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

三、实验原理：

N-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 $O(N)$ ，进而导致具有 N 个粒子的实例的单次计算具有 $O(N^2)$ 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$O(N^2)$。

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。

四、实验目的：

1. 学习 CUDA 程序在 Linux 操作系统下的编译、执行和调试方法。
2. 基于 CUDA 平台设计和实现并行 N-Body 算法。
3. 掌握 CUDA 程序的性能分析以及调优方法。

五、实验内容：

1. 学习和使用 CUDA 平台。
2. 基于 CUDA 环境，完成 N-Body 程序基准代码的并行化。
3. 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

七、实验步骤及操作：

1. 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。

```
std2019081308021@cu05:~$ ./ori
Simulator is calculating positions correctly.
4096 Bodies: average 0.034 Billion Interactions / second
```

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。

```
__global__ void bodyForce(Body *p, float dt, int n)
{
    // index of updated body
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    // Resultant force on x,y,z axes
    float Fx=0, Fy=0, Fz=0;
    // iterate on all the bodies
    for (int j = 0; j < n; j++)
    {
        float dx = p[j].x - p[idx].x;
        float dy = p[j].y - p[idx].y;
        float dz = p[j].z - p[idx].z;
        float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
        float invDist = rsqrtf(distSqr);
        float invDist3 = invDist * invDist * invDist;
        // accumualte resultant force on x,y,z axes
        Fx += dx * invDist3;
        Fy += dy * invDist3;
        Fz += dz * invDist3;
    }
    // update velocity on x,y,z axes
    p[idx].vx += dt * Fx;
    p[idx].vy += dt * Fy;
    p[idx].vz += dt * Fz;
}
```

图 2

(2) 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。

```
__global__ void integrate_position(Body *p, float dt, int n)
{
    // index of updated body
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    // update position on x,y,z axes
    p[idx].x += p[idx].vx * dt;
    p[idx].y += p[idx].vy * dt;
    p[idx].z += p[idx].vz * dt;
}
```

图 3

(3) 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。

```
int bytes = nBodies * sizeof(Body);
float *buf; cudaMallocHost(&buf, bytes);

randomizeBodies(buf, 6 * nBodies); // Init pos / vel data

double totalTime = 0.0;

size_t blockNum = (nBodies + BLK - 1) / BLK;

float *bufDev; cudaMalloc(&bufDev, bytes); Body *pDev = (Body *)bufDev;
/*
* This simulation will run for 10 cycles of time, calculating gravitatior
* interaction amongst bodies, and adjusting their positions to reflect.
*/

cudaMemcpy(bufDev, buf, bytes, cudaMemcpyHostToDevice);
```

图 4

(4) 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。

```
 * Feel free to modify code below.
 */
  cudaFree(bufDev);
  cudaFreeHost(buf);
```

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。

```
std2019081308021@cu05:~$ nvcc -o nv1 ./nbody1.cu
std2019081308021@cu05:~$ ./nv1
Simulator is calculating positions correctly.
4096 Bodies: average 52.039 Billion Interactions / second
std2019081308021@cu05:~$
```

