第3章 图像并行处理环境构建

如果说多CPU并行的解决方案象征着处理器外的高性能计算解决方案的话，那么GPU的崛起就代表着高性能计算方案向处理器本身性能的回归。在不同应用中，各自有各自的特点和优势，孰优孰劣目前尚无定论，因此需要依据具体应用来确定。

对于图像处理应用来说，如果不是极其特殊的图像处理需求，现有的GPU硬件已经能够满足绝大多数的图像处理要求，且从实时性、便携性和成本等方面来考虑，GPU也是高性能图像处理非常好的选择。对于图像算法的研究者来说，终于可以专注于算法本身，而不用再为算法实时性的优化而绞尽脑汁了。

本书所用并行图像处理方案为Nvidia（英伟达）费米架构计算能力1.2的GPU显卡，相应地，也用到了英伟达显卡并行计算的通用标准CUDA来完成核心代码的设计和编译，所用CUDA版本为3.0。此外，还用到目前主流的开源计算机视觉库OpenCV来负责图像的载入和显示等工作。操作系统为Windows 7 32位系统，所用编程环境为VS2005。

Nvidia、AMD、Intel、三星都生产GPU，本书选择了Nvidia的GPU作为图像处理的硬件。作为当前主流高性能计算硬件提供商，Nvidia的GPU和CUDA编程规范设计十分经典，便于开发者进行自己的设计开发，掌握了CUDA设计思路和方法的读者也能很容易掌握其他GPU的编程。从图像处理的角度来看，不管是学习并行算法，还是要实际应用，Nvidia的GPU都是一个不错的选择。

Nvidia成立于1993年，是一家以设计显示芯片和主板芯片组为主的半导体公司。2001年，发布了第一个可编程GPU(GeForce 3)，其后的十几年间，Nvidia的GPU不断进步，浮点运算能力也不断提高。目前，GPU性能远超其他公司，并且有专业且开源的编程标准CUDA作为支持，Nvidia的GPU正迅速成为高性能计算方案的首选。

前面提到过，CUDA是Compute Unified Device Architecture的缩写，是英伟达公司推出的一个完整的通用计算图形处理器解决方案，提供了硬件的直接访问接口，方便GPU解决复杂的计算问题。它包含了CUDA指令集架构(ISA)以及GPU内部的并行计算引擎，开发者可以使用C语言来为CUDA架构编写程序，所编写出的程序可

以在支持 CUDA 的 GPU 处理器上以超高性能运行。

OpenCV 的全称是 Open Source Computer Vision Library，即开源计算机机器视觉库，是一个跨平台的计算机视觉库，可以运行在 Linux、Windows 和 Mac OS 操作系统上。它由一系列 C 函数和少量 C++类构成，同时提供了 Python、Ruby、MATLAB 等语言的接口，高效地实现了图像处理和计算机视觉方面的很多通用算法。

OpenCV 中的函数能够很方便地对图像数据进行操作，并且 OpenCV 的图像结构通用性非常好，只需要调用简单的函数就能完成原本复杂的功能，可以节约很多 VC++中操作 BMP 图像的代码，使得程序更加精炼，也让读者在看程序时能专注于理解并行计算函数代码本身，而不被其他诸如载入和显示 BMP 图像的复杂代码所干扰。

书中所用 OpenCV 版本为 OpenCV 1.0。在新版本的 OpenCV 中，已经增加了对 CUDA 的支持，部分图像处理算法在安装有英伟达显卡的计算机上能够有更快的速度。

这一章将介绍如何从头搭建并配置一个用于图像处理的 VC++并行计算环境。所用操作系统为 Windows 7 的 32 位系统，编程环境为 VS2005，CUDA 版本为 3.0，OpenCV 版本为 1.0。内容包括建立一个基于 VS2005 的简单对话框工程、安装和使用 OpenCV，以及安装配置 CUDA 环境。

注意：支持 CUDA 的 Visual Studio 版本为 2005～2010，低版本的开发环境不支持。

3.1 建立一个简单的对话框工程

先从一个简单的 MFC 对话框工程开始。如图 3.1.1 所示，打开 VS2005，选择 File（文件）→New（新建）→Project（工程），新建基于 Dialog 的工程。

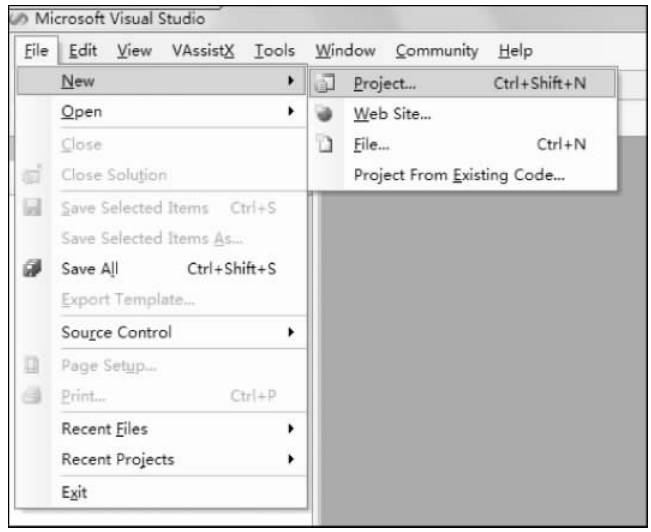

图 3.1.1 新建工程

如图 3.1.2 所示，在弹出的 New Project（新建项目）对话框中，在左侧的 Project types（项目类型）中选中 MFC，在右侧的 Templates（模板）中选择 MFC Application（MFC 应用），在 Name 文本框中输入自己想要的工程名字，如 MyProcess，单击 OK 按钮。

