项目五报告

吕梓🄵 12313605

2025 年 6 月 22 日

1 前言

本次项目是针对项目四的延伸，在测试过程中我不免觉得，自己的优化做的还是不怎么样。和 Opencv 对比，几乎对于全阶段从小到大的图像，我都全面落后他十倍左右。这让我非常非常的好奇，opencv 到底用了多么极端的手段来让图像处理变得那么那么快。我从网上下载了我非常喜欢的四张图片，尺寸分别为，50×50,427×640,1920×1228,6000×4000，涵盖了从大到小的图像。测量了从轻量级到重量级的三种运算，分别是调整亮度与对比度，图片融合，图片尺寸变换，分析他们在花费的时间。这也是我第一次尝试写 python 和 Rust 语言，对他们的理解的感受也加深了。

2 测试过程

2.1 测试环境

本次测试的设备是 MacBook Pro，装载了 M4 pro 芯片，24GB 运存。使用了苹果自带的 python 解释器，版本为 3.13，以及自带的 clang 编译器。使用 Cmakelist 链接动态库并生成可执行文件。使用了 “-O3” 和 “-march=native”，是为了在编译层面加快程序运行速度。使用 “-fopenmp”，确保了并行计算。在 python 中，使用 opencv-python 库，直接使用 opencv 库中的函数。在 rust 中也使用了并行和 SIMD 加速。并使用了”cargo build –release” 提高了编译器优化。

2.2 运行测试说明

在测试的过程中，我设置了一系列的措施来使测试时间稳定。随着计算机组成原理的学习，我了解到了缓存 (cache) 在程序运行过程中的作用，如果我连续测试相同的内容，可能因为 Cache Hit 导致运行时间变短，不稳定，也就会得到错误的结果，尤其是在实际使用中，需要处理的图像往往都不会被重复处理。我查阅电脑的说明书，找到了我的电脑的三级 cache 加在一起大概有 100MB，所以我使用 python 生成了一个 108MB 的垃圾图像 (没有任何意义)，仅仅用来在每次检测操作完成后，填满 cache。为了防止编译器将它优化掉，我还对他进行了操作并输出。此外，对于 python 语言来说，他具有以引用计数为主，以标记清除和分代收集为辅的 gc 机制，与 java 的有共同点。所以在 python 的测试环节，为了防止 gc 突然启动导致程序中断，我在进行操作前禁用了 gc，并在结束后启动 gc 回收。因为 python 是脚本语言，在函数调用环节的开销非常的大，所以为了减少对我们原本目标–opencv 库的测试，我改掉了 gpt 生成的 python 程序中的函数。且对于实验数据的检测都严格在 cpu 利用率小于 4%，内存占用率小于 50% 的情况下测量。任何超过这个范围的数据都会被抛除。

对于测试数据的处理我是这样操作的：我先测试 10 组数据，扔掉最大的三组，和最小的两组，对于剩下的五组取平均值。这样就减少了很多冷启动以及系统噪声对程序运行的影响。

2.3 运行测试数据展示

在这里以检测 python 获得的实验数据为例，展示我采集的实验数据。

数据统计		
football	50x50	
python		
adjust	blend	resize
0.012	0.02	0.033
0.01	0.02	0.038
0.009	0.02	0.035
0.012	0.028	0.038
0.018	0.026	0.04
0.013	0.028	0.038
0.009	0.02	0.037
0.009	0.019	0.038
0.01	0.019	0.038
0.014	0.019	0.047
0.0116	0.0219	0.0382

图 1: python 数据 1

campus	6000x4000	
adjust	blend	resize
9.818	12.636	30.202
10.345	11.951	30.38
11.429	13.896	26.572
9.623	14.127	30.792
10.158	14.045	30.356
9.417	13.88	30.642
9.822	14.59	30.356
10.167	12.848	30.792
10.003	13.871	31.468
11.315	17.846	32.464
10.2097	13.969	30.4024

