

哈尔滨工业大学

实验报告

实验（三）

题 目 优化

专 业 计算学部

学 号 120L020701

班 级 2003005

学 生 董琦

指 导 教 师 吴锐

实 验 地 点 G712

实 验 日 期 2022/4/15

计算学部

目 录

第 1 章 实验基本信息	- 4 -
1.1 实验目的	- 4 -
1.2 实验环境与工具	- 4 -
1.2.1 硬件环境	- 4 -
1.2.2 软件环境	- 4 -
1.2.3 开发工具	- 4 -
第 2 章 实验预习	- 5 -
2.1 程序优化的十大方法 (5 分)	- 5 -
2.2 性能优化的方法概述 (5 分)	- 5 -
2.2.1 一般优化方法	- 5 -
2.2.2 面向编译器的优化	- 5 -
2.2.3 面向超标量 CPU 的优化	- 5 -
2.2.4 面向向量 CPU 的优化: MMX/SSE/AVR	- 6 -
2.2.5 面向存储器的优化	- 6 -
2.2.6 多进程优化	- 6 -
2.2.7 其他	- 6 -
2.3 LINUX 下性能测试的方法 (5 分)	- 6 -
2.4 WINDOWS 下性能测试的方法 (5 分)	- 6 -
第 3 章 性能优化的方法	- 7 -
3.1 减少过程调用	- 7 -
3.2 循环展开	- 7 -
3.3 使用临时变量	- 7 -
3.4 公共子表达式消除	- 7 -
3.5 提高并行性	- 7 -
3.6 使用多线程	- 7 -
3.7 使用合适的数据结构和算法	- 8 -
3.8 分支预测	- 8 -
3.9 利用局部性	- 8 -
3.10 使用向量指令	- 8 -
第 4 章 性能优化实践	- 9 -
4.1 原始程序及说明 (10 分)	- 9 -
4.2 优化后的程序及说明 (20 分)	- 10 -
4.3 优化前后的性能测试 (10 分)	- 12 -
4.3.1 测试方法	- 12 -
4.3.2 测试结果	- 12 -

4.4 优化方法分析（15 分）	- 13 -
4.4.1 成功的方法	- 14 -
4.4.2 失败的尝试	- 14 -
4.5 还可以采取的进一步的优化方案（5 分）	- 14 -
第 5 章 总结	- 15 -
5.1 请总结本次实验的收获	- 15 -
5.2 请给出对本次实验内容的建议	- 15 -
参考文献	- 16 -

第 1 章 实验基本信息

1.1 实验目的

- 1.理解程序优化的 10 个维度
- 2.熟练利用工具进行程序的性能评价、瓶颈定位
- 3.掌握多种程序性能优化的方法
- 4.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

处理器 Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz 8 核
机带 RAM 16.0 GB (15.8 GB 可用)
系统类型 64 位操作系统, 基于 x64 的处理器

1.2.2 软件环境

Windows:

版本 Windows 11 家庭中文版

版本 21H2

安装日期 2022/2/4

操作系统版本 22000.556

体验 Windows 功能体验包 1000.22000.556.0

Ubuntu:

版本 Ubuntu 20.04.3 LTS

类型 64 位

1.2.3 开发工具

Windows: Visual Studio 2019

Ubuntu: VScode gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04)

第 2 章 实验预习

总分 20 分

2.1 程序优化的十大方法（5 分）

- 1.更快：加快运行速度；
- 2.更省：节省内存占用、运行资源占用；
- 3.更美：美化 UI 交互界面；
- 4.更正确：提高得到结果的正确性；
- 5.更可靠：增强鲁棒性；
- 6.可移植：提高兼容性，可以在不同平台下运行；
- 7.更强大：增加程序功能；
- 8.更方便：简化使用方法；
- 9.更规范：格式符合编程规范、接口规范；
- 10.更易懂：代码模块化，命名贴合用处，编写注释、文档。

2.2 性能优化的方法概述（5 分）

2.2.1 一般优化方法

- 1.代码移动；
- 2.复杂指令简化；
- 3.公共子表达式；
- 4.减少过程调用。

2.2.2 面向编译器的优化

- 1.使用寄存器变量：消除不必要的内存引用；
- 2.使用 O1, 2, 3；
- 3.由于函数副作用和内存别名的原因，编译器采用保守的优化策略，所以个人编写代码时候应该根据目的采取高效的代码。

