

華中科技大學

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	10
2.1 实验目的与要求.....	10
2.2 算法描述.....	10
2.3 实验方案.....	11
2.4 实验结果与分析.....	12
3 实验三	14
3.1 实验目的与要求.....	14
3.2 算法描述.....	14
3.3 实验方案.....	14
3.4 实验结果与分析.....	15
4 实验四	17
4.1 实验目的与要求.....	17
4.2 算法描述.....	17
4.3 实验方案.....	18
4.4 实验结果与分析.....	20
5 实验五	21
5.1 实验目的与要求.....	21
5.2 算法描述.....	21
5.3 实验方案.....	22
5.4 实验结果与分析.....	23

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 $n \times 1$ 维的向量所组成的 $n \times 2$ 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

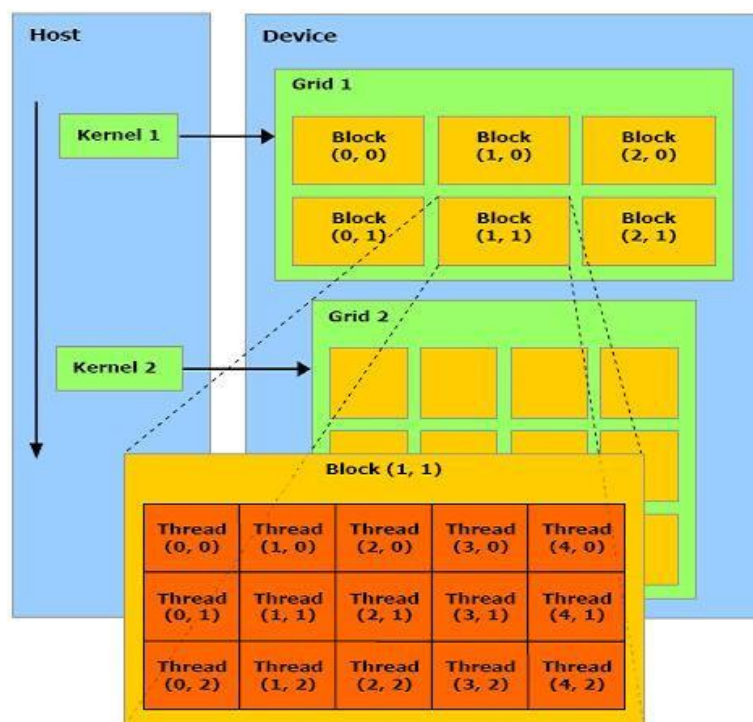


图 1-1 CUDA 内部机制

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 $\text{blockDim} = \left\lceil \frac{n}{\text{BLOCKSIZE} * \text{BLOCKSIZE}} \right\rceil$ 个线程块，即 block 的维

数大小。设置线程网络 grid，grid 大小为 $\text{gridDim} = \left\lceil \sqrt{\left\lceil \frac{n}{\text{BLOCKSIZE} * \text{BLOCKSIZE}} \right\rceil} \right\rceil$ ，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 + threadIdx.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。

