

A BMP Image Processor

Student ID: 12312710
Name: 胥熠/Brighton
Course: CS219: Advanced Programming
Date: 2025 年 4 月 13 日

目录

1 前言	2
2 功能展示	2
2.1 运行环境	2
2.2 程序使用教程	2
2.3 为图片增添亮度	3
2.4 平均两张图片	3
2.5 让图片变成灰色调	4
2.6 水平竖直翻转	5
2.7 为图片添加模糊效果	6
2.8 基于 Soble 算子的边缘检测	7
2.9 错误输入识别	7
3 功能实现	9
3.1 24 位无压缩 BMP 文件的存储结构	9
3.2 数据存储方式	10
3.3 内存对齐	11
3.4 图片处理	11
4 如何科学优化	12
4.1 线性存储像素	12
4.2 编译器优化	12
4.3 并行化操作	13
4.4 避免重复索引计算	13
4.5 多种封装方式	14
4.6 性能对比	14
5 AI 辅助了哪些工作	17
5.1 程序主体逻辑	17
5.2 加速撰写报告	17
5.3 Python 统计数据程序生成	17
6 结语	17

1 前言

你说的对，但是「BMP」是由微软自主研发的一种全新无压缩位图存储标准。图像被存储在一个被称作「DIB 设备无关位图」的二进制体系中... 看到这个 Project 的第一眼——唉，今年 Project 好像更新换代了，和往年不太一样了。再仔细看一眼，其实还是换汤不换药——矩阵！最后定睛一看，今年的其实要更加友善，毕竟不涉及矩阵的乘法，但是我们依旧可以逐步发掘「优化」的真相。同时在图像处理方面，我还引入了经典 Sobel 算子（一种简单的卷积操作）来检测图像的边缘。

2 功能展示

2.1 运行环境

本程序是由 C 语言编写。运行设备为 Lenovo Legion Y9000P 运行 x86 下的 Windows 11，使用 Intel i9-13900HX(8 P-cores + 16 E-cores)，24GB DDR5 运存。使用 Powershell 通过 GCC 编译器运行下列命令编译和运行 (bmpedit.c 放置在 path 文件夹内)：

```
PS *path*> gcc -o bmpedit bmpedit.c -O3 -march=native -fopenmp -lm
PS *path*> ./bmpedit *Tokens*
```

其中，“-O3”和“-march=native”是可选项，是为了在编译层面加快程序运行速度。“-fopenmp”选项是必要的，因为这个程序中使用了 OpenMP 并行处理指令。“-lm”是链接的数学库，在绝大多数情况不需要但是也可以加上。观测软件使用的是 Intel VTune Profiler。

2.2 程序使用教程

直接输入./bmpedit 或者错误使用指令时，程序会展示使用方法，内容如下：

```
Usage: ./bmpedit -i input.bmp [-i input2.bmp] -o output.bmp -op operation [args]
```

Operations:

- | | |
|-------------------|---|
| add VALUE | - Adjust brightness by adding VALUE to all pixels |
| average | - Blend two images by averaging their pixel values |
| grayscale | - Convert image to grayscale |
| flip h v | - Flip image horizontally (h) or vertically (v) |
| blur [RADIUS] | - Apply blur effect with optional radius (default: 1) |
| sobel [THRESHOLD] | - Apply Sobel edge detection (default threshold: 100) |

Examples:

- ./bmpedit -i input.bmp -o output.bmp -op add 50
- ./bmpedit -i input1.bmp -i input2.bmp -o output.bmp -op average
- ./bmpedit -i input.bmp -o output.bmp -op grayscale
- ./bmpedit -i input.bmp -o output.bmp -op flip h

```
./bmpedit -i input.bmp -o output.bmp -op blur 3  
./bmpedit -i input.bmp -o output.bmp -op sobel 120
```