图 2: python 数据 2

3 测试结果分析

3.1 I/O 分析

要对图片进行处理，首先就要将图片从磁盘中读入。所以我们就先分析分析 I/O 的过程和效率吧。

3.1.1 Imread

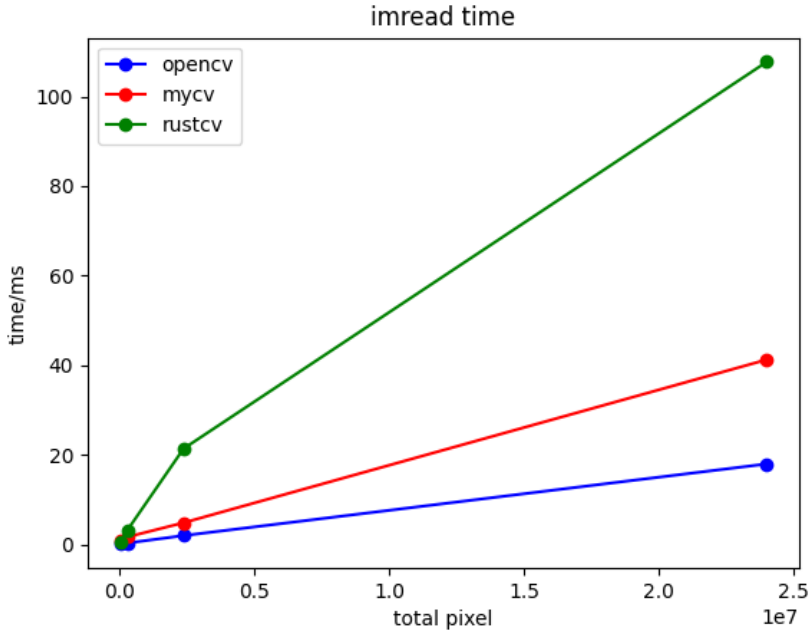


图 3: imread_time

在这个折线图中，绿色线代表 rust 的读取，红色线代表我的 C++ 库的读取，蓝色代表 opencv 库的读取。图像中我们可以清晰的看到，rust 的 image 库的读取在每个图片下的表现都不是很好，而且对于图片尺寸的增加，rust 的读取时间出现了激增，而 opencv 的以及我的库，对于图片尺寸的增加过度都比较的平滑。虽然我的库的读取速度不如 opencv，但是在读取阶段没有被拉开很大的差距。下面我就来简单的分析分析，为什么 opencv 的快，为什么 rust 的慢。Rust 的慢主要是因为 image 库中的读取函数并

没有使用并行优化，所以他的读取是串行读取的，这样的操作势必是缓慢的，在对小型图片上，这种劣势并不明显。因为并行调度也需要一定的开销，所以 rust 在最小的图片上没有与另外两个甩的很远。但是在图片规模越来越大的时候，这个劣势就越来越明显了。而 opencv 使用了行缓冲区（row buffer），一次性读入图片的一整行，并且保证了内存对齐。并且大量使用了裸指针，然后位移读写，这样避免了边界检查，和索引计算。而且对于头部文件采用一次性跳转，直接跳到数据区的开头读取。而我的读取，也使用了和 opencv 一样的一些处理思想，但是处理的没有那么极致。

