

Cross-Language Performance Analysis in Image Processing

Student ID: 12312710
Name: 胥熠/Brighton
Course: CS219: Advanced Programming
Date: 2025 年 6 月 1 日

目录

1	前言	2
2	实验设计与环境	2
2.1	实验设计	2
2.2	实验环境	3
3	开发实现与测试	3
3.1	图像处理实现	3
3.2	开发效率评估与打分	6
4	性能测试与分析	6
4.1	测试基准	6
4.2	运行时间与优化水平	7
4.3	运行阶段评分	11
5	运行安全分析	11
6	实验结论	12

1 前言

有人说, “Life is short, use Python.” 但也有人认为不尽如此。你怎么看? 请写一篇不少于 800 字的文章, 题目自拟...

如果说 Project2 的侧重点是了解 C 语言和 Java 语言实现区别, 通过分析源代码和性能表现来挖掘了两种语言根本上的差异, 理解了 C 是一个相对底层且效率高的语言, 但这并不代表它能在任何条件下都有最快速度。那么 Project5, 作为 “Advanced Programming” 课程的结尾, 侧重点则更偏向于这门课所涉及到的三种不同语言的开发。

我们需要从一个综合的视角批判性的分析不同语言的开发效率、运行速度与内存管理的优缺点。而这三个优点, 正好对应上了 Python、C++ 和 Rust 三种语言的 (传闻中的) 特性。但传闻毕竟是传闻, 实际上手起来是不是这么一回事呢? 我们需要实践来验证。

2 实验设计与环境

2.1 实验设计

跨语言开发的对比是一个相对不好量化的东西, 每一种编程语言都会有长处和短处。因此, 我设计了一个结构化评分表, 并在本次实验每一个步骤中给对应语言打分, 最后颁奖!

角度 \ 语言	Python	C++	Rust
开发者角度 (8 分)	?/8	?/8	?/8
运行角度 (8 分)	?/8	?/8	?/8
安全角度 (4 分)	?/4	?/4	?/4
总分	?/20	?/20	?/20

可以看到, 上述表格中。我采用了三种角度来量化这个语言的优劣。下面我将详细解释每个角度的含义以及如何打分。

- 开发者角度 (8 分): 对于开发者来说, 追求主要是用尽量少的学习成本, 尽量少的写代码时间实现核心功能。因此, 在进行这次实验的时候, 需要关注与开发效率相关的问题。例如, 我需要写多少行代码才能完成主要功能? 我的学习曲线陡峭吗? 我能否快速找到相关的文档和资料? 这些问题其实是很隐性而且主观的, 但却是一个开发者在选择语言时需要考虑的关键因素。

比如说, 我现在想画一个统计图, 或者想要做一个 Gradient Descent 的步骤示意图, 我肯定不会想用 C++, 因为 Python 的 Matplotlib 太方便快捷了。所以, 这一栏我将会在后续实验中, 根据我的开发体验来 (主观) 打分。

- 运行角度 (8 分): 对于用户来说, 我们非常希望能够程序使用最少的硬件资源, 最快的配置速度、运行速度来完成任务。因此, 我们将采用性能分析的手段来量化每个语言的运行速度和内存占用, 并按照这个结果客观打分。具体性能分析方法将在后续实验中详细介绍。

- 安全角度 (4 分): 我们需要关注语言的内存安全性, 以及其他可能出现的问题。我将会分析这三个语言的内存安全性, 并给出一个我认知范围下的分数。Rust 的内存安全性是其一大卖点, 而 Python 和 C++ 则有各自的优缺点, 但这毕竟不是大多数需求下所考虑的主流问题, 因此量化总分较少。

2.2 实验环境

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, 支持 OpenMP 以及 AVX2 指令集。

- **Python:** 我们使用 Python 3.9 以及 OpenCV (4.11.0.8)。OpenCV 的核心算法主要是由 C/C++ 实现, OpenCV 的代码优化程度非常高, 包括对特定硬件循环的优化以及并行, 调用起来也非常简单, 通常只需要几行代码就能完成复杂的图像处理任务。同时, Python 是一个著名的脚本语言, 而且实际运行性能取决于 OpenCV 库, 因此我们直接使用 PyCharm 内的运行按钮运行。
- **C++:** 我们使用上一个 Project 中实现的图像处理库, 因此语言标准是 C++17, g++ 编译器的版本为 8.1.0。在我的库中, 运行方法均采用了 OpenMP 并行优化, 以及 O3 级别的编译器优化和部分 SIMD 支持, 实际速度可观但是优化程度肯定无法媲美 OpenCV。
- **Rust:** 版本为 1.86.0。Rust 可以通过 Cargo 下载第三方图像处理库, 分别为 image(0.24) 和 imageproc(0.23)。这个图像库支持直接调用高斯模糊, 但是增加亮度和边缘检测功能则是部分手动实现。同时需要指出的是 Rust 不支持 OpenMP 并行化, 因此我们使用 rayon(1.5) 库来实现并行化处理。