2.3 为图片增添亮度

在命令行中调用：

```
./bmpedit -i input.bmp -o output.bmp -op add 200
```

其中文件都在当前目录，input.bmp 为输入文件名，output.bmp 为输出文件名，add 后面数字范围为 [1,255]。

输入案例（input.bmp）



图 1: 处理前

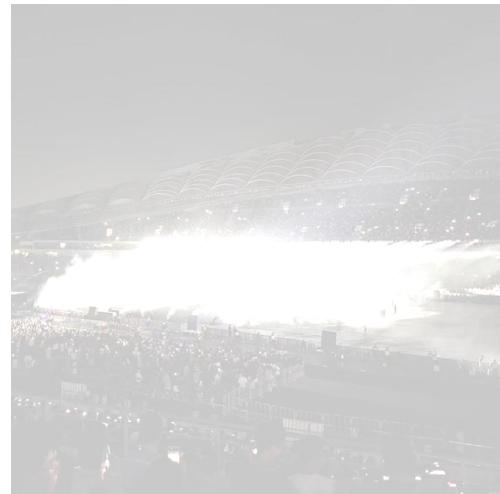


图 2: 处理后

可以看到在全局加了 200 点亮度后，一张很黑的图片被处理成了比较亮的图片，同时注意到图片中间发光区域的细节消失了。

2.4 平均两张图片

在命令行中调用：

```
./bmpedit -i input.bmp -i input2.bmp -o output.bmp -op average
```

对两张相同长宽的照片进行像素级别的平均，其对应像素 RGB 是两个图片对应像素 RGB 值的算数平均数。

输入案例（input.bmp, input2.bmp）



图 3: 一只可爱的企鹅

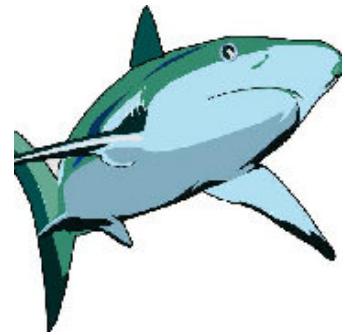


图 4: 一条可爱的鲨鱼



图 5: 平均结果

可以看到两张图片平均后原本的颜色都变淡了，且重合部分可以同时看到两个图片的细节，产生了半透明叠加的效果。

2.5 让图片变成灰色调

在命令行中调用：

```
./bmpedit -i input.bmp -o output.bmp -op grayscale
```

将彩色图像转换为灰度图像，模拟黑白照片效果。本程序采用常用的加权平均法 (Luminance method: $Gray = 0.299 \times R + 0.587 \times G + 0.114 \times B$)。

输入案例 (input.bmp)



图 6: 一张五彩斑斓的图像



图 7: 转换后的灰度图像

2.6 水平竖直翻转

在命令行中调用：

```
./bmpedit -i input.bmp -o output1.bmp -op flip h  
./bmpedit -i input.bmp -o output2.bmp -op flip v
```

将图像左右镜像翻转，或者上下镜像翻转。

输入案例（input.bmp）



图 8: 原始图像

输出案例（output1.bmp、output2.bmp）



图 9: 左右翻转后的图像



图 10: 上下翻转后的图像

2.7 为图片添加模糊效果

在命令行中调用：

```
./bmpedit -i input.bmp -o output.bmp -op blur 4
```

对图像应用简单的盒子模糊（Box Blur）效果，使图像看起来模糊或失焦。其中“4”代表模糊半径为 4 格像素（即处理 $(2 * 4 + 1) \times (2 * 4 + 1) = 9 \times 9$ 的邻域）。可以换成其他任意大小的正整数（程序内限制了上限为 10）。



图 11: 原始图像



图 12: 处理后图像

2.8 基于 Soble 算子的边缘检测

在命令行中调用:

```
./bmpedit -i input.bmp -o output.bmp -op sobel 200
```

对图像应用简单的 Soble 算子，检测出图像中 RGB 信息变化剧烈的边缘部分。其中“200”是设定的阈值，近似梯度高于阈值的部分会被认为是边缘并输出黑色 (0,0,0)，否则输出白色 (255,255,255)。



