**电 子 科 技 大 学**

**实 验 报 告**

|  |  |
| --- | --- |
| **学生姓名：王正仁** | **学 号：2019081308021** |
| **一、实验室名称：品学楼-A107** | |
| **二、实验项目名称：N-Body问题并行程序设计及性能优化** | |
| 1. **实验原理：** 2. **Body问题**   N-Body问题又称为多体问题，*N*为任意正整数。它是天体力学和经典力学的基本问题之一，研究N个质点受万有引力支配下相互作用的运动规律，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、速度和质量的多个质点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。同时其契合当前科学前沿，在高性能计算领域也具有一定的代表性。  具体地：在三维欧几里得空间中，分布有一定数量的粒子(忽略它们的体积)，每对粒子间都存在万有引力相互作用。他们的初始位置和质量是确定的，由于粒子间引力的作用，每个粒子根据各自的受力情况会获得不同的加速度，即对于每一个粒子，需要把另外 *N - 1* 个粒子对它作用的结果叠加起来。单一粒子的计算量为 ，进而导致具有 *N* 个粒子的实例的单次计算具有 的总时间复杂度。  该问题的两个重要特征是：  1.计算规模大。因为无论是宇观的天体尺寸还是微观的分子尺度，都包含了大量的粒子，粒子的规模大到数百万、千万。由于在系统中任意的两个粒子间都存在着相互作用，因此直接计算粒子间的相互作用的量级就是。  2.系统是动态变化的。为了反映系统的具体变化，尤其是在微观分子结构中，要求时间步足够小。这两个特征决定了计算机模拟时巨大的计算量，这对于任何高性能的单台计算机来说都是一个很难突破的瓶颈，因此采用并行计算是解决N-Body问题的必然选择。  **CUDA平台**  CUDA（Compute Unified Device Architecture），是显卡厂商NVIDIA推出的运算平台。CUDA是一种由NVIDIA推出的通用并行计算架构，该架构使GPU能够解决复杂的计算问题。 它包含了CUDA指令集架构（ISA）以及GPU内部的并行计算引擎。  开发人员可以使用C语言来为CUDA架构编写程序（C语言是应用最广泛的一种高级编程语言），所编写出的程序可以在支持CUDA的处理器上以超高性能运行。  本次实验运用CUDA进行并行编程以提高性能。 | |
| 1. **实验目的：** 2. 学习CUDA程序在Linux操作系统下的编译、执行和调试方法。 3. 基于CUDA平台设计和实现并行N-Body算法。 4. 掌握CUDA程序的性能分析以及调优方法。 | |
| 1. **实验内容：** 2. 学习和使用CUDA平台。 3. 基于CUDA环境，完成N-Body程序基准代码的并行化。 4. 分析和优化N-Body并行程序。 | |
| **六、实验器材（设备、元器件）：**  本地操作系统：Windows 10 x64 Pro  计算节点配置：OS：CentOS 7.2  CPU：E5-2660 v4\*2  GPU：Nvidia K80\*2  CUDA：10.0 | |
| 1. **实验步骤及操作：** 2. **基准代码并行化**   首先将基准代码保存并命名为“nbody1.cu”并传输至管理节点“mu01”。  登录管理节点“mu01”，使用“ssh cu04”或“ssh cu05”进入计算节点。  输入命令“nvcc -o ori ./nbody1.cu”进行编译，并在完成后输入“./ori”进行执行。图1所示为基准代码运行结果。    图1  考虑到每一个点的受力计算及速度位置更新是相互独立的，并且每个粒子需要的计算流程是相同的（符合CUDA的SIMT架构）。故对于大量的粒子，直观地我们可以让CUDA中每个线程负责一个粒子的更新，从而并行更新所有粒子的信息，获得巨大的加速。  本次实验中，给定4096个粒子，通过参数调优，程序中将使用4096个线程，并将其划分为128个块，每个块包含32个线程。   1. 修改bodyForce函数   对于每个线程而言，首先应根据“threadIdx.x+blockIdx.x\*blockDim.x”计算其线程的索引，也就是其负责更新的粒子的索引。然后统计其受力，并更新其速度。代码如图2所示。    图2   1. 修改integrate\_position函数   同样对于每个线程而言，首先应根据“threadIdx.x+blockIdx.x\*blockDim.x”计算其线程的索引，也就是其负责更新的粒子的索引，然后更新粒子位置。代码如图3所示。    图3   1. 申请并初始化主存和显存空间   如下图4所示为主存和显存空间分配，主存数据初始化、主存到显存的拷贝。    图4   1. 释放主存和显存   CUDA并行程序执行完成后，注意应该将申请的主存和显存都释放掉，从而归还系统。代码如图5所示。    图5  至此，基准代码的并行化已经完成。使用“nvcc -o nv1 ./nbody1.cu”和“./nv1”重新编译和执行，并行执行结果如图6所示，可以看出相比串行版本获得了1500倍的加速。    图6   1. **并行版本性能优化**   首先将并行化版本“nbody1.cu”另存为“nbody2.cu”，下文优化“nbody2.cu”性能。  