**电 子 科 技 大 学**

**实 验 报 告**

|  |  |
| --- | --- |
| **学生姓名** | **学 号：** |
| **一、实验室名称：** | |
| **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问题是一个常见的物理模拟问题，每个粒子会与其他粒子产生相互作用，从而使得整个系统进行变化。不同的粒子直接有着相互的吸引力，因此会随着时间改变各个粒子的位置和速度。每隔一个时间间隔进行一次计算，有N2的复杂度。这次实验是使用CUDA，利用GPU进行这种重复大量的计算。 | |
| **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）：**  浪潮 5280M4  Nvidia GTX1080  CPU： E5-2660 v4 2颗  操作系统：CentOS 7.2  CUDA：10.0 | |
| **七、实验步骤及操作：**   1. 首先将串行代码改成并行算法。将4096个粒子分给4096个线程进行计算，这里需要计算每个线程对应要处理的粒子号。 2. int i = threadIdx.x + blockDim.x \* blockIdx.x; 3. 随后对每个粒子进行计算，代码和串行算法相同。要注意的是计算有多少Block的时候，有可能出现有多出一个Block又不填满的情况。当然在这次实验中因为Block的大小必须是Warp的整数倍，Warp大小为32，4096是32的倍数，因此不会发生这种情况。在最后要注意同步，防止没有计算完成就输出或者进行下一个运算。 4. size\_t Block\_num = (nBodies + BLOCK\_SIZE - 1) / BLOCK\_SIZE; 5. bodyForce<<<Block\_num, BLOCK\_SIZE>>>(p, dt, nBodies); *// compute interbody forces* 6. cudaDeviceSynchronize(); 7. 增加计算线程优化。因为数据量很小，因此不能充分利用GPU，所以可以通过增加Block的方法加速。具体思路是本来一个粒子由一个线程进行计算，现在多分配几个进程用来计算这个粒子。要注意的是对力的相加要用原子操作，防止数据冲突。具体代码改动如下： 8. int id = threadIdx.x + blockDim.x \* (int) (blockIdx.x / BLOCK\_STRIDE); 9. int block\_start = blockIdx.x % BLOCK\_STRIDE; 10. for (int block\_id = block\_start; block\_id < total\_block; block\_id += BLOCK\_STRIDE) { 11. *//计算部分* 12. \_\_syncthreads(); 13. } 14. atomicAdd(&v[id].x, dt \* Fx); 15. atomicAdd(&v[id].y, dt \* Fy); 16. atomicAdd(&v[id].z, dt \* Fz); 17. 共享内存优化。为了优化内存的利用率，因为一个Block中的共享内存速度比全局内存快，因此使用共享内存。但是要注意一个Block中的线程都要将数据保存到共享内存才能进行下一步操作，因此需要同步。修改部分如下： 18. \_\_shared\_\_ float3 spos[BLOCK\_SIZE]; 19. temp = p[block\_id \* BLOCK\_SIZE + threadIdx.x]; 20. spos[threadIdx.x] = make\_float3(temp.x, temp.y, temp.z); 21. \_\_syncthreads(); 22. dx = spos[j].x - p\_temp.x; 23. dy = spos[j].y - p\_temp.y; 24. dz = spos[j].z - p\_temp.z; 25. 内存对齐优化。观察计算过程可以发现，在第一步计算力的作用的时候只用到了每个粒子的位置，计算下一时刻的位置的时候才会用到粒子的速度，因此可以将每个粒子的位置合并在一起，有利于内存的利用。也就是将内存的前一半防止粒子的位置，后一半放置速度。并且改用float4类型保存，有利于内存的对齐。但是因为老师的check函数用于原本的结构，并且随机数是固定的，因此只能手动进行结构的转换，这部分浪费的时间在正常的算法中是不需要的，因此不用于计时。 26. typedef struct { 27. float4 \*pos, \*vel; 28. } Body; 29. float \*temp\_buf = (float \*) malloc(nBodies \* sizeof(float) \* 6); 30. randomizeBodies(temp\_buf, 6 \* nBodies); *// Init pos / vel data* 31. *//buf转换成我定义的格式* 32. int idx = 0; 33. for (int i = 0; i < 4 \* nBodies; i++) { 34. if ((i + 1) % 4 == 0) { 35. buf[i] = 0; 36. idx += 3; 37. continue; 38. } 39. buf[i] = temp\_buf[idx++]; 40. } 41. idx = 3; 42. for (int i = 4 \* nBodies; i < 8 \* nBodies; i++) { 43. if ((i + 1) % 4 == 0) { 44. buf[i] = 0; 45. idx += 3; 46. continue; 47. } 48. buf[i] = temp\_buf[idx++]; 49. } 50. *//把我的格式转换成老师的* 51. Body temp = {(float4 \*) buf, ((float4 \*) buf) + nBodies}; 52. for (int i = 0; i < nBodies; i++) { 53. temp\_buf[i \* 6] = temp.pos[i].x; 54. temp\_buf[i \* 6 + 1] = temp.pos[i].y; 55. temp\_buf[i \* 6 + 2] = temp.pos[i].z; 56. temp\_buf[i \* 6 + 3] = temp.vel[i].x; 57. temp\_buf[i \* 6 + 4] = temp.vel[i].y; 58. temp\_buf[i \* 6 + 5] = temp.vel[i].z; 59. } 60. buf = temp\_buf; 61. 编译优化以及手动分配内存。使用cudaMallocHost(&buf, bytes); cudaMemcpy(device\_buf, buf, bytes, cudaMemcpyHostToDevice) #pragma unroll 32;的函数和设置，能对程序有一些提升。编译优化指的是将一个循环中的代码多次运行，减少循环次数。 62. 对于Block\_Size和Block\_stride的取值问题，我使用一个python脚本进行编辑源文件和统计运行时间，得到了在这台机器上特定的解。由于我做实验的时间比较早，因此没有人和我竞争GPU的资源，也没有影响到其他同学。 | |
| **八、实验数据及结果分析：**   1. 测试了不同优化方法的性能，得到如下图表。可以看出从串行到并行的提升速度是极为明显的，增加计算的线程量的提升也比较明显。内存对齐的方法提升较小。      1. 在对于Block\_Size和Block\_stride的测试中，我发现cuda程序的调参非常简单，因为Block\_Size是32的倍数，可操作空间少，最终发现最优的参数是我头几个试验得到的结果。 | |
| **九、实验结论：**  对于nbody的串行算法学习了如何改成并行算法，并且通过增加计算线程，使用共享内存，内存对齐等方法优化了cuda程序的性能，对于理解并行程序有了深刻的认识。 | |
| **十、总结及心得体会：**  1、 学习了CUDA编译环境  2、 学习了用CUDA实现nbody问题  3、 学习了并行程序性能分析和优化 | |
| **十一、对本实验过程及方法、手段的改进建议：**  老师上课讲有基于CUDA流的优化方法，但是我并不明白如何使用。 | |
| **报告评分：**  **指导教师签字：** | |