

电子科技大学

实验报告

学生姓名：	学号：
一、实验室名称：主楼 A2-412	
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： 1. CUDA： CUDA 是 NVIDIA 专为图形处理单元 (GPU) 上的通用计算开发的并行计算平台和编程模型。借助 CUDA，开发者能够利用 GPU 的强大性能显著加速计算应用。 在经 GPU 加速的应用中，工作负载的串行部分在 CPU 上运行，且 CPU 已针对单线程性能进行优化，而应用的计算密集型部分则以并行方式在数千个 GPU 核心上运行。使用 CUDA 时，开发者使用主流语言（如 C、C++、Fortran、Python 和 MATLAB）进行编程，并通过扩展程序以几个基本关键字的形式来表示并行性。 2. N-Body 问题 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。	

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）：

计算节点配置：

CPU E5-2660 v4*2

Nvidia K80*2

操作系统：CentOS 7.2

CUDA：10.0

本机机器配置：

CPU：Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz

内存：16 GB DDR4

操作系统：Windows11

软件：VSCODE、REMOTE-SSH

七、实验步骤及操作：

1. 使用远程连接工具，连接 MPI 跳板机后，以 SSH 模式连接配有 CUDA 环境的虚拟机。

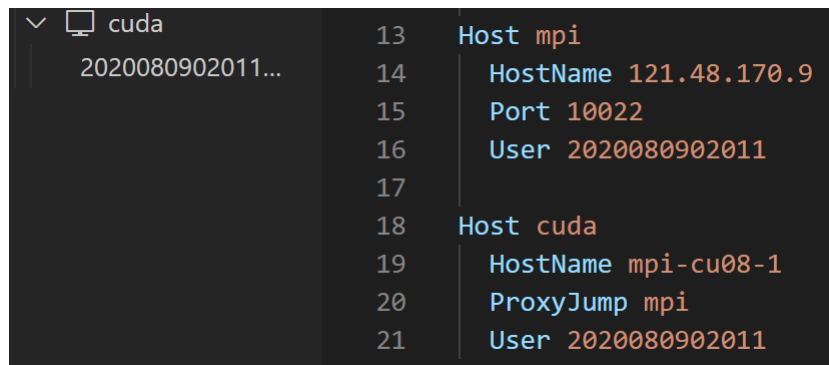


图 1 VSCODE 远程连接配置

2. 基准串行代码的运行及程序说明。

首先我们分析串行代码。程序使用 `randomizeBodies` 随机初始化 4096 个 `Body` 的位置和速度，而后我们需要在此基础上迭代十次。

每次经过 `dt` 的时间，对每个 `Body`，计算出其他 4095 个 `Body` 对它自身的合力，才能得到 `Body` 下一次迭代的速度，进而得到下一次迭代的位置。最后将这十次迭代的时间取平均值，将单次计算过的 `Body` 数量除以该值，得到评判标准。

下图为计算 `Body` 合力和速度的函数 `bodyForce`，直接两层循环，暴力计算了 4096×4096 次，才能把所有的 `body` 的速度更新完毕。

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

图 2 串行更新 body 速度代码

下图为计算 Body 位置的代码，循环了 4096 次，把所有的 body 的位置更新完毕。

```
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;
}
```

