

# 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

## 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，可以将基本向量加法程序的源程序修改成如下形式：

#pragma omp parallel for

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

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

OpenMP不需要专门的库，而是内建在编译器中的。使用CMake进行项目构建时，可以使用FindOpenMP库来检测OpenMP支持，以及设置OpenMP编译相关的flags。

**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, &world\_rank); //获取进程标识符

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

elems\_per\_proc = N / world\_size; // 每个进程处理的元素数

MPI\_Scatter(B, elems\_per\_proc, MPI\_INT,

B\_recv, elems\_per\_proc, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Scatter(C, elems\_per\_proc, MPI\_INT,

C\_recv, elems\_per\_proc, MPI\_INT, 0, MPI\_COMM\_WORLD);

for (i in 0 until 100)

B\_recv[i] = B\_recv[i] + C\_recv[i]

MPI\_Gather(B\_recv, elems\_per\_proc, MPI\_INT, A, elems\_per\_proc, MPI\_INT, 0, MPI\_COMM\_WORLD) // 收集计算结果

if world\_rank == 0 then print all elements in A // 在主进程打印结果

MPI\_Finalize();//结束MPI环境

将要进行计算的B与C向量划分为等大小（elems\_per\_proc）的数据块，通过MPI\_Scatter分布到每个子进程中进行计算，再通过MPI\_Gather函数将数据汇总到主进程，并在主进程输出结果。

**1.2.4 使用CUDA做向量加法**

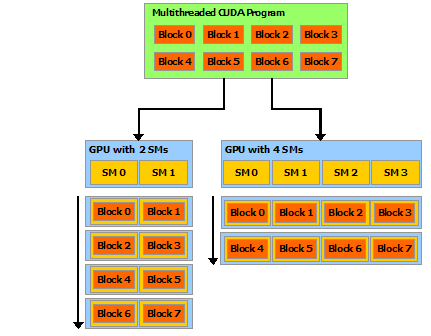


图 1.1 CUDA编程模型

CUDA的编程模型如图 1.1所示。CUDA程序为Host，在调用Kernel函数的时候访问Device。Device中的计算是并行执行，最小的执行单元是线程（Thread），每个Block可同时执行多个Thread，每个Grid中可同时执行多个Block。

我们通过两个三维索引blockIdx和threadIdx来获取当前函数调用所在的block/thread信息。Kernel函数的定义如下：

\_\_global\_\_ void VecAdd(int\* A, int\* B, int\* C, int N) {

int i = blockDim.x \* blockIdx.x + threadIdx.x;

if (i < N) {

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

}

}

每个thread执行一个索引的加法运算。通过blockIdx和threadIdx计算出该thread对应计算的索引号，并执行计算。

调用kernel的过程如下：

int \*dstA, \*dstB, \*dstC;

// 分配设备内存

cudaMalloc((void\*\*) &dstA, N \* sizeof(int));

cudaMalloc((void\*\*) &dstB, N \* sizeof(int));

cudaMalloc((void\*\*) &dstC, N \* sizeof(int));

int threadsPerBlock = 256;

int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

// 将数据拷贝到设备

cudaMemcpy(dstB, B, sizeof(int) \* N, cudaMemcpyHostToDevice);

cudaMemcpy(dstC, C, sizeof(int) \* N, cudaMemcpyHostToDevice);

// 调用kernel

VecAdd<<<blocksPerGrid, threadsPerBlock>>>(dstA, dstB, dstC, N);

// 将计算结果拷贝回host

cudaMemcpy(A, dstA, N \* sizeof(int), cudaMemcpyDeviceToHost);

// 释放设备内存

cudaFree(dstA);

cudaFree(dstB);

cudaFree(dstC);

1.3 实验结果

**1.3.1 std::thread方法**

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 1.2所示。

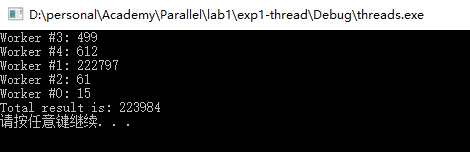


图 1.2 pthread方法示例

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**

编译运行：

* 在lab1/exp2-openmp文件夹中cmake .创建工程
* 使用平台相关的构建方法进行构建运行

为了让实验结果更为直观，我们在循环体中可以通过omp\_get\_thread\_num()函数获取每次执行工作时的进程id并打印出来。测试结果如图 1.3所示。