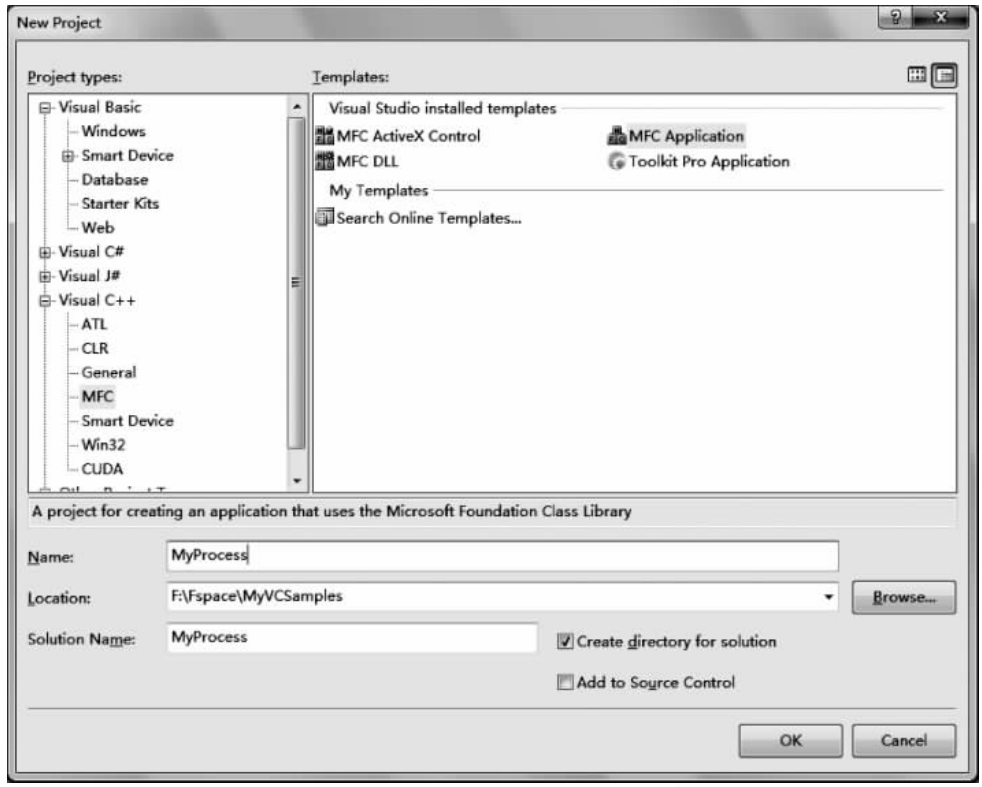

图 3.1.2　选择 MFC 工程

在弹出的 MFC Application Wizard(MFC 向导)对话框中(如图 3.1.3 所示),左侧选中 Application Type(应用类型),然后在右侧选中 Dialog based(基于对话框),然后单击 Finish(完成)按钮。这样,就建立了一个最简单的对话框工程。

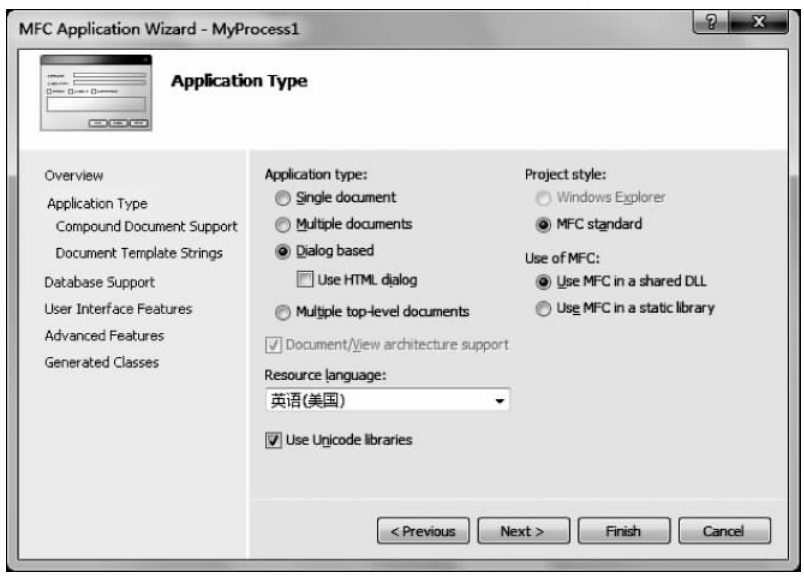

图 3.1.3　建立对话框

要查看工程是否已经建立完成，可按 F5 键运行，将出现如图 3.1.4 所示界面。

图 3.1.4　建立完成的对话框

至此，一个基于 MFC 的对话框工程就建立完成了。

3.2　用 OpenCV 显示 Hello World

注意，本节 OpenCV 的配置是建立在作者提供的 OpenCV 库的基础上，该库是在安装完 OpenCV 1.0 之后编译生成的，编译生成库的过程就不在本书介绍了。

在 MyProcess 工程上右击，在弹出的对话框中选择 Properties(属性)，如图 3.2.1 所示。

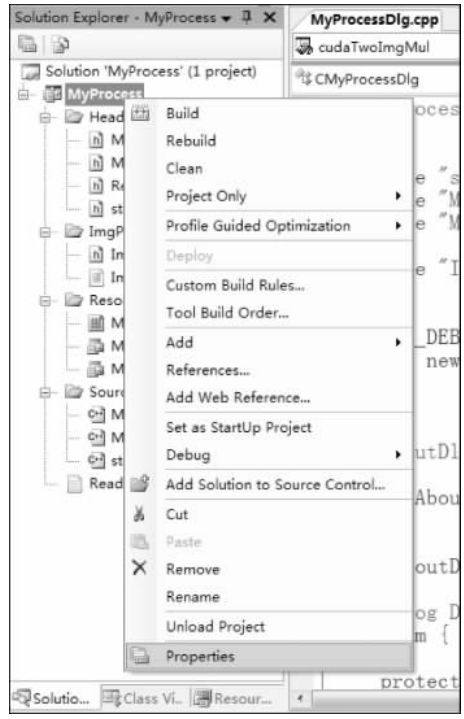

图 3.2.1　选择工程属性

如图 3.2.2 所示，会弹出一个属性对话框，选择 Configuration Properties（结构特性）→C/C++→General（常规），在右侧的 Additional Include Directories（附加包含路径）中输入所示的内容。然后在 Language（语言）中将 OpenMP Support（OpenMP 支持）的属性改为 Yes（是）。

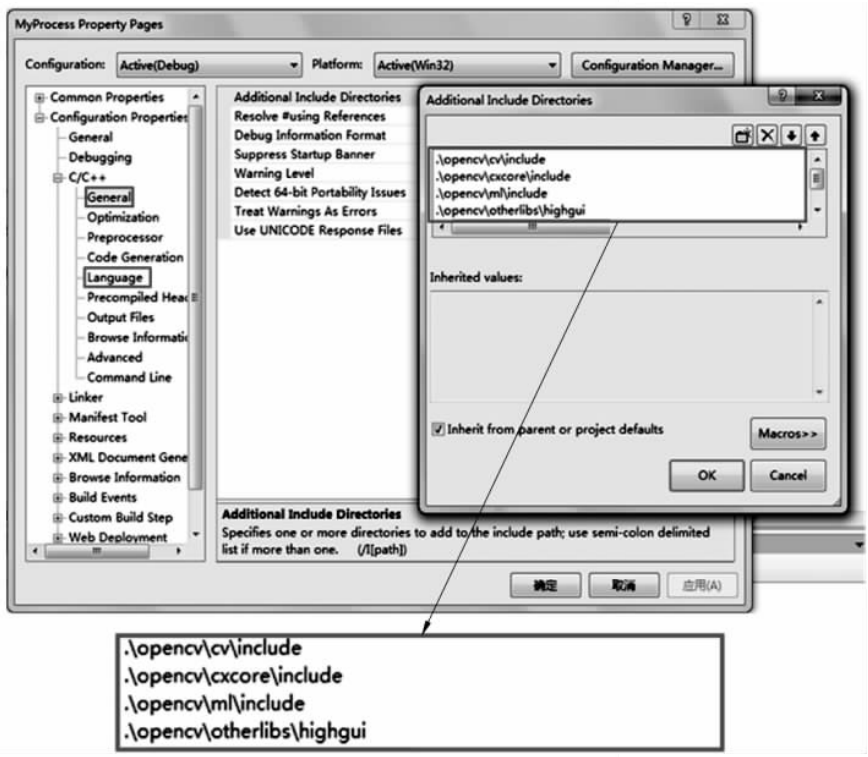

图 3.2.2　添加要包含的目录路径

再选中左侧 Linker（链接器）中的 General（常规），在 Additional Include Directories 中输入“.\opencv\lib”；并在 Input（输入）的 Additional Dependencies（附加依赖项）中加入如图 3.2.3 所示内容。