Rust 编译阶段提供了很丰富的性能优化选择。通过修改 Cargo.toml 可以调控如: opt-level, 就是编译器优化水平, 有 0/1/2/3 档可选; lto, 意思是跨模块连接优化; codegen-units, 意思是在编译的时候分成多少并行工作单元, 如果为 1, 则很慢但是允许全面的优化; 如果为较大的值如 16, 则快但是优化水平低; 还有什么 overflow-checks, debug, panic 等选项, 可以根据需要进行调整, 我觉得这个还是比较有意思的。

对于性能测试和内存分析软件, 我们采用 Intel VTune Profiler, 他可以分析出这个程序很多运行参数。给我们提供除运行时间外的很多参考。

3 开发实现与测试

3.1 图像处理实现

为了保证代码运行效果相对统一, 所有图像处理函数都使用相同的参数。这个实验中涉及三种不同图像处理方法, 分别为亮度调整、高斯模糊和边缘检测。对于亮度调整, 偏移量 (offset) 为 50; 对于高斯模糊, 卷积核 (ksize) 大小为 5, sigma 值为 1.5; 对于 Sobel 边缘检测, 卷积核大小为 3, 阈值化 (threshold) 值为 100, 都使用 $G_x + G_y$ 计算距离。

对 Python, 调用 OpenCV 的函数进行图像处理, 调用函数代码如下:

```

# * 调用亮度调整函数
bright_img = cv2.convertScaleAbs(img, alpha=1.0, beta=float(offset))
# * 调用高斯模糊函数 ksize = 5, sigma = 1.5
blurred_img = cv2.GaussianBlur(img, (ksize, ksize), sigmaX=sigma, sigmaY=sigma)
# * Sobel边缘检测, 没有直接的函数, 分步处理
# 1. 转灰度
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# 2. 计算Sobel梯度 (输出类型为16位有符号整数, 以保留负梯度值, ksize = 3)
grad_x = cv2.Sobel(gray_img, cv2.CV_16S, 1, 0, ksize=ksize)
grad_y = cv2.Sobel(gray_img, cv2.CV_16S, 0, 1, ksize=ksize)
# 3. 合并两个梯度
abs_grad_x = cv2.convertScaleAbs(grad_x)
abs_grad_y = cv2.convertScaleAbs(grad_y)
grad_magnitude = cv2.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)
# 4. 阈值化生成二值边缘图 threshold = 100
_, edge_img = cv2.threshold(grad_magnitude, threshold, 255, cv2.THRESH_BINARY)

```

可以看到, 调用 OpenCV 内置的图像处理函数, 可以非常快速的完成图像处理, 而且这一切只需要 click 一下运行按钮, 不需要手动的去编译和链接。

对于 C++, 调用自定义的图像处理函数, 调用函数代码如下:

```

// * 调用亮度调整函数, 可以选择原生SIMD优化版本
result_img_bgr = ILib::adjustBrightness(original_bgr_img, offset);
result_img_bgr = ILib::adjustBrightness_simd(original_bgr_img, offset);
// * 调用高斯模糊函数
result_img_bgr = ILib::gaussianBlur(temp_src, ksize, sigma);
// * Sobel边缘检测, 需要先转灰度
ILib::ImageU8C1 temp_gray = ILib::convertToGrayscale(original_bgr_img);
ILib::sobelMagnitude(temp_gray, result_img_gray, ksize, grad_magnitude, true);

```

如果不考虑 C++ 库的设计本身耗时的话, 其实我的调用函数更加简洁。但如果考虑进来, 那么 C++ 会变得非常繁琐。

对于 Rust, 我们使用 image 和 imageproc 库进行图像处理, 下面是增加亮度的方法:

```

bright_img_buffer
    .pixels_mut()
    .par_bridge()
    .for_each(|pixel_out| {
        // Rgb<u8> is [u8; 3]
        for i in 0..3 {
            let val = pixel_out[i] as i16 + offset;
            pixel_out[i] = val.clamp(0, 255) as u8;
        }
    });

```

嗯，有一点怪异，逐步解读一下：首先 `brigh_img_buffer` 是 `RgbImage` 类型的变量。然后我们对这个变量调用 `.pixels_mut()` 方法，返回一个可变的像素迭代器。然后我们使用 `par_bridge()` 方法将其转换为并行迭代器，最后使用 `for_each()` 方法遍历每个像素点，并对每个像素点的 RGB 值进行偏移处理。额，这给我一种套娃的感觉，很不直接，把 `for` 循环写入一个函数还是有点新奇的。

然后我们来看一下另外一个图像处理方法：

```
fn process_gaussian_blur(
    img_dyn: &DynamicImage,
    sigma: f32,
) -> Result<RgbImage, image::ImageError> {
    let img_rgb = img_dyn.to_rgb8(); // Convert to Rgb<u8>
    let blurred_img_u8 = filter::gaussian_blur_f32(&img_rgb, sigma);
    Ok(blurred_img_u8)
}
```