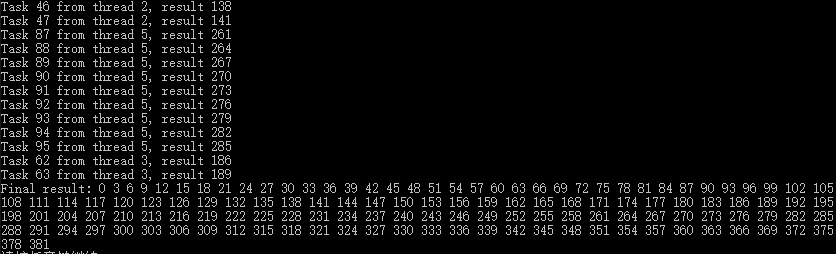


图 1.3 OpenMP计算向量加法样例, n=128

**1.3.3 OpenMPI方法**

编译运行：

* 在lab1/exp3-mpi文件夹中cmake .创建工程
* 使用平台相关的构建方法进行构建运行

将B和C数组划分成等大小的数据块，并分布到每个进程。每个进程对获取的数据块执行加法计算。运行效果如图 1.4所示。

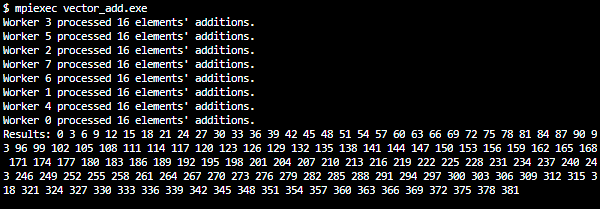


图 1.4 OpenMPI方法计算向量加法, n=128

**1.3.4 CUDA方法**

编译运行：

* 在lab1/exp4-cuda中cmake -A x64 .创建工程
  + 如果使用windows：需要先安装CUDA。目前CUDA 9.2对Visual Studio 2017的支持有问题。如果安装了VS2017，需先额外安装msvc140 toolset，并且使用cmake -A x64 -G “Visual Studio 14 2015”指令生成vs2015编译器的项目。
* 使用平台相关的构建方法进行构建运行

运行结果如图 1.5所示。可以看到加法计算结果正确。

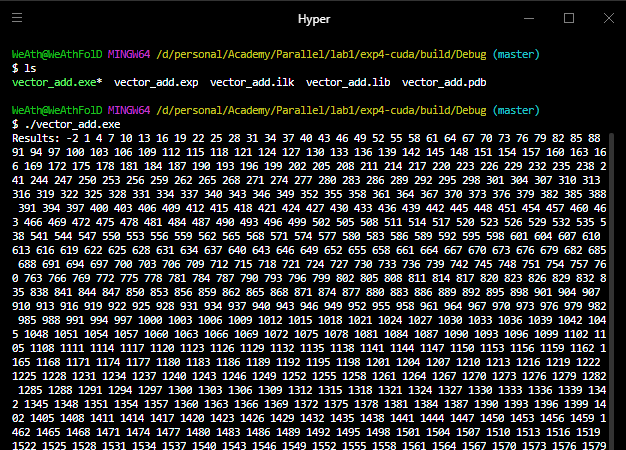


图 1.5 CUDA向量加法运行结果

## 实验二

2.1 实验目的与要求

1. 掌握使用thread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thread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：**

在这里选择erosion（扩张）算法进行实现。后续所有涉及形态学图像处理处皆使用erosion卷积核，不再重复说明。

function morph\_erosion(Image source, Image dest):

for each pixel position (x, y) in source:

hit = false

for (dx, dy) in (-1, -1) … (1, 1):

if source pixel is set (x + dx, y + dy):

hit = true

break

set dest pixel (x, y) to hit ? 1 : 0

**std::thread方法：**

使用thread进行并行化本质上是将上述的顺序算法分解成多个线程。在这里我们将图像数据考虑为一个大小为width \* height的连续数组，将数组尽量均匀的划分到thread\_count（可调整）个线程，每个线程负责一块连续区域。算法如下：

function worker(Image source, Image dest, int offset, int count):

for i in offset until offset + count:

y = i / source.width // 反算出像素坐标

x = i / source.width

bool hit = false

for (dx, dy) in (-1, -1) … (1, 1):

if source pixel is set (x + dx, y + dy):

hit = true

break

set dest pixel (x, y) to hit ? 1 : 0

function morph\_erosion(Image source, Image dest):

thread\_count = ??? // 设置线程数

size = source.width \* source.height

per\_thread = size / thread\_count // 计算单个线程负责像素的数量

for i in 0 until thread\_count: // 生成线程

spawn thread worker(source, dest, per\_thread \* i, per\_thread)

join all child threads // 等待所有线程执行结束

2.3 实验方案

开发/运行环境：Windows10+ CMake +Visual Studio2017

由于后续实验lab2-lab5中都要以不同的方式实现形态学图像处理，且都有相同的读取图像、写图像、计算运行时间的需求，我们先编写了一个简单的库morphlib来执行上述操作。它定义了一个morph命名空间，其中包含：

