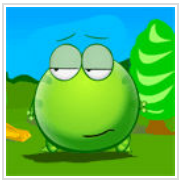
登录 | 注册

网络资源是无限的

目录视图 摘要视图 RSS 订阅

个人资料



fengbingchun

访问： 2252522次
积分： 25003
等级： BLOC 7
排名： 第202名

原创： 341篇 转载： 144篇
译文： 0篇 评论： 1434条

文章分类

Android (9)
ActiveX (18)
Bar Code (16)
Caffe (20)
C# (5)
CImg (4)
Contour Detection (9)
CxImage (6)
Code::Blocks (3)
Cloud Computing (1)
C/C++ (82)
CUDA (10)
CMake (3)
Design Patterns (25)
Database/Dataset (4)
Deep Learning (9)
Eclipse (3)
Emgu CV (1)
Eigen (1)
FFmpeg (1)
Feature Extraction (1)
FreeType (1)
Face (8)
GPU (3)
Git (3)
GCC (1)
GDAL (5)

《GPU高性能编程CUDA实战》中代码整理

2015-05-24 19:35 5081人阅读 评论(2) 收藏 举报

分类： CUDA（10）

CUDA架构专门为GPU计算设计了一种全新的模块，目的是减轻早期GPU计算中存在的一些限制，而正是这些限制使得之前的GPU在通用计算中没有得到广泛的应用。

使用**CUDA C**来编写代码的前提条件包括：(1)、支持CUDA的图形处理器，即由NVIDIA推出的GPU显卡，要求显存超过256MB；(2)、NVIDIA设备驱动程序，用于实现应用程序与支持CUDA的硬件之间的通信，确保安装最新的驱动程序，注意选择与开发环境相符的图形卡和操作系统；(3)、CUDA开发工具箱即CUDA Toolkit，此工具箱中包括一个编译GPU代码的编译器；(4)、标准C编译器，即CPU编译器。CUDA C应用程序将在两个不同的处理器上执行计算，因此需要两个编译器。其中一个编译器为GPU编译代码，而另一个为CPU编译代码。

一般，将**CPU**以及系统的内存称为主机**(Host)**，而将**GPU**及其内存称为设备**(Device)**。在**GPU**设备上执行的函数通常称为核函数**(Kernel)**。

cudaMalloc函数使用限制总结：(1)、可以将cudaMalloc()分配的指针传递给在设备上执行的函数；(2)、可以在设备代码中使用cudaMalloc()分配的指针进行内存读/写操作；(3)、可以将cudaMalloc()分配的指针传递给在主机上执行的函数；(4)、不能在主机代码中使用cudaMalloc()分配的指针进行内存读/写操作。

不能使用标准C的free()函数来释放cudaMalloc()分配的内存；要释放cudaMalloc()分配的内存，需要调用cudaFree()。

设备指针的使用方式与标准**C**中指针的使用方式完全一样。主机指针只能访问主机代码中的内存，而设备指针也只能访问设备代码中的内存。

在主机代码中可以通过调用cudaMemcpy()来访问设备上的内存。

有可能在单块卡上包含了两个或多个GPU。

在集成的**GPU**上运行代码，可以与**CPU**共享内存。

计算功能集的版本为**1.3**或者更高的显卡才能支持双精度浮点数的计算。

尖括号的第一个参数表示设备在执行核函数时使用的并行线程块的数量，

并行线程块集合也称为一个线程格(Grid)。线程格既可以是一维的线程块集合，也可以是二维的线程块集合。

GPU有着完善的内存管理机制，它将强行结束所有违反内存访问规则的进程。

在启动线程块数组时，数组每一维的最大数量都不能超过65535.这是一种硬件限制,如果启动的线程块数量超过了这个限值,那么程序将运行失败。

CUDA运行时将线程块(Block)分解为多个线程。当需要启动多个并行线程块时，只需将尖括号中的第一个参数由1改为想要启动的线程块数量。在尖括号中，第二个参数表示CUDA运行时在每个线程块中创建的线程数量。

HTML (3)
Image Recognition (8)
Image Processing (18)
Image Registration (13)
ImageMagick (3)
Java (5)
Linux (20)
Log (2)
Makefile (2)
Mathematical Knowledge (6)
Multi-thread (4)
Matlab (33)
MFC (8)
MinGW (3)
Mac (1)
Neural Network (13)
OCR (9)
Office (2)
OpenCL (2)
OpenSSL (7)
OpenCV (86)
OpenGL (2)
OpenGL ES (3)
OpenMP (3)
Photoshop (1)
Python (4)
Qt (1)
SIMD (14)
Software Development (4)
System architecture (2)
Skia (1)
SVN (1)
Software Testing (4)
Shell (2)
Socket (3)
Target Detection (2)
Target Tracking (2)
VC6 (6)
VS2008 (16)
VS2010 (4)
VS2013 (3)
vigra (2)
VLC (5)
VLFeat (1)
wxWidgets (1)
Watermark (4)
Windows7 (6)
Windows Core Programming (9)
XML (2)

Free Codes

pudn
freecode
Peter's Functions
CodeProject
SourceCodeOnline
Computer Vision Source Code
Codesoso
Digital Watermarking
SourceForge
HackChina
oschina

硬件将线程块的数量限制为不超过**65535**.同样，对于启动核函数时每个线程块中的线程数量，硬件也进行了限制。具体来说，最大的线程数量不能超过设备属性结构中maxThreadsPerBlock域的值。这个值并不固定，有的是512，有的是1024.

内置变量**blockDim**，对于所有线程块来说，这个变量是一个常数，保存的是线程块中每一维的线程数量。

内置变量**gridDim**，对于所有线程块来说，这个变量是一个常数，用来保存线程格每一维的大小，即每个线程格中线程块的数量。

内置变量**blockIdx**，变量中包含的值就是当前执行设备代码的线程块的索引。

内置变量**threadIdx**，变量中包含的值就是当前执行设备代码的线程索引。

CUDA运行时允许启动一个二维线程格，并且线程格中的每个线程块都是一个三维的线程数组。

CUDA C支持共享内存：可以将CUDA C的关键字__share__添加到变量声明中，这将使这个变量驻留在共享内存中。CUDA C编译器对共享内存中的变量与普通变量将分别采取不同的处理方式。

CUDA架构将确保，除非线程块中的每个线程都执行了__syncthreads()，否则没有任何线程能执行__syncthreads()之后的指令。

由于在GPU上包含有数百个数学计算单元，因此性能瓶颈通常并不在于芯片的数学计算吞吐量，而是在于芯片的内存带宽。

常量内存用于保存在核函数执行期间不会发生变化的数据。NVIDIA硬件提供了64KB的常量内存，并且对常量内存采取了不同于标准全局内存的处理方式。在某些情况下，用常量内存来替换全局内存能有效地减少内存带宽。要使用常量内存，需在变量前面加上__constant__关键字。

在CUDA架构中，线程束是指一个包含**32**个线程的集合，这个线程集合被“编织在一起”并且以“步调一致(Lockstep)”的形式执行。在程序中的每一行，线程束中的每个线程都将在不同的数据上执行相同的指令。

纹理内存是在**CUDA C**程序中可以使用的另一种只读内存。与常量内存类似的是，纹理内存同样缓存在芯片上，因此在某些情况中，它能够减少对内存的请求并提供更高效的内存带宽。纹理缓存是专门为那些在内存访问模式中存在大量空间局部性(Spatial Locality)的图形应用程序而设计的。

NVIDIA将GPU支持的各种功能统称为计算功能集(Compute Capability)。高版本计算功能集是低版本计算功能集的超集。

只有**1.1**或者更高版本的**GPU**计算功能集才能支持全局内存上的原子操作。此外，只有1.2或者更高版本的GPU计算功能集才能支持共享内存上的原子操作。CUDA C支持多种原子操作。

C库函数malloc函数将分配标准的，可分页的(Pageble)主机内存。而cudaHostAlloc函数将分配页锁定的主机内存。页锁定内存也称为固定内存(Pinned Memory)或者不可分页内存，它有一个重要的属性：操作系统将不会对这块内存分页并交换到磁盘上，从而确保了该内存始终驻留在物理内存上。因此，操作系统能够安全地使某个应用程序访问该内存的物理地址，因为这块内存将不会被破坏或者重新定位。

固定内存是一把双刃剑。当使用固定内存时，你将失去虚拟内存的所有功能。特别是，在应用程序中使用每个页锁定内存时都需要分配物理内存，因为这些内存不能交换到磁盘上。这意味着，与使用标准的malloc函数调用相比，系统将更快地耗尽内存。因此，应用程序在物理内存较少的机器上会运行失败，而且意味着应用程序将影响在系统上运行的其它应用程序的性能。

建议，仅对cudaMemcpy()调用中的源内存或者目标内存，才使用页锁定内存，并且在不再需要使用它们时立即释放，而不是等到应用程序关闭时才释放。

关闭

CUDA流表示一个**GPU**操作队列，并且该队列中的操作将以指定的顺序执行。

通过使用零拷贝内存，可以避免**CPU**和**GPU**之间的显式复制操作。

对于零拷贝内存，独立GPU和集成GPU，带来的性能提升是不同的。对于集成GPU，使用零拷贝内存通常都会带来性能提升，因为内存在物理上与主机是共享的。将缓冲区声明为零拷贝内存的唯一作用就是避免不必要的数据复制。但是，所有类型的固定内存都存在一定的局限性，零拷贝内存同样不例外。每个固定内存都会占用系统的可用物理内存，这最终将降低系统的性能。对于独立GPU，当输入内存和输出内存都只能使用一次时，那么在独立GPU上使用零拷贝内存将带来性能提升。但由于GPU不会缓存零拷贝内存的内容，如果多次读取内存，那么最终将

- libsvm
- joys99
- CodeForge
- cvchina
- tesseract-ocr
- sift
- TiRG
- imgSeek
- OpenSURF

Friendly Link

- OpenCL
- Python
- poesia-filter
- TortoiseSVN
- imgSeek
- Notepad
- Beyond Compare
- CMake
- VIGRA
- CodeGuru
- vchome
- aforgenet
- CVLAB
- Doxygen
- Coursera
- OpenMP

Technical Forum

- Matlab China
- OpenCV China
- The CImg Library
- Open Computer Vision Library
- CxImage
- ImageMagick
- ImageMagick China
- OpenCV_China
- Subversion China