图 13: 原始图像

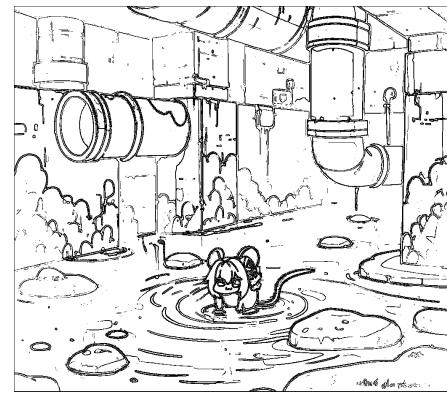


图 14: 处理后的图像

可以看到处理后的图像保留了原图像的主要轮廓和纹理边缘。

2.9 错误输入识别

程序能够识别一些常见的用户输入错误或文件问题，并给出相应的提示信息，避免程序异常退出或产生错误结果，具有良好的 Robustness。

```
# 缺少操作
./bmpedit -i demo5.bmp -o out.bmp
Error: No operation specified
*Display Usage*

# 缺少必要参数（如输出文件）
PS *path*> ./bmpedit -i input.bmp -op add 50
Error: No output file specified
Usage: ./bmpedit ...

# 操作参数不足（如 'add' 缺少数值）
PS *path*> ./bmpedit -i input.bmp -o out.bmp -op add
Error: 'add' operation requires a value

# 无效操作名称
PS *path*> ./bmpedit -i input.bmp -o out.bmp -op rotate 90
Error: Unknown operation 'rotate'

# 文件无法打开或不存在
PS *path*> ./bmpedit -i non_existent.bmp -o out.bmp -op grayscale
Error: Cannot open file non_existent.bmp
Error: Could not load input file non_existent.bmp

# 文件格式不支持（非"BM"签名，或非24位，或压缩格式）
PS *path*> ./bmpedit -i not_a bmp.jpg -o out.bmp -op grayscale
Error: Not a BMP file (invalid signature)
PS *path*> ./bmpedit -i indexed_color.bmp -o out.bmp -op grayscale
Error: Only 24-bit uncompressed BMP files are supported

# 'average' 操作缺少第二个输入文件
PS *path*> ./bmpedit -i input1.bmp -o out.bmp -op average
Error: 'average' operation requires two input images

# 'average' 操作两输入文件尺寸不匹配
PS *path*> ./bmpedit -i input1.bmp -i different_size.bmp -o out.bmp -op average
Error: Images must have the same dimensions for averaging
Error: Operation failed

# 程序成功运行时提示（blur 和 sobel 操作还会返回运行时间供参考）
PS *path*> ./bmpedit -i demo12.bmp -o outputtemp.bmp -op blur 5
Blur operation took 2.588488 seconds
Operation 'blur' completed successfully. Output saved to outputtemp.bmp
```

这些检查能够保证程序能够被正确使用，并且给出对应错误提示方便用户纠正。

3 功能实现

3.1 24 位无压缩 BMP 文件的存储结构

BMP(Bitmap) 是一种简单的图像格式，而其中 24 位无压缩 BMP 则是其中的相对好处理的一种格式。在这个 Project 中，我将只关注 24 位无压缩 BMP 文件。因此在下述的报告中，所有 BMP 都指 24 位无压缩 BMP 文件。

一个标准的 24 位无压缩 BMP 文件主要由三部分构成：14 字节的文件头、通常为 40 字节的信息头和像素数据。这部分位于文件的开头，提供了文件的基本标识和信息。紧随文件头之

表 1: BMP 文件头 (BMP Header - 14 字节)

偏移量	大小 (字节)	描述
0x00	2	文件类型标识符 (必须为"BM", 0x42 0x4D)
0x02	4	文件大小 (以字节为单位)
0x06	2	保留，通常设为 0
0x08	2	保留，通常设为 0
0x0A	4	从文件头到像素数据的偏移量

表 2: 信息头 (DIB Header - 40 字节)

偏移量	大小 (字节)	描述
0x0E	4	信息头大小 (对于 BITMAPINFOHEADER 为 40)
0x12	4	图像宽度 (像素)
0x16	4	图像高度 (像素, 可为负值)
0x1A	2	色彩平面数 (必须为 1)
0x1C	2	每像素位数 (对于 24 位 BMP 为 24)
0x1E	4	压缩方式 (0 表示无压缩)
0x22	4	图像大小 (可设为 0 表示使用默认值)
0x26	4	水平分辨率 (像素/米)
0x2A	4	垂直分辨率 (像素/米)
0x2E	4	调色板颜色数 (0 表示使用最大值)
0x32	4	重要颜色数 (通常为 0)

后，描述了图像的具体属性，如尺寸、颜色深度等。

对于像素部分，24 位无压缩 BMP 就是指图片中一个像素占 24 位，也就是 3 字节。其中三个字节分别存储蓝色、绿色、红色，也就是经典的 RGB 存储（额，不应该叫 BGR 吗）。一个 unsigned 字节能够表示的数字范围为 0-255，数字越小代表这个颜色越暗。所以 RGB(0,0,0) 代表的是黑色，而 RGB(255,0,0) 代表的是纯红色。

像素按行存储，而且是从下到上存储。每行像素数据按 4 字节对齐。例如一行中有五个像素，则一行的数据会有 $5 \times 3 = 15$ 个字节，为了字节对齐 4 的倍数，BMP 文件在这一行后会填充一 byte 的对齐数据，因此在读取的时候要计算好 Padding 量，在一行结束时谨慎确定下一行开始位置。

3.2 数据存储方式

在这里，我们为了方便使用 `fread` 读取，我们定义了四个结构体，分别为：`BMPFileHeader`（14 字节文件头）、`BMPInfoHeader`（40 字节信息头）、`Pixel`（24 位像素）和 `BMPImage`（图片总封装）。

