# OpenCL 实验手册

目录

[OpenCL 实验手册 1](#_Toc360915143)

[1. 实验目的 2](#_Toc360915144)

[2. 实验环境 2](#_Toc360915145)

[2.1 OpenCL简介 2](#_Toc360915146)

[2.2 NBody简介 2](#_Toc360915147)

[2.3 实验要求 3](#_Toc360915148)

[3. 实验步骤 4](#_Toc360915149)

[3.1 安装显卡驱动程序和SDK 4](#_Toc360915150)

[3.1.1 安装显卡驱动 4](#_Toc360915151)

[3.1.2 安装SDK 4](#_Toc360915152)

[3.2 在Visual Studio中配置环境 4](#_Toc360915153)

[3.2.1 添加附加包含目录 4](#_Toc360915154)

[3.2.2 添加附加库目录 5](#_Toc360915155)

[3.2.3 添加附加依赖项，如下图所示： 6](#_Toc360915156)

[3.3 设计数据结构 7](#_Toc360915157)

[3.4 编写图像显示代码 7](#_Toc360915158)

[3.5 设计基于OpenCL的并行计算模型 8](#_Toc360915159)

[3.6 设计编写OpenCL kernel 9](#_Toc360915160)

[3.7 编写OpenCL host 17](#_Toc360915161)

[4. 实验结果与分析 33](#_Toc360915162)

[4.1 实验结果 33](#_Toc360915163)

[4.2 结果分析 33](#_Toc360915164)

# 实验目的

1. 学习了解基于OpenCL异构并行计算的理念
2. 熟悉OpenCL并行编程开发工具和环境
3. 掌握利用AMD APP SDK进行OpenCL编程开发
4. 学习基于OpenCL的并行开发的设计思想

# 实验环境

## OpenCL简介

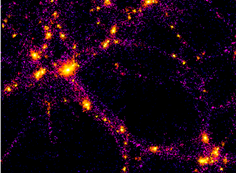
OpenCL（全称Open Computing Language，开放运算语言）是第一个面向异构系统通用目的并行编程的开放式、免费标准，也是一个统一的编程环境，便于软件开发人员为高性能计算服务器、[桌面](http://baike.baidu.com/view/79807.htm)计算系统、手持设备编写高效轻便的代码，而且广泛适用于多核心处理器(CPU)、图形处理器(GPU)、Cell类型架构以及[数字信号处理器](http://baike.baidu.com/view/529231.htm)(DSP)等其他并行处理器，在游戏、娱乐、科研、医疗等各种领域都有广阔的发展前景。

OpenCL类似于另外两个开放的工业标准[OpenGL](http://baike.baidu.com/view/9222.htm)和[OpenAL](http://baike.baidu.com/view/1355367.htm)，这两个标准分别用于三维图形和计算机音频方面。OpenCL扩展了GPU用于图形生成之外的能力。OpenCL由非盈利性技术组织[Khronos Group](http://baike.baidu.com/view/3507329.htm)掌管。当前最新标准为OpenCL 1.2。

更多详细信息和参考文档可在官网网址下载：<http://www.khronos.org/>

## NBody简介

NBody系统模拟在粒子间的相互物理力作用，如在大量粒子的物理力（重力，库仑力）的影响下运动情况，可应用与分子动力学、天体物理学和拉格朗日力学。NBody典型应用场景是模拟星系系统。每个粒子表示一个星星，多个粒子之间的相互作用，就呈现出星系的效果。下图为一个粒子模拟星系的图片：



* NBody 问题的理论基础

初始输入一组nbodies, *b1, b2, …, bn*, 每个body的质量为*mi*, 速度为*vi*，位置是*Pi*，body之间的距离是*dij*，在*dt*时间间隔后计算所有body新的位置和速度，应用了柔化因子*fc*。

1. 计算加速度，:

是 和间距离，

此处，为的质量，是距离向量的大小

1. 由初始位置pi和速度vi以及得到的ai，计算新的位置和速度

## 实验要求

软件：

* + - Windows XP SP3 or later
    - 显卡驱动程序
    - AMD APP SDK
    - Visual Studio 2010

硬件：

* + - 带有支持OpenCL的图形处理器的计算机

# 实验步骤

## 安装显卡驱动程序和SDK

### 安装显卡驱动

先查看实验机的显卡型号，到相应的官网下载显卡驱动并安装。例如：本次实验使用的是AMD的显卡（型号：Radeon HD 6970），显卡驱动下载网址：<http://support.amd.com/us/gpudownload/Pages/index.aspx>

### 安装SDK

去以下网址下载AMD APP SDK 并安装：

<http://developer.amd.com/sdks/AMDAPPSDK/downloads/pages/AMDAPPSDKDownloadArchive.aspx>

