**电 子 科 技 大 学**

**实 验 报 告**

|  |  |
| --- | --- |
| **学生姓名：** | **学 号：** |
| **一、实验室名称：**主楼A2-412 | |
| **二、实验项目名称：**N-Body问题并行程序设计及性能优化 | |
| **三、实验原理：**  根据万有引力定律，物体i与j之间的万有引力：    其中为引力常数，为各自的质量，是从物体到物体的距离矢量，是距离的平方，是个标量。最后乘以，则得到了力的矢量。对于个物体，若已知每个物体的质量、初始位置和速度（），那么作用在物体的合力为：    由于两个物体距离为0时，那么它们之间的力将变为无穷大，上式中因而将的限制加上，然而，这在数值模拟上并不是十分方便：CUDA底层采用SIMT的运行，引入额外的分支处理将会显著降低程序性能。引入软化因子，上式可以变为：    那么，物体在某时刻的加速度便可以由下式得到：    因此，如果已知所有物体质量，以及它们在某时刻的速度和位置，通过加速度，便可以得到它们在下一时刻的速度和位置。 | |
| **四、实验目的：**  1. 使用CUDA编程环境实现N-Body并行算法。  2. 掌握CUDA程序进行性能分析以及调优方法。 | |
| **五、实验内容：**  1. 学习和使用集群及CUDA编译环境  2. 基于CUDA实现N-Body程序并行化  3. N-Body并行程序的性能优化 | |
| **六、实验器材（设备、元器件）：**  1. 计算节点配置：CPU E5-2660 v4\*2，Nvidia K80\*2  2. 操作系统：CentOS 7.2  3. CUDA：10.0 | |
| **七、实验步骤及操作：**  **1. 在集群上运行基准代码**  （1）在校园网环境下使用远程连接工具通过ssh方式登录跳板机。    图7-1-1 登录跳板机  输入密码，登录成功。    图7-1-2 成功登录界面  （2）输入命令ssh mpi-cu07-1进入运行节点。    图7-1-3 进入运行节点  （3）粘贴实验指导书给出的串行代码编译并运行。    图7-1-4 运行基准代码  **2. 并行化基准代码**  （1）将函数bodyForce()定义为\_\_global\_\_，并将i从0到n的循环，改为给i赋值threadIdx.x + blockDim.x \* blockIdx.x，i小于n时执行语句：    图7-2-1 bodyForce函数修改前    图7-2-2 bodyForce函数修改后  （2）将位置整合部分的语句定义为一个函数positionIntegration()，同函数bodyForce()，定义为\_\_global\_\_，并将i从0到n的循环，改为给i赋值threadIdx.x + blockDim.x \* blockIdx.x，i小于n时执行语句：    图7-2-3 位置更新修改前    图7-2-4 位置更新修改后  （3）更改函数bodyForce()和integrate\_position()的调用方式，加入块数和每块的线程数：    图7-2-5 更改函数调用方式  （4）增加GPU上的数组d\_p以记录N个天体的位置信息，并将CPU上的数据通过cudaMemcpy拷贝到GPU上：    图7-2-6 GPU与CPU的资料拷贝  （5）定义块数和每块的线程数：    图7-2-7 global函数所需参数的计算  （6）如果迭代到最后一轮，GPU将d\_p再拷贝回CPU上，由于kernel的创建是异步的，cudaMemcpy是同步的，所以这里也起到同步化的作用：    图7-2-8 同步问题处理  **3. 优化并行代码**  **3.1 使用shared memory**  经过理论课的学习得知，在通过cudaMemcpy函数将CPU上的资料拷贝到GPU上后，默认是存在GPU里的global memory中。在GPU内部，thread与global memory的交互速度远低于shared memory。  经过分析，在原来的bodyForce函数中，对于j的每一次循环，需要访问global memory6次（分别访问p[i]和p[j]的x、y、z的值），但计算只有19个FLOPs。事实上，global\_memory的访问速度低于计算速度10倍以上，而这里访存和计算比只有1：3，所以很明显访存速度是瓶颈。    图7-3-1 基础并行代码中的bodyForce函数  因此，我们可以考虑将p数组从global\_memory搬到shared memory中，shared\_memory属于片上存储，位于每个SM中，其访问速度比global\_memory快很多，因此非常适合存在date reuse的存储优化。  