

电子科技大学计算机科学与工程学院

实验报告

学 号 2020080903009

姓 名 李皓

(实验) 课程名称 CUDA 并行程序设计实验

教师 卢国明

电子科技大学

实验报告

学生姓名：李皓 学号：2020080903009 指导教师：卢国明

一、实验名称: N-Body 问题并程序设计及性能优化

二、实验目的:

- 1) 实现 N-body 的 cuda 并行算法。
- 2) 使用各种方法对程序进行调优。

三、实验原理:

3.1 N-Body 问题概述

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。天体模拟就是一个非常经典的 N-body 系统，根据牛顿的万有引力定律，宇宙中的不同天体之间会产生相互作用的吸引力，吸引力根据两个天体之间的质量和距离的不同而各不相同，一个天体的运动轨迹最终取决于剩下的所有的天体对该天体的引力的合力。

3.2 N-body 问题的 CUDA 实现

1) 串行算法

串行算法只使用了 CPU 串行计算粒子间引力并更新位置,还未引入 CUDA 使用 GPU 并行计算。首先，对每个点的位置和速度进行随机初始化。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。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。

2) 并行算法

由于在同一周期内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 高达数千核的并行计算特点，用 n 个线程并发计算速度的变化，每个线程处理一个粒子的计算任务，然后再用 n 个线程并发更新位置。在 CUDA 编程中，设备端指 GPU 端，数据存放在显存中；主机端指 CPU，数据存放在内存中。一般情况下，设备端是不能直接访问主机端内存的，而数据

通常情况下都是存放在主机端内存中，要在 GPU 中执行算法运算就必须先把数据拷贝至设备端，运算完成再把结果拷回至主机端。

四、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 并程序的性能优化

五、实验设备:

- 1) 操作系统: Windows 11 专业版
- 2) CPU: Ryzen 5-5700X
- 3) 编程环境: vscode
- 4) 执行环境: 实验室服务器集群
- 5) Node-CPU: Intel(R) Xeon(R) Gold 5318Y x2
- 6) GPU: Nvidia K80*2
- 7) 操作系统: CentOS 7.2
- 8) CUDA: 10.0
- 9) 内存: 64G

六、实验步骤

下面进行实验操作:

6.1 环境连接和配置

为了在 cuda 机上进行开发，因此需要连接到相应的机器上进行在线配置和链接。首先配置 ssh config 文件:

```
1 Host CUDA
2   HostName mpi-cu08-1
3   ProxyJump MPI
4   User 2020080903009
5
6 Host MPI
7   HostName 121.48.170.9
8   User 2020080903009
9   Port 10022
```

随后进行链接后可以直接打开远程工作区进行开发。如下图所示:

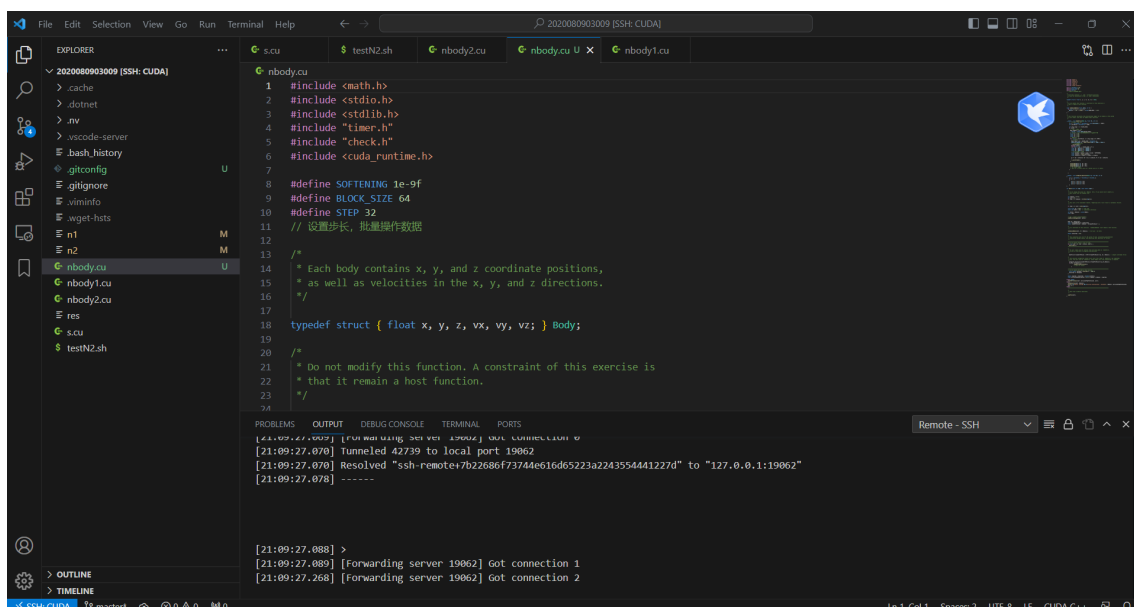


图 1: 远程开发示例

利用基本的代码进行性能和环境测试。首先编译代码：

```
1 2020080903009@mpi-cu08-1: nvcc -arch sm_37 -o s s.cu
```

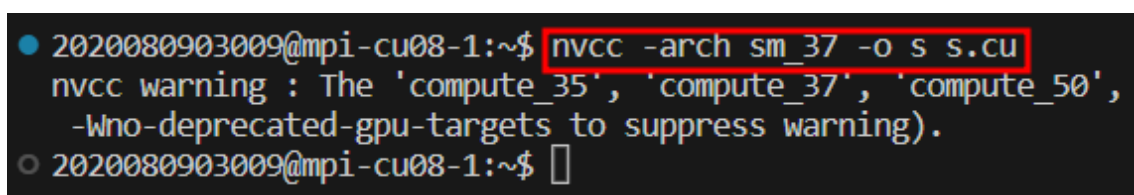


图 2: 代码编译运行结果

随后运行，测试代码运行的效果：

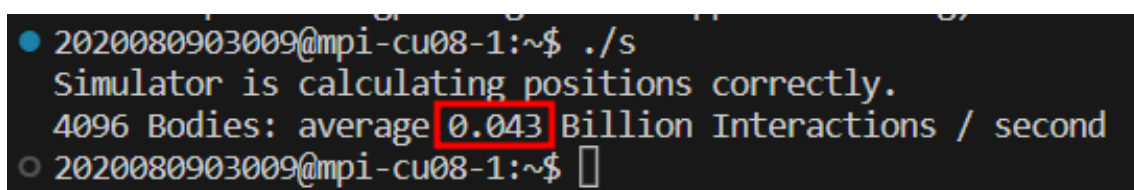


图 3: 基准代码测试结果

可见由于使用 CPU 来进行位置的模拟，只能做到有限的并行，因此程序在一定时间内作出的交互行为有限。接下来的目标就是利用 GPU 加速运行，使得程序加速。

6.2 并行化操作

在计算更新粒子位置的时候，需要根据粒子间的相对位置计算力的大小，得到各个方向的加速度 v_x, v_y, v_z ，随后更新当前的位置信息。显然，计算加速度的过程中能够并行进行计算，这是因为计算位置的行为没有结果的依赖。只需知道当前粒子的位置和相对的位置，根据万有引力公式计算出该对粒子产生出来的加速度。

为了对此过程进行优化，需要将计算距离的函数 `bodyForce` 函数，`integrate position` 函数进行修改，使之成为在 `cuda` 设备上计算运行的函数。为了存储结果，还需要在显存中分配空间存储粒子的状态信息。如下是分配显存的部分代码：

```
1 int bytes = nBodies * sizeof(Body);
2 float *buf;
3 // buf = (float *)malloc(bytes);
4 cudaMallocManaged(&buf, bytes);
5 Body *p = (Body*)buf;
```

使用 `cudaMalloc` 命令完成分配，随后分析代码的并行方法。由于每次位置迭代是所有粒子（4096 个）都参与运算，因此需要为每个粒子的位置迭代进行线程的分配。受限于显卡的并行架构设计，线程要以线程块的形式分批计算。随后规定进程块的大小和个数：

```
1 size_t threadsPerBlock = BLOCK_SIZE;
2 size_t numberOfBlocks = nBodies / threadsPerBlock + 1;
```