2.2.3 面向超标量 CPU 的优化

- 1.乱序处理器；
- 2.使用条件传送风格的代码；

- 3.缩短循环的关键路径;
- 4.循环展开。

2.2.4 面向向量 CPU 的优化: MMX/SSE/AVR

2.2.5 面向存储器的优化

- 1.重新排列提高空间局部性;
- 2.分块提高时间局部性。

2.2.6 多进程优化

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。

2.2.7 其他

CPU 编程、算法优化、并行计算等等。

2.3 Linux 下性能测试的方法 (5 分)

使用 Oprofile\valgrind 等工具。

2.4 Windows 下性能测试的方法 (5 分)

使用 VS 自带的性能探测器。还可以使用 clock 等一些库函数编写代码。

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 减少过程调用

原理：将函数调用替换为函数体。这样可以减少对函数调用的开销，并且还可以对展开的代码做进一步优化。

实现方案：将函数调用直接改为相应的函数体即可。

3.2 循环展开

原理：在增加循环的步数，在一次循环中执行多次动作。这样可以减少条件判断时的开销，进而优化程序。

实现方法：增加每次迭代计算的元素的数量，减小循环的迭代次数。

3.3 使用临时变量

原理：使用临时变量来储存迭代过程中的累积量，这样可以减少对目的变量的读写次数，减少不必要的储存器引用。

实现方法：声明一个临时变量来进行循环迭代，迭代结束后再写回目的变量。

3.4 公共子表达式消除

原理：提取不同计算中的公共子表达式，减少计算开销。

实现方法：声明临时变量来储存中间计算步骤的结果值。

3.5 提高并行性

原理：流水线化的功能执行单元可以同时进行多条指令，通过提高代码的并行性，可以充分利用 CPU 的能力，进而提高执行速度。

实现方法：使上下代码之间没有相关性，进而可以并发执行。

3.6 使用多线程

原理：本质是提高并行性，在多核处理器的多个核上运行不同的线程。

实现方法：使用 pthread 等库函数来实现多线程。

3.7 使用合适的数据结构和算法

原理：数据类型和算法决定了一个函数的复杂度上下限，使用合适的算法才能从根本上优化程序。

实现方法：根据具体问题来选择最优的数据结构和算法。

3.8 分支预测

原理：采用好的分支预测方法，进而减少分支预测错误所带来的额外开销。

实现方法：linux 系统下可以使用 `__builtin_expect(long exp, long c)` 函数来告诉编译器分支倾向于走向何方。

3.9 利用局部性

原理：读取 cache 速度要快于低级储存器的速度，充分利用局部性可以使程序运行在高速 cache 上，从而优化程序速度。

实现方法：编写高速缓存友好的代码。尽量重复使用已经加载到高速缓存中的数据；在循环中最好采用步长为 1 的方法等等。

3.10 使用向量指令

原理：CPU 的向量寄存器可以并行运算多个相同的数据加载、加减法等操作，可以大大提高 CPU 的运行速度。

实现方法：在代码中使用 AVX、SSE 等指令集。

第 4 章 性能优化实践

总分 60 分

4.1 原始程序及说明（10 分）

说明程序的功能、流程，分析程序可能瓶颈

```
1.int main(int argc, char* argv[])
2.{
3.    srand((unsigned)time(NULL));
4.    long img[lenth][width];
5.    for (int i = 0; i < lenth; i++)
6.        for (int j = 0; j < width; j++)
7.            img[i][j] = rand() % 10000;
8.    LARGE_INTEGER num;
9.    long long start, end, freq;
10.    QueryPerformanceFrequency(&num);
11.    freq = num.QuadPart;
12.    QueryPerformanceCounter(&num);
13.    start = num.QuadPart;
14.    for(int k=0;k<20;k++)
15.        process(img);
16.    QueryPerformanceCounter(&num);
17.    end = num.QuadPart;
18.    printf("total time=%lld\n", (end - start) * 1000 / freq);
19.    return 0;
20.}
```