搬运的过程也可以并行化，对于每个Block中，共有BLOCK\_SIZE个线程，每个线程只用搬运其中一个元素，即可实现整个Block的shared memory中存储BLOCK\_SIZE个元素的效果，修改代码如下：    图7-3-2 引入共享内存后的处理  由于每个线程负责计算一个天体的运动情况，故可以将该天体的数据放在register中，访存速度更快，这里用sx0，sy0，sz0进行存储。上面代码中的sp数组即为将p数组搬运到shared memory之后的结果。  **3.2 消除Bank Conflict**  通过理论课的学习，我们知道shared memory其实是由16个bank组成的，shared memory的访问也是以half-warp为单位。如果同一个half-warp里的多个thread同时访问同一个bank会发生bank conflict，也即会使存取效率下降。  特殊地，如果half-warp里的所有thread访问同一个bank里的同一个数据时，硬件做了优化，可以进行广播，使得不存在bank conflict。因此，在优化中，我选择让同一个half-warp中的thread访问同一个bank里的同一个数据，以消除bank conflict。    图7-3-3 消除bank conflict  **3.3 数据进一步分块**  在完成优化存取方式后，程序性能明显提升。此时，由于每个线程要计算某一天体对另外N个天体的受力，所以需要进行N次计算。但是，此时GPU并没有最大程度上利用。因此，我们可以考虑将数据进一步分块，即每个线程只负责一部分天体的受力计算，最后进行叠加即可。  记计算同一天体受力的进程数为BLOCK\_TILE，则需要把n分为BLOCK\_TILE块，每一块的下标计算可以利用blockIdx.x/BLOCK\_NUM获得，由于本次测试的N为4096，BLOCK\_TILE为32，BLOCK\_SIZE为128，故只需要进行一次分块即可完成所有数据：    图7-3-4 数据进一步分块的计算处理  在随后的速度更新中，因为可能有属于不同block的线程对同一个天体的位置进行更新，所以更新的时候会有并发问题，因此采用了原子操作防止读后写问题：    图7-3-5 数据进一步分块之后的位置更新  **3.4 合并位置更新与受力计算**  在基础并行代码中，位置更新是在受力计算完成后，创建新的一个kernel进行计算。但这样会使得效率大打折扣：第一，同一个GPU上创建kernel的过程是异步的，但是kernel与kernel之间会被强制串行执行。第二，在第一个kernel结束后，程序会切换回CPU，进行资料拷贝，因为CPU与GPU之间进行资料传输的速度很慢，所以这样的切换会带来巨大的开销。  经过对程序结构进行分析后，发现其实位置更新并不依赖于受力计算全部完成。事实上，只要对某个天体i的所有部分受力计算完成后，就可以进行该天体位置的更新。因此，在进一步的优化中我把位置更新的代码整合进了受力计算的kernel中。  但是这样会带来一个问题，由于3.2所介绍的数据分块思路，其实是由不同kernel上的thread计算同一个物体的部分受力，因此需要进行block间的同步。但是nvidia没有提供block间同步的方法，因为这样会使得GPU运算效率大大下降。因此，我选择了引入state数组来记录对于同一个天体还剩多少个thread未完成计算。由于在本程序中BLOCK\_TILE设置为4，因此初始时state数组的值应设置为4，然后当一个thread计算完成后对state进行原子减操作（因为一样会有并发问题）。当state减为0后，说明对于当前天体，所有部分受力计算均已完成，且速度更新已完成，因此可以进行位置的更新。    图7-3-6 合并位置更新与受力计算  值得注意的是，在位置更新部分我并没有使用原子操作进行更新，因为p[i].x、p[i].y、p[i].z其实在前面我们已经存入register中，故直接使用会更快，这样也会使得纵使存在读后写问题，但结果依然不会受到影响。因为原子操作会涉及到加锁和解锁的问题，其实是牺牲了部分效率的，所以能不用最好不用。最后位置更新完再把state[i]恢复为BLOCK\_TILE，一边下一次迭代继续使用。  **3.5 循环展开**  通过理论课的学习，我了解到其实对于同一个block上同一个warp的thread属于SIMD架构，因此减少控制分支很重要，因此在本次优化中还采用了循环展开技术，以期减少控制分支判断次数。因为如果展开的次数不能被N整除的话会增加很多额外的判断，因此测试使用的展开数均为4096的公因数，测试结果见：八、实验数据及结果分析部分。    图7-3-7 循环展开优化  **3.6 合理选择BLOCK\_SIZE和BLOCK\_TILE的大小**  BLOCK\_SIZE与BLOCK\_NUM之间存在代数关系，其乘积应该等于N，故下面只考虑BLOCK\_SIZE和BLOCK\_TILE的大小设置。  由于本次的测试数据N仅为4096，并不是很大，再加上shared memory容量足够，于是可以分块之后一次便计算完，因此BLOCK\_TILE和BLOCK\_SIZE的乘积也应该等于N，即BLOCK\_TILE的数值应该与BLOCK\_NUM相同，否则无法最大程度利用GPU性能。  