

CS219 Project3: BMP Image Processing

蔡易霖 12312927

April 2025

目录

1 功能的理解与拓展	2
1.1 调整亮度	3
1.2 融合图片	3
1.3 图像缩放	4
1.4 调整对比度	5
1.5 灰度化	5
1.6 颜色反转	6
1.7 图像裁剪	6
1.8 旋转图像	6
1.9 添加边缘阴影	7
1.10 修改基准色	8
1.11 添加虚化	8
1.12 功能的命令行输入	10
2 优化与提速	10
2.1 数据储存方式对运行速度的影响	11
2.2 SIMD 与 openMP 优化	12
2.3 数据溢出	13
2.4 神秘条纹	15
2.5 优化成果分析	16

基于 DeepSeek 的开始

拿到题目之后，我感受到了一些手足无措，于是先将这次的 project 文件丢给了 DeepSeek。在一通分析之后，DeepSeek 给了我一段可以运行的代码，可以执行调整亮度和融合图像的操作。

为了检测 DeepSeek 给我的代码的合理性，我创造了一个纯色的 test.bmp 文件作为测试，运行的结果和我想象的结果别无二致：

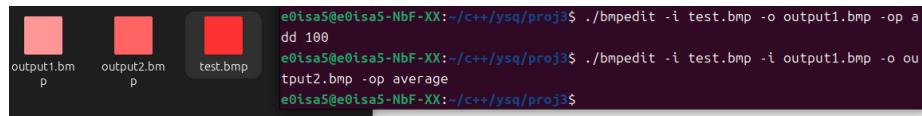


图 1：运行结果如上，看起来运行的非常合理！

令人悲伤的是，DeepSeek 似乎已经把这个 project 做的很好了，那我又能做什么呢？幸好很快我就意识到，我经常会使用 ps 等图像处理软件，可以先从为我的图像处理程序添加一些功能入手。

1 功能的理解与拓展

为了创造新的功能，我首先观察了 DeepSeek 给我生成的代码中用于储存图片的数据类型。我发现 BMPImage 类型中包含 BITMAPFILEHEADER 和 BITMAPINFOHEADER 两个类。BITMAPFILEHEADER 类包含了 BMP 文件的头部信息，其包含了文件的全局信息和位置信息。BITMAPINFOHEADER 类包含了 BMP 文件的具体信息，包含图像大小，压缩方式，分辨率等等。依据这些信息，BMPImage 类可以记录图片大小 width 和 height 与像素信息 data。值得一提的是，图片的像素信息是以一维数组的方式储存的，每一个像素点有蓝绿红三个参数，于是对于每一个像素点，数组中会有连续的三位装着它的信息，分别代表蓝绿红的参数。

这里有一点很有意思，DeepSeek 刚开始给我的代码对于所有操作都会输出一张上下颠倒的图片。这一度让我感到很奇怪，查阅资料之后才知道 BMP 格式会默认从下到上储存行的数据，这与一般的直觉是不符的。于是，我在输入 BMP 文件的时候将接受的数据反转了，这样接收到的数据会按照像素点从左到右，从上到下，按行输入的方式储存，并且三个像素点的数据

会按照蓝绿红的顺序相邻储存。

观察了 DeepSeek 的代码之后，我发现它是对输入进行拆分，然后解析-op 后面的字符串进行对应操作。因此，我添加操作只需要写好对应的函数，然后在字符串识别的时候加上对应选项并连接到对应函数就好了。

1.1 调整亮度

为了实现调整亮度的功能，我将对每一个像素点的数值分别进行修改。由于之前我已经用结构体封装了 BMP 图像类型，我只需要对对应结构体的 data 指针指向的数组的每一个数值进行修改就能起到调整亮度的效果。

为了验证实际的效果，我需要一张 BMP 图片，于是我找了一张直接从相机拍出来的照片。但是我发现 linux 系统下无法直接将 JPG 图片转化为 BMP 图片，于是我让 DeepSeek 写了一个脚本帮我进行了图片的格式转化。之后，我使用转化后的图片验证了我的程序：

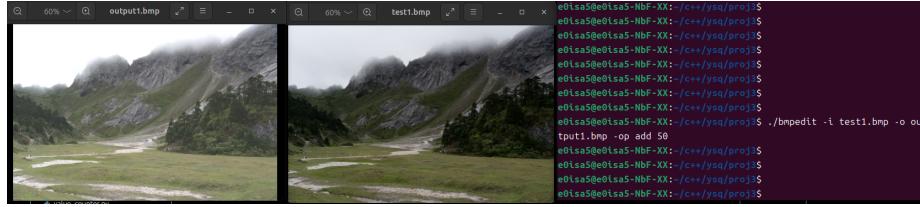


图 2：增加亮度操作

可以发现，生成的图片明显比原来的图片更亮了，同时图片的基本细节得以得到了保存。

1.2 融合图片

融合图片也是题目的要求，但我却犯了难：如果两个图片尺寸相同就还好说，如果两个图片的尺寸不同，我该如何融合呢？直接报错，还是缩放后融合呢？经过漫长的思考，我决定保留两种思路：对于融合图片，其只接受相同大小的图片的操作，但是程序也将提供对图片的缩放操作，进而允许让两个图片大小变得相同以进行融合。