得不偿失，还不如一开始就将数据复制到GPU。

CUDA工具箱(CUDAToolkit)包含了两个重要的工具库：(1)、**CUFFT**(Fast FourierTransform，快速傅里叶变换)库；(2)、**CUBLAS**(Basic Linear Algebra Subprograms,BLAS)是一个线性代数函数库。

NPP(NVIDIA Performance Primitives)称为NVIDIA性能原语，它是一个函数库，用来执行基于CUDA加速的数据处理操作，它的基本功能集合主要侧重于图像处理和视频处理。

新建一个基于CUDA的测试工程testCUDA，此工程中除了包括common文件外，还添加了另外三个文件，分别为testCUDA.cu、funset.cu、funset.cuh，这三个文件包括了书中绝大部分的测试代码：

testCUDA.cu:

[cpp]

```cpp
#include "funset.cuh"
#include <iostream>
#include "book.h"
#include "cpu_bitmap.h"
#include "gpu_anim.h"

using namespace std;

int test1();//简单的两数相加
int test2();//获取GPU设备相关属性
int test3();//通过线程块索引来计算两个矢量和
int test4();//Julia的CUDA实现
int test5();//通过线程索引来计算两个矢量和
int test6();//通过线程块索引和线程索引来计算两个矢量和
int test7();//ripple的CUDA实现
int test8();//点积运算的CUDA实现
int test9();//Julia的CUDA实现，加入了线程同步函数__syncthreads()
int test10();//光线跟踪(Ray Tracing)实现，没有常量内存+使用事件来计算GPU运行时间
int test11();//光线跟踪(Ray Tracing)实现，使用常量内存+使用事件来计算GPU运行时间
int test12();//模拟热传导，使用纹理内存，有些问题
int test13();//模拟热传导，使用二维纹理内存，有些问题
int test14();//ripple的CUDA+OpenGL实现
int test15();//模拟热传导,CUDA+OpenGL实现，有些问题
int test16();//直方图计算，利用原子操作函数atomicAdd实现
int test17();//固定内存的使用
int test18();//单个stream的使用
int test19();//多个stream的使用
int test20();//通过零拷贝内存的方式实现点积运算
int test21();//使用多个GPU实现点积运算

int main(int argc, char* argv[])
{
    test21();

    cout<<"ok!"<<endl;
    return 0;
}

int test1()
{
    int a = 2, b = 3, c = 0;
    int* dev_c = NULL;
    HANDLE_ERROR(cudaMalloc((void**)&dev_c, sizeof(int)));

    //尖括号表示要将一些参数传递给CUDA编译器和运行时系统
    //尖括号中这些参数并不是传递给设备代码的参数，而是告诉运行时如何启动设备代码,
    //传递给设备代码本身的参数是放在圆括号中传递的，就像标准的函数调用一样
    add<<<1, 1>>>(a, b, dev_c);
    HANDLE_ERROR(cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost));

    printf("%d + %d = %d\n", a, b, c);
    cudaFree(dev_c);

    return 0;
}

int test2()
{
    int count = -1;
    HANDLE_ERROR(cudaGetDeviceCount(&count));
    printf("device count: %d\n", count);
```

关闭

Technical Blog

邹宇华
深之JohnChen
HUNNISH
周伟明
superdont
carson2005
OpenHero
Netman(Linux)
wqvbjhc
yang_xian521
gnuhpc
gnuhpc
千里8848
CVART
tornadomeet
gotosuc
onezeros
hellogv
abcjennifer
crzy_sparrow

评论排行

Windows7 32位机上，O	(120)
tiny-cnn开源库的使用(MI	(93)
Ubuntu 14.04 64位机上不	(89)
tesseract-ocr3.02字符识	(63)
Windows7上使用VS201:	(47)
tesseract-ocr	(42)
图像配准算法	(41)
Windows 7 64位机上Ope	(36)
OpenCV中resize函数五利	(34)
小波矩特征提取matlab代	(30)

最新评论

Tesseract-OCR 3.04在Windows
fengbingchun: @ilikede:没有密码，那个commit只是提示是从哪个commit fork过来的，无需管那个

Tesseract-OCR 3.04在Windows
ilikede: 问一下，你第一句中的commit的那个密码，怎么用啊

卷积神经网络(CNN)的简单实现(
fengbingchun: @hugl950123:是需要opencv的支持，你在本地opencv的环境配好了吗，配好了就应该没...

卷积神经网络(CNN)的简单实现(
hugl950123: @fengbingchun:博主请问一下，test_CNN_predict()函数是不是需要open...

卷积神经网络(CNN)的简单实现(
hugl950123: @fengbingchun:博主请问一下，test_CNN_predict()函数是不是需要open...

卷积神经网络(CNN)的简单实现(
hugl950123: @fengbingchun:谢谢，能够成功运行了现在

卷积神经网络(CNN)的简单实现(
fengbingchun: @hugl950123:NN中一共有四个工程，它们之间没有任何关系，都是独立的，如果要运行这篇文章的...

```

    cudaDeviceProp prop;
    for (int i = 0; i < count; i++) {
        HANDLE_ERROR(cudaGetDeviceProperties(&prop, i));

        printf("   --- General Information for device %d ---\n", i);
        printf("Name:  %s\n", prop.name);
        printf("Compute capability:  %d.%d\n", prop.major, prop.minor);
        printf("Clock rate:  %d\n", prop.clockRate);
        printf("Device copy overlap:  ");
        if (prop.deviceOverlap) printf("Enabled\n");
        else printf("Disabled\n");
        printf("Kernel execution timeout :  ");
        if (prop.kernelExecTimeoutEnabled) printf("Enabled\n");
        else printf("Disabled\n");

        printf("   --- Memory Information for device %d ---\n", i);
        printf("Total global mem:  %ld\n", prop.totalGlobalMem);
        printf("Total constant Mem:  %ld\n", prop.totalConstMem);
        printf("Max mem pitch:  %ld\n", prop.memPitch);
        printf("Texture Alignment:  %ld\n", prop.textureAlignment);

        printf("   --- MP Information for device %d ---\n", i);
        printf("Multiprocessor count:  %d\n", prop.multiProcessorCount);
        printf("Shared mem per mp:  %ld\n", prop.sharedMemPerBlock);
        printf("Registers per mp:  %d\n", prop.regsPerBlock);
        printf("Threads in warp:  %d\n", prop.warpSize);
        printf("Max threads per block:  %d\n", prop.maxThreadsPerBlock);
        printf("Max thread dimensions:  (%d, %d, %d)\n", prop.maxThreadsDim[0],
            prop.maxThreadsDim[1], prop.maxThreadsDim[2]);
        printf("Max grid dimensions:  (%d, %d, %d)\n", prop.maxGridSize[0],
            prop.maxGridSize[1], prop.maxGridSize[2]);
        printf("\n");
    }

    int dev;

    HANDLE_ERROR(cudaGetDevice(&dev));
    printf("ID of current CUDA device:  %d\n", dev);

    memset(&prop, 0, sizeof(cudaDeviceProp));
    prop.major = 1;
    prop.minor = 3;
    HANDLE_ERROR(cudaChooseDevice(&dev, &prop));
    printf("ID of CUDA device closest to revision %d.%d:  %d\n", prop.major, prop.minor, 

    HANDLE_ERROR(cudaSetDevice(dev));

    return 0;
}

int test3()
{
    int a[NUM] = {0}, b[NUM] = {0}, c[NUM] = {0};
    int *dev_a = NULL, *dev_b = NULL, *dev_c = NULL;

    //allocate the memory on the GPU
    HANDLE_ERROR(cudaMalloc((void**)&dev_a, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_b, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_c, NUM * sizeof(int)));

    //fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<NUM; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    //copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR(cudaMemcpy(dev_a, a, NUM * sizeof(int), cudaMemcpyHostToDevice));
    HANDLE_ERROR(cudaMemcpy(dev_b, b, NUM * sizeof(int), cudaMemcpyHostToDevice));

    //尖括号中的第一个参数表示设备在执行核函数时使用的并行线程块的数量
    add_blockIdx<<<NUM,1>>>( dev_a, dev_b, dev_c );

    //copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR(cudaMemcpy(c, dev_c, NUM * sizeof(int), cudaMemcpyDeviceToHost));

    //display the results
    for (int i=0; i<NUM; i++) {
```

关闭

```
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    //free the memory allocated on the GPU
    HANDLE_ERROR(cudaFree(dev_a));
    HANDLE_ERROR(cudaFree(dev_b));
    HANDLE_ERROR(cudaFree(dev_c));

    return 0;
}

int test4()
{
    //globals needed by the update routine
    struct DataBlock {
        unsigned char* dev_bitmap;
    };

    DataBlock   data;
    CPUBitmap bitmap(DIM, DIM, &data);
    unsigned char* dev_bitmap;

    HANDLE_ERROR(cudaMalloc((void**)&dev_bitmap, bitmap.image_size()));
    data.dev_bitmap = dev_bitmap;

    //声明一个二维的线程格
    //类型dim3表示一个三维数组，可以用于指定启动线程块的数量
    //当用两个值来初始化dim3类型的变量时，CUDA运行时将自动把第3维的大小指定为1
    dim3 grid(DIM, DIM);
    kernel_julia<<<grid,1>>>(dev_bitmap);

    HANDLE_ERROR(cudaMemcpy(bitmap.get_ptr(), dev_bitmap,
        bitmap.image_size(), cudaMemcpyDeviceToHost));

    HANDLE_ERROR(cudaFree(dev_bitmap));

    bitmap.display_and_exit();

    return 0;
}

int test5()
{
    int a[NUM], b[NUM], c[NUM];
    int *dev_a = NULL, *dev_b = NULL, *dev_c = NULL;

    //在GPU上分配内存
    HANDLE_ERROR(cudaMalloc((void**)&dev_a, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_b, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_c, NUM * sizeof(int)));

    //在CPU上为数组'a'和'b'赋值
    for (int i = 0; i < NUM; i++) {
        a[i] = i;
        b[i] = i * i;
    }

    //将数组'a'和'b'复制到GPU
    HANDLE_ERROR(cudaMemcpy(dev_a, a, NUM * sizeof(int), cudaMemcpyHostToDevice));
    HANDLE_ERROR(cudaMemcpy(dev_b, b, NUM * sizeof(int), cudaMemcpyHostToDevice));

    add_threadIdx<<<1, NUM>>>(dev_a, dev_b, dev_c);

    //将数组'c'从GPU复制到CPU
    HANDLE_ERROR(cudaMemcpy(c, dev_c, NUM * sizeof(int), cudaMemcpyDeviceToHost));

    //显示结果
    for (int i = 0; i < NUM; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }

    //释放在GPU分配的内存
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}