图 3 串行更新 body 位置的代码

这 bodyForce 函数的部分是消耗计算资源的大头，由于我们的基准代码是 CPU 串行的，因此容易知道这样所得到的结果并不理想。

3. 代码优化及优化后代码的程序说明。

1) 基本并行化

由于每一个 Body 速度的更新都只依赖于上一轮迭代的所有 Body 的位置，每一个 Body 位置的更新都只依赖于本轮迭代的自身的速度，因此我们认为：在同一轮迭代中，各个 Body 的计算都是相互独立的。容易想到，我们可以为每一个 Body 分配一个线程，线程彼此之间互不干扰，单独负责该 Body 的计算。

```
__global__ void bodyForce(Body* p, float dt, int n)
{
    // 每个进程负责一个body, index是全局的进程号
    int index = threadIdx.x + blockIdx.x * blockDim.x;

    float Fx = 0.0f;
    float Fy = 0.0f;
    float Fz = 0.0f;

    // 计算合力
    for (int j = 0; j < n; j++) {
        float dx = p[j].x - p[index].x;
        float dy = p[j].y - p[index].y;
        float dz = p[j].z - p[index].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;
    }

    // 计算速度
    p[index].vx += dt * Fx;
    p[index].vy += dt * Fy;
    p[index].vz += dt * Fz;
}
```

图 4 基本串行化后的 bodyForce 函数

对 bodyForce 函数进行串行化，将该函数声明为 CUDA 的核函数。使用 4096 个线程计

算，即可将原来串行函数里的第一层循环去掉。

同理，Body 位置的计算也可以并行。将计算 Body 位置的代码，提取成一个核函数叫做 `integratePosition`。

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

    // 计算位置
    p[index].x += p[index].vx * dt;
    p[index].y += p[index].vy * dt;
    p[index].z += p[index].vz * dt;
}
```

图 5 基本并行化后的 `integratePosition` 函数

对线程块大小 `BLOCK_SIZE` 进行调参，首先如下图查看 GPU 的 CUDA 相关信息：

```
Device0:"Tesla K80"
CUDA Driver Version:      11.4
CUDA Runtime Version:     11.4
Device Prop:              3.0
Total amount of Global Memory: 4209704960 bytes
Number of SMs:            13
Total amount of Constant Memory: 65536 bytes
Total amount of Shared Memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size:                32
Maximum number of threads per SM: 2048
Maximum number of threads per block: 1024
Maximum size of each dimension of a block: 1024 x 1024 x 64
Maximum size of each dimension of a grid: 2147483647 x 65535 x 65535
Maximum memory pitch:     2147483647 bytes
Texture alignment:        32 bytes
Clock rate:               0.82 GHz
Memory Clock rate:        2505 MHz
Memory Bus Width:         384-bit
```

图 6 GPU 的 CUDA 相关信息

因为线程束 `warp` 为 32，若 `BLOCK_SIZE` 小于 32 则会产生浪费，因此 `BLOCK_SIZE` 应该为 32 的倍数。同时线程块大小的最大值为 1024，若超过 1024，程序运行错误。经过实际运行测试，`BLOCK_SIZE` 为 32、64、128 时，性能最好且差异不大。

2) 共享内存

`bodyForce` 和 `integratePosition` 函数的入参数组 `p` 是存放在 Device 的全局内存中，每一个线程块可以共同访问。在 CUDA 编程中，合理利用线程块的共享内存是一个常见的优化手段，将我们所需要的数据提前从全局内存拷贝到共享内存中，使用实际数据可以节省大量时间。

首先根据图 6 查看线程块共享内存的最大值为 49152 bytes, 这恰好能够保存 Body 的 3 个 float 类型的位置信息 ($4096 \times 3 \times 4 = 49152$ bytes)。因此一开始我试图将所有 Body 的位置信息直接一起放到每一个线程块的共享内存中。但是由于线程块的线程个数有限, 放入共享内存的过程还需要循环执行, 性能其实并不好。而且可能是由于共享内存完全占满导致的边界问题, 运行结果是错误的。只好放弃这种简单粗暴的想法。

我们选择将一次载入共享内存的大小设置为与 BLOCK_SIZE 相同的值, 为什么呢? 因为这样我们可以使得, 线程块里的每一个线程, 恰好都只需要将它所对应的 Body 的信息载入共享内存, 不需要循环载入多次, 这样实现容易且代码简洁。

使用共享内存改进后的 bodyForce 代码如下:

```
1. __global__ void bodyForce(Body* p, float dt, int n, int nBlocks)
2. {
3.     // 每个进程负责一个 body, index 是 grid 全局的进程号
4.     int index = threadIdx.x + blockIdx.x * blockDim.x;
5.     Body body = p[index];
6.
7.     float Fx = 0.0f;
8.     float Fy = 0.0f;
9.     float Fz = 0.0f;
10.
11.    // 使用线程块的共享内存
12.    __shared__ float3 tile[BLOCK_SIZE];
13.
14.    // 分批计算合力, 一批大小为 BLOCK_SIZE
15.    // 为什么一批的大小要为 BLOCK_SIZE? 因为用 BLOCK_SIZE 个线程刚好拷贝 BLOCK_SIZE 个 body 的位置信息到共享内存
16.    for (int i = 0; i < nBlocks; i++) {
17.        // 将全局内存的数据拷贝到共享内存
18.        Body temp = p[threadIdx.x + i * BLOCK_SIZE];
19.        tile[threadIdx.x] = make_float3(temp.x, temp.y, temp.z);
20.
21.        __syncthreads();
22.
23.        for (int j = 0; j < BLOCK_SIZE; j++) {
24.            float dx = tile[j].x - body.x;
25.            float dy = tile[j].y - body.y;
26.            float dz = tile[j].z - body.z;
27.            float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
28.            float invDist = rsqrtf(distSqr);
29.            float invDist3 = invDist * invDist * invDist;
30.        }
```

```

31.         Fx += dx * invDist3;
32.         Fy += dy * invDist3;
33.         Fz += dz * invDist3;
34.     }
35.
36.     __syncthreads();
37. }
38.
39. // 计算速度
40. // 这里虽然用不着使用原子加法，但是使用原子加法却更快？
41. // p[index].vx += dt * Fx;
42. // p[index].vy += dt * Fy;
43. // p[index].vz += dt * Fz;
44. atomicAdd(&p[index].vx, dt * Fx);
45. atomicAdd(&p[index].vy, dt * Fy);
46. atomicAdd(&p[index].vz, dt * Fz);
47. }

```

3) 提高计算 Body 速度的并行度

由图 6 知道，一张 K80s 有十三个 SM。SM 其实是将线程分成一个个 warp 进行处理的，当 warp 的线程进行需要等待的事件时，比如访存，此时 SM 需要切换另一个 warp 继续进行运算。若所有的 warp 都需要等待，SM 只能处于无事可做的状态，这样 SM 的占用率就低了。所以我们的想法是，进一步提高代码并行度，提高 SM 的占用率，以掩盖访存的时延。

基本并行化中，我们使用一个线程单独负责一个 Body 的更新。而对于一个 Body 而言，其他 Body 对它的作用力也是相互独立的，因此我们可以使用多个线程共同负责一个 Body 速度的更新。

接下来要考虑的就是：到底使用多少个线程负责一个 Body 呢？

若 BLOCK_SIZE 为 128，按照我们步骤 2) 的基本并行化的思想，一个线程块读到共享内存的 Body 数量是 128，需要循环 $nBodies / BLOCK_SIZE = 32$ 次，才能将 4096 个 Body 遍历完。于是我们可以想到：直接将线程倍增到原来的 MULTIPLE 倍，使得 MULTIPLE 就等于 $nBodies / BLOCK_SIZE = 32$ 的值，每个线程负责将一个固定的 Body 位置信息拷贝到共享内存，这样就可以恰好将步骤 3) 的共享内存优化的外层循环直接去掉。

这样的话，对于一个线程而言，其所负责的 Body 在全局 p 数组的位置就是： $(threadIdx.x + blockIdx.x * blockDim.x) \% n$ ，其所负责拷贝到共享内存的 Body 的全局 p 数组的位置就是： $threadIdx.x + blockIdx.x / nBlocks * BLOCK_SIZE$ 。

因为线程的数量增多会导致每个线程所分配的资源减少，所以线程数量并非越多越好，因此该优化也需要进行调参，只需要保持 nBodies 等于 BLOCK_SIZE 与 MULTIPLE 的乘积即可。

经过测试，在本代码中，当 BLOCK_SIZE 为 512、MULTIPLE 为 8 时性能达到最佳。

提高计算速度并行度后的 bodyForce 代码如下：

