

**课 程 实 验 报 告**

**课程名称： 并行编程原理**

**专业班级： CS1503**

**学 号： U201514557**

**姓 名： 王国瑞**

**指导教师： 陆 枫**

**报告日期： 2018年7月15日**

**计算机科学与技术学院**

目录

1实验一 3

1.1 实验目的与要求 3

1.2 实验内容 3

1.3 实验结果 7

2实验二 11

2.1 实验目的与要求 11

2.2 算法描述 11

2.3 实验方案 12

2.4 实验结果与分析 13

3实验三 15

3.1 实验目的与要求 15

3.2 算法描述 15

3.3 实验方案 15

3.4 实验结果与分析 16

4 实验四 18

4.1 实验目的与要求 18

4.2 算法描述 18

4.3 实验方案 19

4.4 实验结果与分析 21

5 实验五 22

5.1 实验目的与要求 22

5.2 算法描述 22

5.3 实验方案 23

5.4 实验结果与分析 24

# 1实验一

## 1.1 实验目的与要求

实验目的：熟悉并行开发环境，掌握并行编程的基本原理和方法，了解Linux系统下pthread、OpenMP和OpenMPI等工具和框架的优化性能。

实验要求：使用最简单的任务划分方法——每个线程（进程）完成循环体中一次循环的工作，共有n个线程同时计算，从而实现对基本向量加法程序的优化。向量加法程序如下所示：

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

C[i] = A[i] + B[i];

## 1.2 实验内容

**1.2.1 使用pthread做向量加法**

算法描述：

i = 0, j = 0;

for i < n pthread\_create; //创建线程,并将i传递给线程函数plus\_pthread

for j < n pthread\_join; //等待线程j结束

定义三个全局变量vector\_a[]、vector\_b[]和vector\_result[]分别表示相加向量和结果向量,线程函数plus\_pthread做vector\_result[i] = vector\_a[i]+vector\_b[i]操作。

**1.2.2 使用OpenMP做向量加法**

使用特殊的编译引导语句，OpenMP会自动将for循环分解为多个线程，源程序修改成如下形式：

#pragma omp parallel for

for(i=0;i<5;++i)

vector\_result[i] = vector\_a[i] + vector\_b[i];

**1.2.3 使用OpenMPI做向量加法**

向量加法可以看成是一对多的通信机制，因此采用MPI\_Scatter散发机制实现进程间通信。算法描述如下：

MPI\_Init(&argc, &argv); //初始化，启动MPI环境

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rankID); //获取进程标识符

MPI\_Comm\_size(MPI\_COMM\_WORLD, &totalNumTasks);//获取进程数

MPI\_Scatter(sendBuf, sendCount, MPI\_FLOAT,

recvBuf, recvCount, MPI\_FLOAT, source, MPI\_COMM\_WORLD);

MPI\_Finalize();//结束MPI环境

MPI\_Scatter()函数接口中，sendBuf表示发送缓冲区，即我们定义的由两个n\*1维的向量所组成的n\*2维的矩阵数组，sendCount表示发送数据时的数据块的大小，MPI\_FLOAT表示发送的数据类型，recvBuf表示接收缓冲区，recvCount表示接收数据时的数据块大小，source表示根进程的进程号。在使用mpirun 时，–np参数大小应该为向量长度n。

**1.2.4 使用CUDA做向量加法**

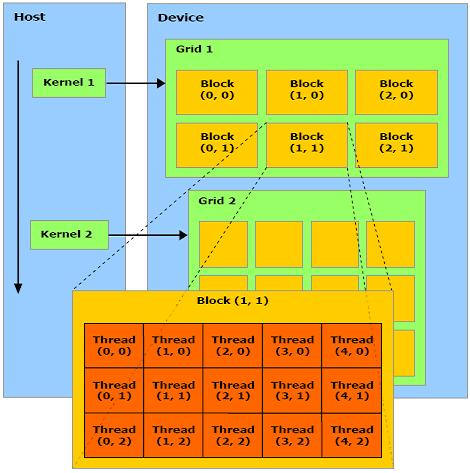


图1-1 CUDA内部机制

我们定义了四个128维的向量host\_a、host\_b、host\_c和host\_c2，分别表示主机端的A、B和C向量，host\_c2用于检验计算结果是否正确。