对于融合图片操作，其实比较简单。只需要对于每一个 data 的数据让两张图片的数据进行加和并求平均就好了。为了测评这个函数的效果，我又找了一张 bmp 图像，测试的融合结果如下：

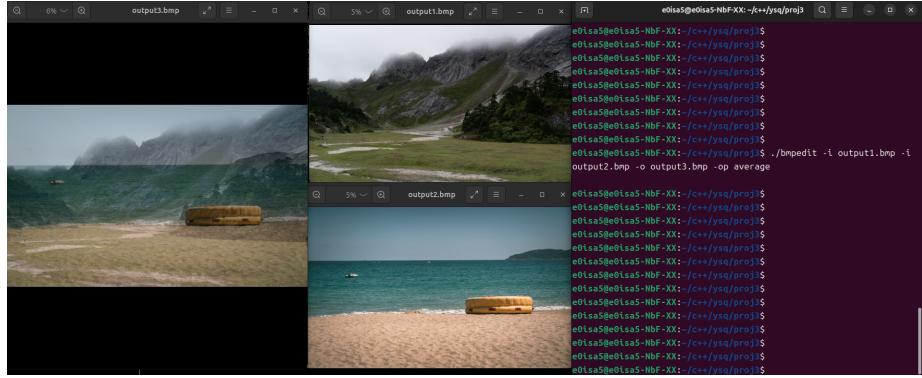


图 3: 融合照片操作

我们可以再融合得到的图片中明显的看到两张图片的影子，这非常好！

1.3 图像缩放

为了允许图片的大小变得相等，我认为我有必要引入缩放函数。实现缩放函数所需要的其实是找到新生成的图片对应的像素是原来图像对应的哪个位置的像素，所以我计算出了 x 和 y 分别的缩放比，再拿现在的位置乘上缩放比得到原来的位置的坐标，最后复制对应位置的像素就好了。

实现了这个操作之后，我尝试了一下缩放图片：

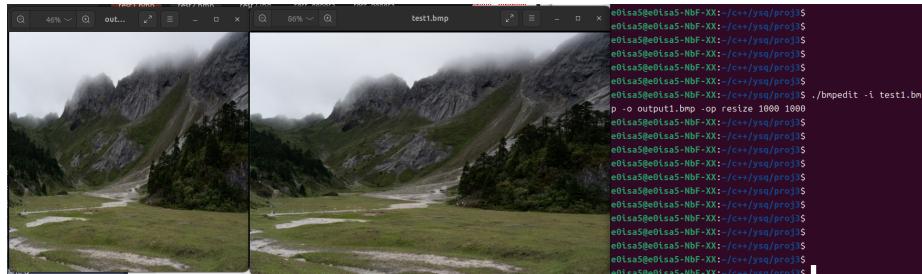


图 4: 缩放图片操作

我们可以发现两张图片的总体十分相似，只有缩放的比例不同。

1.4 调整对比度

在使用图像编辑软件调整人像的时候，可以通过拉对比度来获得更好的图像质量。调整对比度的方法是以 128 为基准，让 data 中的数据尽可能远离 128。新的数据的计算公式是 $data = \phi(data - 128) + 128$ 。其中的 ϕ 是输入的参数，代表了拉高对比度的程度。

拉高对比度的结果如下：



图 5：提高对比度操作

可以发现提高对比度后的图片阴暗的部分变得更暗，空白的部分变得更白，这便是对比度提高了的标志。

1.5 灰度化

将图片灰度化会将图片变成灰白色的色调。为了让图片达到这个效果，有一个公式能够转化 data 中的每一个数据：

$$gray = 0.299red + 0.587green + 0.114blue$$

也就是说，我需要按顺序将 data 中的元素分别乘上 0.299, 0.587, 0.114，再把得到的值赋给三个通道，就可以把图片变成灰色！以下是灰度化的结果：

