C++图像处理代码性能优化

22336044　陈圳煌

一、优化思路

针对原本代码中涉及到较大规模的二维矩阵的计算，采用OpenMP进行多线程并行优化；同时在高斯模糊函数的实现中，涉及到了对数组的连续访问，因此可以使用SIMD指令加速（在本次实验中使用了AVX2指令集）。在幂次变换步骤中，涉及到了对原图像的像素值归一化再进行开平方计算，由于像素值为0~255，每个像素值对应一个固定的幂次变换结果，所以可以通过预先计算一个查找表，可以减少每个像素点都进行开平方计算。

二、实现过程

步骤1：

首先确定全篇代码的框架，发现原本的存储类型是二维的vector容器，考虑到二维的vector每一行都是独立的动态数组，因此不能保证数据存储的内存地址的连续性。而在我们处理的过程中涉及的连续的访问，因此将数据结构修改成一维指针，在调用时充分利用局部性原理，避免太多的访存缺失。

```
// 修改前
vector<vector<unsigned char>> figure;
vector<vector<unsigned char>> result;

// 修改后
unsigned char *figure;
unsigned char *result;
```

步骤2：对函数gaussianFilter()的优化

首先观察到该函数包含一个16384x16384的两层for循环，因此可以使用OpenMP进行优化，设置线程数为4（测评机仅有4个逻辑核心）。

进行OpenMP优化后性能大约能够提升3~4倍之间，接着使用SIMD指令加速，在这里我使用AVX2指令集。

对于原本的内部区域是从1~16382，大小为16382，不是16或者8的倍数，不适合批次处理，因此我将整个像素点分成了5块区域进行处理，分别是内部区域1（从矩阵角度看是行是1~16382，列是64~16319），内部区域2（从矩阵角度看行是1~16382，列是32~63以及16320~16351），内部区域3（从矩阵角度看行是1~16382，列是1~31以及16352~16382），边界区域（第0行、第16383行、第0列、第16383列除去四个端点）以及四个端点。

其中内部区域1一次循环步长为64，内部进行一次训换展开，每次读取32个像素点（1个像素点大小为0~255，即8bit），拼接成256bit的数（__**m256i**类型）然后对每个16位进行高位和低位的分离。

