

# 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 120L021809

班 级 2003006

学 生 刘雨尘

指 导 教 师 吴锐

实 验 地 点

实 验 日 期

计算学部

## 目 录

第 1 章 实验基本信息 .....	- 3 -
1.1 实验目的 .....	- 3 -
1.2 实验环境与工具 .....	错误！未定义书签。
1.2.1 硬件环境 .....	错误！未定义书签。
1.2.2 软件环境 .....	错误！未定义书签。
1.2.3 开发工具 .....	- 3 -
1.3 实验预习 .....	- 3 -
第 2 章 实验预习 .....	- 4 -
2.1 程序优化的十大方法（5 分） .....	- 4 -
2.2 性能优化的方法概述（5 分） .....	- 4 -
2.3 LINUX 下性能测试的方法（5 分） .....	- 5 -
2.4 WINDOWS 下性能测试的方法（5 分） .....	- 5 -
第 3 章 性能优化的方法 .....	- 6 -
第 4 章 性能优化实践 .....	- 8 -
第 5 章 总结 .....	- 12 -
5.1 请总结本次实验的收获 .....	- 12 -
5.2 请给出对本次实验内容的建议 .....	- 12 -
参考文献 .....	- 13 -

## 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

基于 X64 的处理器，64 位操作系统，2.9GHZ，16GRAM，512G Disk

#### 1.2.2 软件环境

Windows10 64 位，VMWare 16，Ubuntu 16.04

#### 1.2.3 开发工具

VS Code 64 位；Codeblocks 64 位；vim+gcc；

### 1.3 实验预习

1. 上实验课前，必须认真预习实验指导书
2. 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。
3. 请写出程序优化的十个维度
4. 如何编写面向编译器、CPU、存储器友好的程序。
5. 性能测试方法：time、RDTSC、clock
6. 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 O<sub>3</sub>、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 程序优化的十大方法（5 分）

1. 更快
2. 更省（存储空间、运行空间）
3. 更美（UI 交互）
4. 更正确
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
  - (1) 代码移动
  - (2) 复杂指令简化
  - (3) 公共子表达式
2. 面向编译器的优化：障碍
  - (1) 函数副作用
  - (2) 内存别名
3. 面向超标量 CPU 的优化
  - (1) 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
  - (2) 只有保持能够执行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 O<sub>x</sub>:0 1 2 3 g
8. 面向存储器的优化：Cache

- (1) 重新排列提高空间局部性
- (2) 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
  - 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。
- 11. 文件访问优化：带 Cache 的文件访问
- 12. 并行计算：多线程优化：第 12 章
- 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）

使用 Oprofile 等工具（gprof、google-perftools）

linux 下使用 valgrind:callgrind/Cachegrind

## 2.4 Windows 下性能测试的方法（5 分）

VS 自带性能调试组件：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

#### 1. 代码移动

原理：对于需要重复调用并且不改变值的函数，可以从循环中移出。

实现方案：利用变量储存函数的值，将该值放入循环中，可以减少函数的调用次数。

#### 2. 公共子表达式

原理：对于一些表达式，尤其是标记数组序号的变量，如  $i+1$  等，可能被多次使用，每次使用时会额外进行加法计算等，占用了 CPU 的性能。

实现方案：使用变量提前对需要复用的表达式做出计算，并将这些变量带入，可以减少 CPU 执行运算的次数，从而优化性能。

#### 3. 复杂指令简化

原理：对于一些除法操作，使用除法器进行计算是比较耗时的，而除以 2、4 等操作可以简化为移位操作，会比进行除法计算快很多。

实现方案：利用左移两位的操作替换  $/4$  的操作。

#### 4. 消除内存别名使用

原理：在程序编译中，可能会遇到两个指针指向同一内存位置的情况，由于编译器只进行安全的优化，将会在化简指针调用方面倾向于保守的优化，从而限制 CPU 性能的提升。

实现方案：解决办法是尽可能少的同时使用多个指针，使用更高级的优化等级。

#### 5. 减少过程调用

原理：在循环计算过程中，可能会出现下一次循环的计算结果依赖于这一次循环的计算结果，如果计算的对象是储存在内存中的数组，那么可能出现将结果存入数组，再将结果从数组中取出的重复性操作。

