華中科技大學

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：	胡思勖
专业班级：	计卓 1501
学　　号：	U201514898
指导教师：	金海

报告日期：2018 年 7 月 15 日

目录

实验一　熟悉并行编程环境

1.1　实验目的与要求

熟悉并行编程环境，并行编程的基本法则和方法以及在 Linux 下使用 pthread、OpenMP、MPI 的性能优化方法。

1.2　实验内容

1.2.1　熟悉 Linux 下的并行编程环境

首先，需要熟悉 Linux 下的并行编程环境。并行编程的平台有很多，而广泛使用的有 pthread、OpenMP、MPI、CUDA 等平台，其中前 3 种运行在 CPU 上而第四种则在 CPU 与 GPU 上混合运行。下面使用简单的向量加法来说明各个环境的运行机制。

串行的向量加法　令 $\vec{a},\vec{b},\vec{c}$ 为 3 个长度为 n 的向量，则求 $\vec{c}=\vec{a}+\vec{b}$ 的串行过程如下：

Algorithm 1: 串行加法

```
1: procedure SerialAdd(a⃗, b⃗, c⃗, n)
2:     for i = 1 to n do
3:         c[i] = a[i] + b[i]
```

这样，$c[i]=a[i]+b[i]$ 需要被执行 n 次，设每次执行的平均时间为 t 则共需至少 $n\times t$ 的时间。当 n 非常大时，此算法需要的时间则较长，而并行算法则是使执行这些加法的时间有不同程度的重叠，从而减少程序执行所需要的总时间。

使用 pthread 执行向量加法　Pthread（POSIX Threads），是 POSIX 的线程标准，定义了创建和操纵线程的一套 API。一般用于 Unix-like POSIX 系统，如 Linux、Solaris。其中包含了以下的功能：

1. 线程管理，例如创建线程，等待 (join) 线程，查询线程状态等。
2. 互斥锁（Mutex）：创建、摧毁、锁定、解锁、设置属性等操作

3. 条件变量（Condition Variable）：创建、摧毁、等待、通知、设置与查询属性等操作

4. 使用了互斥锁的线程间的同步管理

要使用 pthread 执行线程加法，需要创建线程，使用多个线程对于不同的分块进行加法运算，最后等待所有的线程完成。创建线程以及等待所有线程完成的一个代码示例如下，使用`pthread_create`创建线程，使用`pthread_join`等待所有线程完成。`pthread_create`世纪上传给`sum_array`的参数只有一个，因此如果需要在创建线程时传递多个参数，可以通过 class 进行参数的打包，也可以使用分配在堆上的变量（如全局变量等）进行传参。

```
1  for (int i = 0; i < MAX_THREAD; i++)
2      pthread_create(&threads[i], NULL, sum_array, (void*)NULL);
3  for (int i = 0; i < MAX_THREAD; i++)
4      pthread_join(threads[i], NULL);
```

上述创建以及等待线程的代码时几乎所有使用 pthread 的程序都需要使用的代码。pthread 库中大致有 100 个函数调用，此处不再详细列举。在后面的实验中可以看到，这些函数是如何紧密配合从而写出高性能的串行代码的。

使用 OpenMP 执行向量加法　OpenMP（Open Multi-Processing）是一套支持跨平台共享内存方式的多线程并发的编程 API，是用于共享内存并行系统的多线程程序设计的一套 Compiler Directive（使用 #pragma 开头的注释）。也就是说，其实现是由预编译器与编译器共同完成的，对于 gcc 的实现版本而言其最终实现依然是通过 pthread 进行的，但其简单的语法规则使得程序编写者能够专注与算法而不是并行的底层实现。

使用 openmp 将两个向量相加的部分代码如下：

```
1  #pragma omp parallel shared(a, b, c, chunk) private(i)
2  {
3  #pragma omp for schedule(dynamic,chunk) nowait
4  for (i=0; i < N; i++) {
5      c[i] = a[i] + b[i];
6  }
7  }
```

其中，紧跟在`#pragma omp`后面的称为 directive，OpenMP 一共定义了包含 atomic、barrier、parallel、for 在内的 11 个 directive 以表示不同的功能，其后的如 shared、schedule 称为 clause，OpenMP 中一共定义了 13 个 clause 用于作为功能的参数。此外，OpenMP 库中还有 20 多个库函数用于读取线程数、线程编号等线程有关信息。

使用 MPI 执行向量加法　MPI（Message Passing Interface）是一个并行计算的 API，用于在各个处理器及计算机的进程之间传递消息。需要注意的是主要的 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 的

编程模型中并行的单元是进程而不是线程。与 OpenMP 不同，大多 MPI 是通过库实现的，不需要编译器的支持。

使用 MPI 进行向量相加，首先需要将向量广播到各个处理机上，如果需要所有的处理机知道全部的向量内容可以使用`MPI_Bcast`进行广播，而如果各个处理机只需要知道向量的一部分则可以使用`MPI_Scatter`进行数据的分散。在各个进程处理完成之后，需要使用`MPI_Gather`对于数据进行回收和组装。

在 Master 进程（0 号进程），进程间通信的关键代码如下：

```
1  // Broadcast control sequence
2  MPI_Bcast (&n_per_prc, 1, MPI_LONG_LONG_INT, MASTER, MPI_COMM_WORLD);
3  // Broadcast element per process
4  MPI_Scatter(a, n_per_prc, MPI_INT, ap, n_per_prc, MPI_INT, 0,
       MPI_COMM_WORLD);
5  // scattering array a
6  MPI_Scatter(b, n_per_prc, MPI_INT, bp, n_per_prc, MPI_INT, 0,
       MPI_COMM_WORLD);
7  // scattering array b
8  for(i=0;i<n_per_prc;i++)
9      cp[i] = ap[i]+bp[i];
10 MPI_Gather(cp, n_per_prc, MPI_INT, c, n_per_prc, MPI_INT, MASTER,
       MPI_COMM_WORLD);
```

在运行时，与 pthread、OpenMP 程序不同的是，在编译时需要使用包装的编译器 mpicc 或 mpic++ 进行编译。同时，由于 MPI 是进程间并行，因此需要使用 mpirun 同时执行多个程序，mpirun 中的-np 参数用于指定程序的个数，如 mpirun -np 4 ./a.out 命令的作用是同时启动 4 个进程执行 a.out 程序。

使用 CUDA 执行向量加法　上述的并行均在 CPU 平台执行，而遇到并行量极大的程序则可以 offload 到 GPU 上执行。CUDA（Compute Unified Device Architecture）是由 NVIDIA 所推出的一种集成技术，是该公司对于 GPGPU 的正式名称。透过这个技术，用户就可利用 NVIDIA 的 GeForce 8 以后的 GPU 和较新的 Quadro GPU 进行计算，从而实现使用 GPU 对于大量的数据进行并行计算。

在 CUDA 中，有线程（Thread）、块（Block）以及网格（Grid）的概念，其关系如图1.1所示。其中 Thread 和 block 都是 1~3 维的，可以根据需要进行分配。此外，由于 GPU 处理复杂逻辑的能力有限，因此不可能让所有的程序都在 GPU 上运行，整个程序的控制流应该为 CPU–GPU–CPU–…，如图1.2 所示。由于 CUDA 编程模型假设主机和设备分别在 DRAM 中维护它们各自的存储空间（分别称为主机存储器和设备存储器），因此在 CPU 与 GPU 的控制流转换之间，使用 CUDA 提供的 cudaMalloc、cudaMemcpy 以及 cudaFree 对于 CPU（host）以及 GPU（device）的内存进行管理。

