哈爾濱工業大學

实验报告

实 验（三）

题　　目　优化(Optimize)

专　　业　人工智能

学　　号　2021112845

班　　级　2103601

学　　生　张智雄

指 导 老 师　郑贵滨

实 验 地 点　G.709

实 验 日 期　2023.4.16

计算学部

目 录

第 1 章 实验基本信息

1.1 实验目的

理解程序优化的 10 个维度

熟练利用工具进行程序的性能评价、瓶颈定位

掌握多种程序性能优化的方法

熟练应用软件、硬件等底层技术优化程序性能

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；2.30GHz；16G RAM；1.5THD disk

1.2.2 软件环境

Windows11 64 位；Vmware Workstation 17 Pro；Ubuntu 22.10

1.2.3 开发工具

Visual Studio 2019 64 位；CodeBlocks 64 位；vim+gcc

1.3 实验预习

上实验课前，必须认真预习实验指导书

了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。

程序优化的十个维度

如何编写面向编译器、CPU、存储器友好的程序。

性能测试方法：time、RDTSC、clock

性能测试准确性的文献查找：流水线、超线程、超标量、向量、多核、GPU、多级 CACHE、编译优化 Ox、多进程、多线程等多种因素对程序性能的综合影响。

第 2 章 实验预习

总分 20 分

2.1 程序优化的十大目标（5 分）

1. 更快（本课程重点!）
2. 更省（存储空间、运行空间）
3. 更美（UI 交互）
4. 更正确（本课程重点！各种条件下）
5. 更可靠
6. 可移植
7. 更强大（功能）
8. 更方便（使用）
9. 更规范（格式符合编程规范、接口规范 ）
10. 更易懂（能读明白、有注释、模块化）

2.2 性能优化的方法概述（5 分）

1. 一般有用的优化：

   代码移动、复杂指令简化、公共子表达式……。

2. 面向编译器的优化：障碍

   处理函数副作用、内存别名等障碍

3. 面向超标量 CPU 的优化：

   流水线、超线程、多功能部件、分支预测投机执行、乱序执行、多核：分离的循环展开!

   只有保持能够执行该操作的所有功能单元的流水线都是满的，程序才能达到这个操作的吞吐量界限

4. 面向向量 CPU 的优化：MMX/SSE/AVR
5. CMOVxx 等指令：

   代替 test/cmp+jxx

6. 嵌入式汇编
7. 面向编译器的优化：Ox:0 1 2 3 g

8. 面向存储器的优化：

   面向 Cache 重新排列提高空间局部性、分块提高时间局部性

9. 内存作为逻辑磁盘：内存够用的前提下。
10. 多进程优化：

    利用 fork 函数，使得每个进程负责各自的工作任务，并通过 mmap 共享内存或磁盘等进行交互。

11. 文件访问优化：带 Cache 的文件访问
12. 并行计算：

    第 12 章、多线程优化……

13. 网络计算优化：

    第 11 章、分布式计算、云计算……

14. GPU 编程、算法优化
15. 超级计算

2.3 Linux 下性能测试的方法（5 分）

Linux 下 Oprofile 等工具（gprof、google-perftools）

- https://blog.csdn.net/Blaider/article/details/7730792 用 OProfile 彻底了解性能
- https://www.cnblogs.com/jkkkk/p/6520381.html《Linux 调优工具 oprofile 的演示分析》
- https://www.cnblogs.com/MYSQLZOUQI/p/5426689.html

Linux 下的 valgrind：Callgrind/Cachegrind

- https://www.jianshu.com/p/1e423e3f5ed5 将 Cachegrind 和 Callgrind 用于性能调优
- https://blog.csdn.net/u010168781/article/details/84303954

2.4 Windows 下性能测试的方法（5 分）

Windows 下 VS，本身就有性能评测的组件

- 调试：性能探测器：CPU、RAM、GPU

第 3 章 性能优化的方法

总分 20 分
逐条论述性能优化方法名称、原理、实现方案（至少 10 条）

3.1 一般有用的优化

3.1.1 代码移动

原理：识别要执行多次（例如在循环里）但是计算结果不会改变的计算，将计算移动到代码前面不会被多次求值的地方，可以减少运行程序执行的总操作数，消除循环的低效率。

实现方案：将多次重复计算的值放在循环外，用一个新变量表示，也可使用-O1 编译器优化实现。

3.1.2 复杂指令简化

原理：相较于复杂指令系统计算机(CISC)方案，精简指令系统计算机(RISC)方案有一个精简的指令系统，指令数、寻址方式和指令格式种类少，又设有多个通用寄存器，采用流水线技术，从而提高了微理器的效率，但需要更复杂的外部程序。

