

# 课 程 实 验 报 告

# 课程名称： 并行编程原理与实践

院 系 计算机科学与技术

姓 名 周俊辰

专业班级 CS1402

学 号 U201414608

指导教师 金海

报告日期

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：对于每个工具，挑选一种简单的可以并行化的算法，配置框架、编写程序并进行运行演示。

例如，可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std::thread做向量加法**

由于我在Windows平台下进行实验，在这里选择C++11标准库提供的std::thread进行实验。std::thread可以提供和pthread完全等价的功能。

为了演示std::thread的功能，我们实现一个简单的“并行加法”程序：每个进程执行一个特定的加法任务，结果在主线程中进行汇总累加再输出。

算法描述：

data = { {1, 2, 3, 4, 5}, {2, 3, 4, 5, 6}, … }

sum = 0

for each array in data:

spawn worker thread add(array)

for each worker thread:

join thread and get result

sum = sum + result

print sum

使用std::thread创建线程。为了将数据从worker线程传递到main线程，使用了<future>头文件提供的std::promise。

为了更好的展示效果，在每个worker进程执行结束后，会单独打印其计算结果。这里为了让不同线程打印的内容不相互干扰，设置了一个静态的互斥锁变量std::mutex m，并在打印时占用该锁。

**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**

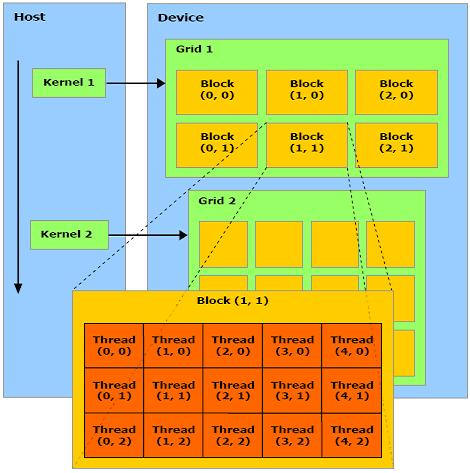


图1-1 CUDA内部机制

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。

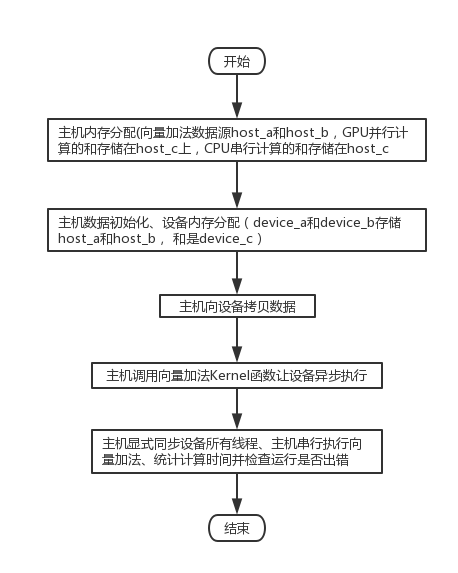


图1-2 CUDA做向量加法算法流程图

1.3 实验结果

**1.3.1 pthread**

编译运行：

* 在lab1/exp1-thread/文件夹中执行cmake .创建工程
* 使用平台相关的编译方法进行编译运行

测试结果如图1-3所示。

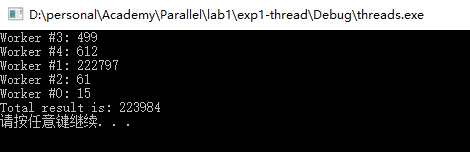


图1-3 pthread方法示例

设置了5个数组分别执行加法工作。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：./Lab1\_2

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1-3所示。因此我们把向量长度n增加为100000，计算结果如图1-4所示。虽然已经增大了n的级数，但是多次运行的结果可以发现二者执行速度差别很小，若果仅仅只做一次简单的for循环，OpenMP的加速情况并不是特别明显。