实现方案：使用中间变量保存结果，一边存入数组指定位置，一边作为计算的参量参与计算，可以减少冗余的过程调用。

#### 6. 面向超标量 CPU 的优化

由于超标量 CPU 具有优秀的流水线性能，编写代码时可以倾向于使用流水线友好的代码，例如加法和乘法是完全流水线化的操作，把函数计算尽可能转化为乘法和加法可以在一定程度上进行性能的优化。具体操作方法是尽量少的使用同一个寄存器变量参与多次乘法与加法计算，这样的反复调用是依赖于寄存器数据的，只有当前指令执行完后才可以进行下一个指令。可以将计算分布在多个寄存器中。

### 7. 面向向量 CPU 的优化: SSE/AVX

原理: X86 指令只能实现一次处理一个数据, 而 SSE/AVX 指令集提供了一条指令同时处理多个数据的能力, 增强了单指令多数据的 CPU 处理性能, 采用 AVX 进行计算可以加快浮点数和整型数的运算效率, 提升程序性能。

实现方案: 加载数据成为 AVX 支持的向量, 使用向量进行加法移位等操作。

### 8. 面向编译器的优化:

原理及实现方案: 在默认情况下, 编译器倾向于保守的非积极的代码优化, 重新设置编译命令, 采用 O1/O2 或更高等级的优化, 可以在编译过程中自动完成流水线化或循环展开等操作, 从而实现程序性能优化。

### 9. 并行计算: 多线程优化

原理: 多线程 (multithreading), 是指从软件或者硬件上实现多个线程并发执行的技术。具有多线程能力的计算机因有硬件支持而能够在同一时间执行多于一个线程, 进而提升整体处理性能。

实现方法: 主要基于 Linux 使用 C 多线程。在编译 C 的多线程时候, 一方面必须指定 Linux C 语言线程库多线程库 pthread, 才可以正确编译 (例如: `gcc test.c -o test -lpthread`); 另一方面要包含有关线程头文件 `#include <pthread.h>`。

### 10. GPU 编程

原理: GPU 多 ALU 核, 可以进行并行处理计算, 并且线程较轻, 数量多, 可以实现大规模的并发, 同时占用少量 Cache. 利用 GPU 进行计算, 也可以优化程序的性能。

实现方法: CUDA 是 NVIDIA 发明的一种并行计算平台和编程模型。它通过利用图形处理器 (GPU) 的处理能力, 可大幅提升计算性能。利用 CUDA 编程即可实现。

## 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：实现一个图像处理程序以实现图像的平滑，其图像分辨率为 1920\*1080，每一点颜色值为 64b，用 long img[1920][1080] 存储屏幕上的所有点颜色值，颜色值可从 0 依行列递增，或真实图像。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： $\text{img}[i][j] = (\text{img}[i-1][j] + \text{img}[i+1][j] + \text{img}[i][j-1] + \text{img}[i][j+1]) / 4$ 。

流程：首先使用全局变量声明；long long img[1920][1080]，并按照从 0 递增方式初始化 img 变量。由于平滑函数使用的是原始数据取平均，所以需要缓存另外储存所得数据，在计算完成后覆盖初始数据。

可能存在的影响性能的因素：

- 循环过程中计算量不够大，未达到延迟界限且可以针对吞吐量优化；
- 除法运算相对比较耗时
- 初始代码中，访问矩阵使用顺序为列优先，内存不友好，每次访问都将出现内存不命中，空间局部性很差，需要进一步优化。

### 4.2 优化后的程序及说明（20 分）

面向 Cache：由于初始代码中，矩阵访问为列优先，内存访问不命中，空间局部性较差，现将列优先访问改为行优先访问，每次从次级内存中取出的数组元素都将在下一次循环中访问，优化了空间局部性，从而优化了程序性能。

面向 CPU：对于现代的超标量 CPU，使用循环展开技术，可以最大限度的利用 CPU 的吞吐量优势，在没有数据依赖的情况下，可以流水线式的进行循环内多个表达式的计算，从而大大提升程序性能。对于本题，首先采用了 4\*1 的循环展开方式。

面向向量 CPU：针对 X86 一条指令只能处理一个数据的不足，AVX 指令集有效的解决了多数据同时操作的优化问题。利用 AVX 指令集提供的函数进行 inline 指令进行函数内联，可以快速实现多数据流的计算。（同时也减少了内存存取的次数，可以视为对 Cache 的优化）