实现方案：使用低开销的指令代替高开销的指令，提高效率，如利用位移代替乘除法等（但实际效果依赖于机器，取决于乘法或除法指令的成本）。

3.1.3 共享公用子表达式

原理：重用表达式的一部分，替换为变量，可以减少重复计算的次数。

实现方案：代码实现或者用-O1 编译器优化

3.2 面向编译器的优化：障碍

3.2.1 处理函数副作用

原理：函数每次被调用可能改变全局变量/状态，对于给定的参数，函数返回值依赖于全局状态/变量的其他部分，且可能与其他函数相互作用。

实现方案：使用 inline 内联函数（局限于单个文件），代码移动

3.2.2 内存别名

原理：两个不同的内存引用指向相同的位置，修改可能出现时间或者逻辑上的问题。

实现方案：引入局部变量，省去中间变量存储的方法减少这种影响

3.3 面向超标量 CPU 的优化

原理：一个时钟周期可发射、执行多条指令。只有保持能够执行该操作的所有功能单元的流水线都是满的，程序才能达到这个操作的吞吐量界限

实现方案：循环展开到某一程度K，并行累计A个，从而灌满流水线。但需注意不能超过执行单元的吞吐量限制。

3.4 面向向量 CPU 的优化：MMX/SSE/AVR

原理：使用新一代的专用指令与硬件资源来达到同样效果，新指令使用更新的专用硬件资源，具有更高的计算速度。

实现方案：利用 MMX/SSE/AVR 等指令集优化程序，可以利用底层硬件的运作方式，充分利用硬件，提高代码性能，如利用 cache 的按块读取优化程序。

3.5 面向存储器的优化

原理：cpu 访问 cache 的速度远远高于访问主存的速度，减少 cache 不命中的次数提高局部性能够显著提高程序性能。

实现方案：

1. 重新排列提高空间局部性：对多维数组进行步长为 1 的访问/存储，利用局部性原理，提高访存（cache）效率，有利于提高执行效率。
2. 分块提高时间局部性：将大区域分块计算，从内存中读入一个数据对象后，尽可能频繁/多地使用它。

3.6 嵌入式汇编

原理：汇编语言更接近底层，可以使用汇编语言通过对底层直接操作，防止编译器编译出一些冗繁的操作，省去一些不必要的过程。

实现方案：利用嵌入汇编的方法，人为地对底层进行优化。

3.7 多进程优化

原理：在操作系统层面上，进程是资源分配的最小单位。通过将这些应用程序分解

成可以准并行运行的多个顺序进程，程序设计模型会变得更简单。

实现方案：fork，每个进程负责各自的工作任务，通过 mmap 共享内存或磁盘等进行交互，提高执行效率。

3.8 文件访问优化

原理：cpu 访问读取外设存储信息速度远远慢于访问 cache。访问文件时先将文件内一大块数据加载到 cache 中，下次访问附近的数据时，先在磁盘 cache 中看下是否能够命中，如果命中了就不必再访问磁盘读取数据。

实现方案：带 Cache 的文件访问。对文件读写并不是每次都进行磁盘 I/O，而是将对应的磁盘文件缓存到内存上，之后对该文件的操作实际上也是对内存的读写。

3.9 网络计算优化

原理：将需要进行大量计算的项目数据分割成小块，由多台计算机分别计算，再上传运算结果后统一合并得出数据结论。

实现方案：分布式计算、云计算。

3.10 GPU 编程、算法优化

原理：利用 GPU 多核的特点，拥有更强的算力，可以更快的完成任务；相同的任务利用更优的算法，可以降低时间复杂度和空间复杂度

实现方案：利用 GPU 运行程序、改用更优算法。

第 4 章 性能优化实践

总分 60 分

4.1 原始程序及说明（10 分）

功能：一个图像处理程序实现图像的平滑，其图像分辨率为 1920*1080，每一点颜色值为 64bits，用 long img[1920][1080]存储屏幕上的所有点颜色值，平滑算法为任一点的颜色值为其上下左右 4 个点颜色的平均值，即：

$$img[i][j] = (img[i-1][j] + img[i+1][j] + img[i][j-1] + img[i][j+1])/4$$

流程：随机生成图像数据→执行平滑算法→记录时间。

```
#define WIDTH 1920
#define HEIGHT 1080
long img[WIDTH][HEIGHT];

int main()
{
    for(int i = 0; i <  WIDTH ; i++)
        for(int j = 0; j < HEIGHT; j++)
            {
                img[i][j] = rand();
```