```
1. __global__ void bodyForce(Body* p, float dt, int n, int nBlocks)
2. {
3.     // index 是 body 的序号
4.     int index = (threadIdx.x + blockIdx.x * blockDim.x) % n;
5.     Body body = p[index];
6.
7.     float Fx = 0.0f;
8.     float Fy = 0.0f;
9.     float Fz = 0.0f;
10.
11.    // 使用线程块的共享内存
12.    __shared__ float3 tile[BLOCK_SIZE];
13.
14.    // 该线程应该考虑受力的第一个 body
15.    int start = blockIdx.x / nBlocks * BLOCK_SIZE;
16.
17.    // 将全局内存的数据拷贝到共享内存
18.    Body temp = p[threadIdx.x + start];
19.    tile[threadIdx.x] = make_float3(temp.x, temp.y, temp.z);
20.
21.    __syncthreads();
22.
23.    for (int j = 0; j < BLOCK_SIZE; j++) {
24.        float dx = tile[j].x - body.x;
25.        float dy = tile[j].y - body.y;
26.        float dz = tile[j].z - body.z;
27.        float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
28.        float invDist = rsqrtf(distSqr);
29.        float invDist3 = invDist * invDist * invDist;
30.
31.        Fx += dx * invDist3;
32.        Fy += dy * invDist3;
33.        Fz += dz * invDist3;
34.    }
35.
36.    __syncthreads();
37.
```

```

38.    // 计算速度
39.    atomicAdd(&p[index].vx, dt * Fx);
40.    atomicAdd(&p[index].vy, dt * Fy);
41.    atomicAdd(&p[index].vz, dt * Fz);
42. }

```

4) 循环展开

循环展开其实是串行上的优化，由于 GPU 不像 CPU 的核心有复杂的逻辑控制功能，如：转移预测等。所以在遇到需要跳转计算的指令时，流水线性能损耗较大，因此我们需要尽量减少不必要的判断语句。

for 循环每一次循环都需要一次判断，这会产生许多跳转，因此我们可以考虑对其进行循环的展开，即在一次循环内计算原来多次循环计算的内容。但是循环展开虽然能一定程度上提高性能，但是并不能无限展开，无限展开不仅没有实际性能的提升，甚至可能适得其反。因为循环展开次数过多的话会导致汇编的中间变量增加，而中间变量的存储需要用到有限的寄存器，若中间变量过多只能存到内存中，频繁访存导致性能下降。

因此循环展开的次数需要进行调参，在本代码中，当循环展开的次数达到 64 时性能达到最佳。

5) 提高计算 Body 位置的并行度

思路同步骤 3)，考虑到 Body 的 x、y、z 位置的计算相互之间也是独立的，因此我们可以将 1 个进程增加到 3 个进程分别计算 x、y、z。

但是由于计算 Body 位置并非计算的大头，且由不得不引入 if-else 语句，增加了分支判断，因此这对整体性能的提升并不大，执行结果大概只有增加了一点几。

提高计算速度并行度后的 integratePosition 代码如下：

```

__global__ void integratePosition(Body* p, float dt, int n) {
    // 全局id
    int global_index = threadIdx.x + blockIdx.x * blockDim.x;
    // 负责的body序号
    int index = global_index % n;

    // 计算位置
    if (global_index < n) p[index].x += p[index].vx * dt;
    else if (global_index < 2 * n) p[index].y += p[index].vy * dt;
    else p[index].z += p[index].vz * dt;
}

```

图 7 计算速度并行度后的 integratePosition 函数

八、调试说明：

在本次实验中，我遇到的绝大多数需要调试的情况都可以指定一个进程的全局 id，并打印（printf）出该进程相关变量的信息进行 DEBUG。若 CUDA 程序运行出现错误，往往所有线程出现的错误都一样，因此只需要看随意选择一个进程即可。

九、实验数据及结果分析：

以下是基准代码和优化后的代码的执行结果及对比：

