

电子科技大学

实验报告

学生姓名：蔡与望	学号：2020010801024
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	
<p>三、实验原理：</p> <p>N 个物体相互施加引力，会使得它们的速度和位置不断改变。N-Body 问题就模拟了这样的系统，力求解得每个物体在某时刻的确定坐标。</p> <p>在三维空间内，每个物体都拥有三个坐标 x、y、z，以及在三个方向上的分速度 v_x、v_y、v_z，并且假定每个物体的质量 m 都为 1。</p> <p>在计算两物体间引力时，采用公式：</p> $\begin{cases} F_x = \frac{d_x}{d^3} \\ F_y = \frac{d_y}{d^3} \\ F_z = \frac{d_z}{d^3} \end{cases}$ <p>其中，F_x、F_y、F_z 是引力在各方向上的分力，d_x、d_y、d_z 是两个物体在各方向上的投影距离。</p> <p>在更新速度时，采用公式：</p> $\begin{cases} v_x = \sum t F_x \\ v_y = \sum t F_y \\ v_z = \sum t F_z \end{cases}$ <p>其中，v_x、v_y、v_z 是物体在各方向上的分速度，t 是一个极小的时间段。</p> <p>在更新坐标时，采用公式：</p> $\begin{cases} x = v_x t \\ y = v_y t \\ z = v_z t \end{cases}$ <p>在串行程序中，我们可以不断重复“叠加受力、更新速度、更新坐标”的循环，得到最终坐标。</p> <p>在使用 Cuda C 并行化程序时，可以让每个线程单独负责一个物体。迭代开始前，使用 <code>CudaMemcpy</code> 将主机内存分发给各个设备，然后每个线程各自重复“叠加受力、更新速度和更新坐标”的循环，最后再使用 <code>CudaMemcpy</code> 把计算好的数据传回主机。</p>	

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 并程序的性能优化。

六、实验器材（设备、元器件）：

计算节点配置：

1. CPU E5-2660 v4*2
2. Nvidia K80*2

操作系统：

1. CentOS 7.2
2. CUDA 10.0

七、实验步骤及操作：

1. 基准代码

基准代码是一个串行程序，思路如下：

- a) 使用 `randomizeBodies` 函数，随机初始化 4096 个物体的坐标和速度。
- b) 使用 `bodyForce` 函数，计算每个物体的总受力情况，并依此更新速度。
- c) 在新的速度下，运行 0.01 秒，计算出新的坐标。
- d) 重复执行 b-c 步骤 10 次。
- e) 提交给 `check` 库检查。

2. 并行化

并行化的基本思路包括两部分：①使用 Cuda 的 API 分发内存；②每个进程负责一个物体。

在 Cuda 程序中，我们可以使用 `CudaMalloc`、`CudaMemcpy` 等函数，管理多机的内存。具体来说，就是先在主机上划一块总的内存，然后拷贝给各台设备；各台设备在计算完成之后，再拷贝回主机。

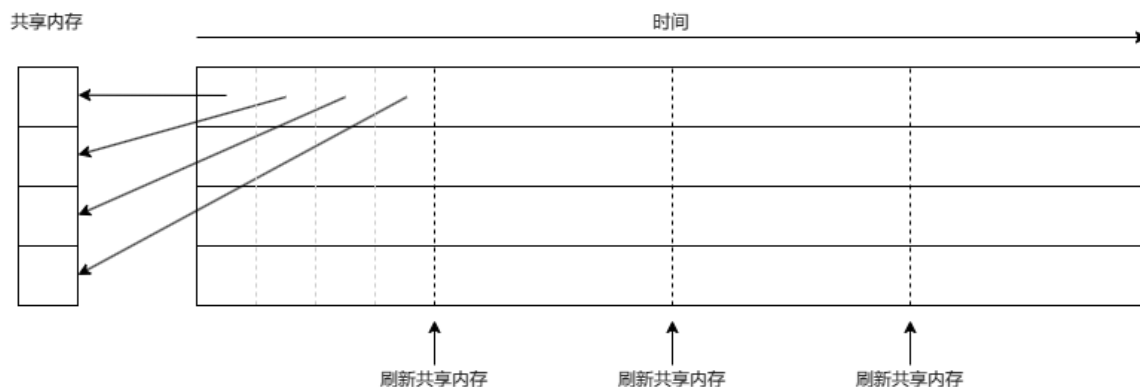
对于 `bodyForce` 函数，我们可以让每个线程都负责更新一个物体的速度。负责物体的下标为：`threadIdx.x + blockDim.x * blockIdx.x`。更新坐标时，也可以让每个线程负责一个物体，下标计算方式与 `bodyForce` 相同。