图 4-1-1　Img 初始化

具体细节上，利用一个两行的数组缓存当前和上一行的原始数据，以免数据被错误覆盖，而此函数中并未考虑边缘点的处理。

```
void img_smooth1()   //程序未优化版本--未考虑边缘点处理
{
    clock_t start = clock();
    int i, j, m, n;
    long temp[2][HEIGHT];
    for(n = 0; n < HEIGHT ; n++)
        temp[1][n] = img[0][n];
    for(i = 1; i < WIDTH - 1; i++)
    {
        for(n = 0; n < HEIGHT ; n++)
        {
            temp[0][n] = temp[1][n];
            temp[1][n] = img[i][n];
        }

        for(j = 1; j < HEIGHT - 1; j++)
        {
            img[i][j] = (temp[1][j - 1] + img[i][j + 1] + temp[0][j] + img[i + 1][j]) / 4;
            //printf("%d,%d %d\n ",i+1,j+1,img[i][j]);
        }
    }
    clock_t end = clock();
    printf("Time cost: %d(ms)\n",(end - start) * 1000 / CLOCKS_PER_SEC);
}
```

图 4-1-2　未优化的图像平滑算法

用一个 5*5 的随机矩阵运行 demo 验证函数的正确性，结果如下：

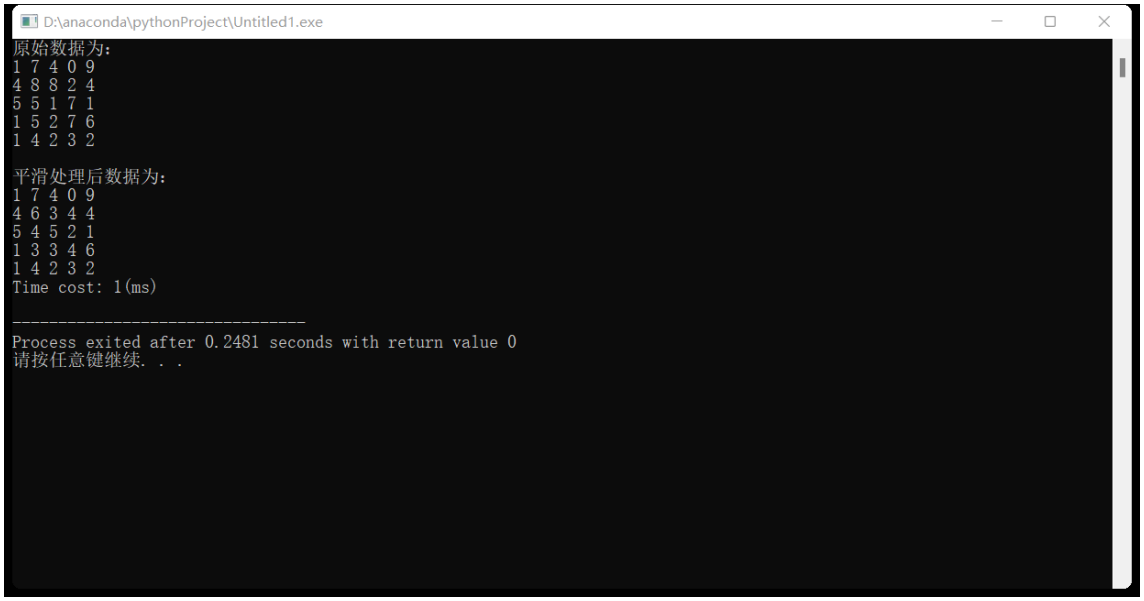

图 4-1-3　demo 运行示例

程序可能瓶颈：

1. 空间局部性已经较为良好，如何优化其时间局部性
2. 吞吐量的限制、CPU 内核数的限制等硬件约束对优化上限的影响

4.2 优化后的程序及说明（20 分）

本节共采用了 6 种优化方式，分别是一般有用的优化、面向 CPU 的优化策略、面向 Cache 的优化策略、多进程优化、多线程优化以及面向编译器的优化。

4.2.1 一般有用的优化

如图，使用低开销的逻辑右移指令代替平滑算法中高开销的除法指令：

```
60  void img_smooth2()   //一般有用的优化
61  {
62      clock_t start = clock();
63      int i, j, m, n;
64      long temp[2][HEIGHT];
65      for(n = 0; n < HEIGHT ; n++)
66          temp[1][n] = img[0][n];
67      for(i = 1; i < WIDTH - 1; i++)
68      {
69          for(n = 0; n < HEIGHT ; n++)
70          {
71              temp[0][n] = temp[1][n];
72              temp[1][n] = img[i][n];
73          }
74
75          for(j = 1; j < HEIGHT - 1; j++)
76          {
77              //用逻辑右移取代除法
78              img[i][j] = (temp[1][j - 1] + img[i][j + 1] + temp[0][j] + img[i + 1][j]) >> 2;
79              //printf("%d,%d %d\n ",i+1,j+1,img[i][j]);
80          }
81      }
82      clock_t end = clock();
83      printf("Time cost: %d(ms)\n",(end - start) * 1000 / CLOCKS_PER_SEC);
84  }
```