Kernel函数配置如下：

#define BLOCKSIZE 4

int gridsize = (int)ceil(sqrt(ceil(n / (BLOCKSIZE \* BLOCKSIZE))));

dim3 dimBlock(BLOCKSIZE, BLOCKSIZE, 1);

dim3 dimGrid(gridsize, gridsize, 1);

add<<<dimGrid, dimBlock>>>(device\_a, device\_b, device\_c, n);

初始化dimBlock为4\*4\*1的dim3类型，执行线程块的三个维度，这里第三维是1，即退化为4\*4的二维线程块。为了最大化并行，安排每一个线程负责一次向量加法，那么需要个线程块，即block的维数大小。设置线程网络grid，grid大小为，即grid的维度，Grid只能是二维以下，第三个维度设置默认忽略。设置中采用向上取整是为了保证至少有一个线程完成向量每对元素的相加，那么这样设置可能会导致线程数多于向量长度，因此在Kernel函数中需要让这些线程直接退出，避免数组下标越界。

将线程块号为blockIdx、线程号为threadIdx的线程映射到向量计算的数组下标：

块内地址：threadIdx.x \*blockDim.x + blockIdx.y

块内地址区间：[0, blockDim.x\*blockDim.y-1]

线程块地址：blockIdx.x\*gridDim.x+blockIdx.y

线程块地址区间：[0, gridDim.x\*gridDim.y-1]

因此线程号为threadIdx对应的数组下标为：

i = (blockIdx.x\*gridDim.x+blockIdx.y)\* blockDim.x\* blockDim.y

+( threadIdx.x \*blockDim.x + blockIdx.y)

因此，向量加法Kernel函数中，先计算出线程操作数组下标i，若i < n则计算，否则该线程直接退出。Kernel函数定义如下：

\_\_global\_\_ void add(const int \*a, const int \*b, int \*c, int n)

{

int i = (blockIdx.x \* gridDim.x + blockIdx.y) \* blockDim.x \* blockDim.y + threadIdx.x \* blockDim.x + threadIdx.y;

if (i < n) c[i] = a[i] + b[i];

}

程序流程图如图1-2所示，先将数据从主机内存拷贝到GPU内存设备上，然后主机调用向量加法Kernel函数让设备异步并行执行，由于CPU启动的Kernel函数是异步的，并不会阻塞等到GPU执行完kernel才执行后续的CPU部分，因此显示设置同步障来阻塞CPU程序。最后验证执行结果，统计执行时间。