随后进行设备代码的修改，对于 `bodyForce` 代码而言，需要先计算出自身进程所对应的索引，这个行为是由块号和块内 ID 所参与计算的。随后迭代计算即可，该函数和原函数的差异之处在于将外层的循环进行了优化，使得 4096 个粒子都在并行计算。在调用函数的过程则传入了块数量和每块线程的线程数。代码修改如下：

```
1 // 调用函数
2 bodyForce<<<numberOfBlocks, threadsPerBlock>>>(p, dt, nBodies); // compute
   interbody forces
3 // 函数入口
4 __global__ void bodyForce(Body *p, float dt, int n) {
5     int i = threadIdx.x + blockDim.x * blockIdx.x;
6     // 取消了 i 的循环，计算每个进程对应的 i
7     float Fx = 0.0f;
8     float Fy = 0.0f;
9     float Fz = 0.0f;
10    if (i < n){
11
12        for (int j = 0; j < n; j++) {
13            float dx = p[j].x - p[i].x;
14            float dy = p[j].y - p[i].y;
15            float dz = p[j].z - p[i].z;
16            float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
17            float invDist = rsqrtf(distSqr);
18            float invDist3 = invDist * invDist * invDist;
19            Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
```

```

20     }
21
22     p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;
23 }
24 }

```

对于粒子的位置更新问题，由于加速度已经计算出来，因此直接计算更新后的位置即可。

6.3 自由优化

在完成基本的优化后，代码还有进行调整的空间。在计算各个粒子对之间产生的引力和加速度方面，还是使用了循环遍历所有的粒子。这部分可以进行并行化改进。可以通过分配更多的进程来完成计算，为此，引入变量 STEP, 每个 STEP 之间计算引力的行为可以并行操作。

引入 STEP 后对函数内的循环进行重构,使得每次循环都由原来的 $n/BLOCK_SIZE$ 次变为了 $n/(BLOCK_SIZE * STEP)$ 次。通过测试不同的 STEP, 可以有效加大代码的并行度，提高程序的执行效率。随着并行度的增加，对于加速度累计的情况，需要进行互斥相加的操作，使用函数 `atomicadd` 来实现。

该部分代码改动如下：

```

1  __global__ void bodyForce(Body *p, float dt, int n) {
2      // 计算力的过程并行化
3      int i = threadIdx.x + blockDim.x * (int)(blockIdx.x / STEP);
4      int startBlock = blockIdx.x % STEP;
5
6      int step_range = n / BLOCK_SIZE;
7      if (i < n){
8          // 设置标志记录 step
9          Body myBody = p[i];
10         __shared__ float3 bdPos[BLOCK_SIZE];
11         // float3 是三个 float 构成的数据类型，共享以访问
12         float Fx = 0.0f;
13         float Fy = 0.0f;
14         float Fz = 0.0f;
15         // 批量间串行：
16         for (int k = startBlock; k < step_range; k += STEP){
17             // 同步数据
18             Body temp = p[k * BLOCK_SIZE + threadIdx.x];
19             bdPos[threadIdx.x] = make_float3(temp.x, temp.y, temp.z);
20             // 阻塞等待数据同步
21             __syncthreads();
22             #pragma unroll
23             for (int j = 0; j < BLOCK_SIZE; j++) {
24                 float dx = bdPos[j].x - myBody.x;
25                 float dy = bdPos[j].y - myBody.y;
26                 float dz = bdPos[j].z - myBody.z;
27                 // 从自己的块进行访问
28                 float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
29                 float invDist = rsqrtf(distSqr);
30                 float invDist3 = invDist * invDist * invDist;
31                 Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
32             }
33             __syncthreads();
34         }

```

```

35
36     atomicAdd(&p[i].vx, dt * Fx);
37     atomicAdd(&p[i].vy, dt * Fy);
38     atomicAdd(&p[i].vz, dt * Fz);
39     // 并行化防止脏操作
40     // p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;
41 }
42 }

```

七、实验结果与分析:

7.1 并行化操作结果

编译运行并行化的程序后，得到以下结果：

```

● 2020080903009@mpi-cu08-1:~$ ./n1
Simulator is calculating positions correctly.
4096 Bodies: average 9.732 Billion Interactions / second
● 2020080903009@mpi-cu08-1:~$ ./n1
Simulator is calculating positions correctly.
4096 Bodies: average 9.736 Billion Interactions / second
● 2020080903009@mpi-cu08-1:~$ ./n1
Simulator is calculating positions correctly.
4096 Bodies: average 9.768 Billion Interactions / second

