电 子 科 技 大 学

实 验 报 告

学生姓名：董文龙	学 号：2018081309003

一、实验室名称：立人楼 B209

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：

3.1 概论

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 基于 CUDA 的天体的 N-body

1.CUDA 是一个 NVIDIA 的 GPU 编程模型，采用 CUDA 实现一个 N-body 模拟系统，可以充分利用 GPU 的并行能力去加速 N-body 的巨额计算过程。

2.一种最简单的求解 N-body 的方法就是暴力法，直接计算一个粒子体与剩下的所有粒子体的相互作用，对每一个粒子都做类似的处理，这样可以确保每个粒子体都与剩下的所有粒子体都产生交互作用，这种算法的复杂度非常高，达到了 O(N2)量级，对于非常庞大的 N-body 系统，会非常耗时，具体的优化措施会在实验步骤中进行描述。

3.模拟万有引力：给定 N 个天体，我们记每个天体 0≤i<N 的位置向量为 xi、速度向量为 vi、质量为 mi，根据牛顿的万有引力定律，任意两个不同天体 i 和 j 之间的万有引力计算公式如下所示(已经引入了矢量的概念)：

$$f_{ij} = G\frac{m_i m_j}{\|r_{ij}\|^2}\cdot\frac{r_{ij}}{\|r_{ij}\|}$$

上述公式描述的是天体 j 对天体 i 的引力，那么除 i 之外的所有天体对天体 i 的引力合力的计算公式为：

$$F_i = \Sigma_{0\le j<N, j\neq i} f_{ij} = Gm_i\cdot\Sigma_{0\le j<N, j\neq i}\frac{m_j r_{ij}}{\|r_{ij}\|^3}$$

但这个公式有个潜在的不稳定问题，当两个粒子之间的距离向量 rij→0 时，||rij||取值为 0，从而导致除零错误，在计算机系统中这将导致程序崩溃。为了避免这种情况，公式的分母加上一个软化因子（softening factor）ε>0，从而万有引力近似计算公式为：

$$F_i \approx Gm_i\cdot\Sigma_{0\le j<N}\frac{m_j r_{ij}}{\left(\|r_{ij}\|^2+\varepsilon\right)^{3/2}}$$

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）：

1. 计算节点配置：CPU E5-2660 v4*2，Nvidia K80*2
2. 服务器操作系统：CentOS 7.2，CUDA：10.0
3. 本地电脑系统：Windows 10
4. SSH 工具：XShell，Xftp
5. IDE：sublime Text3

七、实验步骤及操作：

1. 使用远程连接工具通过 ssh 方式登录集群。

2. 编译运行 nBodyOriginal.cu 文件，实验结果见实验数据及分析 nBodyOriginal 模块。

由于代码主要部分为 bodyForce 和后面的 integratePosition，后面也主要对其进行并行的优化，故对其进行详细分析，代码分析：

```
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;
  }
}
```

```
  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;
  }
```

代码中的 rsqrtf 代表求开方的倒数，SOFTENGINE 就是其中的软化因子，代码中默认 Gm 为 1,就可以得到实验原理中的万有引力公式。F=ma，所以 a 和 F 的大小和方向一致，此方法用的就是串行的暴力解法，时间复杂度为 O(N2)，取极短的时间 dt，dx=vdt，于是一个物体的三维坐标在其他粒子的影响下就会进行改变。

3. 编译运行 nBodyParallel.cu 文件，实验结果见实验数据及分析 nBodyParallel 模块。

代码分析（CUDA 编程基本步骤）：

1. 分配内存：

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

cudaMallocHost(&buf, bytes);
cudaMalloc(&d_buf, bytes);
Body *d_p = (Body *)d_buf;
```

2. 数据传输从 CPU 到 GPU：

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

3. 加载核函数

```
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;
  }
}
```

```
__global__ void integratePosition(Body *p, float dt, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n){
        p[i].x += p[i].vx * dt;
        p[i].y += p[i].vy * dt;
        p[i].z += p[i].vz * dt;
    }
}
```

4. 数据传输从 GPU 到 CPU

