**上海大学**

计算机工程与科学学院

计算机创新实践作业

实践报告

badge

课题名称： N-Body程序的CUDA和OpenMP的混合并行调度

**成员：学号 11122212 姓名 蔡林林 成绩：**

**学号 11121234 姓名 杨治均 成绩：**

**学号 11121309 姓名 李清城 成绩：**

**学号 11122124 姓名 马卿 成绩：**

**指导教师签名：**

**2014年8月18日**

1. **实验目的**
2. 了解GPU/CPU, GPU/CPU的架构以及GPU的特点
3. 了解CUDA，学习安装CUDA，并进行编程
4. 会使用CUDA进行混合编程并且优化
5. **实验要求**
6. 完成例子程序中nbody（多体问题）在OPM-CU的混合并行计算。
7. 自选题目，完成OPM-CU（或者仅仅是CU）的并行，要求在任何一个层次能提出自己的想法，例如调度，优化等。
8. **实验背景**
9. CPU的内部结构：控制单元、逻辑单元和存储单元

* 运算器：算术运算(如加减乘除)和逻辑运算( 如逻辑加、逻辑乘和非运算)
* 控制器：读取各种指令,并对指令进行分析,做出相应的控制。
* 寄存器：可直接参与运算并存放运算的中间结果。

1. GPU采用了大量的执行单元，这些执行单元可以轻松的加载并行处理，而不像CPU那样的单线程处理。另外，现代的GPU也可以在每个指令周期执行更多的单一指令。 GPU将更多的器件用于数据处理。
2. 目前CUDA除了可以用C语言开发，也已经提供FORTRAN的应用接口，未来可以预计CUDA会支持C++、Java、Python等各类语言。现在更多的应用在游戏、图形动画、科学计算、地质、生物、物理模拟等领域
3. CUDA优势与限制

采用CUDA技术相比传统的GPU计算的优势：

* 1. CUDA的编程接口采用了标准的C语言程序进行扩展，有利于CUDA的学习
  2. 每个线程之间CUDA为他们提供了16kb的共享内存，可用于设置缓存，同时还拥有更高的带宽纹理
  3. 系统内存和显存之间更有效地进行数据传输
  4. 无需图形API接口进行连接