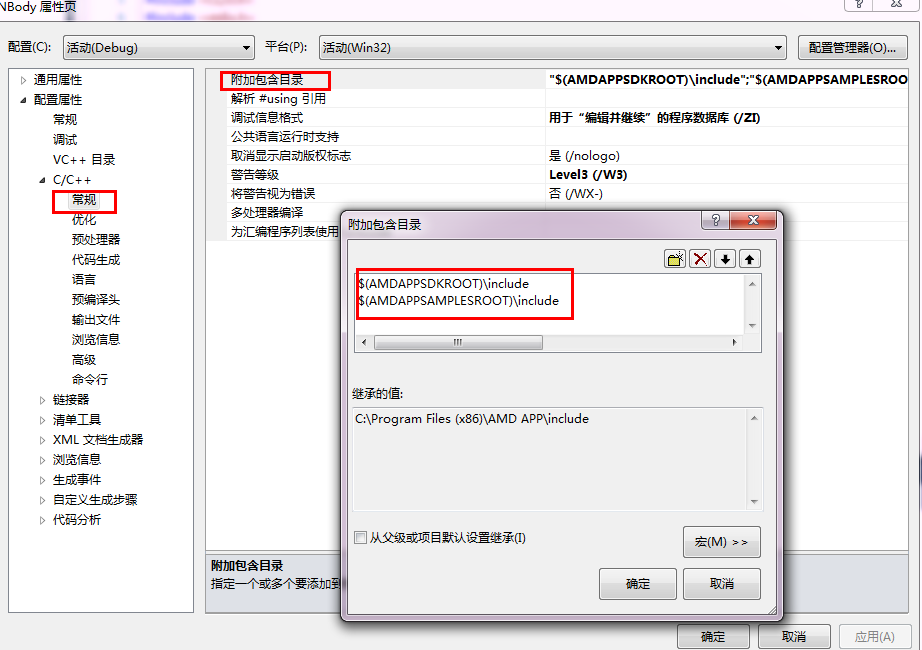
## 在Visual Studio中配置环境

### 添加附加包含目录

打开项目属性页，配置属性->C/C++->常规，在附加包含目录项目中添加以下项：

"$(AMDAPPSDKROOT)\include";"$(AMDAPPSAMPLESROOT)\include"

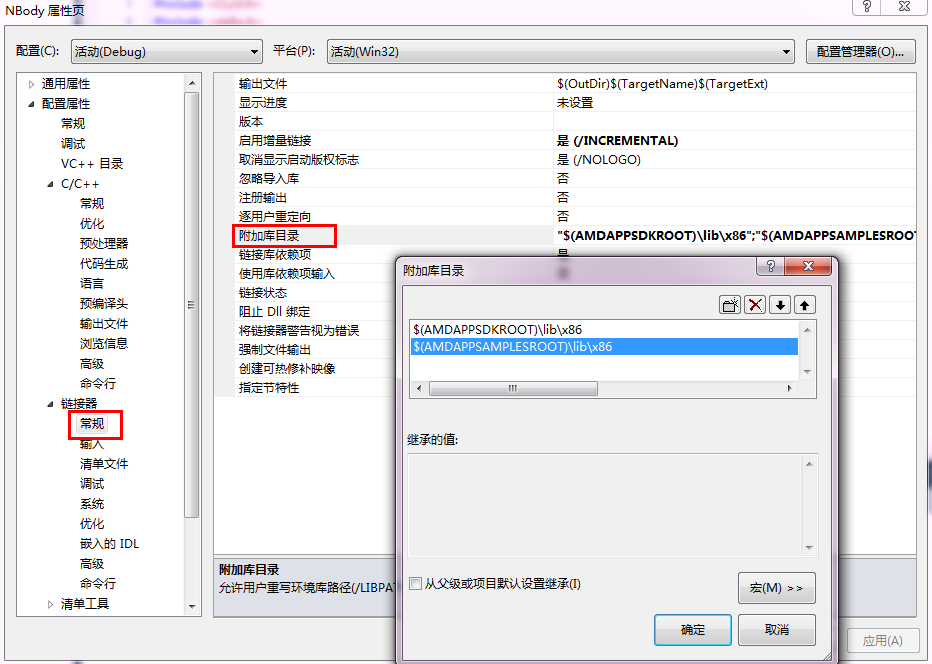
其中$(AMDAPPSDKROOT)是AMD APP SDK安装后新增的系统环境变量，include目录包含OpenCL相关的头文件。$(AMDAPPSAMPLESROOT)是SDK自带的Samples安装时路径，其include附带了OpenGL相关头文件，由于此次的实验需要使用OpenGL。