```

图 4: 并行化操作结果

和串行操作相比，每秒处理的运动迭代次数提升了 227 倍，没能够达到更高的提升倍数是因为并行操作的开销，以及函数内还有优化的空间。

7.2 自由优化结果

编译运行文件，得到以下结果：

```

● 2020080903009@mpi-cu08-1:~$ ./n2
Simulator is calculating positions correctly.
4096 Bodies: average 50.246 Billion Interactions / second
● 2020080903009@mpi-cu08-1:~$ ./n2
Simulator is calculating positions correctly.
4096 Bodies: average 50.763 Billion Interactions / second
● 2020080903009@mpi-cu08-1:~$ ./n2
Simulator is calculating positions correctly.
4096 Bodies: average 49.345 Billion Interactions / second

```

图 5: 自由优化后的运行结果

在之前的基础上通过分配更多的线程来完成并行计算的任务，速度提升了五倍。还使用了共享内存的方法来加速访问的效率，取得了较好效果。

八、总结及心得体会:

在本次实验中，我基本掌握了 CUDA 并程序程序设计的方法。并且通过两次程序的优化，直观认识了不同性能优化方法的效果以及特点。

经过这次实验我对并程序设计的优化有了新的认识，在后续的实践中可以对这些技巧灵活应用。


九、对本实验过程及方法、手段的改进建议:

在实验时需要对于 CUDA 的原理有更加深刻和直观的理解，需要在进行加强。

报告评分:

本人签字:

指导教师签字:



十、代码清单

1) 代码 I: CUDA 并行

```
1 #include <math.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "timer.h"
5 #include "check.h"
6 #include <cuda_runtime.h>
7
8 #define SOFTENING 1e-9f
9 #define BLOCK_SIZE 32
10
11 /*
12  * Each body contains x, y, and z coordinate positions,
13  * as well as velocities in the x, y, and z directions.
14  */
15
16 typedef struct { float x, y, z, vx, vy, vz; } Body;
17
18 /*
19  * Do not modify this function. A constraint of this exercise is
20  * that it remain a host function.
21  */
22
23 void randomizeBodies(float *data, int n) {
24     for (int i = 0; i < n; i++) {
25         data[i] = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
26     }
27 }
28
29 /*
30  * This function calculates the gravitational impact of all bodies in the
31  * system
32  * on all others, but does not update their positions.
33  */
34
35 __global__ void bodyForce(Body *p, float dt, int n) {
36     int i = threadIdx.x + blockDim.x * blockIdx.x;
37     float Fx = 0.0f;
38     float Fy = 0.0f;
39     float Fz = 0.0f;
40     if (i < n) {
41         for (int j = 0; j < n; j++) {
42             float dx = p[j].x - p[i].x;
43             float dy = p[j].y - p[i].y;
44             float dz = p[j].z - p[i].z;
45             float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
46             float invDist = rsqrtf(distSqr);
47             float invDist3 = invDist * invDist * invDist;
48
49             Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
50         }
51
52         p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;
53     }
54 }
55
56
57
58 __global__ void integrate_position(Body *p, float dt, int n)
59 {
60     int i = threadIdx.x + blockDim.x * blockIdx.x;
61     if (i < n)
62     {
63         p[i].x += p[i].vx * dt;
64         p[i].y += p[i].vy * dt;
65         p[i].z += p[i].vz * dt;
```

```

66     }
67 }
68
69 int main(const int argc, const char** argv) {
70
71     /*
72      * Do not change the value for 'nBodies' here. If you would like to modify
73      * it,
74      * pass values into the command line.
75      */
76     int nBodies = 2<<11;
77     int salt = 0;
78     if (argc > 1) nBodies = 2<<atoi(argv[1]);
79
80     /*
81      * This salt is for assessment reasons. Tampering with it will result in
82      * automatic failure.
83      */
84
85     if (argc > 2) salt = atoi(argv[2]);
86
87     const float dt = 0.01f; // time step
88     const int nIters = 10; // simulation iterations
89
90     int bytes = nBodies * sizeof(Body);
91     float *buf;
92
93     // buf = (float *)malloc(bytes);
94     cudaMallocManaged(&buf, bytes);
95
96
97     Body *p = (Body*)buf;
98     size_t threadsPerBlock = BLOCK_SIZE;
99     size_t numberOfBlocks = nBodies / threadsPerBlock + 1;
100
101     /*
102      * As a constraint of this exercise, 'randomizeBodies' must remain a host
103      * function.
104      */
105     randomizeBodies(buf, 6 * nBodies); // Init pos / vel data
106
107     double totalTime = 0.0;
108
109     /*
110      * This simulation will run for 10 cycles of time, calculating gravitational
111      * interaction amongst bodies, and adjusting their positions to reflect.
112      */
113
114     /* ***** */
115     // Do not modify these 2 lines of code.
116     for (int iter = 0; iter < nIters; iter++) {
117         StartTimer();
118         /* ***** */
119
120         /*
121          * You will likely wish to refactor the work being done in 'bodyForce',
122          * as well as the work to integrate the positions.
123          */
124
125         bodyForce<<<numberOfBlocks, threadsPerBlock>>>(p, dt, nBodies); // compute
            interbody forces
126
127         /*
128          * This position integration cannot occur until this round of 'bodyForce'
129          * has completed.
130          * Also, the next round of 'bodyForce' cannot begin until the integration is
131          * complete.
132          */