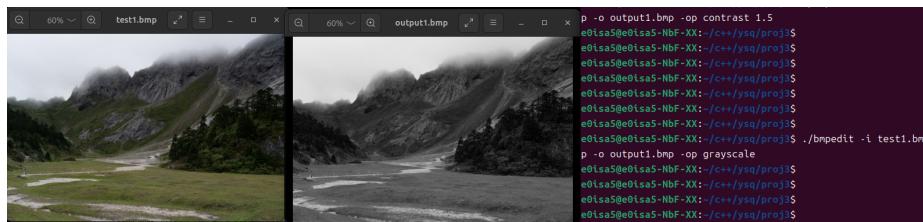


图 6：灰度化操作

1.6 颜色反转

颜色反转其实是一个没什么用的功能，但是它可以生成非常炫酷的效果！所以我把它做成了一个功能。其根本原理是将 data 中的每一个数据都变为 255 减去其本身。其效果如下：

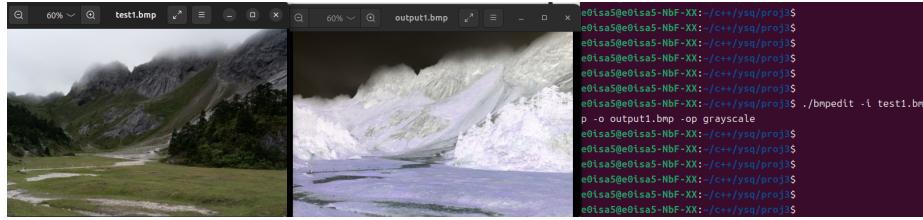


图 7：颜色反转操作

可以发现图片在这样在操作之后会变成一个非常炫酷的状态！

1.7 图像裁剪

图像裁减不涉及到 data 数据的转换，比较重要的是找到每一个像素对应原来的坐标。只要减去起始点的坐标就能做到这一点。以下是一个裁减的例子：

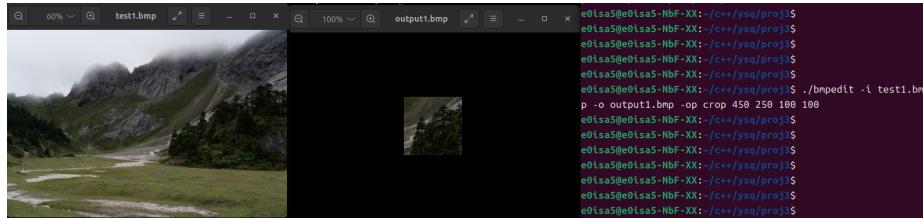


图 8：裁减操作

此处裁减的是图像中间的那棵树的位置。

1.8 旋转图像

旋转图像和裁减操作其实接近，无非是为每一个目标图中的像素点找到它们原来的位置。顺时针旋转时， (x, y) 会变成 $(y, width - 1 - x)$ ；逆时针旋转时， (x, y) 会变成 $(height - 1 - y, x)$ 。依据这个，我实现了旋转操作，以下是一个旋转的例子：

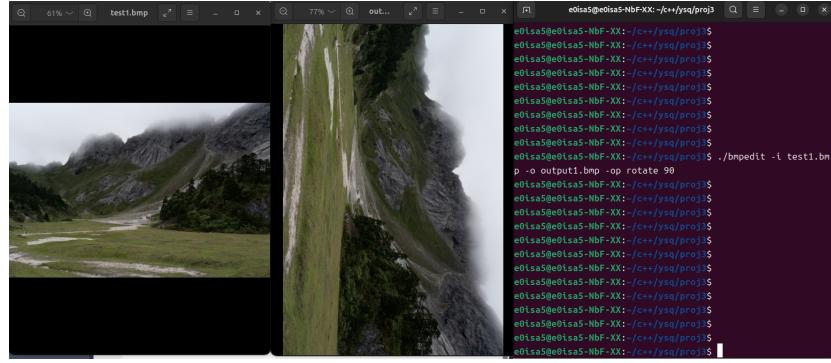


图 9: 旋转操作

1.9 添加边缘阴影

在使用图片编辑软件时，有一个常用的功能：在图片的边缘添加阴影，进而突出图片中心的内容。为了实现这一功能，我对于每一个像素点计算了它与图片中心的距离，并且通过这个距离依照距离越远，阴影越重的规律，对于阴影应当重的地方降低亮度处理。