其中，`BMPFileHeader` 和 `BMPInfoHeader` 的封装如下，其顺序符合上述表格：

```
typedef struct {
    unsigned short bfType;          // Magic identifier "BM" (0x4D42)
    unsigned int   bfSize;           // File size in bytes
    unsigned short bfReserved1;     // Reserved
    unsigned short bfReserved2;     // Reserved
    unsigned int   bfOffBits;        // Offset to image data
} BMPFileHeader;

typedef struct {
    unsigned int   biSize;           // Header size
    int            biWidth;          // Width of the image
    int            biHeight;         // Height of the image
    unsigned short biPlanes;        // Color planes
    unsigned short biBitCount;      // Bits per pixel
    unsigned int   biCompression;   // Compression type
    unsigned int   biSizeImage;     // Image size in bytes
    int            biXPelsPerMeter; // X resolution
    int            biYPelsPerMeter; // Y resolution
    unsigned int   biClrUsed;        // Colors used
    unsigned int   biClrImportant;   // Important colors
} BMPInfoHeader;
```

对于一个像素，包含红绿蓝三种颜色，因此也需要封装三个 `unsigned char`。

```
typedef struct {
    unsigned char blue;
    unsigned char green;
    unsigned char red;
} Pixel;
```

而对于整体 `BMPImage` 封装，则需要稍微小心。这里不采用二维数组（双指针）存储像素，而是采用线性存储，即数组的第 $[0, width-1]$ 个元素代表第一行， $[width, 2*width-1]$ 代表第二行... 访问 (x, y) 处的像素通过计算索引 $y \times width + x$ 实现。也可以定义一个宏 `define PIXEL_AT(image, x, y) ((image)->pixels + (y) * (image)->width + (x))` 来方便计算索引。

其实对于矩阵，采用线性存储是普遍的常识，也是后续 SIMD 实现的必要数据结构。如果使用双重指针，不仅多很多次不必要的解指针操作，内存不连续还会导致 CPU 根本没有办法进行并行操作。同时显而易见的，线性存储只用一次 `malloc`，这为内存管理提供了很大的方便。

以及，存储矩阵的长宽属性，我们应该使用 `size_t`，在比 `int` 范围大一倍的同时免去了检查参数负数的必要。集齐以上要素，我的 BMP 封装的设计为：

```

typedef struct {
    BMPFileHeader fileHeader;
    BMPInfoHeader infoHeader;
    Pixel* pixels;
    size_t width;
    size_t height;
} BMPImage;

```

这种设计既符合直觉，也为后续的高效处理（尤其是并行化和向量化）奠定了基础。

3.3 内存对齐

在读取文件的时候，我们必须要保证文件的每一个字节都被映射到适当的位置。但是 C/C++ 的结构体成员内存对齐是一个相对混沌邪恶的东西，它可能会在某些 2 字节的变量后面插入两个填充字节以保证 4 字节对齐（例如在 *BMPFileHeader* 的 *bfType* 和 *bfSize* 之间）。这样会导致 *sizeof(BMPFileHeader)* 不等于 14 字节，*fread* 读取的数据就会错位。

因此对于两种文件头封装结构，我们必须使用禁用自动对齐，保证结构体成员紧密排列，与文件格式完全一致。

```

#pragma pack(push, 1) // 禁用自动对齐
typedef struct { /* Header members */ } BMPFileHeader;
typedef struct { /* Info header members */ } BMPInfoHeader;
#pragma pack(pop) // 恢复自动对齐

```

保证了正确对齐后，我们就可以直接使用 *fread* 和 *fwrite* 直接读写结构体。

3.4 图片处理

现在，我们成功的将一张图片的数据导入到了结构体中，终于到了这个程序最核心，但也是相对简单的一部分，处理图片。

亮度调整 (addBrightness): 为了调整亮度，我们只需要 *for* 循环，将每一个 *Pixel* 元素的红绿蓝三个值加上某一个数值，大于 255 的数字设置成 255 就可以了。上面事例图的高光部分消失就是因为加上一个数字时候，那一片区域的 RGB 都超出阈值都被设置成上限 255，所以细节丢失了。

图像平均 (averageImages): 首先检查两张输入图像的尺寸（宽度和高度）是否完全一致，如果不一致则报错退出。如果尺寸相同，就创建一个新的空白图像作为结果。然后遍历所有像素位置计算 RGB 算数平均值即可。

