代码） | | 算法源代码（C/C++/JAVA描述）  **矩阵乘矩阵和矩阵乘向量**:              **采用共享内存的代码如下**: | | | | |