这段是 main 函数，用于生成初始数组和计算时间，由于时间较小，故重复 20 次。

```
1. //不处理边界像素点
2. void process(long long img[lenth][width])
3. {
4.     long long buf[width];
5.     for (int i = 0; i < lenth; i++)
6.         buf[i] = img[0][i];
7.     for (int j = 1; j < width - 1; j++)
8.         for (int i = 1; i < lenth - 1; i++)
9.         {
10.            long long temp = (img[i - 1][j] + img[i + 1][j] + img[i][j - 1] +
img[i][j + 1]) / 4;
11.            img[i - 1][j] = buf[j];
12.            buf[j] = temp;
13.        }
14.     for (int i = 0; i < width; i++)
15.         img[lenth - 2][i] = buf[i];
16. }
```

要优化的函数块，流程就是用一个二重循环来计算像素点的平均值。为了防止像素点信息丢失的问题，使用一个 buf 数组来储存上一行已处理的像素点信息，在处理本行时更新（如果立即更新，那么当前像素点的信息将丢失）。

程序主要时间占用在内部的二重循环内，需要取内存中四个点的值。由于四

个点位置相差较大，取值时很可能出现不命中情况，需要从低级缓存中加载，消耗较多时间。故这部分是程序运行时间的瓶颈。

4.2 优化后的程序及说明（20 分）

至少包含面向 CPU、Cache 的两种优化策略（20 分），额外每增加 1 种优化方法加 5 分至第 4 章满分。

```

1. void process(long long img[lenth][width])
2. {
3.     long long buf[width];
4.     register long long* imge = (long long*)img;
5.     for (register unsigned int i = 0; i < width; i++)
6.     {
7.         buf[i] = img[0][i];
8.     }
9.     register int row_bd = lenth - 1;
10.    register int col_bd = width - 1;
11.    int row_size = sizeof(long long) * width;
12.    long long buf2[width];
13.    for (register unsigned int i = 1; i < row_bd; i++)
14.    {
15.        long long * bias1 = imge+i * width;
16.        for (register unsigned int j = 1; j < col_bd; j+=14)
17.        {
18.            int m = j;
19.            int n = j + 1;
20.            register long long * bias2 = bias1 + m;
21.            memcpy(buf2+m, bias1 + width+m, 14*8);
22.            {register long long temp1 = ((buf[m] + buf2[m]) + (*(bias2 - 1) +
23.            *(bias2 + 1))) >> 2;
24.            buf[m] = *(bias2);
25.            *bias2++ = temp1;
26.            m += 2;
27.            register long long temp2 = ((buf[n] + buf2[n]) + (*(bias2 - 1) +
28.            (bias2 + 1))) >> 2;
29.            buf[n] = *(bias2);
30.            *bias2++ = temp1;
31.            n += 2; }
32.            {register long long temp1 = ((buf[m] + buf2[m]) + (*(bias2 - 1) +
33.            *(bias2 + 1))) >> 2;
34.            buf[m] = *(bias2);
35.            *bias2++ = temp1;
36.            m += 2;
37.            register long long temp2 = ((buf[n] + buf2[n]) + (*(bias2 - 1) +
38.            (bias2 + 1))) >> 2;
39.            buf[n] = *(bias2);
40.            *bias2++ = temp1;
41.            n += 2; }
42.            {register long long temp1 = ((buf[m] + buf2[m]) + (*(bias2 - 1) +
43.            *(bias2 + 1))) >> 2;
44.            buf[m] = *(bias2);
45.            *bias2++ = temp1;
46.            m += 2;
47.            register long long temp2 = ((buf[n] + buf2[n]) + (*(bias2 - 1) +
48.            (bias2 + 1))) >> 2;
49.            buf[n] = *(bias2);
50.            *bias2++ = temp1;
51.            n += 2; }
52.            {register long long temp1 = ((buf[m] + buf2[m]) + (*(bias2 - 1) +
53.            *(bias2 + 1))) >> 2;
54.            buf[m] = *(bias2);
55.            *bias2++ = temp1;
56.            m += 2;
57.            register long long temp2 = ((buf[n] + buf2[n]) + (*(bias2 - 1) +
58.            (bias2 + 1))) >> 2;
59.            buf[n] = *(bias2);
60.            *bias2++ = temp1;
61.            n += 2; }
62.            {register long long temp1 = ((buf[m] + buf2[m]) + (*(bias2 - 1) +
63.            *(bias2 + 1))) >> 2;
64.            buf[m] = *(bias2);
65.            *bias2++ = temp1;
66.            m += 2;
67.            register long long temp2 = ((buf[n] + buf2[n]) + (*(bias2 - 1) +
68.            (bias2 + 1))) >> 2;
69.            buf[n] = *(bias2);
70.            *bias2++ = temp1;
71.            n += 2; }
72.            {register long long temp1 = ((buf[m] + buf2[m]) + (*(bias2 - 1) +
73.            *(bias2 + 1))) >> 2;
74.            buf[m] = *(bias2);
75.            *bias2++ = temp1;
76.            m += 2;
77.            register long long temp2 = ((buf[n] + buf2[n]) + (*(bias2 - 1) +
78.            (bias2 + 1))) >> 2;
79.            buf[n] = *(bias2);
80.            *bias2++ = temp1;
81.            n += 2; }
82.            {register long long temp1 = ((buf[m] + buf2[m]) + (*(bias2 - 1) +
83.            *(bias2 + 1))) >> 2;
84.            buf[m] = *(bias2);
85.            *bias2++ = temp1;
86.            m += 2;
87.            register long long temp2 = ((buf[n] + buf2[n]) + (*(bias2 - 1) +
88.            (bias2 + 1))) >> 2;
89.            buf[n] = *(bias2);
90.            *bias2++ = temp1;
91.            n += 2; }
92.            {register long long temp1 = ((buf[m] + buf2[m]) + (*(bias2 - 1) +
93.            *(bias2 + 1))) >> 2;
94.            buf[m] = *(bias2);
95.            *bias2++ = temp1;
96.            m += 2;
97.            register long long temp2 = ((buf[n] + buf2[n]) + (*(bias2 - 1) +
98.            (bias2 + 1))) >> 2;
99.            buf[n] = *(bias2);
100.           *bias2++ = temp1;
101.           n += 2; }
102.        }
103.    }
104.}