优化1 共享内存  基于对CUDA系统执行模型和内存模型的理解，我们知道每个线程块内存在一块共享内存，其仅能被块内线程访问，但访问耗时远少于全局内存。利用“nvprof”分析并行化后的程序可以发现：bodyForce函数消耗了程序绝大部分执行时间。通过进一步分析可知：每一次bodyForce函数的调用都意味着O(n)的内存访问次数，而全局内存的大量访问会引起大量的时间开销。因此，我们引入Shared Memory，即同一个线程块中使用Shared Memory缓存全局内存中的数据，从而有效降低内存访问开销。代码如图7所示。    图7  优化2 循环展开  循环展开是一种牺牲程序的尺寸来加快程序的执行速度的优化方法，可以由程序员完成，也可由[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8)自动优化完成。循环展开为编译器进行指令调度带来了更大的空间，为具有多个功能单元的处理器提供[指令级并行](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4%E7%BA%A7%E5%B9%B6%E8%A1%8C)。我们可以使用“#pragma unroll”来展开任意一个给定迭代次数的循环，这里我们展开优化1中引入的Cache-Size大小的循环。代码如图8所示。    图8  优化3 计算分解：  此前所有的优化版本中，我们都让一个线程处理一个粒子，导致每个线程的计算量太大，并行度不够高。因此，我们可以考虑将一个粒子的处理任务分解到多个线程。相关代码如图9所示。（注意多线程并发时操作的原子性）    图9 | |
| 1. **实验数据及结果分析：** 2. **基准代码并行化**   将基准代码并行化后，编译执行。结果如图10所示，可以看出相比串行版本获得了1500倍的加速。    图10   1. **并行版本性能优化**   将并行版本性能优化后，编译执行。结果如图11示，可以看出相比串行版本获得了近6000倍的加速，即进一步获得了4倍的性能优化。    图11 | |
| 1. **实验结论：**   1.利用CUDA平台，对于特定的任务或执行单元，并行化可以轻松获取大量的加速。  2.特别要注意数据传输开销，包括全局内存的访问、主机和设备之间的内存拷贝。  3.针对不同的计算任务，应结合理论并实事求是，具体问题具体分析, 选择合适的数据结构以及设计高效的算法。  4.实践中越多的线程不意味着更多的加速比，对于规模较小的任务，并行的额外开销和进程竞争现象可能会影响效率。应该综合考虑数据规模、软硬件、系统状态来决定应该如何应用并行程序。 | |
| 1. **总结及心得体会：**   CUDA即Compute Unified Device Architecture，是NVIDIA推出的通用并行计算架构，包含了指令集架构以及并行计算引擎。  本次实验中，我紧密结合课内知识，通过多体问题的并行化和性能调优，实践上深化了我对CUDA平台的了解和掌握，拓宽了我的知识面，加强了我并行程序设计和实现的能力，激发了我对并行计算方向的学习兴趣。  感谢老师和助教的辛勤付出，收获颇丰、大有裨益！ | |
| 1. **对本实验过程及方法、手段的改进建议：** 2. 实验环境应配置开放nvprof命令，让学生即使无root权限也能评估程序性能。 3. 提供本地系统CUDA平台搭建和测试的指导。 | |
| **报告评分：**  **指导教师签字：** | |

**十二、附录：**

1. nbody1.cu
2. #include <math.h>
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include "timer.h"
6. #include "check.h"
7. #include <cuda\_runtime.h>
8. #define SOFTENING 1e-9f
9. #define BLK 32
10. */\**
11. \* Each body contains x, y, and z coordinate positions,
12. \* as well as velocities in the x, y, and z directions.
13. \*/
14. typedef struct
15. {
16. float x, y, z, vx, vy, vz;
17. } Body;
18. */\**
19. \* Do not modify this function. A constraint of this exercise is
20. \* that it remain a host function.
21. \*/
22. void randomizeBodies(float \*data, int n)
23. {
24. for (int i = 0; i < n; i++)
25. {
26. data[i] = 2.0f \* (rand() / (float)RAND\_MAX) - 1.0f;
27. }
28. }
29. */\**
30. \* This function calculates the gravitational impact of all bodies in the system
31. \* on all others, but does not update their positions.