Rust 语法有点不同，但有点意思。首先是函数声明，是先变量名，然后对象类型。

最重要的是这个：“`-> Result <RgbImage, image::ImageError>`” `Result <T,E>` 是一个内置 tuple, `T` 预期返回结果，而 `E` 为错误类型。也就是说，返回值可能是两种类型之一：成功时返回 `RgbImage` 类型的图像，失败时返回 `image::ImageError` 类型的错误信息。

这是 Rust 的一个语法巧思，要求函数调用者必须显式地处理错误情况，如果说我直接使用类似“`let processed_img = process_gaussian_blur(&img_copy, sigma);`”使用这个函数，而不处理错误的话，编译器会爆出以下错误：

```
error[E0308]: mismatched types
--> src\main.rs:174:70
|
174 | ...Some(DynamicImage::ImageRgb8(processed_rgb_img));
|           ~~~~~ expected
|           `ImageBuffer<Rgb<u8>, Vec<u8>>`, found `Result<ImageBuffer<Rgb<u8>, ...>, ...>`
|           |
|           arguments to this enum variant are incorrect
|
= note: expected struct `ImageBuffer<_, _>`
        found enum `Result<ImageBuffer<_, _>, ImageError>`
note: tuple variant defined here
```

我比较喜欢这个特性。C++ 里面的 `throw exception` 虽然也能实现类似功能，`try...catch` 语法也能处理异常，但他没有强制要求函数调用方处理这个异常。而 Rust 则是强制要求函数调用方处理异常，从代码的角度保证了函数的安全性。也就是说，想要调用这个函数，我必须进行以下操作：

```
let processed_rgb_img = match process_gaussian_blur(&img_copy, sigma) {  
  Ok(img) => img,  
  Err(e) => {  
    println!("高斯模糊处理失败，使用原始图像：{}", e);  
    img_copy.to_rgb8() // 使用原始图像作为备选  
  }  
};  
// 或者说，实在不想处理异常的话，也可以用?运算符：  
let processed_rgb_img = process_gaussian_blur(&img_copy, GAUSSIAN_SIGMA)?;
```

对于 Sobel 函数，则是上述两种方法的混合，就不展开了。

3.2 开发效率评估与打分

这个非常主观。我由于第一个学习的语言是 Batch（母语？），天生对 Python 语言就有一种亲切感。Python 的语法非常简洁明了，几乎不需要任何额外的配置就能运行起来，并且支持很多强大的第三方库，所以说我毫不犹豫的给 Python 的开发效率打上 8/8。

对于 C++ 来说，这是一个非常复杂的语言。它的语法很强大也很复杂，如果说比较熟悉，确实能够达到相对高的开发效率，但很显然我没有到这个水平。所以说我倾向于给 C++ 打 6/8 分。

对于 Rust，他的语法与我先前接触的 C、C++、Java 都有显著差异。在理解代码的过程中，实际学习难度很陡峭，而且由于是相对新的语言，大模型对 Rust 的辅助能力也显著弱很多。我很多 Rust 都是 Gemini 写的，但是它写完编译错误可以爆出几大页，这其实对于其他语言是相对罕见的，需要我自己人工 debug。因此我给 Rust 打上 5/8 分。

角度 \ 语言	Python	C++	Rust
开发者角度（8 分）	8/8	6/8	5/8

4 性能测试与分析

4.1 测试基准

本次实验将使用三种图像尺寸，分别为 $600 \times 400 = 240000 \text{ Pixels}$ （小面积数据，测试常数项）， $1920 \times 1080 = 2073600 \text{ Pixels}$ （常见分辨率，测试普遍情况）， $6000 \times 4000 = 24000000 \text{ Pixels}$ （测试极限规模下的速度）。其中亮度调整就是简单的标量加法，负荷比较小，主要测试语言本身开销。高斯模糊和边缘检测都使用到了卷积操作，但高斯模糊会更高强度的调用浮点数乘除操作，而 Sobel 卷积次数更多。

测试过程中。我们会进行两次的预热，能够让 CPU 频率更加稳定，然后进行五次图像处理测试去平均值。然后在这之上再做三次实验，保证测试结果的准确性。同时仅记录图像处理时间，不计入图像 IO 时间。