图 6

2. 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

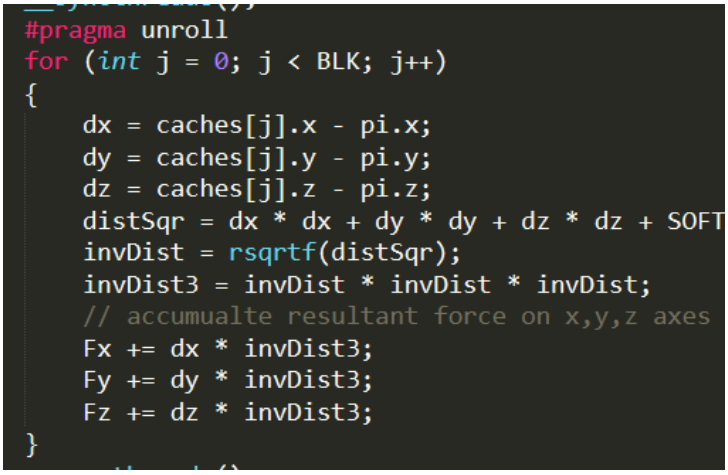
基于对 CUDA 系统执行模型和内存模型的理解，我们知道每个线程块内存在一块共享内存，其仅能被块内线程访问，但访问耗时远少于全局内存。利用“nvprof”分析并行化后的程序可以发现：bodyForce 函数消耗了程序绝大部分执行时间。通过进一步分析可知：每一次 bodyForce 函数的调用都意味着 O(n)的内存访问次数，而全局内存的大量访问会引起大量的时间开销。因此，我们引入 Shared Memory，即同一个线程块中使用 Shared Memory 缓存全局内存中的数据，从而有效降低内存访问开销。代码如图 7 所示。

```
// shared_memory as caches
__shared__ float3 caches[BLK];
float dx, dy, dz, distSqr, invDist, invDist3;
// Resultant force on x,y,z axes
float Fx=0, Fy=0, Fz=0;
// iterate on bodies
for (int currID = startID; currID < blockNum; currID += STRIDE)
{
    Body tmp = p[currID * BLK + threadIdx.x];
    caches[threadIdx.x] = make_float3(tmp.x, tmp.y, tmp.z);
    __syncthreads();
```

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。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。

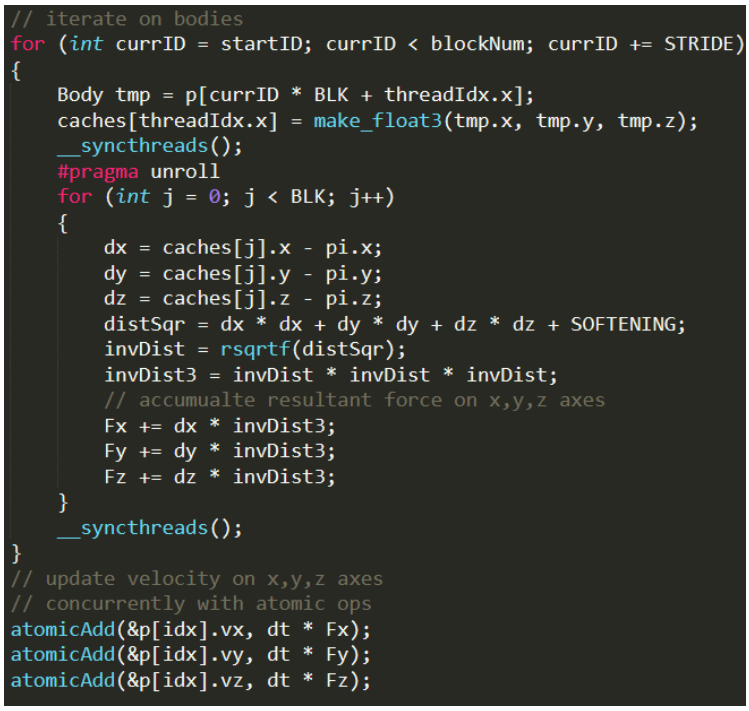


```
#pragma unroll
for (int j = 0; j < BLK; j++)
{
    dx = caches[j].x - pi.x;
    dy = caches[j].y - pi.y;
    dz = caches[j].z - pi.z;
    distSqr = dx * dx + dy * dy + dz * dz + SOFT
    invDist = rsqrtf(distSqr);
    invDist3 = invDist * invDist * invDist;
    // accumualte resultant force on x,y,z axes
    Fx += dx * invDist3;
    Fy += dy * invDist3;
    Fz += dz * invDist3;
}
```