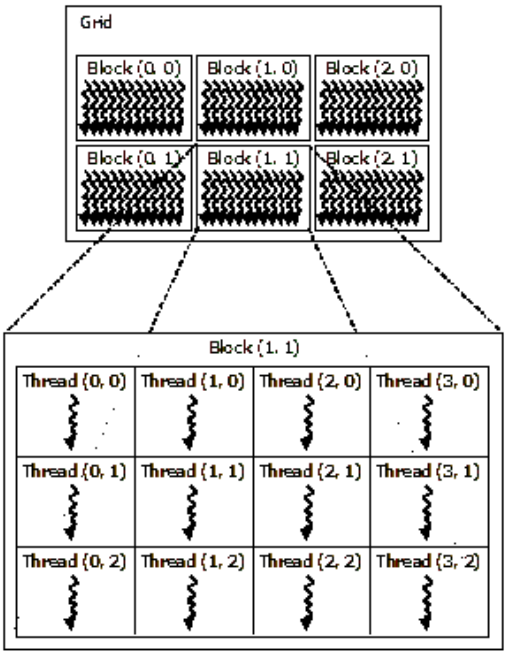

图 1.1: CUDA 中的 Thread、Block 以及 Grid

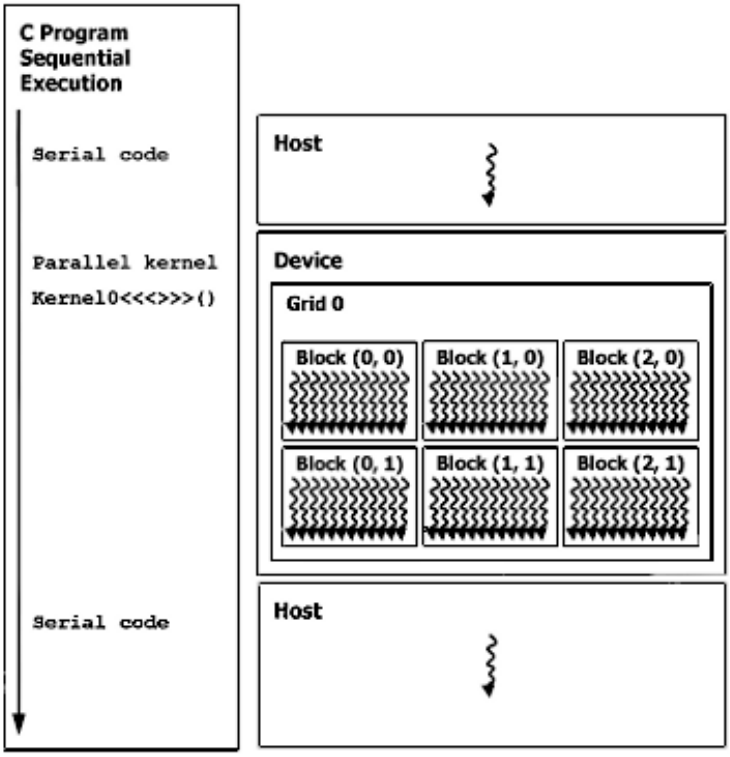

图 1.2: CUDA 程序控制流

1.2.2　编写图像处理串行参照程序

在实际的编写并行的形态学图像处理程序之前，需要编写一个串行的图像处理算法，以用于作为功能上以及性能上的参照。

首先，需要将磁盘中的图片读入内存。为了简便起见，使用 cimg 库读入图片。cimg[1]库是一个 C 语言的图像处理库，其只包含头文件，且性能较为优秀。图1.3为使用 cimg 读入图片，然后进行灰度化后的结果。对于每一个点而言，计算其灰度（亮度）的公式如下（使用 PAL 制式）：

$$luminosity = 0.299 \times red + 0.587 \times green + 0.114 \times blue$$

其中，red、$green$ 和 $blue$ 分别为红、绿、蓝三个通道的亮度值，而 $luminosity$ 则为所求灰度值。为了便于与其他程序进行对比，使用 Lenna[2]作为接下来所有图片处理程序的输入数据。

为了能够对图片进行膨胀或蚀刻处理，首先需要将图片进行二值化处理。使用 cimg 中的`threshold()`函数对于图像进行二值化，设置阈值为 128，二值化的图形如图1.4所示。需要注意，二值化的图片在内存中是使用 0/1 来表示的，而这一显示的图片还进行了在 $[0, 255]$ 区间上的归一化处理。

图 1.3: 程序使用的样例图片（灰度化）

图 1.4: 程序使用的样例图片（二值化）

在上述预处理完成后，接下来对于图片进行蚀刻和膨胀处理。蚀刻和膨胀均采用 5×5 的 kernel，其取值如下。kernel 大小、形状和取值不同会导致膨胀或蚀刻的效果不同。

$$kernel_{\text{蚀刻}} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}, kernel_{\text{膨胀}} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

[1] `http://cimg.eu/`

[2] `https://en.wikipedia.org/wiki/Lenna`

对于图片进行膨胀和蚀刻的串行算法如下：

Algorithm 2: 蚀刻和膨胀串行算法

procedure ErodeAndDilate($image, kernel_e, kernel_d$)
  for each pixel a **in** $image$ **do**
    $result_e \leftarrow 1, result_d \leftarrow 0$
    for each pixel b_e, b_d **in** $kernel_e, kernel_d$ **do**
      find b_e or b_d's corresponding pixel c
      $result_e \leftarrow result_e \wedge (c \vee b_e), result_d \leftarrow result_d \vee (c \wedge b_d)$
    $dilatedImage.at(a.pos) \leftarrow result_d, erodedImage.at(a.pos) \leftarrow result_e$
  return $dilatedImage, erodedImage$

可以看出，由于对于 $image$ 的各个点之间的计算是没有依赖的，因此具有很强的可并行性。编写程序显示时刻和膨胀和膨胀后的图片，分别如图1.5和1.6所示。由于是对于白色的部分（二值图像中 “1” 的部分）进行蚀刻和膨胀，因此在腐蚀的图片中人像部分看起来变厚了，而在膨胀的图片中人像看起来变薄了。

图 1.5: 蚀刻后的图片

图 1.6: 膨胀后的图片

将蚀刻后的图片部分放大，可以看到反色的 “十” 字形状图案，如图1.7所示，这是由于在蚀刻时使用了 “十” 字形的 kernel 导致的。如果 kernel 取值不同，将产生不同形状的细节。