```
● 2020080902011@mpi-cu08-1:~$ ./base
Simulator is calculating positions correctly.
4096 Bodies: average 0.043 Billion Interactions / second
```

图 8 基准串行代码的执行结果

```
● 2020080902011@mpi-cu08-1:~$ ./nbody1-parallel
Simulator is calculating positions correctly.
4096 Bodies: average 11.125 Billion Interactions / second
```

图 9 基本并行化后的执行结果

```
● 2020080902011@mpi-cu08-1:~$ ./nbody2-shared-memory
Simulator is calculating positions correctly.
4096 Bodies: average 32.457 Billion Interactions / second
```

图 10 使用共享内存后的执行结果

```
● 2020080902011@mpi-cu08-1:~$ ./nbody3-multiple-thread
Simulator is calculating positions correctly.
4096 Bodies: average 51.942 Billion Interactions / second
```

图 11 提高计算 Body 速度的并行度后的执行结果

```
● 2020080902011@mpi-cu08-1:~$ ./nbody4-unroll
Simulator is calculating positions correctly.
4096 Bodies: average 56.911 Billion Interactions / second
```

图 12 循环展开后的执行结果

```
● 2020080902011@mpi-cu08-1:~$ ./nbody5-position-mul-thread
Simulator is calculating positions correctly.
4096 Bodies: average 58.416 Billion Interactions / second
```

图 13 提高计算 Body 位置的并行度后的执行结果

1. 由图 8 和图 9 执行结果的对比可以知道：若计算的内容大部分是独立分离的，那么使用 GPU 实现的并行化对程序运行性能有着巨大的提升。

2. 由图 9 和图 10 执行结果的对比可以知道：CUDA 编程可以合理使用共享内存，减少获取数据的时间，以获得性能上的提升。

3. 由图 10 和图 11 执行结果的对比可以知道：我们可以将程序并行的程度进一步提高，以掩盖线程延迟等待的时间，尽可能发挥 SM 的性能，使得程序性能随之提高。

4. 由图 11 和图 12 执行结果的对比可以知道：循环展开是优化代码性能的常用手段。对

于 GPU 核心较弱的情况下，循环展开所获得的提升更加明显。

5. 由图 12 和图 13 执行结果的对比可以知道：程序优化的效果何如，与所优化代码部分原来所占时间在整体中的比例相关。

十、实验结论：

通过对基准串行代码采取一系列的优化措施，CUDA 并程序的性能得到了相当大的提升，提升了将近 1359 倍。其中，基本并行的实现和进一步提高并行度的优化效果最为显著，这两个措施的考虑归根结底都是进一步提高程序的并行程度。

十一、总结及心得体会：

1. 通过本次实验，熟悉了 CUDA 编程，对并行编程有了更加深刻的理解。更培养了调试并行代码的能力。

2. 让我对 N-Body 问题有了进一步的了解。过去 N-Body 问题只在书上接触过，并未实践落实，这次终于有机会来思考这个有趣的问题。

3. 本次实验让我深刻认识到了：并行能够对代码产生多么巨大的提升。过去我优化串行的代码从来是在串行算法本身下功夫，但是往往这样的优化是很困难，而且不能通用，往往是换了个问题就不得不重新思考。而并行算法为我提供了另一个优化代码的视角与思路，并且并行的思想几乎在许多问题上都能使用。

4. 我意识到了，优化问题要对问题的大头下手所获得的收益才是最大的。而在不是问题的瓶颈处考虑的话，收效甚微。

十二、对本实验过程及方法、手段的改进建议：

1. 多进行类似有趣的并行编程实验，能提高同学的学习积极性。
2. 希望能提供更多 CUDA 优化相关的资料供同学们学习。

报告评分：

指导教师签字：