中山大学计算机院本科生实验报告

（2024 学年秋季学期）

实验	**C++图像处理代码性能优化**	专业（方向）	**信息与计算科学**
学号	**22336313**	姓名	**郑鸿鑫**
Email	**zhenghx57@mail2.sysu.edu.cn**	完成日期	**2024/12/5**

1. 实验环境

本次实验采用 Linux 操作系统配置实验环境，详细参数如下：

```
n@XiaoxinPro:~/Optimize$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 24.04 LTS
Release:        24.04
Codename:       noble
```

编译器版本如下：

```
n@XiaoxinPro:~/Optimize$ g++ --version
g++ (Ubuntu 13.2.0-23ubuntu4) 13.2.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

2. 实验过程和核心代码

我们将原始版本的代码记为版本 0，编译运行得到结果如下：

```
n@XiaoxinPro:~/Optimize$ g++ -O0 -o Optimize_0 Optimize_0.cpp
n@XiaoxinPro:~/Optimize$ ./Optimize_0 2333
Checksum: 684550983
Benchmark time: 18385 ms
```

注：我们每次执行代码时指定 2333 为随机数种子来生成固定的图像。

a. 版本 1：优化数据结构来提高程序性能，具体过程如下：

将存储 figure 和 result 的二维 vector 修改为一维的 char 数组，然后在构造函数中进行对应动态内存分配和初始化，并在析构函数中增加对应的 delete 释放内存，最后所有访问修改为一维数组的索引方式。

优化说明：原始代码中，figure 和 result 是作为 vector<vector<unsigned char>>动态分配的。版本 1 的代码将它们改为使用单个 unsigned char*指针来分配一整块连续的内存。这样做的好处是减少了内存分配和访问的开销，因为 vector 在每次增加元素时可能会重新分配内存，这会导致额外的时间开销。同时，连续的内存布局对于缓存友好，可以提高缓存命中率。由于数据在内存中是连续存储的，CPU 缓存可以更有效地加载和存储数据，这提高了内存访问的局部性。

编译运行后得到结果如下：

```
n@XiaoxinPro:~/Optimize$ g++ -O0 -o Optimize_1 Optimize_1.cpp
n@XiaoxinPro:~/Optimize$ ./Optimize_1 2333
Checksum: 684550983
Benchmark time: 9021 ms
```

b. 版本 2：建立一个查找表来优化幂次变换，具体过程如下：

在类成员变量中增加图像尺寸 size 和一个 power 数组长度为 256 作为查找表，在构造函数中对查找表进行初始化（对下标作幂次变换后作为该下标索引的数组元素值），并对应修改幂次变换的函数逻辑，即只需通过下标索引而不再是重复计算。

优化说明：在成员变量中加入一个一维数组来存储 0~255 对应进行幂次变换之后的结果作为查找表，在实际的图像处理循环中，只需通过数组索引直接获取结果，而不需要进行浮点数的幂运算，大大加快了处理速度。

编译运行得到结果如下：

```
n@XiaoxinPro:~/Optimize$ g++ -O0 -o Optimize_2 Optimize_2.cpp
n@XiaoxinPro:~/Optimize$ ./Optimize_2 2333
Checksum: 684550983
Benchmark time: 5546 ms
```

c. **版本 3：使用 Pthread 来优化高斯滤波，具体过程如下：**

利用 Pthread 来并行化高斯滤波，首先定义线程参数结构体和线程执行函数 Internal_Gaussion。结构体存储了每个线程需要的参数，包括线程 ID、处理的起始和结束行索引、输入输出图像数据的指针以及图像尺寸。线程执行函数用于专门处理图像内部区域的高斯滤波，边界及四个角仍然保留单独处理逻辑。然后将高斯滤波函数中处理内部区域的逻辑对应修改为 Pthread 创建线程与分配任务，分配按照块分配，一个线程会分到矩阵的若干行的计算。

优化说明：通过创建多个线程并行处理任务，从而允许程序同时在多个处理器核心上执行，有效提高像高斯滤波这样的计算密集型任务的执行效率，减少总体处理时间。我们对高斯滤波的并行优化，是将图像按行分配，每一个线程独立处理分配到的对应行。这种方法使得每个处理器核心可以同时对图像内部区域应用高斯滤波，大幅减少总体处理时间。

编译运行得到结果如下：

```
n@XiaoxinPro:~/Optimize$ g++ -O0 -o Optimize_3 Optimize_3.cpp -lpthread
n@XiaoxinPro:~/Optimize$ ./Optimize_3 2333
Checksum: 684550983
Benchmark time: 1599 ms
```

注：编译时需要链接 Pthread 库。

d. **版本 4：使用 SIMD 来优化幂次变换，具体操作如下：**

定义一个新的函数 powerLawTransformationSIMD 通过 SIMD 来处理幂次变换，函数有四个参数，一个 int 参数为图像大小 size，一个数组参数为我们版本 2 中定义的查找表，最后两个参数将 figure 数组作为输入数组，result 数组作为

输出数组。遍历输入数组，每次处理 16 个像素。这是因为 avx2 指令集可以一次性处理 128 位数据，而每个像素占 8 位，所以 128 位可以存储 16 个像素。从输入数组加载 128 位数据到寄存器，通过查找表进行变换，并将变换后的像素值存储到输出数组。最后将原来幂次变换的逻辑调整为调用这个 SIMD 版本的函数。

优化说明：使用 SIMD 指令集来同时处理多个像素，而不是逐个像素处理。这样可以大大减少循环次数，提高处理速度。优势在于能够利用现代 CPU 的 SIMD 能力，通过单条指令同时对多个数据执行操作，从而提高数据吞吐量和程序性能，而且减少了内存访问的次数。

编译运行，得到结果如下：

