电 子 科 技 大 学

实 验 报 告

学生姓名：张镕麒	学 号：2019081301026

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（多体问题）是天体力学和一般力学的基本问题之一，研究 N 个质点相互之间作用的运动规律，它切合当前科学前沿，在高性能计算领域也具有一定的代表性。N-body 方法有几种，常用的算法设计思想比如 PP(Particle-Particle)、PM(Partical-Mesh)、BH(Barnes-Hut)、FMM(Fast Multipole Method)等。其中 PP 算法是两层循环求粒子间的作用力产生的位置变化，其时间复杂度为 O（N^2），PM、BH 算法是对 PP 算法的改进与创新，其时间复杂度是 O（NlogN），而 FMM 算法则更是将时间复杂度降到 O（N）。在上述算法中，选择 PP 算法对该 N-body 问题进行编程获得解决方案。PP 算法是两层循环求出粒子间的作用力，然后根据作用力大小，对其位置进行改动，具体步骤为：

1. 对粒子进行受力分析，将空间分成 x、y、z 三个方向，求出两粒子在万有引力作用下下，其加速度在 x、y、z 方向的大小。

2. 进行通过累加后的各方向的加速度，更新其速度以及空间位置。

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. 优化 N-Body 并行程序性能。

六、实验器材（设备、元器件）：

CPU E5-2660 v4*2
Nvidia K80*2
操作系统：CentOS 7.2
CUDA：10.0

七、实验步骤及操作：

1. 设计内核函数完成并行化

观察原串行代码可知，可并行化部分由两部分组成，分别为 O(N^2)的加速度计算和 O(N)的位置更新。

对于加速度计算部分：分配 N 个线程，每个线程 i 负责计算对应物体 i 的加速度，即遍历 N 个物体，获取其位置并计算对物体 i 加速度的贡献加入到速度中，时间复杂度为 O(N)。

```
__global__ void bodyForce(Body *p, float dt, int n) {
    int i=threadIdx.x+blockIdx.x*blockDim.x;
    if(i<n)
    {
        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;
            p[i].vx += dt*dx*invDist3;
            p[i].vy += dt*dy*invDist3;
            p[i].vz += dt*dz*invDist3;
        }
    }
}
```

对于位置更新部分：分配 N 个线程，每个线程 i 负责更新物体 i 经过 dt 后的新位置。时

间复杂度为 O(1)。

```
__global__ void integrate_position(Body *p,float dt,int n)
{
    int i=threadIdx.x+blockIdx.x*blockDim.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;
    }
}
```

此外，对于 cuda 程序，需要在运算前申请 CPU 与 GPU 的空间，并将 CPU 中的原始数据拷贝到 GPU 的全局内存中，并在运算结束后将数据从 GPU 拷贝到 CPU 中。此外在调用 Kernal 函数时还需要给定每个 BLOCK 的线程数量与 BLOCK 个数。

```
int threadNum=64;
int blockNum=(nBodies+threadNum-1)/threadNum;
int bytes = nBodies * sizeof(Body);
float *buf=NULL,*d_buf=NULL;
cudaMallocHost((void**)&buf,bytes);
cudaMalloc((void**)&d_buf,bytes);
Body *d_p=(Body*)d_buf,*p=(Body*)buf;
randomizeBodies(buf, 6 * nBodies); // Init pos / vel data
cudaMemcpy(d_buf,buf,bytes,cudaMemcpyHostToDevice);

bodyForce<<<blockNum,threadNum>>>(d_p, dt, nBodies); // compute interbody forces
integrate_position<<<blockNum,threadNum>>>(d_p,dt,nBodies);
```

至此，程序已成功并行化。

2. 优化一：tiling 优化

由于在并行代码计算加速度函数中，每个线程需要遍历 N 次其他物体，N 个线程共需要访问 O(N^2)次全局内存，但由于总位置数据量共有 N 个，存在大量重复读取，故可采用 tiling 技术，将全局内存中的数据放入 Shared Memory，读取后再从 Shared Memory 中读取进行计算。

