基于OpenCV的Resize函数实现与优化

> **组队人数**：2~3 人
>
> **答疑联系**：钟万里 12332469@mail.sustech.edu.cn

项目简介

图像Resize 是计算机视觉领域中基础而重要的操作，广泛应用于数据预处理、缩略图生成以及多分辨率分析等场景。虽然 OpenCV 提供了高效的 resize 函数，但理解其背后的插值方法和性能优化机制对学习图像处理的底层实现有重要意义。

本项目旨在开发一个高效的 resize 函数，要求支持最近邻插值，支持放大缩小操作，并通过多线程优化提升运行效率。同时提供更多功能和优化策略，如多种数据类型的支持，双线性插值以及SIMD加速的使用。

项目要求与实现细节

基础功能（80 分）

最近邻插值实现（20 分）

- 支持最近邻插值方法 (Nearest Neighbour Interpolation)
- 最近邻插值算法实现步骤：注意此思路中 x 是水平向右， y 是竖直向下，这与矩阵的索引相反。
  i. 根据输入图像和输出图像的宽度、高度，计算图像的缩放比例：

$$scale\_width = \frac{input\_width}{output\_width}, \quad scale\_height = \frac{input\_height}{output\_height}$$

  ii. 对于输出图像的每个像素 (

$$x_{dst}, y_{dst}$$

)，通过反向映射找到输入图像中对应的最近像素：

$$x_{src} = \text{round}(x_{dst} \times scale\_width), \quad y_{src} = \text{round}(y_{dst} \times scale\_height)$$

  iii. 从输入图像的 (

x_{src}, y_{src}

) 坐标获取像素值，将该值赋给输出图像的 (

x_{dst}, y_{dst}

) 坐标

iv. 确保映射的输入坐标 (

x_{src}, y_{src}

) 不超过输入图像的范围

- OpenCV函数的实现参考：resizeNN

多通道支持（15 分）

- 支持单通道 `CV_8UC1` （灰度图）和3 通道 `CV_8UC3` （RGB 或 BGR图）
- 数据类型为无符号 8 位整型（ `uchar` ，即 `uint8_t` ）
- 使用 OpenCV 的 `Mat` 数据结构，正确读取和写入多通道图像

图像放大缩小和不定尺寸的支持（15 分）

- 在最近邻插值的基础上，支持图像的放大和缩小两种操作
- 放大缩小的尺寸不一定是整数倍
- 放大缩小不一定保持原有的长宽比例

切片和多线程加速（20 分）

- 使用 OpenCV 的 `parallel_for_` 实现多线程处理
- 对图像的行或区域进行分块并行处理，显著提升运行效率
- 实现思路：由于每个像素的处理是相互独立的，假设线程数是 `n` ，可以按照行或块均匀划分成 `n` 个区域，每个线程同时独立处理一部分图像数据。确保线程之间无冲突，任务分配合理

对比分析（10 分）

- 对比自实现函数与 OpenCV 的 `resize` 函数，自行设定测试样例，合理即可
- 测试耗时对比，以柱状图或图表展现，通过对比 OpenCV 的实现，了解其高效性能背后的原因，并发现自己实现中的优化潜力
- 至少三种不同尺寸的输入图像，至少两种不同的放大和缩小倍数，两种通道都需测试
- **注意**：文档提供的实现方法为简化版本，OpenCV的内部实现会更复杂，因此准确度对比的结果可能会不同

Bonus 功能（40 分上限）

双线性插值实现（15 分）

- 支持双线性插值方法 (Bilinear Interpolation)
- 双线性插值算法实现步骤(参考：Understanding Bilinear Image Resizing)：注意此思路中 x 是水平向右， y 是竖直向下，这与矩阵的索引相反。
  i. 根据输入图像和输出图像的宽度、高度，确定缩放比例：

$$\text{scale\_width} = \frac{\text{input\_width}}{\text{output\_width}}, \quad \text{scale\_height} = \frac{\text{input\_height}}{\text{output\_height}}$$

  ii. 对于输出图像中的每个像素坐标

$$(x_{\text{dst}}, y_{\text{dst}})$$

  ，通过反向映射找到输入图像中对应的浮点坐标（这里的0.5目的是变换时映射到几何中心，与OpenCV的实现保持一致）：

$$x_{\text{src}} = (x_{\text{dst}} + 0.5) \times \text{scale\_width} - 0.5, \quad y_{\text{src}} = (y_{\text{dst}} + 0.5) \times \text{scale\_height} - 0.5$$

