HHCV Programming Guide

孟朝晖

河海大学 **HHCV**

Computer Vision Group of Hohai University

第一章

AIcourse2016 安装配置指南

AIcourse2016 是一个基于 VS、CUDA、OpenCV 的处理视频的项目。

以下安装配置步骤针对 VS2010，CUDA4.2，OpenCV2.4，高于此版本的新版与此有差别，但步骤原理基本类似，要在理解各个步骤含义的基础上变通。

1.1 安装配置 VS2010

省略

1.2 安装配置 CUDA4.2

省略

1.3 安装配置 OpenCV2.4

1.3.1 运行 OpenCV-2.4.0.exe 解压缩到 C:\opencv240

1.3.2 设置环境变量

系统 Path 里添加(注销当前用户或者重启系统会生效)

设置环境变量对是 OpenCV 必须的：为什么？

凡是安装的软件大多不需要自己设置环境变量，由安装软件负责设置，但是 OpenCV 是解压的，不是安装的，不设置环境变量，并不影响编译和链接自己编写的 OpenCV 程序，但运行中需要的.dll 文件就不能自动找到，会报错。

设置方法：我的电脑->右键->属性->高级->环境变量->系统变量：
在 Path 变量上添加两个路径
(这个是 32 位的)：
C:\opencv240\opencv\build\x86\vc10\bin;C:\opencv240\opencv\build\common\tbb\ia32\vc10

(这个是 64 位的)：
C:\opencv240\opencv\build\x64\vc10\bin;C:\opencv240\opencv\build\common\tbb\intel64\vc10

设置完成后必须重启电脑。

附加注意事项:
OpenCV 本身并不包含视频编码解码模块，一般需要另外安装，比如 xvid（群文件中的\software\目录中有），如果电脑中已经安装了其它视频处理软件，也可能不需要 xvid。这个问题比较复杂，需要尝试。

1.4 打开 vs2010，新建一个 win32 控制台项目 AIcourse2016

1.4.1 新建一个项目

按照 VC2010 入门经典（Ivor Horton's Beginning Visual C++ 2012）的 Ex1_01 为例，包含 stdafx.h

File | New | Project

选择 Win32，Win32 Console Application

进入 Application Setting 选择 Empty Project，要包含 Precompiled Header，

这样会自动产生三个文件: stdafx.h, targetver.h, stdafx.cpp,

这三个文件的作用主要是定义许多预制类型等，OpenCV 也需要。

项目名称 AIcourse2016，路经 D:\OpenCV\（路径可以自己选择更换）

OK

完成后，项目中包含如下文件：

stdafx.h
targetver.h

AIcourse2016.cpp 这个是主程序入口
stdafx.cpp

建议：新建相对应的文件名，将对应的代码（文本格式）粘贴进来。

1.4.2 代码复制

将代码复制进新建项目 AIcourse2016，此包有 13 个文件，建议在新项目中先建好对应的文件名，再将文件内容对应 copy 进项目文件中，注意要干净的文本格式 copy。

推荐比较好用的文本编辑器：Notepad++（可以方便进行“文本块编辑”，按住 Alt 键，拖动鼠标选择文本

块）

1.5 关联配置 OpenCV

这一步骤的目的：项目 AIcourse2016 中需要调用 OpenCV 的函数，要配置相关的包含链接库。

1.5.1 配置 Include Directories

(Debug and Release)

In 2010
Project -> AIcourse2016 Properties...Configuration Properties -> VC++ Directories

Include Directories... add:

加入以下几个 include 文件的目录（我的电脑目录），看你的 OpenCV 安装在哪里了，

C:\opencv240\opencv\include\opencv
C:\opencv240\opencv\build\include
C:\opencv240\opencv\build\include\opencv
C:\opencv240\opencv\build\include\opencv2

注意这里有选项 Configuration : Debug and Release, Platform : Win32 and x64
include 文件都是一些头文件，不区分 32 和 64 位。
可以一起配置 Configuration : All Configurations, Platform : All platforms

在 VS2010 中，设定 Configuration 和 Platform 时要注意其含义及关联的库，有时可以一起设置，有时必须分开设置。

1.5.2 配置 Library Directories

Configuration(Debug and Release)可以一起配置 Configuration : All Configrations
Platform : Win32 and x64 要分开设置，因为 Library 是目标文件库的目录，32 和 64 是不同的。

Library Directories... add:

这个是 32 位的（我的电脑目录）

C:\opencv240\opencv\build\x86\vc10\lib
C:\opencv240\opencv\build\common\tbb\ia32\vc10

这个是 64 位的（我的电脑目录）

C:\opencv240\opencv\build\x64\vc10\lib
C:\opencv240\opencv\build\common\tbb\intel64\vc10

1.5.3 配置 Linker

实际上是 4 组文件，Debug and Release 文件名是不同的，Win32 and x64 文件名是相同的，但目录不同。

32 位的文件在 C:\opencv240\opencv\build\x86\vc10\lib
64 位的文件在 C:\opencv240\opencv\build\x64\vc10\lib

Linker -> Input -> Additional Dependencies...

For Debug Builds... add:

opencv_calib3d240d.lib
opencv_contrib240d.lib
opencv_core240d.lib
opencv_features2d240d.lib
opencv_flann240d.lib
opencv_gpu240d.lib
opencv_haartraining_engined.lib
opencv_highgui240d.lib
opencv_imgproc240d.lib
opencv_legacy240d.lib
opencv_ml240d.lib
opencv_nonfree240d.lib
opencv_objdetect240d.lib
opencv_photo240d.lib
opencv_stitching240d.lib
opencv_ts240d.lib
opencv_video240d.lib
opencv_videostab240d.lib

For Release Builds... add:

opencv_calib3d240.lib
opencv_contrib240.lib
opencv_core240.lib
opencv_features2d240.lib
opencv_flann240.lib
opencv_gpu240.lib
opencv_haartraining_engine.lib
opencv_highgui240.lib

opencv_imgproc240.lib
opencv_legacy240.lib
opencv_ml240.lib
opencv_nonfree240.lib
opencv_objdetect240.lib
opencv_photo240.lib
opencv_stitching240.lib
opencv_ts240.lib
opencv_video240.lib
opencv_videostab240.lib