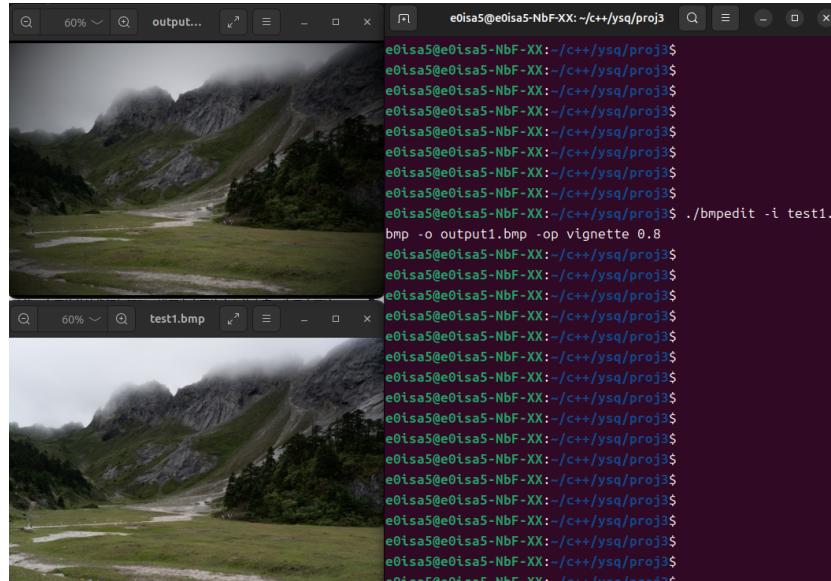


图 10: 边缘阴影化操作

观察图片，会发现其边缘出现了阴影。这样操作后的图片会突显出中央

的景象。

1.10 修改基准色

所谓修改基本色，原本指的是直接在红，蓝，绿三个基本的底色中直接进行修改，但我发现实现起来只需要找到对应通道，将对应颜色的通道的数据值乘上对应的比例就能达成修改基准色的操作。