图 4-2-1　复杂指令简化

4.2.2 面向 CPU 的优化策略

首先，每次迭代平滑处理四个位置的数据，因而将需要的迭代次数减至原来的 1/4。

```
void img_smooth3()   //面向CPU的优化 循环展开
{
    clock_t start = clock();
    int i, j, k, m, n;
    long temp[2][HEIGHT];
    for(n = 0; n < HEIGHT ; n++)
        temp[1][n] = img[0][n];
    for(i = 1; i < WIDTH - 1; i++)
    {
        for(n = 0; n < HEIGHT ; n++)
        {
            temp[0][n] = temp[1][n];
            temp[1][n] = img[i][n];
        }

        for(j = 1; j < HEIGHT - 1; j += 4)
        {
            //循环展开
            img[i][j]     = (temp[1][j - 1] + img[i][j + 1] + temp[0][j]     + img[i + 1][j])     >> 2;
            img[i][j + 1] = (temp[1][j]     + img[i][j + 2] + temp[0][j + 1] + img[i + 1][j + 1]) >> 2;
            img[i][j + 2] = (temp[1][j + 1] + img[i][j + 3] + temp[0][j + 2] + img[i + 1][j + 2]) >> 2;
            img[i][j + 3] = (temp[1][j + 2] + img[i][j + 4] + temp[0][j + 3] + img[i + 1][j + 3]) >> 2;
            //printf("%d,%d %d\n ",i+1,j+1,img[i][j]);
        }
    }
    clock_t end = clock();
    printf("Time cost: %d(ms)\n",(end - start) * 1000 / CLOCKS_PER_SEC);
}
```

图 4-2-2-1 循环展开

而后，注意到同一行各个位置元素的计算时互不干扰的，因而可以每隔两个位置平滑处理一次，充分利用 CPU 流水线的并发性。

```
void img_smooth5()   //面向CPU的优化 循环展开+重组
{
    clock_t start = clock();
    int i, j, k, m, n;
    long temp[2][HEIGHT];
    for(n = 0; n < HEIGHT ; n++)
        temp[1][n] = img[0][n];
    for(i = 1; i < WIDTH - 1; i++)
    {
        for(n = 0; n < HEIGHT ; n++)
        {
            temp[0][n] = temp[1][n];
            temp[1][n] = img[i][n];
        }
        //重组
        for(j = 1; j < HEIGHT - 1; j += 8)
        {
            //循环展开
            img[i][j]     = (temp[1][j - 1] + img[i][j + 1] + temp[0][j]     + img[i + 1][j])     >> 2;
            img[i][j + 2] = (temp[1][j + 1] + img[i][j + 3] + temp[0][j + 2] + img[i + 1][j + 2]) >> 2;
            img[i][j + 4] = (temp[1][j + 3] + img[i][j + 5] + temp[0][j + 4] + img[i + 1][j + 4]) >> 2;
            img[i][j + 6] = (temp[1][j + 5] + img[i][j + 7] + temp[0][j + 6] + img[i + 1][j + 6]) >> 2;
            //printf("%d,%d %d\n ",i+1,j+1,img[i][j]);
        }
        for(k = 2; j < HEIGHT - 1; j += 8)
        {
            //循环展开
            img[i][j]     = (temp[1][j - 1] + img[i][j + 1] + temp[0][j]     + img[i + 1][j])     >> 2;
            img[i][j + 2] = (temp[1][j + 1] + img[i][j + 3] + temp[0][j + 2] + img[i + 1][j + 2]) >> 2;
            img[i][j + 4] = (temp[1][j + 3] + img[i][j + 5] + temp[0][j + 4] + img[i + 1][j + 4]) >> 2;
            img[i][j + 6] = (temp[1][j + 5] + img[i][j + 7] + temp[0][j + 6] + img[i + 1][j + 6]) >> 2;
            //printf("%d,%d %d\n ",i+1,j+1,img[i][j]);
        }
    }
    clock_t end = clock();
    printf("Time cost: %d(ms)\n",(end - start) * 1000 / CLOCKS_PER_SEC);
```