灰度化 (convertToGrayscale): 利用加权平均数 $0.299R + 0.587G + 0.114B$ 计算亮度 (Luminance Preserving，原理基于人眼对光的敏感度)，然后将这个数值在同时赋给 RGB 三色，呈现出来的就是对应亮度的灰色调。

图像翻转 (flipImage): 原理非常简单，根据是 H 还是 V 进行行或列内的数列翻转操作。

盒子模糊 (blurImage): *blur* 操作的原理是对一个像素而言，后模糊信息来自于距离为半径窗口范围内的像素的平均值。也就是说假如说我们的模糊半径为 2，则我们计算以此像素为中心，边长为 5 的正方形像素的 RGB 平均值，超出图像边缘的像素按照反射处理。这么看我们的算法时间复杂度为 $O(r^2 * pixels)$ ，我们可以通过滑动窗口算法来优化。即计算完第一个像素后，右侧的像素只需要在左侧计算结果的基础上，减去最左侧一边，加上最右边的数据即可，优化后，算法的时间复杂度为 $O(r * pixels)$ 。

Sobel 边缘检测 (detectEdges): Sobel Sobel 算子是基于一阶导数近似的经典边缘检测方法。通过将 3×3 的 Sobel 卷积核 (G_x 用于水平梯度, G_y 用于垂直梯度) 应用于图像的每个像素 (及其邻域) 的 RGB 通道，分别计算出各通道的梯度分量。随后，计算每个像素的梯度幅度 ($\sqrt{G_x^2 + G_y^2}$)。

为了效率，本实现使用各通道梯度绝对值之和的最大值作为总梯度，并使用曼哈顿距离 ($|G_x| + |G_y|$) 作为梯度幅度。将此幅度与用户设定的阈值比较：若大于阈值，则输出像素设为黑色 (0,0,0)，表示边缘；否则设为白色 (255,255,255)。图像的边界像素（最外一圈）被直接设置为黑色。下面是 Sobel 算子的矩阵表示：

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

4 如何科学优化

4.1 线性存储像素

线性存储像素的优势已经在上面提及了，它不仅避免了多次 malloc 和解指针，还是后续并行操作的基石，还可以改善 Cache 命中率，因此再次强调一下。

4.2 编译器优化

首先，所有不开 O3 优化的性能分析都没有任何意义。其次，C 语言基础语法性能不够，打算手写 SIMD？我们在这个 Project 中涉及的算法都是矩阵加法，由于我们线性存储像素，所以一切操作在编译器面前都是 $Vector_A = Vector_B + Vector_C$ 的形式，和向量点乘一样属于编译器看得懂，能做优化的类型。因此在 gcc 的-O3 和-march=native 的优化下，能够生成 SIMD 技术加持的机器码，因此也没必要手写了。这是 addBrightness 方法中的部分逆向机器码，展示了向量化操作：

addBrightness: 这是外部调用的主函数接口。任务是设置环境、检查参数，并启动 OpenMP。
 4035e0: ^__ICallq 4039c0 <GOMP_parallel> ; 调用 OpenMP 并行区域
 addBrightness._omp_fn.0: 这是由编译器为 OpenMP 并行区域生成的工作函数。

```

4015f5:^^Vpmovzxbw %xmm7,%ymm10      ; 零扩展8位到16位
401635:^^Vpaddd %ymm14,%ymm2,%ymm2    ; 向量加法
40163a: vinserti128...                  ; 合并成32字节YMM5
40164f: vmovdqu -0x40(%r8),%xmm6       ; 预取下一块数据
40165b: vpmovzxbw %xmm7,%ymm10      ; 8位→16位无符号扩展
40166e: vpaddd %ymm14,%ymm2,%ymm2    ; 32位加法
4016a0:^^Vpminsd %ymm13,%ymm9,%ymm9    ; 向量最小值饱和
4017ac:^^Vpackuswb %ymm5,%ymm2,%ymm5  ; 打包32位到8位
...

```

可以看到，在 gcc 生成的 SIMD 中，一如既往的将标量计算转换成向量计算，一次迭代处理 32 字节。在这种相对简单直白的任务中，gcc 能够应付过来。

但是需要注意的是当任务变得复杂，比如很多分支或者内存不规则的情况，gcc 很可能就不能理解你在干什么进而无法生成对应 SIMD 了，因此优化也不能全部指望编译器，或者说要写编译器能看懂的代码。

4.3 并行化操作

OpenMP 伟大，我们只需要在需要并行化的循环前加上一行 `#pragma omp parallel for (args)`，然后在编译的时候加上 `-fopenmp` 连接上 OpenMP 的库就可以享受并行给程序带来的速度。使用 OpenMP 需要注意的点就是尽量减少数据竞争，就比如说单向量点乘加到一个变量的操作，可以优化成点乘分散加到一个向量上最后相加。多个线程竞争着写入一个变量会导致一定的性能削减。在我们的图像处理中，每个像素都是相对独立的，因此这种顾虑比较小。