以下是修改基准色的操作：

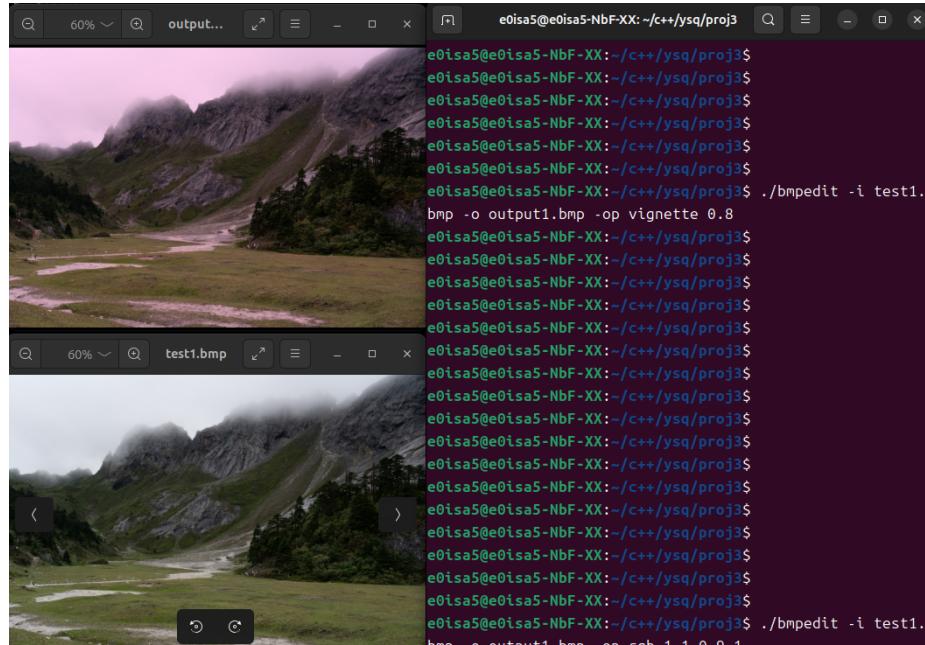


图 11：修改基准色操作

1.11 添加虚化

添加虚化的功能，也就是打码，可以通过将区域内像素的值替换为周围像素的平均值来起到平滑图像，减少细节的作用。其操作如下：

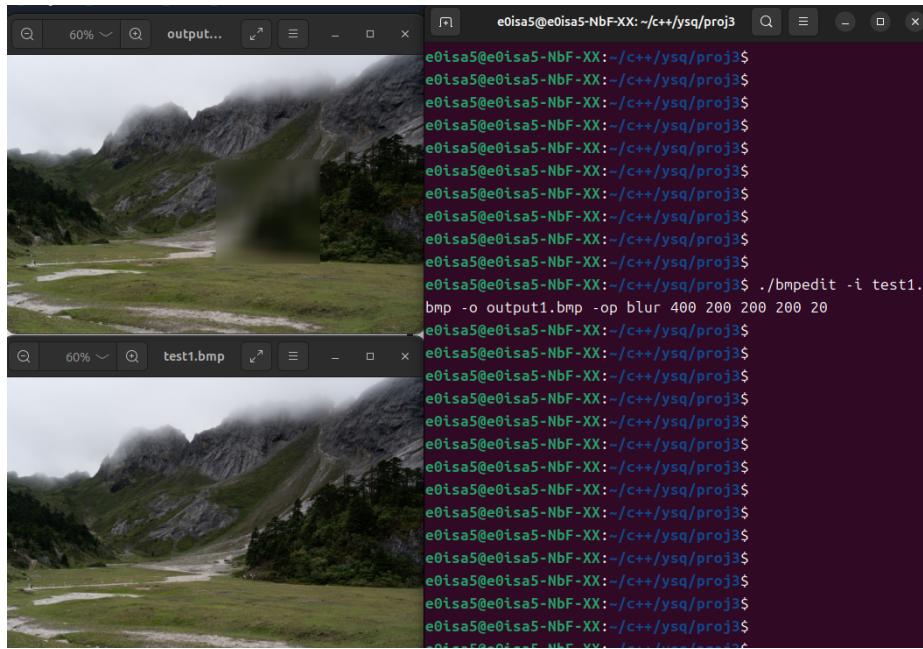


图 12: 添加虚化操作

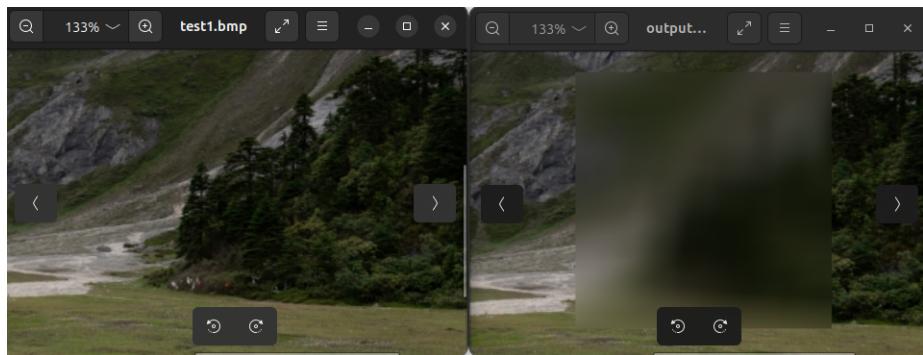


图 13: 对应位置对比

可以发现，图片的对应位置被修改，起到了保留轮廓但是去掉细节的作用。

1.12 功能的命令行输入

我比较自信我的命令设计还是比较简洁的，但是有一些参数的输入还是需要给出提示的。因此，用户可以通过输入./bmpedit 或者./bmpedit -help 获取命令帮助：

```
e0isa5@e0isa5-NbF-XX:~/c++/ysq/proj3$ ./bmpedit -i bmp.bmp -help

Usage:
  Basic operations:
    ./bmpedit -i input.bmp -o output.bmp -op <operation> [parameters]

  Supported operations:
    - add <value>          Adjust brightness (-255 to 255)
    - average               Blend two images (requires two -i inputs)
    - contrast <alpha>       Adjust contrast (recommended 0.1-5.0)
    - grayscale             Convert to grayscale
    - invert                Invert colors
    - crop x y w h         Crop image
    - rotate <degrees>      Rotate image (90, -90, 180)
    - resize new_w new_h    Resize image
    - vignette <intensity> Apply vignette effect (0.0-1.0)
    - rgb r_scale g_scale b_scale  Adjust RGB channels
    - blur x y w h radius  Apply partial blur

Examples:
  Vignette effect:      ./bmpedit -i input.bmp -o output.bmp -op vignette 0.8
  RGB adjustment:       ./bmpedit -i input.bmp -o output.bmp -op rgb 1.2 0.9 1.0
  Partial blur:         ./bmpedit -i input.bmp -o output.bmp -op blur 100 100 200 200 3
```

图 14：命令帮助

2 优化与提速

其实我发现，图像处理在某种程度上就是一种矩阵处理。因此，我可以使用 SIMD 和 openMP 来进行图像处理的优化与加速。

2.1 数据储存方式对运行速度的影响

对于图像的每一个像素点，有红绿蓝三个通道。这就产生了一个问题：对于一个 $n \times m$ 的图片，我应该把一个像素点的三个通道放到一起，最后变成一个大小为 $3(n \times m)$ 的向量 data 的交错储存，还是用三个矩阵分别装这三个通道，形成三个 $n \times m$ 的图片的平面储存？

我使用两种存储方式针对分别试了一下，结果如下：

图像大小	交错/ms	平面/ms	优化后的交错/ms	优化后的平面/ms
1000×600	0.93	1.49	0.75	0.78
2000×1200	3.95	6.42	2.94	2.97
3000×1800	9.36	13.65	7.55	6.88
4000×2400	17.32	24.19	13.70	13.03
5000×3000	23.92	38.03	21.61	20.98
8000×4800	60.50	96.14	56.48	55.52
10000×6000	94.36	150.28	86.46	84.99
12000×7200	134.26	217.63	125.58	123.39
15000×9000	210.24	344.31	197.37	196.27
20000×12000	374.45	607.79	351.69	349.25