```
if(iter == nIters-1){
    cudaMemcpy(buf, d_buf, bytes, cudaMemcpyDeviceToHost);
}
```

5. 释放内存

```
cudaFree(d_buf);
cudaFreeHost(buf);
```

4. 编译运行 nBodyFinalOptimization.cu 文件，实验结果见实验数据及分析 nBodyFinalOptimization 模块。

优化思路：

1.采用 nvprof ./nBodyParallel 命令，发现 GPU 的很大时间都浪费在了 bodyForce 上面，所以该函数是主要优化的目标，这很大程度上由有全局内存的访问时间过慢而导致的，所以我们可以引入共享内存，来加快访存速度。

2.共享内存仅限于同一个线程块内，不同线程块之间的线程无法进行共享内存。因为共享内存仅限于同一个线程块，所以就需要进行分块，同一个线程块的线程访问这个共享内存做相应的数值计算。

```
          Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.32%  13.852ms        10  1.3852ms  1.3726ms  1.3970ms  bodyForce(Body*, float, int)
                   0.43%  60.317us        10  6.0310us  5.8230us  6.7840us  integratePosition(Body*, float, int)
                   0.15%  20.256us         1  20.256us  20.256us  20.256us  [CUDA memcpy HtoD]
                   0.10%  14.240us         1  14.240us  14.240us  14.240us  [CUDA memcpy DtoH]
      API calls:   87.48%  148.97ms         1  148.97ms  148.97ms  148.97ms  cudaHostAlloc
                   8.18%  13.935ms         2  6.9673ms  58.873us  13.876ms  cudaMemcpy
                   2.07%  3.5204ms       384  9.1670us     175ns  374.32us  cuDeviceGetAttribute
                   1.40%  2.3840ms         4  596.00us  364.07us  683.02us  cuDeviceTotalMem
                   0.23%  388.38us         1  388.38us  388.38us  388.38us  cudaFreeHost
                   0.17%  290.38us        20  14.519us  6.0890us  160.35us  cudaLaunchKernel
                   0.17%  289.49us         4  72.372us  66.030us  78.293us  cuDeviceGetName
                   0.15%  248.49us         1  248.49us  248.49us  248.49us  cudaFree
                   0.14%  244.64us         1  244.64us  244.64us  244.64us  cudaMalloc
                   0.01%  8.9850us         4  2.2460us  1.2690us  3.4980us  cuDeviceGetPCIBusId
                   0.00%  4.5300us         8     566ns     183ns  2.4880us  cuDeviceGet
                   0.00%  1.5700us         3     523ns     256ns     779ns  cuDeviceGetCount
                   0.00%  1.1170us         4     279ns     223ns     396ns  cuDeviceGetUuid
```

3.bodyForce 函数中的 invDist3 数值由乘除计算得出，再计算后被加到 Fx，Fy，Fz 上。然而 Fx，Fy，Fz 本身未参与乘除运算，只是每次在求那个物体的合力。所有循环完成后最后将其加到 body[i] 的受力上。所以对 body_force 函数进行拆分，将单个物体的受力情况交给多个线程去计算，而不是现在的一个线程负责一个位置的物体受力情况。

优化的代码如下，（这里以经过多次测试后的 BLOCK_SIZE=256,BLOCK_REDU=32，nBodies=4096 为最优结果进行分析）

```
__global__ void bodyForce(Body *p, float dt, int n) {
  int index= threadIdx.x + blockDim.x * blockIdx.x;
  int num=index/n;
  __shared__ float tempx[BLOCK_SIZE / BLOCK_REDO], tempy[BLOCK_SIZE / BLOCK_REDO], tempz[BLOCK_SIZE / BLOCK_REDO];
  float Fx = 0.0f; float Fy = 0.0f;float Fz = 0.0f;
  float px0 = p[index % n].x, py0 = p[index % n].y, pz0 = p[index % n].z;
  //share
  for (unsigned int i = 0; i < n; i += BLOCK_SIZE){
      if (threadIdx.x % BLOCK_REDO == num){
      tempx[threadIdx.x / BLOCK_REDO] = p[i+threadIdx.x].x;
      tempy[threadIdx.x / BLOCK_REDO] = p[i+threadIdx.x].y;
      tempz[threadIdx.x / BLOCK_REDO] = p[i+threadIdx.x].z;
    }
    __syncthreads();//时钟同步