4.2 运行时间与优化水平

图片处理时间的实验结果如下表所示：

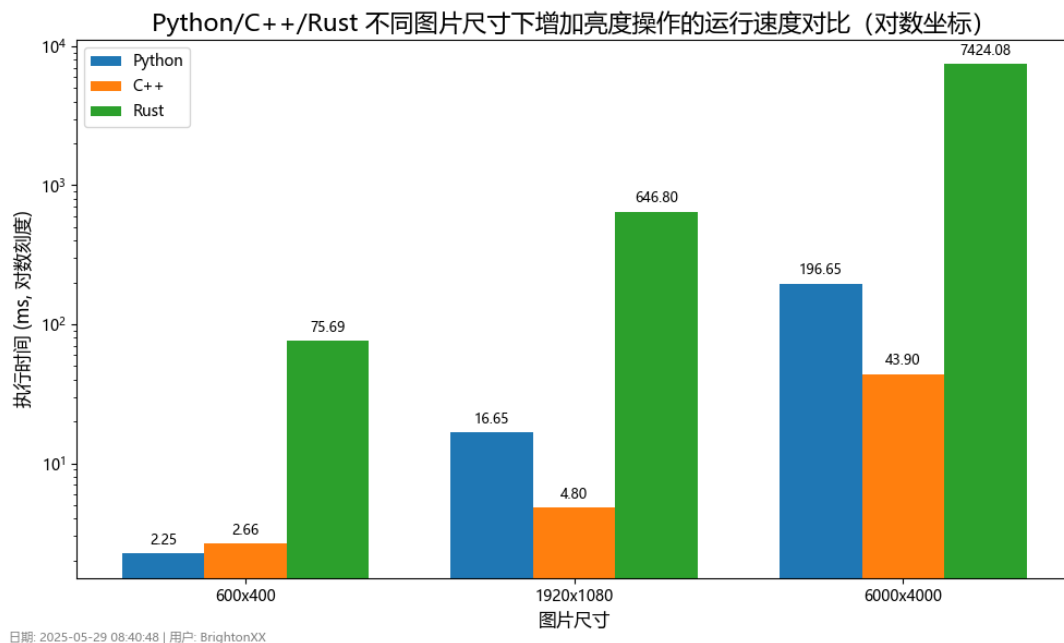


图 1: 亮度调整时间

Rust 太慢了，尽管我已经采用了最高编译器优化水平，但还是无法与 C++ 和 Python 的 OpenCV 库相提并论。同时，我的 C++ 库居然在这个操作上速度超过了 OpenCV！