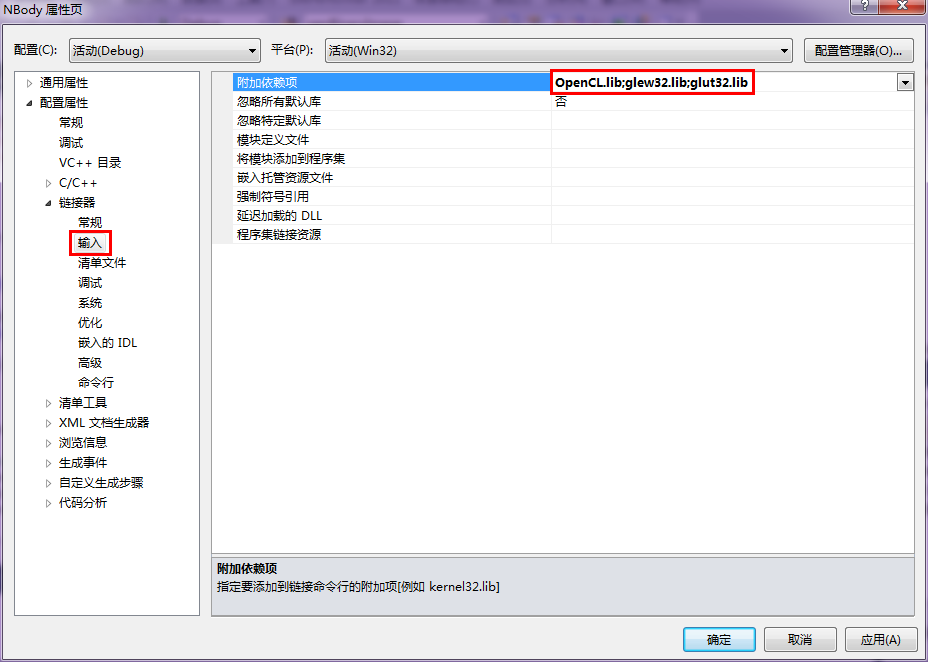
### 添加附加库目录

打开项目属性页，配置属性->链接器->常规，在附加库目录项目中添加OpenCL.lib，glut32.lib，glew32.lib 等库文件路径项：

"$(AMDAPPSDKROOT)\lib\x86"; $(AMDAPPSAMPLESROOT)\lib\x86



### 添加附加依赖项，如下图所示：



## 设计数据结构

根据Nbody问题中的需要，计算过程中使用的数据类型应该为浮点数，描述每个质点需要四个数据，分别是关于位置的三维坐标和质点的质量，在OpenCL 标准中提供了内置数据类型float4，float4是一个四维的浮点向量，可以同时记录4个数据，这样每个计算单元可以一次访问所需的四个数据，提高内存访问的性能。

## 编写图像显示代码

实验中，选择使用OpenGL展现粒子运动的轨迹。以下是OpenGL实现的展示功能的代码，具体的OpenCL编程详细见3.7编写OpenCL host。

/\*\*

\* @brief OpenGL display function

\*/

void displayfunc()

{

glClearColor(0.0 ,0.0, 0.0, 0.0);

glClear(GL\_COLOR\_BUFFER\_BIT);

glClear(GL\_DEPTH\_BUFFER\_BIT);

glPointSize(1.0);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE);

glEnable(GL\_BLEND);

glDepthMask(GL\_FALSE);

glColor3f(1.0f,0.6f,0.0f);

NBody \*nb = (NBody \*)me;

if (nb->isFirstLuanch)

{

//Calling kernel for calculatig subsequent positions

nb->runCLKernels();

nb->isFirstLuanch = false;

return;

}

clWaitForEvents(1, &nb->glEvent);

glBegin(GL\_POINTS);

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

{

//divided by 300 just for scaling

glVertex3d(pos[i \* 4 + 0] / 300, pos[i \* 4 + 1] / 300, pos[i \* 4 + 2] / 300);

}

glEnd();

//Calling kernel for calculatig subsequent positions

nb->runCLKernels();

glFlush();

glutSwapBuffers();

}

## 设计基于OpenCL的并行计算模型

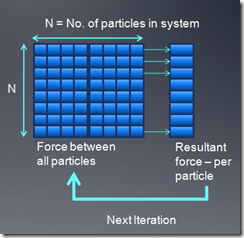
每个特定时刻粒子的轨迹都是根据其前一时刻的轨迹以及当前其他粒子对其的引力作用决定的，所受合力、由此形成的加速度、新速度、新位置等信息都是需要计算的。由于每个粒子的轨迹信息计算都是相互独立的，可以同时进行，因此，Nbody问题算法具有很高的计算并行性。计算并行模式包括任务并行和数据并行。本次实验处理的是大量的粒子信息数据，是属于数据并行的计算模型。

**实现的设计具体采用以下方式：**

1. 执行模型：对每个质点安排一个workItem计算其新的轨迹信息，包括 加速度、新的速度和轨迹位置。
2. 内存模型：

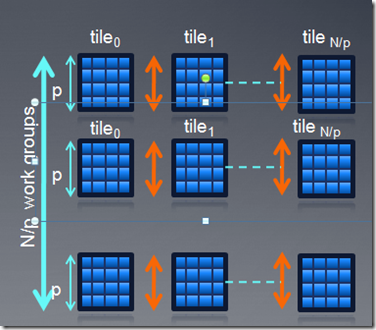
**优化前：**

通过OpenCL机制来优化，直接是每个Item负责计算自己的轨迹，计算过程中访问的是全局的数据。在这种实现中，每次都要从global memory中读取其它粒子的位置，速度。整个计算模型的内存模型如下图所示：



**使用local memory优化:**

我们可以通过local memory进行优化，一个粒子数据读进来以后，可以被p\*p个线程共用，p\*p即为workgroup的大小，对于每个粒子，我们通过迭代p\*p的tile，累积得到最终结果。内存模型如图所示：