（以下仅展示内部区域1的优化，详细请见代码）

```
// 未优化
for (size_t i = 1; i < size - 1; ++i) {
    for (size_t j = 1; j < size - 1; ++j) {
        result[i][j] =
            (figure[i - 1][j - 1] + 2 * figure[i - 1][j] +
             figure[i - 1][j + 1] + 2 * figure[i][j - 1] + 4 * figure[i][j] +
```

```
             2 * figure[i][j + 1] + figure[i + 1][j - 1] +
             2 * figure[i + 1][j] + figure[i + 1][j + 1]) /
            16;
    }
}
// 优化后
for (size_t i = 1; i < size -1; ++i) {
    for (size_t j = 64; j < size - 64; j += 64) {
        // 提取元素
        __m256i left_up = _mm256_loadu_si256((__m256i*)&figure[(i - 1) * size + j
- 1]);
        __m256i middle_up = _mm256_loadu_si256((__m256i*)&figure[(i - 1) * size +
j]);
        __m256i right_up = _mm256_loadu_si256((__m256i*)&figure[(i - 1) * size +
j + 1]);
        __m256i left_middle = _mm256_loadu_si256((__m256i*)&figure[(i * size) + j
- 1]);
        __m256i middle_middle = _mm256_loadu_si256((__m256i*)&figure[(i * size) +
j]);
        __m256i right_middle = _mm256_loadu_si256((__m256i*)&figure[(i * size) +
j + 1]);
        __m256i left_under = _mm256_loadu_si256((__m256i*)&figure[(i + 1) * size
+ j - 1]);
        __m256i middle_under = _mm256_loadu_si256((__m256i*)&figure[(i + 1) *
size + j]);
        __m256i right_under = _mm256_loadu_si256((__m256i*)&figure[(i + 1) * size
+ j + 1]);
        // 每16位的高位部分
        __m256i left_up2 = _mm256_srli_epi16(left_up, 8);
        __m256i middle_up2 = _mm256_srli_epi16(middle_up, 8);
        __m256i right_up2 = _mm256_srli_epi16(right_up, 8);
        __m256i left_middle2 = _mm256_srli_epi16(left_middle, 8);
        __m256i middle_middle2 = _mm256_srli_epi16(middle_middle, 8);
        __m256i right_middle2 = _mm256_srli_epi16(right_middle, 8);
        __m256i left_under2 = _mm256_srli_epi16(left_under, 8);
        __m256i middle_under2 = _mm256_srli_epi16(middle_under, 8);
        __m256i right_under2 = _mm256_srli_epi16(right_under, 8);
        // 每16位的低位部分
        left_up = _mm256_and_si256(left_up, f);
        middle_up = _mm256_and_si256(middle_up, f);
        right_up = _mm256_and_si256(right_up, f);
        left_middle = _mm256_and_si256(left_middle, f);
        middle_middle = _mm256_and_si256(middle_middle, f);
        right_middle = _mm256_and_si256(right_middle, f);
        left_under = _mm256_and_si256(left_under, f);
        middle_under = _mm256_and_si256(middle_under, f);
        right_under = _mm256_and_si256(right_under, f);
        // 由于其中有四个进行x2，一个进行x4
        // 可以先将中心点x2后再与四方相加，再进行x2，再加上四个角，仅需进行两次乘法
        middle_middle = _mm256_slli_epi16(middle_middle, 1);
        middle_middle = _mm256_adds_epu16(middle_middle, middle_up);
        middle_middle = _mm256_adds_epu16(middle_middle, left_middle);
        middle_middle = _mm256_adds_epu16(middle_middle, right_middle);
        middle_middle = _mm256_adds_epu16(middle_middle, middle_under);
        middle_middle = _mm256_slli_epi16(middle_middle, 1);
        middle_middle = _mm256_adds_epu16(middle_middle, left_up);
```

```
            middle_middle = _mm256_adds_epu16(middle_middle, right_up);
            middle_middle = _mm256_adds_epu16(middle_middle, left_under);
            middle_middle = _mm256_adds_epu16(middle_middle, right_under);

            // 原理同上
            middle_middle2 = _mm256_slli_epi16(middle_middle2, 1);
            middle_middle2 = _mm256_adds_epu16(middle_middle2, middle_up2);
            middle_middle2 = _mm256_adds_epu16(middle_middle2, left_middle2);
            middle_middle2 = _mm256_adds_epu16(middle_middle2, right_middle2);
            middle_middle2 = _mm256_adds_epu16(middle_middle2, middle_under2);
            middle_middle2 = _mm256_slli_epi16(middle_middle2, 1);
            middle_middle2 = _mm256_adds_epu16(middle_middle2, left_up2);
            middle_middle2 = _mm256_adds_epu16(middle_middle2, right_up2);
            middle_middle2 = _mm256_adds_epu16(middle_middle2, left_under2);
            middle_middle2 = _mm256_adds_epu16(middle_middle2, right_under2);

            // 按16位进行右移4位(即除以16)
            middle_middle = _mm256_srli_epi16(middle_middle, 4);
            middle_middle2 = _mm256_srli_epi16(middle_middle2, 4);
            // 两个结果都仅保留低8位，并将原先的进行左移8位作为高位进行组合
            middle_middle = _mm256_and_si256(middle_middle, f);
            middle_middle2 = _mm256_and_si256(middle_middle2, f);
            middle_middle2 = _mm256_slli_epi16(middle_middle2, 8);
            middle_middle = _mm256_add_epi16(middle_middle, middle_middle2);
            // 将结果回传内存
            _mm256_storeu_si256((__m256i*)&result[i * size + j], middle_middle);
            //以下进行循环展开，对后32个数据进行处理
        }
    }
```