* BinaryImage类，可以读取二进制图数据和写出数据
* 函数类型test\_func，是执行形态学图像处理操作的核心函数。每个lab中只需要重新实现一个test\_func即可。
* 函数exec\_test，它接受程序的命令行参数和一个test\_func，并读取文件、调用test\_func并测量执行时间，最后输出运算后的图像文件。

为了方便C++代码的编写，我们定义了一种简单的二值图像的二进制格式，它的构成是：

1. 两个32位无符号整数，分别代表width和height；
2. 余下长度为(width\*height)字节的图像数据。每个字节非0即1，代表该像素是否为白。

我们又通过Kotlin（它可以调用java的图像读取、写函数，对各种图片格式都有较好的支持）编写了一个简单的命令行工具，来将一般图片格式和这种二值格式相互转换。其用法如下：

* Java -jar imgcvt.jar b image.png binary-image.bin // 将png转换为二值图
* Java -jar imgcvt.jar p binary-image.bin image.png // 将二值图转换为png

在周边工具编写完成之后，我们首先编写了一个线性执行erosion操作的程序，用于参考和后续数据比较，最后才用thread的形式实现了erosion操作。

2.4 实验结果与分析



图 2.1 erosion形态学图像处理程序运行效果

对大小为900\*600的二值图片进行erosion操作的效果如图 2.1所示。由于后续lab2-lab5的执行结果都相同（在运行正确的前提下），运行结果将不会重复说明。

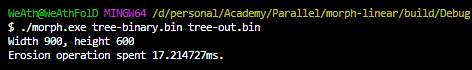


图 2.2 线性计算程序的执行示例

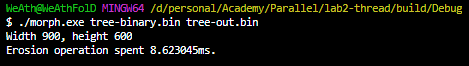


图 2.3 std::thread多线程计算的执行示例

## 实验三

3.1 实验目的与要求

1. 掌握使用openmp的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. 然后对程序执行结果进行简单的分析和总结。

3.2 算法描述

和单线程方法相同，只是在循环体上加了#pragma omp parallel for宏，让编译器自动进行多线程执行优化。

function morph\_erosion(Image source, Image dest):

#pragma omp parallel for

for each pixel position (x, y) in source:

hit = false

#pragma omp parallel for

for (dx, dy) in (-1, -1) … (1, 1):

if source pixel is set (x + dx, y + dy):

hit = true

break

set dest pixel (x, y) to hit ? 1 : 0

3.3 实验方案

让编译器开启OpenMP的CMake关键代码如下：

include (FindOpenMP)

if (OPENMP\_FOUND)

set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} ${OpenMP\_CXX\_FLAGS}")

else()

message("WARNING: OpenMP not supported!")

endif()