1.6 配置 CUDA 编译环境

CUDA 是适用于 NVIDIA 公司各类显卡、计算卡（N 卡）的开发平台，基于 CUDA 开发的项目称为 GPU 项目。

GPU 项目的运作逻辑是，一部分程序（串行程序）在主机端（host 端：主板上的 CPU、内存等）部署和运行，另一部分程序（并行程序）在显卡端（device 端：显卡或计算卡上的 GPU 芯片和显卡内存等）部署和运行，注意：可以同时运行，host 程序调用 device 程序后可以不必等待 device 程序返回而继续。

实际编程中主要通过两种方式完成 host 端和 device 端的交互：
（1）子程序调用：host 端可以通过一个特殊代码格式调用 device 端的代码，相当于调用子程序；
（2）数据交换：host 端可以通过调用 cuda 函数完成 host 端与 device 端之间的数据交换；

CUDA 有自己的 c/c++编译器 nvcc，可以独立生成目标代码，并且部署在显卡上。在 VS 平台环境下，通过关联设置，可以将 host 端代码和数据与 device 端代码和数据结合在一起构成一个运行项目。

NVIDIA 公司推荐的编程指南上，配置 cuda 是一个比较复杂的过程，主要是要带有其 cuda-SDK，实际上只侧重于计算的项目并不需要其复杂的 SDK 包。下面推荐的方法是比较简单的配置方案。

《源代码文件独立编译配置方案》：不使用 SDK 中 D:/cudasdk42 的函数，具体步骤如下。

1.6.1 建立一个空的项目并建立各个源代码文件

按照建立控制台 C++ 项目的方式建立一个空的项目 AIcourse2016，并建立各个源代码文件，前面已经完成。

1.6.2 设置编译环境

在项目 AIcourse2016 右键打开，选择 Build Customizations，如果正确安装了 CUDA，就会看到，CUDA4.2(.targets, .props)，选定，这个就是以前版本的编译规则，.rule，有了这个，编译 CUDA 就正常了。

1.6.3 增加链接依赖的库文件

在项目 AIcourse2016 右键打开，选择 Properties，先选择 Configuration(Debug, Release) 和 Platform(Win32, x64) 四种组合分别设定，这一步是必须的。也可以：Configuration(All) 和 Platform(All) 一起设定。

Configuration Properties -> Linker -> Input -> Additional Dependencies

增加 cudart.lib（这是关键文件：cuda run time）

变成：
cudart.lib
kernel32.lib
user32.lib
gdi32.lib
winspool.lib
comdlg32.lib
advapi32.lib
shell32.lib
ole32.lib
oleaut32.lib
uuid.lib
odbc32.lib
odbccp32.lib

1.6.4 设定单个源代码文件的属性

先介绍几个基本概念：

（1）分类：主要有三种类型的程序文件

(1) AIcourse2016.cpp 等，是正常的 c++ 程序，不含 cuda 代码块，无须特别设定属性，可以查看一下，
(2) gpuCall.cu,gpuSet.cu 是包含 cuda 代码块的程序，但不包含核函数，
(3) gpuKernel.cu 是只含有核函数的程序文件。

（2）源代码文件：是代码存放的基本盒子，也是编译链接的基本单位，通常具有同类功能特征的代码块会放在一个源代码文件中。本项目的源代码文件有如下几种：

> func_list.h　hhcv 函数的头文件，
> hhcvDsCore.h　hhcv 数据结构的头文件，
> hhcvExtern.h　全局变量说明文件，

> stdafx.h　targetver.h　stdafx.cpp 三个文件是由 VS App Wizard 生成的预编译文件。

> AIcourse2016.cpp　主程序文件，
> readFile.cpp　读外部数据的各种函数的文件，
> hostMultiLayer1.cpp　视频处理的主要部分，在 host 端运行的程序，
> hostMultiLayer2.cpp　视频处理的主要部分，实现与 hostMultiLayer1 一样的功能，在 device 端运行的程序，

> gpuSet.cu　在 host 端对 device 端进行必要的配置，包括给显卡进行编号、重置等（可以有多块显卡），在显卡内存进行数据块的分配等。

> gpuCall.cu　从 host 端对 device 端的函数进行调用的函数在这个文件中存放。

注释：gpuSet.cu 和 gpuCall.cu，是一种过桥代码包，目的是将与 gpu 有关的函数单独打包为独立的代码文件。这是个 2012 年最初对 opencv 和 gpu 组合编程时遇到的遗留问题，目的是将 host 端对 gpu 代码的配置和调用与 host 端对 opencv 代码的调用分开存放在不同的文件，这样就不会编译出错。不过新版的组合好像解决了这个问题。

> gpuKernel.cu　这里只存放将要在显卡上并行运行 gpu 代码，称为核函数。

（3）两种编译器 cl 和 nvcc：cl 是 vc2010 提供的编译器，负责一般 c++（opencv）代码的编译，nvcc 是 CUDA 提供的编译器，负责 GPU 核函数代码的编译。

（4）需要特别设置属性的源代码文件

先介绍两个概念：cuda 函数与纯 gpu 核函数，cuda 函数是与 gpu 配置、分配内存、程序同步等有关的函数（cuda 参考手册上有很多这种函数），在 host 端执行，需要由 cl 编译器对其进行编译。纯 gpu 核函数是在显卡 device 端部署运行的并行代码，cl 编译器不对其编译，由 nvcc 负责编译。

上述各个程序文件并不都需要特别设置，只有与 gpu 有关的源代码文件需要设置，gpuSet.cu、gpuCall.cu、gpuKernel.cu，而且设置方案有所不同。

操作方法：
(a) AIcourse2016.cpp 等，无须配置。

(b) gpuCall.cu, gpuSet.cu 是包含 cuda 函数代码块的程序，但不包含核函数，仍然由 VS 的 cl 编译器对其进行编译。

在文件 gpuCall.cu, gpuSet.cu 分别右键打开，选择 Properties, 打开对话框，选择
Configuration Properties -> General
Excludes From Build : No cl 编译器不排除对其编译
Item Type : CUDA C/C++ 这是一个包含 CUDA 的程序
注意，四种组合分别设定，也可以一起设置。

(c) gpuKernel.cu 这是一个纯 gpu 核函数，cl 编译器不对其编译，由 nvcc 负责编译。

在文件 gpuKernel.cu 右键打开，选择 Properties, 打开对话框，选择
Configuration Properties -> General
Excludes From Build : Yes cl 编译器不对其编译，由 nvcc 负责编译
Item Type : CUDA C/C++ 这是一个包含 CUDA 的程序