其余内部区域3以及边界情况均只使用OpenMP进行优化。理论上SIMD加速上，32个元素并行计算应有32倍的性能提升，但是由于内存带宽、数据访问模式、硬件特性等因素，实际提升远达不到这个效果，结合OpenMP并行优化后的**gaussianFilter**()加速比约为35.44（±3.0）。

```
hzc_hho@hzc-hho:/mnt/c/Users/陈圳煌/Desktop/计算机学院/计算机学院（大三上）/并行/大作业$
o before_optimize && ./before_optimize 100
time1: 7124 ms
time2: 6924 ms
Checksum: 684878143
Benchmark time: 14048 ms
```

```
hzc_hho@hzc-hho:/mnt/c/Users/陈圳煌/Desktop/计算机学院/计算机学院（大三上）/并行/大作业$
 after_optimize && ./after_optimize 100
Number of threads = 4
time1: 201 ms
time2: 201 ms
Checksum: 684878143
Benchmark time: 402 ms
```

步骤3：对函数powerLawTransformation()的优化

对于这部分的处理，对于幂次变换，实际上就是将像素值进行下面的变换：

$$f(x) = (\frac{f(x)}{255})^r \times 255$$

由于所有像素点f(x)的值都是包含着0~255的范围内的整数，因此可以预先计算查找表，在后续的计算中可以通过直接调用来节省计算时间，并且在内层循环进行循环展开。

```
// 未优化
constexpr float gamma = 0.5f;
for (size_t i = 0; i < size; ++i) {
    for (size_t j = 0; j < size; ++j) {
        if(figure[i][j] == 0) {
            result[i][j] = 0;
            continue;
        }
        float normalized = (figure[i][j]) / 255.0f;
        result[i][j] = static_cast<unsigned char>(
            255.0f * std::pow(normalized, gamma) + 0.5f);
    }
}
// 优化后
unsigned char LUT[256];
constexpr float gamma = 0.5f;
const float delta = std::pow(255.0f, gamma);
for (size_t i = 0; i < 256; ++i) {
    LUT[i] = static_cast<unsigned char>(255.0f * std::pow(i, gamma) / delta +
0.5f);
}

for (size_t i = 0; i < size; ++i) {
    for (size_t j = 0; j < size; j += 4) {
        result[i * size + j] = LUT[figure[i * size + j]];
        result[i * size + j + 1] = LUT[figure[i * size + j + 1]];
        result[i * size + j + 2] = LUT[figure[i * size + j + 2]];
        result[i * size + j + 3] = LUT[figure[i * size + j + 3]];
    }
}
```

可以大量减少重复计算的时间，实际上LUT的计算过程可以放在初始化的过程中，可以省去一部分计算时间，不过这里考虑到函数功能，因此保留在这个函数中。根据测试，使用OpenMP加上LUT查找表优化后的**powerLawTransformation**()加速比约为34.45（±2.0）。（对比图见高斯模糊函数的例子）

三、优化效果

本地测试结果：

```
hzc_hho@hzc-hho:/mnt/c/Users/陈圳煌/Desktop/计算机学院/计算机学院（大三上）/并行/大作业$ ./run.sh
g++ -O0 -march=native -mavx2 -std=c++23 -fopenmp before_optimize.cpp -o before_optimize && ./before_optimize 100
Checksum: 684878143
Benchmark time: 13952 ms
g++ -O2 -march=native -mavx2 -std=c++23 -fopenmp before_optimize.cpp -o before_optimize && ./before_optimize 100
Checksum: 684878143
Benchmark time: 2295 ms
g++ -O3 -march=native -mavx2 -std=c++23 -fopenmp before_optimize.cpp -o before_optimize && ./before_optimize 100
Checksum: 684878143
Benchmark time: 2339 ms
g++ -O0 -march=native -mavx2 -std=c++23 -fopenmp after_optimize.cpp -o after_optimize && ./after_optimize 100
Number of threads = 4
Checksum: 684878143
Benchmark time: 366 ms
g++ -O2 -march=native -mavx2 -std=c++23 -fopenmp after_optimize.cpp -o after_optimize && ./after_optimize 100
Number of threads = 4
Checksum: 684878143
Benchmark time: 85 ms
g++ -O3 -march=native -mavx2 -std=c++23 -fopenmp after_optimize.cpp -o after_optimize && ./after_optimize 100
Number of threads = 4
Checksum: 684878143
Benchmark time: 86 ms
```

```
hzc_hho@hzc-hho:/mnt/c/Users/陈圳煌/Desktop/计算机学院/计算机学院（大三上）/并行/大作业$ ./run.sh
g++ -O0 -march=native -mavx2 -std=c++23 -fopenmp before_optimize.cpp -o before_optimize && ./before_optimize 100
Checksum: 684878143
Benchmark time: 13940 ms
g++ -O2 -march=native -mavx2 -std=c++23 -fopenmp before_optimize.cpp -o before_optimize && ./before_optimize 100
Checksum: 684878143
Benchmark time: 2315 ms
g++ -O3 -march=native -mavx2 -std=c++23 -fopenmp before_optimize.cpp -o before_optimize && ./before_optimize 100
Checksum: 684878143
Benchmark time: 2351 ms
g++ -O0 -march=native -mavx2 -std=c++23 -fopenmp after_optimize.cpp -o after_optimize && ./after_optimize 100
Number of threads = 4
Checksum: 684878143
Benchmark time: 386 ms
g++ -O2 -march=native -mavx2 -std=c++23 -fopenmp after_optimize.cpp -o after_optimize && ./after_optimize 100
Number of threads = 4
Checksum: 684878143
Benchmark time: 120 ms
g++ -O3 -march=native -mavx2 -std=c++23 -fopenmp after_optimize.cpp -o after_optimize && ./after_optimize 100
Number of threads = 4
Checksum: 684878143
Benchmark time: 117 ms
```