```

```

45.         n += 2; }
46.         {register long long temp1 = ((buf[m] + buf2[m]) + (*(bias2 - 1) +
    *(bias2 + 1))) >> 2;
47.         buf[m] = *(bias2);
48.         *bias2++ = temp1;
49.         m += 2;
50.         register long long temp2 = ((buf[n] + buf2[n]) + (*(bias2 - 1) + *
    (bias2 + 1))) >> 2;
51.         buf[n] = *(bias2);
52.         *bias2++ = temp1;
53.         n += 2; }
54.         {register long long temp1 = ((buf[m] + buf2[m]) + (*(bias2 - 1) +
    *(bias2 + 1))) >> 2;
55.         buf[m] = *(bias2);
56.         *bias2++ = temp1;
57.         m += 2;
58.         register long long temp2 = ((buf[n] + buf2[n]) + (*(bias2 - 1) + *
    (bias2 + 1))) >> 2;
59.         buf[n] = *(bias2);
60.         *bias2++ = temp1;
61.         n += 2; }
62.         {register long long temp1 = ((buf[m] + buf2[m]) + (*(bias2 - 1) +
    *(bias2 + 1))) >> 2;
63.         buf[m] = *(bias2);
64.         *bias2++ = temp1;
65.         m += 2;
66.         register long long temp2 = ((buf[n] + buf2[n]) + (*(bias2 - 1) + *
    (bias2 + 1))) >> 2;
67.         buf[n] = *(bias2);
68.         *bias2++ = temp1;
69.         n += 2; }
70.         {register long long temp1 = ((buf[m] + buf2[m]) + (*(bias2 - 1) +
    *(bias2 + 1))) >> 2;
71.         buf[m] = *(bias2);
72.         *bias2++ = temp1;
73.         m += 2;
74.         register long long temp2 = ((buf[n] + buf2[n]) + (*(bias2 - 1) + *
    (bias2 + 1))) >> 2;
75.         buf[n] = *(bias2);
76.         *bias2++ = temp1;
77.         n += 2; }
78.     }
79. }
80. long long * bias1 = imge+(lenth - 2) * width;
81. for (register int j = 1; j < col_bd; j++)
82. {
83.     long long * bias2 = bias1 + j;
84.     *(bias2) = ((buf[j] + *( bias2 + width)) + (*(bias2 - 1) + *( bias2 +
    1))) >> 2;
85. }
86. }