完成前面几个步骤，就能编译链接了。

1.6.5、设置针对 GPU 型号的编译条件：（看你的显卡的计算能力）

在项目 AIcourse2016 右键打开，选择 Properties,
先选择 Configuration(Debug, Release) 和 Platform(Win32, x64) 四种组合分别设定，必须的。
也可以：Configuration(All) 和 Platform(All) 一起设定。

Configuration Properties -> CUDA C/C++ -> Device
Code Generation

compute_10, sm_10
改为
compute_13, sm_13

我的 Tesla C1060 支持 1.3，支持双精度计算。

1.7 运行调试

文件 D:\VideoData2016\HHCV_run_para.txt

是程序的运行配置文件，必须放在目录 D:\VideoData2016\ 中。

HHCV_run_para.txt 是以行为单位，按位置读取数据的文本文件，修改参数时注意数据所在的列位置。

第二章

AIcourse2016 中 C 数据结构与函数

2.1 C 数据结构

char * 字符串指针，用于表示字符串变量，
例子：
```
char * fname_para_readin = "D:\\VideoData2016\\HHCV_run_para.txt";
```
定义了一个字符串指针并赋值，一个文件名。

char 字符或字符数组，可以表示定长的字符串，
例子：
```
char fname_writerecord[100];
```
定义了一个长度 100 的字符串数组。

FILE * 文件指针，用于表示文件变量，
例子：
```
FILE * fptr_writerecord;
```
定义了一个文件指针变量。

2.2 C 函数

fopen，标准 c 函数，其原始定义为：`FILE* fopen(const char* path, const char* mode);`
作用为打开path字符串指定的磁盘文件，并返回其对应的文件指针，其包含头文件为stdio.h。但是在VS中其兼容定义比较繁琐，并且基于面向对象的设计原则，不鼓励使用这种直接操纵外部I/O的函数。比如，JAVA中就不支持这种操作方式。
例子：
```
fptr_writerecord = fopen(fname_writerecord, "w+");
```
其中的mode可选有："r"、"rb"、"w"、"w+"、"wb"、"a"、"at"等等。

fclose，标准 c 函数，其原始定义为：`int fclose (FILE* stream);`
作用为关闭 stream 文件指针指定的流文件，其包含头文件为 stdio.h。
例子：
```
fclose(fptr_writerecord);
```

fgets，标准 c 函数，其原始定义为：`char * fgets (char * str, int num, FILE* stream);`
函数在参数 stream 指定的文件中最多读取 num-1 个字符，存放在 str 指向的数组中，遇到文件结束符或回车换行符则提前结束，并载 str 的相应位置加上字符结束符'\0'，其包含头文件为 stdio.h。
例子：
```
while (fgets(one_line_strs, sizeof(one_line_strs), fptr_readin))
```

feof，标准 c 函数，其原始定义为：int feof (FILE* stream);
函数检查 stream 的当前操作位置，判断是否到文件末尾，是则返回一个非 0 值，不是末尾则返回 0 值，其包含头文件为 stdio.h。

fgetc，标准 c 函数，其原始定义为：int fgetc (FILE* stream);
函数从 stream 的当前操作位置读入一个字符，读入的为 unsigned char 类型，强制转换为 int 类型作为返回值，其包含头文件为 stdio.h。

memcpy，标准c函数，按字节数，以字节为单位，复制内存块数据，其原始定义为：
void * memcpy(void * dest, const void * src, size_t n);
dest表示目的内存区，src表示源内存区，n表示复制的字节数，返回指向dest的指针。
这是一个功能强大、速度快、效率高的函数，并且很容易出错，而且出错后很难找到原因，关键是要算准了将要复制的字节数。
例子：
memcpy(Src_hostImageData, Src_iplFrame->imageData, Src_size_imageData);
将图片帧 Src_iplFrame 的数据块指针 imageData 指向的像素块数据，复制到 Src_hostImageData 指向的数据块，字节数总数为 Src_size_imageData。

malloc，标准c函数，按字节数，以字节为单位，分配内存块，返回内存块首地址，分配失败则返回NULL，其原始定义为：
void * malloc(size_t size);
该函数可以直接按字节数分配内存块并返回内存块首地址（指针），而无须指定返回指针的类型，void* 的含义是指不限定返回指针的类型，但只要成功分配了，这个首地址（指针）是一定有的。实际使用当中，通常需要指定分配的内存块用来存放数据的类型，这时可以使用 cast 方式：
例子：
```
// 计算需要分配的总的字节数
int Src_size_imageData = sizeof(char) * Src_height * Src_width * Src_channels;

// 分配内存块，并指定返回指针所关联的数据类型
char * Src_hostImageData = (char *) malloc(Src_size_imageData);
```

例子：
```
// 计算需要分配的总的字节数
int Src_mem_size_pixelFrame = sizeof(HhcvPixel) * Src_height * Src_width;

// 分配内存块，并指定返回指针所关联的数据类型
HhcvPixel * Src_hostPixelFrame = (HhcvPixel *) malloc(Src_mem_size_pixelFrame);
```

free，标准c函数，释放由malloc分配的内存块，其原始定义为：
```
void * free(void* ptr);    // ptr必须是由malloc函数返回的首地址（指针）
```
例子：
```
free(Src_hostImageData);
free(Src_hostPixelFrame);
```

这个函数的使用方法很简单，其功能也很容易理解，但是初学 c 编程时，经常会忘记使用这个函数，原则上，由 malloc 函数分配的内存块，在完成了其功能后，都应该由 free 函数释放内存。

注记：内存的分配、使用、回收由编程者自己全面负责，这是 c 语言区别于其它语言的重要核心部分，精心设计安排内存的管理，可以避免相当大部分的 bug。

第三章

AIcourse2016 中 OpenCV 数据结构与函数

3.1 OpenCV 数据结构

CvCapture* 这个结构包含了所有要读入的视频流的信息，包括状态信息，比如下一帧的位置。
其具体结构没有公开，在highgui_c.h中标记为 "black box" capture structure
例子：

```
CvCapture* capture_camera;   // 表示摄像头采集的视频流
CvCapture* capture_file;     // 表示从文件读入的视频流
```

CvVideoWriter *，与CvCapture* 相对应，用于控制向外部文件中写入视频流，其具体结构没有公开，在highgui_c.h中标记为 "black box" video file writer structure
例子：

```
CvVideoWriter * writer = 0;
writer = cvCreateVideoWriter(fname_video_writeout,
                 CV_FOURCC('X','V','I','D'),
                 fps,cvSize(frameW, frameH),  isColor);
```