```

```

131     integrate_position<<<numberOfBlocks,threadsPerBlock>>>(p,dt,nBodies);
132     if(iter == nIters-1){
133         cudaDeviceSynchronize();
134         // wait for finish
135     }
136
137     /* ***** */
138     // Do not modify the code in this section.
139     const double tElapsed = GetTimer() / 1000.0;
140     totalTime += tElapsed;
141 }
142
143 double avgTime = totalTime / (double)(nIters);
144 float billionsOfOpsPerSecond = 1e-9 * nBodies * nBodies / avgTime;
145
146 #ifdef ASSESS
147     checkPerformance(buf, billionsOfOpsPerSecond, salt);
148 #else
149     checkAccuracy(buf, nBodies);
150     printf("%d Bodies: average %0.3f Billion Interactions / second\n", nBodies,
151           billionsOfOpsPerSecond);
152     salt += 1;
153 #endif
154 /* ***** */
155
156 /*
157  * Feel free to modify code below.
158  */
159
160     cudaFree(buf);
161 }

```

2) 代码 II: 优化

```

1  #include <math.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "timer.h"
5  #include "check.h"
6  #include <cuda_runtime.h>
7
8  #define SOFTENING 1e-9f
9  #define BLOCK_SIZE 64
10 #define STEP 32
11 // 设置步长, 批量操作数据
12
13 /*
14  * Each body contains x, y, and z coordinate positions,
15  * as well as velocities in the x, y, and z directions.
16  */
17
18 typedef struct { float x, y, z, vx, vy, vz; } Body;
19
20 /*
21  * Do not modify this function. A constraint of this exercise is
22  * that it remain a host function.
23  */
24
25 void randomizeBodies(float *data, int n) {
26     for (int i = 0; i < n; i++) {
27         data[i] = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
28     }
29 }
30
31 /*
32  * This function calculates the gravitational impact of all bodies in the
33  * system
34  * on all others, but does not update their positions.
35  */