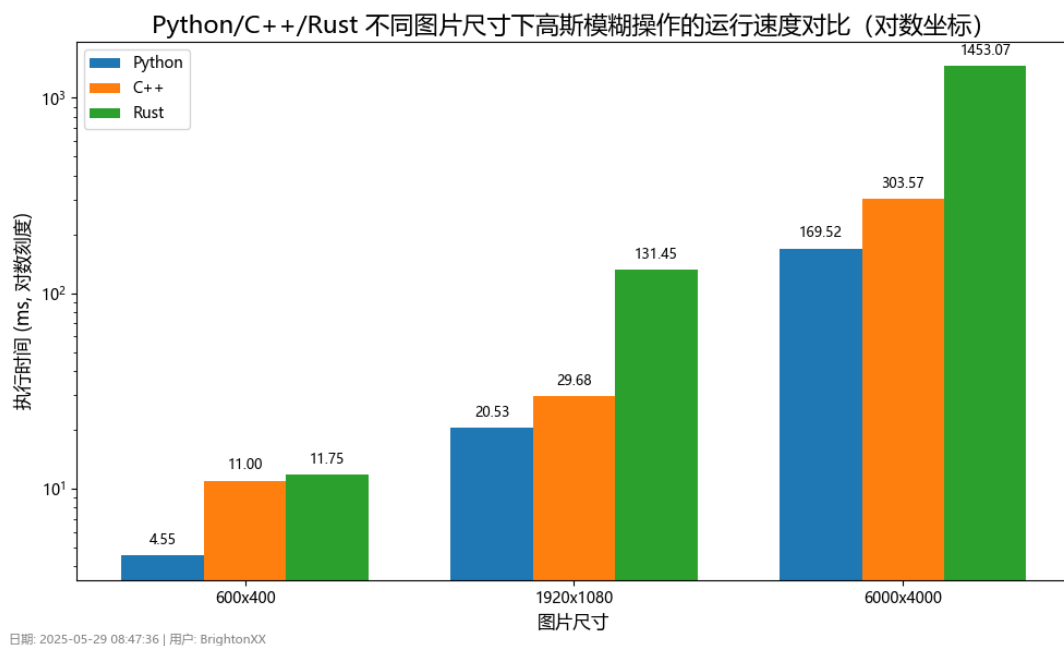


图 2: 高斯模糊时间

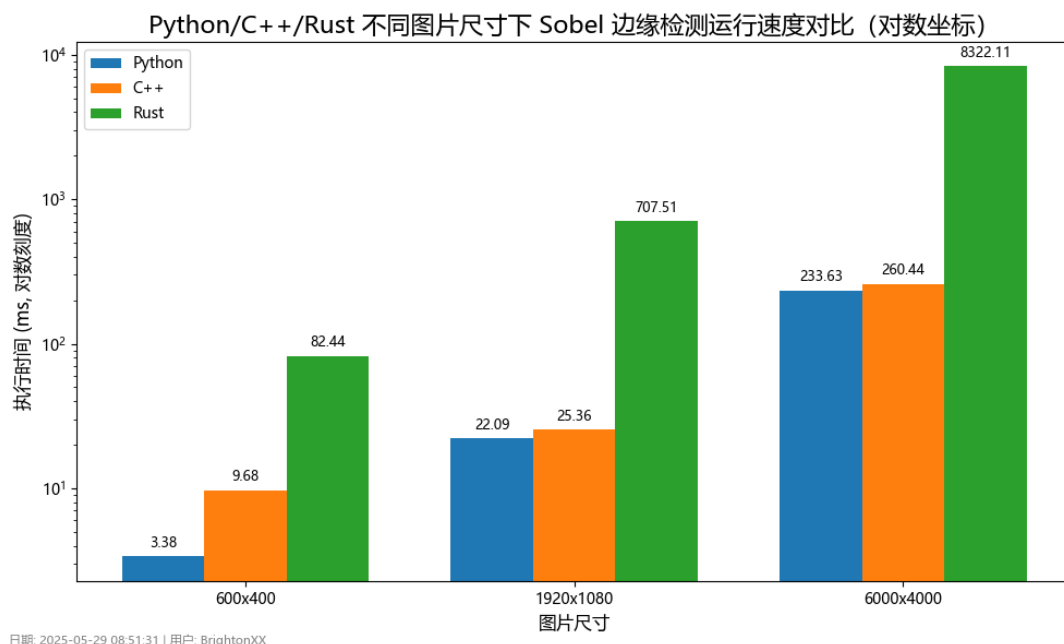


图 3: Sobel 边缘检测时间

在两个涉及到卷积复杂操作的图像处理上，我的 C++ 库就稍逊于 OpenCV 了，但是依旧比 Rust 快了很多。

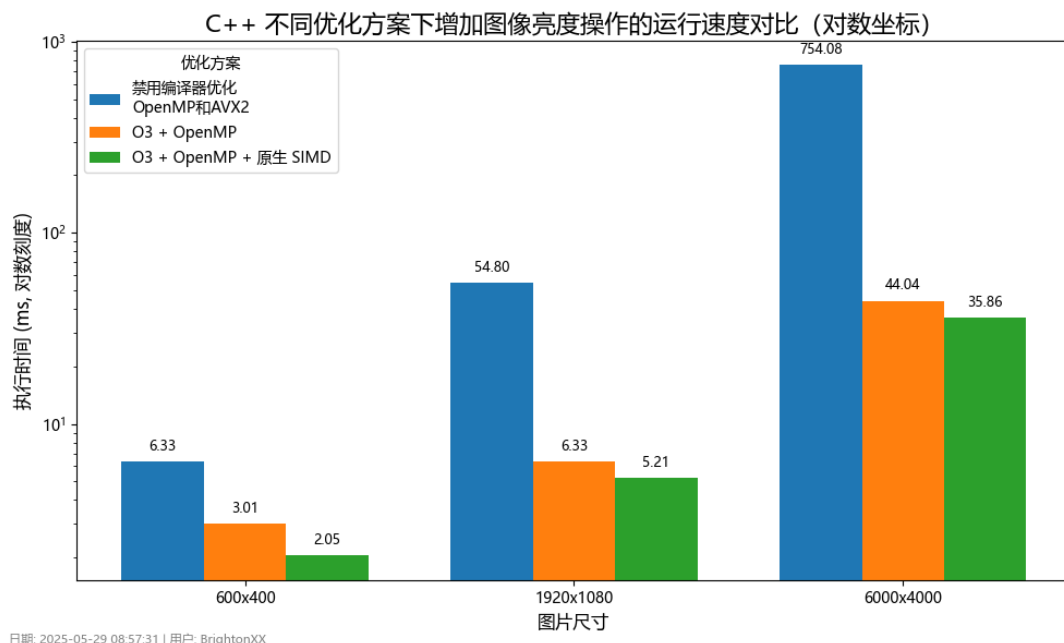


图 4: 不同优化下 C++ 的运行时间

可以看到，C++ 的运行时间在不同优化水平下有显著差异。原生 SIMD 的速度确实还是最好的，但是编译器提供的优化也非常不错了。

总的来说，但从运行速度来看，Python 的 OpenCV 库是最快的，其次是 C++，最后是 Rust。但是由于 C++ 我是使用的我自己的库，而且 OpenCV 实际代码实现也是 C++，所以说在代码

效率方面，肯定还是 C++ 更好一些。至于 Rust，它表现非常差，但这到底是什么原因呢？

我们继续使用 Intel VTune Profiler 来分析性能表现和内存占用情况，只关注：执行全部图像处理函数的需要的 Instructions 数量以及 CPI。内存占用峰值：