```

使用的优化方法:

1. 面向 cache: ①: 改变循环的方式: 使用步长为 1 的循环方法, 利用空间局部性。②: 开辟了俩行 buf 数组, 用于缓存计算时用到的上下俩行元素。③: 将循环指示变量声明为 register 类型。
2. 面向 CPU: 在循环展开相邻的两个代码块中使用了不同临时变量, 使之没有相关, 提高了并行性。
3. 循环展开: 展开了 14 次。

4. 使用了公共子表达式：大幅减少了在通过数组索引取值时的计算量，提高了程序效率。

4.3 优化前后的性能测试（10 分）

4.3.1 测试方法

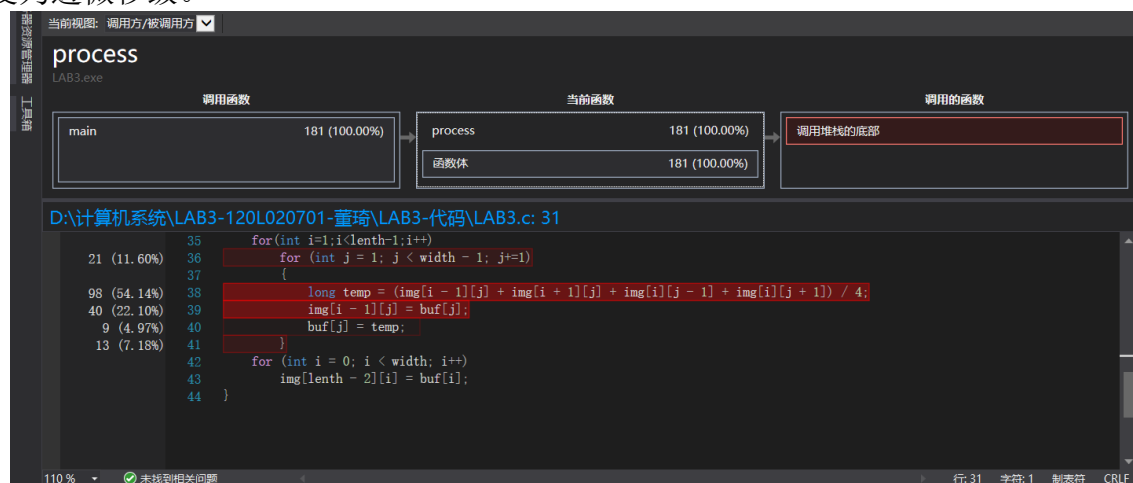
```

LARGE_INTEGER num;
long long start, end, freq;
QueryPerformanceFrequency(&num);
freq = num.QuadPart;
QueryPerformanceCounter(&num);
start = num.QuadPart;
for(int k=0;k<20;k++)
    process(img);
QueryPerformanceCounter(&num);
end = num.QuadPart;
printf("total time=%lld\n", (end - start) * 1000 / freq);

```

使用 window.h 中的方法。在进行定时之前，先调用 QueryPerformanceFrequency() 函数获得机器内部定时器的时钟频率，然后在需要严格定时的事件发生之前和发生之后分别调用 QueryPerformanceCounter() 函数，利用两次获得的计数之差及时钟频率，计算出事件经历的精确时间。