图 4-2-2-2 循环展开+重组

最后，注意到，计算表达式中存在许多共享子表达式，可以将其简化，减少不必要的重复计算，进一步优化程序性能。

```
153 void img_smooth4()   //面向CPU的优化 循环展开+重组+共享公用子表达式
154 {
155     clock_t start = clock();
156     int i, j, k, m, n;
157     long temp[2][HEIGHT];
158     for(n = 0; n < HEIGHT ; n++)
159         temp[1][n] = img[0][n];
160     for(i = 1; i < WIDTH - 1; i++)
161     {
162         for(n = 0; n < HEIGHT ; n++)  {temp[0][n] = temp[1][n];temp[1][n] = img[i][n];}
163         int z = i + 1; //公用子表达式
164         //重组
165         for(j = 1; j < HEIGHT - 1; j += 8)
166         {
167             //公用子表达式
168             int x1 = j + 1, x2 = j + 2, x3 = j + 3, x4 = j + 4, x5 = j + 5, x6 = j + 6;
169             //循环展开
170             img[i][j]  = (temp[1][j - 1] + img[i][x1]    + temp[0][j]     + img[z][j])  >> 2;
171             img[i][x2] = (temp[1][x1]    + img[i][x3]    + temp[0][x2]    + img[z][x2]) >> 2;
172             img[i][x4] = (temp[1][x3]    + img[i][x5]    + temp[0][x4]    + img[z][x4]) >> 2;
173             img[i][x6] = (temp[1][x5]    + img[i][j + 7] + temp[0][x6]    + img[z][x6]) >> 2;
174         }
175         for(k = 2; k < HEIGHT - 1; k += 8)
176         {
177             //公用子表达式
178             int y1 = k + 1, y2 = k + 2, y3 = k + 3, y4 = k + 4, y5 = k + 5, y6 = k + 6;
179             //循环展开
180             img[i][k]  = (temp[1][k - 1] + img[i][y1]    + temp[0][k]     + img[z][k])  >> 2;
181             img[i][y2] = (temp[1][y1]    + img[i][y3]    + temp[0][y2]    + img[z][y2]) >> 2;
182             img[i][y4] = (temp[1][y3]    + img[i][y5]    + temp[0][y4]    + img[z][y4]) >> 2;
183             img[i][y6] = (temp[1][y5]    + img[i][k + 7] + temp[0][y6]    + img[z][y6]) >> 2;
184         }
185     }
186     clock_t end = clock();
187     printf("Time cost: %d(ms)\n",(end - start) * 1000 / CLOCKS_PER_SEC);
188 }
```

图 4-2-2-3 循环展开+重组+公共子表达式

4.2.3 面向 Cache 的优化策略

结合上述程序可以发现重新排列循环顺序不能提高空间局部性，因而本节采用块来提高时间局部性，首先查看本机 Cache 相关信息。

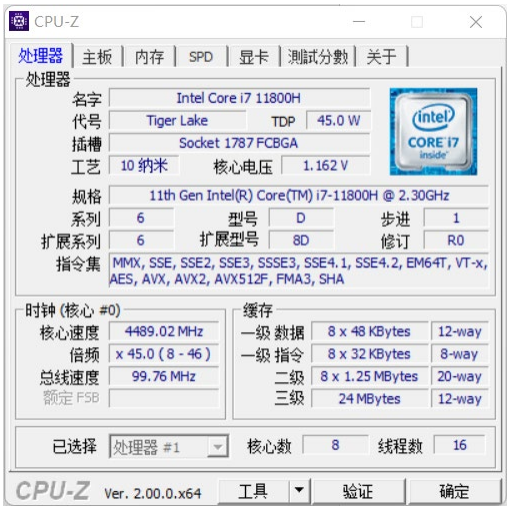

图 4-2-3-1 Cache 相关信息

可以看到本机 L1 D-Cache 是 12 路相连的，每行 64byte，每组可以存下多个 4*4 的矩阵（long 在 linux 64 位下为 8 字节，在 Windows 下为 4 字节）。取块大小为 4*4，缓存矩阵为 5*5 的一维数组，不考虑边缘处理，优化如下：

```
void img_smooth6()   //面向cache的优化
{
    int i0, j0, i, j, m, n;
    int T_w = 4, T_h = 4;              //分块信息
    long temp[(T_w+1)*(T_h+1)];              //记录原始信息
    for(i0 = 1; i0 < WIDTH - 1; i0 += T_w)
    {
        for(j0 = 1; j0 < HEIGHT - 1; j0 += T_h)
        {
            //以一个5*5的一维数组记录原始信息（左多一列，上多一行，不考虑边缘处理）
            for(i = i0 - 1; i < min(i0 + T_w , WIDTH); i++)
            {
                for(j = j0 - 1; j < min(j0 + T_h , HEIGHT); j++)
                {
                    temp[(i+1-i0)*(T_h+1)+j+1-j0] = img[i][j];
                }
            }
            //4*4分块计算
            for(i = i0; i < min(i0 + T_w, WIDTH - 1); i++)
            {
                for(j = j0; j < min(j0 + T_h, HEIGHT - 1); j++)
                {
                    //平滑算法
                    img[i][j] = (temp[(i+1-i0)*(T_h+1)+j-j0] + img[i][j + 1] + temp[(i-i0)*(T_h+1)+j-j0+1] + img[i + 1][j]) / 4;
                    //printf("左%d,右%d,上%d,下%d %d\n ",temp[(i+1-i0)*(T_w+1)+j-j0],img[i][j + 1],temp[(i-i0)*(T_w+1)+j],img[i + 1][j],i
                }
            }
        }
    }
}
```