实现细节上，设置 Tilesize 和 Blocksize 相等，每次所有线程读取一个全局变量中的位置信息即可。

```
__shared__ float3 sp[blocksize];
for (int tile = 0, pos = 0; tile < numtiles; tile++, pos += blocksize)
{
    sp[threadIdx.x].x = p[pos + threadIdx.x].x;
    sp[threadIdx.x].y = p[pos + threadIdx.x].y;
    sp[threadIdx.x].z = p[pos + threadIdx.x].z;
    __syncthreads();//Ensure all data are written
    for(int j = 0; j < blocksize; j++)
    {
        float3 r={0.0f,0.0f,0.0f};
        r.x=sp[j].x-mypos.x;
        r.y=sp[j].y-mypos.y;
        r.z=sp[j].z-mypos.z;
        float distSqr = r.x * r.x + r.y * r.y + r.z * r.z + SOFTENING;
        float invDist = rsqrtf(distSqr);
        float invDist3 = invDist * invDist * invDist;
        acc.x+=r.x*invDist3;
        acc.y+=r.y*invDist3;
        acc.z+=r.z*invDist3;
    }
    __syncthreads();//Prevent writing before others read
}
```

由于 Shared Memory 完全写完之前，不能从中读取数据（否则会读到无效数据），故需要添加__syncthreads()以保证写入完毕。此外，还需要在循环结束前保证同步，否则会导致某线程未完全计算完，其他线程修改 Shared Memory 导致该线程读到下一次循环对应的数据。

3．优化二：使用多重线程加速

由于每个线程需要对 N 个其他物体进行运算，可以考虑增加重复的线程来为同一个计算任务加速，即对 N 个物体进行再次划分。对于每一个线程而言，由于增加了 dup 倍，故线程对应最大数量为 dup*nBodies-1。我们将其映射为（任务标号 i，重复标号 st）。计算方式如下：

```
register int i=threadIdx.x+blockIdx.x*blocksize;
register int st=i%dup;
i/=dup;
```

在计算贡献时，由于重复线程的设计，每个线程不需要利用所有的 Shared Memory 中的内存，故可以每隔 dup 计算一次，起始位置从 st 开始，即可保证计算不重不漏。

```
for(int j = st; j < blocksize; j+=dup)
```

由于多线程计算同一物体的速度贡献，在分别计算完毕后需要对速度贡献进行累加。即多线程对某一个变量进行修改，需要使用 cuda 提供的原子操作 atomicAdd()，在修改前加锁，修改后解锁以保证不存在写冲突问题。

```
atomicAdd(&p[i].vx,acc.x*dt);
atomicAdd(&p[i].vy,acc.y*dt);
atomicAdd(&p[i].vz,acc.z*dt);
```

4．优化三：减少同步+滚动数组优化

由于 GPU 在调用 kernal 函数时是有序的，即前一个函数调用结束后下一个函数才开始运算。可以发现，位置更新可以在物体 i 的所有线程加速度计算结束后立刻进行，而不需要等待所有物品对应的任务全部计算结束后再进行。显然后者等待所有任务完成是更慢的，所以我们可以在 BodyForce 函数中进行位置的更新。