3.1.2 Imwrite

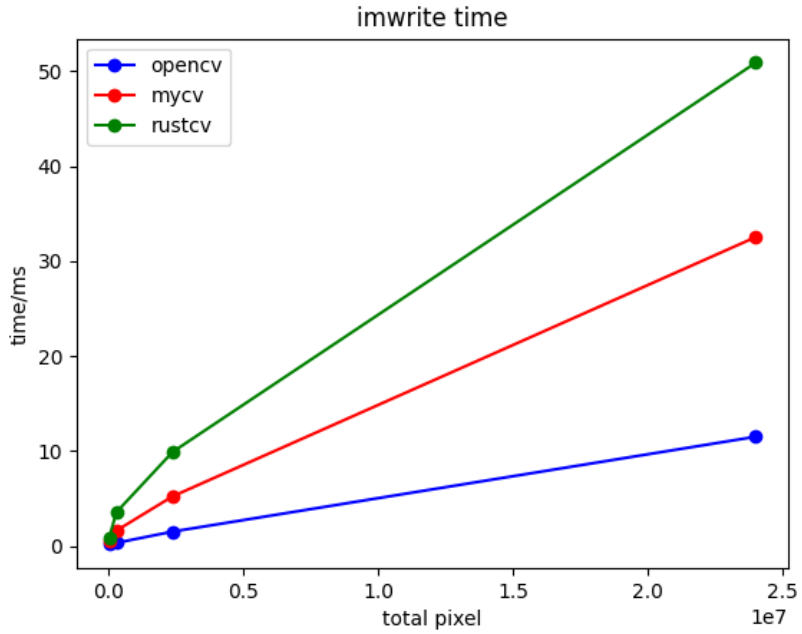


图 4: imwrite_time

opencv 还是那么的快而 Rust 依然那么慢。由于我对写文件不是很了解，所以我询问了 GPT 为什么 opencv 的写文件操作这么快。在 opencv 库中，内存已经是连续的了，所以 opencv 设计了 imencode () 将数据变为字节流，通过 fwrite 一次性写入所有内容，不用进行任何重复。我在项目三中

的输出差不多是这样的，但是我在项目四中为了兼容更多的格式，牺牲了优化（我没有想好怎么同时兼具这两点）。

3.2 运行操作分析

3.2.1 adjust 操作

adjust 操作我是这样来检测的，我让三个程序同时调整相同的亮度和对比度。然后统计他们的运行时间。

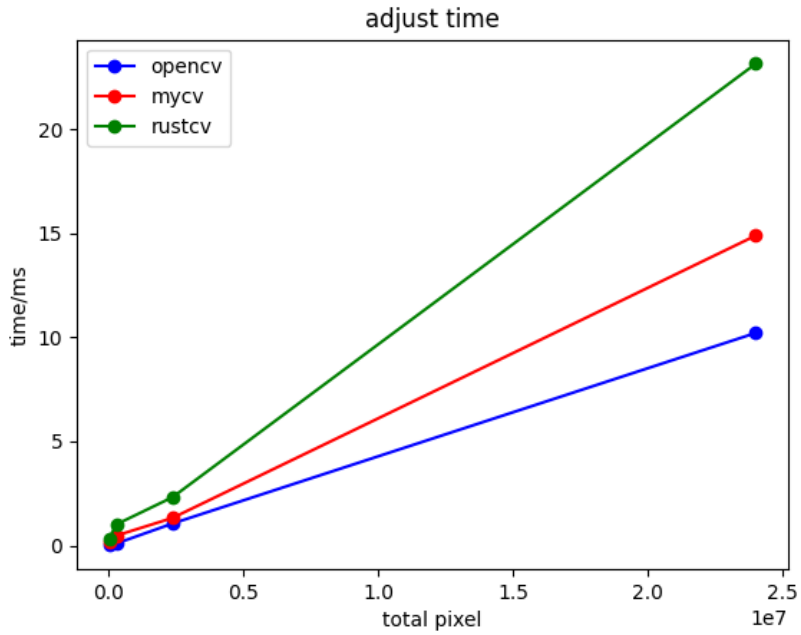


图 5: adjust_time

在检测时我发现，opencv 的图像处理在全部图上的效果都很好。我一直觉得我对小图像的兼容做的还是很不错的，那我们来看看 opencv 到底做了什么。小图像避免使用多线程，这个我的库也做了。对于小图像分支，单独写高度优化的 SIMD 路径，并且将逐像素的操作改为 inline，避免了调用函数的开销。这部分我并没有做。可能就是这部分导致了一些性能上的差距。对于 Rust 语言，首先我对这个语言不够熟悉，而且 Rust 对应的图像处理库中并没有调整亮度和对比度的操作，所以我需要自己实现一个，我使用