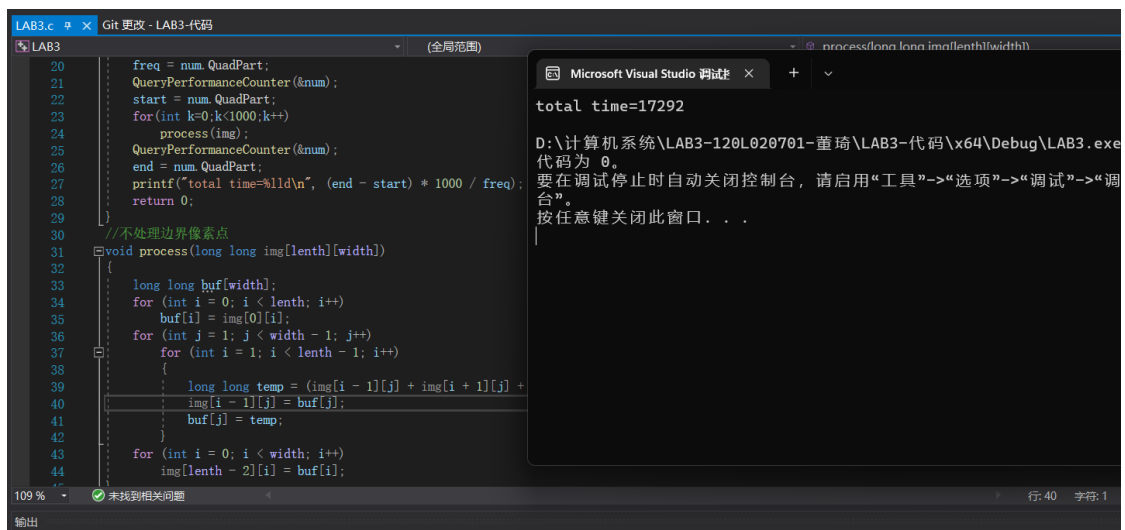
这种方法的定时误差不超过 1 微秒，精度与 CPU 等机器配置有关，一般认为精度为透微秒级。



VS 性能探测器，可以看到函数体各语句的占时情况。

4.3.2 测试结果

优化前：1000 次重复运行，时间大概在 17000 左右（故意写了很差 QAQ）。



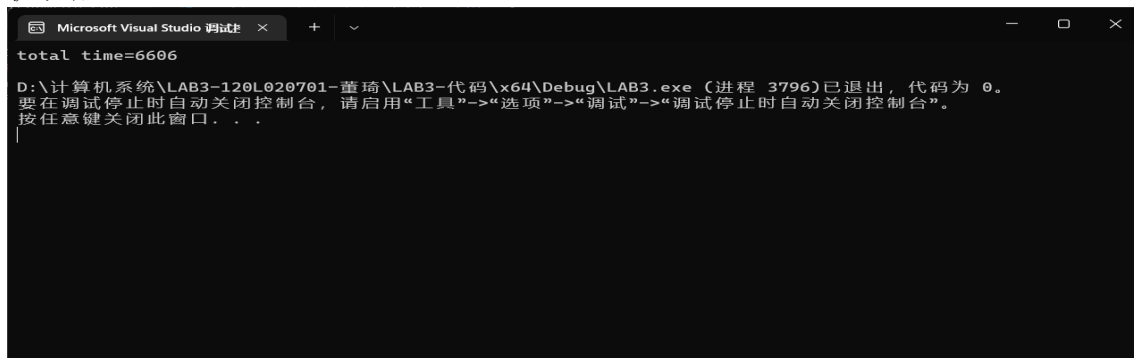
```
LAB3.c  Git 更改 - LAB3-代码
(全局范围)
20      freq = num.QuadPart;
21      QueryPerformanceCounter(&num);
22      start = num.QuadPart;
23      for (int k=0; k<1000; k++)
24          process(img);
25      QueryPerformanceCounter(&num);
26      end = num.QuadPart;
27      printf("total time=%lld\n", (end - start) * 1000 / freq);
28      return 0;
29  }
30  //不处理边界像素点
31  void process(long long img[lenth][width])
32  {
33      long long buf[width];
34      for (int i = 0; i < lenth; i++)
35          buf[i] = img[0][i];
36      for (int j = 1; j < width - 1; j++)
37          for (int i = 1; i < lenth - 1; i++)
38          {
39              long long temp = (img[i - 1][j] + img[i + 1][j] +
40                  img[i - 1][j] + buf[j]);
41              buf[j] = temp;
42          }
43      for (int i = 0; i < width; i++)
44          img[lenth - 2][i] = buf[i];
45  }
109 %  未找到相关问题  行: 40  字符: 1
输出
```

Microsoft Visual Studio 调试

total time=17292

D:\计算机系统\LAB3-120L020701-董琦\LAB3-代码\x64\Debug\LAB3.exe
代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试台”。
按任意键关闭此窗口...