表 1: 运行时间

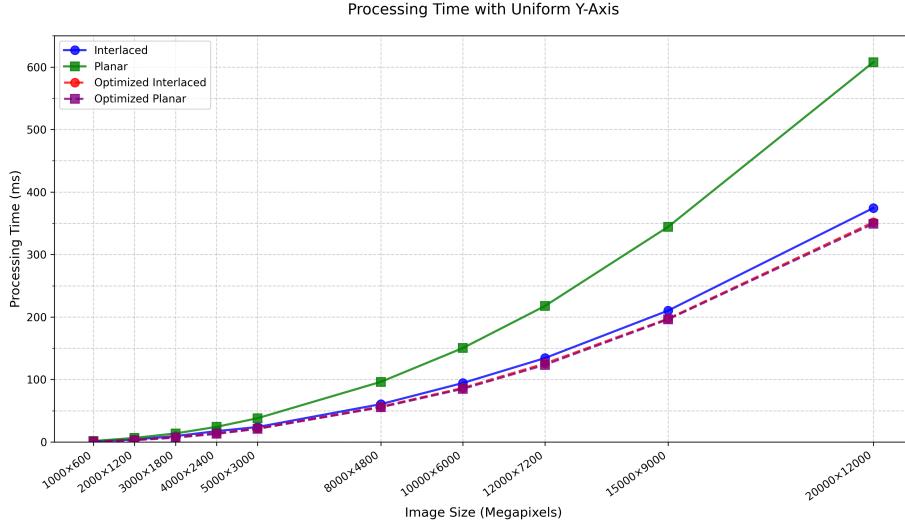


图 15：四种情况下的运行时间随数据大小的变化

曲线大致呈现与数据量大小正相关的关系，这看起来还是很合理的。我发现，分别处理三个通道所花费的时间要比交错处理慢上 50%。而在经过优化之后，二者的时间消耗则近乎相同。最终，我使用了平面处理，因为 SIMD 针对的是对于很多数据的相同操作，而相同通道的操作是相同的，因此把相同的通道放在一起有益于 SIMD 的优化。

2.2 SIMD 与 openMP 优化

SIMD 和 openMP 的操作都比较繁琐，所以我让 DeepSeek 为我生成了 SIMD 和 openMP 优化的基本操作。然而，当我检验 DeepSeek 给我生成的结果的时候，我发现它无法将像素点正确地放回图中，进而会生成一些非常奇怪的图片。



图 16: 错误的图像

仔细的排查后，我发现错误的根源是因为图片有红绿蓝三个通道，而 SIMD 的操作都是针对 32 位的数据块，DeepSeek 在调用的时候错误地想要以三个一组填入 33 位的数据，进而导致了错误。修正了这一点将数据对齐之后，我成功实现了用 SIMD 的寄存器运算得到正确答案。