了并行以及 SIMD 优化，但是优化效果仍比较的有限。我在写 Rust 并行的时候，在一个我第一次遇到的报错信息中了解到，Rust 的并行计算有一种数据“锁”。Rust 在语言层面保证了安全。这给 Rust 语言带来了一些额外的锁机制和复制的开销，这进一步加大了调度开销。C++ 的并行是无锁的，当然了这可能有一点点危险，但是这应该会更快一点。python 调用 opencv 库实际上也只是使用了 C++ 的 python 接口，所以在并行等等操作上会更快。

3.2.2 blend 操作

在这个操作上，应该是我的库与 opencv 库差距最小的一个操作。我们先看看实验结果吧。

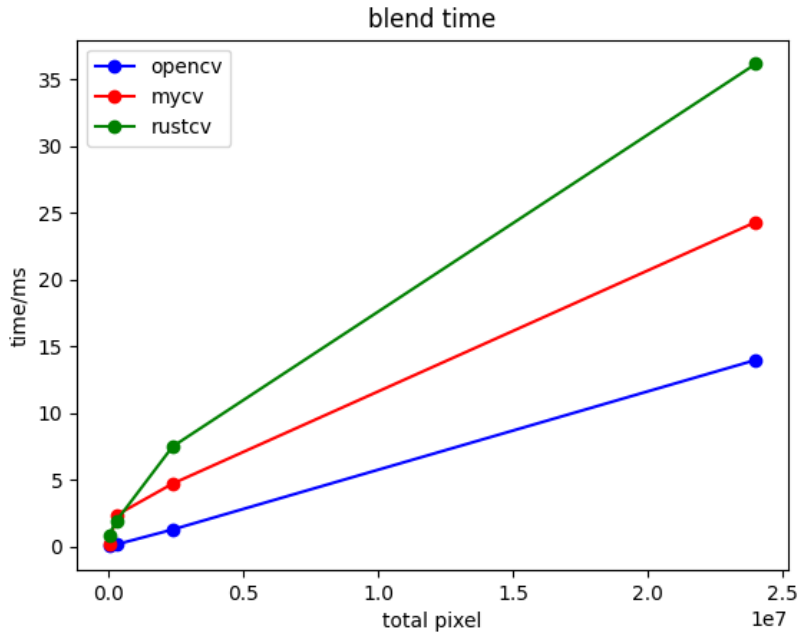


图 6: blend_time

在 opencv-python 中，是用 addWeighted 实现的，他实现的融合是可调整权重的。可以将两个图片以不同权重融合，而我的库没有实现这一点，都是等权融合。所以在计算方面，我的融合天然就比 opencv 的更简单一点。在实验中，我发现了一个现象，更加说明了我的对比度和亮度功能写的有多

么的糟糕。我的融合函数竟然比我的调整函数用时更短？而 opencv 的融合函数要比调整函数更快？经过一番分析我觉得，应该是我的 blend 功能的优化做的比较到位，而 adjust 的优化非常不合适导致的。我又把我的 adjust 函数简化为只调整亮度，也就是只执行加法的。测试之后发现，快了约 7 倍。那么我简单的就可以分析出我的 adjust 功能的性能瓶颈应该是浮点数乘法处理的不是很好导致的。Rust 里面我按照几乎相同的优化实现的，但是他还是稍慢，这可能是受语言特性的限制吧，不过也没有拉开很大差距。

