OpenCV-based Resize Function Implementation and Optimization

> **Team Size**: 2~3
>
> **Q&A**: Zhong Wanli 12332469@mail.sustech.edu.cn

Project Overview

Image Resize is a fundamental and important operation in the field of computer vision, widely used in tasks such as data preprocessing, thumbnail generation, and multi-resolution analysis. Although OpenCV provides an efficient `resize` function, understanding the interpolation methods and performance optimization mechanisms behind it is significant for learning the low-level implementation of image processing.

This project aims to develop an efficient `resize` function that supports nearest neighbor interpolation, both upscaling and downscaling operations, and improves runtime performance through multithreaded optimization. Additionally, it will provide more features and optimization strategies, such as support for multiple data types, bilinear interpolation, and SIMD acceleration.

Project Requirements and Implementation Details

Basic Features (80 Points)

Nearest Neighbor Interpolation (20 Points)

- Support for the nearest neighbor interpolation method.
- Steps for implementing the nearest neighbor interpolation algorithm: Note that in this approach, `x` refers to the horizontal direction (right), and `y` refers to the vertical direction (down), which is opposite to matrix indexing.
    i. Calculate the scaling factors for the width and height based on the input and output image dimensions:

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

ii. For each pixel (

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

) in the output image, find the corresponding nearest pixel in the input image through reverse mapping:

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

iii. Obtain the pixel value at (

$$x_{src}, y_{src}$$

) from the input image and assign it to the pixel at (

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

) in the output image.

iv. Ensure the mapped input coordinates (

$$x_{src}, y_{src}$$

) are within the valid range of the input image.

- Reference for OpenCV function implementation: resizeNN

Multi-Channel Support (15 Points)

- Support for single-channel `CV_8UC1` (grayscale images) and 3-channel `CV_8UC3` (RGB or BGR images).
- Data type should be unsigned 8-bit integer (`uchar` , equivalent to `uint8_t`).
- Use OpenCV's `Mat` data structure and handle reading and writing for multi-channel images correctly.

Support for Upscaling, Downscaling, and Arbitrary Dimensions (15 Points)

- Based on nearest neighbor interpolation, support both upscaling and downscaling operations.
- Scaling factors are not restricted to integer values.
- Scaling does not necessarily maintain the original aspect ratio.

Multithreading and Optimization (20 Points)

- Use OpenCV's `parallel_for_` to implement multithreaded processing.
- Divide the image into rows or regions for parallel processing to significantly improve runtime efficiency.

- Implementation idea: Since each pixel's processing is independent, assume `n` threads, and divide the image into `n` regions (rows or blocks) for each thread to process simultaneously. Ensure no conflicts between threads and distribute tasks reasonably.

Comparison and Analysis (10 Points)

- Compare the self-implemented function with OpenCV's `resize` function using test cases you define.
- Analyze and present runtime comparison results in a bar chart or table. Understand the reasons behind OpenCV's efficiency and explore optimization potential in your implementation.
- Test with at least three different input image sizes, two different scaling factors (upscaling and downscaling), and both single- and three-channel images.
- **Note**: The implementation method provided here is a simplified version, and OpenCV's internal implementation is more complex. Thus, accuracy comparison results may differ.

Bonus Features (Up to 40 Points)

Bilinear Interpolation (15 Points)

- Support for bilinear interpolation method.
- Steps for implementing bilinear interpolation (reference: Understanding Bilinear Image Resizing): Note that in this approach, `x` refers to the horizontal direction (right), and `y` refers to the vertical direction (down), which is opposite to matrix indexing.
  i. Calculate scaling factors for the width and height:

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

  ii. For each pixel coordinate

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

  in the output image, find the corresponding floating-point coordinate in the input image (The purpose of 0.5 here is to map to the geometric center during transformation, consistent with the implementation of OpenCV):

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

  iii. Find the four neighboring pixel integer coordinates:

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

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

iv. Obtain the pixel values at these coordinates:

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

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

v. Calculate weights based on the distances between

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

and neighboring pixels:

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

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

vi. Perform linear interpolation in the x-direction:

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

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

vii. Perform linear interpolation in the y-direction to obtain the final interpolated value:

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

viii. Assign the result to the corresponding pixel

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

in the output image.

ix. Ensure all computed coordinates

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

are within the valid range to avoid out-of-bound access.

- OpenCV source code reference for this section: resize

Support for Multiple Data Types (5 Points per Type)

- `16U` (unsigned 16-bit integer), i.e., `CV_16UC1` and `CV_16UC3`.
- `32F` (32-bit floating point), i.e., `CV_32FC1` and `CV_32FC3`.
- Ensure no overflow occurs during the implementation.

Comprehensive Optimization (5 Points per Strategy)

- **Multithreading Optimization**: Modern processors often have a mix of large and small cores. To improve overall speed, consider dividing tasks into smaller units (e.g., at least one row or 1024 pixels), allowing the operating system to schedule tasks more efficiently.
- **Memory Access Optimization**: When processing multi-channel pixels, prioritize row-wise data access to handle all pixel channels simultaneously, improving cache efficiency and overall performance.
- Other advanced optimization strategies that significantly enhance performance.

SIMD Acceleration (20 Points)

- This bonus is suitable for students who have taken Principles of Computer Organization courses and have a certain understanding or desire to learn about performance optimization and hardware instruction sets.
- Use SIMD (Single Instruction, Multiple Data) instructions to optimize the core computations of nearest neighbor or bilinear interpolation. For example, using a 256-bit vector, you can process 32 pixels at once, theoretically achieving a 32x speedup.
- Use Intel x86_64's AVX instructions or Arm's Neon instructions.
- Implementation steps:
  i. Use SIMD instructions to load multiple image data points into registers.
  ii. For nearest neighbor interpolation, compute coordinate mappings and copy the corresponding pixel values.
  iii. For bilinear interpolation, perform weighted multiplication calculations simultaneously.
  iv. Store the results back into memory using SIMD instructions.

OpenCV Universal Intrinsics (10 Points)

- OpenCV provides Universal Intrinsics for cross-platform optimization, enabling a single implementation to work on x86_64, Arm, and RISC-V architectures.
- If interested, try implementing your SIMD optimization using Universal Intrinsics and test the speedup on different platforms.

Evaluation Criteria

Basic Features (80 Points)

Module	Scoring Criteria	Points
Nearest Neighbor Interpolation	Correct logic and good performance	20
Multi-Channel Support	Support for 1 and 3 channel images, correct results	15
Upscaling and Downscaling	Support for both operations	20
Multithreading	Use `parallel_for` , significant performance improvement	15
Comparison and Analysis	Provide results and performance comparison report	10

Bonus Features (Up to 40 Points)

Module	Scoring Criteria	Points
Bilinear Interpolation	Supports both upscaling and downscaling, correct logic	15
Support for Multiple Data Types	Supports `16U` , `32F`	5 per type
Comprehensive Optimization	Memory, cache, or other advanced optimization	5 per strategy
SIMD Acceleration	SIMD optimization for nearest or bilinear	20
Universal Intrinsics	With SIMD, uses OpenCV's Universal Intrinsics	10

Execution Instructions

Function Interface

The function interface to be implemented is as follows:

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

Submission Requirements

Source Code: Submit all relevant source files for the project.

Project Documentation: Write a **brief** doc that explains your design choices, features, and usage instructions.