图 4-2-3-1　分块提高时间局部性

4. 2. 4 多进程优化

利用 fork()创建子进程，让子进程负责矩阵上半部分的平滑处理，父进程负责下半部分的平滑处理，从而并行提高执行效率。

```
void img_smooth7()   //多进程优化
{
    clock_t start = clock();           //记录开始时间
    int i, j, m, n;
    long temp[2][HEIGHT];              //记录原始信息
    int pid = fork();
    if(pid == 0)    //子进程处理矩阵的上半部分
    {
        for(n = 0; n < HEIGHT ; n++)
            temp[1][n] = img[0][n];
        for(i = 1; i < WIDTH / 2; i++)
        {
            for(n = 0; n < HEIGHT ; n++)
            {
                temp[0][n] = temp[1][n];
                temp[1][n] = img[i][n];
            }

            for(j = 1; j < HEIGHT - 1; j++)
            {
                //平滑算法
                img[i][j] = (temp[1][j - 1] + img[i][j + 1] + temp[0][j] + img[i + 1][j]) / 4;
                printf("%d,%d %d\n ",i+1,j+1,img[i][j]);
            }
        }
        exit(0);
    }

    else           //父进程处理矩阵的下半部分
    {
```

```
        for(n = 0; n < HEIGHT ; n++)
            temp[1][n] = img[WIDTH / 2 - 1][n];
        for(i = WIDTH / 2; i < WIDTH - 1; i++)
        {
            for(n = 0; n < HEIGHT ; n++)
            {
                temp[0][n] = temp[1][n];
                temp[1][n] = img[i][n];
            }

            for(j = 1; j < HEIGHT - 1; j++)
            {
                //平滑算法
                img[i][j] = (temp[1][j - 1] + img[i][j + 1] + temp[0][j] + img[i + 1][j]) / 4;
                printf("%d,%d %d\n ",i+1,j+1,img[i][j]);
            }
        }
    }

    clock_t end = clock();
    printf("Time cost: %d(ms)\n",(end - start) * 1000 / CLOCKS_PER_SEC);
}
```

图 4-2-4　双进程优化

4. 2. 5 多线程优化

利用 pthread 创建多个进程并发运行，函数如下图：

```
void img_smooth7()   //多线程优化
{
    int i;
    pthread_t tid[PTHREAD_NUMBER];
    pthread_attr_t attr[PTHREAD_NUMBER];
    struct bound pth[PTHREAD_NUMBER];
    for(i = 0; i < PTHREAD_NUMBER; i++) {
        pth[i].l_bound = max(1, i*(WIDTH/PTHREAD_NUMBER));
        pth[i].r_bound = min(WIDTH - 1, (i+1)*(WIDTH/PTHREAD_NUMBER));
        pthread_attr_init(&attr[i]);
    pthread_create(&tid[i], &attr[i], smooth, (void*)&pth[i]);

    }
    //回收线程
    for(i = 0; i < PTHREAD_NUMBER; i++) {
        pthread_join(tid[i], NULL);
    }
}
void *smooth(void *pt)
{
    struct bound *p = (struct bound*)pt;
    int x = p->l_bound, y = p->r_bound;
    int i, j, m, n;
    long temp[2][HEIGHT];               //记录原始信息
    for(n = 0; n < HEIGHT ; n++)
        temp[1][n] = img[x-1][n];
```

```
    for(n = 0; n < HEIGHT ; n++)
        temp[1][n] = img[x-1][n];
    for(i = x; i < y; i++){
        for(n = 0; n < HEIGHT ; n++){
            temp[0][n] = temp[1][n];
            temp[1][n] = img[i][n];
        }
        for(j = 1; j < HEIGHT - 1; j++){
            //平滑算法
            img[i][j] = (temp[1][j - 1] + img[i][j + 1] + temp[0][j] + img[i + 1][j]) / 4;
            //printf("%d,%d %d\n ",i+1,j+1,img[i][j]);
        }
    }
}
```