3.2.3 resize 操作

这个操作是用双线性插值的方式，将图片的大小调整一下。是一个计算比较密集的操作，每个像素都要进行比较多的运算操作。我们先来看看测试的结果。

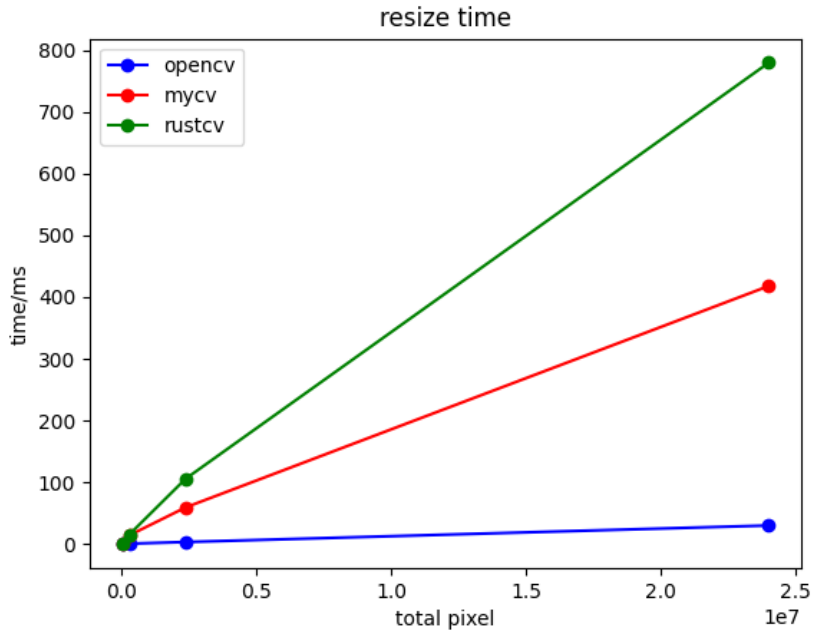


图 7: resize_time

opencv 的性能在这个图片里得到了很好的展示。我用 interpolation=cv2.INTER_LINEAR 确保使用 opencv 中的双线性插值。与我的 C++ 库和 Rust 的库相比，随着像素总数的增加，甚至变化甚至与有点像一条水平的直线。这里

其实我觉得做的最不好的是 Rust 的库，他速度实在是太慢了，可能稳定性更好吧。我又去学习了一下 opencv 的这个双线性插值，将它与我的双线性插值做了比较，这个应该是算法上的优越。opencv 使用了预计算权重表，这个避免了对于同一像素块的多次计算，在我的 resize 里面没有实现这部分，确实是会慢很多，而且随着像素点的增加，我的 resize 功能重复计算的像素点越来越多越来越多，这也就导致了运行时间的增加。