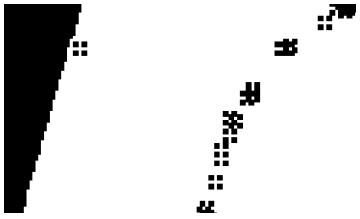

图 1.7: 蚀刻后的图片（部分放大）

1.3　实验结果

通过这一阶段的调查和研究，以及对于各种并行方法的简单实践，并行编程的平台与环境已经准备充分。此外还有串行程序的结果作为参考。程序所示有的 Lenna 图片的大小为 512×512 由于处理一张图片的速度过快，无法进行性能的比较，因此在所有的程序中将图片处理的过程（仅蚀刻和膨胀的过程，不包含预处理）重复 1000 次，然后统计总时间以进行性能上的比较，这样可以减小各个并行平台初始化时间以及图片的预处理时间带来的影响。

串行的程序在 Intel® Core™ i7-7700K CPU @ 4.20GHz 上运行，之后的并行程序也在此 CPU 上运行，以便于进行性能的比较。运行结果如图1.8所示。

```
round 986 finished
round 987 finished
round 988 finished
round 989 finished
round 990 finished
round 991 finished
round 992 finished
round 993 finished
round 994 finished
round 995 finished
round 996 finished
round 997 finished
round 998 finished
round 999 finished
~/H/Parallel  dev  lab1                    44.2s 〈 Sat Jul 14 21:49:56 2018
~/H/Parallel  dev  lab1  history | head -n 1
./lab1
~/H/Parallel  dev  lab1                          Sat Jul 14 21:52:15 2018
```

图 1.8: 串行图像处理程序运行结果

从图中可以看出，程序运行的时间为 44.2s，经过三次重复实验，平均时间也为 44.2s，这将作为后续的时间比较基准。此处使用的是 fish-shell 插件 omf 的时间统计功能，统计的是一条命令执行的完整时间（walltime）。若需要在 bash 或 zsh 等其他 shell 内运行程序则需要自行编写统计时间的 wrapper script。不使用程序内的时间统计是由于 clock 统计的为 cpu 时间，在执行并行程序时会将多个线程的时间相加，而在程序内直接统计 walltime 也可能由于程序的初始化步骤不同而不准确，因此直接在 shell 中统计时间才能较为完整的比较各个并行平台对于程序的优化能力。

此外由于使用的是 fish-shell，而 fish-shell 内建的 history 命令对于历史记录的排列与 bash 是相反的，因此在 fish-shell 中显示上一条命令的命令为 history | head -n 1，在 bash 中等价的命令应为 history | tail -n 1。

实验二　基于 pthread 的形态学图像处理

2.1　实验目的与要求

- 掌握使用 pthread 的基本的并行编程设计方法以及调优方法；
- 掌握并行编程中基本的数据分块以及任务分解的方法。
- 使用 pthread 实现并行的形态学图像处理。
- 简要分析以及总结处理的结果。

2.2　算法描述

使用多个线程对于一个图像进行蚀刻以及膨胀的算法如下，算法为一个线程的流程，而有多个这样的线程同时进行。

Algorithm 3: pthread 并行处理算法 (一个线程)

```
1: procedure PthreadParallel(blocks)
2:     while true do
3:         lock(blocks)
4:         get first block blk from blocks
5:         if blocks.empty() then
6:             unlock(blocks)
7:             return
8:         unlock(blocks)
9:         ErodeAndDilate(blk, kernel_e, kernel_d)
```

算法中，*blocks* 参数为一个工作队列，队列中的工作为原预处理过后的图片的子图片。在每个线程的每个循环中，首先锁住队列，从队列中获取一个子图片 *blk*、解锁队列然后使用上一章中的 ErodeAndDilate 过程进行处理。如果 *blocks* 中没有子图片，说明处理完成，则此线程退出。

主线程的流程如图2.1所示。在进行预处理过后启动多个线程，然后等待所有线程竞争子图像、处理然后结束即可，最后保存处理的结果即可。

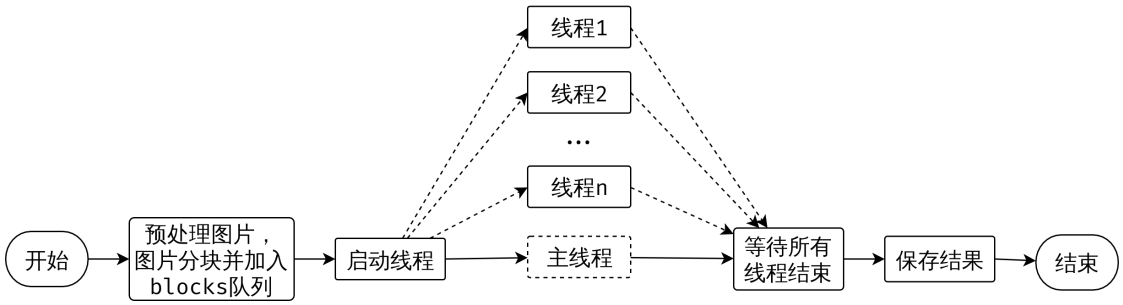

图 2.1: 主线程流程

由于是使用 pthread 的并行算法，每一个线程处理一个部分，因此首先需要将数据分块（即分为算法中的 *blocks*）。分块方式如图2.2所示。每块大小一样，在边缘部分如果块大小不符则按照原图的边缘进行裁减。因此，在进行处理时需要对于边缘部分进行考虑。

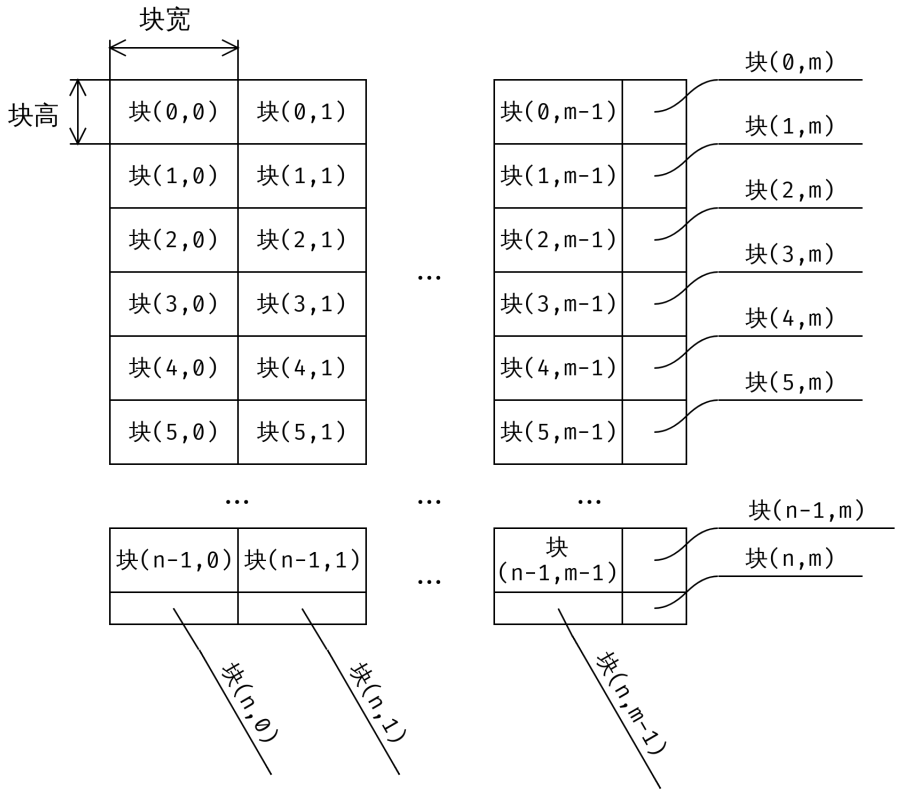

图 2.2: 分块方法

2.3　实验方案

所有的开发与运行环境见附录A，表A.1，此后实验的开发与运行环境均相同，不再赘述。根据算法描述、分块方法以及主线程的流程编写程序并运行，然后观察结果并与串行的程序比较。经过多轮的比较以及参数调试后得出一个较好的效果。