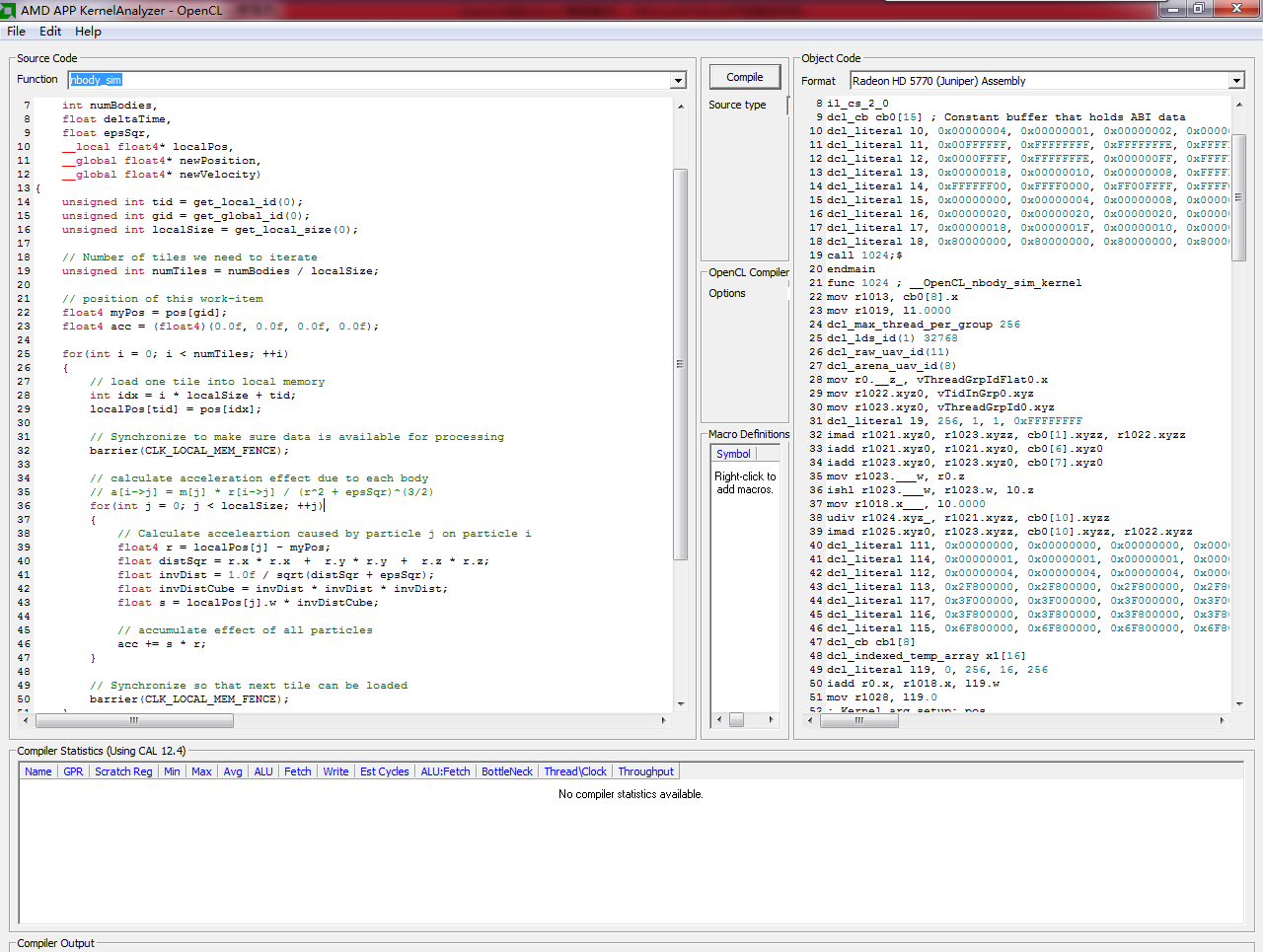
将整个工作空间划分成numTitles=numBodies / groupsize个组tiles，每个Item计算时会迭代numTitle次，第I次迭代中Item从global memory中拷贝其在第I个分组中对应位置处的质点数据到组内的local memory中，这里需要控制同步，当同组内的所有Items都拷贝完数据后才继续进行其他操作。具体描述如上图中黑色箭头所示。