3.2.4 其他格式的图片

本次实验中我还检测了对灰度图的操作，与预测一致的是，操作时间都节约了一半以上。

3.3 链接开销

除此之外，我再来探究一下链接开销吧。毕竟我的库是一个动态库而不是一个静态库。

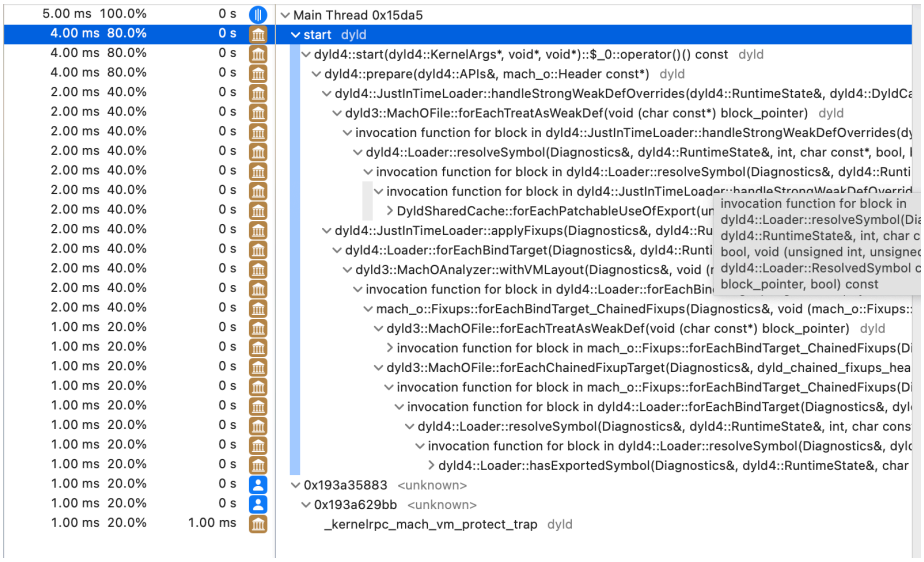


图 8: link time

我使用了 Xcode 的 Time Profiler 分析了时间，从图中我们不难看出，动态库的链接需要一定的时间开销。但是对于较大的输入图片来说，这部分可以忽略。