C++代码方面，只需要在实验二中所写的顺序执行代码的基础上，在两个循环体的上方添加#pragma omp parallel for宏即可。

3.4 实验结果与分析

执行程序的演示如图 3.1所示。

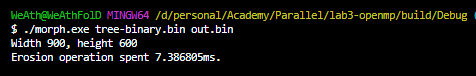


图 3.1 执行omp优化的形态学图像处理程序的结果

## 实验四

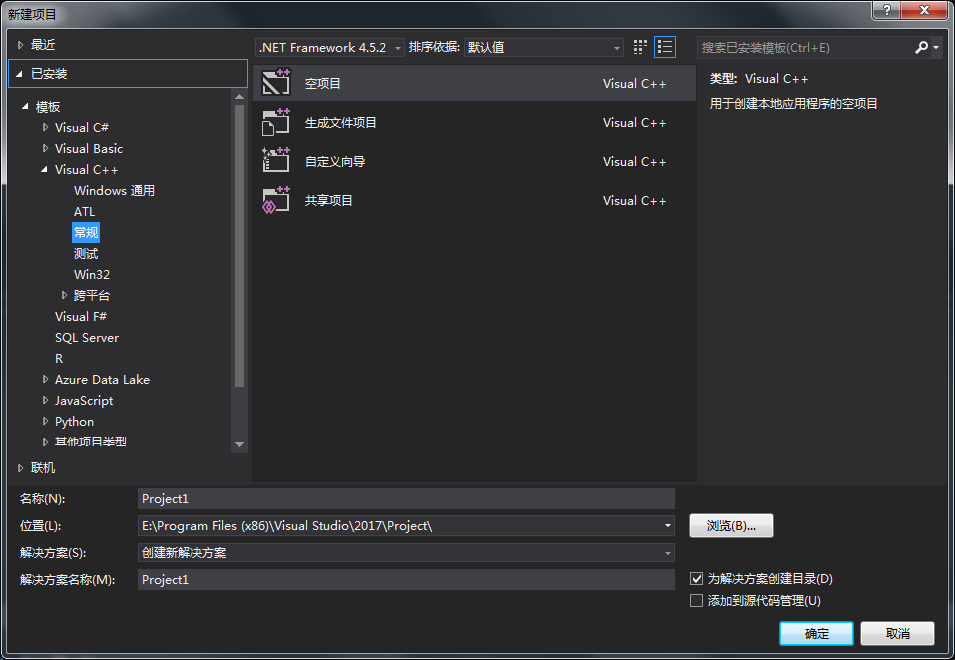
## 实验五

## 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：

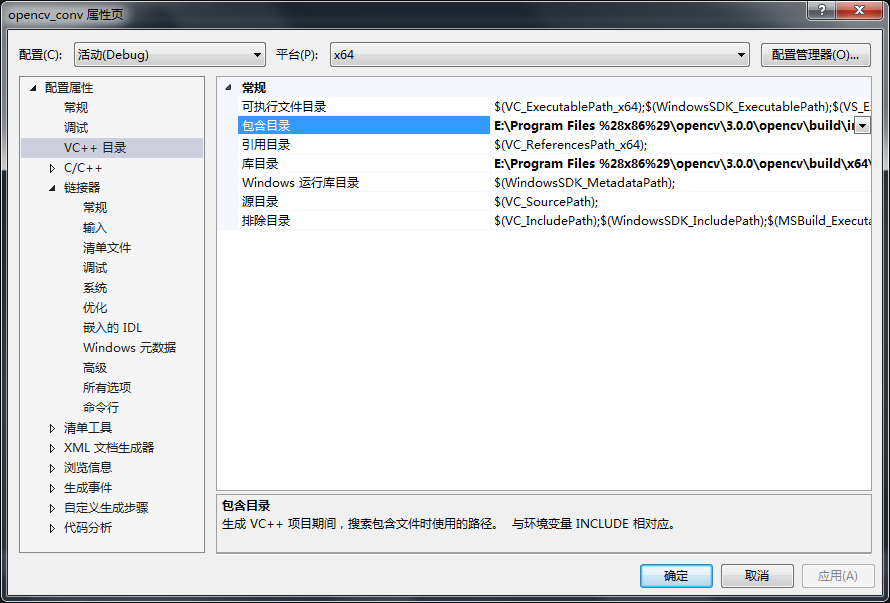
${OpenCV解压路径}\opencv\build\include

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

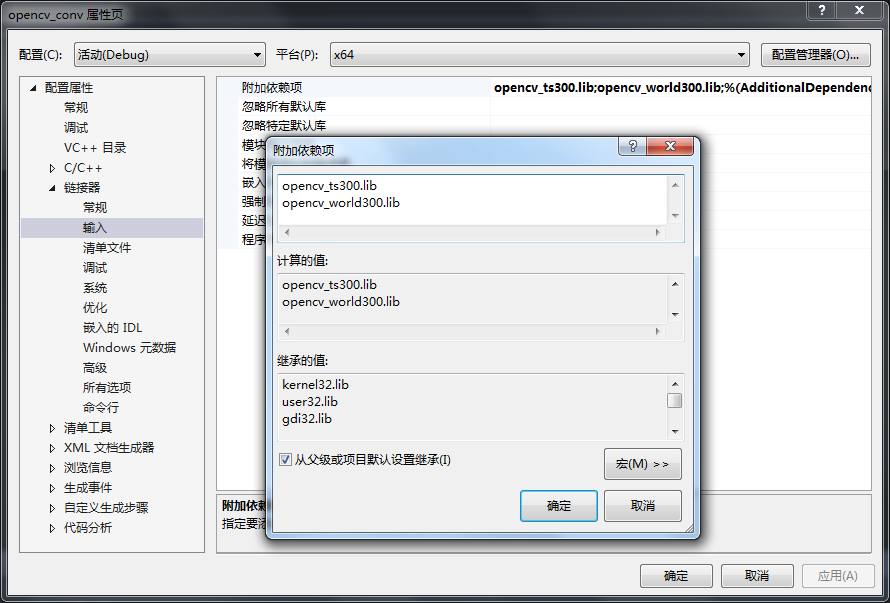
${OpenCV解压路径}\opencv\build\include\opencv2