2.4　实验结果与分析

功能上，程序处理后的图片与串行处理后的图片一致，此处不再给出。4 线程，分块大小为 128 的情况下程序的运行时间如图2.3所示。在 4 个线程的情况下，运行三次的平均运行时间为 11.7s，相比于串行算法，程序的加速比为 $44.2 \div 11.7 = 3.77$，已经十分接近理想加速比 4。

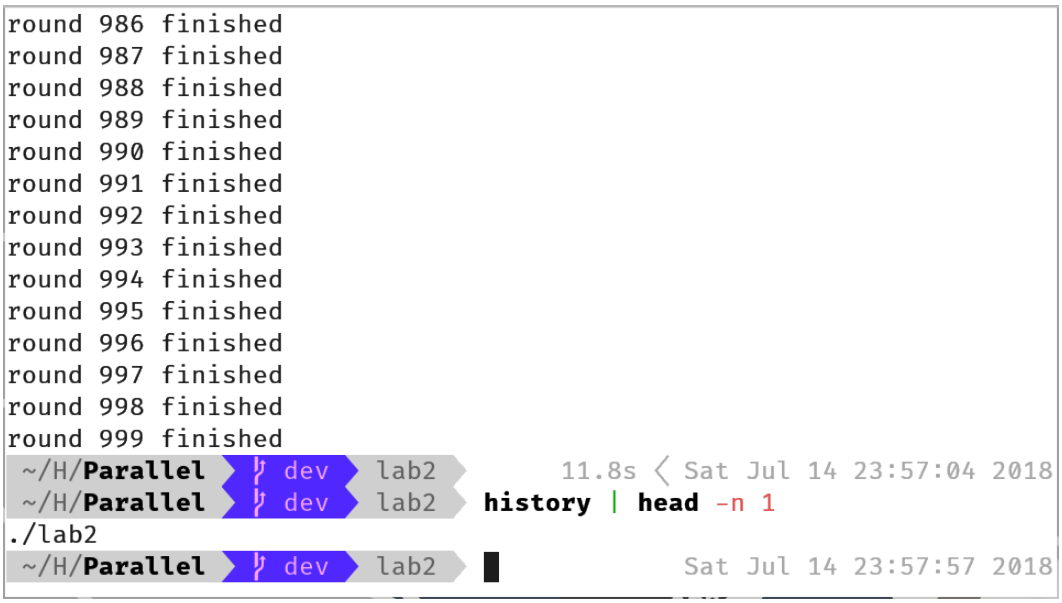

图 2.3: pthread 程序运行时间

经过 8 组、每组 3 次的测试，加速比随线程变化的曲线如图2.4所示。可以看出，在线程数为 1~4 时加速比随着线程数几乎呈线性变化，而在线程数为 1 时加速比为 1.006，overhead 所占用的时间几乎可以不计。在线程数达到 4 时由于物理内核已经被占满，因此后面加速比不再增加，随着线程数量的进一步增大，由于线程调度的开销，因此程序的加速比不再增加，反而有所下降。

对于分块大小而言，加速比随着分块大小的变化如图2.5所示，在分块大小较小时，加速比随着分块大小的变化并不大，只在分块大小过小时由于线程调度导致一点性能开销。当分块大小大于原图的一半时总时间则取决于分到最大分块线程所用的时间，因此在这个区间内性能随分块大小呈下降趋势。

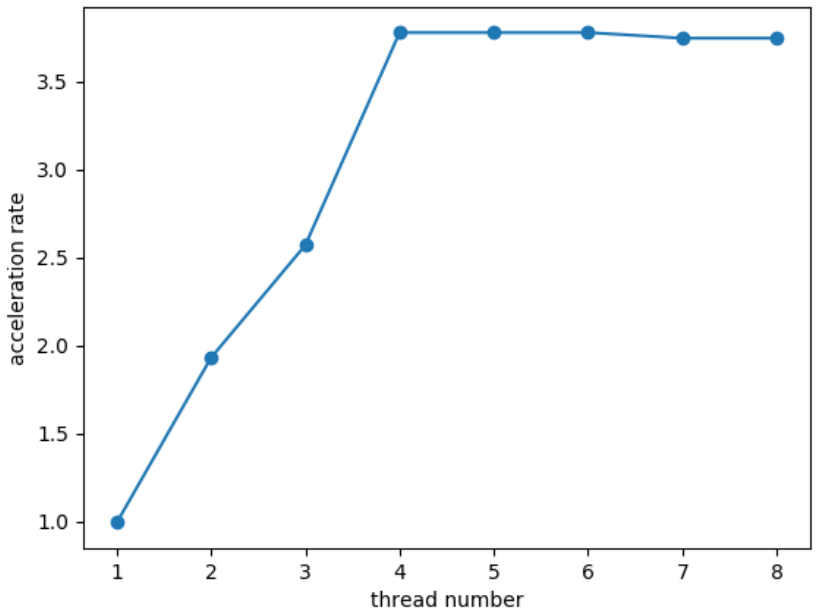

图 2.4: 加速比随线程数变化

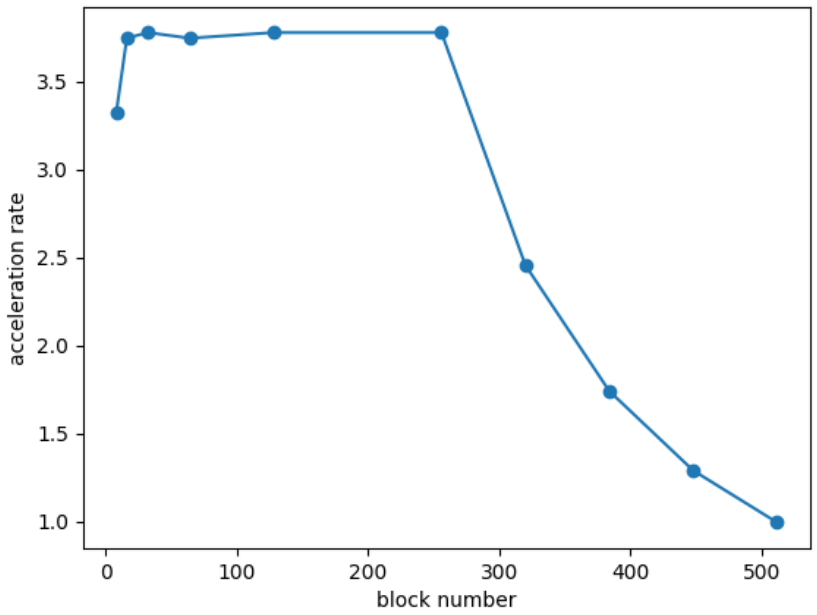

图 2.5: 加速比随分块大小变化

实验三　基于 OpenMP 的形态学图像处理

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 实现形态图像处理操作的并行算法
3. 对程序执行结果进行简单的分析和总结
4. 将其与 Lab2 的结果进行比较