4 内存分析

说完了运行时间，我们来聊一聊内存吧。

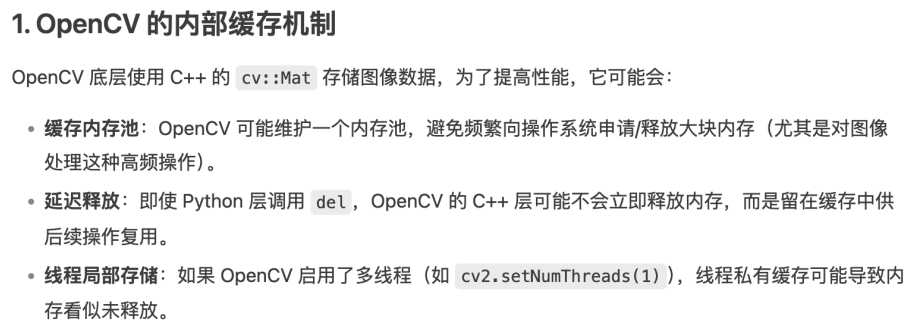
4.1 python 的 opencv 库

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     7     49.5 MiB     49.5 MiB           1   @profile
     8                                         def main():
     9                                             # 1. 获取用户输入：图像路径 + 操作类型
    10     49.5 MiB      0.0 MiB           1       image_path = "campus.bmp"
    11     49.5 MiB      0.0 MiB           1       operation = "resize"
    12
    13                                             # 2. 加载图像
    14     49.5 MiB      0.0 MiB           1       start_0 = time.perf_counter()
    15    187.4 MiB    137.9 MiB           1       img = cv2.imread(image_path)
    16    187.4 MiB      0.0 MiB           1       end_0 = time.perf_counter()
    17    187.4 MiB      0.0 MiB           1       elapsed_ms = (end_0 - start_0) * 1000
    18    187.4 MiB      0.0 MiB           1       print(f"读入耗时：{elapsed_ms:.3f} 毫秒")
    19
    20    187.4 MiB      0.0 MiB           1       if img is None:
    21                                                 print("图像加载失败！请检查路径。")
    22                                                 sys.exit(1)
    23
    24                                             #img = cv2.cvtColor(cv2.imread(image_path), cv2.COLOR_BGR2GRAY)
    25
    26    187.4 MiB      0.0 MiB           1       gc.disable()#计时前关闭gc机制，有利于测量时间的稳定
    27                                             # 4. 计时开始
    28    187.4 MiB      0.0 MiB           1       start_1 = time.perf_counter()
    29
    30                                             # 5. 执行图像处理操作
    31    187.4 MiB      0.0 MiB           1       if operation == "adjust":
    32                                                 result = cv2.convertScaleAbs(img, alpha=1.5, beta=40)
    33    187.4 MiB      0.0 MiB           1       elif operation == "blend":
    34                                                 result = cv2.addWeighted(img, 0.5, img, 0.5, 0)
    35    187.4 MiB      0.0 MiB           1       elif operation == "resize":
    36    476.8 MiB    289.4 MiB           1           result = cv2.resize(img, (img.shape[1] * 2, img.shape[0] * 2), interpolation=cv2.INTER_LINEAR)
    37                                             else:
    38                                                 print("不支持的操作类型！")
    39                                                 sys.exit(1)
    40
    41                                             # 6. 计时结束
    42    476.8 MiB      0.0 MiB           1       end_1 = time.perf_counter()
    43    476.8 MiB      0.0 MiB           1       gc.enable()
    44    476.8 MiB      0.0 MiB           1       start_2 = time.perf_counter()
    45    476.9 MiB      0.1 MiB           1       cv2.imwrite("output.bmp", result)
    46    476.9 MiB      0.0 MiB           1       end_2 = time.perf_counter()
    47    476.9 MiB      0.0 MiB           1       elapsed_ms = (end_2 - start_2) * 1000
    48    476.9 MiB      0.0 MiB           1       print(f"输出耗时：{elapsed_ms:.3f} 毫秒")
    49
    50    476.9 MiB      0.0 MiB           1       elapsed_ms = (end_1 - start_1) * 1000
    51    476.9 MiB      0.0 MiB           1       print(f"操作耗时：{elapsed_ms:.3f} 毫秒")
    52
    53    682.9 MiB    206.0 MiB           1       img2 = cv2.imread("garbage.bmp");
    54    408.3 MiB   -274.6 MiB           1       result = cv2.convertScaleAbs(img2,alpha = 1.2,beta = 10);
    55    408.3 MiB      0.0 MiB           1       cv2.imwrite("garbage_out.bmp",result)
```

图 9: python memory analysis

我使用了内存分析工具，memory_ profiler，使用命令 $python -m\ memory\_profiler\ image.py$. 测试了输入超大图像时，内存分配的情况。从上面表格中，首先 49.5MB 的是对于我使用库的加载，然后第一次大量的内存分配发生在加载图像阶段，下面就是进行 resize 操作时新建的图像。但是突然我发现，输出图像之后，内存竟然没有释放? 这是怎么回事? 难道 opencv 其实是有内存安全隐患的? 怀着这个疑问我找向了 GPT，GPT 告诉我，请放心，opencv 是内存安全的。这个问题是由于 python 的 gc 机制导致的，使用完并不会立刻清除。真的是这样吗? 我不相信，我来试试。在程序中加上了 del result，内存果然释放了一部分。但是我使用 del img，内存一点儿也没有释放。我百思不得其解，继续求助 GPT，

原来是这样，这么来看应该是没有内存泄漏的情况的。我又使用 blend



1. OpenCV 的内部缓存机制

OpenCV 底层使用 C++ 的 `cv::Mat` 存储图像数据，为了提高性能，它可能会：

- **缓存内存池**：OpenCV 可能维护一个内存池，避免频繁向操作系统申请/释放大块内存（尤其是对图像处理这种高频操作）。
- **延迟释放**：即使 Python 层调用 `del`，OpenCV 的 C++ 层可能不会立即释放内存，而是留在缓存中供后续操作复用。
- **线程局部存储**：如果 OpenCV 启用了多线程（如 `cv2.setNumThreads(1)`），线程私有缓存可能导致内存看似未释放。

图 10: GPT

和 adjust 测试了内存情况。在测试时我发现，opencv 在实现这个两个功能的时候应该是使用了浅拷贝，操作过程中并没有更多的内存分配出现，说明这只是在原图像上操作然后让 result 指向那块内存。

4.2 我的动态库

在检测我的动态库时，我使用了 gperftools 来检测内存的状况