从以上结果上来看最后得到的校验和checksum与原始代码一致，运行时间从13900ms左右优化到380ms左右，加速比大约为36。但是在测试过程中，由于个人电脑插电影响以及本地可能有其他应用占用CPU资源，因此测试结果有一定偏差，有时候会出现超出350ms~450ms这个区间的结果，综合计算大约能够提升35~40倍左右的性能，与前一部分中两个函数提升的分析较符合，并且使用O2、O3编译优化时均有3~4倍左右的提升，表明手工优化的代码能够较为适配硬件，与参考所给例子相当，可以认为在实验中所做的处理有一定的优化效果。

【以下结果为借助同学的台式计算机进行测试，加速比有几率达到40以上。】

```
root@DESKTOP-2PSKQ05:/mnt/c/Users/preju/Desktop/czh# source run.sh
g++ -O0 -march=native -mavx2 -std=c++23 -fopenmp before_optimize.cpp -o before_optimize && ./before_optimize 100
Checksum: 684878143
Benchmark time: 12983 ms
g++ -O2 -march=native -mavx2 -std=c++23 -fopenmp before_optimize.cpp -o before_optimize && ./before_optimize 100
Checksum: 684878143
Benchmark time: 2454 ms
g++ -O3 -march=native -mavx2 -std=c++23 -fopenmp before_optimize.cpp -o before_optimize && ./before_optimize 100
Checksum: 684878143
Benchmark time: 2403 ms
g++ -O0 -march=native -mavx2 -std=c++23 -fopenmp after_optimize.cpp -o after_optimize && ./after_optimize 100
Number of threads = 4
Checksum: 684878143
Benchmark time: 313 ms
g++ -O2 -march=native -mavx2 -std=c++23 -fopenmp after_optimize.cpp -o after_optimize && ./after_optimize 100
Number of threads = 4
Checksum: 684878143
Benchmark time: 69 ms
g++ -O3 -march=native -mavx2 -std=c++23 -fopenmp after_optimize.cpp -o after_optimize && ./after_optimize 100
Number of threads = 4
Checksum: 684878143
Benchmark time: 84 ms
```

-

四、心得体会

在这次完成计算机体系结构的课程作业过程中，我自学了SIMD指令加速优化，从一窍不通到能够初步使用AVX2指令集进行并行优化，并且将高性能程序设计中所学的OpenMP用于实践之中还了解了查找表这一办法在重复计算固定值这个问题上的妙用，感到收获满满。在实验中，刚开始对SIMD指令不熟悉，用循环进行转换类型赋值，最后再重新存储回结果位置，并不能有效进行优化。为了解决这个问题，我查找了相关文档，了解了相关函数的功能以及相关原理，最后能够使用SIMD指令进行加速优化。最后感谢胡老师和助教老师在体系结构这一门课程上的指导。