```

关闭

```
int test6()
{
    int a[NUM], b[NUM], c[NUM];
    int *dev_a = NULL, *dev_b = NULL, *dev_c = NULL;

    //在GPU上分配内存
    HANDLE_ERROR(cudaMalloc((void**)&dev_a, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_b, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_c, NUM * sizeof(int)));

    //在CPU上为数组'a'和'b'赋值
    for (int i = 0; i < NUM; i++) {
        a[i] = i;
        b[i] = i * i / 10;
    }

    //将数组'a'和'b'复制到GPU
    HANDLE_ERROR(cudaMemcpy(dev_a, a, NUM * sizeof(int), cudaMemcpyHostToDevice));
    HANDLE_ERROR(cudaMemcpy(dev_b, b, NUM * sizeof(int), cudaMemcpyHostToDevice));

    add_blockIdx_threadIdx<<<128, 128>>>(dev_a, dev_b, dev_c);

    //将数组'c'从GPU复制到CPU
    HANDLE_ERROR(cudaMemcpy(c, dev_c, NUM * sizeof(int), cudaMemcpyDeviceToHo

    //验证GPU确实完成了我们要求的工作
    bool success = true;
    for (int i = 0; i < NUM; i++) {
        if ((a[i] + b[i]) != c[i]) {
            printf("error: %d + %d != %d\n", a[i], b[i], c[i]);
            success = false;
        }
    }

    if (success)
        printf("we did it!\n");

    //释放在GPU分配的内存
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}

int test7()
{
    DataBlock data;
    CPUAnimBitmap bitmap(DIM, DIM, &data);
    data.bitmap = &bitmap;

    HANDLE_ERROR(cudaMalloc((void**)&data.dev_bitmap, bitmap.image_size()));

    bitmap.anim_and_exit((void(*)(void*,int))generate_frame, (void(*)(void*))cleanup);

    return 0;
}

void generate_frame(DataBlock *d, int ticks)
{
    dim3 blocks(DIM/16, DIM/16);
    dim3 threads(16, 16);
    ripple_kernel<<<blocks,threads>>>(d->dev_bitmap, ticks);

    HANDLE_ERROR(cudaMemcpy(d->bitmap->get_ptr(), d->dev_bitmap, d->bitmap->image_size(),
}

//clean up memory allocated on the GPU
void cleanup(DataBlock *d)
{
    HANDLE_ERROR(cudaFree(d->dev_bitmap));
}

int test8()
{
    float *a, *b, c, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;

    //allocate memory on the cpu side
```

关闭

```
    a = (float*)malloc(NUM * sizeof(float));
    b = (float*)malloc(NUM * sizeof(float));
    partial_c = (float*)malloc(blocksPerGrid * sizeof(float));

    //allocate the memory on the GPU
    HANDLE_ERROR(cudaMalloc((void**)&dev_a, NUM * sizeof(float)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_b, NUM * sizeof(float)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_partial_c, blocksPerGrid*sizeof(float)));

    //fill in the host memory with data
    for (int i = 0; i < NUM; i++) {
        a[i] = i;
        b[i] = i*2;
    }

    //copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR(cudaMemcpy(dev_a, a, NUM * sizeof(float), cudaMemcpyHostToDevice));
    HANDLE_ERROR(cudaMemcpy(dev_b, b, NUM * sizeof(float), cudaMemcpyHostToDevice));

    dot_kernel<<<blocksPerGrid,threadsPerBlock>>>(dev_a, dev_b, dev_partial_c);

    //copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR(cudaMemcpy(partial_c, dev_partial_c, blocksPerGrid * sizeof(float), cuda

    //finish up on the CPU side
    c = 0;
    for (int i = 0; i < blocksPerGrid; i++) {
        c += partial_c[i];
    }

    //点积计算结果应该是从0到NUM-1中每个数值的平方再乘以2
    //闭合形式解
#define sum_squares(x)  (x * (x + 1) * (2 * x + 1) / 6)
    printf("Does GPU value %.6g = %.6g?\n", c, 2 * sum_squares((float)(NUM - 1)));

    //free memory on the gpu side
    HANDLE_ERROR(cudaFree(dev_a));
    HANDLE_ERROR(cudaFree(dev_b));
    HANDLE_ERROR(cudaFree(dev_partial_c));

    //free memory on the cpu side
    free(a);
    free(b);
    free(partial_c);

    return 0;
}

int test9()
{
    DataBlock data;
    CPUBitmap bitmap(DIM, DIM, &data);
    unsigned char *dev_bitmap;

    HANDLE_ERROR(cudaMalloc((void**)&dev_bitmap, bitmap.image_size()));
    data.dev_bitmap = dev_bitmap;

    dim3 grids(DIM / 16, DIM / 16);
    dim3 threads(16,16);
    julia_kernel<<<grids, threads>>>(dev_bitmap);

    HANDLE_ERROR(cudaMemcpy(bitmap.get_ptr(), dev_bitmap, bitmap.image_size(), cudaMemcpy

    HANDLE_ERROR(cudaFree(dev_bitmap));

    bitmap.display_and_exit();

    return 0;
}

int test10()
{
    DataBlock data;
    //capture the start time
    cudaEvent_t start, stop;
    HANDLE_ERROR(cudaEventCreate(&start));
    HANDLE_ERROR(cudaEventCreate(&stop));
    HANDLE_ERROR(cudaEventRecord(start, 0));

```

关闭

```
    CPUBitmap bitmap(DIM, DIM, &data);
    unsigned char *dev_bitmap;
    Sphere *s;

    //allocate memory on the GPU for the output bitmap
    HANDLE_ERROR(cudaMalloc((void**)&dev_bitmap, bitmap.image_size()));
    //allocate memory for the Sphere dataset
    HANDLE_ERROR(cudaMalloc((void**)&s, sizeof(Sphere) * SPHERES));

    //allocate temp memory, initialize it, copy to memory on the GPU, then free our temp r
    Sphere *temp_s = (Sphere*)malloc(sizeof(Sphere) * SPHERES);
    for (int i = 0; i < SPHERES; i++) {
        temp_s[i].r = rnd(1.0f);
        temp_s[i].g = rnd(1.0f);
        temp_s[i].b = rnd(1.0f);
        temp_s[i].x = rnd(1000.0f) - 500;
        temp_s[i].y = rnd(1000.0f) - 500;
        temp_s[i].z = rnd(1000.0f) - 500;
        temp_s[i].radius = rnd(100.0f) + 20;
    }

    HANDLE_ERROR(cudaMemcpy( s, temp_s, sizeof(Sphere) * SPHERES, cudaMemcpyHostToDevice)
    free(temp_s);

    //generate a bitmap from our sphere data
    dim3 grids(DIM / 16, DIM / 16);
    dim3 threads(16, 16);
    RayTracing_kernel<<<grids, threads>>>(s, dev_bitmap);

    //copy our bitmap back from the GPU for display
    HANDLE_ERROR(cudaMemcpy(bitmap.get_ptr(), dev_bitmap, bitmap.image_size(), cudaMemcpyI

    //get stop time, and display the timing results
    HANDLE_ERROR(cudaEventRecord(stop, 0));
    HANDLE_ERROR(cudaEventSynchronize(stop));
    float elapsedTime;
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
    printf("Time to generate:  %3.1f ms\n", elapsedTime);

    HANDLE_ERROR(cudaEventDestroy(start));
    HANDLE_ERROR(cudaEventDestroy(stop));

    HANDLE_ERROR(cudaFree(dev_bitmap));
    HANDLE_ERROR(cudaFree(s));

    // display
    bitmap.display_and_exit();

    return 0;
}

int test11()
{
    DataBlock data;
    //capture the start time
    cudaEvent_t start, stop;
    HANDLE_ERROR(cudaEventCreate(&start));
    HANDLE_ERROR(cudaEventCreate(&stop));
    HANDLE_ERROR(cudaEventRecord(start, 0));

    CPUBitmap bitmap(DIM, DIM, &data);
    unsigned char *dev_bitmap;

    //allocate memory on the GPU for the output bitmap
    HANDLE_ERROR(cudaMalloc((void**)&dev_bitmap, bitmap.image_size()));

    //allocate temp memory, initialize it, copy to constant memory on the GPU, then free t
    Sphere *temp_s = (Sphere*)malloc(sizeof(Sphere) * SPHERES);
    for (int i = 0; i < SPHERES; i++) {
        temp_s[i].r = rnd(1.0f);
        temp_s[i].g = rnd(1.0f);
        temp_s[i].b = rnd(1.0f);
        temp_s[i].x = rnd(1000.0f) - 500;
        temp_s[i].y = rnd(1000.0f) - 500;
        temp_s[i].z = rnd(1000.0f) - 500;
        temp_s[i].radius = rnd(100.0f) + 20;
    }

    HANDLE_ERROR(cudaMemcpyToSymbol(s, temp_s, sizeof(Sphere) * SPHERES));
```