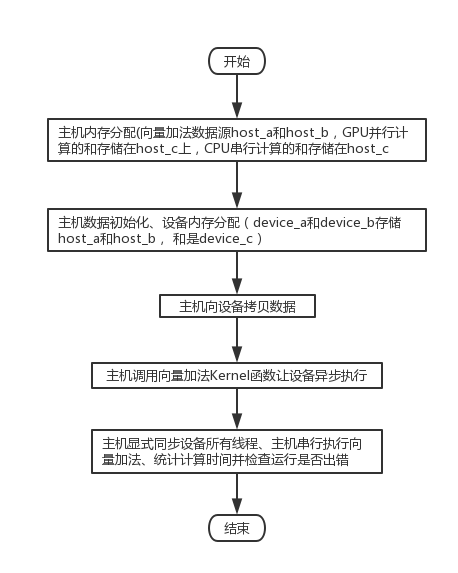


图1-2 CUDA做向量加法算法流程图

## 1.3 实验结果

**1.3.1 pthread**

编译：gcc Lab1\_1.c -o Lab1\_1 -lpthread

运行：./Lab1\_1

测试结果如图1-3所示。

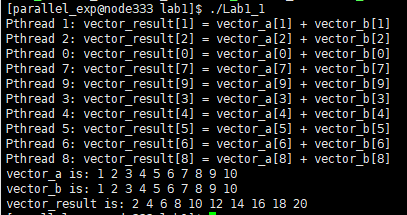


图1-3 pthread方法计算向量加法

由于将向量维度n设置为10，图中可以看到一共创建了10个进程，每个线程分别做了一次加法运算，由于线程并行，所以打印的结果随机，对比计算结果可知计算结果正确。

**1.3.2 OpenMP方法**

编译：gcc Lab1\_2.c -o Lab1\_2 –openmp

运行：./penmp

由于该实验是通过OpenMP特殊的编译引导语句自动将for循环分解为多个线程并行的，测试结果不是十分直观，如图1-4所示。因此我们把向量长度n增加为100000000，计算结果如图1-5所示。可以看到，在并行情况下速度有了一定的提升。

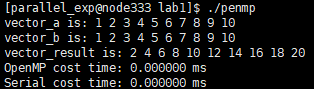


图1-4 OpenMP计算向量加法，n=10



图1-5 OpenMP计算向量加法，n=108

**1.3.3 OpenMPI方法**

编译：mpic Lab1\_3.c –o Lab1\_3

运行：mpirun –np 6 ./Lab1\_3

用一个n\*2维数组矩阵表示两个向量，通过MPI\_Scatter接口每次分发相同大小的数据块，每个数据块包含同行向量元素，每个进程执行一次加法运算。运行效果如图1-6所示。

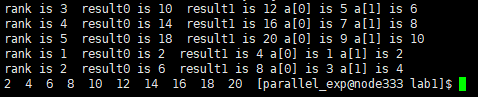


图1-6 OpenMPI方法计算向量加法

**1.3.4 CUDA方法**

编译：nvcc Lab1\_4.cu –o Lab1\_4

运行：./Lab1\_4

在图1-7中，我们在程序中设置向量长度n=128，块大小blocksize=4，验证计算结果正确，但是执行效率远不如CPU线性执行，而且测试到时二者效率几乎相同。图1-8为修改blocksize=16后的测试结果，我们看到随着数据量的增大，CUDA方法的计算效率逐渐增加，最终在时效率超过了CPU。当我们将blocksize设置为32时发现效率又降下来了，查阅资料才知道每个线程块（Block）一般最多可以创建512个并行线程，即blocksize<=16。