优化后：



Microsoft Visual Studio 调试

total time=6606

D:\计算机系统\LAB3-120L020701-董琦\LAB3-代码\x64\Debug\LAB3.exe (进程 3796) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...

4.4 优化方法分析（15 分）

这个是我的电脑参数：



这个是代码时间占用分析：

```

D:\计算机系统\LAB3-120L020701-董琦\LAB3-代码\LAB3.c: 39
3 (0.04%) 52      long long * bias1 = image1 * width;
1152 (16.72%) 53      memcpy(buf2, bias1*width, row_size);
41 (0.59%) 54      for (register unsigned int j = 1; j < col_bd; j+=14)
55      {
26 (0.38%) 56          int m = j;
2 (0.03%) 57          int n = j + 1;
54 (0.78%) 58          register long long * bias2 = bias1 + m;
59          //register long long * bias3 = bias2 + width;
60
507 (8.66%) 61          (register long long templ = ((buf[m] + buf2[m]) + (*(bias2 - 1) + *(bias2 + 1))) >> 2;
70 (1.02%) 62          buf[m] = *(bias2);
154 (2.23%) 63          *bias2++ = templ;
41 (0.59%) 64          m += 2;
65          register long long * bias3 = ((buf[m-1] + buf2[m-1]) + (*(bias2-1) + *(bias2+1))) >> 2;

```

4.4.1 成功的方法

首先我使用了公共子表达式的方法，大大减少了数组索引时所需的计算，然后尽量使用加法，因为加法完全流水线化，可以并发执行，故程序在计算上所花时间占比较小。

然后我使用了循环展开，多达 14 次，已达到循环展开所能提升的上限。进一步缩短了运行时间。但是由于在循环过程中，没用关键路径上的相关数据链，故无法进一步展开。

之后我使用了较多临时变量，减少了循环展开块中的代码相关性，意在提高程序的并行性。不过效果甚微，估计是早已达到了 CPU 同时运行指令的上限，异或是间隔指令数太多，不会并发执行。

再然后面向 cache。首先我更改了循环访存方式，变为步长为 1 的方法，这一步大大提高了空间局部性，大幅缩短了函数消耗时间。

这时我检查函数，发现用时最多的部分在数组元素的读写上面，于是思考如何在这上面改进。初始时我使用了一个行缓存，于是我再添加了一个行缓存，用于缓存之后一行。然后程序稍有优化，从 7000 优化到了 6600 左右。

4.4.2 失败的尝试

1.分线程：使用 pthread 库，分了四个线程来执行程序。结果消耗时间翻了数倍。

原因分析：程序规模较小，每个线程执行的动作较少，然后在向内核请求线程这个过程中消耗了较多时间，故优化无效。

2.矩阵分块操作：尝试使用了矩阵分块处理，结果为负优化，且分块越大，消耗时间越少，当不分块时消耗时间最少，故分块操作纯为负优化。

原因分析：仔细检查了一下 L1 缓存的大小，发现 L1 缓存大小为 32kb，完全可以装下三行数组。故所有分块操作都降低了命中率，完全顺序执行便为最优的方式。但是当矩阵规模变大后，L1 无法将所需的三行都缓存下来时，进行分块操作应该可以正优化。

4.5 还可以采取的进一步的优化方案（5 分）

1.使用向量指令集。

第 5 章 总结

5.1 请总结本次实验的收获

复习了 CPU、储存器等方面的知识，对于程序的优化有了直观的认识和初步经验，更加熟悉并深入理解了课堂知识。

5.2 请给出对本次实验内容的建议

实验内容设计略有些不合理，难以用到书上的 $k \times n$ 等优化方式，且没有具体的实验目标，在执行过程中容易陷入迷茫，不知道需要做到何种程度。

注：本章为酌情加分项。

参考文献

- [1] Kernighan B W, Ritchie D M. The C Programming Language[M]. 2. Dennis Ritchie & Bell Labs, / June 2018.
- [2] Bryant R E, David R. O'Hallaron. Computer Systems a Programmer's Perspective[M]. 3rd. Carnegie Mellon University, 2016.