关闭

```
    free(temp_s);

    //generate a bitmap from our sphere data
    dim3 grids(DIM / 16, DIM / 16);
    dim3 threads(16, 16);
    RayTracing_kernel<<<grids, threads>>>(dev_bitmap);

    //copy our bitmap back from the GPU for display
    HANDLE_ERROR(cudaMemcpy(bitmap.get_ptr(), dev_bitmap, bitmap.image_size(), cudaMemcpyI

    //get stop time, and display the timing results
    HANDLE_ERROR(cudaEventRecord(stop, 0));
    HANDLE_ERROR(cudaEventSynchronize(stop));
    float elapsedTime;
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
    printf("Time to generate:  %3.1f ms\n", elapsedTime);

    HANDLE_ERROR(cudaEventDestroy(start));
    HANDLE_ERROR(cudaEventDestroy(stop));

    HANDLE_ERROR(cudaFree(dev_bitmap));

    //display
    bitmap.display_and_exit();

    return 0;
}

int test12()
{
    Heat_DataBlock data;
    CPUAnimBitmap bitmap(DIM, DIM, &data);
    data.bitmap = &bitmap;
    data.totalTime = 0;
    data.frames = 0;

    HANDLE_ERROR(cudaEventCreate(&data.start));
    HANDLE_ERROR(cudaEventCreate(&data.stop));

    int imageSize = bitmap.image_size();

    HANDLE_ERROR(cudaMalloc((void**)&data.output_bitmap, imageSize));

    //assume float == 4 chars in size (ie rgba)
    HANDLE_ERROR(cudaMalloc((void**)&data.dev_inSrc, imageSize));
    HANDLE_ERROR(cudaMalloc((void**)&data.dev_outSrc, imageSize));
    HANDLE_ERROR(cudaMalloc((void**)&data.dev_constSrc, imageSize));

    HANDLE_ERROR(cudaBindTexture(NULL, texConstSrc, data.dev_constSrc, imageSize));
    HANDLE_ERROR(cudaBindTexture(NULL, texIn, data.dev_inSrc, imageSize));
    HANDLE_ERROR(cudaBindTexture(NULL, texOut, data.dev_outSrc, imageSize));

    //intialize the constant data
    float *temp = (float*)malloc(imageSize);

    for (int i = 0; i < DIM*DIM; i++) {
        temp[i] = 0;
        int x = i % DIM;
        int y = i / DIM;
        if ((x>300) && (x<600) && (y>310) && (y<601))
            temp[i] = MAX_TEMP;
    }

    temp[DIM * 100 + 100] = (MAX_TEMP + MIN_TEMP) / 2;
    temp[DIM * 700 + 100] = MIN_TEMP;
    temp[DIM * 300 + 300] = MIN_TEMP;
    temp[DIM * 200 + 700] = MIN_TEMP;

    for (int y = 800; y < 900; y++) {
        for (int x = 400; x < 500; x++) {
            temp[x + y * DIM] = MIN_TEMP;
        }
    }

    HANDLE_ERROR(cudaMemcpy(data.dev_constSrc, temp, imageSize, cudaMemcpyHostToDevice));

    //initialize the input data
    for (int y = 800; y < DIM; y++) {
        for (int x = 0; x < 200; x++) {
```

关闭

```
            temp[x+y*DIM] = MAX_TEMP;
        }
    }

    HANDLE_ERROR(cudaMemcpy(data.dev_inSrc, temp,imageSize, cudaMemcpyHostToDevice));
    free(temp);

    bitmap.anim_and_exit((void (*)(void*,int))Heat_anim_gpu, (void (*)(void*))Heat_anim_e

    return 0;
}

int test13()
{
    Heat_DataBlock data;
    CPUAnimBitmap bitmap(DIM, DIM, &data);
    data.bitmap = &bitmap;
    data.totalTime = 0;
    data.frames = 0;
    HANDLE_ERROR(cudaEventCreate(&data.start));
    HANDLE_ERROR(cudaEventCreate(&data.stop));

    int imageSize = bitmap.image_size();

    HANDLE_ERROR(cudaMalloc((void**)&data.output_bitmap, imageSize));

    //assume float == 4 chars in size (ie rgba)
    HANDLE_ERROR(cudaMalloc((void**)&data.dev_inSrc, imageSize));
    HANDLE_ERROR(cudaMalloc((void**)&data.dev_outSrc, imageSize));
    HANDLE_ERROR(cudaMalloc((void**)&data.dev_constSrc, imageSize));

    cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
    HANDLE_ERROR(cudaBindTexture2D(NULL, texConstSrc2, data.dev_constSrc, desc, DIM, DIM,
    HANDLE_ERROR(cudaBindTexture2D(NULL, texIn2, data.dev_inSrc, desc, DIM, DIM, sizeof(f
    HANDLE_ERROR(cudaBindTexture2D(NULL, texOut2, data.dev_outSrc, desc, DIM, DIM, sizeof

    //initialize the constant data
    float *temp = (float*)malloc(imageSize);
    for (int i = 0; i < DIM*DIM; i++) {
        temp[i] = 0;
        int x = i % DIM;
        int y = i / DIM;
        if ((x > 300) && ( x < 600) && (y > 310) && (y < 601))
            temp[i] = MAX_TEMP;
    }

    temp[DIM * 100 + 100] = (MAX_TEMP + MIN_TEMP) / 2;
    temp[DIM * 700 + 100] = MIN_TEMP;
    temp[DIM * 300 + 300] = MIN_TEMP;
    temp[DIM * 200 + 700] = MIN_TEMP;

    for (int y = 800; y < 900; y++) {
        for (int x = 400; x < 500; x++) {
            temp[x + y * DIM] = MIN_TEMP;
        }
    }

    HANDLE_ERROR(cudaMemcpy(data.dev_constSrc, temp, imageSize, cudaMemcpyHostToDevice));

    //initialize the input data
    for (int y = 800; y < DIM; y++) {
        for (int x = 0; x < 200; x++) {
            temp[x + y * DIM] = MAX_TEMP;
        }
    }

    HANDLE_ERROR(cudaMemcpy(data.dev_inSrc, temp,imageSize, cudaMemcpyHostToDevice));
    free(temp);

    bitmap.anim_and_exit((void (*)(void*,int))anim_gpu, (void (*)(void*))anim_exit);

    return 0;
}

void Heat_anim_gpu(Heat_DataBlock *d, int ticks)
{
    HANDLE_ERROR(cudaEventRecord(d->start, 0));

    dim3 blocks(DIM / 16, DIM / 16);
```

关闭

```
    dim3 threads(16, 16);
    CPUAnimBitmap *bitmap = d->bitmap;

    //since tex is global and bound, we have to use a flag to
    //select which is in/out per iteration
    volatile bool dstOut = true;

    for (int i = 0; i < 90; i++) {
        float *in, *out;
        if (dstOut) {
            in  = d->dev_inSrc;
            out = d->dev_outSrc;
        } else {
            out = d->dev_inSrc;
            in  = d->dev_outSrc;
        }

        Heat_copy_const_kernel<<<blocks, threads>>>(in);
        Heat_blend_kernel<<<blocks, threads>>>(out, dstOut);
        dstOut = !dstOut;
    }

    float_to_color<<<blocks, threads>>>(d->output_bitmap, d->dev_inSrc);

    HANDLE_ERROR(cudaMemcpy(bitmap->get_ptr(), d->output_bitmap, bitmap->image_size(), cud

    HANDLE_ERROR(cudaEventRecord(d->stop, 0));
    HANDLE_ERROR(cudaEventSynchronize(d->stop));
    float elapsedTime;
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, d->start, d->stop));
    d->totalTime += elapsedTime;
    ++d->frames;

    printf( "Average Time per frame:  %3.1f ms\n", d->totalTime/d->frames );
}

void anim_gpu(Heat_DataBlock *d, int ticks)
{
    HANDLE_ERROR(cudaEventRecord(d->start, 0));
    dim3 blocks(DIM / 16, DIM / 16);
    dim3 threads(16, 16);
    CPUAnimBitmap  *bitmap = d->bitmap;

    //since tex is global and bound, we have to use a flag to
    //select which is in/out per iteration
    volatile bool dstOut = true;
    for (int i = 0; i < 90; i++) {
        float *in, *out;
        if (dstOut) {
            in  = d->dev_inSrc;
            out = d->dev_outSrc;
        } else {
            out = d->dev_inSrc;
            in  = d->dev_outSrc;
        }
        copy_const_kernel<<<blocks, threads>>>(in);
        blend_kernel<<<blocks, threads>>>(out, dstOut);
        dstOut = !dstOut;
    }

    float_to_color<<<blocks, threads>>>(d->output_bitmap, d->dev_inSrc);

    HANDLE_ERROR(cudaMemcpy(bitmap->get_ptr(), d->output_bitmap, bitmap->image_size(), cud

    HANDLE_ERROR(cudaEventRecord(d->stop, 0));
    HANDLE_ERROR(cudaEventSynchronize(d->stop));
    float elapsedTime;
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, d->start, d->stop));
    d->totalTime += elapsedTime;
    ++d->frames;
    printf("Average Time per frame:  %3.1f ms\n", d->totalTime/d->frames);
}

void Heat_anim_exit(Heat_DataBlock *d)
{
    cudaUnbindTexture(texIn);
    cudaUnbindTexture(texOut);
    cudaUnbindTexture(texConstSrc);

```

关闭

```
    HANDLE_ERROR(cudaFree(d->dev_inSrc));
    HANDLE_ERROR(cudaFree(d->dev_outSrc));
    HANDLE_ERROR(cudaFree(d->dev_constSrc));

    HANDLE_ERROR(cudaEventDestroy(d->start));
    HANDLE_ERROR(cudaEventDestroy(d->stop));
}

//clean up memory allocated on the GPU
void anim_exit(Heat_DataBlock *d)
{
    cudaUnbindTexture(texIn2);
    cudaUnbindTexture(texOut2);
    cudaUnbindTexture(texConstSrc2);
    HANDLE_ERROR(cudaFree(d->dev_inSrc));
    HANDLE_ERROR(cudaFree(d->dev_outSrc));
    HANDLE_ERROR(cudaFree(d->dev_constSrc));

    HANDLE_ERROR(cudaEventDestroy(d->start));
    HANDLE_ERROR(cudaEventDestroy(d->stop));
}

int test14()
{
    GPUAnimBitmap  bitmap(DIM, DIM, NULL);

    bitmap.anim_and_exit((void (*)(uchar4*, void*, int))generate_frame_opengl, NULL);

    return 0;
}

int test15()
{
    DataBlock_opengl data;
    GPUAnimBitmap bitmap(DIM, DIM, &data);
    data.totalTime = 0;
    data.frames = 0;
    HANDLE_ERROR(cudaEventCreate(&data.start));
    HANDLE_ERROR(cudaEventCreate(&data.stop));

    int imageSize = bitmap.image_size();

    //assume float == 4 chars in size (ie rgba)
    HANDLE_ERROR(cudaMalloc((void**)&data.dev_inSrc, imageSize));
    HANDLE_ERROR(cudaMalloc((void**)&data.dev_outSrc, imageSize));
    HANDLE_ERROR(cudaMalloc((void**)&data.dev_constSrc, imageSize));

    HANDLE_ERROR(cudaBindTexture(NULL, texConstSrc ,data.dev_constSrc, imageSize));
    HANDLE_ERROR(cudaBindTexture(NULL, texIn, data.dev_inSrc, imageSize));
    HANDLE_ERROR(cudaBindTexture(NULL, texOut, data.dev_outSrc, imageSize));

    //intialize the constant data
    float *temp = (float*)malloc(imageSize);
    for (int i = 0; i < DIM*DIM; i++) {
        temp[i] = 0;
        int x = i % DIM;
        int y = i / DIM;
        if ((x>300) && (x<600) && (y>310) && (y<601))
            temp[i] = MAX_TEMP;
    }

    temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP)/2;
    temp[DIM*700+100] = MIN_TEMP;
    temp[DIM*300+300] = MIN_TEMP;
    temp[DIM*200+700] = MIN_TEMP;

    for (int y = 800; y < 900; y++) {
        for (int x = 400; x < 500; x++) {
            temp[x+y*DIM] = MIN_TEMP;
        }
    }

    HANDLE_ERROR(cudaMemcpy(data.dev_constSrc, temp, imageSize, cudaMemcpyHostToDevice));

    //initialize the input data
    for (int y = 800; y < DIM; y++) {
        for (int x = 0; x < 200; x++) {
            temp[x+y*DIM] = MAX_TEMP;
        }
```