3.2　算法描述

使用 OpenMP 进行并行计算与使用 pthread 进行并行计算的算法原理几乎一样，只不过 OpenMP 由编译器进行底层实现，使用 for 循环进行包装，且主线程不是空闲的，而是也参与计算。而 pthread 则需要人工进行底层的编写。使用 OpenMP 的关键部分代码如下：

```
1 #pragma omp parallel for num_threads(threadNum) schedule(dynamic, 2)
2 for(int i=0; i < workNum; ++i){
3     // Code equvilent to calling ErodeAndDilate(blk, kernel_e, kernel_d)
4 }
```

OpenMP caluse 中的 schedule(dynamic, 2) 将所有的人物按照 2 个一组进行分组，而每个线程领取一组任务、执行完成后在领取下一组可能的任务，从而避免了由于静态调度以及线程执行时间不等带来的 CPU 空闲。

3.3　实验方案

由于算法与 pthread 基本相同，因此可以在 pthread 的实现上进行修改，将 pthread 中的线程竞争调度改为使用 openmp directive 包装的 for 循环，并在编译条件中去除-lpthread，加上-fopenmp 即可。

在运行时，通过多次调节参数并进行重复实验，得到一侧较优的参数，然后将这个参数与 phtread 实现进行对比，观察 OpenMP 实现与 pthread 实现的区别。

3.4　实验结果与分析

功能上，OpenMP 能够正确的处理图片并给出与串行处理相同的结果。性能上，使用 4 线程、分块大小 128 时程序的运行结果如图3.1所示。三次测试平均结果为 12.63s，相对于串行程序加速比为 $44.2 \div 12.63 = 3.49$，与同条件下的 pthread 实现比起来加速比要低不少，这是由于 OpenMP 需要进行环境的初始化，且其调度过程没有特别为底层实现的 pthread 程序优化，因此 pthread 实现的并行程序性能较为优秀。

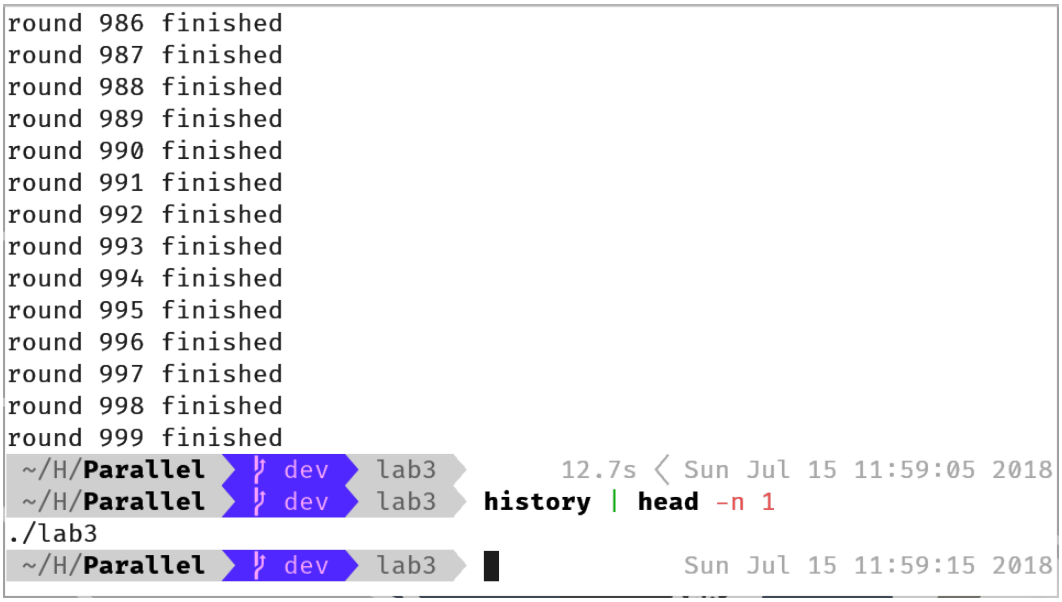

图 3.1: 使用 OpenMP 进行处理的输出

加速比随线程数量变化如图3.2所示，其中实线部分为使用 OpenMP 的程序，而虚线部分为上述使用 pthread 的程序。从图中可以看出，使用 OpenMP 和使用 pthread 程序的性能随着线程数量的变化趋势是一样的，因为其算法本质上是相同的。而使用 OpenMP 的程序在同样的情况下总是比使用 pthread 的程序要慢一些，这也是由于 OpenMP 并不是一个针对特殊程序的调度算法，因此没有更为底层的，为特殊目的设计的 pthread 程序性能高，但在仅需要编写较少代码的情况下能够达到与 pthread 程序接近的性能说明其实现也是十分优秀的。

而加速比随分块大小的变化如图3.3所示，同样，实线为 OpenMP 实现，虚线为 pthread 实现。与随着线程变化的趋势一样，在同样的情况下 OpenMP 的性能比 pthread 性能低。在分块大小超过图片的 1/2 时更为明显，其下降趋势更为剧烈，推测是由于 OpenMP 的启动与初始化时间以及调度的分块引起的。

从上述结果可以得出结论：在对于并行性能要求不是非常高时，使用 OpenMP 可以简化程序编写者的工作量，而使用 pthread 作出针对特定程序的并行算法则相对于 OpenMP 算法可以较大的提高程序的性能。