2.3 数据溢出

还有一个错误是在实现提高对比度操作的时候，我发现 DeepSeek 给我的程序会实现反色的效果

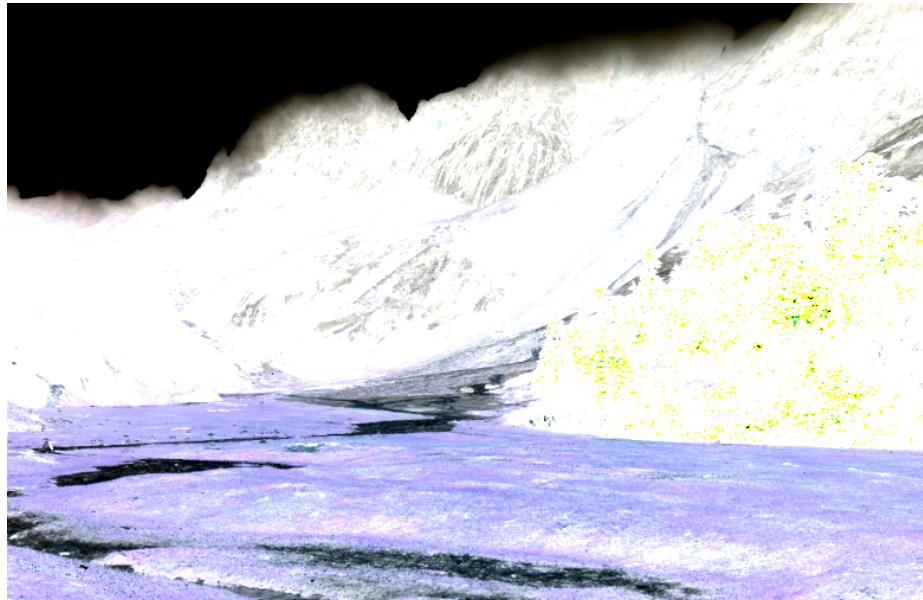


图 17：错误的图像

这个错误排查了很长时间，我通过输出大量中间变量，才发现问题出在小数乘法的部分。

提高对比度的参数是一个 float 类型的变量，需要用 float 与经过处理的 data 值相乘来改变图像的对比度。为了实现 float 的乘法，DeepSeek 使用了 Q15 类型来进行运算，其原理是使用定点数，通过位移留出小数点后的位置，在进行乘法之后自动左移 15 位得到近似的答案。为了使用 Q15 进行乘法，我将输入的数据左移了 8 位，将参数乘上 32768 转化为 Q15 格式，再进行运算。这在逻辑上没有什么问题，但是虽然输入的数据是在与 128 进行减法之后的数，进而不会溢出，但是如果参数大于 1，乘上 32768 后会变成 16 位，进而溢出到符号位上造成错误。正负刚好相反，进而导致了反色的结果。

我思考了很久，没有想出一个很优雅地解决这个问题的方法。因此，我决定先将参数乘上 32768 的一半，在运行完成位移 15 位完成之后再将得数乘 2。这样做的原因是因为对于这里的乘法操作，其溢出的值刚好是 2 倍，因此只要针对 2 倍进行修改就好了。这固然不是一个很完美的解决溢出的方法，但是可以在不牺牲空间的前提下以小常数级的时间牺牲修复这个问题，我认为也是可以接受的。

2.4 神秘条纹

在实现边缘阴影的优化的时候，依据 DeepSeek 生成的代码，我的程序会将图片变成下图这样：



图 18：很对的图片... 吗？

这样看似乎这张图片没有问题，但如果将图像放大，就会发现图像上密密麻麻地分布着竖向的直线



图 19：像这样子

很奇怪的是，DeepSeek 坚持认为他给我的代码没有出现错误。这就令

人感到十分费解了。在排除了图片处理软件本身的问题之后，我输出了代码 SIMD 优化中的中间量来进行运行检测。

```

813
814 // 合并低 64 位和高 64 位到 128 位中
815     _m128i result_r = _mm_unpacklo_epi64(packed_low_r, packed_high_r);
816     _m128i result_g = _mm_unpacklo_epi64(packed_low_g, packed_high_g);
817     _m128i result_b = _mm_unpacklo_epi64(packed_low_b, packed_high_b);
818
819     printf("result_r-0: %d\n", _mm_extract_epi16(result_r, 0));
820     printf("result_g-0: %d\n", _mm_extract_epi16(result_g, 0));
821     printf("result_b-0: %d\n", _mm_extract_epi16(result_b, 0));
822     printf("result_r-1: %d\n", _mm_extract_epi16(result_r, 1));
823     printf("result_g-1: %d\n", _mm_extract_epi16(result_g, 1));
824     printf("result_b-1: %d\n", _mm_extract_epi16(result_b, 1));
825     printf("result_r-2: %d\n", _mm_extract_epi16(result_r, 2));
826     printf("result_g-2: %d\n", _mm_extract_epi16(result_g, 2));
827     printf("result_b-2: %d\n", _mm_extract_epi16(result_b, 2));
828     printf("result_r-3: %d\n", _mm_extract_epi16(result_r, 3));
829     printf("result_g-3: %d\n", _mm_extract_epi16(result_g, 3));
830     printf("result_b-3: %d\n", _mm_extract_epi16(result_b, 3));
831     printf("result_r-4: %d\n", _mm_extract_epi16(result_r, 4));
832     printf("result_g-4: %d\n", _mm_extract_epi16(result_g, 4));
833     printf("result_b-4: %d\n", _mm_extract_epi16(result_b, 4));
834     printf("result_r-5: %d\n", _mm_extract_epi16(result_r, 5));
835     printf("result_g-5: %d\n", _mm_extract_epi16(result_g, 5));
836     printf("result_b-5: %d\n", _mm_extract_epi16(result_b, 5));
837     printf("result_r-6: %d\n", _mm_extract_epi16(result_r, 6));
838     printf("result_g-6: %d\n", _mm_extract_epi16(result_g, 6));
839     printf("result_b-6: %d\n", _mm_extract_epi16(result_b, 6));
840     printf("result_r-7: %d\n", _mm_extract_epi16(result_r, 7));
841     printf("result_g-7: %d\n", _mm_extract_epi16(result_g, 7));
842     printf("result_b-7: %d\n", _mm_extract_epi16(result_b, 7));
843