CUDA技术目前的一些限制：

* 1. 不支持函数递归
  2. CUDA架构仅限于NVIDIA产品

1. OpenMP并行编程模型

* 基于线程的并行编程模型(Programming Model)
* OpenMP使用Fork-Join并行执行模型

1. 混和编程方式

CPU做辅助工作，GPU做密集计算都参加密集计算，本质上是一个动态调度问题

GPU作为协处理器参与一个（或者多个线程）计算

1. **实验步骤**
2. CUDA的安装
   1. 安装vs2008或者安装vs2010
   2. 下载CUDA安装文件：CUDA\_5.5.20\_winvista\_win7\_win8\_general\_64（需要下载正确的32或者64位版本）
   3. 安装并学习和运行例子程序
3. CUDA的编程，使用vs2012参照实例进行编写程序并且运行
4. 记录实验结果并且进行分析
5. **实验结果**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 实验序号 | CPU并行耗时 | GPU并行耗时 | 理论任务分配比 | 当前任务分配比 | GPU&CPU混合并行耗时 | 相对CPU加速比 | 相对GPU加速比 |
| 1 | 7.363 | 4.696 | 0.610581 | 0.5 | 4.103 | 1.794541 | 1.144528 |
| 2 | 7.347 | 4.695 | 0.610115 | 0.505 | 4.025 | 1.825342 | 1.16646 |
| 3 | 7.363 | 4.696 | 0.610581 | 0.51 | 3.978 | 1.85093 | 1.180493 |
| 4 | 7.347 | 4.695 | 0.610115 | 0.515 | 3.947 | 1.861414 | 1.189511 |
| 5 | 7.363 | 4.68 | 0.611393 | 0.52 | 3.931 | 1.87306 | 1.190537 |
| 6 | 7.379 | 4.696 | 0.611097 | 0.525 | 3.884 | 1.899846 | 1.209063 |
| 7 | 7.363 | 4.68 | 0.611393 | 0.53 | 3.853 | 1.910978 | 1.214638 |
| 8 | 7.332 | 4.68 | 0.61039 | 0.535 | 3.76 | 1.95 | 1.244681 |
| 9 | 7.347 | 4.695 | 0.610115 | 0.54 | 3.775 | 1.946225 | 1.243709 |
| 10 | 7.348 | 4.696 | 0.610096 | 0.545 | 3.744 | 1.962607 | 1.254274 |
| 11 | 7.332 | 4.68 | 0.61039 | 0.55 | 3.666 | 2 | 1.276596 |
| 12 | 7.379 | 4.68 | 0.611908 | 0.555 | 3.682 | 2.004074 | 1.271048 |
| 13 | 7.378 | 4.68 | 0.611876 | 0.56 | 3.635 | 2.029711 | 1.287483 |
| 14 | 7.363 | 4.68 | 0.611393 | 0.565 | 3.619 | 2.03454 | 1.293175 |
| 15 | 7.348 | 4.696 | 0.610096 | 0.57 | 3.573 | 2.056535 | 1.314302 |
| 16 | 7.363 | 4.68 | 0.611393 | 0.575 | 3.541 | 2.079356 | 1.321661 |
| 17 | 7.363 | 4.696 | 0.610581 | 0.58 | 3.525 | 2.088794 | 1.332199 |
| 18 | 7.347 | 4.696 | 0.610064 | 0.585 | 3.417 | 2.150132 | 1.374305 |
| 19 | 7.363 | 4.68 | 0.611393 | 0.59 | 3.432 | 2.145396 | 1.363636 |
| 20 | 7.363 | 4.68 | 0.611393 | 0.595 | 3.401 | 2.164951 | 1.376066 |
| 21 | 7.348 | 4.696 | 0.610096 | 0.6 | 3.354 | 2.190817 | 1.400119 |
| 22 | 7.348 | 4.68 | 0.610908 | 0.605 | 3.339 | 2.200659 | 1.401617 |
| 23 | 7.363 | 4.695 | 0.610632 | 0.61 | 3.291 | 2.237314 | 1.426618 |
| 24 | 7.348 | 4.696 | 0.610096 | 0.615 | 3.214 | 2.286248 | 1.461108 |
| 25 | 7.332 | 4.695 | 0.609628 | 0.62 | 3.168 | 2.314394 | 1.482008 |
| 26 | 7.379 | 4.695 | 0.611148 | 0.625 | 3.167 | 2.329965 | 1.482476 |
| 27 | 7.363 | 4.696 | 0.610581 | 0.63 | 3.073 | 2.39603 | 1.528148 |
| 28 | 7.332 | 4.696 | 0.609578 | 0.635 | 3.089 | 2.373584 | 1.520233 |
| 29 | 7.363 | 4.696 | 0.610581 | 0.64 | 3.042 | 2.420447 | 1.543721 |
| 30 | 7.364 | 4.696 | 0.610614 | 0.645 | 3.058 | 2.40811 | 1.535644 |
| 31 | 7.363 | 4.68 | 0.611393 | 0.65 | 3.089 | 2.383619 | 1.515053 |
| 32 | 7.348 | 4.696 | 0.610096 | 0.655 | 3.166 | 2.32091 | 1.48326 |
| 33 | 7.348 | 4.696 | 0.610096 | 0.66 | 3.136 | 2.343112 | 1.497449 |
| 34 | 7.378 | 4.695 | 0.611116 | 0.665 | 3.136 | 2.352679 | 1.49713 |
| 35 | 7.363 | 4.695 | 0.610632 | 0.67 | 3.167 | 2.324913 | 1.482476 |
| 36 | 7.379 | 4.681 | 0.611857 | 0.675 | 3.182 | 2.318982 | 1.471087 |
| 37 | 7.363 | 4.696 | 0.610581 | 0.68 | 3.229 | 2.280273 | 1.45432 |
| 38 | 7.348 | 4.696 | 0.610096 | 0.685 | 3.37 | 2.180415 | 1.393472 |
| 39 | 7.363 | 4.696 | 0.610581 | 0.69 | 3.26 | 2.258589 | 1.440491 |
| 40 | 7.348 | 4.696 | 0.610096 | 0.695 | 3.292 | 2.232078 | 1.426488 |
| 41 | 7.379 | 4.696 | 0.611097 | 0.7 | 3.291 | 2.242176 | 1.426922 |
| 42 | 7.348 | 4.68 | 0.610908 | 0.705 | 3.339 | 2.200659 | 1.401617 |
| 43 | 7.348 | 4.68 | 0.610908 | 0.71 | 3.464 | 2.121247 | 1.351039 |
| 44 | 7.347 | 4.695 | 0.610115 | 0.715 | 3.369 | 2.180766 | 1.393589 |
| 45 | 7.348 | 4.696 | 0.610096 | 0.72 | 3.417 | 2.150424 | 1.374305 |
| 46 | 7.332 | 4.695 | 0.609628 | 0.725 | 3.432 | 2.136364 | 1.368007 |
| 47 | 7.347 | 4.695 | 0.610115 | 0.73 | 3.588 | 2.047659 | 1.308528 |
| 48 | 7.348 | 4.696 | 0.610096 | 0.735 | 3.463 | 2.12186 | 1.35605 |
| 49 | 7.379 | 4.68 | 0.611908 | 0.74 | 3.479 | 2.121012 | 1.345214 |
| 50 | 7.348 | 4.68 | 0.610908 | 0.745 | 3.526 | 2.083948 | 1.327283 |
| 51 | 7.347 | 4.695 | 0.610115 | 0.75 | 3.542 | 2.074252 | 1.325522 |