```
n@XiaoxinPro:~/Optimize$ g++ -O0 -mavx2 -mfma -o Optimize_4 Optimize_4.cpp
n@XiaoxinPro:~/Optimize$ ./Optimize_4 2333
Checksum: 684550983
Benchmark time: 1005 ms
```

注：编译时需要增加-mavx2 和-mfma 两个编译参数启用这 AVX2 和 FMA 这两个指令集。

e. 版本 5：使用 SIMD 来优化高斯滤波，具体操作如下：

同理可以联想到对于大型图像的高斯滤波也可以利用 SIMD 来进行优化，由于我们将图像内部的高斯滤波处理已经采用 Pthread 进行优化，所以我们的 SIMD 优化应该在线程执行函数 Internal_Gaussion 中进行修改：对于内部区域的像素，使用 SIMD 技术一次性处理 16 个像素：用_mm256_cvtepu8_epi16 将 8 位的像素值扩展到 16 位，以避免在后续计算中溢出。加载当前像素及其周围像素的值到不同的寄存器中。使用_mm256_mullo_epi16 对每个像素值进行加权乘法。使用_mm256_add_epi16 将加权后的像素值相加。通过右移 4 位（相当于除

以 16）来归一化求和结果。使用_mm_packus_epi16 将结果截断回 8 位无符号整数。使用_mm_storeu_si128 将处理后的 16 个像素值存储回输出图像数据中。

优化说明：与 SIMD 优化幂次变换的原理相同，是需要注意的是应该将结果转换为 8 位的无符号整数再存储回去，否则会因为计算过程中的溢出和截断等使得校验和发生变化。

编译运行后得到结果如下：

```
n@XiaoxinPro:~/Optimize$ g++ -O0 -mavx2 -mfma -o Optimize_5 Optimize_5.cpp -lpthread
n@XiaoxinPro:~/Optimize$ ./Optimize_5 2333
Checksum: 684550983
Benchmark time: 579 ms
```

注：编译命令与版本 4 相同。

f. 版本 6：对于剩余的无数据依赖的 for 循环使用 Openmp 进一步并行化：

在头文件中增加 include<omp.h>,并且在 main 函数中将并行的线程数设置为 4

```
omp_set_num_threads(4);
```

分析 for 循环中是否存在数据依赖：

- powerLawTransformationSIMD 函数中的循环： 这个循环对每个像素应用幂律变换，使用 SIMD 指令集进行优化。每个像素的处理是独立的，因为每个像素的变换只依赖于查找表（LUT）中的值，而不依赖于其他像素的值。因此，这个循环没有数据依赖，可以安全地并行
- Internal_Gaussion 函数中处理 SIMD 部分的循环： 这个循环处理图像的内部区域，每次处理 16 个像素。由于使用了 SIMD 指令集，每个像素的处理也是独立的，因为每个像素的高斯滤波计算只依赖于其周围的像素值，而这些值在循环开始前就已经加载到 SIMD 寄存器中。因此，这个循环也没有数据依赖，可以并行化。

所以我们在这两个循环前面加上：

```
#pragma omp parallel for
```

来实现进一步的并行优化。

优化说明：使用 OpenMP 优化的原理是通过在代码中添加简单的编译器指令，使得编译器能够自动将程序中的循环和任务分配给多个线程并行执行，从而提高程序的执行效率。这种并行化可以显著减少程序的运行时间，由于作业说明中提到评测机器的核心数不超过 4 核，所以我们将线程数设置为 4。

编译运行得到结果如下：

```
n@XiaoxinPro:~/Optimize$ g++ -O0 -fopenmp -mavx2 -mfma -o Optimize_6 Optimize_6.cpp -lpthread
n@XiaoxinPro:~/Optimize$ ./Optimize_6 2333
Checksum: 684550983
Benchmark time: 294 ms
```

注：编译时需要增加-fopenmp 参数来启用 openmp 支持。

3. 实验结果

整理上述每个版本的运行时间，并计算加速比整理如下表：

版本	运行时间 (ms)	相对前一个版本的加速比	相对原始版本的加速比
Optimize_0 （原始代码）	18385	-	-
Optimize_1 （数据结构优化）	9021	2.04	2.04
Optimize_2 （查找表优化）	5546	1.63	3.31
Optimize_3 （Pthread优化）	1599	3.47	11.48
Optimize_4 （SIMD优化1）	1005	1.59	18.28
Optimize_5 （SIMD优化2）	579	1.73	31.71
Optimize_6 （Openmp优化）	294	1.97	62.52

表格中可以看到各个版本相较原始版本的绝对加速比和自己本身优化带来的相对加速比，可以看到我们最终版本 6 的运行时间已经低至 294ms，与原来的 18385ms 相比有 62.52 的绝对加速比，至此实验圆满完成。