**思考：**这种优化的优势在哪里？请从内存访问效率、计算开销等方面进行分析

## 设计编写OpenCL kernel

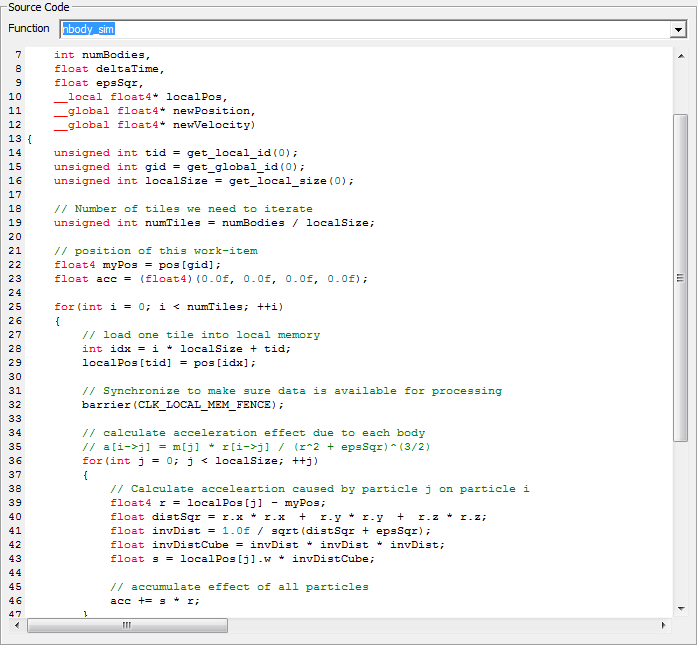
1. **Kernel开发工具**

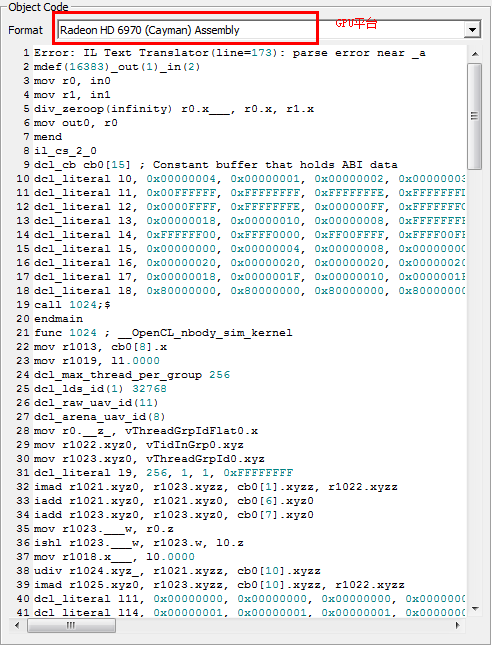
OpenCL kernel函数是独立于编译器的文件，以.cl为后缀名。可使用任何编辑器进行开发，AMD APP SDK 自带了KernelAnalyzer工具，可帮助开发者单独编译调试kernel函数。

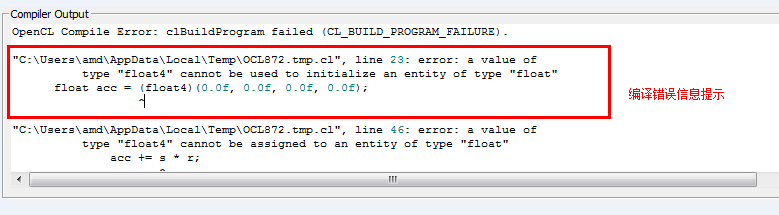


该工具主要由三部分构成

* Kernel Source code，源代码
* Object code，在OpenCL设备上编译结果
* Compiler output，编译输出信息







1. **NBody模拟的串行算法如下：**

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

{

ax = ay = az = 0;

// Loop over all particles "j”

for (j=0; j<n; j++) {

//Calculate Displacement

dx=x[j]-x[i];

dy=y[j]-y[i];

dz=z[j]-z[i];

// small eps is delta added for dx,dy,dz = 0

invr= 1.0/sqrt(dx\*dx+dy\*dy+dz\*dz +eps);

invr3 = invr\*invr\*invr;

f=m[ j ]\*invr3;

// Accumulate acceleration

ax += f\*dx;

ay += f\*dy;

az += f\*dx;

}

// Use ax, ay, az to update particle positions

}

其中一个较小的正数eps是为了避免两个质点重合的情况出现距离为0，导致除0错误。

1. **OpenCL 并行实现NBody**

使用OpenCL机制来并行计算NBody，对每个粒子计算作用在它上面的合力，然后求在合力作用下，delta时间内粒子的新位置，并把这个新位置当作下次计算的输入参数。则kernel代码如下：

\_\_kernel void nbody\_sim\_notile(

\_\_global float4\* pos ,

\_\_global float4\* vel,

int numBodies,

float deltaTime,

float epsSqr,

\_\_local float4\* localPos,

\_\_global float4\* newPosition,

\_\_global float4\* newVelocity)

{

unsigned int tid = get\_local\_id(0);

unsigned int gid = get\_global\_id(0);

unsigned int localSize = get\_local\_size(0);

// position of this work-item

float4 myPos = pos[gid];

float4 acc = (float4)(0.0f, 0.0f, 0.0f, 0.0f);

// load one tile into local memory

int idx = tid \* localSize + tid;

localPos[tid] = pos[idx];

// calculate acceleration effect due to each body

// a[i->j] = m[j] \* r[i->j] / (r^2 + epsSqr)^(3/2)

for(int j = 0; j < numBodies; ++j)

{

// Calculate acceleartion caused by particle j on particle i

localPos[tid] = pos[j];

float4 r = localPos[j] - myPos;

float distSqr = r.x \* r.x + r.y \* r.y + r.z \* r.z;

float invDist = 1.0f / sqrt(distSqr + epsSqr);

float invDistCube = invDist \* invDist \* invDist;

float s = localPos[j].w \* invDistCube;

// accumulate effect of all particles

acc += s \* r;

}

float4 oldVel = vel[gid];

// updated position and velocity

float4 newPos = myPos + oldVel \* deltaTime + acc \* 0.5f \* deltaTime \* deltaTime;

newPos.w = myPos.w;

float4 newVel = oldVel + acc \* deltaTime;

// write to global memory

newPosition[gid] = newPos;

newVelocity[gid] = newVel;

}

1. **使用Local Memory再优化**

利用LocalMemory来对kernel中读写性能进行优化，避免GlobalMemory的多次访问和访问冲突。方法就如上节所述，优化过的代码如下：

\_\_kernel void nbody\_sim(

\_\_global float4\* pos ,

\_\_global float4\* vel,

int numBodies,

float deltaTime,

float epsSqr,

\_\_local float4\* localPos,

\_\_global float4\* newPosition,

\_\_global float4\* newVelocity)

{

unsigned int tid = get\_local\_id(0);

unsigned int gid = get\_global\_id(0);

unsigned int localSize = get\_local\_size(0);

// Number of tiles we need to iterate

unsigned int numTiles = numBodies / localSize;

// position of this work-item

float4 myPos = pos[gid];

float4 acc = (float4)(0.0f, 0.0f, 0.0f, 0.0f);