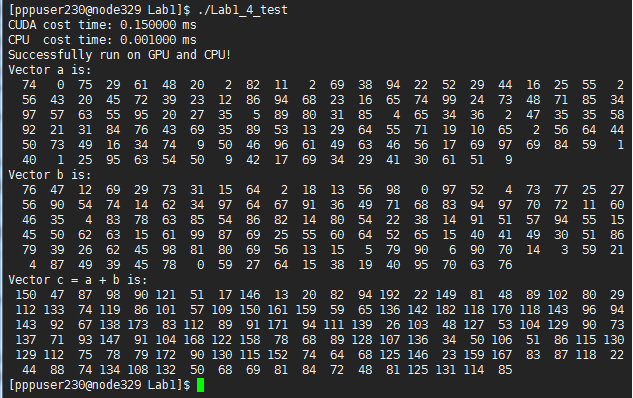


图1-7 CUDA方法计算向量加法，n=128，blocksize=4

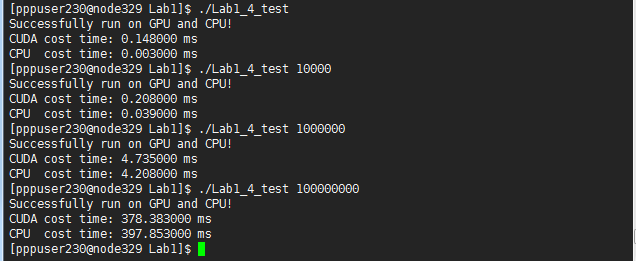


图1-8 CUDA方法计算向量加法，blocksize=16，n显示设置

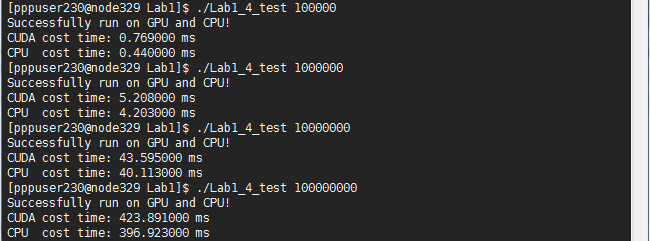


图1-9 CUDA方法计算向量加法，blocksize=32，n显示设置

# 2实验二

## 2.1 实验目的与要求

（1）掌握使用pthread的并行编程设计和性能优化的基本原理和方法；

（2）了解并行编程中数据分区和任务分解的基本方法；

（3）使用pthread实现图像卷积运算的并行算法；

（4）然后对程序执行结果进行简单的分析和总结。

## 2.2 算法描述

#### 2.2.1图像卷积相关介绍

在图像处理中，卷积操作指的是使用一个卷积核对图像中的每个像素进行一系列操作。

卷积核（算子）是用来做图像处理时的矩阵,图像处理时也称为掩膜，是与原图像做运算的参数。卷积核通常是一个四方形的网格结构（例如3\*3的矩阵或像素区域），该区域上每个方格都有一个权重值。

使用卷积进行计算时，需要将卷积核的中心放置在要计算的像素上，一次计算核中每个元素和其覆盖的图像像素值的乘积并求和，得到的结构就是该位置的新像素值。

#### 2.2.2 Pthread的并行编程方法

POSIX线程（POSIX threads），简称Pthreads，是线程的POSIX标准。该标准定义了创建和操纵线程的一整套API。在类Unix操作系统（Unix、Linux、Mac OS X等）中，都使用Pthreads作为操作系统的线程。

Pthread编程中，首先用 pthread\_create()函数创建指定个数的线程，并利用pthread\_join()函数等待多个线程的任务执行结束。值得注意的是，因为 pthread\_create()函数只能传入 void\*类型的参数，故将所有需要传入的参数打包成结构体，在并行时强制转化，将参数一一取出。

## 2.3 实验方案

#### 2.3.1 实验环境

开发环境：windows10+visual studio2017+opencv3.0.0

运行环境：Xshell远程连接到Linux服务器

#### 2.3.2 具体方案

对图像进行锐化操作，卷积核如图2-1。因为每一个像素点的计算仅仅取决于原图的像素点与卷积核矩阵，都与其他像素点的计算无关，而原图的像素点也不会改变。所以理论上每一个像素点都是可以并行计算，最后将计算结果合并起来就能形成锐化的图片。并且对图像中的第一行、最后一行、第一列、最后一列的像素点不做保持原值处理。

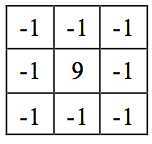


图2-1 卷积和

同时为了对并行粒度进行探究，证明由于硬件物理核心数为16，当并行度为16时效率最高的假设，在本次实验中增加对并行粒度研究部分，通过调整并行度，检测执行时间来体现并行粒度对效率的影响进行探索并得出结论。实验中分别采用1、2、4、8、16、32、64作为并行度进行研究，同时，当并行度为1时，可能近似认为程序串行执行。

#### 2.3.3核心代码

for(i = row\_start; (i < row\_end)&&(i < imagein.rows - 1) ;i++){

const uchar\*pre = imagein.ptr<uchar>(i-1);

const uchar\*cur = imagein.ptr<uchar>(i);

const uchar\*next = imagein.ptr<uchar>(i+1);

uchar\*outData = imageout.ptr<uchar>(i);

for (j = channel; j < (columnt-1)\*channel; j++) {

outData[j] = saturate\_cast<uchar>( cur[j]\*9 - pre[j- channel]- pre[j]- pre[j + channel]-cur[j- channel]-cur[j + channel] -next[j- channel]-next[j]-next[j + channel]);