然后将 OpenCV 1.0 的 cxcore100.dll、highgui100.dll、libguide40.dll 三个动态库复制到系统目录或程序运行目录，这样程序就可以使用 OpenCV 的功能了。

现在，用 OpenCV 来进行图像数据的读入和显示，初步测试一下程序和环境配置。

在建立好的对话框中加入一个按钮，将其命名为 Hello World，并添加其单击响应函数，响应函数代码如下所示。

```
//////////////////////////////////////////////////////////////////////////
void CMyProcessDlg::OnBnClickedButtonHelloWorld()
{
    IplImage * pImgLoad;
    pImgLoad = cvLoadImage("..\\MyProcess\\TestImg\\Hello World\\Hello World.bmp",
        CV_LOAD_IMAGE_ANYDEPTH | CV_LOAD_IMAGE_ANYCOLOR);
    cvNamedWindow("Hello World",0);
```

```
    cvShowImage("Hello World",pImgLoad);
    cvReleaseImage(&pImgLoad);
}
//////////////////////////////////////////////////////////////////////////
```

编译运行,单击界面上的 Hello World 按钮,将弹出如图 3.2.4 所示的图像。

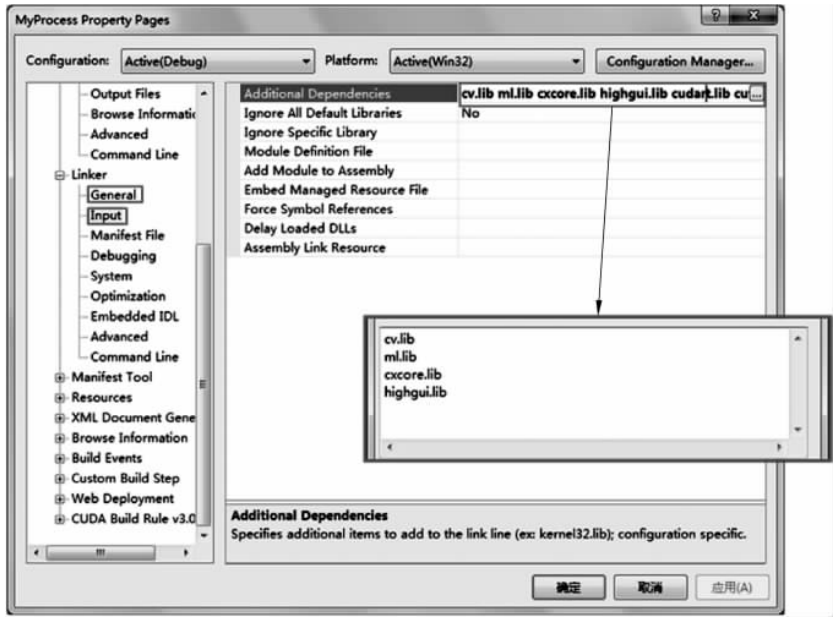

图 3.2.3　链接器设置

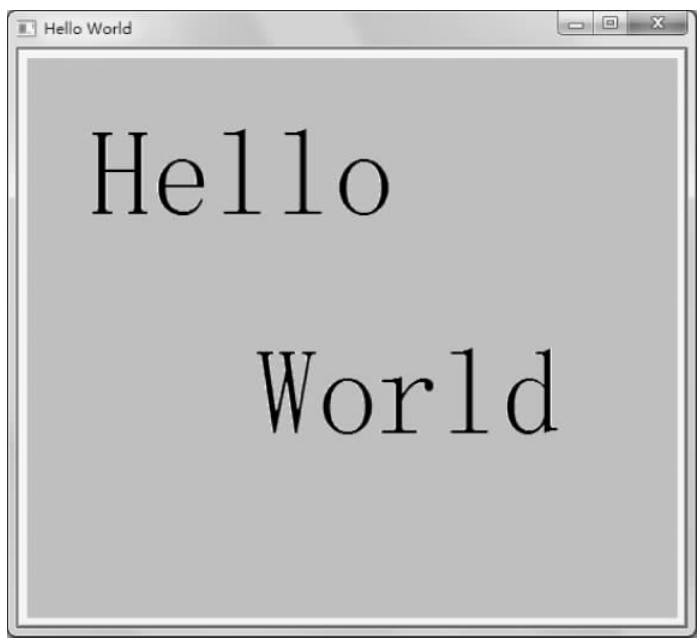

图 3.2.4　OpenCV 显示 Hello World

这样就用 OpenCV 实现了图像数据的载入和显示。同时也得到 IplImage 结构，方便对其中的图像数据进行需要的算法处理。

3.3 安装配置 CUDA 环境

上面实现了对话框和 OpenCV 环境的搭建，本节将安装和配置 CUDA 环境。

依次安装 cudatoolkit_3.0_win_32.exe 和 gpucomputingsdk_3.0_win_32.exe，安装路径选择默认即可。安装完成后，可以先运行 SDK 中的 deviceQuery.exe。作者的安装路径在 C:\Win7Program\NVIDIA Corporation\NVIDIA GPU Computing SDK\C\bin\win32\Release，运行这个自带的程序，将会在当前文件夹中输出一个 deviceQuery.txt 的文档，记录当前使用显卡的性能。

这里使用的显卡信息如下：

```
////////////////////////////////////////////////////////////////////////////
C:\Win7Program\NVIDIA Corporation\NVIDIA GPU Computing SDK\C\bin\win32\Release\
deviceQuery.exe Starting...

CUDA Device Query (Runtime API) version (CUDART static linking)

There is 1 device supporting CUDA

Device 0: "GeForce GTS 450"
  CUDA Driver Version:                           5.50
  CUDA Runtime Version:                          3.0
  CUDA Capability Major revision number:         2
  CUDA Capability Minor revision number:         1
  Total amount of global memory:                 1073414144 bytes
  Number of multiprocessors:                     4
  Number of cores:                               128
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.57 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     Yes
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads can use this
                                                 device simultaneously)
```

```
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 5.50, CUDA Runtime Version =
3.0, NumDevs = 1, Device = GeForce GTS 450


PASSED

Press < Enter > to Quit...
////////////////////////////////////////////////////////////////////////////
```

如果测试最后显示 PASSED,那么说明你的显卡支持 CUDA 并行计算,可以继续进行下面的操作了。

然后,在工程属性中进行 CUDA 配置,方法类似于 OpenCV 的配置。

在 Additional Include Directories(附加包含路径)中加入 $(CUDA_INC_PATH)(安装 CUDA toolkit 之后自动配置的系统变量,如果没有,可以自己手动添加)。

在 Additional Include Directories(附加库路径)中加入 $(CUDA_LIB_PATH)。

在 Additional Dependencies(附加依赖项)中加入 cudart.lib 和 cufft.lib。

再在工程上右击选择 Custom Build Rule(自定义生成规则),在弹出的对话框中选择 Find Existing,在弹出的对话框中找到“C:\Win7Program\NVIDIA Corporation\NVIDIA GPU Computing SDK\C\common”路径下的 Cuda.rules 文件,如图 3.3.1 所示。