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）



```
// iterate on bodies
for (int currID = startID; currID < blockNum; currID += STRIDE)
{
    Body tmp = p[currID * BLK + threadIdx.x];
    caches[threadIdx.x] = make_float3(tmp.x, tmp.y, tmp.z);
    __syncthreads();
    #pragma unroll
    for (int j = 0; j < BLK; j++)
    {
        dx = caches[j].x - pi.x;
        dy = caches[j].y - pi.y;
        dz = caches[j].z - pi.z;
        distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
        invDist = rsqrtf(distSqr);
        invDist3 = invDist * invDist * invDist;
        // accumualte resultant force on x,y,z axes
        Fx += dx * invDist3;
        Fy += dy * invDist3;
        Fz += dz * invDist3;
    }
    __syncthreads();
}
// update velocity on x,y,z axes
// concurrently with atomic ops
atomicAdd(&p[idx].vx, dt * Fx);
atomicAdd(&p[idx].vy, dt * Fy);
atomicAdd(&p[idx].vz, dt * Fz);
```

图 9

八、实验数据及结果分析：

1. 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。

```
std2019081308021@cu05:~$ nvcc -o nv1 ./nbody1.cu
std2019081308021@cu05:~$ ./nv1
Simulator is calculating positions correctly.
4096 Bodies: average 52.039 Billion Interactions / second
std2019081308021@cu05:~$
```

图 10

2. 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。

```
std2019081308021@cu05:~$ nvcc -o nv2 ./nbody2.cu
std2019081308021@cu05:~$ ./nv2
Simulator is calculating positions correctly.
4096 Bodies: average 201.407 Billion Interactions / second
std2019081308021@cu05:~$
```

图 11

九、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。

十、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！

王 正 仁

十一、对本实验过程及方法、手段的改进建议：

1. 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。
2. 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：

一、 nbody1.cu

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "timer.h"
#include "check.h"
#include <cuda_runtime.h>

#define SOFTENING 1e-9f
#define BLK 32

/*
 * Each body contains x, y, and z coordinate positions,
 * as well as velocities in the x, y, and z directions.
 */

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;
    }
}

/*
 * This function calculates the gravitational impact of all bodies in the system
 * on all others, but does not update their positions.
 */

__global__ void bodyForce(Body *p, float dt, int n)
```

```
40.{
41.    // index of updated body
42.    int idx = threadIdx.x + blockDim.x * blockIdx.x;
43.    // Resultant force on x,y,z axes
44.    float Fx=0, Fy=0, Fz=0;
45.    // iterate on all the bodies
46.    for (int j = 0; j < n; j++)
47.    {
48.        float dx = p[j].x - p[idx].x;
49.        float dy = p[j].y - p[idx].y;
50.        float dz = p[j].z - p[idx].z;
51.        float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
52.        float invDist = rsqrtf(distSqr);
53.        float invDist3 = invDist * invDist * invDist;
54.        // accumualte resultant force on x,y,z axes
55.        Fx += dx * invDist3;
56.        Fy += dy * invDist3;
57.        Fz += dz * invDist3;
58.    }
59.    // update velocity on x,y,z axes
60.    p[idx].vx += dt * Fx;
61.    p[idx].vy += dt * Fy;
62.    p[idx].vz += dt * Fz;
63.}
64.
65.__global__ void integrate_position(Body *p, float dt, int n)
66.{
67.    // index of updated body
68.    int idx = threadIdx.x + blockDim.x * blockIdx.x;
69.    // update position on x,y,z axes
70.    p[idx].x += p[idx].vx * dt;
71.    p[idx].y += p[idx].vy * dt;
72.    p[idx].z += p[idx].vz * dt;
73.}
74.
75.int main(const int argc, const char **argv)
76.{
77.
78.    /*
79.    * Do not change the value for `nBodies` here. If you would lik
   e to modify it,
80.    * pass values into the command line.
81.    */
82.
```