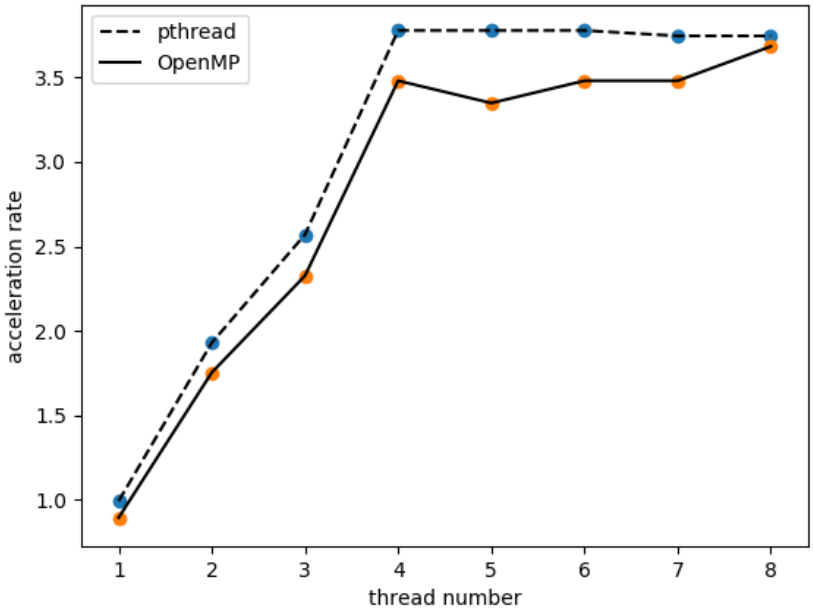

图 3.2: 加速比随线程数量变化

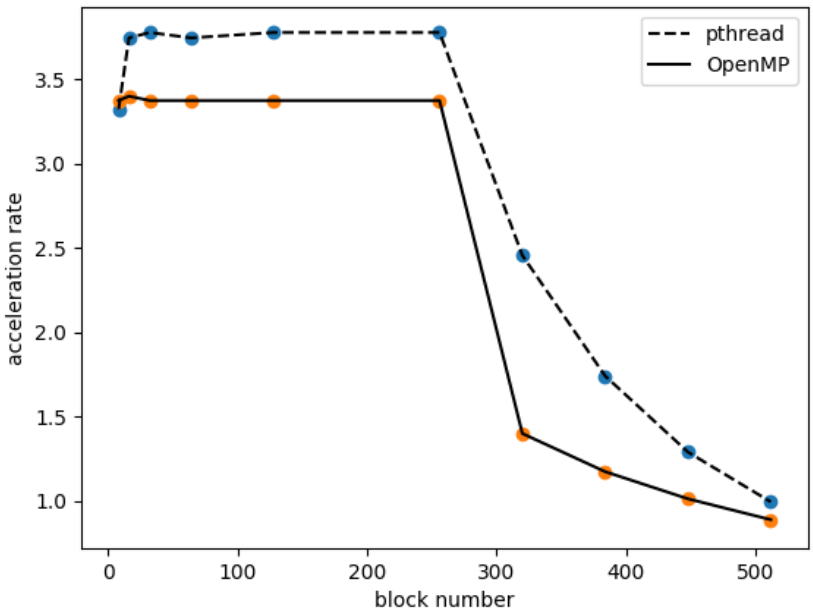

图 3.3: 加速比随线分块大小变化

实验四　基于 MPI 的形态学图像处理

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 实现形态图像处理操作的并行算法
3. 对程序执行结果进行简单的分析和总结
4. 将其与 Lab2 和 Lab3 的结果进行比较

4.2　算法描述

对于 MPI 而言，由于其编程框架与 OpenMP 是不同的，因此需要使用不同的方法。由于在计算蚀刻以及膨胀时要使用一个像素周围的所有像素，如图4.1所示，因此如果使用`MPI_Scatter`进行数据的分散则需要考虑分块边缘重叠的部分，实际上加大了程序编写的复杂度，而 OpenMP 和 pthread 程序由于是基于内存共享模型的因此不存在这一问题。而且由于需要对于重叠的部分进行额外的复制，若使用 Scatter 进行分散因此实际上降低了程序的性能。出于以上的考虑，直接使用 Bcast 将整张图片进行广播。

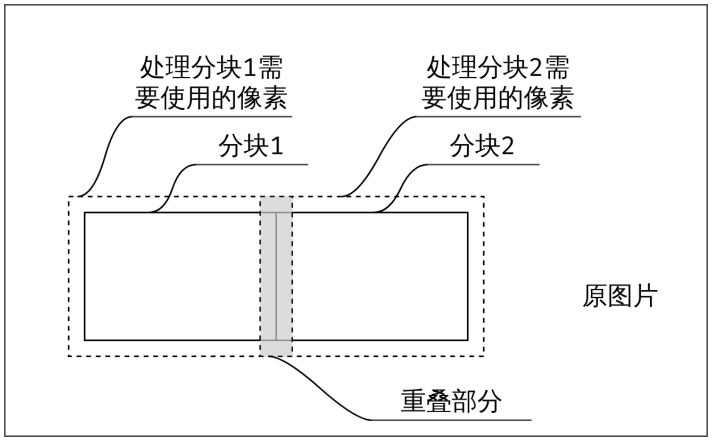

图 4.1: 两个分块的重叠部分

将整张图片进行广播后，每个进程根据总的进程数量处理自己的分块，而新的分块方式如图4.2所示，在原图片的 y 方向上进行分块，除了主进程外每一个进程取与自己的编号相同的分块进行处理，最后使用`MPI_Gather`对于处理的部分进行回收与拼接。由于在广播阶段广播整张图片，因此在处理时不需要考虑边缘重叠问题，此外，考虑到进程间数据传递的效率较线程间数据传递的效率低，因此不使用一个进程处理多个分块的方式，而是直接处理一个较大的分块直到处理完成。由于各个进程处理的数据量几乎一致，且处理逻辑是同质的，因此预计各个进程的处理时间不会相差太多。

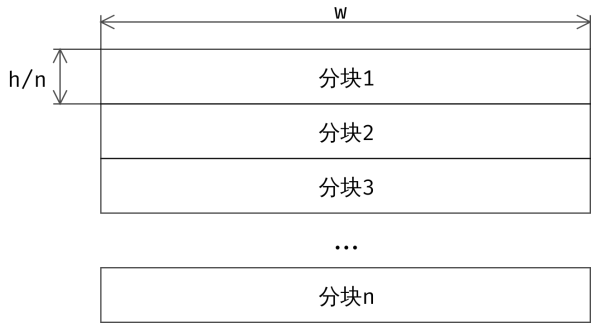

图 4.2: 新的分块方式

与 pthread 类似，在使用 MPI 进行并行处理时需要有一个主线程等待其他所有线程的完成，起到调度的作用。

4.3　实验方案

依据实验思路进行代码的编写后编译进行测试。需要注意的是编译需要使用 OpenMPI 的包装编译器 mpic++ 进行编译，在运行时也需要使用 mpirun 运行。多次改变进程数量后与前两次实验进行对比。

4.4　实验结果与分析

使用 4 进程运行的结果如图4.3所示，此输出在 4 个线程下运行得到。可以看到，程序运行时间为 15.9s，相对于串行的加速比为 $44.2 \div 15.9 = 2.77$，由于主线程不参与计算，因此这一结果与理想加速比 3 是比较接近的。

对于其他进程数而言，由于有一个主进程，因此至少需要两个进程才能完整运行程序，而由于进程数不能超过物理内核数，因此实际测试的进程数只有 2、3、4 这 3 种情况，除掉主进程，有效进程数分别为 1、2、3，与 pthread 和 OpenMP 进行对比的结果如图4.4所示。可以看到其与 pthread 的性能相差无几。

```
round 986 finished
round 987 finished
round 988 finished
round 989 finished
round 990 finished
round 991 finished
round 992 finished
round 993 finished
round 994 finished
round 995 finished
round 996 finished
round 997 finished
round 998 finished
round 999 finished
~/H/Parallel  dev *  lab4     15.9s  Sun Jul 15 15:27:55 2018
~/H/Parallel  dev *  lab4  history | head -n 1
mpirun -np 4 ./lab4
~/H/Parallel  dev *  lab4           Sun Jul 15 15:28:02 2018
```