  for (unsigned int  j = 0; j < BLOCK_SIZE / BLOCK_REDO; ++j) {
      float dx = ( tempx[j] - px0 );
      float dy = ( tempy[j] - py0 );
      float dz = ( tempz[j] - pz0 );
      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;
    }
  __syncthreads();//时钟同步
  }
  //各个方向做原子性加法
  atomicAdd(&p[index % n].vx, dt*Fx);
  atomicAdd(&p[index % n].vy, dt*Fy);
  atomicAdd(&p[index % n].vz, dt*Fz);
}
```

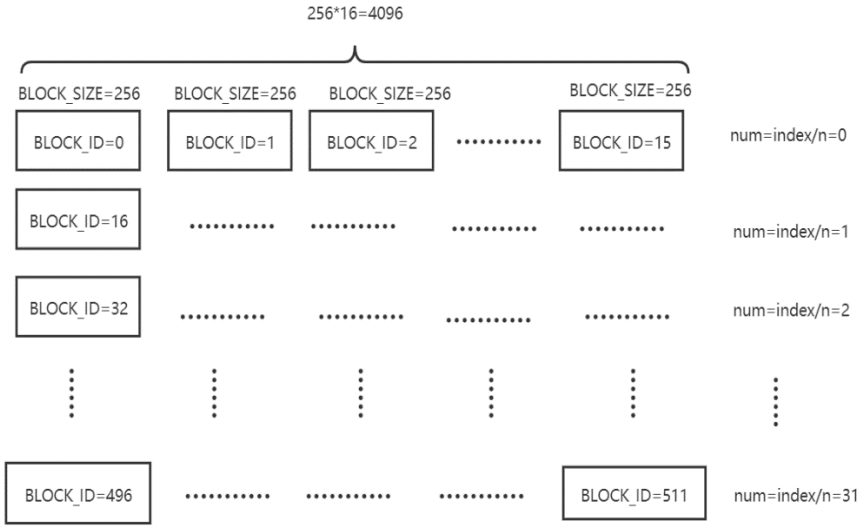
特别说明，

1． 在执行这个函数的时候，传入的块数为 BLOCK_REDO*nBodies/BLOCK_SIZE =32*4096/256=32*16=512，传入的每块的线程为 BLOCK_SIZE 为 256，那么传入的总线程数为 512*256=131072。

2． 创建的共享内存空间为：

tempx[BLOCK_SIZE/BLOCK_REDO]=tempx[8]
tempy[BLOCK_SIZE/BLOCK_REDO]=tempy[8]
tempz[BLOCK_SIZE/BLOCK_REDO]=tempz[8]

3． 下图为示意图：



4.

```
float px0 = p[index % n].x, py0 = p[index % n].y, pz0 = p[index % n].z;
```

对于 index=0,4096,8192，….,131072 存放的都是同一个物体的地址。

同理，index=1,4097,8193，…存放的都是同一个物体的地址。

也就是每一横排的线程存放的是不同物体的位置，每一竖排的线程存放的是同一物体的线程

5. 接下来，读数到共享内存并进行计算

第一轮，从 BLOCK_ID=0,threadID.x=0,32,64,96,128,160,192,224

取数到内存：temp[0]=p[0],temp[1]=p[32],....,temp[7]=224

取数计算万有引力，依次取出 temp[0],temp[1],...,temp[7]与 px0 进行计算。

与此同时,从 BLOCK_ID=16, threadID.x=1,33,65,97,129,161,193,225

取数到内存：temp[0]=p[1],temp[1]=p[33],....,temp[7]=225

第二轮，BLOCK_ID=1,threadID.x=0,32,64,96,128,160,192,224

取数到内存，temp[0]=p[256],temp[1]=[288],temp[7]=480

取数计算万有引力，依次取出 temp[0],temp[1],....,temp[7]与 px0 进行计算，如此往复直到计算完

与此同时，从 BLOCK_ID=17, threadID.x=1,33,65,97,129,161,193,225

取数到内存：temp[0]=p[257],temp[1]=p[289],....,temp[7]=481

依次类推，所有物体的万有引力都会被计算出来。

6. 单个物体受力由多线程计算做原子性加法