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

{

// load one tile into local memory

int idx = i \* localSize + tid;

localPos[tid] = pos[idx];

// Synchronize to make sure data is available for processing

barrier(CLK\_LOCAL\_MEM\_FENCE);

// calculate acceleration effect due to each body

// a[i->j] = m[j] \* r[i->j] / (r^2 + epsSqr)^(3/2)

for(int j = 0; j < localSize; ++j)

{

// Calculate acceleartion caused by particle j on particle i

float4 r = localPos[j] - myPos;

float distSqr = r.x \* r.x + r.y \* r.y + r.z \* r.z;

float invDist = 1.0f / sqrt(distSqr + epsSqr);

float invDistCube = invDist \* invDist \* invDist;

float s = localPos[j].w \* invDistCube;

// accumulate effect of all particles

acc += s \* r;

}

// Synchronize so that next tile can be loaded

barrier(CLK\_LOCAL\_MEM\_FENCE);

}

float4 oldVel = vel[gid];

// updated position and velocity

float4 newPos = myPos + oldVel \* deltaTime + acc \* 0.5f \* deltaTime \* deltaTime;

newPos.w = myPos.w;

float4 newVel = oldVel + acc \* deltaTime;

// write to global memory

newPosition[gid] = newPos;

newVelocity[gid] = newVel;

}

## 编写OpenCL host

1. 设计此实验所需要的变量。

cl\_float delT; /\*\*< dT (timestep) \*/

cl\_float espSqr; /\*\*< Softening Factor\*/

cl\_float\* initPos; /\*\*< initial position \*/

cl\_float\* initVel; /\*\*< initial velocity \*/

cl\_float\* vel; /\*\*< Output velocity \*/

cl\_float\* refPos; /\*\*< Reference position \*/

cl\_float\* refVel; /\*\*< Reference velocity \*/

cl\_context context; /\*\*< CL context \*/

cl\_device\_id \*devices; /\*\*< CL device list \*/

cl\_mem currPos; /\*\*< Position of partciles \*/

cl\_mem currVel; /\*\*< Velocity of partciles \*/

cl\_mem newPos; /\*\*< Position of partciles \*/

cl\_mem newVel; /\*\*< Velocity of partciles \*/

cl\_command\_queue commandQueue; /\*\*< CL command queue \*/

cl\_program program; /\*\*< CL program \*/

cl\_kernel kernel; /\*\*< CL kernel \*/

size\_t groupSize; /\*\*< Work-Group size \*/

cl\_int numParticles;

1. 构造函数，初始化参数。

NBody(void)

{

delT = 0.005f;

espSqr = 50.0f;

initPos = NULL;

initVel = NULL;

vel = NULL;

groupSize = 256;

numParticles = 1024;//粒子个数

}

1. 编写random函数，用来随机产生最大数和最小数。实现代码如下：

random(float randMax, float randMin)

{

float result;

result =(float)rand() / (float)RAND\_MAX;

return ((1.0f - result) \* randMin + result \*randMax);

}

1. 编写setupNBody：为主机内存数组分配存储空间并进行初始化，实现代码如下：

int setupNBody()

{

// make sure numParticles is multiple of group size

numParticles = (cl\_int)(((size\_t)numParticles

< groupSize) ? groupSize : numParticles);

numParticles = (cl\_int)((numParticles / groupSize) \* groupSize);

numBodies = numParticles;

initPos = (cl\_float\*)malloc(numBodies \* sizeof(cl\_float4));

initVel = (cl\_float\*)malloc(numBodies \* sizeof(cl\_float4));

pos = (cl\_float\*)\_aligned\_malloc(numBodies \* sizeof(cl\_float4), 16);

vel = (cl\_float\*)\_aligned\_malloc(numBodies \* sizeof(cl\_float4), 16);

/\* initialization of inputs \*/

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

{

int index = 4 \* i;

// First 3 values are position in x,y and z direction

for(int j = 0; j < 3; ++j)

{

initPos[index + j] = random(3, 50);

}

// Mass value

initPos[index + 3] = random(1, 1000);

// First 3 values are velocity in x,y and z direction

for(int j = 0; j < 3; ++j)

{

initVel[index + j] = 0.0f;

}

// unused

initVel[3] = 0.0f;

}

memcpy(pos, initPos, 4 \* numBodies \* sizeof(cl\_float));

memcpy(vel, initVel, 4 \* numBodies \* sizeof(cl\_float));

return 0;

}

1. 编写setupCL函数：进行OpenCL相关的初始化，查找平台，设置上下文，设备，命令队列，内存buffer以及构建kernel执行程序，实现代码如下：

int setupCL()

{

cl\_uint status;

cl\_platform\_id platform;

//创建平台对象

status = clGetPlatformIDs( 1, &platform, NULL );

//创建GPU设备，若需要替换成CPU并行计算，使用CL\_DEVICE\_TYPE\_CPU

clGetDeviceIDs( platform, CL\_DEVICE\_TYPE\_GPU,

1,

&device,

NULL);

cl\_context\_properties cps[3] =

{

CL\_CONTEXT\_PLATFORM,

(cl\_context\_properties)platform,

0

};

cl\_int ev;

context = clCreateContextFromType(

cps,

CL\_DEVICE\_TYPE\_ALL,

NULL,

NULL,

&ev);

queue = clCreateCommandQueue(

context,

device,

0,

&ev);

currPos = clCreateBuffer(

context,

CL\_MEM\_READ\_WRITE,

numBodies \* sizeof(cl\_float4),

0,

&ev);

status = clEnqueueWriteBuffer(queue,

currPos,

1,

0,

numBodies \* sizeof(cl\_float4),

pos,

0,

0,

0);