图 4-2-5　多线程优化

4.2.6 面向编译器的优化

利用 Visual Studio 编译器自带的优化（优选速度）Ox、最大优化（优选大小）O1 以及最大优化（优化速度）O2 对程序进行优化。

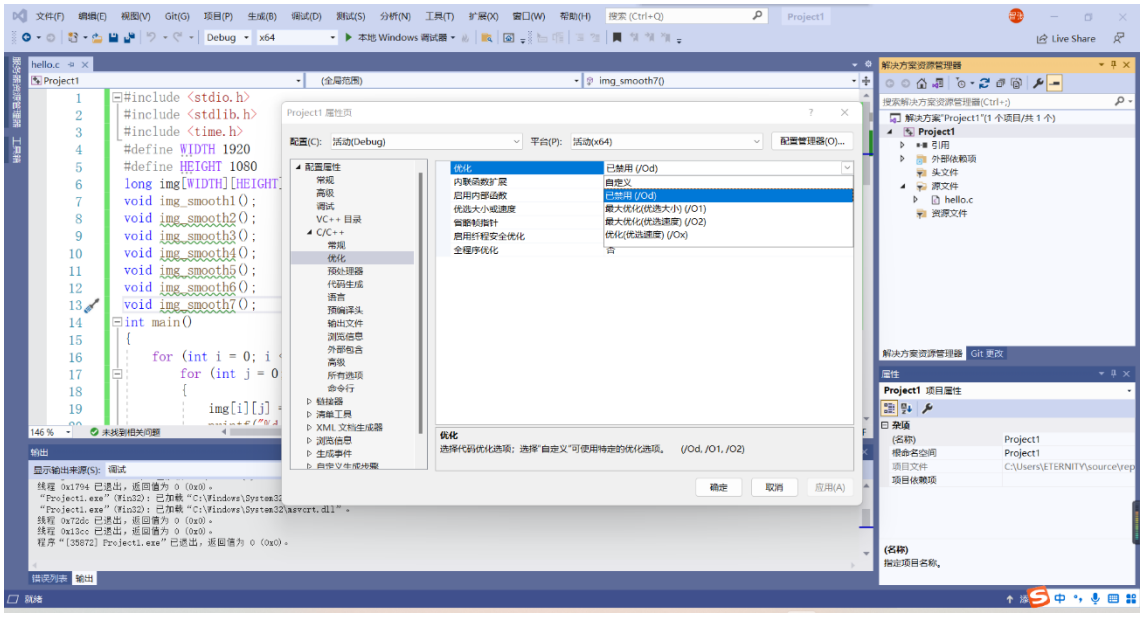

图 4-2-6　Visual Studio 编译器优化

4.3 优化前后的性能测试（10 分）

测试方法：为方便观测比较重复执行函数 100 次，测量比较函数的执行时间，具体实现上，利用函数指针减少代码冗余，同时利用 c 语言自带时钟进行计时，时间单位为毫秒（ms），函数如下：

```
18  void test(void (*func)()) {
19      clock_t start = clock();
20      int t;
21      for(t = 0; t < TESE_NUMBER; t++)
22          func();
23      clock_t end = clock();
24      printf("测试时间: %d(ms)\n",(end - start) * 1000 / CLOCKS_PER_SEC);
25  }
```

图 4-3-1　测试函数

在 Linux 上测试结果如下：

```
zzxiong@zzxiong-virtual-machine:~/xxxx/lab3$ gcc optimize.c -o optimize
zzxiong@zzxiong-virtual-machine:~/xxxx/lab3$ ./optimize
未优化 测试时间：763(ms)
一般有用的优化 测试时间：793(ms)
循环展开 测试时间：746(ms)
循环展开+重组 测试时间：754(ms)
循环展开+重组+子表达式 测试时间：726(ms)
面向Cache分块 测试时间：1936(ms)
多进程优化 测试时间：813(ms)
多线程优化 测试时间：790(ms)
```

图 4-3-2　测试结果

观察测试结果发现，

1. 一般有用的优化的优化效果并不明显，说明在本机上乘法或除法指令的成本低于右移的成本；
2. 而循环展开对程序有较明显的优化，而在此基础上加入累积量或者公共子表达式后时间反而增加了，推测程序受吞吐量的限制，且申请变量内存的时间大于重复计算公共子表达式的时间；
3. 而利用 Cache 分块耗费的时间大幅增加的原因可能是分块选择不合理，Cache 不命中的次数较多，经重复实验，发现 80*64 的分块效果较好，但仍远大于未优化时的耗费时间，推测其根本原因可能是在本问题中，除边界像素点外，其余像素点均需要访问 3 次，没有多次重复使用的值，而多嵌套两层循环带来的额外时空消耗大于分块带来的时间局部性优化。

```
面向Cache分块80*64 测试时间：1387(ms)
```

4. 多线程的结果优于多进程，但二者优化效果均不明显，可能原因是进程创建所需的内存和资源分配非常昂贵，对于本问题中较简单的计算并行计算的优化小于其创建进程或线程耗费的代价。