我们需要利用信号量 flag[i]，保证每个任务 i 的计算全部结束。对于任务 i，我们需要 dup 个线程在更新速度完毕后对信号量进行自增。当信号量等于 dup 时，证明所有线程已经完成任务 i，此时可以进行位置的更新。下面 atomicInc()完成自增操作及清空操作，atomicExch()完成位置更新操作。

```
atomicAdd(&p[i].vx,acc.x*dt);
atomicAdd(&p[i].vy,acc.y*dt);
atomicAdd(&p[i].vz,acc.z*dt);
if(!now)
{
    atomicAdd(&oldp[i].vx,acc.x*dt);
    atomicAdd(&oldp[i].vy,acc.y*dt);
    atomicAdd(&oldp[i].vz,acc.z*dt);
    if(dup==1||!atomicInc(&flag[i],dup-1))
    {
        atomicExch(&newp[i].x,oldp[i].x+oldp[i].vx*dt);
        atomicExch(&newp[i].y,oldp[i].y+oldp[i].vy*dt);
        atomicExch(&newp[i].z,oldp[i].z+oldp[i].vz*dt);
    }
}
else
{
    atomicAdd(&newp[i].vx,acc.x*dt);
    atomicAdd(&newp[i].vy,acc.y*dt);
    atomicAdd(&newp[i].vz,acc.z*dt);
    if(dup==1||!atomicInc(&flag[i],dup-1))
    {
        atomicExch(&newp[i].x,oldp[i].x+newp[i].vx*dt);
        atomicExch(&newp[i].y,oldp[i].y+newp[i].vy*dt);
        atomicExch(&newp[i].z,oldp[i].z+newp[i].vz*dt);
    }
}
```

在上述代码中还有另外两个细节。由于__syncthreads()仅能保证 BLOCK 内部 THREAD 的同步，不能保证 BLOCK 之间的同步，所以在更新位置信息时，不能更改原先的位置信息，否则其他 BLOCK 在计算时会使用到下一轮的位置信息从而导致计算错误。这时我们需要额外开辟一个内存空间，将新位置信息写入到该内存空间中。然后在 KERNAL 函数调用时，交换 newp 与 oldp 的指针，从而实现新旧地址的交换。下为函数定义及调用的代码：

```
__global__ void bodyForce(Body *newp,Body *oldp, unsigned int *flag,int n, int numtiles,int now)
```

```
for (int iter = 0, now = 0; iter < nIters; iter++, now = 1 - now) {
    StartTimer();
    bodyForce<<<blocknum*dup,blocksize>>>(d_p[1-now],d_p[now],d_f,nBodies,tilenum);
    if(iter+1==nIters)cudaMemcpy(buf,d_buf[1-now],bytes,cudaMemcpyDeviceToHost);
    else cudaDeviceSynchronize();
    const double tElapsed = GetTimer() / 1000.0;
    totalTime += tElapsed;
}
```

而速度信息的更新没有必要使用额外的地址空间，但由于 Body 结构体封装了位置与速度信息，所以我们根据 now 的取值，将速度信息仅存储到 buf[0]指向的位置处，即选择 newp 或者 oldp 进行读写。

5. 优化四：编址&访存优化

在优化二中由于在计算物体间加速度时，每个线程需要隔 dup 个进行读取，重新考虑对优化二进行重新编码，使得每个线程读取的内容在 Shared Memory 中连续分布。重新计算（任务标号 i，重复标号 st）如下

```
register int i=threadIdx.x+blockIdx.x*blocksize;
int st=i/n;
i%=n;
```

此时，每个 BLOCK 的所有线程与优化二中的不同，每个线程的任务标号不同，而不是重复 dup 次相同的任务标号。同时，由于 dup 为 BLOCKsize 的因子，使得每个 BLOCK 中的 st 相同，即在计算加速度过程中，仅需要使用在全局变量中下标为 st（模 dup 意义下）对应的数据进行运算，故仅需要读取这些数据到 Shared Memory 中。

```
if(threadIdx.x%dup==st)
{
    sp[threadIdx.x/dup].x = p[pos + threadIdx.x].x;
    sp[threadIdx.x/dup].y = p[pos + threadIdx.x].y;
    sp[threadIdx.x/dup].z = p[pos + threadIdx.x].z;
}
__syncthreads();//Ensure all data are written
for(int j = 0; j < blocksize/dup; j++)
```