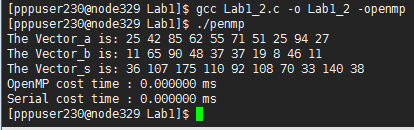


图1-4 OpenMP计算向量加法样例，n=10

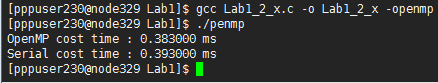


图1-5 OpenMP计算向量加法样例，n=

**1.3.3 OpenMPI方法**

编译：mpic Lab1\_3.c –o Lab1\_3

运行：mpirun –np 4 ./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1-5所示。

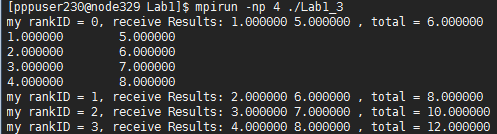


图1-6 OpenMPI方法计算向量加法

**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。

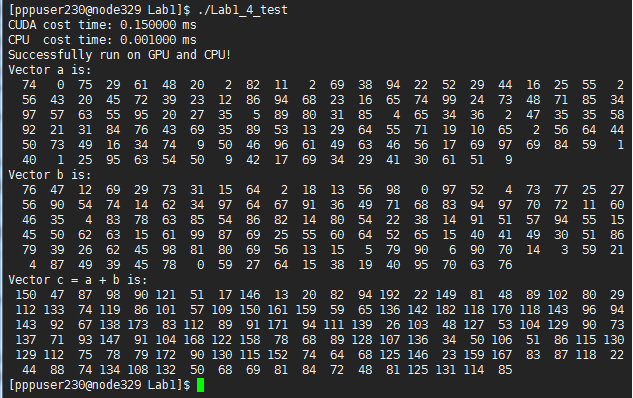


图1-7 CUDA方法计算向量加法，n=128，blocksize=4

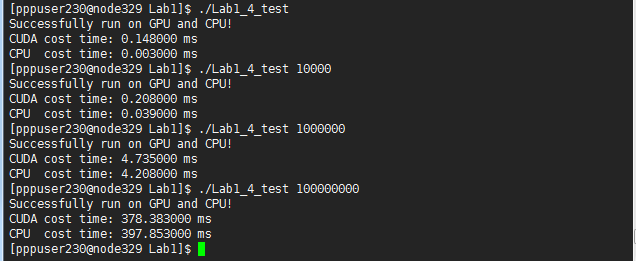


图1-8 CUDA方法计算向量加法，blocksize=16，n显示设置

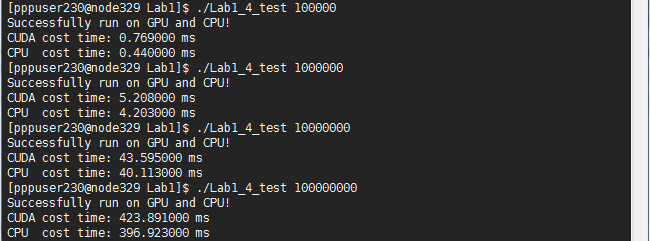


图1-9 CUDA方法计算向量加法，blocksize=32，n显示设置

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

**CPU方法：**

程序开始;

Mat image = imread(); //载入图像

for遍历image除边界外的所有像素点：

取中心点周围卷积核大小的像素块point\_ROI;

Convolute(point\_ROI, kernel); //卷积操作

对边界点赋值为0;

程序结束;

**Pthread方法：**

程序开始;

Mat image = imread; //载入图像

;

创建n个pthread;

每个线程执行一次point\_ROI\*kernel矩阵乘并修改image像素值；

等待线程结束，显示image;

程序结束;