IplImage*，是OpenCV的关键数据结构，用来表达一幅图片或视频中的一帧，其定义如下：

```
typedef struct _IplImage
{
    int  nSize;           //sizeof(IplImage)
    int  ID;              //version (=0)
    int  nChannels;       // Most of OpenCV functions support 1,2,3 or 4 channels
    int  alphaChannel;    // Ignored by OpenCV
    int  depth;           // Pixel depth in bits:
                          // IPL_DEPTH_8U, IPL_DEPTH_8S, IPL_DEPTH_16S,
                          // IPL_DEPTH_32S, IPL_DEPTH_32F and IPL_DEPTH_64F
    char colorModel[4];   // Ignored by OpenCV
    char channelSeq[4];   // ditto
    int  dataOrder;       // 0 - interleaved color channels,
                          // 1 - separate color channels.
                          // cvCreateImage can only create interleaved images
    int  origin;          // 0 - top-left origin,
                          // 1 - bottom-left origin (Windows bitmaps style).
    int  align;           //  Alignment of image rows (4 or 8).
                          // OpenCV ignores it and uses widthStep instead.
    int  width;           // Image width in pixels.
    int  height;          // Image height in pixels.
```

```
    struct _IplROI *roi;    // Image ROI. If NULL, the whole image is selected.
    struct _IplImage *maskROI;     // Must be NULL.
    void  *imageId;                 // "          "
    struct _IplTileInfo *tileInfo;  // "          "
    int  imageSize;         // Image data size in bytes
                            // (==image->height*image->widthStep
                            // in case of interleaved data)
    char *imageData;        // Pointer to aligned image data.
    int  widthStep;         // Size of aligned image row in bytes.
    int  BorderMode[4];     // Ignored by OpenCV.
    int  BorderConst[4];    // Ditto.
    char *imageDataOrigin;  // Pointer to very origin of image data
                            //(not necessarily aligned) -
                             //needed for correct deallocation */
}
IplImage;
```

其中内容多数为图片参数，具体的像素矩阵存储在 imageData 指向的 char 类型数据块。

例子：

```
IplImage* Src_iplFrame;    // 表示一帧当前图片
Src_iplFrame = cvQueryFrame(capture_camera); //从摄像头视频流的当前位置读取一帧图片
```

char *imageData，该指针指向一个 char 类型的数据块，该数据块的尺寸为：

```
nChannels * width * height = widthStep * height
```

以像素为单位，width 为图片宽度，height 为图片高度，nChannels 为通道数，即每个像素用几个数来表达，一般是三个，即红绿蓝三原色，OpenCV 按 bgr（蓝绿红）的次序存储每个像素的三色值。数据块是连续的线性次序，OpenCV 按行为单位排列像素数据，即一行像素排完后再排第二行，为了处理方便，用

```
widthStep = nChannels * width
```

表示一行像素所占用的以字节计数的内存块尺寸。

OpenCV 用 char 类型表示一个颜色值。通常，一个字节的颜色值的取值范围为 0--255，但 OpenCV 用 char 类型来表示颜色值，其取值范围为-128 到+127，在后续计算时有所不便。转换方法也很简单，直接用 cast：

举例：

```
char b=100;
unsigned char b1=(unsigned char)b;     // 结果：b1=100
char g=-100;
unsigned char g1=(unsigned char)g;     // 结果：g1=156
```

解释：char 型数转换为 unsigned char 型数时，对区间[0,+127]的值，不变；对区间[-128,-1]的值，加 256，挪到区间[128,255]。

3.2 OpenCV 函数

cvCreateCameraCapture，从指定编号的摄像头获取视频流，返回指向CvCapture类型的指针，并指向视频流的开始位置。

原型：CVAPI(CvCapture*) cvCreateCameraCapture(int index);

例子：

```
CvCapture* capture_camera;
capture_camera = cvCreateCameraCapture(1);
```

cvCreateFileCapture，从指定的视频文件获取视频流，返回指向CvCapture类型的指针，并指向视频流的开始位置。

原型：CVAPI(CvCapture*) cvCreateFileCapture(const char* filename);

例子：

```
CvCapture* capture_file;
capture_file = cvCreateFileCapture(fname_video_readin);
```

cvReleaseCapture，释放CvCapture类型的指针变量指向的相关资源。原型如下：

```
CVAPI(void) cvReleaseCapture( CvCapture** capture );
```

cvQueryFrame，从指定的视频流的当前位置获取一帧图片，

例子：

```
IplImage* Src_iplFrame;    // 表示一帧当前图片
Src_iplFrame = cvQueryFrame(capture_camera);  // 从摄像头视频流的当前位置读取一帧图片
```

cvCreateImage，生成一张新的图片，原型如下

```
CVAPI(IplImage*)  cvCreateImage( CvSize size, int depth, int channels );
```

例子：

```
IplImage* Dst_iplFrame =
cvCreateImage(cvSize(Dst_width,Dst_height),IPL_DEPTH_8U,3);
```

此语句完成后，Dst_iplFrame指向的IplImage数据块的参数部分都有内容了，而存储像素矩阵的imageData指针还没有指向具体的char类型数据块，就是还没有具体的像素。

cvReleaseImage，释放IplImage类型的指针变量指向的相关资源。原型如下：

```
CVAPI(void)  cvReleaseImage( IplImage** image );
```

cvCreateVideoWriter，生成一个用于向文件写入视频流的控制器，原型：

```
CVAPI(CvVideoWriter*) cvCreateVideoWriter( const char* filename, int fourcc,
                                    double fps, CvSize frame_size,
                                    int is_color CV_DEFAULT(1));
```

例子：

```
CvVideoWriter * writer = 0;
int isColor = 1;    // 彩色
double fps = 25;    // 帧率
int frameW = Dst_width;    // 帧宽
int frameH = Dst_height;   // 帧高
writer = cvCreateVideoWriter(fname_video_writeout,   // 文件指针
                  CV_FOURCC('X','V','I','D'),         // 编码方案
                  fps,cvSize(frameW, frameH),  isColor);
```

cvWriteFrame，将一帧图片写入外部文件，原型为：

```
CVAPI(int) cvWriteFrame( CvVideoWriter* writer, const IplImage* image );
```