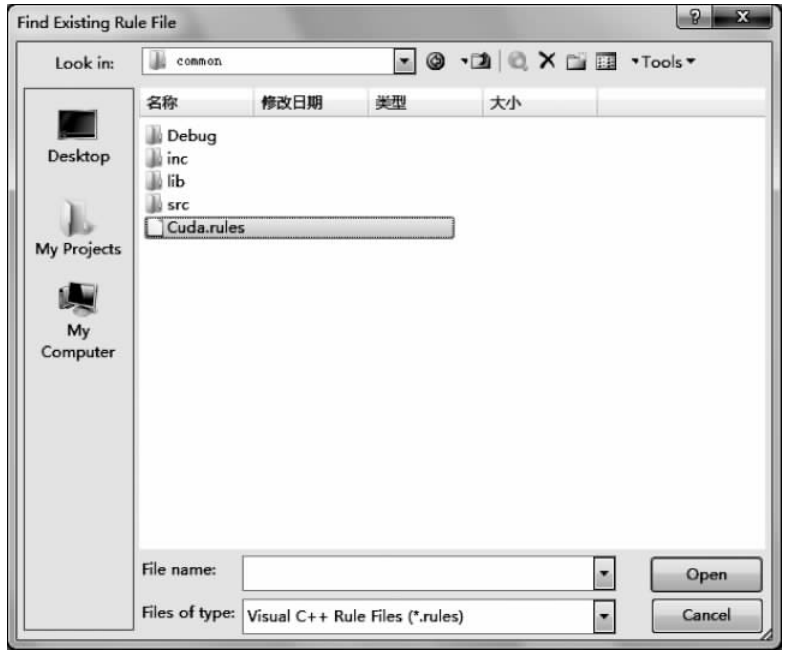

图 3.3.1 自定义生成规则

然后在 Available Rule Files 中选中 CUDA Build Rule v3.0.14,单击 OK 按钮,完成 CUDA 的整个设置。

3.4 用 CUDA 进行并行图像处理

完成 CUDA 安装和配置之后，可以用一个小程序简单测试 CUDA 环境。这里，来看一下分别用 MATLAB、C++和 CUDA 这三种方式来实现同一个图像负相的功能。

负相变换也叫反相变换，即将每一个像素灰度值反转，是与正常采集目标明暗度相反或色彩互为补色的表现效果。对于灰度图像或彩色图像每个通道，其变换公式为：

$$g(x,y) = 255 - f(x,y) \tag{3.4.1}$$

【例 3.4.1】 MATLAB 实现：MATLAB 中使用了两个 for 循环来遍历图像中的每个像素点，并进行处理。完整的 MATLAB 代码如下：

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
f = imread('Hello World.bmp');
[M,N] = size(f);
g = zeros(M,N);
f = double(f);
g = double(g);
  for i = 1:M
     for j = 1:N
          g(i,j) = 255 - f(i,j);
     end
  end
  figure;
  subplot(121);
  imshow(f,[]);
  subplot(122);
  imshow(g,[]);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

【例 3.4.2】 C 语言实现：将上述程序用普通的 C 语言代码实现，先通过 OpenCV 得到要处理的原图数据，并用原图信息建立出结果图信息及数据空间。对 RGB 三通道的计算都相同，图像处理的搜索宽度可以是图像宽度（nWidth）×通道数（nWidthStep），实现代码如下：

```
//////////////////////////////////////////////////////////////////////////
    int iii,jjj;
    for(jjj = 0; jjj < nHeight; jjj++)
    {
        for(iii = 0; iii < nWidth * nChannels; iii++)
        {
            pDestCPU[jjj * nWidthStep + iii] = 255 - pSrc[jjj * nWidthStep + iii];
        }
    }
//////////////////////////////////////////////////////////////////////////
```

【例 3.4.3】 CUDA 内核函数并行实现：在工程中添加一个 InversHelloWorld.cu 文件，流程将分配给 GPU 进行计算，按照一个线程处理一个像素点的思想分配 GPU 资源，可实现像素点的完全并行计算。这里给出了一个比较常见的 CUDA 内核(kernel)函数并行计算调用方式，读者可以先对其结构有一个了解，更为详细的使用方法将在下一章介绍。

内核(kernel)函数 InverseImg_kernel 代码如下：

```
////////////////////////////////////////////////////////////////////////
__global__ void
InverseImg_kernel(BYTE * pImgOut, BYTE * pImgIn,int nWidth, int nHeight, int nWidthStep)
{
    const int ix = blockIdx.x * blockDim.x + threadIdx.x;
    const int iy = blockIdx.y * blockDim.y + threadIdx.y;

    if( ix < nWidth && iy < nHeight )
    {
        pImgOut[ iy * nWidthStep + ix ] = 255 - pImgIn[ iy * nWidthStep + ix ];
    }
}
////////////////////////////////////////////////////////////////////////
```

对于内核(kernel)函数的线程分配如下：

```
////////////////////////////////////////////////////////////////////////
dim3 ts(16,16);
dim3 bs( (nWidth * nChannels + 15)/16,(nHeight + 15)/16);
InverseImg_kernel <<< bs,ts >>>(d_pImgOut, d_pImgIn, nWidth * nChannels, nHeight, nWidthStep);
////////////////////////////////////////////////////////////////////////
```

运行结果如图 3.4.1 所示，得到的 Hello World 是不是和图 3.2.4 有所不同，字体和背景颜色都相反了。

图 3.4.1 经过 CUDA 并行处理为负相的 Hello World

感兴趣的读者可以将内存中的数据(pDestCPU)与显存中的数据(pDest)进行一对一比较,所得的结果应该是完全一致的。

至此,OpenCV 和 CUDA 环境搭建和配置完全成功,可以开始尝试进行图像处理算法的并行实现。

3.5 使用 OpenCV 读入、处理和显示图像

在开始介绍 CUDA 编程和使用 GPU 对图像进行并行处理之前,首先来了解一下 OpenCV。在 OpenCV 中,图像是通过 IPLImage 结构体定义的,里面包含了图像的宽(width)、高(height)、存储地址(imageData)、通道数(nChannels)、一行所占的字节数(widthStep)等等信息,对于一幅图像 pImage 可以通过 pImage→imageData 来访问其图像数据,其他信息的访问方式类似。一幅单通道灰度图像在 OpenCV 中的数据形式如图 3.5.1 所示。

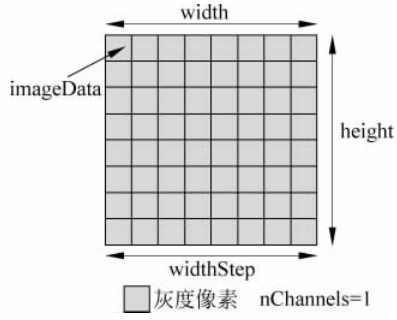

图 3.5.1 单通道灰度图像在 OpenCV 中的数据形式

例 3.5.1 是一段添加到“负相引例”按钮单击响应函数的代码,主要完成读入 bmp 图像、创建图像文件、图像负相处理、显示等工作。其中包含了 OpenCV 几个常用的图像操作函数。

