# 异构计算试验报告

---实验2：基于CUDA的GPU实现矩阵的幂

第一部分：实验环境

OS：Windows 10

CPU：intel(R) Core(TM) i7-10510U CPU@ 1.80Ghz

GPU: NVIDIA GeForce MX250

编译器： cl :v19.29.30133

nvcc: Cuda compilationn tools, release 11.4, V11.4.120

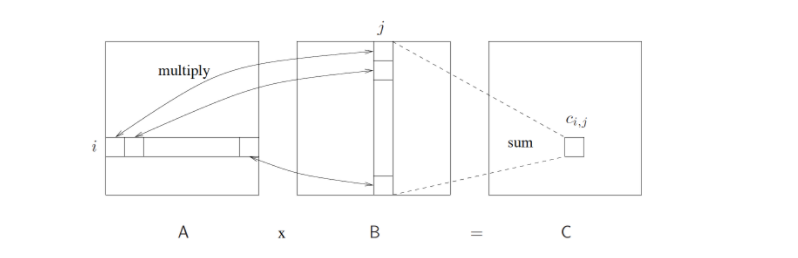
gcc v11

第二部分：实验内容

对于一个𝑚×𝑚的方阵A=[𝑎\*0]，计算𝐴的𝑛次幂。

1. 暴力算法 𝑛个矩阵相乘
2. .高效算法 利用矩阵乘法的结合律

第三部分：实验原理



第四部分：程序流程图

初始化矩阵&定义必要变量

|

确定各个线程的运算量

|

开始计算

|

等待计算完成，归约结果

第五部分：实验结果及分析

定义：N 矩阵的单边大小，总大小为 N\*N

定义：TIMES 计算矩阵的TIMES次幂

Ⅰ．暴力方法

1. 1CPU部分

首先分析在CPU上的执行效率，在CPU上写出循环矩阵乘法，使用gcc编译，单核运行。

指定 N = 512 TIEMS = 32 初始=1.01



结果 结果是一个85位长的十进制数，用时 ：24205ms 内存占用：6MB

在此基础上，考虑使用CPU并行计算优化，考虑使用8个线程

分析矩阵乘法的运算过程我们可以发现，结果矩阵中的每个元素都是独立的，计算时对于右值只有读操作，没有写操作。因此每一个线程计算[0,N/thread\_number]的行即可。

一次乘法计算的过程为（记最初的矩阵为A）

TEMP = RESULT

RESULT = TEMP \* A

这里的计算思路有两种，第一个是在每一次循环都创建8个线程，然后等待join.

创建线程 -> 并行计算 -> join -> 创建线程 -> 并行计算 -> join -> …

这种方法的坏处在于创建线程的次数太多 32次循环 要创建32\*8 = 256个线程

第二个方法是加入barrier，每次循环的结束都等待全部线程并更新RESULT

创建线程 -> 并行计算 -> 等待barrier1 -> 更新RESULT -> 等待barrier2 -> 并行计算 ….

总之，这次使用第二种方法。

 多线程可以显著的减少计算的时间，加速比为 3.55

用时：6820ms 内存占用：6MB

综上所述，CPU运行此程序的最大性能为 6820ms

1.2 GPU部分

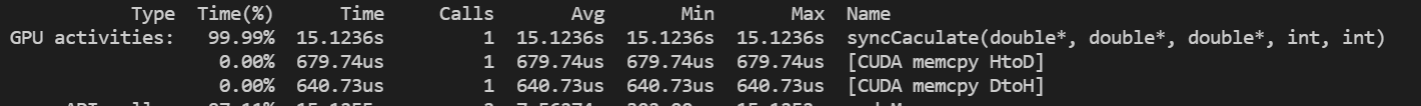
这部分使用的技术与1.1中的barrier十分相似，使用\_syncthread（）可以同步一个块内的所有线程。

但是实际执行的时候，出现了非常严重而且难以解决的问题，问题的描述如下

1. 在本设备的GPU端执行的程序结果不正确（大约有万分之一的误差）
2. 经过研究发现从第20次矩阵乘法之后结果开始不正确，在此之前都正确
3. 在服务器端也是如此
4. 如果使用了-G选项进行调试，会发现一切正常。
5. 因为-G选项会禁用所有优化，这说明了异常是由于优化产生的

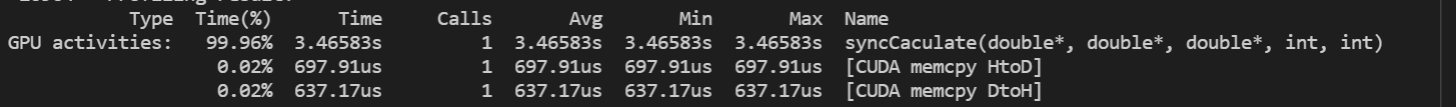
这个问题暂时没能解决，不过对性能分析应该不会产生太大的影响

方案一：一个block，32个线程，一个线程负责16行



用时：15.12s

方案二:一个block，512线程，一个线程负责1行

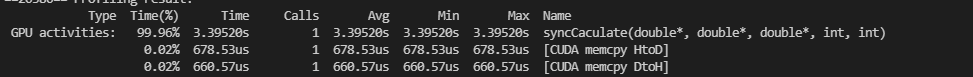


用时：3.46s，但这应该不是整个GPU的最佳效率，因为只有一个SM在活动

方案三：三个block，512线程，也就是16个warp。实际需要3\*6=18个warp，其中两个warp执行空指令。每个block 192线程



用时：2.20s。 想要继续提速就需要把矩阵放入shareMemory 中了，但是由于共享内存的大小只有48K，放不下整个矩阵，所以采用一种替代的方法。由于矩阵的每个元素都是一样的，所以只要把一个元素放入共享内存中，所有线程都访问这一个内存就行了。