}

}

## 2.4 实验结果与分析

运行编译指令g++ Lab2.cpp -lpthread -o Lab2 `pkg-config --libs --cflags opencv`，生成可执行文件Lab2。

输入指令./Lab2运行，输入线程数，进行卷积操作，输出图像卷积所的时间。首先输入线程数为16，如图2-2所示。当线程数为16时，运行时间为1.08983ms。



图2-2 线程数为16时的运算时间

当输入线程数为1时可以认为是串行运算，运行时间如图2-3所示。可以看到，串行运算的时间为2.83472ms，相比线程数为16的运算时间，慢了将近3倍的时间。



图2-3线程数为1时的运算时间

进行卷积运算的原图如图2-4所示，程序运行生成的图片如图2-5所示。对比发现，相较于原图，生成的图片有了明显的锐化，实验结果正确。



图2-4 进行卷积运算的原图



图2-5 程序输出的图片

# 3实验三

## 3.1 实验目的与要求

（1）掌握使用OpenMP进行并行编程设计和性能优化的基本原理和方法

（2）使用OpenMP实现图像卷积运算的并行算法

（3）对程序执行结果进行简单的分析和总结；

（4）将其与实验二的结果进行比较。

## 3.2 算法描述

#### 3.2.1 OpenMP相关介绍

OpenMP是一种用于共享内存并行系统的多线程程序设计方案，支持的编程语言包括C、C++和Fortran。OpenMP提供了对并行算法的高层抽象描述，特别适合在多核CPU机器上的并行程序设计。编译器根据程序中添加的pragma指令，自动将程序并行处理，使用OpenMP降低了并行编程的难度和复杂度。当编译器不支持OpenMP时，程序会退化成普通（串行）程序。程序中已有的OpenMP指令不会影响程序的正常编译运行。

#### 3.2.2 并行算法

并行算法与实验二相同，只需要在程序的for循环前加上特殊的编译引导语句，OpenMP会自动将for循环分解为多个线程。

## 3.3 实验方案

#### 3.3.1 实验环境

开发环境：windows10+visual studio2017+opencv3.0.0

运行环境：Xshell远程连接到Linux服务器

#### 3.3.2 具体方案

由于OpenMP并行框架的特殊性，不需在代码上进行很大修改即可完成并行化，只需要在程序的for循环前加上特殊的编译引导语句即可。

#### 3.3.3核心代码

#pragma omp parallel for

for (int i = 1; i < row - 1; i++) {

const uchar\*pre = rp.ptr<uchar>(i - 1);

const uchar\*cur = rp.ptr<uchar>(i);

const uchar\*next = rp.ptr<uchar>(i + 1);

uchar\*outData = imgout.ptr<uchar>(i);

for (int j = channel; j < (column - 1)\*channel; j++) {

outData[j] = saturate\_cast<uchar>(

pre[j - channel] \* kernel[0][0] + pre[j] \* kernel[0][1] + pre[j + channel] \* kernel[0][2]

+ cur[j - channel] \* kernel[1][0] + cur[j] \* kernel[1][1] + cur[j + channel] \* kernel[1][2]

+ next[j - channel] \* kernel[2][0] + next[j] \* kernel[2][1] + next[j + channel] \* kernel[2][2]);

}

}

## 3.4 实验结果与分析

运行编译指令g++ Lab3.cpp -fopenmp -o Lab3 `pkg-config --libs --cflags opencv`，生成可执行文件Lab3。