图 4.3: MPI 程序输出

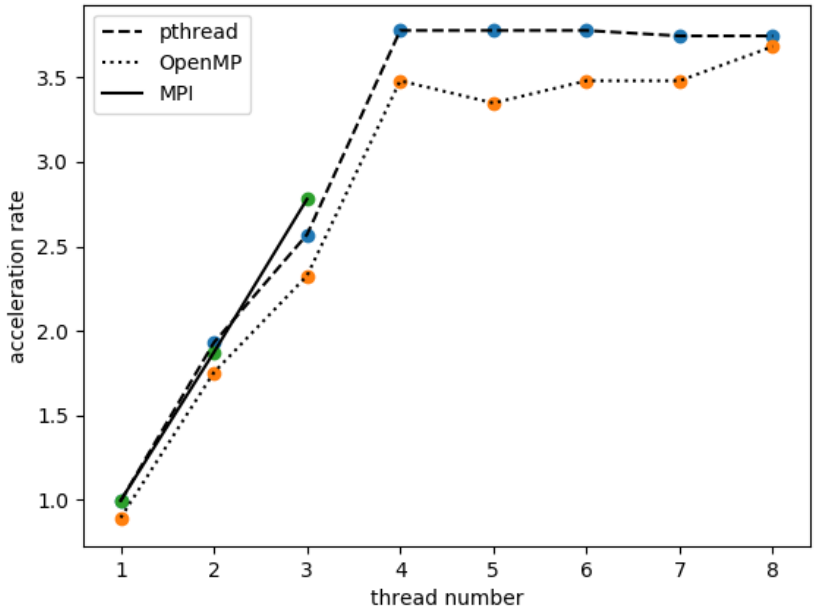

图 4.4: MPI 程序与 OpenMP、pthread 程序的对比结果

实验五　基于 CUDA 的形态学图像处理

5.1　实验目的与要求

1. 深入理解 GPGPU 的架构并掌握 CUDA 编程模型
2. 使用 CUDA 实现形态图像处理操作的并行算法
3. 对程序执行结果进行简单的分析和总结
4. 根据执行结果和硬件环境提出优化解决方案
5. 将其与 Lab2，Lab3 和 Lab4 的结果进行比较

5.2　算法描述

与上述并行算法不同的是，CUDA 使用的是异构的并行算法，由于 GPU 本身就比较适合进行图像处理，因此预计相对于串行算法的加速比会比较高。同样，按照 pthread 实验中的方法对于图片进行分块，然后将每一个图片块分配给不同的 CUDA block，将块中的不同像素分配给不同的 CUDA Thread 进行处理。如图5.1所示，线程的处理过程与算法 ErodeAndDilate 相同。

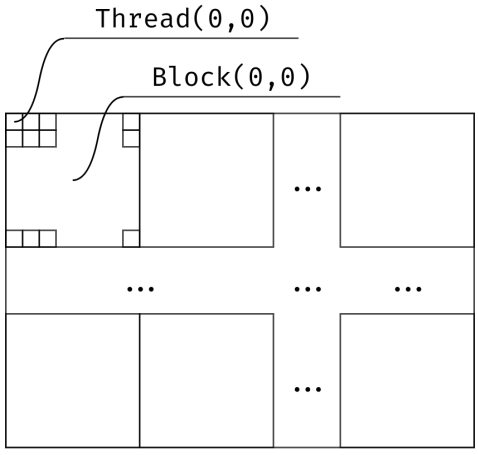

图 5.1: CUDA 块与线程分配

5.3　实验方案

编写 CUDA 程序并进行测试，在编译时需要使用 CUDA 的包装编译器 nvcc 进行编译，，在运行时需要保证 Nvidia 驱动已被正确加载。

5.4　实验结果与分析

图像处理的结果与串行图像处理的结果相同，性能方面，程序在使用分块大小为 128，每个像素使用一个线程处理的情况下，处理时间如图5.2所示，可以看出，程序只使用了 989ms 就完成了 1000 次处理，相对与并行程序加速比达到了 44.7，由此可见 CUDA 是较为适合并行处理图片的。

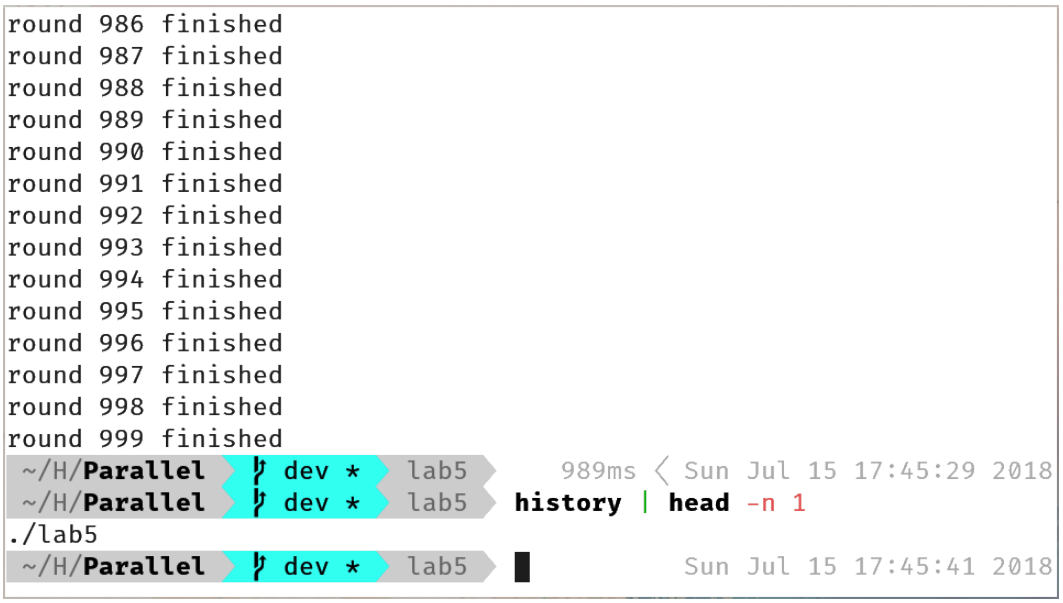

图 5.2: CUDA 程序运行结果

由于 CUDA 架构与仅使用 CPU 并行的架构完全因此不同，因此不具备与 CPU 程序在性能上的可比性。在不同的块大小情况下，CUDA 程序的最优时间为 686ms，相对于串行程序的加速比达到了 64.31，由此可见，对于数据易于分块，数据间无依赖的简单数据使用 GPU 进行处理占有绝对优势。

实验六　基于并行回溯算法的数独求解算法

6.1　实验目的与要求

1. 掌握并行化和改进程序的方法
2. 了解并行粒度与性能之间的关系
3. 掌握如何分割数据和分解复杂算法的任务

6.2　算法描述

数独的规则见https://zh.wikipedia.org/zh-sg/%E6%95%B8%E7%8D%A8。一个未求解和已经进行求解的数独如图6.2和6.1所示。

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

图 6.1: 未求解的数独

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

图 6.2: 已经求解的数独

首先，需要一个串行的数独求解程序进行参考，串行的程序使用回溯的方法对于数独进行求解，同样，为了便于时间的统计，因此需要增加求解的数据量，此时，对于 100 个数独进行求解。

然后需要设计一个并行的数独求解程序。由于数独前后的逻辑依赖性较强，因此不便于进行并行。在本实验中采用的数独求解方法为并行化的测试一个 cell 中的所有 case，以求获得最小的

可能的 case 的时间。此时，case 的选择可以在不同的遍历层级进行。递归的层级越高则粒度越低。

6.3　实验方案

按照算法描述中的算法编写 OpenMP 程序，在小于特定层级时将所有的可能性如队列，然后在这一特定的层级使用 OpenMP 进行并行，在大于这一层级不再使用并行算法，以避免线程数量指数爆炸。