【例 3.5.1】 “负相引例”按钮单击响应函数代码。

```
//////////////////////////////////////////////////////////////////////////
void CMyProcessDlg::OnBnClickedButtonCudaInverse()
{
    cvDestroyAllWindows();
    LoadIPLImage("..\\MyProcess\\TestImg\\Hello World\\Hello World.bmp",
        m_pcvImgIn1);
    CreateIPLImage(m_pcvImgOut1,m_pcvImgIn1);
    int nWidth = m_pcvImgIn1->width;
    int nHeight = m_pcvImgIn1->height;
    int nChannels = m_pcvImgIn1->nChannels;
    int nWidthStep = m_pcvImgIn1->widthStep;
    BYTE* pSrc = (BYTE*)m_pcvImgIn1->imageData;
    BYTE* pDest = (BYTE*)m_pcvImgOut1->imageData;

    //进行图像负相处理的函数
```

```
    cudaInverseImg(pDest,pSrc,nWidth,nHeight,nWidthStep,nChannels);

    CreateIPLImage(m_pcvImgOut2,m_pcvImgIn1);
    BYTE * pDestCPU = (BYTE * )m_pcvImgOut2 -> imageData;
    int nWindowWidth = 512;
    int nWindowHeight = 512;
    int nOffX = 128;
    int nOffY = 256;
    cvNamedWindow("Hello World",0);
    cvResizeWindow("Hello World",nWindowWidth,nWindowHeight);
    cvMoveWindow("Hello World",nOffX,nOffY);
    cvShowImage("Hello World",m_pcvImgIn1);
    cvNamedWindow("Inverse Img",0);
    cvResizeWindow("Inverse Img",nWindowWidth,nWindowHeight);
    cvMoveWindow("Inverse Img",nOffX + nWindowWidth + 40,nOffY);
    cvShowImage("Inverse Img",m_pcvImgOut1);
}
//////////////////////////////////////////////////////////////////////////
```

上面这段代码中，调用了 OpenCV 中的 cvDestroyAllWindows、LoadIPLImage、CreateIPLImage、cvNamedWindow、cvResizeWindow、cvMoveWindow、cvShowImage 这几个函数。其中：

cvDestroyAllWindows 的作用是销毁所有 OpenCV 创建的 HighGUI 窗口；

LoadIPLImage 函数调用了 OpenCV 中的 cvLoadImage，从指定路径读入图像，并存放到给定的内存空间中；

CreateIPLImage 函数调用了 OpenCV 中的 cvCreateImage，使用一幅图像创建另一幅相同的图像；

cvNamedWindow 用来创建指定的窗口；

cvResizeWindow 用来设定窗口大小；

cvMoveWindow 用来设定窗口的位置；

cvShowImage 用来在指定窗口中的显示图像。

此外，例 3.5.1 中还涉及了图像数据结构的读取，操作方法如下：

```
int nWidth = m_pcvImgIn1 -> width;                    //图像行像素数
int nHeight = m_pcvImgIn1 -> height;                  //图像列像素数
int nChannels = m_pcvImgIn1 -> nChannels;             //图像的通道数
int nWidthStep = m_pcvImgIn1 -> widthStep;            //排列的图像行大小，即存储一行
                                                      //像素需要的字节数
BYTE * pSrc = (BYTE * )m_pcvImgIn1 -> imageData;      //图像数据的指针
```

而 cudaInverseImg 这个函数才是用来处理图像负相的核心函数。

3.6 CUDA 编程简介及其在图像处理中应用

使用 CUDA 进行 GPU 的并行程序设计是一门独立系统的学问，涉及方法学、软件、硬件、GPU 架构、相互协同以及优化等诸多问题，可以说，CUDA 是一个“易于上手，难于

精通”的编程方式。

3.6.1 主机端和设备端

目前的 GPU 还只是一个高性能的处理核心，无法像 CPU 一样完全独当一面。使用 GPU 进行图像处理，除了需要 GPU 本身外，还需要 CPU 的配合和参与，成为一个 CPU 和 GPU 组成的异构计算系统。在这个系统中，主机端(host)和设备端(device)是 GPU 编程首先要了解的概念，这里所说的主机端主要是指 CPU 端，而设备端就是指 GPU 显卡，其各自关系如图 3.6.1 所示。CPU 操作范围包括内存和其他主机外设，GPU 操作范围包括显存和显卡上的其他存储器，而内存和显存之间使用 cudaMemcpy 函数移动数据。在图像处理过程中，各自所负责的功能是不同的，下面通过一个例子来帮助读者理解。

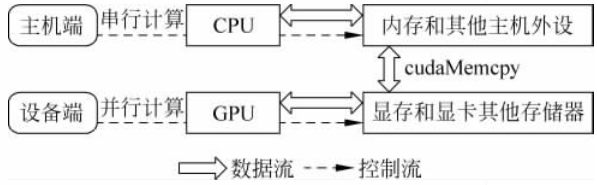

图 3.6.1 主机端和设备端

例 3.6.1 中 cudaInverseImg 是实现一幅灰度图像负相的主机端函数，该函数在示例中由“负相引例”按钮单击响应函数来调用，其参数为：pImgOut 输出图像内存地址指针、pImgIn 输入图像内存地址指针、nWidth 图像像素宽、nHeight 图像像素高、nWidthStep 存储一行像素需要的字节数、nChannels 图像通道数。

cudaInverseImg 函数中又调用了内核函数 InverseImg_kernel 来进行图像每一个像素灰度值的负相操作。对像素的操作中，用一个线程负责一个像素点。

【例 3.6.1】 图像负相的主机端和设备端函数。

```
////////////////////////////////////////////////////////////////////////
//设备端内核函数(kernel 函数)
__global__ void
InverseImg_kernel(
                    BYTE * pImgOut,
                    BYTE * pImgIn,
                    int nWidth,
                    int nHeight,
                    int nWidthStep)
{
    const int ix = blockIdx.x * blockDim.x + threadIdx.x;
    const int iy = blockIdx.y * blockDim.y + threadIdx.y;

    if( ix < nWidth && iy < nHeight )
    {
```

```
        pImgOut[ iy * nWidthStep + ix ] = 255 - pImgIn[ iy * nWidthStep + ix ];
    }
}

//主机端函数
extern "C"
double cudaInverseImg(
                        BYTE * pImgOut,
                        BYTE * pImgIn,
                        int nWidth,
                        int nHeight,
                        int nWidthStep,
                        int nChannels)
{
    BYTE * pImgCPU = NULL;
    BYTE * d_pImgInGPU = NULL;
    BYTE * d_pImgOutGPU = NULL;
    //准备空间
    pImgCPU = ( BYTE * )malloc( nWidthStep * nHeight * sizeof(BYTE) );
    cudaMalloc((void ** )&d_pImgOutGPU, nWidthStep * nHeight * sizeof(BYTE));
    cudaMalloc((void ** )&d_pImgInGPU, nWidthStep * nHeight * sizeof(BYTE));
    //数据初始化
    memset(pImgCPU, 0,nWidthStep * nHeight * sizeof(BYTE));
    cudaMemset(d_pImgOutGPU,0, nWidthStep * nHeight * sizeof(BYTE));
    cudaMemset(d_pImgInGPU,0, nWidthStep * nHeight * sizeof(BYTE));
    memcpy(pImgCPU,pImgIn,nWidthStep * nHeight * sizeof(BYTE));
    cudaMemcpy(d_pImgInGPU,pImgCPU,nWidthStep * nHeight * sizeof(BYTE),
        cudaMemcpyHostToDevice);
    //启动设备端 kernel 函数进行并行处理
    dim3 threads(16,16);
    dim3 grid(iDivUp(nWidth * nChannels + 15, threads.x), iDivUp(nHeight + 15, threads.y));
    InverseImg_kernel <<< grid,threads >>>( d_pImgOutGPU, d_pImgInGPU,
        nWidth * nChannels, nHeight, nWidthStep);
    //输出结果
    cudaMemcpy(pImgOut,d_pImgOutGPU,nWidthStep * nHeight * sizeof(BYTE),
        cudaMemcpyDeviceToHost);
    //释放空间
    if (pImgCPU!= NULL)
        free(pImgCPU);
    cudaFree(d_pImgOutGPU);
    cudaFree(d_pImgInGPU);
    return 0;
}
////////////////////////////////////////////////////////////////////////
```