同由于读取到 Shared Memoey 中的数据量减少，内层循环次数需要遍历的次数需要除 dup，所有线程做到了连续访问，使得访存更快。

6. 优化五：减少访问全局变量

由于优化四中，在读取全局内存时，部分线程不进行读取（但因为没有 else 块，不会产生控制分歧，也不会影响性能），所以读取的数量少，读取的次数多。我们可以让所有的线程都读取合法的数据，使得减少读取的次数，修改如下：

```
for (int tile = 0, pos = 0; tile < numtiles/dup; tile++, pos += blocksize*dup)
{
    sp[threadIdx.x/dup].x = p[pos + st + dup * threadIdx.x].x;
    sp[threadIdx.x/dup].y = p[pos + st + dup * threadIdx.x].y;
    sp[threadIdx.x/dup].z = p[pos + st + dup * threadIdx.x].z;
    __syncthreads();//Ensure all data are written
    for(int j = 0; j < blocksize; j++)
```

7．优化六：其他优化

1.cuda 对于三维向量运算提供硬件支持，使用 float3 类型存储各种信息，如位置、速度、加速度，可以使运算更快。

2.使用循环展开，对于里层计算物体间加速度循环体，由于循环次数是常数，使用#pragma unroll 编译器指令将循环进行展开，减少因控制分值导致的计算，提高计算效率。

3.根据计算效率调整 blocksize 与 dup 数值大小使得效率最高。

八、实验数据及结果分析（仅列出部分优化的最佳结果）：

1.基础并行代码

```
std2019081301026@cu05:~/cuda$ ./nbody1
Simulator is calculating positions correctly.
4096 Bodies: average 58.416 Billion Interactions / second
```

2.优化一+优化三+优化四+优化六（BLOCKSIZE=256，DUP=1）

```
std2019081301026@cu05:~/cuda$ ./nbody2
Simulator is calculating positions correctly.
4096 Bodies: average 82.362 Billion Interactions / second
```

2.优化一+优化二+优化四+优化六（BLOCKSIZE=256，DUP=8）

```
std2019081301026@cu05:~/cuda$ ./nbody2
Simulator is calculating positions correctly.
4096 Bodies: average 182.361 Billion Interactions / second
```

3.优化一+优化二+优化四+优化五+优化六（BLOCKSIZE=128，DUP=32）

```
std2019081301026@cu05:~/cuda$ ./nbody2
Simulator is calculating positions correctly.
4096 Bodies: average 174.581 Billion Interactions / second
```

九、实验结论：

成功实现了 n-body 程序的并行化，并通过各种优化手段使得相对初始代码效率提升 3 倍以上。其中英伟达官方 cuda 的 n-body 演示程序使用的是优化一+优化三+优化四+优化六。在本实验中，其效率并不是最高的，其中的因素可能涉及其他本报告未涉及的因素。本实验最优的优化为优化一+优化二+优化四+优化六，优化五和优化三作为理论上的优化，在实际测试中反而产生了负作用，说明理论与实践对于优化手段来说都是必要的。

十、总结及心得体会：

丰富了并行运算的手段，CPU 的 MPI 与 GPU 的 cuda。学习了 cuda 的使用，并通过查找多方资料了解到 cuda 的各种优化手段。深入了解了支持 cuda 显卡的物理架构，学会利用该架构加速高性能运算。

十一、对本实验过程及方法、手段的改进建议：

代码作为实现计算的手段，不应设置无意义的代码禁区，这与代码本身的意义相违背。既然是高性能计算，代码的任何时间都需要被计入。我认为设置禁区的意义是防止学生改变计时器的数据，但是这样也会限制代码的编写（比如在优化三中，因为滚动数组的使用，最高效率的写法是将位置信息与速度信息分开存储，而不是使用 Body 结构体，这回产生空间冗余）。为保证时间的正确计入与代码的顺利编写，希望可以将计时功能放在代码之外，额外编写一个用于计时的脚本，在程序运行前计时，然后启动程序，在程序结束时停止计时。这样的话计入的时间是全部时间，只要代码可以得到正确的结果，就说明代码正确、学生编写代码不受限并且学生无法修改时间信息。

报告评分：

指导教师签字：