复杂指令简化：除以 4 的操作可以简化为右移两位。

公共子表达式：在循环展开过程中，出现了 j+1 等表达式多次复用的情况，利用单独的变量 m、n 等提前算出，并将变量代入计算。

针对

### 4.3 优化前后的性能测试（10 分）

利用 C 语言中的 clock 函数编写计时程序，计算函数运行时长，比较各函数性能。同时可以利用 VS 自带的性能探查器进行性能测试。

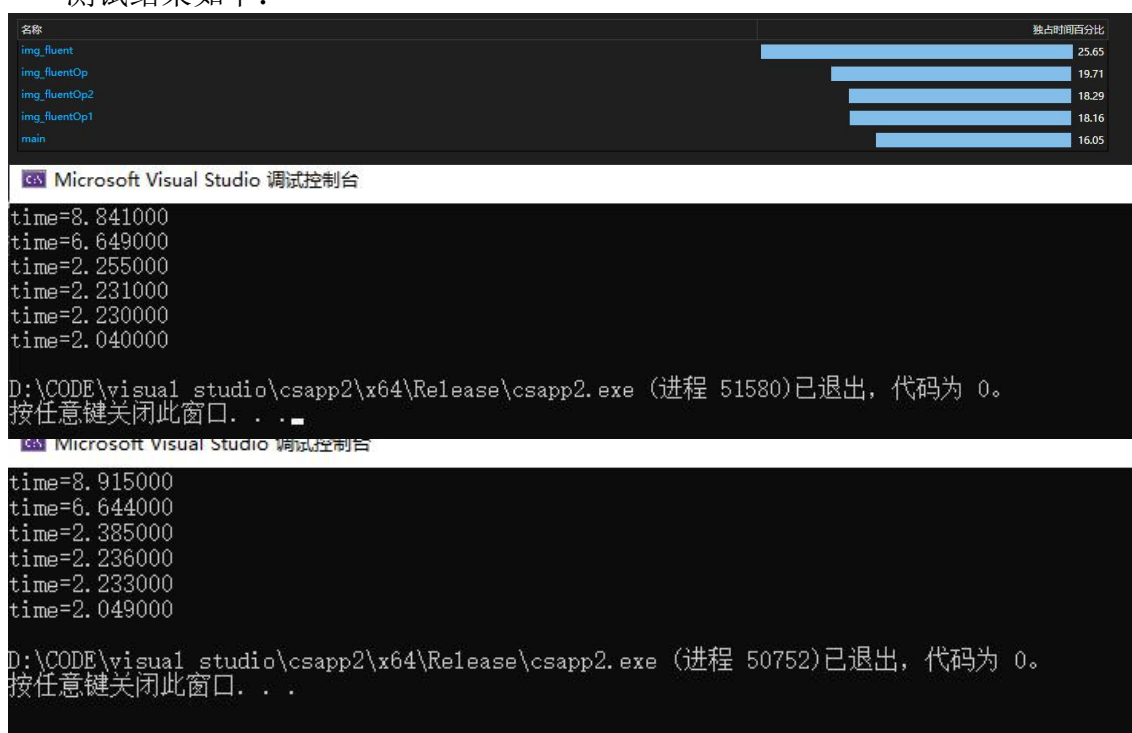


为了比较的更方便，我在实验中分别编写了 `image_fluent_`、`image_fluent`、`image_fluentOp`、`image_fluentOp1`、`image_fluentOp2`、`image_fluentOp3` 这 6 个函数。其中：

`image_fluent_` 采用了原始函数（列优先访问）；  
`image_fluent` 采用了行优先访问的 Cache 优化，增加了空间局部性；  
`image_fluentOp` 采用了 4\*1 循环展开的优化，更好的利用了 CPU 流水线优势；  
`image_fluentOp1` 在 `image_fluentOp` 的基础上采用移位代替除法操作；  
`image_fluentOp2` 在 `fluentOp1` 的基础上增加了公共子表达式；  
`image_fluentOp3` 使用了 AVX2 优化加法计算。