因此，BLOCK\_SIZE和BLOCK\_TILE的乘积一定，即两者成反比。但是，随着BLOCK\_SIZE的增大，同一个block中的thread越多，data reuse越大；随着BLOCK\_TILE的增大，数据分块越多，单独一个thread需要计算的部分减小。  不难发现，BLOCK\_SIZE和BLOCK\_TILE的变化是冲突的：如果增大BLOCK\_SIZE，从数据复用的角度GPU效率会得到提高，但这又会影响到BLOCK\_TILE，使其减小，从单个thread计算时间的角度分析GPU效率会降低。因此，需要折中找到一个效率最高的点。于是，改变BLOCK\_SIZE的值分别为256、128、64、32进行测试，经过对比，选取最佳的BLOCK\_SIZE和BLOCK\_TILE的值。测试结果见：八、实验数据及结果分析部分。    图7-3-8 BLOCK\_SIZE以及BLOCK\_TILE的选取 | |
| **八、实验数据及结果分析：**  **1. 基准代码**  结果如下图所示，平均每秒3300万次interactions：    图8-1 基准代码运行结果  **2. 并行代码**  结果如下图所示，平均每秒111.25亿次interactions，与基准代码结果对比，可以看出并行化大幅提升了性能，大概是347倍。    图8-2 并行代码运行结果  **3. 优化代码**  **3.1 循环展开次数的选取**  以下各实验BLOCK\_SIZE的选取均为128。  当不展开时，结果如下图所示，平均每秒544.54亿次interacitons。    图8-3-1 循环不展开时的性能  当循环展开次数为2时，结果如下图所示，平均每秒455.65亿次interacitons。    图8-3-2 循环展开次数为2时的性能  当循环展开次数为4时，结果如下图所示，平均每秒540.33亿次interacitons。    图8-3-3 循环展开次数为4时的性能  当循环展开次数为8时，结果如下图所示，平均每秒568.14亿次interacitons。    图8-3-4 循环展开次数为8时的性能  当循环展开次数为16时，结果如下图所示，平均每秒579.12亿次interacitons。    图8-3-5 循环展开次数为16时的性能  当循环展开次数为32时，结果如下图所示，平均每秒590.96亿次interacitons。    图8-3-6 循环展开次数为32时的性能  因为本次测试数据循环次数只有32次，因此只测试到32。根据比较，最佳展开次数应该设置为32。  **3.2 BLOCK\_SIZE的选取**  以下各实验循环展开次数的选取均为32。  当BLOCK\_SIZE为64时，结果如下图所示，平均每秒521.36亿次interacitons。    图8-3-7 当BLOCK\_SIZE为64时的性能  当BLOCK\_SIZE为128时，结果如下图所示，平均每秒589.29亿次interacitons。    图8-3-8 当BLOCK\_SIZE为128时的性能  当BLOCK\_SIZE为256时，结果如下图所示，平均每秒573.38亿次interacitons。    图8-3-9当BLOCK\_SIZE为256时的性能  当BLOCK\_SIZE为512时，结果如下图所示，平均每秒586.82亿次interacitons。    图8-3-10 当BLOCK\_SIZE为512时的性能  其实当BLOCK\_SIZE为512时，其性能不稳定，55-61billion都有出现，所以最后综合考虑选择了128作为BLOCK\_SIZE大小。 | |
| **九、实验结论：**  1. 成功使用CUDA编程实现N-Body问题并行程序并对其进行性能优化。  2. 由实验数据得知，并行化能大幅提升程序性能，优化也能在一定程度上提升程序性能。该结果也证明了本次实验是成功的。  3. 优化中的分块这一步，块的大小对程序性能有一定的影响。经实验得知，在多次测试的过程中，测试结果会有一些波动，但总的来说，在块大小取128时，性能最好。 | |
| **十、总结及心得体会：**  通过本次实验，我复习了课程中所学的CUDA相关的知识，包括基本的并行化方法和一些常用的优化方法。对课程中学到的一些较为抽象的知识有了更深入的了解，也对使用CUDA并行化串行程序，提升其性能有了更深入的理解。在编写、调试、修改程序的过程中，巩固课堂知识的同时，我还锻炼了编程能力与实践能力。在对并行程序进行一步步优化的过程中，我学会了在网上查找相关资料，拓宽了知识面。总的来说，通过本次实验，我锻炼了分析问题、解决问题、得出结论的能力，获益匪浅。 | |
| **十一、对本实验过程及方法、手段的改进建议：**  无 | |
| **报告评分：**  **指导教师签字：** | |

电子签名：