例子：

```
cvWriteFrame(writer, Dst_iplFrame);
```

writer 为 CvVideoWriter* 类型，为写入文件控制器，Dst_iplFrame为IplImage*类型，为一帧图片。

cvReleaseVideoWriter，释放CvVideoWriter类型的指针变量指向的相关资源。原型如下：

```
CVAPI(void) cvReleaseVideoWriter( CvVideoWriter** writer );
```

cvNamedWindow，设置一个屏幕显示窗口，原型：

```
CVAPI(int) cvNamedWindow( const char* name, int flags
CV_DEFAULT(CV_WINDOW_AUTOSIZE) );
```

例子：

```
cvNamedWindow("ORIGINAL", CV_WINDOW_AUTOSIZE);   // 窗口名字、自动尺寸
cvNamedWindow("ALL_IN_ONE", CV_WINDOW_AUTOSIZE);
```

cvShowImage，在指定窗口显示一帧图片，

例子：

```
cvShowImage("ORIGINAL",Src_iplFrame);   // Src_iplFrame 指针变量指向一帧图片
```

cvDestroyWindow，释放由cvNamedWindow函数产生的窗口及相关资源。原型如下：

```
CVAPI(void) cvDestroyWindow( const char* name );  // 只需要指定窗口名字
```

cvWaitKey，按毫秒计，键盘等待函数。

注记：OpenCV 有很多函数和数据结构，我们的项目只使用与视频输入、输出、显示相关的几个函数。

第四章

AIcourse2016 中 cuda 数据结构与函数

4.1 cuda 数据结构

cudaDeviceProp，这是一个用来存储显卡参数的数据结构，包括很多内容，比如，设备名称、内存总容量、版本号等等。

4.2 cuda 函数

cudaSetDevice，原型为：

```
extern __host__ cudaError_t CUDARTAPI cudaSetDevice(int device);
```

__host__ 是指该函数只由 host 端程序调用；
cudaError_t 是返回错误类型；
CUDARTAPI 是表明这是 cuda 运行时函数，在 cuda_runtime_api.h 中声明（外部声明 extern）；
device 为设备号；

将要使用显卡时，需要调用此函数做好准备，这个函数对编程的意义，主要在于指明将要使用哪个显卡设备进行当下的调用。

如何知道设备号？如果电脑中只有一块显卡，设备号默认为0，如果有多块显卡的情况下，可以在cudaSDK开发包中找到一个程序deviceQuery.exe，在命令行下运行该程序，即可显示电脑上的所有显卡设备的详细参数和设备号。

cudaDeviceReset，原型为：

```
extern __host__ cudaError_t CUDARTAPI cudaDeviceReset(void);
```

原注解：

```
Destroy all allocations and reset all state on the current device in the current process.
Explicitly destroys and cleans up all resources associated with the current device in the current process.  Any subsequent API call to this device will  reinitialize the device.
Note that this function will reset the device immediately.  It is the caller's responsibility to ensure that the device is not being accessed by any other host threads from the process when this function is called.
```

该函数调用后，重置当前进程中的当前设备，所以函数不必指出设备号。

cudaGetDeviceProperties，原型为：

```
extern __host__ cudaError_t CUDARTAPI
```

```
cudaGetDeviceProperties(struct cudaDeviceProp *prop, int device);
```

该函数返回设备号为 device 的显卡的各项静态参数，存储在指向 cudaDeviceProp 类型数据结构的指针 prop 指向的数据块中。

cudaDeviceSynchronize，原型为：

```
extern __host__ cudaError_t CUDARTAPI cudaDeviceSynchronize(void);
```

原文及解释如下：

```
Wait for compute device to finish
Blocks until the device has completed all preceding requested tasks.
cudaDeviceSynchronize() returns an error if one of the preceding tasks has failed.
If the cudaDeviceScheduleBlockingSync flag was set for this device, the host thread
will block until the device has finished its work.
```

当host端程序调用device端程序后，host端程序会继续运行下面的程序，并不会停下来等待device端程序运行完成并返回结果，因为device端程序的运行并不占用host端即主板上的资源，这是cuda编程中必须注意的新机制。

但是在实际运行中，host端程序也许需要等待device端程序的运行结果，这时可以调用此函数，使得host端程序暂停下来，等待device端程序完成运行。

cudaMalloc，该函数在显卡内存中按字节数分配数据块并返回其首地址，原型为：

```
cudaMalloc(void **devPtr, size_t size);
```

原文及解释如下：

```
Allocates size bytes of linear memory on the device and returns in *devPtr a pointer
to the allocated memory.
```

在显卡上分配 size 字节数的线性内存区并返回其指针（首地址）给*devPtr。
该解释不太好懂。

例子：
过桥函数：在显卡上分配一块内存区用于存放char类型数据，并返回其首地址。

```
char * cuda_malloc_device_block_of_char(int mem_size)
{
    char * device_block_of_char;
    // 先定义一个指针变量device_block_of_char，
    // 此时其值（一个地址）还没有赋予，

    cudaMalloc((void**) &device_block_of_char, mem_size);
    // cudaMalloc在显卡上分配了内存块，并返回该块首地址给*&device_block_of_char

    return device_block_of_char;
```

```
}
```

问题：*&device_block_of_char是什么？
回答：device_block_of_char 是一个指针变量，假设其存储了一个地址 A，
&device_block_of_char 是该指针变量本身所在的地址 B，
*&device_block_of_char 是取地址 B 中所存储的数据，即地址 A。

根据上述分析，cudaMalloc 在显卡上分配了内存块，并返回该块首地址给 device_block_of_char。注意到，这个地址不在 host 端，而在 device 端，这就好比 device 大楼中的某房间的一把钥匙，可以在 host 大楼内保存并传来传去，但在 host 大楼内却无法使用，要想真正使用这把钥匙，还是要去 device 大楼。

更简单、更一般的过桥函数方案，不必指明要分配的内存块用于何种数据结构，只要给出块尺寸，就返回指针（显卡上的地址），至于这个内存块以后用来存储什么数据，这个函数暂且不急于确定。

例子：
过桥函数：在显卡上分配一块内存区用于存放数据（暂时不必说明数据类型），并返回其首地址。

```
void * cuda_malloc_on_device(int mem_size)
{
    // allocate device memory
    void * device_block_Ptr;
    cudaMalloc((void**) &device_block_Ptr, mem_size);

    return device_block_Ptr;
}
```