需要注意的是在一个线程找到结果后需要及时停止其他的线程，而在停止时不可以使用 C++ exception，经过性能测试，exception 带来的开销约为 20%，极大的开销会为程序带来负优化。线程的错误处理一定因此通过返回值的方式进行。

6.4　实验结果与分析

串行程序解决 100 个数独的结果如图6.3所示。数独题目由 qqwing 程序生成，所有答案经过验证均正确。从图中可以看出，串行的程序需要使用 25.8s 进行求解。

```
8 7 6 2 5 4 9 3 1
9 2 4 1 6 3 8 5 7
1 5 3 7 8 9 6 2 4

Puzzle 98 solved.

1 2 3 9 8 6 5 4 7
8 9 4 7 2 5 1 3 6
7 5 6 3 1 4 8 2 9
5 4 7 1 3 2 6 9 8
2 6 1 5 9 8 4 7 3
9 3 8 6 4 7 2 1 5
4 8 5 2 7 3 9 6 1
6 7 9 4 5 1 3 8 2
3 1 2 8 6 9 7 5 4

Puzzle 99 solved.
~/H/Parallel  dev …  project1 ▉      25.8s  Sun Jul 15 20:34:27 2018
```

图 6.3: 串行数独求解

在递归的第二层进行 4 线程并行的时间如图6.4所示，从图中可以看出，需要 2.96s 进行求解，相比于串行程序，加速比为 8.71，达到了较为理想的加速比。

加速比与粒度的关系如图6.5所示。可以看出，粒度越细加速比越低，这是由于前边的并行部分增大以及线程竞争引起的。

```
8 7 6 2 5 4 9 3 1
9 2 4 1 6 3 8 5 7
1 5 3 7 8 9 6 2 4

Puzzle 98 solved.

1 2 3 9 8 6 5 4 7
8 9 4 7 2 5 1 3 6
7 5 6 3 1 4 8 2 9
5 4 7 1 3 2 6 9 8
2 6 1 5 9 8 4 7 3
9 3 8 6 4 7 2 1 5
4 8 5 2 7 3 9 6 1
6 7 9 4 5 1 3 8 2
3 1 2 8 6 9 7 5 4

Puzzle 99 solved.
~/H/Parallel  dev …  project1 2964ms 〈 Sun Jul 15 20:36:35 2018
```

图 6.4: 4 线程并行数独求解

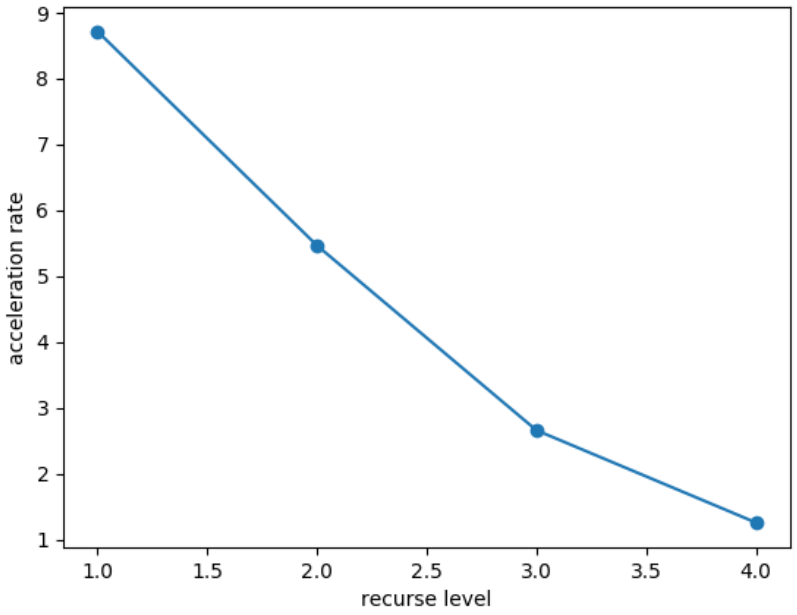

图 6.5: 加速比随粒度的变化关系

附录 A　开发与运行环境

所有 lab 的开发与运行的环境的平台、版本如表A.1所示。

表 A.1: 开发与运行环境

	项目	版本
软件	操作系统	ArchLinux, Kernel version 4.17.5-1
	gcc	8.1.1
	GNU Make	4.2.1
	OpenMP	6.0.1
	OpenMPI	3.1.0-1
	CUDA	9.2.148-1
	Nvidia driver	396.24-15
	shell	fish 2.7.1
硬件	CPU	Intel Core i7-7700K CPU @ 4.20GHz
	GPU	NVIDIA Corporation GP102 [GeForce GTX 1080 Ti]
	RAM	Kingston DDR4 16GiB (×2)

目录结构如下

```
./                          project根目录
├── README.md               README文件（本文件）
├── Makefile                主Makefile，用于同时编译/清除所有lab
├── data                    数据目录
│   ├── Lenna.png           lab1 ~ lab5 所用图片
│   └── questions.txt       lab6 所用数独数据
├── lab1                    串行形态学图像处理
│   ├── Makefile            lab1 Makefile
│   └── main.cpp            lab1 源码
├── lab2                    使用pthread加速的多线程图像处理
│   ├── Makefile            lab2 Makefile
│   └── main.cpp            lab2 源码
```

```
13  ├── lab3                    使用openmp加速的多线程图像处理
14  │   ├── Makefile            lab3 Makefile
15  │   └── main.cpp            lab3 源码
16  ├── lab4                    使用openmpi加速的多线程图像处理
17  │   ├── Makefile            lab4 Makefile
18  │   ├── main.cpp            lab4 源码
19  │   └── run.sh              openmpi 运行环境的包装，使用mpirun执行编译输出
20  ├── lab5                    使用cuda加速的多线程图像处理
21  │   ├── Makefile            lab5 Makefile
22  │   └── main.cu             lab5 源码
23  ├── project1                数独求解程序
24  │   ├── Makefile            lab6 Makefile
25  │   ├── parallel.cpp        使用openmp加速的并行数独求解程序源码
26  │   └── serial.cpp          串行数独求解程序源码
27  ├── lib                     第三方库文件夹
28  │   └── CImg-2.3.3          CImg图像处理库
29  │       └── ...             CImg图像处理库内容
30  └── report                  报告文件夹
31      └── ...                 报告内容
```

要正常的编译和运行，需要保证 gcc、nvcc 所在的目录在 $PATH 中。

要能够正确的编译并运行 lab5，需要满足下列要求：

- nvcc 在 $PATH 中
- PC 上装有支持 cuda 的 Nvidia 显卡
- 正确版本的 Nvidia 驱动以正确安装并 ** 已加载 **
- Nvidia 显卡有足够的显存以及足够的 Concurrent Thread 数目以支持程序的运行。

此外，lab2、lab3 可以使用 ‘-n <number>‘来指定线程数，使用 ‘-s <number>‘ 来指定图片分块大小 (推荐在 8 256 之间)；lab4 可以通过更改 ‘run.sh‘更改进程数，进程数必须大于 1，但不可超过 CPU 物理内核数。