2.3 实验方案

开发环境：windows7+visual studio2017+opencv3.0.0

运行环境：Xshell远程连接到Linux服务器

2.4 实验结果与分析

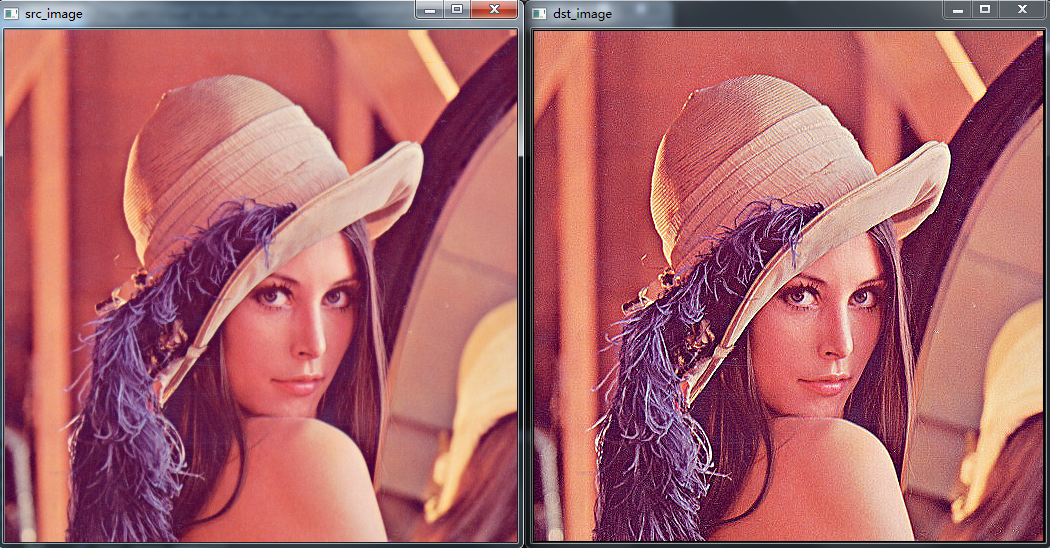


图2-1 windows7+VS2017+opencv3.0.0运行效果

执行结果对比：

实验三

实验四

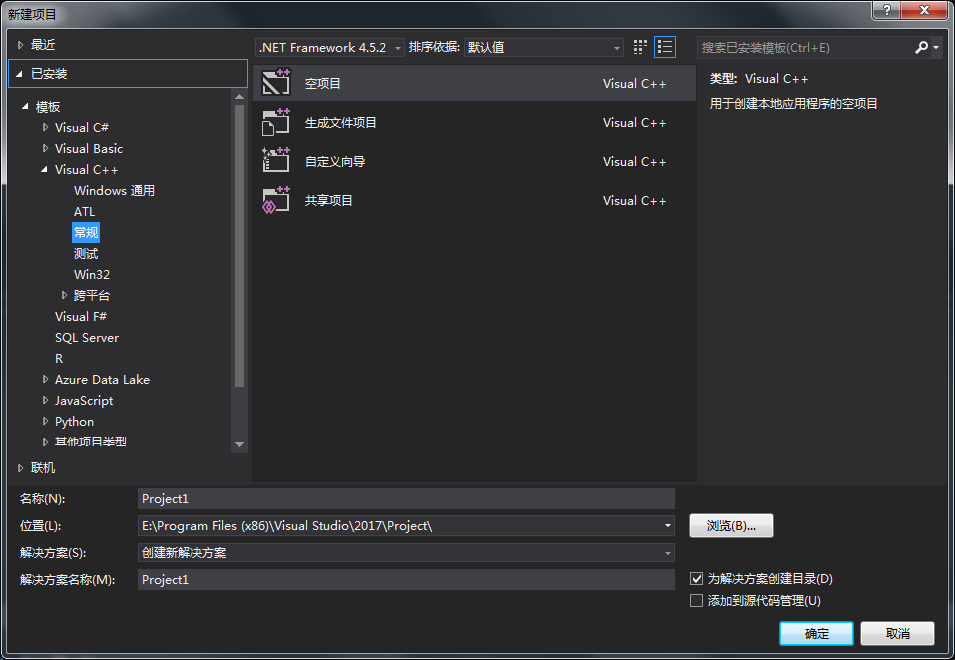
实验五

附录

在OpenCV3.0.0 + Visual Studio 2017环境下创建工程相关配置。

1. 新建项目

点击文件→新建→Visual C++ 空项目→opencv\_conv