```
83.     int nBodies = 2 << 11;
84.     int salt = 0;
85.     if (argc > 1)
86.         nBodies = 2 << atoi(argv[1]);
87.
88.     /*
89.    * This salt is for assessment reasons. Tampering with it will
   result in automatic failure.
90.    */
91.
92.     if (argc > 2)
93.         salt = atoi(argv[2]);
94.
95.     const float dt = 0.01f; // time step
96.     const int nIters = 10;  // simulation iterations
97.
98.     int bytes = nBodies * sizeof(Body);
99.     float *buf; cudaMallocManaged(&buf, bytes); Body *p = (Body *
   )buf;
100.     randomizeBodies(buf, 6 * nBodies); // Init pos / vel data
101.
102.     size_t blockNum = (nBodies + BLK - 1) / BLK;
103.     double totalTime = 0.0;
104.
105.     /*
106.    * This simulation will run for 10 cycles of time, calculating
    gravitational
107.    * interaction amongst bodies, and adjusting their positions t
   o reflect.
108.    */
109.
110.     /*************************************************************
   ********/
111.     // Do not modify these 2 lines of code.
112.     for (int iter = 0; iter < nIters; iter++)
113.     {
114.         StartTimer();
115.     /*************************************************************
   ********/
116.         bodyForce<<<blockNum, BLK>>>(p, dt, nBodies); // compute
    forces
117.         integrate_position<<<blockNum, BLK>>>(p,dt,nBodies); //
   update positions
```

```
        if(iter == nIters-1)cudaDeviceSynchronize(); //sync memory
    /*********************************************************************/
    // Do not modify the code in this section.
        const double tElapsed = GetTimer() / 1000.0;
        totalTime += tElapsed;
    }

    double avgTime = totalTime / (double)(nIters);
    float billionsOfOpsPerSecond = 1e-9 * nBodies * nBodies / avgTime;

#ifdef ASSESS
    checkPerformance(buf, billionsOfOpsPerSecond, salt);
#else
    checkAccuracy(buf, nBodies);
    printf("%d Bodies: average %0.3f Billion Interactions / second\n", nBodies, billionsOfOpsPerSecond);
    salt += 1;
#endif
    /*********************************************************************/