用时：3.40s，这里只有一个SM在活动，所以应该与上上个程序进行对比。并没有显示出明显的加速。具体的原因就不清楚了。

综上所述，现在可以达到的性能指标排名为

单核CPU << 一个WARP << 八核并行 << 一个SM512线程并行 < 三个SM512线程并行

收到矩阵行数的限制，一共只能开启512个线程，如果需要使用三个SM的话，一个SM上只有192个线程，可能会降低GPU的活动线程数量，弱化延迟隐藏的效果。

2.高效算法

2.1 理论分析

如果要计算矩阵A的N次幂，使用暴力方法一共需要计算 N – 1次矩阵乘法。

使用高效算法时，先选取一个整数 k ，使得 2^(k-1) < N < 2^(k+1)，将真个乘法分为2^k + (N – 2^k).在计算2^k部分时，需要计算k次。对于剩下的 （N-2^k）次乘法也进行上述操作。 整体计算复杂度由O(N)降低为O(logN).

实现方法：

构造结构体list，含有一个int a字段和指向矩阵的指针ptr。初始化a =0和ptr =NULL

1. 给定计算次数N
2. 把N转化为二进制
3. 在对应二进制为1的位置，置对应的list[i].a = 1
4. 设置循环32次,temp = 矩阵A
5. A = A \* A
6. 如果list[i].a = 1 ,记录此时list[i].ptr = A
7. 转到4
8. 将list里所有a = 1对应ptr的矩阵乘起来，即为结果

由上述的方法，进行的矩阵乘法的最大次数为

floor(log(2,N)) + C

其中C = 将N转换为二进制后1的个数 - 1

例如进行32次乘法，一共需要 5 + 0 =5 次运算

进行10次乘法，需要 3 + 1 = 4 次运算

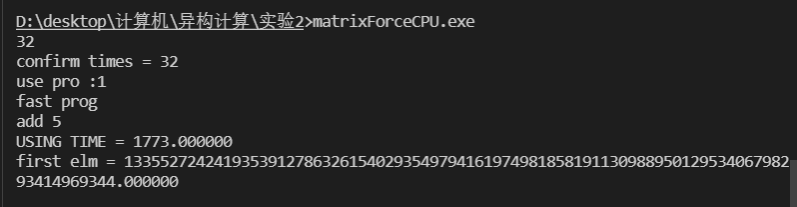
在N很大的情况下，这种方法可以极大的提升效率

测试：CPU单核计算 N =32



用时 3714ms。实际只进行了5次运算

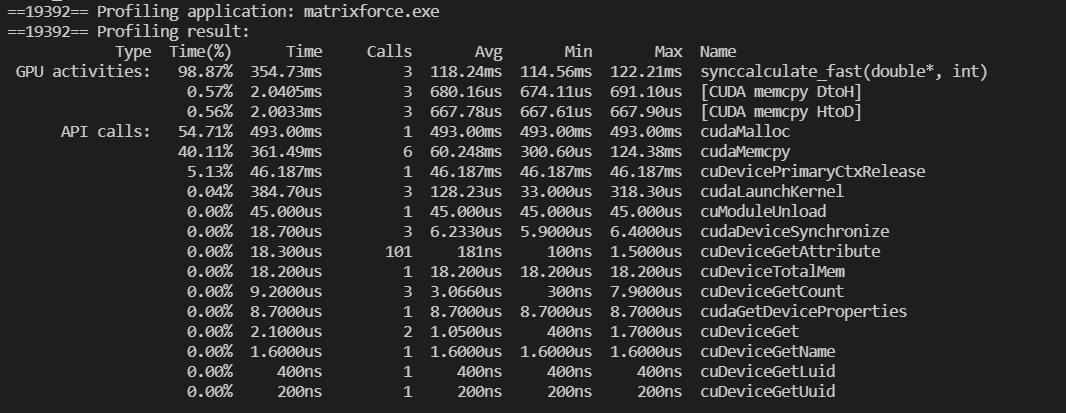
CPU并行八核计算



在八核并行的情况下，可以看到用时降到了 1773 ms

PS：写这个程序的时候由于内存更新不同步导致的bug，花了好几天，换了好几个平台才找出来…。理论上这个程序使用pthread\_wait\_cond 会更好一些，但对 pthread的cond不是很了解，最终还是使用两个barrier实现的。在大量运算的情况下，barrier的开销应该可以不计。

GPU计算



在GPU计算的环境下，用时进一步缩短，一共启动了一个block和512个线程，每个线程计算矩阵中的一行。总用时 360 ms

高效算法对比暴力算法

第六部分:实验总结  
 高效的算法对于计算性能的提升是非常巨大的，在考虑加速时，应该先考虑改进算法。在确认算法已经无法继续改进的情况下，再考虑并行计算。算法和并行结合，尽可能做到最高的效率。