实际调用时再用cast方式：

```
char * Src_deviceImageData =
              (char *)cuda_malloc_on_device(Src_mem_size_imageData);
```

cudaFree，该函数释放由cudaMalloc 分配的显卡上的内存空间。函数原型为：

```
cudaFree(void *devPtr);
```

cudaMemcpy，是cuda内存复制函数，按字节数，以字节为单位，复制内存块数据，其原始定义为：

```
extern __host__ cudaError_t CUDARTAPI
cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind);
```

__host__ 是指该函数只由 host 端程序调用；
cudaError_t 是返回错误类型；
CUDARTAPI 是表明这是 cuda 运行时函数，在 cuda_runtime_api.h 中声明（外部声明 extern）；

原注解如下：

```
brief Copies data between host and device
Copies count bytes from the memory area pointed to by src to the memory area pointed
```

to by dst, where kind is one of
将count字节数的内容从首地址src起始的内存块复制到首地址dst起始的内存块，分为以下四种类型：
cudaMemcpyHostToHost, host端到host端，（相当于memcpy函数）
cudaMemcpyHostToDevice, host端到device端，
cudaMemcpyDeviceToHost, device端到host端，
cudaMemcpyDeviceToDevice, device端到device端，
and specifies the direction of the copy.

The memory areas may not overlap. Calling ::cudaMemcpy() with dst and src pointers that do not match the direction of the copy results in an undefined behavior.
复制与被复制的内存区域不能重叠。

dst   - Destination memory address
src   - Source memory address
count - Size in bytes to copy
kind - Type of transfer

注解：（1）这个函数是在host端与device端传递大块数据的主要方法，速度快，功能强大。（2）该函数在使用时要做好准备工作，源地址与目的地址要预先安排分配好，内存块尺寸要相当。

第五章

AIcourse2016 中 hhcv 数据结构与函数

5.1 hhcv 数据结构

HhcvBasePara，基本参数数据结构，

```
#ifndef _HhcvBasePara_DEFINED
struct HhcvBasePara{
    char para_DATA_DIR[51];             // 数据目录
    char para_VIDEO_IN[51];             // 输入视频文件名
    char para_VIDEO_OUT[51];            // 输出视频文件名
    char para_IMAGE_IN[51];             // 输入图片文件名
    char para_IMAGE_OUT[51];            // 输出图片文件名
    int para_RUN_PROCEDURE;             // 程序运行选项
    int para_WAIT_KEY;                  // 帧间等待时间
    int para_VIDEO_SOURCE;              // 0：摄像头输入， 1：视频文件输入
    int para_FRAME_HEIGHT;              // 帧高度
    int para_FRAME_WIDTH;               // 帧宽度
    int para_FRAME_WIDTH_STEP;          // 像素行数据步长
    int para_FRAME_CHANNELS;            // 像素数据通道数，一般为3，即blue green red
    int para_VIDEO_CUT_UP;              // 帧上切行数
    int para_VIDEO_CUT_DOWN;            // 帧下切行数
    int para_VIDEO_CUT_LEFT;            // 帧左切列数
    int para_VIDEO_CUT_RIGHT;           // 帧右切列数
    int para_CUDA_GRID_DIM_X;           // grid布局
    int para_CUDA_GRID_DIM_Y;           // grid布局
    int para_CUDA_BLOCK_DIM_X;          // block布局
    int para_CUDA_BLOCK_DIM_Y;          // block布局
};
#define _HhcvBasePara_DEFINED
#endif
```

注解：

```
#ifndef _HhcvBasePara_DEFINED
```

定义实体

```
#define _HhcvBasePara_DEFINED
#endif
```

此为预编译控制语句，目的是防止将不同的内容定义为相同的名字。小型项目一般不用这种方式，大型的、多人的、长期的项目就会需要此技术。

HhcvWindowSize，各种图片尺寸数据结构

```
struct HhcvWindowSize{
    int Src_channels;          // 源图片
    int Src_height;
    int Src_width;
    int Src_widthStep;
    int Cut_channels;           // 切过的图片，切的目的是使得宽高比为2:1
    int Cut_height;
    int Cut_width;
    int Cut_widthStep;
    int L0_channels;            // resize过的图片的第一层，宽高比为1024:512
    int L0_height;
    int L0_width;
    int L0_widthStep;
    int Dst_channels;           // 最终输出的图片的尺寸，宽高比为1536:768
    int Dst_height;
    int Dst_width;
    int Dst_widthStep;
};
```

HhcvCudaGridDim，cuda计算的线程布局参数数据结构

```
struct HhcvCudaGridDim{
    int cudaGridDimX;           // grid布局
    int cudaGridDimY;           // grid布局
    int cudaBlockDimX;          // block布局
    int cudablockDimY;          // block布局
};
```

HhcvPixel，像素值数据结构

```
struct HhcvPixel{
    unsigned char ch0;    // blue
    unsigned char ch1;    // green
    unsigned char ch2;    // red
    unsigned char ch3;    // 保留

    unsigned char h;      // 色调 hue
    unsigned char s;      // 饱和度 saturation
    unsigned char v;      // 亮度 value, intensity, brightness
};
```

5.2 hhcv 函数

void read_RunPara(char *fname_readin);
从文件名提供的文本文件中读取数据，主要是运行参数。

int hhcv_strlen(char * str);
小工具函数，计算字符串str的长度，并返回其值。

int hhcv_fetch_substr(char *host_str, char *sub_str, int start, int end);
小工具函数，从主字符串host_str中提取子串赋给sub_str，并返回子串长度，开始位置start，结束位置end。

int hhcv_strcmp(char * str1, char * str2);
小工具函数，比较串str1和串str2，如果一样，返回0，否则返回1。

void hhcv_rtrim_copy(char * str1, char * str2);
小工具函数，将串str1从第一个空格开始截掉，结果赋给串str2。

void hhcv_strmerge(char * str1, char * str2, char * str3);
小工具函数，将串str1和串str2首尾相接组成一个新串，结果赋给串str3。