例 3.6.1 中，主机端 cudaInverseImg 函数负责的主要工作及其常用函数如下：

（1）启动 CUDA，默认情况下，只要一个非运行时设备管理函数被调用，主机线程隐式就使用 0 号设备（也可以通过显式的调用 cudaSetDevice（）函数来启用其他 GPU 设备）；

（2）调用 malloc 函数为图像数据分配内存空间；

（3）调用 cudaMalloc 函数为要准备并行计算的图像数据分配显存空间；

（4）调用 memset 函数初始化分配完成的内存空间；

（5）调用 cudaMemset 函数初始化分配完成的显存空间；

（6）调用 memcpy 函数将所要处理的图像数据复制到准备好的内存中；

（7）调用 cudaMemcpy 函数将内存中的图像数据复制到显存中；

（8）配置内核所要用的线程设置，调用设备（device）端的内核（kernel）进行计算，将结果写到显存的对应区域；

（9）调用 cudaMemcpy 函数将显存中的处理结果复制到内存；

（10）使用 CPU 对数据进行其他处理；

（11）调用 free 函数释放内存空间，以及调用 cudaFree 函数释放显存空间；

（12）退出 CUDA。

设备端 InverseImg_kernel 函数负责的主要工作是：对显存中的数据启动内核（kernel）函数进行并行处理。

可以看出，主机端完成其中绝大多数的工作，而最为核心的并行计算工作交由设备端负责，这也正是 CUDA 编程模型（如图 3.6.2 所示）所提出的概念：一个完整的 CUDA 程序由一系列设备端 kernel 函数并行步骤和主机端的串行步骤共同组成。

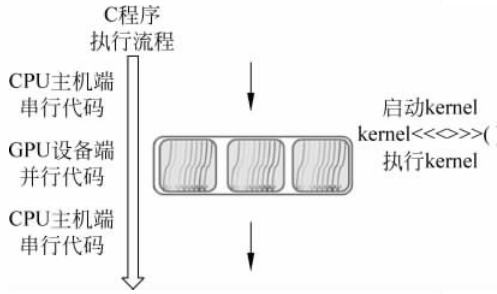

图 3.6.2　CUDA 编程模型

GPU 的操作对象不能超出显卡的范围，因此必须将等待并行处理的图像数据复制到显存中。也就是说，CPU 操作的图像数据存放在内存中，GPU 操作的图像数据习惯上存放在显存中。上述调用的函数为本书图像处理并行实现中常用的函数，读者也可以尝试使用别的函数来实现。

3.6.2　内核函数、CUDA 软件体系和 NVCC 编译器

CUDA 编码与常规的 C 语言编码最大的不同或者说扩展，是引入了允许程序员定义的内核（kernel）函数及其配套的配置执行语法。与普通的 C 语言函数只执行一次的方式

不同，在调用内核函数时，它将由 N 个不同的 CUDA 线程并行执行 N 次。

内核函数使用__global__声明符定义，并且需要配合一种新的<<<…>>>配置语法指定执行。在运算符<<<…>>>里面需要声明内核启动时的线程数的配置情况、共享内存、执行流等，例 3.6.1 中为每个像素负相处理分配的不同 CUDA 线程就是由<<<grid,threads>>>所确定的。内核函数只能在 GPU 中执行，并且需要放到.cu 文件中通过 CUDA 的专用规则编译。

例 3.6.2 简单地给出内核(kernel)函数定义模式及其配置执行过程。

【例 3.6.2】 内核(kernel)函数定义模式。

```
////////////////////////////////////////////////////////////////////////
//内核函数定义
__global__ void
kernelFun(ParaList parlist)
{
}

//配置执行
void main
{
    kernelFun <<< 1,1 >>>(parlist);
}
////////////////////////////////////////////////////////////////////////
```

在这个 kernelFun 核函数启动时，<<<1,1>>>指定了只有一个 Block，并且每个 Block 中只有一个线程。在使用时，可以根据实际情况灵活分配，以达到最大限度使用线程资源，从而提高处理速度。

上文用到的诸如 cudaMemcpy 在内的各种 CUDA 函数，其实是 CUDA 提供的库函数，CUDA 的软件体系包含三层(如图 3.6.3 所示)：CUDA 库函数(CUDA Library)、CUDA 运行时 API(CUDA runtime API)和 CUDA 驱动 API(CUDA driver API)。

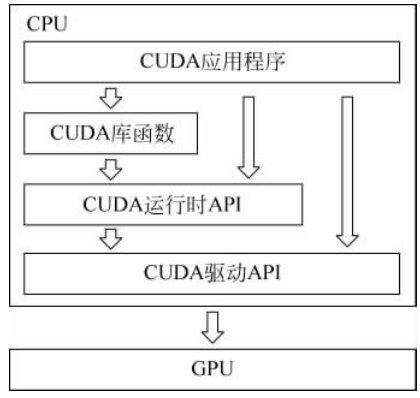

图 3.6.3 CUDA 软件体系

CUDA 库函数在 CUDA 运行 API(高级 API)和 CUDA 驱动 API(低级 API)基础上实现,通过直接调用 CUDA 库函数来对 GPU 进行操作,可以减少很多底层编码。

在对 CUDA 内核函数的编译过程中,将使用特定的编译器——NVCC 编译器。NVCC 编译器将进行 CUDA 语法的编译,而将非 CUDA 语法的编译过程交给 C 编译器,最后再将两部分结果联合编译。在编译的过程中,SDK 提供了编译规则,可以直接利用。