4.4 避免重复索引计算

在 blurImage 的滑动窗口计算中有两个双重循环，大概长这样：

```

// 水平方向模糊
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        * 计算水平窗口的像素和 *
        long long sumRed = 0, sumGreen = 0, sumBlue = 0, int count = 0; ...
        * 存储结果到临时缓冲区 *
        temp[i * width + j].red = (unsigned char)(sumRed / count);
        temp[i * width + j].green = (unsigned char)(sumGreen / count);
        temp[i * width + j].blue = (unsigned char)(sumBlue / count);
        注意这里的计算冗余
    }
}

```

我们发现，在存储结果到临时缓冲区的时候，一个 forJ 循环要计算 $3 \times \text{width}$ 次 $i \times \text{width}$ ，这个计算显然是冗余的，我们可以这样优化：

```

for (int i = 0; i < height; i++) {
    int index_temp = i * width;
    for (int j = 0; j < width; j++) {
        * 存储结果到临时缓冲区 *
        temp[index_temp + j].{Colors} = (unsigned char)(sum.{Colors} / count);
    }
}

```

这样提前计算好索引开始的位置，我们一次 forJ 循环就只用计算一次 $i \times width$ ，这个双重循环总共节省了大约 $3 \times pixels$ 次乘法计算。

测试性能，发现这种操作好像没有带来时间上的优化，似乎是 O3 优化帮我们做了，但这种思想还是值得一提。

4.5 多种封装方式

我们注意到，其实 Pixel 封装像素数据对于某些并行操作而言，是一种相对不利的数据结构。比如说，我们想要让某张图像的颜色变得更绿，假如说我们用三条数组分别存储 RGB 颜色数据。这样的话，对于每一种颜色的内存空间都是连续的了，变绿也就变成真正的向量加法操作了，效率肯定会有提升。

但这也不一定对所有并行操作有利，例如 Sobel 算子，RGB 单独存储反而吃了不连续和多次读写的亏。因此我这里只是抛砖引玉一下，在实际应用中需要按照需求和实际性能表现选择不同的封装方式，才不是因为没时间了。具体可以搜索关键词：AoS: Array of Structures vs SoA: Structure of Arrays。

4.6 性能对比

为了量化各个不同优化手段的加速效果，我们以计算量最大的 blur (Radius = 5) 为例进行计时比较。测试环境同上 (i9-13900HX)，计时方法采用 Project 2 中实现的跨平台高精度计时器，仅计算核心计算时间不计入文件 IO 时间。



图 15: 实验使用的图像，为展示已压缩处理

这里我们设计了四种情况,分别是不使用编译器优化,单采用-O3 -march=native 优化(SIMD 指令), 单使用 OpenMP, SIMD+OpenMP 四种不同优化类型组合。处理一张大小为 $9725 \times 4862 \approx 47M$ 像素的图片(上图), 只记录实际处理时间, 不记录文件 IO 时间, 重复五次记录平均值。这是实验结果:

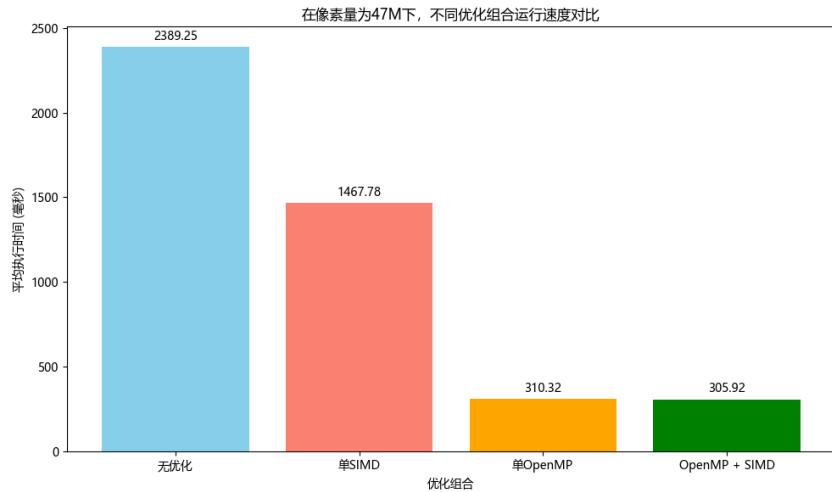


图 16: 47M

感觉加了 OpenMP 后, SIMD 好像没有起到什么优化效果。我们再将原图长宽放大三倍(数据量翻九倍)看看, 现在图片大小为 $29175 \times 14586 \approx 426M$, 占用硬盘空间已经达到了 1.16G, 可以说是一个巨量的数据了:

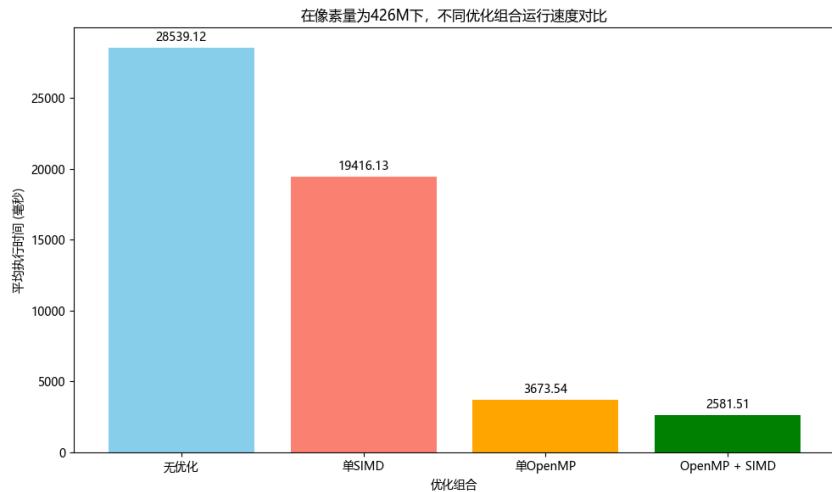


图 17: 426M

可以看到, OpenMP + SIMD 的组合在计算密集型任务中确实是表现最佳的。而且数据量非常大的时候也能够和单 OpenMP 拉开一些差距, 这可能是因为 SIMD 的优化效果常数比较大。在 426M 的数据量下, OpenMP + SIMD 比无优化的速度快了 11 倍左右, 还是相当可观的。