“库目录”添加：

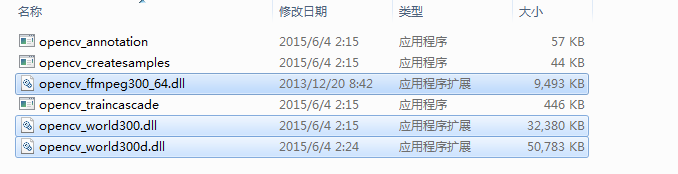
${OpenCV解压路径}\opencv\build\x64\vc12\lib



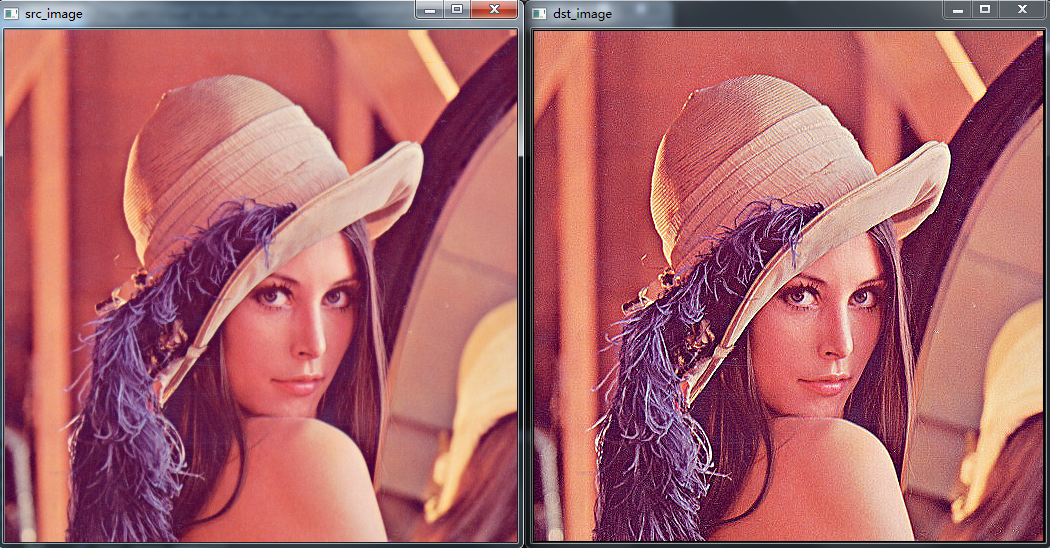
1. 链接器→附加依赖项→添加“opencv\_ts300.lib”和“opencv\_world300.lib”（一定要手动输入！）



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/>.
4. Cmake cannot find OpenMP, <https://stackoverflow.com/questions/17633513>.
5. Microsoft MPI, <https://msdn.microsoft.com/en-us/library/bb524831.aspx>.
6. How to compile an MPI included program using Cmake, <https://stackoverflow.com/questions/23163075>.
7. MPI Hello World, <http://mpitutorial.com/tutorials/mpi-hello-world/>.
8. MPI Scatter, Gather, and Allgather, <http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/>.
9. FindMPI – CMake 3.10.3 Documentation, <https://cmake.org/cmake/help/v3.10/module/FindMPI.html>.
10. Programming Guide :: CUDA Toolkit Documentation, <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
11. CUDA 9.0 does not work with the latest VS 2017 update, <https://devtalk.nvidia.com/default/topic/1027209>.
12. Morphology – Erosion, <https://homepages.inf.ed.ac.uk/rbf/HIPR2/erode.htm>.
13. Java – How to Save a BufferedImage as a File, <https://stackoverflow.com/questions/12674064>.