输入指令./Lab3运行程序，结果如图3-1所示。由图3-1可知，OpenMP进行卷积运算的运行时间为0.27926ms。



图3-1 OpenMP进行卷积操作的运行时间

对比线程数为16的pthread方法进行卷积运算的运算时间，发现快了将近4倍。

进行卷积运算的原图如图3-2所示，程序运行生成的图片如图3-3所示。对比发现，相较于原图，生成的图片有了明显的锐化，实验结果正确。



图3-2进行卷积运算的原图



图3-3 OpenMP方法进行卷积运算输出的图片

# 4 实验四

## 4.1 实验目的与要求

（1）掌握使用MPI进行并行编程设计和性能优化的基本原理和方法

（2）使用MPI实现图像卷积运算的并行算法

（3）对程序执行结果进行简单的分析和总结；

（4）将其与实验二与实验三的结果进行比较。

## 4.2 算法描述

#### 4.2.1 MPI相关介绍

MPI是一个跨语言的通讯协议，用于编写并行计算机，支持点对点和广播。MPI是一个信息传递应用程序接口，包括协议和和语义说明，他们指明其如何在各种实现中发挥其特性。MPI的目标是高性能，大规模性，和可移植性。MPI在今天仍为高性能计算的主要模型。

主要的MPI-1模型不包括共享内存概念，MPI-2只有有限的分布共享内存概念。 但是MPI程序经常在共享内存的机器上运行。在MPI模型周边设计程序比在NUMA架构下设计要好因为MPI鼓励内存本地化。

尽管MPI属于OSI参考模型的第五层或者更高，他的实现可能通过传输层的sockets和Transmission Control Protocol (TCP)覆盖大部分的层。大部分的MPI实现由一些指定惯例集（API）组成，可由C,C++,Fortran,或者有此类库的语言比如C#, Java or Python直接调用。MPI优于老式信息传递库是因为他的可移植性和速度。。

#### 4.2.2并行算法

整体算法思路与前几次实验并无太大差异，本次实验使用MPI并行框架进行编程，使用进程为基本通信单位，保证了可扩展性，但相较于前两种框架，MPI耗费的资源更多，且配置和启动框架的时间更长，在集群环境中优势更加明显。

## 4.3 实验方案

#### 4.3.1 实验环境

开发环境：windows10+visual studio2017+opencv3.0.0

运行环境：Xshell远程连接到Linux服务器

#### 4.3.2 方案细节

本实验实现算法与前几次实验基本相同，需要注意的地方主要是MPI接口函数的使用。主要核心接口有6个，可以简单的分为三类：

1. 开始和结束MPI的接口：MPI\_Init、 MPI\_Finalize
2. 获取进程状态的接口：MPI\_Comm\_rank、MPI\_Comm\_size
3. 传输数据的接口：MPI\_Send、MPI\_Recv

每个接口的具体定义和使用方法如下：

1. MPI\_Init(&argc, &argv) ：

初始化MPI执行环境，建立多个MPI进程之间的联系，为后续通信做准备。

1. MPI\_Comm\_rank(communicator, &myid) ：

用来标识各个MPI进程的，给出调用该函数的进程的进程号,返回整型的错误值。两个参数：MPI\_Comm类型的通信域，标识参与计算的MPI进程组； &rank返回调用进程中的标识号。

1. MPI\_Comm\_size(communicator, &numprocs) ：

用来标识相应进程组中有多少个进程。

1. MPI\_Finalize() ：

结束MPI执行环境。

1. MPI\_Send(buf,counter,datatype,dest,tag,comm) ：

* buf：发送缓冲区的起始地址，可以是数组或结构指针；
* count：非负整数，发送的数据个数；
* datatype：发送数据的数据类型；
* dest：整型，目的的进程号；
* tag：整型，消息标志；comm：MPI进程组所在的通信域

含义:向通信域中的dest进程发送数据，数据存放在buf中，类型是datatype，个数是count，这个消息的标志是tag，用以和本进程向同一目的进程发送的其它消息区别开来。

1. MPI\_Recv(buf,count,datatype,source,tag,comm,status) ：

* source:整型，接收数据的来源，即发送数据进程的进程号；
* status：MPI\_Status结构指针，返回状态信息。

#### 4.3.3 核心代码

if(id != 0 && id != num-1)//第一个进程用于收集数据

{

row\_start = (id-1)\*brow + 1;

row\_end = row\_start + brow;

for(i = row\_start; i < row\_end; i++){//获取数据源阵列

const uchar\* pre = imgin.ptr<uchar>(i-1);

const uchar\* cur = imgin.ptr<uchar>(i);

const uchar\* next = imgin.ptr<uchar>(i+1);

uchar\* outdata = imgout.ptr<uchar>(i);

for(j = channel; j < (column-1)\*channel; j++){//像素点锐化

outdata[j] = saturate\_cast<uchar>( cur[j]\*9 - pre[j-channel] - pre[j] - pre[j+channel] - cur[j-channel] - cur[j+channel] - next[j-channel] - next[j] - next[j+channel]);