第六章

cuda 核函数与调用方法

6.1 核函数基本概念

gpu 计算的基本模式是并行，这个并行不是单核 cpu 或多核 cpu 环境下的模拟并行，而是在很多核的 gpu 环境下的物理并行，很多核有多少呢？这个要看显卡的具体型号，比如 GTX1080 有 2560 个 cuda 处理器（cuda cores），当然，这个核与 cpu 的核（4 核、6 核、8 核等）不是一个概念，如果说 cpu 的核是个大型工厂的话，那么每个 cuda 核只相当于一个小的操作间。

gpu 的并行计算方案是，众多的 cuda 核同时运行相同的代码，处理结构相同、但内容不同的数据，称为一个线程（thread）。

举个例子：cuda 核相当于是工厂的操作间，每个操作间有自己的门牌号，执行相同的操作流程（代码），数据相当于待加工的零部件，初始存放在仓库里，零部件虽然结构类同，但毕竟是不同的实体，有不同的条码，要按预定的计划分别运送到相应的操作间进行加工，各个操作间完成加工后再运送到预备好的仓库里，这就实现了一个并行加工流程。

注意这里的关键是三点：
（1）操作流程：即相同的代码；
（2）操作间的编号：即代码的多个实现场所；
（3）数据的编号：即不同的操作间取不同的数据来加工，并送到相应的地址。

核函数是在显卡上运行的并行代码，这种函数与一般 host 端函数相比的特殊之处在于，要指明此函数的运行场所、以及每个场所从哪里取数据、结果数据送到哪里去。

6.2 线程的布局

前面提到的操作间是形象的比喻，gpu 计算的标准名称是线程（thread），所有线程的总体布局由编程者安排，具体编号由 cuda 运行环境负责，为内建变量，编程者可以使用，不能修改。图 6.1 为线程布局示意图。

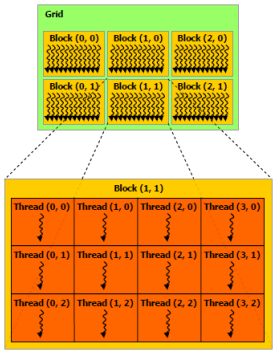

图 6.1 Grid & Blocks & Threads ： 6 blocks in the grid, 12 threads in one block, then 72 threads in the grid

Grid 为总体布局名称，Grid 中包含若干个 Blocks，Grid 中的 Blocks 可以按一维、二维或三维布局，每个 Block 又包含若干个 Threads，Block 中的 Threads 也可以按一维、二维或三维布局。

例子：过桥函数，调用核函数，在gpu中将一帧图像的 imageData 型数据逐点转换为 HhcvPixel 型数据。

```
void cuda_SrcImageData_to_SrcPixel          // 函数名称
(
HhcvPixel * Src_devicePixelFrame,           // 转换完成后数据的存放地址，在显卡上
char * Src_deviceImageData,                 // 待转换数据的的存放地址，在显卡上
HhcvWindowSize cudaWinSize,                 // 图片尺寸数据
HhcvCudaGridDim cudaGridDim                 // 线程布局数据
)
{
    // 先定义两个线程布局变量

    dim3 grid_pixel_level
        (cudaGridDim.cudaGridDimX,cudaGridDim.cudaGridDimY, 1);
    // dim3为cuda内建（built-in）变量，为一个3维整型向量，
    // grid_pixel_level含义为Grid中Blocks的布局，实为一个二维布局，
    // grid_pixel_level为自定义的dim3型变量，其值在后面括号中给出，
    // （cudaGridDimX，cudaGridDimY，1）
    // cudaGridDimX，cudaGridDimY根据图片的尺寸计算得来，
    // 比如640x480的图片，cudaGridDimX=60，cudaGridDimY=30，

    dim3 thread_pixel_level
         (cudaGridDim.cudaBlockDimX,cudaGridDim.cudablockDimY,1);
    // thread_pixel_level含义为每个Block中Threads的布局，实为一个二维布局，
    // thread_pixel_level为自定义的dim3型变量，
    // 其值在后面括号中给出，cudaBlockDimX，cudablockDimY，1
    // cudaBlockDimX，cudablockDimY根据显卡型号推荐值而定，
    // 一般选择32、16、8等，程序中取16，

    // 总结：根据以上两个变量的安排，线程总数为：960x480(宽x高)，图片为640x480，
    // 满足至少一个线程处理一个像素的要求。

    // 下面调用gpu核函数（并行线程），在显卡上并行执行
    global_SrcImageData_to_SrcPixel
     <<<grid_pixel_level, thread_pixel_level>>>
     (Src_devicePixelFrame, Src_deviceImageData, cudaWinSize);
     // global_SrcImageData_to_SrcPixel为核函数名称，从host端调用
     // <<<grid_pixel_level, thread_pixel_level>>>线程布局方案，
     // 注意这个特殊的表达方式<<<blocks_in_grid,threads_in_everyBlock>>>，
     // <<<,>>>由nvcc负责解析
     // Src_devicePixelFrame 转换完成后数据的存放地址，在显卡上
```

```
    // Src_deviceImageData 待转换数据的的存放地址，在显卡上
    // cudaWinSize 图片尺寸数据

    cudaDeviceSynchronize();
    // 等待核函数执行完成的同步函数
}
```

注解：两个线程布局变量的设计原则，在cuda编程指南中，对这个问题没有很深入的探讨，这涉及到显卡硬件结构的深层次认知，有些专著对此有些讨论。

这里给出一个简单的原则：grid中的blocks个数按流处理器数的倍数，block中的threads个数按2的幂次。

这里流处理器数是指 SM（streaming multiprocessors），我的电脑用的是Tesla C1060计算卡，有30个流处理器SM，每个SM有8个CUDA core，共有240个cuda核。

又例，比较经典的 GeForce GTX1080，有 20 个流处理器 SM，每个 SM 有 128 个 CUDA core，共有 2560 个 cuda 核。

6.3 核函数的结构与运行

上节函数

```
global_SrcImageData_to_SrcPixel<<<grid_pixel_level, thread_pixel_level>>>
    (Src_devicePixelFrame, Src_deviceImageData, cudaWinSize);
```

中<<<blocks_in_grid,threads_in_everyBlock>>>用来对线程的布局进行安排，当调用这个函数时，cuda 运行环境首先得到这个布局安排，即对如下两个内建变量(built-in)进行初始化：

```
gridDim = (gridDim.x, gridDim.y, gridDim.z)      // grid 中的 blocks 个数，3 维
blockDim = (blockDim.x, blockDim.y, blockDim.z) // 每个 block 中 threads 个数，3 维
```

这实际上类似于函数的参数表，按照 c 语言风格可以表示为 <<<gridDim, blockDim>>> 或 <<<dim3, dim3>>>，只是这个参数表格式对任何从 host 端调用的核函数而言都是固定的，所以在类似如下核函数定义中就省略了。