关闭

```
    }

    HANDLE_ERROR(cudaMemcpy(data.dev_inSrc, temp, imageSize, cudaMemcpyHostToDevice));
    free(temp);

    bitmap.anim_and_exit((void (*)(uchar4*, void*, int))anim_gpu_opengl, (void (*)(void*)

    return 0;
}

void anim_gpu_opengl(uchar4* outputBitmap, DataBlock_opengl *d, int ticks)
{
    HANDLE_ERROR(cudaEventRecord(d->start, 0));
    dim3 blocks(DIM / 16, DIM / 16);
    dim3 threads(16, 16);

    //since tex is global and bound, we have to use a flag to select which is in/out per
    volatile bool dstOut = true;
    for (int i = 0; i < 90; i++) {
        float *in, *out;
        if (dstOut) {
            in  = d->dev_inSrc;
            out = d->dev_outSrc;
        } else {
            out = d->dev_inSrc;
            in  = d->dev_outSrc;
        }
        Heat_copy_const_kernel_opengl<<<blocks, threads>>>(in);
        Heat_blend_kernel_opengl<<<blocks, threads>>>(out, dstOut);
        dstOut = !dstOut;
    }

    float_to_color<<<blocks, threads>>>(outputBitmap, d->dev_inSrc);

    HANDLE_ERROR(cudaEventRecord(d->stop, 0));
    HANDLE_ERROR(cudaEventSynchronize(d->stop));
    float elapsedTime;
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, d->start, d->stop));
    d->totalTime += elapsedTime;
    ++d->frames;
    printf("Average Time per frame:  %3.1f ms\n", d->totalTime/d->frames);
}

void anim_exit_opengl(DataBlock_opengl *d)
{
    HANDLE_ERROR(cudaUnbindTexture(texIn));
    HANDLE_ERROR(cudaUnbindTexture(texOut));
    HANDLE_ERROR(cudaUnbindTexture(texConstSrc));
    HANDLE_ERROR(cudaFree(d->dev_inSrc));
    HANDLE_ERROR(cudaFree(d->dev_outSrc));
    HANDLE_ERROR(cudaFree(d->dev_constSrc));

    HANDLE_ERROR(cudaEventDestroy(d->start));
    HANDLE_ERROR(cudaEventDestroy(d->stop));
}

int test16()
{
    unsigned char *buffer = (unsigned char*)big_random_block(SIZE);

    //capture the start time starting the timer here so that we include the cost of
    //all of the operations on the GPU.  if the data were already on the GPU and we just
    //timed the kernel the timing would drop from 74 ms to 15 ms.  Very fast.
    cudaEvent_t start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    // allocate memory on the GPU for the file's data
    unsigned char *dev_buffer;
    unsigned int *dev_histo;
    HANDLE_ERROR(cudaMalloc((void**)&dev_buffer, SIZE));
    HANDLE_ERROR(cudaMemcpy(dev_buffer, buffer, SIZE, cudaMemcpyHostToDevice));

    HANDLE_ERROR(cudaMalloc((void**)&dev_histo, 256 * sizeof(int)));
    HANDLE_ERROR(cudaMemset(dev_histo, 0, 256 * sizeof(int)));

    //kernel launch - 2x the number of mps gave best timing
    cudaDeviceProp prop;
```

关闭

```
    HANDLE_ERROR(cudaGetDeviceProperties(&prop, 0));
    int blocks = prop.multiProcessorCount;
    histo_kernel<<<blocks*2, 256>>>(dev_buffer, SIZE, dev_histo);

    unsigned int histo[256];
    HANDLE_ERROR(cudaMemcpy(histo, dev_histo, 256 * sizeof(int), cudaMemcpyDeviceToHost))

    //get stop time, and display the timing results
    HANDLE_ERROR(cudaEventRecord(stop, 0));
    HANDLE_ERROR(cudaEventSynchronize(stop));
    float elapsedTime;
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
    printf("Time to generate:  %3.1f ms\n", elapsedTime);

    long histoCount = 0;
    for (int i=0; i<256; i++) {
        histoCount += histo[i];
    }
    printf("Histogram Sum:  %ld\n", histoCount);

    //verify that we have the same counts via CPU
    for (int i = 0; i < SIZE; i++)
        histo[buffer[i]]--;
    for (int i = 0; i < 256; i++) {
        if (histo[i] != 0)
            printf("Failure at %d!\n", i);
    }

    HANDLE_ERROR(cudaEventDestroy(start));
    HANDLE_ERROR(cudaEventDestroy(stop));
    cudaFree(dev_histo);
    cudaFree(dev_buffer);
    free(buffer);

    return 0;
}

float cuda_malloc_test(int size, bool up)
{
    cudaEvent_t start, stop;
    int *a, *dev_a;
    float elapsedTime;

    HANDLE_ERROR(cudaEventCreate(&start));
    HANDLE_ERROR(cudaEventCreate(&stop));

    a = (int*)malloc(size * sizeof(*a));
    HANDLE_NULL(a);
    HANDLE_ERROR(cudaMalloc((void**)&dev_a,size * sizeof(*dev_a)));

    HANDLE_ERROR(cudaEventRecord(start, 0));

    for (int i=0; i<100; i++) {
        if (up)
            HANDLE_ERROR(cudaMemcpy(dev_a, a, size * sizeof( *dev_a ), cudaMemcpyHostToDe
        else
            HANDLE_ERROR(cudaMemcpy(a, dev_a, size * sizeof(*dev_a), cudaMemcpyDeviceToHo
    }
    HANDLE_ERROR(cudaEventRecord(stop, 0));
    HANDLE_ERROR(cudaEventSynchronize(stop));
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));

    free(a);
    HANDLE_ERROR(cudaFree(dev_a));
    HANDLE_ERROR(cudaEventDestroy(start));
    HANDLE_ERROR(cudaEventDestroy(stop));

    return elapsedTime;
}

float cuda_host_alloc_test(int size, bool up)
{
    cudaEvent_t start, stop;
    int *a, *dev_a;
    float elapsedTime;

    HANDLE_ERROR(cudaEventCreate(&start));
    HANDLE_ERROR(cudaEventCreate(&stop));

```

关闭

```
    HANDLE_ERROR(cudaHostAlloc((void**)&a,size * sizeof(*a), cudaHostAllocDefault));
    HANDLE_ERROR(cudaMalloc((void**)&dev_a, size * sizeof(*dev_a)));

    HANDLE_ERROR(cudaEventRecord(start, 0));

    for (int i=0; i<100; i++) {
        if (up)
            HANDLE_ERROR(cudaMemcpy(dev_a, a,size * sizeof(*a), cudaMemcpyHostToDevice));
        else
            HANDLE_ERROR(cudaMemcpy(a, dev_a,size * sizeof(*a), cudaMemcpyDeviceToHost));
    }
    HANDLE_ERROR(cudaEventRecord(stop, 0));
    HANDLE_ERROR(cudaEventSynchronize(stop));
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));

    HANDLE_ERROR(cudaFreeHost(a));
    HANDLE_ERROR(cudaFree(dev_a));
    HANDLE_ERROR(cudaEventDestroy(start));
    HANDLE_ERROR(cudaEventDestroy(stop));

    return elapsedTime;
}

int test17()
{
    float elapsedTime;
    float MB = (float)100 * SIZE * sizeof(int) / 1024 / 1024;

    //try it with cudaMalloc
    elapsedTime = cuda_malloc_test(SIZE, true);
    printf("Time using cudaMalloc:  %3.1f ms\n", elapsedTime);
    printf("\tMB/s during copy up:  %3.1f\n", MB/(elapsedTime/1000));

    elapsedTime = cuda_malloc_test(SIZE, false);
    printf("Time using cudaMalloc:  %3.1f ms\n", elapsedTime);
    printf("\tMB/s during copy down:  %3.1f\n", MB/(elapsedTime/1000));

    //now try it with cudaHostAlloc
    elapsedTime = cuda_host_alloc_test(SIZE, true);
    printf("Time using cudaHostAlloc:  %3.1f ms\n", elapsedTime);
    printf("\tMB/s during copy up:  %3.1f\n", MB/(elapsedTime/1000));

    elapsedTime = cuda_host_alloc_test(SIZE, false);
    printf("Time using cudaHostAlloc:  %3.1f ms\n", elapsedTime);
    printf("\tMB/s during copy down:  %3.1f\n", MB/(elapsedTime/1000));

    return 0;
}

int test18()
{
    cudaDeviceProp prop;
    int whichDevice;
    HANDLE_ERROR(cudaGetDevice(&whichDevice));
    HANDLE_ERROR(cudaGetDeviceProperties(&prop, whichDevice));

    if (!prop.deviceOverlap) {
        printf("Device will not handle overlaps, so no speed up from streams\n");
        return 0;
    }

    cudaEvent_t start, stop;
    float elapsedTime;
    cudaStream_t stream;
    int *host_a, *host_b, *host_c;
    int *dev_a, *dev_b, *dev_c;

    //start the timers
    HANDLE_ERROR(cudaEventCreate(&start));
    HANDLE_ERROR(cudaEventCreate(&stop));

    //initialize the stream
    HANDLE_ERROR(cudaStreamCreate(&stream));

    //allocate the memory on the GPU
    HANDLE_ERROR(cudaMalloc((void**)&dev_a, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_b, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_c, NUM * sizeof(int)));