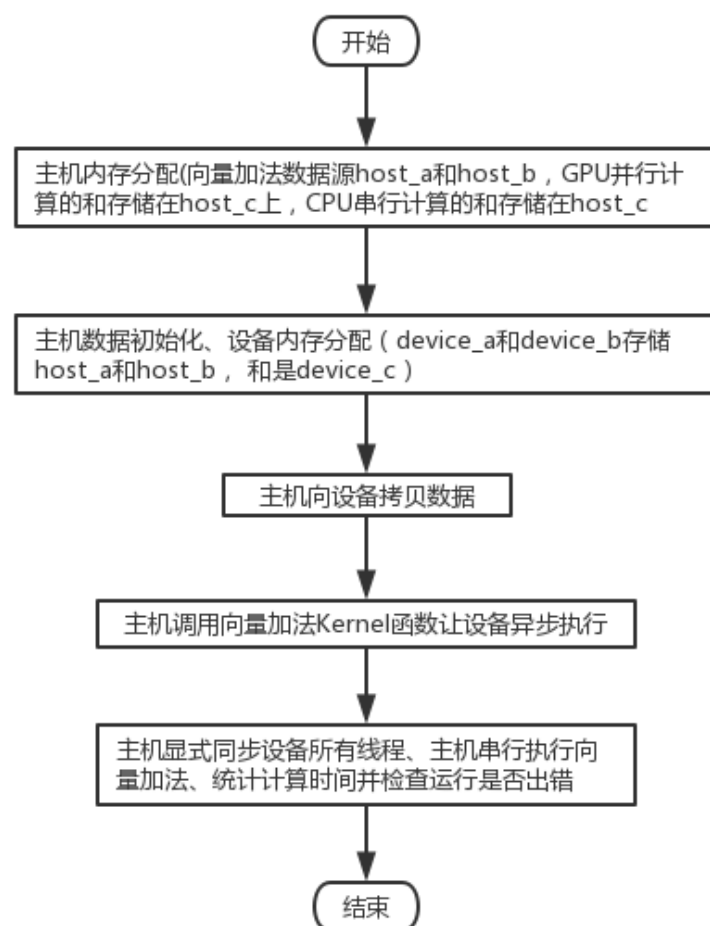


图 1-2 CUDA 做向量加法算法流程图

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。

```
[parallel_exp@node333 lab1]$ ./Lab1_1
Pthread 1: vector_result[1] = vector_a[1] + vector_b[1]
Pthread 2: vector_result[2] = vector_a[2] + vector_b[2]
Pthread 0: vector_result[0] = vector_a[0] + vector_b[0]
Pthread 7: vector_result[7] = vector_a[7] + vector_b[7]
Pthread 9: vector_result[9] = vector_a[9] + vector_b[9]
Pthread 3: vector_result[3] = vector_a[3] + vector_b[3]
Pthread 4: vector_result[4] = vector_a[4] + vector_b[4]
Pthread 5: vector_result[5] = vector_a[5] + vector_b[5]
Pthread 6: vector_result[6] = vector_a[6] + vector_b[6]
Pthread 8: vector_result[8] = vector_a[8] + vector_b[8]
vector_a is: 1 2 3 4 5 6 7 8 9 10
vector_b is: 1 2 3 4 5 6 7 8 9 10
vector_result is: 2 4 6 8 10 12 14 16 18 20
```

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。

```
[parallel_exp@node333 lab1]$ ./penmp
vector_a is: 1 2 3 4 5 6 7 8 9 10
vector_b is: 1 2 3 4 5 6 7 8 9 10
vector_result is: 2 4 6 8 10 12 14 16 18 20
OpenMP cost time: 0.000000 ms
Serial cost time: 0.000000 ms
```

图 1-4 OpenMP 计算向量加法， $n=10$

```
[parallel_exp@node333 lab1]$ gcc Lab1_2_1.c -o Lab1_2_1 -openmp
[parallel_exp@node333 lab1]$ ./penmp
OpenMP cost time: 0.000000 ms
Serial cost time: 1.000000 ms
```

图 1-5 OpenMP 计算向量加法， $n=10^8$

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 $n \times 2$ 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。

```
rank is 3 result0 is 10 result1 is 12 a[0] is 5 a[1] is 6
rank is 4 result0 is 14 result1 is 16 a[0] is 7 a[1] is 8
rank is 5 result0 is 18 result1 is 20 a[0] is 9 a[1] is 10
rank is 1 result0 is 2 result1 is 4 a[0] is 1 a[1] is 2
rank is 2 result0 is 6 result1 is 8 a[0] is 3 a[1] is 4
2 4 6 8 10 12 14 16 18 20 [parallel_exp@node333 lab1]$
```

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 $n = 10^8$ 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 $n = 10^8$ 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 ≤ 512`。


```
[pppuser230@node329 Lab1]$ ./Lab1_4_test
CUDA cost time: 0.150000 ms
CPU cost time: 0.001000 ms
Successfully run on GPU and CPU!
Vector a is:
 74  0 75 29 61 48 20  2 82 11  2 69 38 94 22 52 29 44 16 25 55  2
 56 43 20 45 72 39 23 12 86 94 68 23 16 65 74 99 24 73 48 71 85 34
 97 57 63 55 95 20 27 35  5 89 80 31 85  4 65 34 36  2 47 35 35 58
 92 21 31 84 76 43 69 35 89 53 13 29 64 55 71 19 10 65  2 56 64 44
 50 73 49 16 34 74  9 50 46 96 61 49 63 46 56 17 69 97 69 84 59  1
 40  1 25 95 63 54 50  9 42 17 69 34 29 41 30 61 51  9
Vector b is:
 76 47 12 69 29 73 31 15 64  2 18 13 56 98  0 97 52  4 73 77 25 27
 56 90 54 74 14 62 34 97 64 67 91 36 49 71 68 83 94 97 70 72 11 60
 46 35  4 83 78 63 85 54 86 82 14 80 54 22 38 14 91 51 57 94 55 15
 45 50 62 63 15 61 99 87 69 25 55 60 64 52 65 15 40 41 49 30 51 86
 79 39 26 62 45 98 81 80 69 56 13 15  5 79 90  6 90 70 14  3 59 21
  4 87 49 39 45 78  0 59 27 64 15 38 19 40 95 70 63 76
Vector c = a + b is:
150 47 87 98 90 121 51 17 146 13 20 82 94 192 22 149 81 48 89 102 80 29
112 133 74 119 86 101 57 109 150 161 159 59 65 136 142 182 118 170 118 143 96 94
143 92 67 138 173 83 112 89 91 171 94 111 139 26 103 48 127 53 104 129 90 73
137 71 93 147 91 104 168 122 158 78 68 89 128 107 136 34 50 106 51 86 115 130
129 112 75 78 79 172 90 130 115 152 74 64 68 125 146 23 159 167 83 87 118 22
 44 88 74 134 108 132 50 68 69 81 84 72 48 81 125 131 114 85
[pppuser230@node329 Lab1]$
```