32. \*/
33. \_\_global\_\_ void bodyForce(Body \*p, float dt, int n)
34. {
35. *// index of updated body*
36. int idx = threadIdx.x + blockDim.x \* blockIdx.x;
37. *// Resultant force on x,y,z axes*
38. float Fx=0, Fy=0, Fz=0;
39. *// iterate on all the bodies*
40. for (int j = 0; j < n; j++)
41. {
42. float dx = p[j].x - p[idx].x;
43. float dy = p[j].y - p[idx].y;
44. float dz = p[j].z - p[idx].z;
45. float distSqr = dx \* dx + dy \* dy + dz \* dz + SOFTENING;
46. float invDist = rsqrtf(distSqr);
47. float invDist3 = invDist \* invDist \* invDist;
48. *// accumualte resultant force on x,y,z axes*
49. Fx += dx \* invDist3;
50. Fy += dy \* invDist3;
51. Fz += dz \* invDist3;
52. }
53. *// update velocity on x,y,z axes*
54. p[idx].vx += dt \* Fx;
55. p[idx].vy += dt \* Fy;
56. p[idx].vz += dt \* Fz;
57. }
58. \_\_global\_\_ void integrate\_position(Body \*p, float dt, int n)
59. {
60. *// index of updated body*
61. int idx = threadIdx.x + blockDim.x \* blockIdx.x;
62. *// update position on x,y,z axes*
63. p[idx].x += p[idx].vx \* dt;
64. p[idx].y += p[idx].vy \* dt;
65. p[idx].z += p[idx].vz \* dt;
66. }
67. int main(const int argc, const char \*\*argv)
68. {
69. */\**
70. \* Do not change the value for `nBodies` here. If you would like to modify it,
71. \* pass values into the command line.
72. \*/
73. int nBodies = 2 << 11;
74. int salt = 0;
75. if (argc > 1)
76. nBodies = 2 << atoi(argv[1]);
77. */\**
78. \* This salt is for assessment reasons. Tampering with it will result in automatic failure.
79. \*/
80. if (argc > 2)
81. salt = atoi(argv[2]);
82. const float dt = 0.01f; *// time step*
83. const int nIters = 10;  *// simulation iterations*
84. int bytes = nBodies \* sizeof(Body);
85. float \*buf; cudaMallocManaged(&buf, bytes); Body \*p = (Body \*)buf;
86. randomizeBodies(buf, 6 \* nBodies); *// Init pos / vel data*
87. size\_t blockNum = (nBodies + BLK - 1) / BLK;
88. double totalTime = 0.0;
89. */\**
90. \* This simulation will run for 10 cycles of time, calculating gravitational
91. \* interaction amongst bodies, and adjusting their positions to reflect.
92. \*/
93. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
94. *// Do not modify these 2 lines of code.*
95. for (int iter = 0; iter < nIters; iter++)
96. {
97. StartTimer();
98. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
99. bodyForce<<<blockNum, BLK>>>(p, dt, nBodies); *// compute forces*
100. integrate\_position<<<blockNum, BLK>>>(p,dt,nBodies); *// update positions*
101. if(iter == nIters-1)cudaDeviceSynchronize(); *//sync memory*
102. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
103. *// Do not modify the code in this section.*
104. const double tElapsed = GetTimer() / 1000.0;
105. totalTime += tElapsed;
106. }
107. double avgTime = totalTime / (double)(nIters);
108. float billionsOfOpsPerSecond = 1e-9 \* nBodies \* nBodies / avgTime;
109. #ifdef ASSESS
110. checkPerformance(buf, billionsOfOpsPerSecond, salt);
111. #else
112. checkAccuracy(buf, nBodies);
113. printf("%d Bodies: average %0.3f Billion Interactions / second\n", nBodies, billionsOfOpsPerSecond);
114. salt += 1;
115. #endif
116. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
117. */\**
118. \* Feel free to modify code below.