在 Visual Studio 编译器优化后，可以发现程序性能大幅提高，结果如下：

图 4-3-3　编译器未优化结果

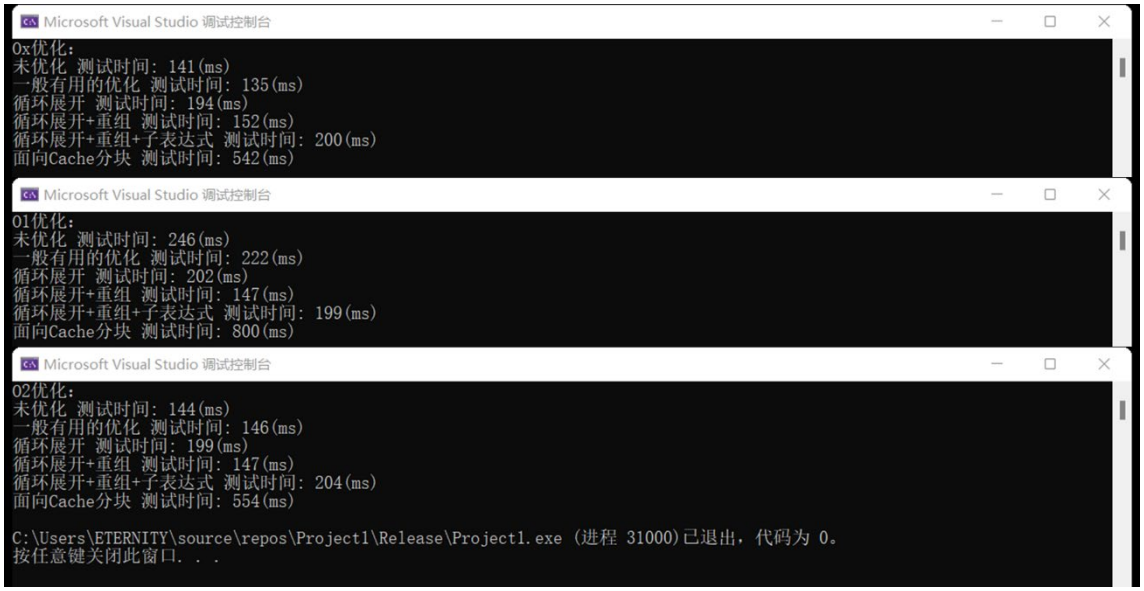

图 4-3-4　编译器优化结果

4.4 面向泰山服务器优化后的程序与测试结果（15 分）

面向泰山服务器优化后的程序均在上述 4.2 节中优化程序基础上仅修改参数，故本部分不在截图叙述，仅对结果进行分析。

4.4.1 多线程优化

根据泰山服务器 CPU 有 96 个物理核，于是尝试多线程并发运行，发现多线程并不比单线程性能更好，这是由于多线程由于轮换使用 CPU，会造成单个线程的执行速度变慢。而且在上下文切换的时候，频繁切换线程也会造成一定的时间浪费。

图 4-4-1　多线程测试结果

4.4.2 面向 CPU 的优化

经测试，循环展开和重组均有一定优化效果，其中展开因子为 6 时、不使用循环累积量的优化效果最好。

图 4-4-2　面向 CPU 测试结果

4.4.3 面向 Cache 的优化

仍然，在本程序中，空间局部性利用得已经较为充分，对不同分块下进行测试，结论同 4.3 节几乎一致，多嵌套两层循环带来的额外时空消耗大于分块带来的时间局部性优化，因此分块很难提高图像平滑算法的性能，以下是不同分块情况下对性能测试的结果：

```
不分块 测试时间: 2578(ms)
面向Cache分块4*4 测试时间: 5492(ms)
面向Cache分块16*16 测试时间: 4221(ms)
面向Cache分块32*32 测试时间: 4359(ms)
面向Cache分块80*64测试时间: 4038(ms)
面向Cache分块128*128测试时间: 4012(ms)
面向Cache分块1*1080（按行读取）测试时间: 4782(ms)
stu2021112845@node210:~/Lab3$
```

图 4-4-3　面向 Cache 测试结果

4.5 还可以采取的进一步的优化方案（5 分）

1. 运用 MMX/SSE/AVR 进行面向向量 CPU 的优化
2. 利用汇编语言进行嵌入式汇编
3. 对算法进一步优化
4. 将内存作为逻辑磁盘
5. 使用 CMOVxx 等指令代替 test/cmp+jxx

第 5 章 总结

5.1 请总结本次实验的收获

1. 学会了并实践多种程序优化的方法，提高程序的执行效率。
2. 熟悉了存储器、CPU 流水线等硬件的工作原理，并能够针对其硬件特点制定较为合理的优化策略。
3. 对循环展开等方法有了更深的理解。
4. 能够借助资料以及已学知识分析优化方法失效的原因。

5.2 请给出对本次实验内容的建议

1. 希望能增加对各种优化方法及适用条件更为细致的讲解。
2. 增加对目标算法更详细的描述以及适当的演示。

注：本章为酌情加分项。

参考文献

[1]RANDALE.BRYANT, DAVIDR.O'HALLARON. 深入理解计算机系统[M]. 机械工业出版社, 2011.