/\* Create memory objects for position \*/

newPos = clCreateBuffer(

context,

CL\_MEM\_READ\_WRITE,

numBodies \* sizeof(cl\_float4),

0,

&ev);

currVel = clCreateBuffer(

context,

CL\_MEM\_READ\_WRITE,

numBodies \* sizeof(cl\_float4),

0,

&ev);

status = clEnqueueWriteBuffer(queue,

currVel,

1,

0,

numBodies \* sizeof(cl\_float4),

vel,

0,

0,

0);

/\* Create memory objects for velocity \*/

newVel = clCreateBuffer(

context,

CL\_MEM\_READ\_ONLY,

numBodies \* sizeof(cl\_float4),

0,

&ev);

const char \* filename = "nbody.cl";

std::string sourceStr;

status = convertToString(filename, sourceStr);

const char \* source = sourceStr.c\_str();

size\_t sourceSize[] = { strlen(source) };

//创建程序对象

cl\_program program = clCreateProgramWithSource(

context,

1,

&source,

sourceSize,

NULL);

//编译程序对象

status = clBuildProgram( program, 1, &device, NULL, NULL, NULL );

if(status != 0)

{

printf("clBuild failed:%d\n", status);

char tbuf[0x10000];

clGetProgramBuildInfo(program, device, CL\_PROGRAM\_BUILD\_LOG, 0x10000, tbuf, NULL);

printf("\n%s\n", tbuf);

return -1;

}

/\* get a kernel object handle for a kernel with the given name \*/

kernel = clCreateKernel(

program,

"nbody\_sim",//nobody\_sim, noboy\_sim\_notile,在这儿切换不同的kernel函数

&ev);

return 0;

}

1. 编写setup函数：执行设置，调用setupNBody函数和setupCL函数，实现代码如下：

int setup()

{

if(setupNBody() != 0)

return 1;

cl\_int retValue = setupCL();

return 0;

}

1. 编写setupCLKernels：设置kernel参量的值，用来传入kernel进行计算，实现代码如下：

int setupCLKernels()

{

cl\_int status;

/\* Set appropriate arguments to the kernel \*/

/\* Particle positions \*/

status = clSetKernelArg(

kernel,

0,

sizeof(cl\_mem),

(void\*)&currPos);

/\* Particle velocity \*/

status = clSetKernelArg(

kernel,

1,

sizeof(cl\_mem),

(void \*)&currVel);

/\* numBodies \*/

status = clSetKernelArg(

kernel,

2,

sizeof(cl\_int),

(void \*)&numBodies);

/\* time step \*/

status = clSetKernelArg(

kernel,

3,

sizeof(cl\_float),

(void \*)&delT);

/\* upward Pseudoprobability \*/

status = clSetKernelArg(

kernel,

4,

sizeof(cl\_float),

(void \*)&espSqr);

/\* local memory \*/

status = clSetKernelArg(

kernel,

5,

256 \* 4 \* sizeof(float),

NULL);

/\* Particle positions \*/

status = clSetKernelArg(

kernel,

6,

sizeof(cl\_mem),

(void\*)&newPos);

/\* Particle velocity \*/

status = clSetKernelArg(

kernel,

7,

sizeof(cl\_mem),

(void \*)&newVel);

return 0;

}

1. 编写runCLKernels函数：执行kernel代码，获取kernel的开始执行时间和执行结束时间，实现代码如下：

int runCLKernels(){

cl\_int status;

cl\_event events[1];

/\*

\* Enqueue a kernel run call.

\*/

size\_t globalThreads[] = {numBodies};

size\_t localThreads[] = {groupSize};

status = clEnqueueNDRangeKernel(

queue,

kernel,

1,

NULL,

globalThreads,

localThreads,

0,

NULL,

NULL);

status = clFinish(queue);

/\* Copy data from new to old \*/

status = clEnqueueCopyBuffer(queue,

newPos,

currPos,

0,

0,

sizeof(cl\_float4) \* numBodies,

0,

0,

0);

status = clEnqueueCopyBuffer(queue,

newVel,

currVel,

0,

0,

sizeof(cl\_float4) \* numBodies,

0,

0,

0);

status = clFinish(queue);

/\* Enqueue readBuffer\*/

status = clEnqueueReadBuffer(

queue,

currPos,

CL\_TRUE,

0,

numBodies\* sizeof(cl\_float4),

pos,

0,

NULL,

&events[0]);

/\* Wait for the read buffer to finish execution \*/

status = clWaitForEvents(1, &events[0]);

clReleaseEvent(events[0]);

return 0;

}

1. 编写run函数：运行OpenCL Nbody。实现代码如下：

int run()

{

/\* Arguments are set and execution call is enqueued on command buffer \*/

if(setupCLKernels() != 0)

{

return 0;

}

runCLKernels();

return 0;

}

1. 编写GLInit函数，初始化GL，编写idle函数，编写reShape函数，对显示窗口进行初始化，编写displayfunc函数对实验结果进行展示，keyboardFunc函数，设置键盘控制运行结果的情况。总之，这几个函数是OpenGL编程，是初始化OpenGL显示窗口并对实验结果进行显示，实现代码如下：