以上是51组实验的原始数据。由上可知，每组实验的只由CPU并行计算及只由GPU计算所花费的时间基本是稳定的。每组实验不同的是GPU和CPU混合计算时GPU和CPU任务分配的比例，由此导致混合计算消耗的时间不同。

1. **数据处理**

根据实验得到的原始数据（GPU和CPU分别单独计算所耗时间），可以得到一个理论任务分配比例，上图中的A点（0.61, 3.291），但由上图可知，混合计算效率的最高点是B点（0.64, 3.042）。由此，可以得到一个结论：GPU和CPU的混合调度存在最优的调度方案，但这个方案和理论计算得到的方案不一样，虽然数据表明有浮动，但总体的趋势还是清晰可见的，整体呈U字型。

以上的实验结果是通过多次不同的静态调度得到的。这些结论对接下来的动态调度有很强的指导作用，表明了动态调度需要解决的问题及难点所在，即找出一个合理的GPU/CPU任务分配方案。

1. **个人体会**

**蔡林林：**

这次的实验，我们小组验收的时候只完成了GPU/CPU混合并行的静态调度。这次实验的代码难度并不是很高，代码量也不多，但还是有很多需要注意的细节。首先就是任务调度的时候，得先用CPU的一个线程来分配任务，再用剩余的CPU线程来完成部分任务。如果使用一次OpenMP，用其中一个CPU线程来控制GPU并行，就会遇到一个问题，因为本程序OpenMP并行的for循环，这样一来就不便于实现用一个CPU线程来控制GPU调度，用剩余CPU线程做计算。我们小组采用的解决办法是嵌套并行，先设置两个CPU线程，用其中一个CPU线程调度GPU并行，另外一个CPU线程来再并行化做

CPU计算。经测试，这个方案完成的效果还是比较理想的。在一个要注意的问题就是计算结果的合并，由于CPU和GPU在计算过程中使用的内存不一样，所以GPU计算完后需要把显存里的计算结果保存到内存里，这时就会涉及到结果合并的问题，如果没解决好，很容易导致计算结果被覆盖。当然，验收的过程中老师给了很多的建议，特别是动态调度方面，老师建议我们设置一个挣抢区间，让CPU和GPU争抢式的完成部分任务，而大部分任务则采用静态调度。我们小组讨论后觉得这个方法是可行的，也是必要的，从我们小组之前做的实验来看，CPU和GPU混合并行调度的时候效率并不是很稳定，设置一个争抢区间，可以有效的解决这个问题。还有就是要增强程序的通用性，让程序在不同的硬件环境能自动调整CPU和GPU的任务分配，实现比较理想的并行调度。要解决这个问题，首先得在程序中实现获得机器硬件信息，根据这些信息来判断CPU和GPU的计算性能，从而自动调整CPU和GPU的任务调度，我觉得这是一个有挑战性的任务，如果实现了，可以增加这份代码的实际应用及传播的意义。

**杨治均：**

GPU这个概念我并不陌生，因为经常玩游戏的人都会知道显卡性能很大一部分就体现在GPU上，它是显卡的心脏，是图像处理专用的CPU。我接触最多的是NVIDIA显卡，相较来说，我认为AMD显卡图像质量更好，但是性能没有N卡高，在CUDA编程中需要的也是N卡及其相关的驱动程序。本次的内容就是关于CUDA的编程，我的电脑上安装的是GeForce GT 755M，按照步骤搭建环境，这些都比较的顺利。不过到编程的话就不太容易了，我的编程能力不强，实现CUDA的并行编程都只是勉强完成，根据网上的一些提示，磕磕碰碰的还算是做出来了，多跟组员交流，向他们请教，我还是打算尝试加入CPU的运算。当然现在只是尝试，还没有结果，不过我觉得都做到这一步了，那就继续学习下去吧。总感觉有很多的东西不经过多多尝试和练习，是做不好的。