具体代码见 `nbody1.cpp`。

3. 块级共享内存

在上述代码中，`bodyForce` 每次计算受力情况时，都会直接从全局内存中取值，访存速度慢。

我们可以使用共享内存来优化速度，如下图所示。



在刷新点上，每个线程都从全局内存中取一个物体放进共享内存，之后该块内所有线程的访存都在共享内存中发生。直到每个进程都计算完了这些物体对自己负责的物体的施力，就开启下一个刷新点，取一批新的物体进入共享内存。如此反复，直到全部算完。

要注意的是，在每个刷新点的前后，都要使用 `__syncthreads` 函数确保所有物体已装入（或已消耗）。

4. 线程数翻倍

容易想到，即使使用了共享内存，整个过程也要更换多次共享内存。解决这个问题最直接的方法就是让线程数翻倍，多个线程同时计算一个物体。例如，原来有 128 个块，物体 0 由线程(0,0)负责；则翻 4 倍后，物体 0 就会由线程(0,0)、(128,0)、(256,0)、(384,0)一起负责。其中，线程(0,0)计算物体 0、4、……的施力，线程(128,0)计算物体 1、5、……的施力，以此类推。

同时，因为现在有多个线程共同修改一个物体，所以原来的加法需要改为原子加法（`atomicAdd`），以避免并发引起的竞态问题。

5. 缓存负责物体

这是一个不起眼但非常有效的优化点。

我们现在每次计算距离时，都需要从全局内存中取出 `p[i]`。在不使用共享内存的程序中，这无关紧要，因为一切的数据反正是从全局内存中获取的。然而，如果在使用共享内存之后，还是每次都要从全局内存里获取 `p[i]`，那就大大降低了访存速度。

所以，我们可以在 `bodyForce` 的一开始，就把 `p[i]` 缓存到局部变量里，这样就仅需访问一次全局内存。

6. 循环展开

编译器优化也是 Cuda 程序常见的优化方式，循环展开就是其中较有成效的一种。

对于 `bodyForce` 中的每个 `for` 循环，我们都可以使用 `#pragma unroll N` 来展开循环；用编译时间的延长，换取运行时负担的减轻。

具体代码见 `nbody2.cpp`。

7. 编译运行

使用命令 `nvcc -arch sm_37 -o main nbody.cu && ./main` 编译并运行 Cuda 程序。结果的第一行指示结果是否正确，第二行报告程序性能（单位为十亿次交互每秒）。

八、实验数据及结果分析：

分别运行基准代码、nbody1.cu 和 nbody2.cu，观察并对比性能。

```
2020010801024@mpi-cu07-1:~/cuda$ nvcc -arch sm_37 -o main base.cu && ./main
nvcc warning : The 'compute_35', 'compute_37', 'compute_50', 'sm_35', 'sm_37'
warning).
Simulator is calculating positions correctly.
4096 Bodies: average 0.033 Billion Interactions / second
2020010801024@mpi-cu07-1:~/cuda$ nvcc -arch sm_37 -o main nbody1.cu && ./main
nvcc warning : The 'compute_35', 'compute_37', 'compute_50', 'sm_35', 'sm_37'
warning).
Simulator is calculating positions correctly.
4096 Bodies: average 11.156 Billion Interactions / second
2020010801024@mpi-cu07-1:~/cuda$ nvcc -arch sm_37 -o main nbody2.cu && ./main
nvcc warning : The 'compute_35', 'compute_37', 'compute_50', 'sm_35', 'sm_37'
warning).
Simulator is calculating positions correctly.
4096 Bodies: average 56.167 Billion Interactions / second
2020010801024@mpi-cu07-1:~/cuda$ |
```

从上图中，我们可以了解到以下几点：

- 1. 所有程序都能够正确地计算出坐标。
- 2. 在并行化后，平均每秒交互 11.156（十亿次），性能变为原来的 338 倍。
- 3. 在最终优化后，平均每秒交互 56.167（十亿次），性能变为原来的 1702 倍。

九、实验结论：

本实验中编写的并行代码和优化代码，可以正确地求解 N-Body 问题；并且优化后的性能是基准代码的 1702 倍，有了巨幅的提升。

因此可以认为，编写的代码完成了实验的所有要求。

十、总结及心得体会：

通过本实验，我对 Cuda C 程序的编写有了基本的了解，知道了如何使用 Cuda API 来管理内存、进行原子操作，并且对 Cuda 的核心概念——尤其是共享内存——有了更深的了解。我现在能够通过分析程序代码，定位重要的可优化点，并做出正确、高性能的优化。

十一、对本实验过程及方法、手段的改进建议：

无。

蔡与莹

报告评分：

指导教师签字：