```

```

36 __global__ void bodyForce(Body *p, float dt, int n) {
37     // 计算力的过程并行化
38     int i = threadIdx.x + blockDim.x * (int)(blockIdx.x / STEP);
39     int startBlock = blockIdx.x % STEP;
40
41     int step_range = n / BLOCK_SIZE;
42     if (i < n) {
43         // 设置标志记录 step
44         Body myBody = p[i];
45         __shared__ float3 bdPos[BLOCK_SIZE];
46         // float3 是三个float构成的数据类型，共享以访问
47         float Fx = 0.0f;
48         float Fy = 0.0f;
49         float Fz = 0.0f;
50         // 批量间串行：
51         for (int k = startBlock; k < step_range; k += STEP) {
52             // 同步数据
53             Body temp = p[k * BLOCK_SIZE + threadIdx.x];
54             bdPos[threadIdx.x] = make_float3(temp.x, temp.y, temp.z);
55             // 阻塞等待数据同步
56             __syncthreads();
57             #pragma unroll
58             for (int j = 0; j < BLOCK_SIZE; j++) {
59                 float dx = bdPos[j].x - myBody.x;
60                 float dy = bdPos[j].y - myBody.y;
61                 float dz = bdPos[j].z - myBody.z;
62                 // 从自己的块进行访问
63                 float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
64                 float invDist = rsqrtf(distSqr);
65                 float invDist3 = invDist * invDist * invDist;
66
67                 Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
68             }
69             __syncthreads();
70         }
71
72         atomicAdd(&p[i].vx, dt * Fx);
73         atomicAdd(&p[i].vy, dt * Fy);
74         atomicAdd(&p[i].vz, dt * Fz);
75         // 并行化防止脏操作
76         // p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;
77     }
78
79 }
80
81
82 __global__ void integrate_position(Body *p, float dt, int n)
83 {
84     int i = threadIdx.x + blockDim.x * blockIdx.x;
85     if (i < n)
86     {
87         p[i].x += p[i].vx * dt;
88         p[i].y += p[i].vy * dt;
89         p[i].z += p[i].vz * dt;
90     }
91 }
92
93 int main(const int argc, const char** argv) {
94
95     /*
96      * Do not change the value for 'nBodies' here. If you would like to modify
97      * it,
98      * pass values into the command line.
99      */
100     int nBodies = 2<<11;
101     int salt = 0;
102     if (argc > 1) nBodies = 2<<atoi(argv[1]);
103

```

```

104  /*
105  * This salt is for assessment reasons. Tampering with it will result in
      automatic failure.
106  */
107
108  if (argc > 2) salt = atoi(argv[2]);
109
110  const float dt = 0.01f; // time step
111  const int nIters = 10; // simulation iterations
112
113  int bytes = nBodies * sizeof(Body);
114  float *buf;
115
116
117  // buf = (float *)malloc(bytes);
118  cudaMallocManaged(&buf, bytes);
119
120
121  Body *p = (Body*)buf;
122  size_t threadsPerBlock = BLOCK_SIZE;
123  size_t numberOfBlocks = nBodies / threadsPerBlock + 1;
124
125  /*
126  * As a constraint of this exercise, 'randomizeBodies' must remain a host
      function.
127  */
128
129  randomizeBodies(buf, 6 * nBodies); // Init pos / vel data
130
131  double totalTime = 0.0;
132
133  /*
134  * This simulation will run for 10 cycles of time, calculating gravitational
135  * interaction amongst bodies, and adjusting their positions to reflect.
136  */
137
138  /* *****
139  // Do not modify these 2 lines of code.
140  for (int iter = 0; iter < nIters; iter++) {
141      StartTimer();
142  /* *****
143
144  /*
145  * You will likely wish to refactor the work being done in 'bodyForce',
146  * as well as the work to integrate the positions.
147  */
148
149      bodyForce<<<numberOfBlocks * STEP, threadsPerBlock>>>(p, dt, nBodies); //
          compute interbody forces
150
151  /*
152  * This position integration cannot occur until this round of 'bodyForce'
          has completed.
153  * Also, the next round of 'bodyForce' cannot begin until the integration is
          complete.
154  */
155      integrate_position<<<numberOfBlocks, threadsPerBlock>>>(p, dt, nBodies);
156      if(iter == nIters-1){
157          cudaDeviceSynchronize();
158          // wait for finish
159      }
160
161  /* *****
162  // Do not modify the code in this section.
163      const double tElapsed = GetTimer() / 1000.0;
164      totalTime += tElapsed;
165  }
166
167  double avgTime = totalTime / (double)(nIters);
168  float billionsOfOpsPerSecond = 1e-9 * nBodies * nBodies / avgTime;

```

```

169
170 #ifdef ASSESS
171     checkPerformance(buf, billionsOfOpsPerSecond, salt);
172 #else
173     checkAccuracy(buf, nBodies);
174     printf("%d Bodies: average %0.3f Billion Interactions / second\n", nBodies,
            billionsOfOpsPerSecond);
175     salt += 1;
176 #endif
177     /******
178
179     /*
180     * Feel free to modify code below.
181     */
182
183     cudaFree(buf);
184 }

```