**李清城：**

CUDA主要通过c语言等运行于游戏图形的各类行业，然而在CUDA中，但是在未来，运用到CUDA的地方将会更多，这种高性能的计算，有着传统GPU算法不具有的优势。此次实验需要N卡和相关驱动，按照步骤，查看例程，程序并不是很难。然而在试验中，也遇到过一些关于配置不能够让实验结果出现极大的误差，并不能体现CUDA编程的高性能处理的优势，然而通过一些配置的更换，终于将CUDA的程序能够正常的运行和实现。CUDA对于我来说，领域仍有许多未知的东西，仍需要进一步的探索

**马卿：**

经过本次实训，我获益良多。首先，通过与队友们之间的互相合作，内容分配，让我们的工作效率显著提升，并让我们各有深刻的收获与分享。其次，通过对CUDA的学习，让我对并行初有了解，从而让我又掌握了一份知识。最后，经过此次对知识的探索与运用，让我的学习能力更提升一步，相信对以后的工作生活大有裨益。

1. **附代码**

**main.cpp：**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <windows.h>

#include <time.h>

#include "omp.h"

#include <cuda\_runtime.h>

#include <cuda\_gl\_interop.h>

#define SOFTENING 1e-9f

typedef struct { float4 \*pos, \*vel; } BodySystem;

void randomizeBodies(float \*data, int n)

{

for (int i = 0; i < n; i++)

{

data[i] = 2.0f \* (rand() / (float)RAND\_MAX) - 1.0f;

}

}

extern "C" void cumain(int nBodies, float \*h\_buf, float dt, int n);

void get\_bodyForce(float4 \*p, float4 \*v, float dt, int n, int div, float \*h\_buf)

{

omp\_set\_nested(1);//允许嵌套并行

#pragma omp parallel num\_threads(2)

{

int tid=omp\_get\_thread\_num();

int tnum=omp\_get\_num\_threads();

//printf("stid=%d,tnum=%d\n",tid,tnum);

if(tid == 0)

{

clock\_t s,f;

s = clock();

cumain(n, h\_buf, dt, div);

f = clock();

double duration = (double)(f - s)/CLOCKS\_PER\_SEC;

}

else

{

clock\_t s,f;

s = clock();

#pragma omp parallel for schedule(dynamic)

for (int i = div; i < n; i++)

{

int mtid=omp\_get\_thread\_num();

int mtnum=omp\_get\_num\_threads();

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 = 1.0f / sqrtf(distSqr);

float invDist3 = invDist \* invDist \* invDist;

Fx += dx \* invDist3; Fy += dy \* invDist3; Fz += dz \* invDist3;

}

v[i].x += dt\*Fx; v[i].y += dt\*Fy; v[i].z += dt\*Fz; // 计算x,y,z方向的速率

}

f = clock();

double duration = (double)(f - s)/CLOCKS\_PER\_SEC;

//printf("cPU time=%f seconds\n",duration);

}

}

}

void bodyForce(float4 \*p, float4 \*v, float dt, int n, int div, float \*h\_buf)

{

#pragma omp parallel for schedule(dynamic)

for (int i = div; 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 = 1.0f / sqrtf(distSqr);

float invDist3 = invDist \* invDist \* invDist;

Fx += dx \* invDist3; Fy += dy \* invDist3; Fz += dz \* invDist3;

}

v[i].x += dt\*Fx; v[i].y += dt\*Fy; v[i].z += dt\*Fz; // 计算x,y,z方向的速率

}

}

void mymain(int BodyDiv)

{

int nBodies = 20000;

//if (argc > 1) nBodies = atoi(argv[1]);

clock\_t s,f;

double duration,CGTime,GTime,CTime;

const float dt = 0.01f; // 步长

const int nIters = 5; // 迭代次数

int bytes = 2\*nBodies\*sizeof(float4);

float \*h\_buf = (float\*)malloc(bytes);

BodySystem h\_p = { (float4\*)h\_buf, ((float4\*)h\_buf) + nBodies };

randomizeBodies(h\_buf, 8\*nBodies); // 初始化body信息

//printf("h\_buf=%p\n",h\_buf);

GTime = 0;

for (int iter = 1; iter <= nIters; iter++)

{

s = clock();

//printf("n=%d,dt=%lf,h\_buf=%p,div=%d\n",nBodies,dt,h\_buf,BodyDiv);

cumain(nBodies, h\_buf, dt, nBodies);

for (int i = 0 ; i < nBodies; i++)

{ // 更新各body位置坐标

h\_p.pos[i].x += h\_p.vel[i].x\*dt;

h\_p.pos[i].y += h\_p.vel[i].y\*dt;

h\_p.pos[i].z += h\_p.vel[i].z\*dt;

}

f = clock();

duration = (double)(f - s)/CLOCKS\_PER\_SEC;

if( iter > 1)