```

关闭

```
    //allocate host locked memory, used to stream
    HANDLE_ERROR(cudaHostAlloc((void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAll
    HANDLE_ERROR(cudaHostAlloc((void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAll
    HANDLE_ERROR(cudaHostAlloc((void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAll

    for (int i=0; i<FULL_DATA_SIZE; i++) {
        host_a[i] = rand();
        host_b[i] = rand();
    }

    HANDLE_ERROR(cudaEventRecord(start, 0));
    //now loop over full data, in bite-sized chunks
    for (int i=0; i<FULL_DATA_SIZE; i+= NUM) {
        //copy the locked memory to the device, async
        HANDLE_ERROR(cudaMemcpyAsync(dev_a, host_a+i, NUM * sizeof(int), cudaMemcpyHostTo
        HANDLE_ERROR(cudaMemcpyAsync(dev_b, host_b+i, NUM * sizeof(int), cudaMemcpyHostTo

        singlestream_kernel<<<NUM/256, 256, 0, stream>>>(dev_a, dev_b, dev_c);

        //copy the data from device to locked memory
        HANDLE_ERROR(cudaMemcpyAsync(host_c+i, dev_c, NUM * sizeof(int), cudaMemcpyDevice

    }

    // copy result chunk from locked to full buffer
    HANDLE_ERROR(cudaStreamSynchronize(stream));

    HANDLE_ERROR(cudaEventRecord(stop, 0));
    HANDLE_ERROR(cudaEventSynchronize(stop));
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
    printf("Time taken:  %3.1f ms\n", elapsedTime);

    //cleanup the streams and memory
    HANDLE_ERROR(cudaFreeHost(host_a));
    HANDLE_ERROR(cudaFreeHost(host_b));
    HANDLE_ERROR(cudaFreeHost(host_c));
    HANDLE_ERROR(cudaFree(dev_a));
    HANDLE_ERROR(cudaFree(dev_b));
    HANDLE_ERROR(cudaFree(dev_c));
    HANDLE_ERROR(cudaStreamDestroy(stream));

    return 0;
}

int test19()
{
    cudaDeviceProp prop;
    int whichDevice;
    HANDLE_ERROR(cudaGetDevice(&whichDevice));
    HANDLE_ERROR(cudaGetDeviceProperties(&prop, whichDevice));
    if (!prop.deviceOverlap) {
        printf( "Device will not handle overlaps, so no speed up from streams\n" );
        return 0;
    }

    //start the timers
    cudaEvent_t start, stop;
    HANDLE_ERROR(cudaEventCreate(&start));
    HANDLE_ERROR(cudaEventCreate(&stop));

    //initialize the streams
    cudaStream_t stream0, stream1;
    HANDLE_ERROR(cudaStreamCreate(&stream0));
    HANDLE_ERROR(cudaStreamCreate(&stream1));

    int *host_a, *host_b, *host_c;
    int *dev_a0, *dev_b0, *dev_c0;//为第0个流分配的GPU内存
    int *dev_a1, *dev_b1, *dev_c1;//为第1个流分配的GPU内存

    //allocate the memory on the GPU
    HANDLE_ERROR(cudaMalloc((void**)&dev_a0, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_b0, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_c0, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_a1, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_b1, NUM * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_c1, NUM * sizeof(int)));

    //allocate host locked memory, used to stream
    HANDLE_ERROR(cudaHostAlloc((void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAll
```

关闭

```
    HANDLE_ERROR(cudaHostAlloc((void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAll
    HANDLE_ERROR(cudaHostAlloc((void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAll

    for (int i=0; i<FULL_DATA_SIZE; i++) {
        host_a[i] = rand();
        host_b[i] = rand();
    }

    HANDLE_ERROR(cudaEventRecord(start, 0));

    //now loop over full data, in bite-sized chunks
    for (int i=0; i<FULL_DATA_SIZE; i+= NUM*2) {
        //enqueue copies of a in stream0 and stream1
        //将锁定内存以异步方式复制到设备上
        HANDLE_ERROR(cudaMemcpyAsync(dev_a0, host_a+i, NUM * sizeof(int), cudaMemcpyHostT
        HANDLE_ERROR(cudaMemcpyAsync(dev_a1, host_a+i+NUM, NUM * sizeof(int), cudaMemcpyH
        //enqueue copies of b in stream0 and stream1
        HANDLE_ERROR(cudaMemcpyAsync(dev_b0, host_b+i, NUM * sizeof(int), cudaMemcpyHostT
        HANDLE_ERROR(cudaMemcpyAsync(dev_b1, host_b+i+NUM, NUM * sizeof(int), cudaMemcpyH

        //enqueue kernels in stream0 and stream1
        singlestream_kernel<<<NUM/256, 256, 0, stream0>>>(dev_a0, dev_b0, dev_c0);
        singlestream_kernel<<<NUM/256, 256, 0, stream1>>>(dev_a1, dev_b1, dev_c1);

        //enqueue copies of c from device to locked memory
        HANDLE_ERROR(cudaMemcpyAsync(host_c+i, dev_c0, NUM * sizeof(int), cudaMemcpyDevic
        HANDLE_ERROR(cudaMemcpyAsync(host_c+i+NUM, dev_c1, NUM * sizeof(int), cudaMemcpyD
    }

    float elapsedTime;

    HANDLE_ERROR(cudaStreamSynchronize(stream0));
    HANDLE_ERROR(cudaStreamSynchronize(stream1));

    HANDLE_ERROR(cudaEventRecord(stop, 0));

    HANDLE_ERROR(cudaEventSynchronize(stop));
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime,start, stop));
    printf( "Time taken:  %3.1f ms\n", elapsedTime );

    //cleanup the streams and memory
    HANDLE_ERROR(cudaFreeHost(host_a));
    HANDLE_ERROR(cudaFreeHost(host_b));
    HANDLE_ERROR(cudaFreeHost(host_c));
    HANDLE_ERROR(cudaFree(dev_a0));
    HANDLE_ERROR(cudaFree(dev_b0));
    HANDLE_ERROR(cudaFree(dev_c0));
    HANDLE_ERROR(cudaFree(dev_a1));
    HANDLE_ERROR(cudaFree(dev_b1));
    HANDLE_ERROR(cudaFree(dev_c1));
    HANDLE_ERROR(cudaStreamDestroy(stream0));
    HANDLE_ERROR(cudaStreamDestroy(stream1));

    return 0;
}

float malloc_test(int size)
{
    cudaEvent_t start, stop;
    float *a, *b, c, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;
    float elapsedTime;

    HANDLE_ERROR(cudaEventCreate(&start));
    HANDLE_ERROR(cudaEventCreate(&stop));

    //allocate memory on the CPU side
    a = (float*)malloc(size * sizeof(float));
    b = (float*)malloc(size * sizeof(float));
    partial_c = (float*)malloc(blocksPerGrid * sizeof(float));

    //allocate the memory on the GPU
    HANDLE_ERROR(cudaMalloc((void**)&dev_a, size * sizeof(float)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_b, size * sizeof(float)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_partial_c, blocksPerGrid * sizeof(float)));

    //fill in the host memory with data
    for (int i=0; i<size; i++) {
        a[i] = i;
```

关闭

```
        b[i] = i * 2;
    }

    HANDLE_ERROR(cudaEventRecord(start, 0));
    //copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR(cudaMemcpy(dev_a, a, size * sizeof(float), cudaMemcpyHostToDevice));
    HANDLE_ERROR(cudaMemcpy(dev_b, b, size * sizeof(float), cudaMemcpyHostToDevice));

    dot_kernel<<<blocksPerGrid, threadsPerBlock>>>(size, dev_a, dev_b, dev_partial_c);
    //copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR(cudaMemcpy(partial_c, dev_partial_c,blocksPerGrid*sizeof(float), cudaMem

    HANDLE_ERROR(cudaEventRecord(stop, 0));
    HANDLE_ERROR(cudaEventSynchronize(stop));
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime,start, stop));

    //finish up on the CPU side
    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i];
    }

    HANDLE_ERROR(cudaFree(dev_a));
    HANDLE_ERROR(cudaFree(dev_b));
    HANDLE_ERROR(cudaFree(dev_partial_c));

    //free memory on the CPU side
    free(a);
    free(b);
    free(partial_c);

    //free events
    HANDLE_ERROR(cudaEventDestroy(start));
    HANDLE_ERROR(cudaEventDestroy(stop));

    printf("Value calculated:  %f\n", c);

    return elapsedTime;
}

float cuda_host_alloc_test(int size)
{
    cudaEvent_t start, stop;
    float *a, *b, c, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;
    float elapsedTime;

    HANDLE_ERROR(cudaEventCreate(&start));
    HANDLE_ERROR(cudaEventCreate(&stop));

    //allocate the memory on the CPU
    HANDLE_ERROR(cudaHostAlloc((void**)&a, size*sizeof(float), cudaHostAllocWriteCombined
    HANDLE_ERROR(cudaHostAlloc((void**)&b, size*sizeof(float), cudaHostAllocWriteCombined
    HANDLE_ERROR(cudaHostAlloc((void**)&partial_c, blocksPerGrid*sizeof(float), cudaHostA

    //find out the GPU pointers
    HANDLE_ERROR(cudaHostGetDevicePointer(&dev_a, a, 0));
    HANDLE_ERROR(cudaHostGetDevicePointer(&dev_b, b, 0));
    HANDLE_ERROR( cudaHostGetDevicePointer(&dev_partial_c, partial_c, 0));

    //fill in the host memory with data
    for (int i=0; i<size; i++) {
        a[i] = i;
        b[i] = i*2;
    }

    HANDLE_ERROR(cudaEventRecord(start, 0));

    dot_kernel<<<blocksPerGrid, threadsPerBlock>>>(size, dev_a, dev_b, dev_partial_c);

    HANDLE_ERROR(cudaThreadSynchronize());
    HANDLE_ERROR(cudaEventRecord(stop, 0));
    HANDLE_ERROR(cudaEventSynchronize(stop));
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime,start, stop));

    //finish up on the CPU side
    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i];
```