1. 右键工程文件夹下的“源文件”→新建项→C++文件→conv\_func.cpp→确定
2. 右键工程文件夹“opencv\_conv”→属性→“VC++目录”下

“包含目录”添加：

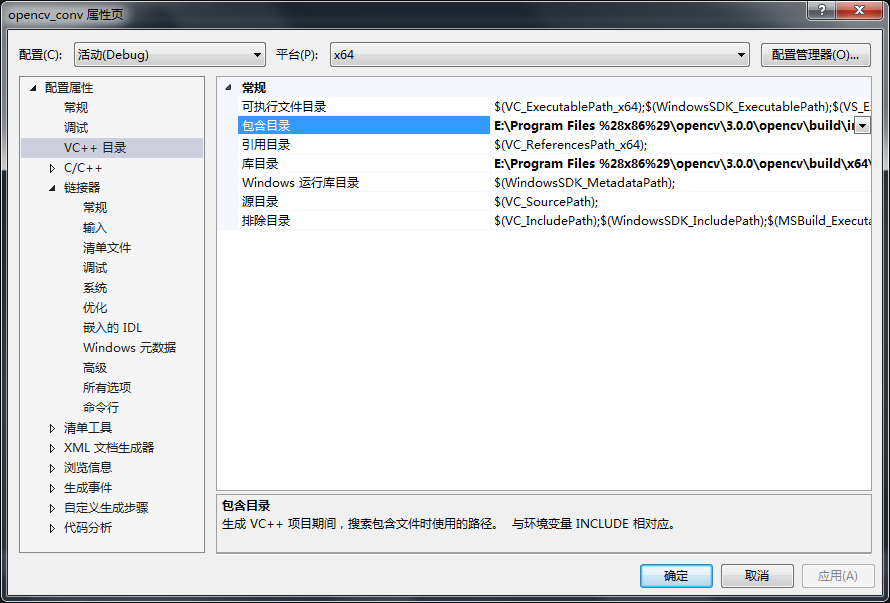
${OpenCV解压路径}\opencv\build\include

${OpenCV解压路径}\opencv\build\include\opencv

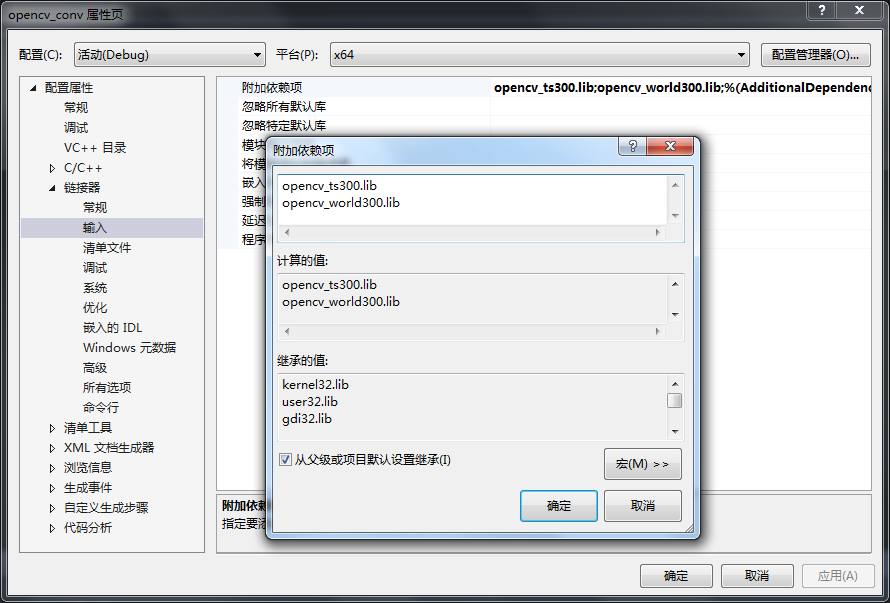
${OpenCV解压路径}\opencv\build\include\opencv2