为了防止内存冷不命中，测试函数前预先运行了几遍函数使内存 warm up。所有的函数均采用运行 1000 次后的总时间。

测试结果如下：



可以明显观察到由 `fluent_` 到 `fluent` 函数运行时间减少了约 25%，说明良好的空间局部性极大的提升了程序运行速度；而由 `_fluent` 到 `fluentOp`，运行速度则提升了近三倍之多，而这是由于 4\*1 循环展开后，由于每个计算间并不存在数据依赖，CPU 可以流水线式的进行计算，很好的利用了 CPU 的吞吐量，同时也提高了程序运行的并行性；由 `fluentOp` 到 `fluentOp1`，移位操作代替除法带来了约 0.1s 的运行时间优化，也足以说明使用除法器计算是相当耗时间的；从 `fluentOp1` 到 `fluentOp2` 则增加了公共子表达式优化，但是尽管运行了 1000 次，为函数节省的时间也是 0.001s 数量级的，说明在当前函数中，可能由于公共子表达式的复用性不是非常高，优化效果并不明显；在 `fluentOp3` 中，对于所有的计算均采用 AVX2 指令代替，利用向量加法实现了一条指令执行多数据的操作，既减少了内存的存取次数，又加快了加法及移位操作的运行速度，并且一次性得到了 4 组结果，实现了约 0.2s 的优化。

#### 4.4 面向泰山服务器优化后的程序与测试结果（15 分）

面向泰山服务器进行优化时，由于 arm 架构不支持 AVX 优化，故从代码中删去。同时由于 1000 次循环在泰山服务器中较为耗费时间，采用单次的运行时间进行比较。由此可见 `fluent_` 函数到 `fluent` 函数的优化，速度快了近四倍，这既体现了保持良好空间局部性的重要性，也说明了泰山服务器的高速缓存存储器具有更大的容量，同时存取的速度对于程序的制约非常明显，才使得 `cache` 优化的效果如此优秀；`fluent` 函数到 `fluentOp` 函数，进行函数展开后，程序运行速度快了约三倍；而移位操作代替除法操作后，也有一定的优化效果，单次运行时间快了约 3ms；而公共子表达式优化效果并不理想，由于多使用了一些寄存器，程序运行速度反而变慢了一些。

```
stu_120L021809@node210: $ ./imageOp
time=0.033425
time=0.009249
time=0.003700
time=0.003432
time=0.003446
```

为了探究循环展开次数对于优化效果的影响，又对 `fluentOp1` 函数分别进行了 8\*1 展开和 12\*1 展开（按顺序依次为 8\*1 和 12\*1 的优化程序运行结果），可以看到相较于 4\*1 展开，8\*1 展开有较为明显的提升，而 12\*1 展开相较于 8\*1 展开提升的不是非常明显，可能达到了延迟界限。

```
stu_120L021809@node210: $ ./imageOp
time=0.033477
time=0.009361
time=0.003566
time=0.003282
time=0.003585
-----
stu_120L021809@node210: $ ./imageOp
time=0.033479
time=0.009456
time=0.003645
time=0.003375
time=0.003562
```

#### 4.5 还可以采取的进一步的优化方案（5 分）

##### 4.5.1 针对寄存器的优化方案

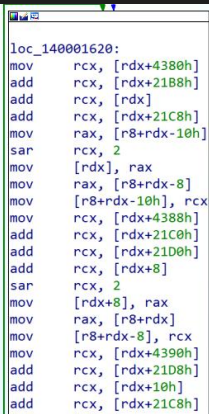


```
Microsoft Visual Studio 调试控制台
time=6.621000
time=2.331000
time=2.209000
time=2.220000
time=2.013000
D:\CODE\visual_studio\csapp2\x64\Release\csapp2.exe (进程 46600)已退出，代码为 0。
按任意键关闭此窗口。 . . .
```

在程序运行过程中，发现了 `Op2` 函数甚至显著慢于 `Op1` 的问题，这激发了我探究的兴趣，使用反汇编工具处理编译后的可执行文件得到汇编代码，观察代码，我发现函数中在反复不停的调用同一个寄存器 `rcx`，检查 C 语言代码发现函数中我