119. \*/
120. cudaFree(buf);
121. }
122. nbody2.cu
123. #include <math.h>
124. #include <stdio.h>
125. #include <stdlib.h>
126. #include "timer.h"
127. #include "check.h"
128. #include <cuda\_runtime.h>
129. #define SOFTENING 1e-9f
130. #define BLK 64
131. #define STRIDE 32
132. typedef struct
133. {
134. float x, y, z, vx, vy, vz;
135. } Body;
136. void randomizeBodies(float \*data, int n)
137. {
138. for (int i = 0; i < n; i++)
139. {
140. data[i] = 2.0f \* (rand() / (float)RAND\_MAX) - 1.0f;
141. }
142. }
143. \_\_global\_\_ void bodyForce(Body \*p, float dt, int n)
144. {
145. *// index of updated body*
146. int idx = threadIdx.x + (blockIdx.x / STRIDE) \* blockDim.x;
147. Body pi = p[idx];
148. int startID = blockIdx.x % STRIDE;
149. int blockNum = n / BLK;
150. *// shared\_memory as caches*
151. \_\_shared\_\_ float3 caches[BLK];
152. float dx, dy, dz, distSqr, invDist, invDist3;
153. *// Resultant force on x,y,z axes*
154. float Fx=0, Fy=0, Fz=0;
155. *// iterate on bodies*
156. for (int currID = startID; currID < blockNum; currID += STRIDE)
157. {
158. Body tmp = p[currID \* BLK + threadIdx.x];
159. caches[threadIdx.x] = make\_float3(tmp.x, tmp.y, tmp.z);
160. \_\_syncthreads();
161. #pragma unroll
162. for (int j = 0; j < BLK; j++)
163. {
164. dx = caches[j].x - pi.x;
165. dy = caches[j].y - pi.y;
166. dz = caches[j].z - pi.z;
167. distSqr = dx \* dx + dy \* dy + dz \* dz + SOFTENING;
168. invDist = rsqrtf(distSqr);
169. invDist3 = invDist \* invDist \* invDist;
170. *// accumualte resultant force on x,y,z axes*
171. Fx += dx \* invDist3;
172. Fy += dy \* invDist3;
173. Fz += dz \* invDist3;
174. }
175. \_\_syncthreads();
176. }
177. *// update velocity on x,y,z axes*
178. *// concurrently with atomic ops*
179. atomicAdd(&p[idx].vx, dt \* Fx);
180. atomicAdd(&p[idx].vy, dt \* Fy);
181. atomicAdd(&p[idx].vz, dt \* Fz);
182. }
183. \_\_global\_\_ void integrate\_position(Body \*p, float dt, int n)
184. {
185. *// index of updated body*
186. int idx = threadIdx.x + blockIdx.x \* blockDim.x;
187. *// update position on x,y,z axes*
188. p[idx].x += p[idx].vx \* dt;
189. p[idx].y += p[idx].vy \* dt;
190. p[idx].z += p[idx].vz \* dt;
191. }
192. int main(const int argc, const char \*\*argv)
193. {
194. int nBodies = 2 << 11;
195. int salt = 0;
196. if (argc > 1)
197. nBodies = 2 << atoi(argv[1]);
198. */\**
199. \* This salt is for assessment reasons. Tampering with it will result in automatic failure.
200. \*/
201. if (argc > 2) salt = atoi(argv[2]);
202. const float dt = 0.01f; *// time step*
203. const int nIters = 10;  *// simulation iterations*
204. int bytes = nBodies \* sizeof(Body);
205. float \*buf; cudaMallocHost(&buf, bytes);
206. randomizeBodies(buf, 6 \* nBodies); *// Init pos / vel data*
207. double totalTime = 0.0;
208. size\_t blockNum = (nBodies + BLK - 1) / BLK;
209. float \*bufDev; cudaMalloc(&bufDev, bytes); Body \*pDev = (Body \*)bufDev;
210. */\**
211. \* This simulation will run for 10 cycles of time, calculating gravitational
212. \* interaction amongst bodies, and adjusting their positions to reflect.
213. \*/
214. cudaMemcpy(bufDev, buf, bytes, cudaMemcpyHostToDevice);
215. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
216. *// Do not modify these 2 lines of code.*
217. for (int iter = 0; iter < nIters; iter++)
218. {
219. StartTimer();
220. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
221. bodyForce<<<blockNum \* STRIDE, BLK>>>(pDev, dt, nBodies); *// compute interbody forces*
222. integrate\_position<<<blockNum, BLK>>>(pDev, dt, nBodies);
223. if (iter == nIters - 1)cudaMemcpy(buf, bufDev, bytes, cudaMemcpyDeviceToHost);
224. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
225. *// Do not modify the code in this section.*
226. const double tElapsed = GetTimer() / 1000.0;
227. totalTime += tElapsed;
228. }
229. double avgTime = totalTime / (double)(nIters);
230. float billionsOfOpsPerSecond = 1e-9 \* nBodies \* nBodies / avgTime;
231. #ifdef ASSESS
232. checkPerformance(buf, billionsOfOpsPerSecond, salt);
233. #else
234. checkAccuracy(buf, nBodies);
235. printf("%d Bodies: average %0.3f Billion Interactions / second\n", nBodies, billionsOfOpsPerSecond);
236. salt += 1;
237. #endif
238. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
239. */\**
240. \* Feel free to modify code below.
241. \*/
242. cudaFree(bufDev);
243. cudaFreeHost(buf);
244. }