我们再用 Intel VTune Profiler 观察一下运行瓶颈：

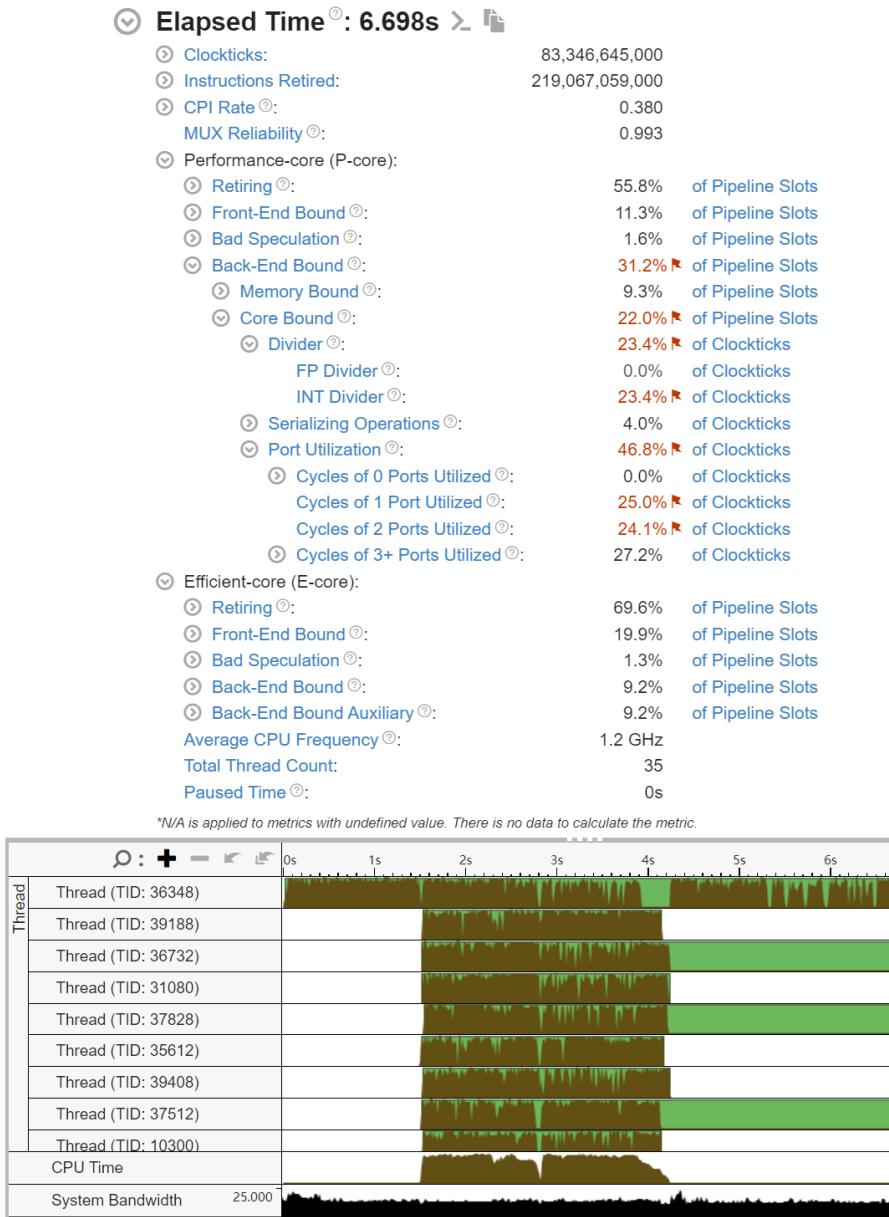


图 18: 分析结果

首先最大的变化，我们发现 Total Thread Count 变成了 35 个，可以看到运行中段处理照片的时候，OpenMP 帮助我们开了很多线程并行处理图片。我们也发现 Memory Bound 很小，主要程序慢在 Divider（取平均值那一步）和 Port Utilization（很多线程同时向 Sum 写入数据，需要等待）。总的来看 Core Bound 是最大的，说明 CPU 核心计算能力是主要限制因素，充分利用了 CPU 的能力，表明当前的优化策略是合理的。进一步优化的话，可以考虑手动把 Sum 向量化后相加的方向。

5 AI 辅助了哪些工作

5.1 程序主体逻辑

这个 Project 比较 LLM 友好，我的程序主体架构是由 Claude 3.7 Sonnet 提供的，它主要帮我完成了程序的 IO 和 Tokenize 部分。它也帮我写了一些方法和结构体，但写的不好。首先它用的是二维数组存储，再其次它用的是 int 存储长宽。我在他的基础上进行了封装体的改进，内存的进一步管理，一些新方法的移植，以及 OpenMP 等进一步的优化。

5.2 加速撰写报告

我有一个神奇的观点——在 LLM 逐渐变强的时代下，我们必须要建立一个自己的语料库。

谷歌最新的 Gemini 2.5 Pro 已经支持了 100M Token 的上下文，最长至 65536Token 的输出。这种量变会造成质变，我可以把前两个 Project 的 latex 丢进去，把我的程序源码丢进去，甚至把我写的一些文章丢进去（模仿语言风格）让他来写 Project3 报告。事实上我虽然没有直接粘贴他的报告，但也边看着 AI 的输出边自己写，起到了一个参考的作用，而且它还自动帮我补全了部分 latex，的确能够加速报告的完成。（缺点是，对 AI 味极其敏感的人还是会察觉到不对劲）

5.3 Python 统计数据程序生成

上述所有统计图均是由 Copilot 内置的 GPT-4o 生成的 Python 程序绘制。

6 结语

本 Project 实现了一个命令行 24 位无压缩 BMP 图像处理器，可以支持很多功能例如灰色调和边缘处理，同时也探索了如何加速图片处理的过程。

本 Project 还有很多可以改进的地方，例如在优化领域做的工作还是太浅了，为了效率肯定还是要手写原生的 SIMD 的。但是由于本 Project 时间被期中周覆盖，工作量太大了实在是无法完成。但作为一次学习来说还是相对成功的，我探索了数据结构、优化方法和 SIMD 并行计算等知识，从不同角度思考了如何加速图片处理。

同时，与 Project 1 和 Project 2 一样，在 DDL 结束后我也会将本次 Project 中涉及到的所有源码开源，放在 <https://github.com/BrightonXX/SUSTech-CPP-Project> 下。

感谢各位能读到这里。

References

参考文献

- [1] Yan W.Q. *CS205 Project4: Matrix Multiplication*. Southern University of Science and Technology, 2022. [Source code]. Available: github.com/YanWQ-monad/CS205_Project4
- [2] Yu Shiqi 快速学习 C 和 C++, 基础语法和优化策略, 学了不再怕指针 (南科大计算机系原版)-12.1-improve-your-source-code. Southern University of Science and Technology, 2021. [Source code]. Available: [bilibili.com/video/BV1Vf4y1P7pq/?p=52](https://www.bilibili.com/video/BV1Vf4y1P7pq/?p=52)