GTime += duration;

}

printf("\n\ngPU time: %.3f seconds\n", GTime);

CTime = 0;

for (int iter = 1; iter <= nIters; iter++)

{

s = clock();

bodyForce(h\_p.pos, h\_p.vel, dt, nBodies, 0 ,h\_buf);

for (int i = 0 ; i < nBodies; i++)

{ // 更新各body位置坐标

h\_p.pos[i].x += h\_p.vel[i].x\*dt;

h\_p.pos[i].y += h\_p.vel[i].y\*dt;

h\_p.pos[i].z += h\_p.vel[i].z\*dt;

}

f = clock();

duration = (double)(f - s)/CLOCKS\_PER\_SEC;

if( iter > 1)

CTime += duration;

}

printf("cPU time: %.3f seconds\n", CTime);

double rate = CTime/(GTime+CTime);

CGTime = 0;

//BodyDiv = 11000;

double BRate = (double)BodyDiv/(double)nBodies;

printf("Theory Rate:%lf\nCurrent Rate:%lf\nBodyDiv:%d\n",rate,BRate,BodyDiv);

for (int iter = 1; iter <= nIters; iter++)

{

s = clock();

get\_bodyForce(h\_p.pos, h\_p.vel, dt, nBodies, BodyDiv ,h\_buf);

for (int i = 0 ; i < nBodies; i++)

{ // 更新各body位置坐标

h\_p.pos[i].x += h\_p.vel[i].x\*dt;

h\_p.pos[i].y += h\_p.vel[i].y\*dt;

h\_p.pos[i].z += h\_p.vel[i].z\*dt;

}

f = clock();

duration = (double)(f - s)/CLOCKS\_PER\_SEC;

if( iter > 1)

CGTime += duration;

}

printf("cPU and gPU time: %.3f seconds\n\n", CGTime);

double CRate,GRate;

CRate = CTime/CGTime;

GRate = GTime/CGTime;

printf("相对gPU加速比：%lf\n",GRate);

printf("相对cPU加速比：%lf\n",CRate);

free(h\_buf);

}

void main()

{

int n;

freopen("cuda-result1.out","w",stdout);

for(n = 10000; n<=15000; n+=100)

mymain(n);

fclose(stdout);

}

**cuda.cu：**

/\*此程序默认引力常量为“1”，每个body的质量为“1”\*/

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <windows.h>

#include <cuda\_runtime.h>

#include <cuda\_gl\_interop.h>

#define BLOCK\_SIZE 256

#define SOFTENING 1e-9f

typedef struct { float4 \*pos, \*vel; } BodySystem;

\_\_global\_\_

void bodyForce(float4 \*p, float4 \*v, float dt, int n, int div)

{

int i = blockDim.x \* blockIdx.x + threadIdx.x;

if (i < div)

{//计算“i”的受力

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

for (int tile = 0; tile < gridDim.x; tile++)

{

\_\_shared\_\_ float3 spos[BLOCK\_SIZE]; //共享内存

float4 tpos = p[tile \* blockDim.x + threadIdx.x];

spos[threadIdx.x] = make\_float3(tpos.x, tpos.y, tpos.z);

\_\_syncthreads(); //块内线程同步

for (int j = 0; j < BLOCK\_SIZE; j++)

{

float dx = spos[j].x - p[i].x;

float dy = spos[j].y - p[i].y;

float dz = spos[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; // 计算x,y,z方向的受力

}

\_\_syncthreads(); //块内线程同步

}

v[i].x += dt\*Fx; v[i].y += dt\*Fy; v[i].z += dt\*Fz; // 计算x,y,z方向的速率

}

}

extern "C" void cumain(int nBodies, float \*h\_buf, float dt, int div);

extern "C" void cumain(int nBodies, float \*h\_buf, float dt, int div)

{

int bytes0 = 2\*nBodies\*sizeof(float4);

int bytes1 = 2\*div\*sizeof(float4);

float \*d\_buf;

cudaMalloc(&d\_buf, bytes0);

BodySystem d\_p = { (float4\*)d\_buf, ((float4\*)d\_buf) + nBodies };

int nBlocks = (nBodies + BLOCK\_SIZE - 1) / BLOCK\_SIZE;

cudaMemcpy(d\_buf, h\_buf, bytes0, cudaMemcpyHostToDevice);

bodyForce<<<nBlocks, BLOCK\_SIZE>>>(d\_p.pos, d\_p.vel, dt, nBodies, div);

cudaMemcpy(h\_buf, d\_buf, bytes1, cudaMemcpyDeviceToHost);

cudaFree(d\_buf);

}