**电 子 科 技 大 学**

**实 验 报 告**

|  |  |
| --- | --- |
| **学生姓名： 张皓** | **学 号：2018051406001** |
| **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：**  1 N-Body算法  N-body问题（或者说N体问题），是一个常见的物理模拟问题。在N-body系统中，每个粒子体都会与剩下的其他粒子产生交互作用（交互作用因具体问题而异），从而产生相应的物理现象。  首先，对每个点的位置和速度进行随机初始化。并存放在一个结构体数组中。   |  | | --- | | typedef struct { float x, y, z, vx, vy, vz; } Body;  /\*  \* Do not modify this function. A constraint of this exercise is  \* that it remain a host function.  \*/  void randomizeBodies(float \*data, int n) {  for (int i = 0; i < n; i++) {  data[i] = 2.0f \* (rand() / (float)RAND\_MAX) - 1.0f;  }  } |   然后，根据位置进行计算相互的作用力，以此为依据更新点的速度，该代码被整合到一个bodyForce函数中。   |  | | --- | | void bodyForce(Body \*p, float dt, int n) {  for (int i = 0; i < n; ++i) {  float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;  for (int j = 0; j < n; j++) {  float dx = p[j].x - p[i].x;  float dy = p[j].y - p[i].y;  float dz = p[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;  }  p[i].vx += dt\*Fx; p[i].vy += dt\*Fy; p[i].vz += dt\*Fz;  }  } |   根据新的速度，更新每个点的位置，然后开始下一个周期的计算。   |  | | --- | | for (int i = 0 ; i < nBodies; i++) { // integrate position  p[i].x += p[i].vx\*dt;  p[i].y += p[i].vy\*dt;  p[i].z += p[i].vz\*dt;  } |   2 CUDA并行程序设计  由于在计算每一个点的受力情况是相互独立互不影响的，可以利用GPU高并发的特点，用n个线程并发计算速度变化，然后用n个线程并发更新位置。详细代码分析会在实验步骤中进一步叙述。  3 CUDA并行程序优化  原理2中的简单并行程序设计虽然也可以实现并行，但是效率过低，这是因为频繁访问全局内存所  导致的，故用BLOCK\_STEP将每一个线程的任务再次划分，每个线程负责某一个点的N/BLOCK\_STEP的速度分量计算。  每个线程的计算时间不一致，在上述的程序逻辑中，需要等待所有线程计算完速度的改变量才进行位置的全局更新，我们可以将更新位置的操作放在计算速度改变量中，用计数器判断某个点的数据是否已经被算完，如果算完则立即更新，无需等待。  详细代码和优化思路会在实验步骤中进一步叙述。 | |
| **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）：**  学校集群服务器，配置如下：  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 并行程序设计**  **Step1：申请CPU和GPU内存空间并对数据进行初始化和拷贝操作。**   |  | | --- | | int bytes = nBodies \* sizeof(Body);  float \*buf;  cudaMallocHost((void \*\*)&buf,bytes);  /\*  \* As a constraint of this exercise, `randomizeBodies` must remain a host function.  \*/  randomizeBodies(buf, 6 \* nBodies); // Init pos / vel data  float \*d\_buf;  cudaMalloc((void \*\*)&d\_buf,bytes);  Body \*d\_p=(Body \*)d\_buf;  cudaMemcpy(d\_buf,buf,bytes,cudaMemcpyHostToDevice); |   **Step2: 设计bodyForce函数**  如下述代码所示，对于每个线程，只需根据threadIdx.x和blockIdx.x计算对应在p中的位置，然后进行全局更新即可。   |  | | --- | | \_\_global\_\_ void bodyForce(Body \*p, float dt, int n) {  int i=threadIdx.x+blockIdx.x\*blockDim.x;  if(i<n){  float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;  for (int j = 0; j < n; j++) {  float dx = p[j].x - p[i].x;  float dy = p[j].y - p[i].y;  float dz = p[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;  }  p[i].vx += dt\*Fx; p[i].vy += dt\*Fy; p[i].vz += dt\*Fz;  }  } |   **Step3: 设计integrate\_position函数**  如下述代码所示，对于每个线程，只需根据threadIdx.x和blockIdx.x计算对应在p中的位置，然后更新位置即可。   |  | | --- | | \_\_global\_\_ void integrate\_position(Body \*p,float dt,int n){  int i=threadIdx.x+blockIdx.x\*blockDim.x;  if(i<n){  // integrate position  p[i].x += p[i].vx\*dt;  p[i].y += p[i].vy\*dt;  p[i].z += p[i].vz\*dt;  }  } |   **Step4：循环**  根据给定的周期数对数据进行更新，最后将GPU的数据拷贝后CPU数据进行评测。  **2 并行程序优化**  **2.1 优化1—— BLOCK\_STEP引入和shared\_memory**  分析基本并行程序可知，程序的主要开销是在bodyForce中。对于每一个线程，需要计算的量是n，而且对于全局内存的频繁访问会导致大量的时间开销，故，我们可以用空间换取时间，使用BLOCK\_STEP倍的线程使每个线程的计算量得到提升，同时在同一个线程块中使用shared\_memory存放最近会访问到的数据。  **2.2 优化2—— 计算合并**  每个线程的计算时间不一致，在上述的程序逻辑中，需要等待所有线程计算完速度的改变量才进行位置的全局更新，我们可以将更新位置的操作放在计算速度改变量中，用计数器判断某个点的数据是否已经被算完，如果算完则立即更新，无需等待。Flag数组是用于记录计算情况的数据，每个线程算完后会对对应的flag[i]原子减一，如果为0则立即更新位置并重置flag[i]，注意操作的原子独立性。   |  | | --- | | atomicSub(&flag[i], 1);  if(!atomicMax(&flag[i], 0)){  // integrate position  atomicAdd(&p[i].x,p[i].vx\*dt);  atomicAdd(&p[i].y,p[i].vy\*dt);  atomicAdd(&p[i].z,p[i].vz\*dt);  atomicExch(&flag[i],BLOCK\_STEP);  } |   **2.2 优化3—— 编译优化**  使用编译优化，加入循环展开，调整得到最佳参数。  **2.4 优化4—— 其他优化方向**  考虑到GPU的硬件特点，一定是最后一组最后算完，故在最后一组顺便计算结果。 | |
| **八、实验数据及结果分析：**  对串行代码，简单并行代码，优化后的并行代码分别编译和运行，得到结果如下：  **1 串行代码：**    **2 简单并行代码**    **3 优化后并行代码** | |
| **九、实验结论：**  利用GPU的高并发多线程计算特性，可以对串行代码进行极大的性能提升，shared\_memory可以有效减少访问数据的时间开销，循环展开也可以极大的提高程序的性能。 | |
| **十、总结及心得体会：**  1. 使用了CUDA编程环境实现 N-Body并行算法。  2. 掌握了CUDA程序进行性能分析以及调优方法。 | |
| **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：** | |