关闭

```
    }

    HANDLE_ERROR(cudaFreeHost(a));
    HANDLE_ERROR(cudaFreeHost(b));
    HANDLE_ERROR(cudaFreeHost(partial_c));

    // free events
    HANDLE_ERROR(cudaEventDestroy(start));
    HANDLE_ERROR(cudaEventDestroy(stop));

    printf("Value calculated:  %f\n", c);

    return elapsedTime;
}

int test20()
{
    cudaDeviceProp prop;
    int whichDevice;
    HANDLE_ERROR(cudaGetDevice(&whichDevice));
    HANDLE_ERROR(cudaGetDeviceProperties(&prop, whichDevice));
    if (prop.canMapHostMemory != 1) {
        printf( "Device can not map memory.\n" );
        return 0;
    }

    HANDLE_ERROR(cudaSetDeviceFlags(cudaDeviceMapHost));

    //try it with malloc
    float elapsedTime = malloc_test(NUM);
    printf("Time using cudaMalloc:  %3.1f ms\n", elapsedTime);

    //now try it with cudaHostAlloc
    elapsedTime = cuda_host_alloc_test(NUM);
    printf("Time using cudaHostAlloc:  %3.1f ms\n", elapsedTime);

    return 0;
}

void* routine(void *pvoidData)
{
    DataStruct *data = (DataStruct*)pvoidData;
    HANDLE_ERROR(cudaSetDevice(data->deviceID));

    int size = data->size;
    float *a, *b, c, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;

    //allocate memory on the CPU side
    a = data->a;
    b = data->b;
    partial_c = (float*)malloc(blocksPerGrid * sizeof(float));

    //allocate the memory on the GPU
    HANDLE_ERROR(cudaMalloc((void**)&dev_a, size * sizeof(float)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_b, size * sizeof(float)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_partial_c, blocksPerGrid*sizeof(float)));

    //copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR(cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice));
    HANDLE_ERROR(cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice));

    dot_kernel<<<blocksPerGrid, threadsPerBlock>>>(size, dev_a, dev_b, dev_partial_c);
    //copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR(cudaMemcpy( partial_c, dev_partial_c, blocksPerGrid * sizeof(float), cuda

    //finish up on the CPU side
    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i];
    }

    HANDLE_ERROR(cudaFree(dev_a));
    HANDLE_ERROR(cudaFree(dev_b));
    HANDLE_ERROR(cudaFree(dev_partial_c));

    //free memory on the CPU side
    free(partial_c);

```

关闭

```cpp
    data->returnValue = c;
    return 0;
}

int test21()
{
    int deviceCount;
    HANDLE_ERROR(cudaGetDeviceCount(&deviceCount));
    if (deviceCount < 2) {
        printf("We need at least two compute 1.0 or greater devices, but only found %d\n",
        return 0;
    }

    float *a = (float*)malloc(sizeof(float) * NUM);
    HANDLE_NULL(a);
    float *b = (float*)malloc(sizeof(float) * NUM);
    HANDLE_NULL(b);

    //fill in the host memory with data
    for (int i=0; i<NUM; i++) {
        a[i] = i;
        b[i] = i*2;
    }

    //prepare for multithread
    DataStruct  data[2];
    data[0].deviceID = 0;
    data[0].size = NUM/2;
    data[0].a = a;
    data[0].b = b;

    data[1].deviceID = 1;
    data[1].size = NUM/2;
    data[1].a = a + NUM/2;
    data[1].b = b + NUM/2;

    CUTThread thread = start_thread(routine, &(data[0]));
    routine(&(data[1]));
    end_thread(thread);

    //free memory on the CPU side
    free(a);
    free(b);

    printf("Value calculated:  %f\n", data[0].returnValue + data[1].returnValue);

    return 0;
}
```

funset.cuh:

[cpp]

```cpp
#ifndef _FUNSET_CUH_
#define _FUNSET_CUH_

#include <stdio.h>
#include "cpu_anim.h"

#define NUM  33 * 1024 * 1024//1024*1024//33 * 1024//10
#define DIM 1024//1000
#define PI 3.1415926535897932f
#define imin(a, b) (a < b ? a : b)
const int threadsPerBlock = 256;
const int blocksPerGrid = imin(32, (NUM/2+threadsPerBlock-1) / threadsPerBlock);
//imin(32, (NUM + threadsPerBlock - 1) / threadsPerBlock);
#define rnd(x) (x * rand() / RAND_MAX)
#define INF 2e10f
#define SPHERES 20
#define MAX_TEMP 1.0f
#define MIN_TEMP 0.0001f
#define SPEED 0.25f
#define SIZE (100*1024*1024)
#define FULL_DATA_SIZE (NUM*20)

//__global__关键字将告诉编译器，函数应该编译为在设备而不是主机上运行
__global__ void add(int a, int b, int* c);
```

关闭

```
__global__ void add_blockIdx(int* a, int* b, int* c);
__global__ void add_threadIdx(int* a, int* b, int* c);
__global__ void add_blockIdx_threadIdx(int* a, int* b, int* c);

struct cuComplex {
    float r, i;

    __device__ cuComplex(float a, float b) : r(a), i(b)  {}

    __device__ float magnitude2(void)
    {
        return r * r + i * i;
    }

    __device__ cuComplex operator*(const cuComplex& a)
    {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }

    __device__ cuComplex operator+(const cuComplex& a)
    {
        return cuComplex(r+a.r, i+a.i);
    }
};

__device__ int julia(int x, int y);
__global__ void kernel_julia(unsigned char *ptr);
__global__ void ripple_kernel(unsigned char *ptr, int ticks);
struct Sphere;

struct DataBlock {
    unsigned char *dev_bitmap;
    CPUAnimBitmap *bitmap;
    Sphere *s;
};

void generate_frame(DataBlock *d, int ticks);
void cleanup(DataBlock *d);

__global__ void dot_kernel(float *a, float *b, float *c);
__global__ void julia_kernel(unsigned char *ptr);

//通过一个数据结构对球面建模
struct Sphere {
    float r,b,g;
    float radius;
    float x,y,z;
    __device__ float hit(float ox, float oy, float *n)
    {
        float dx = ox - x;
        float dy = oy - y;
        if (dx*dx + dy*dy < radius*radius) {
            float dz = sqrtf(radius*radius - dx*dx - dy*dy);
            *n = dz / sqrtf(radius * radius);
            return dz + z;
        }
        return -INF;
    }
};

//声明为常量内存,__constant__将把变量的访问限制为只读
__constant__ Sphere s[SPHERES];

__global__ void RayTracing_kernel(Sphere *s, unsigned char *ptr);
__global__ void RayTracing_kernel(unsigned char *ptr);

//these exist on the GPU side
texture<float> texConstSrc, texIn, texOut;
texture<float, 2> texConstSrc2, texIn2, texOut2;

//this kernel takes in a 2-d array of floats it updates the value-of-interest by a
//scaled value based on itself and its nearest neighbors
__global__ void Heat_blend_kernel(float *dst, bool dstOut);
__global__ void blend_kernel(float *dst, bool dstOut);
__global__ void Heat_copy_const_kernel(float *iptr);
__global__ void copy_const_kernel(float *iptr);

struct Heat_DataBlock {
    unsigned char   *output_bitmap;
```

关闭

```cpp
    float           *dev_inSrc;
    float           *dev_outSrc;
    float           *dev_constSrc;
    CPUAnimBitmap  *bitmap;

    cudaEvent_t     start, stop;
    float           totalTime;
    float           frames;
};

//globals needed by the update routine
struct DataBlock_opengl {
    float           *dev_inSrc;
    float           *dev_outSrc;
    float           *dev_constSrc;

    cudaEvent_t     start, stop;
    float           totalTime;
    float           frames;
};

void Heat_anim_gpu(Heat_DataBlock *d, int ticks);
void anim_gpu(Heat_DataBlock *d, int ticks);
//clean up memory allocated on the GPU
void Heat_anim_exit(Heat_DataBlock *d);
void anim_exit(Heat_DataBlock *d);
void generate_frame_opengl(uchar4 *pixels, void*, int ticks);
__global__ void ripple_kernel_opengl(uchar4 *ptr, int ticks);
__global__ void Heat_blend_kernel_opengl(float *dst, bool dstOut);
__global__ void Heat_copy_const_kernel_opengl(float *iptr);
void anim_gpu_opengl(uchar4* outputBitmap, DataBlock_opengl *d, int ticks);
void anim_exit_opengl(DataBlock_opengl *d);
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo);
__global__ void singlestream_kernel(int *a, int *b, int *c);
__global__ void dot_kernel(int size, float *a, float *b, float *c);

struct DataStruct{
    int     deviceID;
    int     size;
    float   *a;
    float   *b;
    float   returnValue;
};

#endif //_FUNSET_CUH_
```

funset.cu:

[cpp]

```cpp
#include "funset.cuh"
#include <stdio.h>

__global__ void add(int a, int b, int* c)
{
    *c = a + b;
}

//__global__：从主机上调用并在设备上运行
__global__ void add_blockIdx(int* a, int* b, int* c)
{
    //计算该索引处的数据
    //变量blockIdx，是一个内置变量，在CUDA运行时中已经预先定义了这个变量
    //此变量中包含的值就是当前执行设备代码的线程块的索引
    int tid = blockIdx.x;//this thread handles the data at its thread id
    if (tid < NUM)
        c[tid] = a[tid] + b[tid];
}

//__device__：表示代码将在GPU而不是主机上运行，
//由于此函数已声明为__device__函数，因此只能从其它__device__函数或者
//从__global__函数中调用它们
__device__ int julia(int x, int y)
{
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);
```

关闭

```

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;

        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}

__global__ void kernel_julia(unsigned char *ptr)
{
    //map from blockIdx to pixel position
    int x = blockIdx.x;
    int y = blockIdx.y;
    //gridDim为内置变量，对所有的线程块来说，gridDim是一个常数，用来保存线程格每一维的大小
    //此处gridDim的值是(DIM, DIM)
    int offset = x + y * gridDim.x;

    //now calculate the value at that position
    int juliaValue = julia(x, y);

    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}

__global__ void add_threadIdx(int* a, int* b, int* c)
{
    //使用线程索引来对数据进行索引而非通过线程块索引(blockIdx.x)
    int tid = threadIdx.x;

    if (tid < NUM)
        c[tid] = a[tid] + b[tid];
}

__global__ void add_blockIdx_threadIdx(int* a, int* b, int* c)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid == 0) {
        printf("blockDim.x = %d, gridDim.x = %d\n", blockDim.x, gridDim.x);
    }

    while (tid < NUM) {
        c[tid] = a[tid] + b[tid];

        tid += blockDim.x * gridDim.x;
    }
}

__global__ void ripple_kernel(unsigned char *ptr, int ticks)
{
    // map from threadIdx/BlockIdx to pixel position
    //将线程和线程块的索引映射到图像坐标
    //对x和y的值进行线性化从而得到输出缓冲区中的一个偏移
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    // now calculate the value at that position
    //生成一个随时间变化的正弦曲线"波纹"
    float fx = x - DIM/2;
    float fy = y - DIM/2;
    float d = sqrtf(fx * fx + fy * fy);
    unsigned char grey = (unsigned char)(128.0f + 127.0f * cos(d/10.0f - ticks/7.0f) / (d/

    ptr[offset*4 + 0] = grey;
    ptr[offset*4 + 1] = grey;
    ptr[offset*4 + 2] = grey;
    ptr[offset*4 + 3] = 255;
}