```

图 20: 添加的调试代码 (这里的%d 是伏笔)

我在中间每一步运算都加入了中间量输出，最终发现中间数据的运算确实是没有错误的。这就令人十分费解了：为什么我会使用正确的数据生成错误的图像呢？在经过长达三小时的长考之后，我突然想起来：我的图像像素值大小都是小于等于 255 的，所以我用的是 unsigned char 来装载这些参数。然而，在运算过程中，DeepSeek 和我都下意识地使用了 int 类型来分割寄存器中的数据。对于一个正的小于等于 255 的整数，其高八位必然都是 0，而八位的 0 又代表了 unsigned char 中的一个 0，0 所代表的又是黑色，进而会产生黑色像素。如此有规律地生成黑色像素的结果就是最终得到黑色条带。

这样简单的错误会难倒 DeepSeek 是让我没想到的，或许是 DeepSeek 关于寄存器操作的数据集不多，对这类操作不熟练？然而在我准备质疑 DeepSeek 的时候，在后面添加虚化的优化实现上，我原以为由于其操作复杂，计算并不直接，DeepSeek 会难以正确实现。但是在我告诉 DeepSeek 观察并模仿其他模块的操作并优化虚化操作之后，DeepSeek 居然直接正确地完成了对虚化操作的优化，令人不得不惊叹于 DeepSeek 恐怖的执行力！

2.5 优化成果分析

完成了所有的优化之后，我分别运行了优化前后各个模块的代码，其时间比较如下：

操作对象 10000×6000	优化前运行时间/ms	优化后运行时间/ms
调整亮度	1368.77	9.53
融合图片	196.59	38.58
图片缩放	186.55	47.34
调整对比度	535.65	92.6
灰度化	71.09	8.81
颜色反转	12.22	7.59
图像裁减	213.13	30.15
旋转图像 90 度	483.60	346.12
旋转图像 180 度	96.18	23.75
添加边缘阴影	350.25	8.06
修改基准色	171.69	12.34
添加虚化	101866.12	11229.54

表 2: 操作方法与运行时间

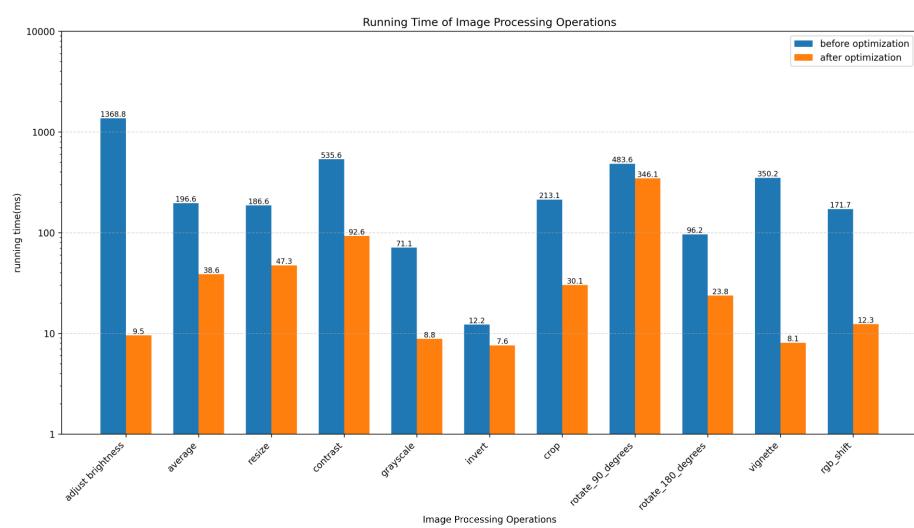


图 21: 图表比较

可以看到，在一些操作上，比如旋转 90 度，优化的效果并不明显。至于原因，我在思考后认为像旋转 90 度这样的操作初始化的时间复杂度较高，使用 SIMD 与 openMP 初始化的代价较大，故优化效果不显著。好在从结果上来看，我的优化成功地让每一种操作都跑的更快了！

结语

写到这里的时候是凌晨六点，我的头脑微胀，心里却是很喜悦的。这次 project 对比前两次 project 对我更有吸引力，因为这次是让我做了一个比较实际的图像编辑软件。在完成之后，我用其编辑了我的一些图片，给我带来了无可比拟的快乐，因为这次的程序让我真正找到了一点“我写的代码是有用”的感觉。因此这次 project 的结尾，我要放上一张使用我的程序编辑处理得到的照片，以此祈祷下次的 project 也这么有趣～

我会把用到的工具函数放到 <https://github.com/10a5/CS202-Advanced-Programming>