使用了单一变量 `cache` 在  $4 \times 1$  展开的优化函数中储存图像平滑函数的计算结果, 这种行为使循环展开中的计算出现了数据依赖, 损害了流水线友好的性质, 与公共子表达式调用寄存器之间相互作用, 减慢了函数的运行速度。于是下一步可以使用 `cache, cache1, cache2, cache3` 分别独立储存循环展开的计算结果, 增强函数对流水线的友好程度。

```
long long cache = (img[m][j] + img[n][j] + img[i][j - 1] + img[i][j + 1]) >> 2;
img[i - 1][j] = buffer[j];
buffer[j] = cache;
```



```
loc_140001620:
mov     rcx, [rdx+4380h]
add     rcx, [rdx+2188h]
add     rcx, [rdx]
add     rcx, [rdx+21C8h]
mov     rax, [r8+rdx-10h]
sar     rcx, 2
mov     [rdx], rax
mov     rax, [r8+rdx-8]
mov     [r8+rdx-10h], rcx
mov     rcx, [rdx+4380h]
add     rcx, [rdx+21C0h]
add     rcx, [rdx+21D0h]
add     rcx, [rdx+8]
sar     rcx, 2
mov     [rdx+8], rax
mov     rax, [r8+rdx]
mov     [r8+rdx-8], rcx
mov     rcx, [rdx+4390h]
add     rcx, [rdx+21D8h]
add     rcx, [rdx+10h]
add     rcx, [rdx+21C8h]
```

后续采用前述的优化方法后, 发现程序性能显著提升, 同时 Op2 较 Op1 有了较为明显的优化效果。

Microsoft Visual Studio 调试控制台

```
time=6.648000
time=2.250000
time=2.227000
time=2.225000
time=2.029000

D:\CODE\visual_studio\csapp2\x64\Release\csapp2.exe (进程 37708)已退出, 代码为 0。
按任意键关闭此窗口。 . . .
```

Microsoft Visual Studio 调试控制台

```
time=6.636000
time=2.339000
time=2.227000
time=2.208000
time=2.015000

D:\CODE\visual_studio\csapp2\x64\Release\csapp2.exe (进程 46976)已退出, 代码为 0。
按任意键关闭此窗口。 . . .
```

#### 4.5.2 针对并行计算的优化

C 语言的开始设计, 并未设计多线程的机制, 由于随着软硬件的发展及需求的发展。后来 C 语言才开发了线程库以支持多线程的操作、应用。主要基于 Linux 介绍 C 多线程。在编译 C 的多线程时候, 一方面必须指定 Linux C 语言线程库多线程库 `pthread`, 才可以正确编译 (例如: `gcc test.c -o test -lpthread`); 另一方面要包含有关线程头文件 `#include <pthread.h>`。

可以使用 `pthread` 库对 C 代码进行进一步的多线程优化。

## 第 5 章 总结

### 5.1 请总结本次实验的收获

本次实验中，着重面向 CPU 和 Cache 方向进行优化，在面向 Cache 优化的过程中，矩阵的先列再行读取和先行再列读取尽管代码差距很小，但是在运行 1000 次之后差距足足有近 3 秒，这使我认识到了缓存冲突不命中对于程序性能的影响是非常隐蔽而且非常大的，应当时刻注意保持良好的程序的时间局部性和空间局部性，将缓存不命中的概率降低到最小。

在面向 CPU 优化的过程中，使用 4\*1 循环展开的方式得到了近 3 倍的程序运行速度，这让我认识到增加程序运行的并行性，可以最大程度利用 CPU 的吞吐量性能，同时处理多条指令，加快程序的运行速度。

在传统 x86 指令集中，一条指令往往只能处理一个或两个数据，然而使用 avx 进行向量优化，可以同时最多处理 256 为数据，增大了数据流，从另一个角度提高了程序运行的并行性，并且达到了比循环展开还要快很多的速度。

综合来讲，本次实验让我认识到了关注程序运行性能的重要性，未经优化的程序与优化后的程序运行速度差距整整有四倍，对于小型代码，这种差距尚且令人震惊，对于需要长期运行的程序，如何细致地进行程序优化更是一件十分重要的事情。

### 5.2 请给出对本次实验内容的建议

对于本次实验等待优化的图像平滑函数，其实现 Cache 优化的方法比较少，如果可以通过设计函数，使得更多有关 cache 优化的方法得到应用就更好了。

同时由于学生的能力有限，希望老师可以在实验结束后给出一些相关的优化方法的实现作为眼界拓宽。

## 参考文献

- [1] <https://blog.csdn.net/zachariah2000/article/details/120731767>
- [2] [https://blog.csdn.net/weixin\\_42034217/article/details/113832051](https://blog.csdn.net/weixin_42034217/article/details/113832051)
- [3] <https://wenku.baidu.com/view/aba08e1d84c24028915f804d2b160b4e767f811c.html>
- [4] <https://www.runoob.com/w3cnote/gcc-parameter-detail.html>