```
Starting tracking the heap
Enter image filename: campus.bmp
Enter command (adjust | resize | blend): resize
[OpenMP] Running with 12 threads
Dumping heap profile to heap.0001.heap (137 MB currently in use)
Operation read completed in 44.909 ms.
Dumping heap profile to heap.0002.heap (412 MB currently in use)
Operation [resize] completed in 424.418 ms.
Dumping heap profile to heap.0003.heap (686 MB currently in use)
Operation write completed in 241.902 ms.
[OpenMP] Running with 12 threads
Dumping heap profile to heap.0004.heap (Exiting, 162 kB in use)
```

图 11: run time

从这里可以看到，我的内存使用是比较多的。这是因为我在每个程序都新建了一个新的图像，然后将操作的结果放在这个新的图像中。从这个情况来看，我的程序确实会比 opencv 慢，因为我需要分配新的内存空间，然后把处理完的图像放在新的内存里面。我这种处理方式其实是一种变相的“深拷贝”，所以这部分多出来的操作势必会增加运行的时间。

4.3 Rust 的内存使用

关于 Rust 的内存检测，在 MacOs 上我没有找到比较合适的工具，所以这部分内容在 window 系统下使用 jemalloc 来进行检测的。由于我的 Rust 的很多函数跟 C++ 的动态库一样，他们共享相同的逻辑。Rust 实际上的内存使用甚至少了一点点。这是由于 Rust 的编译器自动优化了内存布局，比如重排列字段之类的操作。但是总的来说，从内存使用的情况上看，与 C/C++ 是相似的

5 Rust? Or C++?

Rust 到底好不好? 我的答案是，好! 对于我来说，我觉得他唯一的缺点是，有点难学。下面我来说说我的观点，首先我们来说说 Rust 的现状吧，他目前来说使用的人数有点不够，大概只有 7% 的程序员是 Rust 程序员。而 C++ 的使用人数是 Rust 的数倍以上。由于 C++ 的使用人数太多，所以可能有些声势浩大? 根据微软的数据，这个世界上百分之七十的安全问题，均来自内存安全。C++ 在这方面真的被 Rust 给打爆了，他太自由了，但是不是每个人都是 Bjarne Stroustrup，都对 C++ 运用自如。此外 Rust 还有线程安全，真的好安全，甚至有可能出问题的地方都需要加 unsafe {}。此外我觉得 Rust 有一个特别特别好的地方，就是他适合协作。我相信每一个同学都有过组队 project，队友写了一坨让人非常不适的代码，可能有很多各式各样的错误，而 Rust 语言，非常的棒，因为如此严格的语法要求，让你几乎可以完全信任你的协作者，他写的再烂，不会让程序崩溃。从一些现状来看，Rust 的生态是在逐渐完善的。首先就是 Rust 基金会成立了，企业都在支持 Rust 语言的发展。此外 Rust for linux 正在推进，截止现在，Rust 的代码已经被合并入了 linux 的内核主线。Google 已经在安卓设备中使用 Rust 语言写驱动内核，华为和微软等等企业也都有转向 Rust 的趋势。微软的首席技术官在推特上发文表达了微软希望转向使用 rust 的意愿。从微软的首席技术官我们也可以了解到 Rust 在逐渐受到欢迎。Rust 在嵌入式系统中也有很大的优势，但是我不是很了解嵌入式，在这里就不过多的去讲了。Rust 编译器的优化能力也很强，这让垃圾程序员也变得不是那么垃圾了（譬如我）。

总的来说，Rust 对于我来说真的是一门性能很不错而且很安全的语言。我个人比较看好 Rust 的前景。但是短时间内 Rust 也很难做到对 C++

的毁灭性替代，毕竟底蕴有时候也很重要，很多系统和平台的底层都是用 C/C++ 来写的。Java，Go，Python 的一些库也都是 C/C++ 来实现的，比如我们刚刚测试的 opencv-python。

6 总结

这是本学期的最后一次项目了，从第一次的计算器到现在的动态图像库，我真的学到了很多东西。可以说这是我大学两年以来最值得的一门课。在这最后一次项目中，我分析了我自己的动态库，也明白了语言学习道路还很长，我自己写的库确实也不是那么的好。这次项目 ai 也帮了我很多，我的测试工具经常出现一些莫名其妙的问题，都是 GPT 帮我完成了这部分任务。此外虽然上了几次 Rust 课，但是我还是觉得自己的 Rust 写的很差，所以在写 Rust 时，ai 也帮我完成了很多东西。这个学期还有一点比较大的收获就是，我学会了正确使用 ai，学会跟他交流，向他学习。