性能 \ 语言	Python	C++	Rust
Clockticks	19799M	4484M	2090000M
Instruction 数	21367M	1676M	437285M
CPI	0.927	2.675	4.780
内存占用峰值	398.0MB	290.3MB	273.8MB

可以看到，Rust 不知道为什么，指令数是 C++ 的 260 倍，CPI 也是最高的（越低越好）。我一开始以为是 Profiler 测试错进程了，再测试了几次，还是巨大的 Instruction 数。

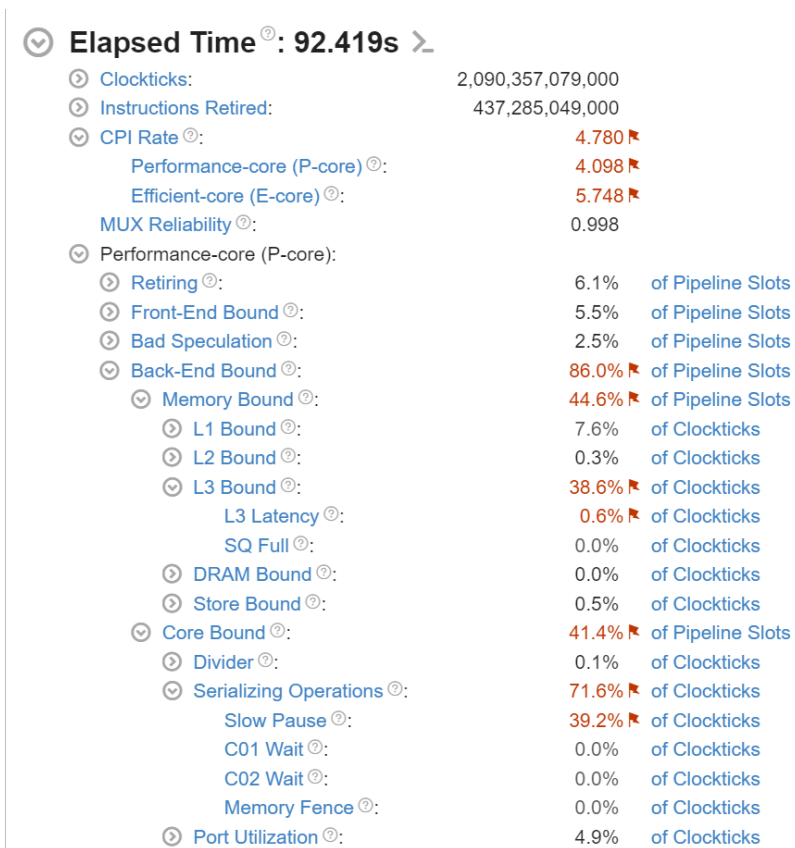




图 5: Rust Profiler 结果

我们继续深入分析 Rust 的 behaviour，分析热点代码，到底是哪里出了问题：

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time 	% of CPU Time 
<code>std::sys::sync::mutex::futex::Mutex::lock_contended</code>	<code>rust_image.exe</code>	2292.847s	92.1%
<code>func@0x1400d7680</code>	<code>rust_image.exe</code>	106.611s	4.3%
<code>core::sync::atomic::atomic_load</code>	<code>rust_image.exe</code>	26.258s	1.1%
<code>func@0x180029fbd</code>	<code>ntdll.dll</code>	19.273s	0.8%
<code>RtlWakeAddressAll</code>	<code>ntdll.dll</code>	11.498s	0.5%
[Others]	N/A*	32.714s	1.3%

*N/A is applied to non-summable metrics.

图 6: Rust 热点代码

我们发现，明明是三个图像处理函数，但是有 92.1% 的时间都花在了一个叫 `std::sys::sync::mutex::futex::Mutex::lock_contended` 上，这个函数是一个锁竞争函数，说明 Rust 的图像处理函数在多线程并行化处理时，存在严重的锁竞争问题。这有十分甚至九分的不对劲，明明亮度调整函数是各个像素独立处理的，为什么会产生进程之间竞争呢。我怀疑是库的设计的问题，anyway，我们不妨把并行化处理函数删掉，看看运行结果如何，以调整亮度为例子：

```
bright_img_buffer
    .pixels_mut()
    .par_bridge()    <<-----这一行是Rayon并行化处理函数，删除掉
    .for_each(|pixel_out| {
        // Rgb<u8> is [u8; 3]
        for i in 0..3 {
            let val = pixel_out[i] as i16 + offset;
            pixel_out[i] = val.clamp(0, 255) as u8;
        }
    });
```