当 gridDim 和 blockDim 确定后，线程的总数以及布局安排就确定了，但是并行执行的核函数还需要对每个线程进行标定，好比每个操作间要有自己的房间号，这样才能与待加工的数据分别对应。

为此，cuda 运行环境安排如下两个内建变量对线程进行标定，分别为块索引 blockIdx 和线程索引 threadIdx：

```
blockIdx = (blockIdx.x, blockIdx.y, blockIdx.z)
// grid 中的每个 block 编号，3 维
```

threadIdx = (threadIdx.x, threadIdx.y, threadIdx.z)
// block 中的每个 thread 编号，3 维

其中各个分量的取值范围为：

$$\begin{cases} \text{blockIdx.x} = 0 \sim \text{gridDim.x} - 1 \\ \text{blockIdx.y} = 0 \sim \text{gridDim.y} - 1 \\ \text{blockIdx.z} = 0 \sim \text{gridDim.z} - 1 \end{cases} \qquad \begin{cases} \text{threadIdx.x} = 0 \sim \text{blockDim.x} - 1 \\ \text{threadIdx.y} = 0 \sim \text{blockDim.y} - 1 \\ \text{threadIdx.z} = 0 \sim \text{blockDim.z} - 1 \end{cases}$$

调用的 gpu 函数即为如下定义的核函数。
例子：核函数，在gpu中将一帧图像的 imageData 型数据逐点转换为 HhcvPixel 型数据，实现方案为一个线程处理一个像素。

```
__global__ void                     // __global__ 为cuda标记，
                                    // 表示该函数只能由host端函数调用
global_SrcImageData_to_SrcPixel     // 函数名
(
HhcvPixel * Src_devicePixelFrame,   // 转换完成后数据的存放地址，在显卡上
char * Src_deviceImageData,         // 待转换数据的的存放地址，在显卡上
HhcvWindowSize cudaWinSize          // 图片尺寸数据
)
{
    int thread_i_y = blockIdx.y * blockDim.y + threadIdx.y;
     // thread_i_y 为线程行编号，将要与像素行编号对应

    int thread_j_x = blockIdx.x * blockDim.x + threadIdx.x;
     // thread_j_x 为线程列编号，将要与像素列编号对应

     // 这里为什么要另外定义线程编号？
     // 线程行编号 thread_i_y 与线程列编号 thread_j_x 是唯一值，
     // 而内建变量 threadIdx.y 与 threadIdx.x 虽然也表示线程编号，但是不是唯一的，
     // 每个 block 中都有相同的 threadIdx.y 与 threadIdx.x

    int frameHeight   = cudaWinSize.Src_height;        // 图片参数
    int frameWidth    = cudaWinSize.Src_width;         // 图片参数
    int frameChannels = cudaWinSize.Src_channels;      // 图片参数
    int widthStep     = cudaWinSize.Src_widthStep;     // 图片参数

    if (thread_i_y < frameHeight && thread_j_x < frameWidth)
    // 当线程编号与图片中的像素能够相对应时，这个线程就执行像素值的格式转换
    // 线程布局总数应该大于等于像素总数，使得有些线程编号不能与像素相对应，即不满足此 if 条件
    {
        Src_devicePixelFrame[thread_i_y * frameWidth + thread_j_x].ch0
```

```
        = (unsigned char)
        Src_deviceImageData[thread_i_y*widthStep + thread_j_x*frameChannels
+0];

      Src_devicePixelFrame[thread_i_y * frameWidth + thread_j_x].ch1
        = (unsigned char)
        Src_deviceImageData[thread_i_y*widthStep + thread_j_x*frameChannels
+1];

      Src_devicePixelFrame[thread_i_y * frameWidth + thread_j_x].ch2
        = (unsigned char)
        Src_deviceImageData[thread_i_y*widthStep + thread_j_x*frameChannels
+2];
    }// endif
}//end of func
```

6.4 cuda 内存结构

cuda 线程可以访问多种内存空间，如图所示，每个线程 thread 有自己的私有内存空间（private local memory）。每个块 block 有对块内的线程 thread 共享内存空间（shared memory），所有的线程 thread 都可以访问全局内存（global memory）。

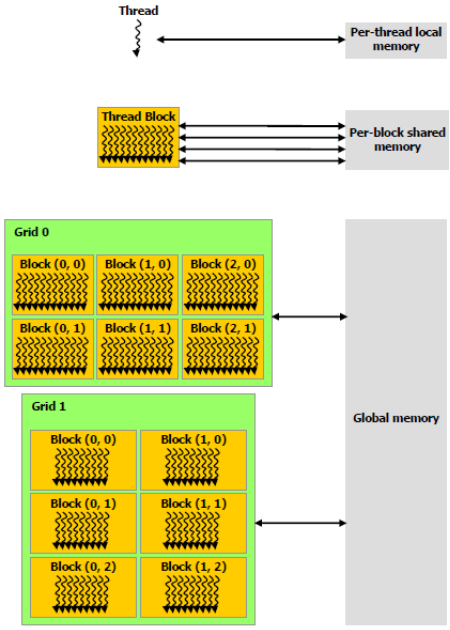

图 6.2 Memory Hierarchy

以前面解析的核函数 **global_SrcImageData_to_SrcPixel** 为例，核函数内定义的几个变量（**thread_i_y** 等）使用的是私有内存空间，参数表中的地址变量（**Src_devicePixelFrame 等**）指向的是全局内存。这里没有使用共享空间。

不同级别的内存空间的物理实现是不同的，访问速度相差很大，私有内存空间和共享内存空间的访问速度远快于全局内存空间。所以，不同的显卡对私有内存空间和共享内存空间的数量都有限制，可以通过前面介绍的 deviceQuery.exe 程序查询，主要是以下三个参数（Tesla C1060）：

全局内存 Total amount of global memory: 4096 MBytes (4294770688 bytes)
每块共享内存 Total amount of shared memory per block: 16384 bytes
每块寄存器变量数（私有内存）Total number of registers available per block: 16384

私有内存中的寄存器变量访问速度是最快的，但是数量不能太多，编程时需要注意。比如上面例子中每块 block 中有 16x16=256 个 threads，每个 thread 中用了 6 个 register 变量，合计每个 block 中使用了 1536 个 register 变量，不超过 16384。