图 3.6.4 截取自 CUDA toolkit 安装文档中的 NVCC 的介绍文档,描绘了 CUDA 编译的整个流程,更详细的内容读者可以参考相应文档。

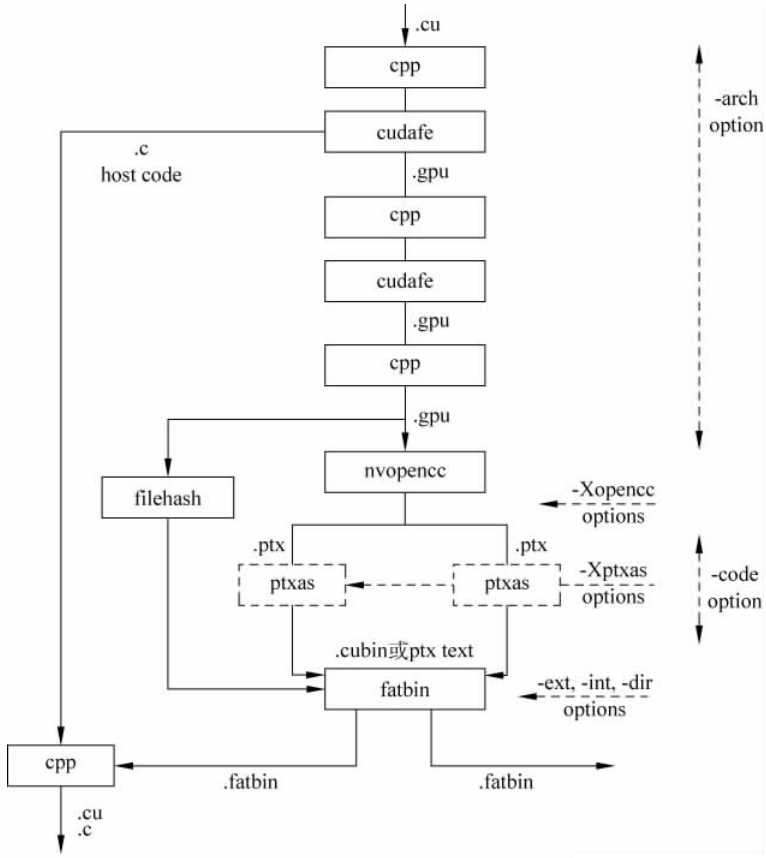

图 3.6.4　CUDA 编译流程

3.6.3　CUDA 线程模型的层次结构

在 GPU 编程中,总是提到多线程(Thread)的概念,一条线程处理一条数据,那么线程到底是什么？严格来说,这其实是显卡厂商提出的一种概念,为了方便用户理解 GPU 编程和 GPU 应用。在 CUDA 中,提出了 Grid(线程网格)、Block(线程块)和 Thread(线程)的层次结构,方便开发者为自己的算法从微观上分配线程来处理并行数据。需要区分的是,这种层次结构是一种编程模型,而非 GPU 硬件的实际结构。实际在 GPU 中执

行并行运算的，是 SM 流多处理器，这才是 GPU 的“核”，而将线程和 SM 联系到一起的，是 Warp(线程束)。通过线程模型，即使在不同架构、不同版本的 GPU 硬件上，对于图像处理应用来说也几乎不需要调整代码，从而保证了并行程序很好的通用性。

如图 3.6.5 所示，从线程模型的结构来看，内核(kernel)函数实际上有两个层次的并行：Grid 里面的 Block 并行，Block 里面的 Thread 并行。

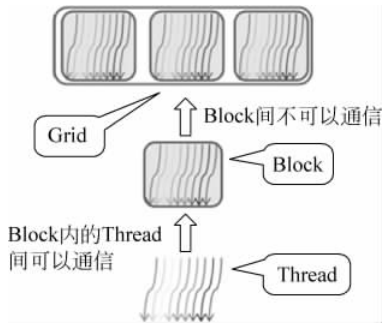

图 3.6.5　CUDA 线程结构示意图

Grid：一个 Grid 可以由很多个 Block 组成，每个 Block 在 Grid 里面是并行的，每个 Block 之间无法实现通信。

Block：一个 Block 可以由很多个 Thread 组成，每个 Thread 在 Block 里面是并行的，一个 Block 内的 Thread 之间可以通信。

上述线程的层次结构又主要由运算符<<<…>>>指定，执行内核的每个线程都会被分配一个独特的线程 ID，可通过内置的 threadIdx 变量在内核中访问此 ID。线程的索引及其线程 ID 有着直接的关系：对于一维块来说，两者是相同的；对于大小为(D_x, D_y)的二维块来说，索引为(x, y)的线程 ID 是$(x+yD_x)$；对于大小为(D_x, D_y, D_z)的三维块来说，索引为(x, y, z)的线程 ID 是$(x+yD_x+zD_xD_y)$。

运算符<<<…>>>对 kernel 函数完整的执行参数配置形式是<<<Dg, Db, Ns, S>>>，其中，Dg 用于定义整个 Grid 的维度和尺寸，Db 用于定义每个 Block 的维度和尺寸，Ns 可选，配置 shared memory 大小，单位为 Byte，S 为备选择项。

线程模型定义了 5 个内置变量。

gridDim：用于说明整个网格的维度与尺寸，与 host 端<<<…>>>中的 Dg 相对应；gridDim. x 和 gridDim. y 分别与 Dg. x 和 Dg. y 相等。

blockIdx：用于说明当前 Thread 所在的 Block 在整个 Grid 中的位置；blockIdx. x 取值范围是[0, gridDim. x−1]，blockIdx. y 取值范围是[0, gridDim. y−1]。

blockDim：用于说明每个 Block 的维度与尺寸，与 host 端<<<…>>>中的 Db 相对应；blockDim. x、blockDim. y 和 blockDim. z 分别与 Db. x、Db. y 和 Db. z 相等。

threadIdx：用于说明当前 Thread 在 Block 中的位置；threadIdx. x 取值范围是[0, blockDim. x−1]，threadIdx. y 取值范围是[0, blockDim. y−1]，threadIdx. z 取值范围是[0, blockDim. z−1]。

warpSize：用于引用 warp size。Warp 是线程的调度单位。

可以通过 Nvidia 公司的《CUDA 编程指南》中的一个例子来进一步理解<<<…>>>

这个符号的使用，这里将大小为 $N\times N$ 的矩阵 A 和矩阵 B 相加，并将结果存储在矩阵 C 中。例 3.6.3 是该方法的 C++串行代码，例 3.6.4 是《CUDA 编程指南》例子中所用的并行加法。

例 3.6.3 使用两个 for 循环遍历矩阵，实现对矩阵每个元素的相加操作。

【例 3.6.3】 矩阵元素相加的 CPU 串行代码。

```
//////////////////////////////////////////////////////////////////////////
int main()
{
    for ( int i = 0; i < N; i++){
        for( int j = 0; j < N; j++){
            C[j * N + i] = A[j * N + i] + [j * N + i];
        }
    }
}
//////////////////////////////////////////////////////////////////////////
```

例 3.6.4 调用内核（kernel）函数来进行矩阵相加并行实现。其中，dim3 是一个 CUDA 内置的结构体，dim3 的结构变量有 x、y、z，表示三维的维度，该结构体一般用来定义运算符<<<…>>>中的线程网格维度 gridDim 和线程块维度 blockDim。

【例 3.6.4】 矩阵元素相加的 GPU 并行代码。

```
//////////////////////////////////////////////////////////////////////////
__global__ void matAdd(float A[N][N], float B[N][N],float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}
int main()
{
    //Kernel invocation
    dim3 block (16, 16);
    dim3 grid ((N + block.x - 1) / block.x, (N + block.y - 1) / block.y);
    matAdd <<< grid, block >>>(A, B, C);
}
//////////////////////////////////////////////////////////////////////////
```

这段代码中，<<<grid，block>>>为内核函数分配了所要使用的线程方案，其中 threadIdx.x 为 0～15（15 即 block.x － 1），threadIdx.y 为 0～15（15 即 block.y － 1）；blockDim.x 为 16（Block 在 x 维度的宽度），blockDim.y 为 16（Block 在 y 维度的宽度）。同理，blockIdx.x 为 0 到 grid.x－1，blockIdx.y 为 0 到 grid.y－1。

可见，理想情况下，线程总数应该等于每个块的线程数乘以块的数量。但是，有的时候图像边长并不一定是 blockDim.x 和 blockDim.y 的倍数，比如 blockIdx.x 为 $N/16$