“库目录”添加：

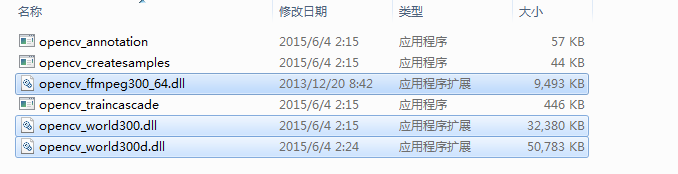
${OpenCV解压路径}\opencv\build\x64\vc12\lib



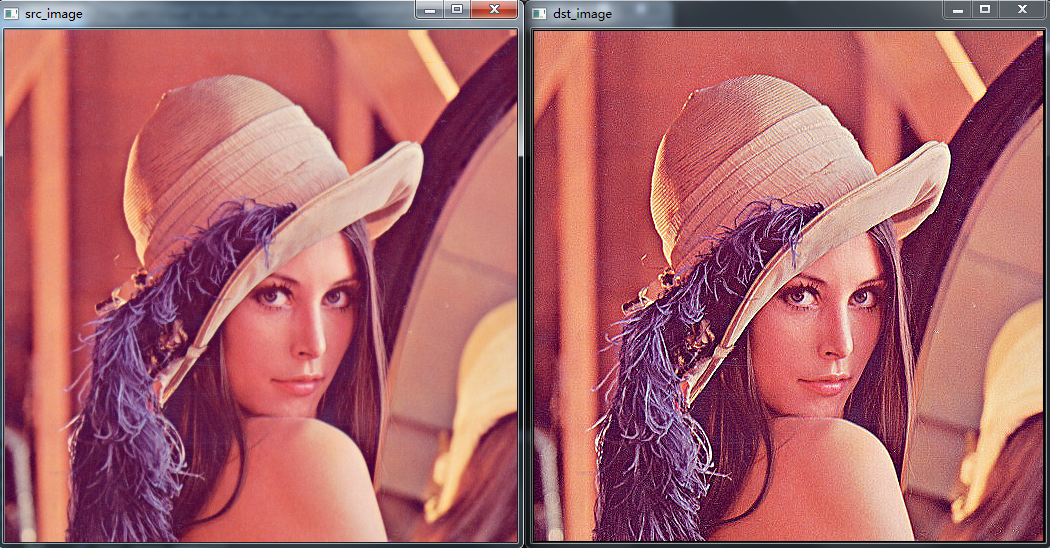
1. 链接器→附加依赖项→添加“opencv\_ts300.lib”和“opencv\_world300.lib”（一定要手动输入！）



1. 将“${OpenCV解压路径}\opencv\build\x64\vc12\bin”中的动态链接文件复制到“C:\Windows\System32”中，该文件夹为Windows7系统的动态链接文件夹，可以省去每次执行“opencv程序.exe”之前需把这三个动态链接文件复制到“opencv.exe”相同文件夹下。



1. 编制程序→编译链接→执行“.exe”文件



参考资料

1. Using std::cout in multiple threads, [https://stackoverflow.com/questions/18277304](https://stackoverflow.com/questions/18277304/using-stdcout-in-multiple-threads).
2. C++: Simple return value from std::thread?, <https://stackoverflow.com/questions/7686939/>.
3. Thread – C++ reference, <http://www.cplusplus.com/reference/thread/thread/>.