GLInit()

{

glClearColor(0.0 ,0.0, 0.0, 0.0);

glClear(GL\_COLOR\_BUFFER\_BIT);

glClear(GL\_DEPTH\_BUFFER\_BIT);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

}

void idle()

{

glutPostRedisplay();

}

void

reShape(int w,int h)

{

glViewport(0, 0, w, h);

glViewport(0, 0, w, h);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

gluPerspective(45.0f, w/h, 1.0f, 1000.0f);

gluLookAt (0.0, 0.0, -2.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0);

}

void displayfunc()

{

glClearColor(0.0 ,0.0, 0.0, 0.0);

glClear(GL\_COLOR\_BUFFER\_BIT);

glClear(GL\_DEPTH\_BUFFER\_BIT);

glPointSize(1.0);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE);

glEnable(GL\_BLEND);

glDepthMask(GL\_FALSE);

glColor3f(1.0f,0.6f,0.0f);

NBody \*nb = (NBody \*)me;

if (nb->isFirstLuanch)

{

//Calling kernel for calculatig subsequent positions

nb->runCLKernels();

nb->isFirstLuanch = false;

return;

}

clWaitForEvents(1, &nb->glEvent);

glBegin(GL\_POINTS);

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

{

//divided by 300 just for scaling

glVertex3d(pos[i \* 4 + 0] / 300, pos[i \* 4 + 1] / 300, pos[i \* 4 + 2] / 300);

}

glEnd();

//Calling kernel for calculatig subsequent positions

nb->runCLKernels();

glFlush();

glutSwapBuffers();

}

void

keyboardFunc(unsigned char key, int mouseX, int mouseY)

{

switch(key)

{

// If the user hits escape or Q, then exit

// ESCAPE\_KEY = 27

case 27:

case 'q':

case 'Q':

{

if(((NBody\*)me)->cleanup() != SDK\_SUCCESS)

exit(1);

else

exit(0);

}

default:

break;

}

}

1. 编写cleanup函数：清楚之前分配的内存空间等，实现代码如下：

cleanup()

{

// Releases OpenCL resources (Context, Memory etc.)

cl\_int status;

status = clReleaseKernel(kernel);

CHECK\_OPENCL\_ERROR(status, "clReleaseKernel failed.(kernel)");

status = clReleaseProgram(program);

CHECK\_OPENCL\_ERROR(status, "clReleaseProgram failed.(program)");

status = clReleaseMemObject(currPos);

CHECK\_OPENCL\_ERROR(status, "clReleaseMemObject failed.(currPos)");

status = clReleaseMemObject(currVel);

CHECK\_OPENCL\_ERROR(status, "clReleaseMemObject failed.(currVel)");

status = clReleaseMemObject(newPos);

CHECK\_OPENCL\_ERROR(status, "clReleaseMemObject failed.(newPos)");

status = clReleaseMemObject(newVel);

CHECK\_OPENCL\_ERROR(status, "clReleaseMemObject failed.(newVel)");

status = clReleaseCommandQueue(commandQueue);

CHECK\_OPENCL\_ERROR(status, "clReleaseCommandQueue failed.(commandQueue)");

status = clReleaseContext(context);

CHECK\_OPENCL\_ERROR(status, "clReleaseContext failed.(context)");

return SDK\_SUCCESS;

}

1. 编写main函数，以将整个执行过程串接起来，形成一个可以运行的代码。实现如下：

int

main(int argc, char \* argv[])

{

NBody clNBody;

me = &clNBody;

cl\_int retValue = clNBody.setup();

if(clNBody.run() != 0)

return 1;

// Run in graphical window if requested

glutInit(&argc, argv);

glutInitWindowPosition(100,10);

glutInitWindowSize(600,600);

glutInitDisplayMode( GLUT\_RGB | GLUT\_DOUBLE );

glutCreateWindow("nbody simulation");

GLInit();

glutDisplayFunc(displayfunc);

glutReshapeFunc(reShape);

glutIdleFunc(idle);

glutKeyboardFunc(keyboardFunc);

glutMainLoop();

if(clNBody.cleanup()!=0)

return 1;

return 0;}

1. 编写计算fps的类，比对效率。

class FPS\_COUNTER

{

public:

FPS\_COUNTER() : fps(0.0f), lastTime(0.0f), frames(0), time(0.0f)

{}

~FPS\_COUNTER() {}

void Update(void);

float GetFps(void) { return fps; }

protected:

float fps;

float lastTime;

int frames;

float time;

};

void FPS\_COUNTER::Update(void)

{

//keep track of time passed and frame count

time=timeGetTime()\*0.001f;

++frames;

//If a second has passed

if(time-lastTime>1.0f)

{

fps=frames/(time-lastTime); //update the number of frames per second

lastTime=time; //set time for the start of the next count

frames=0; //reset fps for this second

}

}

# 实验结果与分析

## 实验结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 质点个数 | 1k | 2k | 4k | 8k | 10k | 16k | 32k |
| CPU FPS | 220 |  |  |  |  |  |  |
| GPU FPS | 698 |  |  |  |  |  |  |

由于fps不停变化，可取前10次fps求平均值。

实验机环境GPU（AMD Radeon HD 6970），CPU（AMD FX-8150 8-core）

## 结果分析