时，如果所得的数不是整数，那么需要多分配一个块来进行计算，即当 $N\%16\neq0$ 时，blockIdx.x = $N/16+1$。习惯上常用一个 iDivUp 函数来分配合适的网格数，形如 dim3 grid (iDivUp(N, block.x), iDivUp(N, block.y))。

```
////////////////////////////////////////////////////////////////////////
static int iDivUp(int a, int b)
{
    return (a % b != 0) ? (a / b + 1) : (a / b);
}
////////////////////////////////////////////////////////////////////////
```

在内核函数中，使用了条件 if (i < N && j < N)来约束进行运算的线程范围，跳过多余数据的计算。

对比例 3.6.3 和例 3.6.4，同样是定义的变量 i 和 j，例 3.6.3 中的 i 和 j 是循环控制变量，而例 3.6.4 的 matAdd 内核函数中，GPU 的 i 和 j 不再是循环控制变量，而成为了标识当前所运行线程的变量，也即 Thread ID (线程 ID)。在 GPU 编程中，可以直接用这个 ID 索引来对数组进行访问。

从另一个角度来看，经 GPU 存储和操作的数据被称为“流”(stream)，类似于 C 语言中的数组。内核函数可以理解为在流上操作的函数。在一系列输入流上调用一个内核函数意味着在流元素上实施了隐含的循环，即对每一个流元素调用内核，这样就实现了对流数据的并行操作。

3.6.4 GPU 组成结构及其与线程模型的关系

上一小节提到，线程模型是一种编程概念，实际在 GPU 硬件中执行并行运算的是 SM 流(Stream Multi-processor)多处理器。SM 是 GPU 的计算核心，以费米架构计算能力为 1.2 的 GPU 为例，如图 3.6.6 所示，每个 SM 中主要包含 8 个 SP(Stream Processor)标量流处理器(其他架构 SP 个数可能不同)、取指与发射单元、共享存储器、寄存器、缓冲存储器、专用单元 SPU 等。

CUDA 将计算任务“映射”为大量可以并行执行的线程，并由 GPU 硬件动态调度和执行。在这里，GPU 使用了一种称为 SIMT(Single Instruction Multiple Threads)的单指令多线程，有时也被称为 SPMD(Single Program Multiple Data)单程序多数据的技术，类似于 CPU 中的 SIMD 单指令多数据结构。这种 SIMT 结构允许 GPU 中的 SM 流多处理器取一次指令，然后反复操作不同的数据，这样可以节约很多显存带宽。实际上，内核函数是以 Block 为单位执行的，一个 Block 中线程往往需要共享数据(实现进程间通信)，必须在一个 SM 中发射执行。进入 SM 之前，Block 会被分割为更小的硬件调度执行单位 Warp(线程束)。费米架构中，Warp 的大小一般是 32(不同 GPU 中可能不同，早期的 GPU 也有 24 的)。也就是说，一个包含 32 个线程的 Warp 会在一个 SM 上执行，SM 中的 8 个 SP 将会以 SIMT 的方式将这条指令执行 4 遍。线程束 Warp 是一个由 SM 硬件决定的概念，不存在于线程模型中，但却是线程模型和 GPU 硬件不能不提的一个概念。线程束 Warp 只和 Thread ID(线程 ID)有关，而与 Block 的维度和每一维度的尺寸

没有关系。GPU 中的 SM 流多处理器如图 3.6.6 所示。

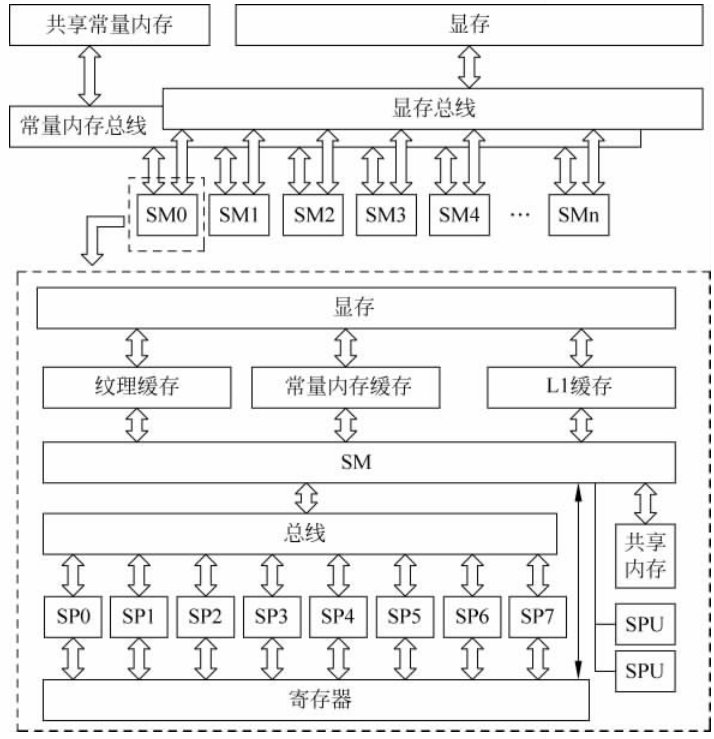

图 3.6.6　GPU 中的 SM 流多处理器

一个 SM 中同一时刻可以有多个活动 Block 在等待执行，但是一次只能运算一个 Block 里的一组 Warp，如果 Warp 中有线程的数据没有取到，则调度下一个 Warp 进行运算。

通过 Warp 这一纽带，线程模型和 GPU 硬件就被关联到了一起。虽然 CUDA 允许在编程过程中，大多数情况下只需关注图像数据、算法和线程分配即可，但了解 GPU 硬件的基本结构和运作，能帮助我们设计更加合理高效的并行程序。比如第 4 章中将要举例的自适应直方图均衡就用到了这里的知识。

3.6.5　SDK 和函数库

除了直接编写 GPU 内核函数，目前也有很多函数库可供调用。

Thrust：一个类似于 STL 的针对 CUDA 的 C++模板库；

NVPP：英伟达基本性能库；

cuBLAS：GPU 的基本线性代数函数库；

CUFFT：GPU 的快速傅里叶函数库；

cuSparse：GPU 的稀疏矩阵数据的线性代数和矩阵操作库；

Magma：一个用于数值计算和线性代数计算的函数库；

GPU AI：GPU 路径规划函数库；

CUDA Math Lib：GPU 标准数学函数。

函数库都经过相关领域的专家优化，性能可靠，执行高效，直接调用往往可以节约很多编程的时间和精力。所以掌握 CUDA 编程的基本方法之后，建议读者朋友花一些时间了解这些函数库的使用，让自己在并行程序设计上事半功倍。

除此以外，还有自己配置编译器来优化整体性能的方法，但这种方法主要针对海量数据的特殊计算需求，图像处理中一般不涉及此类问题。

以上就是对 CUDA 编程的简单介绍，目的是让读者能够明白书中所涉及代码的含义。包括存储器体系、访问优化、纹理和原子函数的使用在内，还有很多方法可以优化提高图像处理算法速度，这些方法需要基于对 CUDA 和算法深入的理解和运用。感兴趣的读者可以通过相关书籍和网站进一步学习 CUDA 并行计算的方法，系统深入地掌握 CUDA 的使用。