  iii. 找到

$$(x_{\text{src}}, y_{\text{src}})$$

  周围的四个邻近像素的整数坐标：

$$x_1 = \lfloor x_{\text{src}} \rfloor, \quad x_2 = x_1 + 1$$

$$y_1 = \lfloor y_{\text{src}} \rfloor, \quad y_2 = y_1 + 1$$

$$\lfloor \cdot \rfloor$$

  表示向下取整
  iv. 从输入图像中获取上述四个坐标对应的像素值：

$$I_{11} = I(x_1, y_1), \quad I_{12} = I(x_1, y_2)$$

$$I_{21} = I(x_2, y_1), \quad I_{22} = I(x_2, y_2)$$

  v. 根据

$$(x_{\text{src}}, y_{\text{src}})$$

  与其邻近像素的距离，计算权重：

$$w_{x2} = x_{\text{src}} - x_1, \quad w_{x1} = 1 - w_{x2}$$

$$w_{y2} = y_{\text{src}} - y_1, \quad w_{y1} = 1 - w_{y2}$$

vi. 首先在 x 方向上对两个像素进行线性插值：

$$I_{y1} = w_{x1} \times I_{11} + w_{x2} \times I_{21}$$

$$I_{y2} = w_{x1} \times I_{12} + w_{x2} \times I_{22}$$

vii. 然后在 y 方向上对上述结果进行线性插值，得到最终的插值结果：

$$I_{\text{dst}} = w_{y1} \times I_{y1} + w_{y2} \times I_{y2}$$

viii. 将计算得到的

$$I_{\text{dst}}$$

赋值给输出图像的对应像素

$$(x_{\text{dst}}, y_{\text{dst}})$$

ix. 确保所有计算的坐标

$$(x_1, x_2, y_1, y_2)$$

均在输入图像的有效范围内，防止越界访问

- 考虑到别的插值方式，此部分OpenCV源码非常复杂，你仍可参考，也可以按照自己的理解实现：resize

支持多种数据类型（每种 5 分）

- `16U` （无符号 16 位整型）即 `CV_16UC1` 和 `CV_16UC3`
- `32F` （32 位浮点型）即 `CV_32FC1` 和 `CV_32FC3`
- 注意实现的过程中，不要发生溢出

综合优化（每种 5 分）

- 多线程优化提示：现代计算机的处理器大都分大核和小核，因此均匀的将每个任务分配到每个线程上（理解为每个核同时处理一个线程），最终的速度并不会快过小核的性能。因此，可以考虑多线程分块时，以更小的单位划分（比如至少一行或1024个像素点），然后让操作系统去调度这每个任务到不同的核心上，以提高整体的运行速度
- 内存读取写入连续性优化提示：多通道像素处理时，按行优先访问图像数据，一次行处理所有通道的像素，而非只处理一个通道，从而优化内存访问连续性，从而提升缓存利用率和整体性能
- 其他能想到的高级优化策略，要求显著提升性能

SIMD 加速（20 分）

- 此bonus适合修过计算机组成课程的同学，对性能优化和硬件指令集有一定了解或希望学习的同学
  - 使用 SIMD (Single Instruction, Multiple Data) 指令优化最近邻插值或双线性插值方法的核心计算。如256宽度的向量，一次可加载，运算和储存32个像素值，理论上速度会有32倍的提升。
  - Intel x86_64芯片使用AVX指令集、Apple Silicon M系列芯片或Arm架构使用Neon指令集
  - 实现步骤提示
    a. 使用 SIMD 指令同时加载多个图像数据到寄存器
    b. 最近邻插值：同时计算目标像素的坐标映射，直接复制对应像素值
    c. 双线性插值：同时进行加权乘法计算
    d. 使用 SIMD 指令将计算结果存回内存

OpenCV通用向量指令集（10 分）

- 考虑到SIMD指令不具备跨平台能力，一套优化在x86_64、arm和risc-v架构上需要重复实现，但常用指令的功能大都相似。OpenCV提供了通用向量指令集 (Universal Intrinsics)指令集，以实现一套优化代码，多平台共同优化的效果
- 如果你对此感兴趣，可以尝试只使用Universal Intrinsics实现一次你的SIMD优化，并在不同的平台测试加速效果

评分标准

基础功能（80 分）

模块	评分标准	分值
最近邻插值实现	逻辑正确，性能表现较好	20 分
多通道支持	支持 1、3 通道图像，数据类型为 uchar (uint8_t)，逻辑清晰，结果正确	15 分
图像放大缩小支持	支持对图片进行放大和缩小两种操作	20 分
多线程实现	使用 `parallel_for` 实现多线程处理，性能提升明显	15 分
对比分析	提供自实现函数与 OpenCV 的结果与性能对比报告	10 分

Bonus 功能（40 分上限）

模块	评分标准	分值
双线性插值实现	双线性插值支持缩小和放大，逻辑正确， 性能表现较好	15 分
支持多种数据类型	支持 `16U`， `32F`	每种 5 分
综合优化	考虑内存对齐、Cache 友好性等高级优化策略， 结果显著	每种 5 分
SIMD加速	使用 SIMD 指令优化最近邻插值或双线性插值的核心计算	20 分
Universal Intrinsics（配合SIMD）	配合 SIMD 使用 OpenCV 的 Universal Intrinsics	10 分

运行说明

函数接口

实现的函数接口如下：

```
void resize_custom(const cv::Mat& input, cv::Mat& output, const cv::Size& new_size, int interpo
```

项目提交

源代码: 项目相关的所有源文件.

项目文档: 一个简洁明了的文档，里面包括你实现的功能的用途和用法。