}

}

uchar\* buf\_send = imgout.ptr<uchar>(row\_start);

MPI\_Send(buf\_send, brow\*column\*channel, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD);

}

else

{

time1 = get\_time();

for(i = 1; i < num-1; i++)

{

uchar\* buf\_recv = imgout.ptr<uchar>((i-1)\*brow + 1);

MPI\_Recv(buf\_recv, brow\*column\*channel, MPI\_CHAR, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

uchar\* buf\_recv = imgout.ptr<uchar>((num-2)\*brow + 1);

row\_start = (num-2)\*brow + 1;

row\_end = row - 1;

MPI\_Recv(buf\_recv, (row\_end-row\_start)\*column\*channel, MPI\_CHAR, num-1, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

time2 = get\_time();

printf("Image convolution time in MPI: %lf\n",(double)(time2-time1));

IplImage tmp = IplImage(imgout);

CvArr\* arr = (CvArr\*)&tmp;

cvSaveImage("output.jpg", arr);

}

## 4.4 实验结果与分析

运行编译指令mpic++ -o Lab4 Lab4.cpp `pkg -config --libs --cflags opencv`，生成可执行文件Lab4。

输入指令mpirun -np 16 ./mpi运行文件，得到输出图像，其中16为线程数。当线程数为16时，利用MPI进行卷积运算的运行时间1.6238ms。

根据执行结果可以发现，通过多次测试当开启的进程数为6的时候，并行执行的时间最短，仅约为0.2ms。

将输出的图片下载到本地后查看，原图如图4-1所示，输出图片如图4-2所示，可以看到有了明显的锐化，实验结果正确。



图4.1 进行卷积运算的原图



图4.4 MPI方法进行卷积运算的输出图片

# 5 实验五

## 5.1 实验目的与要求

（1）掌握使用CUDA进行并行编程设计和性能优化的基本原理和方法

（2）使用CUDA实现图像卷积运算的并行算法

（3）对程序执行结果进行简单的分析和总结；

（4）将其与实验二、实验三与实验四的结果进行比较。

## 5.2 算法描述

CUDA是显卡厂商 NVIDIA 推出的运算平台。CUDA 是一种由 NVIDIA 推出的通用并行计算架构，该架构使 GPU 能够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。开发人员现在可以使用C语言来为 CUDA 架构编写程序。

GPU 协处理器的运算流程:

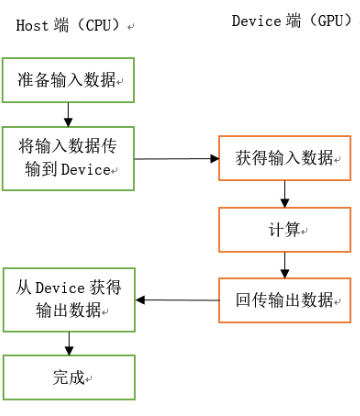


图5-1 GPU 协处理器的处理过程

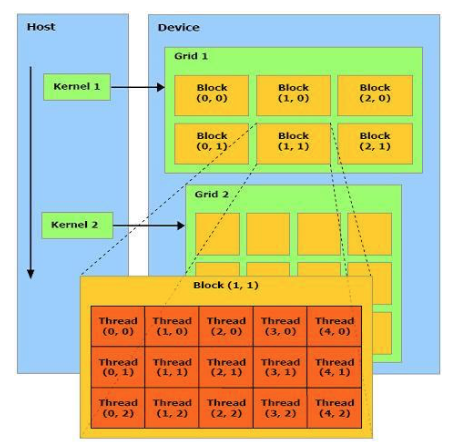


图5-2 cuda 执行过程

CUDA 在执行的时候是让 host 里面的一个一个的 kernel 按照线程网格(Grid)的概念在显卡硬件(GPU)上执行。 每一个线程网格又可以包含多个线程块(block)，每一个线程块中又可以包含多个线程(thread)。