删除之后，重新编译运行。运行时间回到了一个相对合理的水平。结果如下：

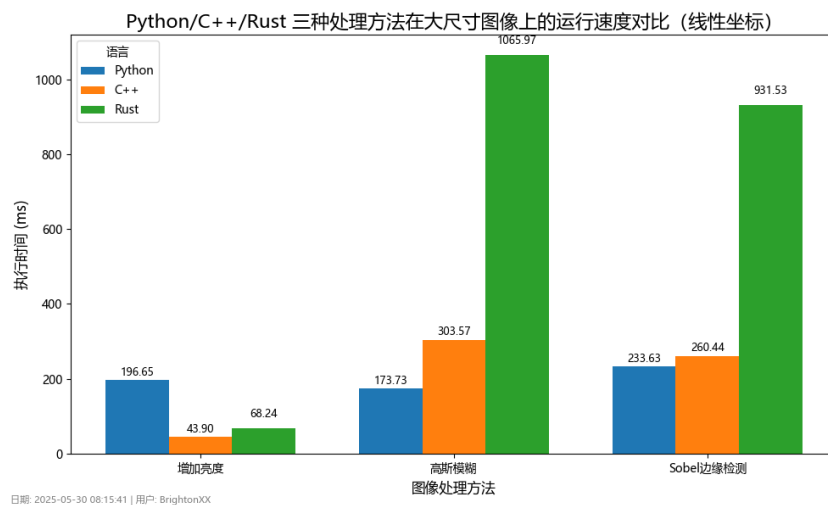


图 7: 时间对比

性能 \ 语言	Python	C++	Rust (去除 rayon)
Clockticks	19799M	4484M	38648M
Instruction 数	21367M	1676M	159985M
CPI	0.927	2.675	0.242
内存占用峰值	398.0MB	290.3MB	350.9MB

虽然说运行时间勉强能接受了，但是 Instruction 数依旧是 C++ 的接近 100 倍。尝试使用 objdump 分析了一下，发现 Rust 居然反汇编出来的代码有 45MB，足足 43 万行，而 C++ 只有 383KB。而且与 C++ 不同的是，反汇编后没有任何函数名给我定位。我怀疑我选择的优化过于激进反而会导致代码量变多，但是降低优化水平后变成了 45 万行。行吧，我有点不想分析这么大一坨东西了。

我们关注 Python 的表现，可以看注意到 OpenCV 提供的优化水平非常好，虽然 Instruction 数比 C++ 多，但是 CPI 优化到了一个不错的水平。

以及，内存占用方面，Python 确实是最高的，原因应该是 Python 的解释性语言性质和 OpenCV 库比较大有关。但其实三个应用程序之间没有差别特别多，因此我认为内存占用方面的差异可以带来的影响基本可以不用考虑。

4.3 运行阶段评分

角度 \ 语言	Python	C++	Rust
运行角度 (8 分)	7/8	8/8	4/8

5 运行安全分析

首先，虽然在 Project4 中，我使用了 C++ 的智能指针来管理内存，而且在 valgrind 中也测试了一次，这是工具告诉我的信息：

```
==1373==    definitely lost: 0 bytes in 0 blocks
==1373==    indirectly lost: 0 bytes in 0 blocks
==1373==    possibly lost: 9,424 bytes in 31 blocks
```

我认为：只有 possibly lost 是相对可以接受的。但是在多次测试中，担心的情况还是发生了：我发现运行的 wsl 所占用的内存正在以每一次程序运行增加 5MB 的速度增长。搜索之后，似乎这种现象的原因是 OpenMP 与智能指针之间会有一些冲突，导致内存无法被正确释放。总之，C++ 的内存管理一直广受诟病，这个不能总寄托于程序员的超高水平。

对于 Python，由于它是一个解释性语言，内存管理是由 Python 的垃圾回收机制自动处理的，因此我们不需要过多担心内存泄漏的问题。但是依旧有缺点，Python 的自动内存管理机制不是很高效，如果使用原生 Python 处理大量数据，会因为高频的触发垃圾回收而导致速度极度下滑。

对于 Rust，它的内存管理机制非常严格，使用所有权和借用规则来确保内存安全。同时，正如上面所说，Rust 的错误处理机制也设计的很好，强制的要求了函数调用方处理错误情况，避免了很多潜在的运行时错误。

角度 \ 语言	Python	C++	Rust
安全角度（4 分）	3/4	2/4	4/4

6 实验结论

本次实验通过对 Python、C++ 和 Rust 三种语言在亮度调整、高斯模糊和 Sobel 边缘检测三种图像处理任务上的性能进行对比分析，我们得出以下主要观察和结论：

• Python

- 开发效率：凭借其简洁直观的语法以及 OpenCV 库强大的高级函数封装，Python 在开发速度和易用性上展现出绝对的优势。同时，Python 在很多领域得益于成熟的第三方库支持，开发效率肯定是最顶级的一档。
- 运行性能：得益于 OpenCV 底层由 C/C++ 实现并经过高度优化，通过 Python 调用的 OpenCV 函数在运行速度上表现出强大的竞争力。但这个主要依附于第三方库支持，如果直接让 Python 参与到计算密集型任务，肯定是不行的。
- 内存占用：运行阶段需要占用更多的内存资源，不过在可接受范围内。归因于 Python 解释器的额外开销、其动态类型系统以及 OpenCV 库。
- 安全性：Python 有自动垃圾回收机制，虽然在效率上吃亏，但犯错机率很小。