图 1-7 CUDA 方法计算向量加法，n=128，blocksize=4

```
[pppuser230@node329 Lab1]$ ./Lab1_4_test
Successfully run on GPU and CPU!
CUDA cost time: 0.148000 ms
CPU cost time: 0.003000 ms
[pppuser230@node329 Lab1]$ ./Lab1_4_test 10000
Successfully run on GPU and CPU!
CUDA cost time: 0.208000 ms
CPU cost time: 0.039000 ms
[pppuser230@node329 Lab1]$ ./Lab1_4_test 1000000
Successfully run on GPU and CPU!
CUDA cost time: 4.735000 ms
CPU cost time: 4.208000 ms
[pppuser230@node329 Lab1]$ ./Lab1_4_test 100000000
Successfully run on GPU and CPU!
CUDA cost time: 378.383000 ms
CPU cost time: 397.853000 ms
[pppuser230@node329 Lab1]$
```

图 1-8 CUDA 方法计算向量加法，blocksize=16，n 显示设置

```
[pppuser230@node329 Lab1]$ ./Lab1_4_test 100000
Successfully run on GPU and CPU!
CUDA cost time: 0.769000 ms
CPU cost time: 0.440000 ms
[pppuser230@node329 Lab1]$ ./Lab1_4_test 1000000
Successfully run on GPU and CPU!
CUDA cost time: 5.208000 ms
CPU cost time: 4.203000 ms
[pppuser230@node329 Lab1]$ ./Lab1_4_test 100000000
Successfully run on GPU and CPU!
CUDA cost time: 43.595000 ms
CPU cost time: 40.113000 ms
[pppuser230@node329 Lab1]$ ./Lab1_4_test 1000000000
Successfully run on GPU and CPU!
CUDA cost time: 423.891000 ms
CPU cost time: 396.923000 ms
```

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与要计算的像素上，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。

-1	-1	-1
-1	9	-1
-1	-1	-1

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 < (column-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。

```

[parallel_exp@node333 lab2]$ ./Lab2
Input thread num:16
Image convolution time in pthread:0.00108983

```

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。

```

[parallel_exp@node333 lab2]$ ./Lab2
Input thread num:1
Image convolution time in pthread:0.00283472

```

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。

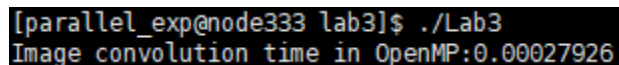
3.3.3 核心代码

```
#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。



```
[parallel_exp@node333 lab3]$ ./Lab3
Image convolution time in OpenMP:0.00027926
```

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。

- (2) 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。

- (3) MPI_Comm_size(communicator, &numprocs) :

用来标识相应进程组中有多少个进程。

- (4) MPI_Finalize() :

结束 MPI 执行环境。

- (5) 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。

(6) 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+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:

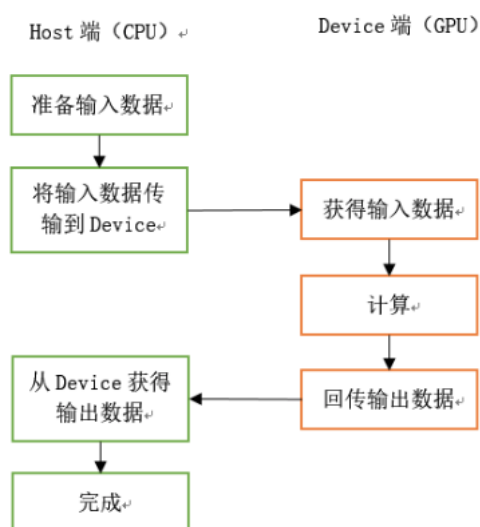


图 5-1 GPU 协处理器的处理过程

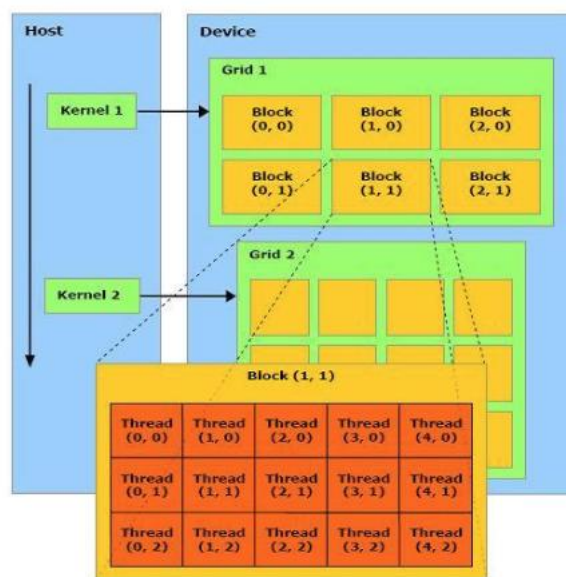


图 5-2 cuda 执行过程

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。

```
[parallel_exp@node333 lab5]$ nvcc -o Lab5 Lab5.cu `pkg-config --libs --cflags opencv`  
[parallel_exp@node333 lab5]$ ./Lab5  
Image convolution time in cuda: 0.000703
```

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。