**电 子 科 技 大 学**

**实 验 报 告**

|  |  |
| --- | --- |
| **学生姓名： 万理** | **学 号：2019081310017** |
| **一、实验室名称：无** | |
| **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：**  N-Body问题又称为多体问题，N表示任意正整数。它是天体力学和一般力学的基本问题之一，研究N个质点相互之间作用的运动规律，对其中每个质点的质量和初始位置、初始速度都不加任何限制。简单的说，N-Body问题是指找出已知初始位置、速度和质量的多个物体在经典力学情况下的后续运动，它既可以应用于宏观的天体，也可以应用于微观的分子、原子。  在有关多体问题（n>=3）的物理文学作品里有时会发现像“解决多体问题是不可能的”这样的描述。*n*体问题包含*6n*个变量，因为每个质点需要3个空间坐标和3个分速度表示。  即对于已知初始位置与速度、质量的n个变量，通过计算它们之间的相互作用力，求出每个变量在下一时刻的运动情况。 | |
| **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）：**  计算节点配置：  CPU E5-2660 v4\*2  Nvidia K80\*2  操作系统：CentOS 7.2  CUDA：10.0 | |
| **七、实验步骤及操作：**   1. 登录远程服务器并测试 2. 使用远程连接工具XShell通过ssh方式登录集群。 3. 成功登录集群后，默认进入管理节点mu01，在此节点上传代码，修改代码。 4. 运行代码需要进入运行节点cu04或cu05。在管理节点使用如下命令进入运行节点：   ssh cu04   1. 进入运行节点后使用如下命令编译代码：   nvcc -o compiledFile sourceFile.cu  使用如下命令运行编译后的文件：  ./compiledFile   1. 串行代码的并行化 2. 通过cudaMemcpy()函数实现将数据拷贝到GPU上：   第一步先将数据从Host拷贝到Device：其中buf是要拷贝的数据，d\_buf是拷贝的目标地址，bytes是需要传送的字节数。第二步是在最后一轮执行完后（设置执行10轮），将数据再从Device拷贝回Host：此时d\_buf是要拷贝的数据，buf是拷贝的目标地址，bytes是需要传送的字节数（拷贝的目标地址与拷贝的数据进行了互换）。  修改的部分代码如图：    完成全部的数据拷贝过程。   1. 数据成功到达GPU后进行并行化处理：   数据成功到达GPU后就可以进行并行化处理，假设1个线程处理1个节点的运动情况，则对于4096颗粒子的N-Body问题共需要4096个线程。其中的每个线程都需要确定处理的节点（线程在4096个节点中对应的索引），每个线程对应一个节点。关键代码如下：   |  | | --- | | void bodyForce(Body \*p, float dt, int n) {  int i = blockDim.x \* blockIdx.x + threadIdx.x;  if (i < n) {  float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f; |   由于每个线程需要的数据都是所有节点的位置信息，因此申请三组大小为块大小的shared\_memory数组，用来存储需要的信息和自己本身节点的位置信息，以便充分利用shared\_memory。关键代码如下：   |  | | --- | | \_\_shared\_\_ float3 spos[BLOCK\_SIZE];  Body tpos = p[tile \* blockDim.x + threadIdx.x];  spos[threadIdx.x] = make\_float3(tpos.x, tpos.y, tpos.z); |  1. 计算每个节点受到的其他节点的合作用力：   每个节点受到其他节点的作用力大小可以分别计算然后相加，通过一个循环累加得到最终的作用力。由于共享内存数组的大小受BLOCK\_SIZE大小限制，因此每轮大循环可以计算BLOCK\_SIZE个节点的分作用力，共循环n/BLOCK\_SIZE次。关键代码如下：（为了避免出现数据不同步的问题，在取数据和计算数据的每个操作后都增加一个syncthreads()同步函数进行同步   |  | | --- | | for (int tile = 0; tile < gridDim.x; tile++) {  \_\_shared\_\_ float3 spos[BLOCK\_SIZE];  Body tpos = p[tile \* blockDim.x + threadIdx.x];  spos[threadIdx.x] = make\_float3(tpos.x, tpos.y, tpos.z);  \_\_syncthreads();  #pragma unroll  for (int j = 0; j < BLOCK\_SIZE; j++) {  float dx = spos[j].x - p[i].x;  float dy = spos[j].y - p[i].y;  float dz = spos[j].z - p[i].z;  float distSqr = dx\*dx + dy\*dy + dz\*dz + SOFTENING;  float invDist = rsqrtf(distSqr);  float invDist3 = invDist \* invDist \* invDist;  Fx += dx \* invDist3;  Fy += dy \* invDist3;  Fz += dz \* invDist3;  }  \_\_syncthreads();  } |   循环结束就得到了每个节点受到其他所有节点的合作用力。   