• C++

- 开发效率：相较于 Python，C++ 的语法更为复杂，例如内存管理和各种复杂的语法，而且需要付出很多时间优化代码。虽然说是 Python 有蹭了 C++ 库之便的嫌疑，但是不可否认的是 Python 确实在大模型加持下的今天，将一些简单任务的开发成本缩减成了一个 prompt，一个 Ctrl+C+V，一个 click 的事情。
- 运行性能：通过细致的手动优化，C++ 基本可以达到目前主流编程语言最快一档。在本次测试中，C++ 版本的指令数最少，体现了其代码执行的直接性。
- 内存占用：表现不错。毕竟作为一个颗粒度更小的语言，手动管理和释放内存的结果比 Python 这类解释性语言好。
- 安全性：尽管现代 C++ 提供了诸多工具来提升内存安全，但本质上没有解决 C++ 的问题。就算使用了智能指针并且良好管理，依旧有泄露的风险。

• Rust

- 开发效率：学习曲线最为陡峭。其核心的所有权和借用系统虽然保证了内存安全，但确实是一个很新的概念，上手很有难度。在图像处理领域，当前可用的库虽然提供了基

础功能，但在 API 的易用性、高级功能的集成度以及文档的全面性方面，与 OpenCV 等成熟库相比仍有较大差距。同时由于市场现有 Rust 代码训练集不多，大模型的较低辅助能力更是为开发雪上加霜。

- 运行性能：在本次对比测试中表现最不理想。其执行所需的 Clockticks 和 Instruction 数量远超 Python (OpenCV) 和 C++，而且自带 Rayon 并行化库甚至出现了严重的锁竞争问题，导致实际运行时间大幅增加，因此得分很低。
- 内存占用：处在中间水平，虽然 Rust 的内存管理机制非常严格，但由于其编译器生成的代码量巨大，导致实际运行时的内存占用并不低。
- 安全性：其编译期的所有权和借用检查机制从根本上杜绝了如悬垂指针、数据竞争等常见的内存安全问题。此外，`Result<T, E>` 错误处理机制强制开发者显式处理可能发生的错误，极大地增强了代码可靠性。

综上所述，我对这三种语言的综合评分如下：

角度 \ 语言	Python	C++	Rust
开发者角度（8 分）	8/8	6/8	5/8
运行角度（8 分）	7/8	8/8	4/8
安全角度（4 分）	3/4	2/4	4/4
总分	18/20	16/20	13/20

这是我个人对于图像处理领域的评分，当然是非常主观的观点。Rust 虽然评分很低，但目前来看主要原因是其生态系统还不够成熟，缺乏足够的第三方库支持和社区资源，也缺乏足够的代码训练集；而且还有一个因素是他的语法确实和主流语言有着较大的差异，学习门槛高。但以上问题主要并非 Rust 语言设计的核心问题，而更多是生态和成熟度问题（但据说 OS 课 lab 已经换成全 Rust 了）。C++ 虽然开发起来难度大，但是依旧是想达到高性能的首选语言，尤其是对于图像处理这种计算密集型任务。而对于 Python，则是用的人越多，Python 越强大；Python 越强大，用的人越多。”Life is short, use Python!”

同时，与先前 Project 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] OpenCV (Open Source Computer Vision Library). *cv::Mat Class Reference*. Available: docs.opencv.org/master/class__Mat.html
- [2] Brighton *Project4: An Image Library*. Southern University of Science and Technology, 2025. [Source code]. Available: github.com/BrightonXX/SUSTech-CPP-Project

碎碎念

说来话长。22 年的 12 月，我作为中学生参加了由南科组织的一个体验营。那个下午，我在工学院的一个会议室遇到了于老师，他为我讲解南科计算机专业是如何如何。那个时候的我一定不会想到，当时埋下的种子正中了这个学期的我的眉心。

今年 1 月份，我和几位同学讨论能不能在四门专业课的基础上再加一门于 ++，周围同学都对我的身心健康表示了担忧。但我还是出于某种执念，想要试一试。

这门课程确实不枉民间的评价，我感觉给我实际带来的压力可以等效 DSAA 的两倍及以上。但我上完课之后，身心健康真的受到了打击吗？还真有，Project1 中间经历了一段漫长的错误调试，那几天的给我带来的颅内高压确实印象深刻。以及这个学期，坐在一丹对着 VSCode，一敲就是一整天。但你说我后悔选这门课吗，倒是没有这个感受。这门课的 Project 隐形地要求着高强度自学，如何写出高性能代码，如何模仿别人写出优质的报告，如何使用不同工具来满足任务需求。而我很开心确实在这门课内训练到了我的这些素质，而肉眼可见的其中很多能力在我以后的学习中是有用的，也算是一个划得来的 trade-off 吧。