```

关闭

```
__global__ void dot_kernel(float *a, float *b, float *c)
{
    //声明了一个共享内存缓冲区，它将保存每个线程计算的加和值
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float temp = 0;
    while (tid < NUM) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    //set the cache values
    cache[cacheIndex] = temp;

    //synchronize threads in this block
    //对线程块中的线程进行同步
    //这个函数将确保线程块中的每个线程都执行完__syncthreads()前面的语句后，才会执行下一条语句
    __syncthreads();

    //for reductions(归
约), threadsPerBlock must be a power of 2 because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        //在循环迭代中更新了共享内存变量cache，并且在循环的下一次迭代开始之前，
        //需要确保当前迭代中所有线程的更新操作都已经完成
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}

__global__ void julia_kernel(unsigned char *ptr)
{
    //map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    __shared__ float shared[16][16];

    //now calculate the value at that position
    const float period = 128.0f;

    shared[threadIdx.x]
[threadIdx.y] = 255 * (sinf(x*2.0f*PI/ period) + 1.0f) *(sinf(y*2.0f*PI/ period) + 1.0f) ,

    //removing this syncthreads shows graphically what happens
    //when it doesn't exist.this is an example of why we need it.
    __syncthreads();

    ptr[offset*4 + 0] = 0;
    ptr[offset*4 + 1] = shared[15 - threadIdx.x][15 - threadIdx.y];
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}

__global__ void RayTracing_kernel(Sphere *s, unsigned char *ptr)
{
    //map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    float ox = (x - DIM/2);
    float oy = (y - DIM/2);

    float r=0, g=0, b=0;
    float maxz = -INF;

    for (int i = 0; i < SPHERES; i++) {
        float n;
        float t = s[i].hit(ox, oy, &n);
        if (t > maxz) {
            float fscale = n;
```

关闭

```
            r = s[i].r * fscale;
            g = s[i].g * fscale;
            b = s[i].b * fscale;
            maxz = t;
        }
    }

    ptr[offset*4 + 0] = (int)(r * 255);
    ptr[offset*4 + 1] = (int)(g * 255);
    ptr[offset*4 + 2] = (int)(b * 255);
    ptr[offset*4 + 3] = 255;
}

__global__ void RayTracing_kernel(unsigned char *ptr)
{
    //map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    float ox = (x - DIM/2);
    float oy = (y - DIM/2);

    float r=0, g=0, b=0;
    float maxz = -INF;

    for(int i = 0; i < SPHERES; i++) {
        float n;
        float t = s[i].hit(ox, oy, &n);
        if (t > maxz) {
            float fscale = n;
            r = s[i].r * fscale;
            g = s[i].g * fscale;
            b = s[i].b * fscale;
            maxz = t;
        }
    }

    ptr[offset*4 + 0] = (int)(r * 255);
    ptr[offset*4 + 1] = (int)(g * 255);
    ptr[offset*4 + 2] = (int)(b * 255);
    ptr[offset*4 + 3] = 255;
}

__global__ void Heat_blend_kernel(float *dst, bool dstOut)
{
    //map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    int left = offset - 1;
    int right = offset + 1;
    if (x == 0) left++;
    if (x == DIM-1) right--;

    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0) top += DIM;
    if (y == DIM-1) bottom -= DIM;

    float t, l, c, r, b;

    if (dstOut) {
        //tex1Dfetch是编译器内置函数，从设备内存取纹理
        t = tex1Dfetch(texIn, top);
        l = tex1Dfetch(texIn, left);
        c = tex1Dfetch(texIn, offset);
        r = tex1Dfetch(texIn, right);
        b = tex1Dfetch(texIn, bottom);

    } else {
        t = tex1Dfetch(texOut, top);
        l = tex1Dfetch(texOut, left);
        c = tex1Dfetch(texOut, offset);
        r = tex1Dfetch(texOut, right);
        b = tex1Dfetch(texOut, bottom);
    }

    dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
```

关闭

```
}

__global__ void blend_kernel(float *dst, bool dstOut)
{
    //map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    float t, l, c, r, b;
    if (dstOut) {
        t = tex2D(texIn2, x, y-1);
        l = tex2D(texIn2, x-1, y);
        c = tex2D(texIn2, x, y);
        r = tex2D(texIn2, x+1, y);
        b = tex2D(texIn2, x, y+1);
    } else {
        t = tex2D(texOut2, x, y-1);
        l = tex2D(texOut2, x-1, y);
        c = tex2D(texOut2, x, y);
        r = tex2D(texOut2, x+1, y);
        b = tex2D(texOut2, x, y+1);
    }
    dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
}

__global__ void Heat_copy_const_kernel(float *iptr)
{
    //map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    float c = tex1Dfetch(texConstSrc, offset);
    if (c != 0)
        iptr[offset] = c;
}

__global__ void copy_const_kernel(float *iptr)
{
    //map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    float c = tex2D(texConstSrc2, x, y);
    if (c != 0)
        iptr[offset] = c;
}

void generate_frame_opengl(uchar4 *pixels, void*, int ticks)
{
    dim3 grids(DIM / 16, DIM / 16);
    dim3 threads(16, 16);
    ripple_kernel_opengl<<<grids, threads>>>(pixels, ticks);
}

__global__ void ripple_kernel_opengl(uchar4 *ptr, int ticks)
{
    //map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    // now calculate the value at that position
    float fx = x - DIM / 2;
    float fy = y - DIM / 2;
    float d = sqrtf(fx * fx + fy * fy);
    unsigned char grey = (unsigned char)(128.0f + 127.0f * cos(d/10.0f - ticks/7.0f) / (d
    ptr[offset].x = grey;
    ptr[offset].y = grey;
    ptr[offset].z = grey;
    ptr[offset].w = 255;
}

__global__ void Heat_blend_kernel_opengl(float *dst, bool dstOut)
{
    //map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
```

关闭

```
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    int left = offset - 1;
    int right = offset + 1;
    if (x == 0) left++;
    if (x == DIM-1) right--;

    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0) top += DIM;
    if (y == DIM-1) bottom -= DIM;

    float t, l, c, r, b;
    if (dstOut) {
        t = tex1Dfetch(texIn, top);
        l = tex1Dfetch(texIn, left);
        c = tex1Dfetch(texIn, offset);
        r = tex1Dfetch(texIn, right);
        b = tex1Dfetch(texIn, bottom);

    } else {
        t = tex1Dfetch(texOut, top);
        l = tex1Dfetch(texOut, left);
        c = tex1Dfetch(texOut, offset);
        r = tex1Dfetch(texOut, right);
        b = tex1Dfetch(texOut, bottom);
    }
    dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
}

__global__ void Heat_copy_const_kernel_opengl(float *iptr)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    float c = tex1Dfetch(texConstSrc, offset);
    if (c != 0)
        iptr[offset] = c;
}

__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
    //clear out the accumulation buffer called temp since we are launched with 256 threads
    //it is easy to clear that memory with one write per thread
    __shared__  unsigned int temp[256]; //共享内存缓冲区
    temp[threadIdx.x] = 0;
    __syncthreads();

    //calculate the starting index and the offset to the next block that each thread will
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd(&temp[buffer[i]], 1);
        i += stride;
    }

    //sync the data from the above writes to shared memory then add the shared memory valu
    //the other thread blocks using global memory atomic adds same as before, since we hav
    //updating the global histogram is just one write per thread!
    __syncthreads();
    atomicAdd(&(histo[threadIdx.x]), temp[threadIdx.x]);
}

__global__ void singlestream_kernel(int *a, int *b, in
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < NUM) {
        int idx1 = (idx + 1) % 256;
        int idx2 = (idx + 2) % 256;
        float as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
        float bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
        c[idx] = (as + bs) / 2;
    }
}

__global__ void dot_kernel(int size, float *a, float *b, float *c)
{
```

关闭

```
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float temp = 0;
    while (tid < size) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    //set the cache values
    cache[cacheIndex] = temp;

    //synchronize threads in this block
    __syncthreads();

    //for reductions(归
约), threadsPerBlock must be a power of 2 because of the following code
    int i = blockDim.x / 2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```

以上来自于对《GPU高性能编程CUDA实战》书中内容整理。

GitHub：https://github.com/fengbingchun/CUDA_Test

顶　　踩
2　　0

上一篇　Git基础(常用命令)介绍

下一篇　Linux下常用的C/C++开源Socket库

我的同类文章

CUDA（10）

CUDA基础介绍	2017-01-23	阅读 0	windows7 64位机上配置支...	2016-12-27	阅读 111
windows7 64位机上安装配...	2016-12-27	阅读 506	Ubuntu14.04 64位机上安装...	2016-12-23	阅读 186
Ubuntu14.04 64位机上安装...	2016-12-23	阅读 433	CUDA Runtime API 汇总	2015-04-26	阅读 2495
windows7 64位机上CUDA7....	2015-04-09	阅读 17677	GPU及GPU通用计算编程模...	2014-02-21	阅读 4017
Windows7 32位机上，Open...	2013-08-08	阅读 17385	OpenCV中G...		

关闭

猜你在找

ArcGIS for javascript 项目实战（环境监测系统）　　《GPU高性能编程CUDA实战》

从零开始学习mac系统视频录制编辑软件Camtasia For 《GPU高性能编程CUDA实战》学习笔记十一

查看评论

1楼 luyyuang 2015-10-08 09:44发表

请问你用的CUDA SDK是那个版本的？？

Re: fengbingchun 2015-10-08 11:27发表

回复luyyuang：7.0

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker Op
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC
coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap

关闭