1. 计算dt时间后每个节点的速度和位置：   dt时间后每个节点的速度计算代码如下：   |  | | --- | | p[i].vx += dt\*Fx; p[i].vy += dt\*Fy; p[i].vz += dt\*Fz; |   dt时间后每个节点的位置：定义一个新的函数nextPos用来计算下一时刻节点的位置。每个线程分别计算一个节点的位置。代码如下：   |  | | --- | | \_\_global\_\_ void nextPos(Body\* p, float dt, int n) {  int i = threadIdx.x + blockIdx.x \* blockDim.x;  if (i < n) {  p[i].x += p[i].vx \* dt;  p[i].y += p[i].vy \* dt;  p[i].z += p[i].vz \* dt;  }  } |   通过nIters次循环后，N-Body问题的并行化就完成了。   1. 登录远程服务器并测试 2. 使用远程连接工具XShell通过ssh方式登录集群。 3. 成功登录集群后，默认进入管理节点mu01，在此节点上传代码，修改代码。 4. 运行代码需要进入运行节点cu04或cu05。在管理节点使用如下命令进入运行节点：   ssh cu04   1. 进入运行节点后使用如下命令编译代码：   nvcc -o nbody1 nbody1.cu  使用如下命令运行编译后的文件：  ./nbody1   1. 并行化代码的优化   当前的并行化代码是1个线程处理1个节点，用到了4096个线程，而GPU上有更多的线程可用，浪费了GPU的计算资源，因此通过多个线程处理1个节点的方法来对并行化代码进行优化。关键是要确定每个线程要处理的节点和需要计算的节点对：将每n个线程划分为一组，分别按组内索引处理节点，需要确定每个线程自己本身对应的节点，在原先索引的基础上对n取余就可以确定。  此外，还可以使用共享内存保存需要多次访问的数据（注意线程同步）以及编译指示进行循环展开的方法，使得性能有所提升。   1. 登录远程服务器并测试 2. 使用远程连接工具XShell通过ssh方式登录集群。 3. 成功登录集群后，默认进入管理节点mu01，在此节点上传代码，修改代码。 4. 运行代码需要进入运行节点cu04或cu05。在管理节点使用如下命令进入运行节点：   ssh cu04   1. 进入运行节点后使用如下命令编译代码：   nvcc -o nbody2 nbody2.cu  使用如下命令运行编译后的文件：  ./nbody2 | |
| 1. **实验数据及结果分析：** 2. 串行代码的运行结果：     串行代码运行后的计算结果正确，但平均每秒只有0.034 Billiom Interactions，代码性能低。   1. 串行代码并行化后的运行结果：     并行化的代码运行后的计算结果正确，并且平均每秒达到了36.855 Billion Interactions，性能相比串行化代码有了显著的提升。   1. 串行代码并行化并优化后的运行结果：     优化后的并行化代码运行后的计算结果正确，并且平均每秒达到了215.093 Billion Interactions，性能相比优化前的并行化代码有了显著的提升。  结果分析：优化前的代码1个线程处理1个节点，还是有较多的空闲线程，没有充分利用GPU的计算资源。实现多个线程处理一个节点后，虽然会增加通信开销，但是多个线程并行带来的计算能力更强，性能也有所提升。同时，共享内存与循环展开的使用，也对性能有一定提升。 | |
| **九、实验结论：**  串行代码的性能为0.034 Billion Interactions /s，对串行代码进行并行化后的代码性能提升到了36.855 Billion Interactions /s，对并行化后的代码再进行优化性能提升到了215.093 Billion Interactions /s，均有显著的提升。在对并行化代码的优化的过程中遇到了一些困难，但根据课上所学知识，通用多个线程处理一个节点、使用共享内存保存需要多次访问的数据以及编译指示进行循环展开的方法，最终使得性能有所提升。不过，循环展开需要权衡寄存器大小和线程数量之间的关系，共享内存使用时也需要注意线程同步。 | |
| **十、总结及心得体会：**  通过本次实验，我掌握了集群的使用和CUDA的编译环境，通过CUDA编译环境实现了N-Body问题的并行化和优化，让我更深入地掌握了基于CUDA的一些函数用法，对串行化代码的并行化不再是一筹莫展，有了初步的认识。通过这次实验也让我对课上学的一些优化代码性能的思路有了回顾并且用于实践，但是掌握的还是非常的少并且在实现的过程中还是遇到了困难，说明课下还需要仔细钻研，还需要更多的思考和实践。 | |
| **十一、对本实验过程及方法、手段的改进建议：**  本实验只要求比较N-Body问题在4096个粒子时的性能，具有局限性，可以增加几组规模不一的粒子数，分别通过调整代码（改变参数或者改变优化方法）来使N-Body问题在不同的数据规模下的性能达到最优，使学生对基于CUDA的并行化有更好的掌握。 | |
| **报告评分：**  **指导教师签字：** | |

学生签名：万理