本次实验使用CUDA进行GPU运算以达到并行优化目的，算法思想与前几次实验无异，都是将很大的任务量分发至各个处理机上进行处理并返回结果后进行聚合得到最终结果。但是在CUDA中，使用GPU作为运算工具，相较于CPU而言，其最大的优势就是拥有远多于CPU的核心数，使得GPU能够开启大量的线程进行运算而无需担心线程数过多导致并行度上升时性能反而下降的现象。

对于简单算法而言，和前几次的算法一致，通过在主机申请一片内存区域，各个线程拿到后通过自己的线程ID计算出自己所需要的任务部分并进行计算并将该部分任务聚合放入内存区域中属于自己的那部分之中。在所有线程计算完毕后主机将该部分内存拷贝回主机并进行聚合得到最终结果。

## 5.3 实验方案

#### 5.3.1 实验环境

开发环境：windows10+visual studio2017+opencv3.0.0

运行环境：Xshell远程连接到Linux服务器

#### 5.3.2 方案细节

利用实验二设计的卷积计算的算法，结合 CUDA 框架来实现计算的并行执 行。具体实现使用256个Block，每个Block内有512个Threads进行CUDA运算，采用上述算法分别进行简单算法测试及优化算法测试，将测试结果分别进行对于及同其他CPU上的并行算法效率进行对比，得出结论。

#### 5.3.3 核心代码

\_\_global\_\_ void parallel\_cuda(uchar \*dev\_src,uchar \*dev\_dst,int row,int col,int NUM,int channel)

{

int thread\_id1 = (blockIdx.x \* gridDim.x + blockIdx.y) \* blockDim.x \* blockDim.y + threadIdx.x \* blockDim.x + threadIdx.y;

int block\_row = row / NUM + 1;

int i, j, row\_start, row\_end;

if(thread\_id1 < NUM){

row\_start = thread\_id1 \* block\_row + 1;

row\_end = row\_start + block\_row;

for(i = row\_start; (i < row\_end)&&(i<row-1) ;i++){

const uchar \*pre = dev\_src+(i-1)\*channel\*col;

const uchar \*cur = dev\_src+(i)\*channel\*col;

const uchar \*next = dev\_src+(i+1)\*channel\*col;

uchar \*outData = dev\_dst+(i)\*channel\*col;

for (j = channel; j < (col-1)\*channel; j++) {

int tmp = ( cur[j]\*9 - pre[j- channel]- pre[j]- pre[j + channel]-cur[j- channel]-cur[j + channel] -next[j- channel]-next[j]-next[j + channel]);

outData[j] = tmp;

}

}

}

}

## 5.4 实验结果与分析

运行编译指令nvcc -o Lab5 Lab5.cu `pkg-config --libs --cflags opencv`，生成可执行文件Lab5，如图5-3所示。由图5-3可知，利用CUDA进行图像卷积运算的运行时间为0.703ms。



图5-3 利用CUDA进行卷积运算的运行时间

进行卷积运算的原图如图5-4所示，输出图片如图5-5所示，可以看到有了明显的锐化，实验结果正确。



图5-4 进行卷积运算的原图



图5-5 CUDA方法进行卷积运算的输出图片

在利用 CUDA 并行化后的执行时间大大减少，说明利用 CUDA 编程实现计算的并行化可以大大提高计算效率，性能较CPU上并行算法有很大提升。而且和 Pthread、OpenMP 以及 MPI 相比， CUDA 的执行时间要快于线程数为16时的Pthread ，但是慢于OpenMP和MPI，所以 CUDA 的执行效率要高于 Pthread，低于OpenMP和MPI。