```
atomicAdd(&p[index % n].vx, dt*Fx);
atomicAdd(&p[index % n].vy, dt*Fy);
atomicAdd(&p[index % n].vz, dt*Fz);
```

八、实验数据及结果分析：

1.nBodyOriginal 模块：

```
[a2018081309003@cu08 ~]$ ./nBodyOriginal
Simulator is calculating positions correctly.
4096 Bodies: average 0.032 Billion Interactions / second
```

最初始的串行代码，在 4096 粒子的情况下，每秒钟大概有 0.032Billion 交互。

2.nBodyParallel 模块：

```
[a2018081309003@cu08 ~]$ ./nBodyParallel
Simulator is calculating positions correctly.
4096 Bodies: average 11.847 Billion Interactions / second
```

经过并行化的代码，在 4096 粒子的情况下，每秒钟大概有 11.847Billion 交互。

3.nBodyFinalOptimization 模块：

```
[a2018081309003@cu08 ~]$  ./nBodyFinalOptimization
Simulator is calculating positions correctly.
4096 Bodies: average 41.344 Billion Interactions / second
```

经过优化的代码，在 4096 粒子的情况下，每秒钟大概有 41.344Billion 交互

通过指令可以发现：

```
==25632== Profiling application: ./nBodyFinalOptimization
==25632== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   97.16%  3.7609ms        10  376.09us  374.75us  378.75us  bodyForce(Body*, float, int)
                    1.97%  76.287us        10  7.6280us  7.4560us  8.1600us  integratePosition(Body*, float
, int)
                    0.50%  19.328us         1  19.328us  19.328us  19.328us  [CUDA memcpy HtoD]
                    0.37%  14.240us         1  14.240us  14.240us  14.240us  [CUDA memcpy DtoH]
      API calls:   92.67%  139.76ms         1  139.76ms  139.76ms  139.76ms  cudaHostAlloc
                    2.54%  3.8373ms         2  1.9187ms  58.282us  3.7790ms  cudaMemcpy
                    2.06%  3.1112ms       384  8.1010us     178ns  318.26us  cuDeviceGetAttribute
                    1.76%  2.6600ms         4  665.00us  658.93us  671.97us  cuDeviceTotalMem
                    0.26%  391.18us         1  391.18us  391.18us  391.18us  cudaFreeHost
                    0.20%  306.67us        20  15.333us  6.7300us  159.87us  cudaLaunchKernel
                    0.17%  260.86us         4  65.215us  62.401us  70.379us  cuDeviceGetName
                    0.16%  237.82us         1  237.82us  237.82us  237.82us  cudaFree
                    0.16%  237.22us         1  237.22us  237.22us  237.22us  cudaMalloc
                    0.01%  8.1480us         4  2.0370us  1.2090us  3.1290us  cuDeviceGetPCIBusId
                    0.00%  4.5220us         8     565ns     226ns  2.0640us  cuDeviceGet
                    0.00%  1.5830us         3     527ns     221ns     790ns  cuDeviceGetCount
                    0.00%  1.0500us         4     262ns     198ns     349ns  cuDeviceGetUuid
[a2018081309003@cu08 ~]$
```

优化侯 bodyForce 的时间得到了极大的缩短。

九、实验结论：

1. 使用 CUDA 编程环境可以很好的实现 N-Body 并行算法。
2. Nbody 的并行化算法会比串行执行的性能高出很多。
3. CUDA 程序进行性能分析以及调优方法的共享内存，对于大部分场景都很适用。

十、总结及心得体会：

1. 掌握了 CUDA 编程的魅力，尤其是面对庞大的计算能力。
2. 可以采用 nvprof 指令观察程序的函数执行情况和 API 调用情况。
3. 可以采用共享内存进行程序的优化。
4. 分块的时候需要考虑的东西很多，算术能力要求较高。

十一、对本实验过程及方法、手段的改进建议：

无。

报告评分：

学生签字：董文龙

指导教师签字：