    /*
   * Feel free to modify code below.
   */
    cudaFree(buf);
}
```

二、 nbody2.cu

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "timer.h"
#include "check.h"
#include <cuda_runtime.h>

#define SOFTENING 1e-9f
#define BLK 64
#define STRIDE 32

typedef struct
{
```

```
14.     float x, y, z, vx, vy, vz;
15. } Body;
16. 
17. void randomizeBodies(float *data, int n)
18. {
19.     for (int i = 0; i < n; i++)
20.     {
21.         data[i] = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
22.     }
23. }
24. 
25. __global__ void bodyForce(Body *p, float dt, int n)
26. {
27. 
28.     // index of updated body
29.     int idx = threadIdx.x + (blockIdx.x / STRIDE) * blockDim.x;
30.     Body pi = p[idx];
31.     int startID = blockIdx.x % STRIDE;
32.     int blockNum = n / BLK;
33.     // shared_memory as caches
34.     __shared__ float3 caches[BLK];
35.     float dx, dy, dz, distSqr, invDist, invDist3;
36.     // Resultant force on x,y,z axes
37.     float Fx=0, Fy=0, Fz=0;
38.     // iterate on bodies
39.     for (int currID = startID; currID < blockNum; currID += STRID
    E)
40.     {
41.         Body tmp = p[currID * BLK + threadIdx.x];
42.         caches[threadIdx.x] = make_float3(tmp.x, tmp.y, tmp.z);
43.         __syncthreads();
44.         #pragma unroll
45.         for (int j = 0; j < BLK; j++)
46.         {
47.             dx = caches[j].x - pi.x;
48.             dy = caches[j].y - pi.y;
49.             dz = caches[j].z - pi.z;
50.             distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
51.             invDist = rsqrtf(distSqr);
52.             invDist3 = invDist * invDist * invDist;
53.             // accumualte resultant force on x,y,z axes
54.             Fx += dx * invDist3;
55.             Fy += dy * invDist3;
56.             Fz += dz * invDist3;
```

```
57.         }
58.         __syncthreads();
59.     }
60.     // update velocity on x,y,z axes
61.     // concurrently with atomic ops
62.     atomicAdd(&p[idx].vx, dt * Fx);
63.     atomicAdd(&p[idx].vy, dt * Fy);
64.     atomicAdd(&p[idx].vz, dt * Fz);
65.}
66.
67.__global__ void integrate_position(Body *p, float dt, int n)
68.{
69.     // index of updated body
70.     int idx = threadIdx.x + blockIdx.x * blockDim.x;
71.     // update position on x,y,z axes
72.     p[idx].x += p[idx].vx * dt;
73.     p[idx].y += p[idx].vy * dt;
74.     p[idx].z += p[idx].vz * dt;
75.}
76.
77.int main(const int argc, const char **argv)
78.{
79.
80.     int nBodies = 2 << 11;
81.     int salt = 0;
82.     if (argc > 1)
83.         nBodies = 2 << atoi(argv[1]);
84.
85.     /*
86.    * This salt is for assessment reasons. Tampering with it will
   result in automatic failure.
87.    */
88.
89.     if (argc > 2) salt = atoi(argv[2]);
90.
91.     const float dt = 0.01f; // time step
92.     const int nIters = 10;  // simulation iterations
93.
94.     int bytes = nBodies * sizeof(Body);
95.     float *buf; cudaMallocHost(&buf, bytes);
96.
97.     randomizeBodies(buf, 6 * nBodies); // Init pos / vel data
98.
99.     double totalTime = 0.0;
```

```
100.
101.     size_t blockNum = (nBodies + BLK - 1) / BLK;
102.
103.     float *bufDev; cudaMalloc(&bufDev, bytes); Body *pDev = (Bod
    y *)bufDev;
104.     /*
105.    * This simulation will run for 10 cycles of time, calculating
     gravitational
106.    * interaction amongst bodies, and adjusting their positions t
    o reflect.
107.    */
108.
109.     cudaMemcpy(bufDev, buf, bytes, cudaMemcpyHostToDevice);
110.     /*************************************************************
    ********/
111.     // Do not modify these 2 lines of code.
112.     for (int iter = 0; iter < nIters; iter++)
113.     {
114.         StartTimer();
115.     /*************************************************************
    ********/
116.         bodyForce<<<blockNum * STRIDE, BLK>>>(pDev, dt, nBodies)
    ; // compute interbody forces
117.         integrate_position<<<blockNum, BLK>>>(pDev, dt, nBodies)
    ;
118.         if (iter == nIters - 1)cudaMemcpy(buf, bufDev, bytes, cu
    daMemcpyDeviceToHost);
119.     /*************************************************************
    ********/
120.     // Do not modify the code in this section.
121.         const double tElapsed = GetTimer() / 1000.0;
122.         totalTime += tElapsed;
123.     }
124.
125.     double avgTime = totalTime / (double)(nIters);
126.     float billionsOfOpsPerSecond = 1e-9 * nBodies * nBodies / av
    gTime;
127.
128. #ifdef ASSESS
129.     checkPerformance(buf, billionsOfOpsPerSecond, salt);
130. #else
131.     checkAccuracy(buf, nBodies);
132.     printf("%d Bodies: average %0.3f Billion Interactions / seco
    nd\n", nBodies, billionsOfOpsPerSecond);
```

```
133.     salt += 1;
134. #endif
135.     /*************************************************************
    ********/
136. 
137.     /*
138.    * Feel free to modify code below.
139.    */
140.     cudaFree(bufDev);
141.     cudaFreeHost(buf);
142. }
```