OpenCV-based Resize Function Implementation and Optimization

Team Members: 12311124 Zhang Zihan 12311624 Lin Pinhao 12312210 Liu Ruihan

---

Project Introduction

Image resizing is a fundamental operation in computer vision, essential for tasks such as data preprocessing, thumbnail generation, and multi-resolution analysis. While OpenCV provides highly optimized built-in resize functions, developing a custom resize implementation offers valuable insights into the underlying interpolation methods and performance optimization techniques. This project focuses on creating a high-performance resize function supporting both nearest neighbor and bilinear interpolation methods, enhanced through multi-threading and SIMD (Single Instruction, Multiple Data) optimizations.

---

1. Implemented Functionalities

1.1. Nearest Neighbor and Bilinear Interpolation Methods

- **Nearest Neighbor Interpolation:**
  Assigns each pixel in the output image the value of the closest corresponding pixel in the input image. This method is straightforward and fast but may produce blocky images when upscaling.
- **Bilinear Interpolation:**
  Calculates the output pixel value as a weighted average of the four nearest pixels in the input image, resulting in smoother and higher-quality resized images compared to nearest neighbor interpolation.

  Both interpolation methods are implemented in single-threaded and multi-threaded versions to balance performance and simplicity. And each method supports scaling at **any arbitrary ratio**.

  To prevent out-of-bounds access during resizing, the implementation ensures that all calculated source coordinates (`x_src`, `y_src`) are clamped within the valid range of the input image dimensions. This guarantees that every pixel mapping remains valid, maintaining image integrity.

1.3. Multi-channel Support

The resize functions support both single-channel (grayscale) and three-channel (color) images. This versatility allows the functions to handle a wide range of image types commonly used in computer vision applications.

1.4. Support for Multiple Data Types

Beyond the standard unsigned 8-bit integers (`CV_8U`), the implementation extends support to:

- **16-bit Unsigned Integers (`CV_16U`)**
- **32-bit Floating Points (`CV_32F`)**

This is achieved through C++ template programming, enabling seamless handling of different data types without code duplication. The program will **automatically** check the depth of this image.

1.5. Usage Instructions

This CMake configuration sets up a C++17 project with OpenCV dependencies, we only support arvx2 for x86-64 ecosystem enables performance optimizations through AVX2 and FMA instruction sets (for supported compilers), and ensures that the necessary OpenCV libraries are linked. It also defines the source files and compiles them into an executable. Please prepare your OpenCV environment at first.

```
~/opencv_resize_custom/build:cmake ..
~/opencv_resize_custom/build:make
~/opencv_resize_custom/build:./custom_resize
```

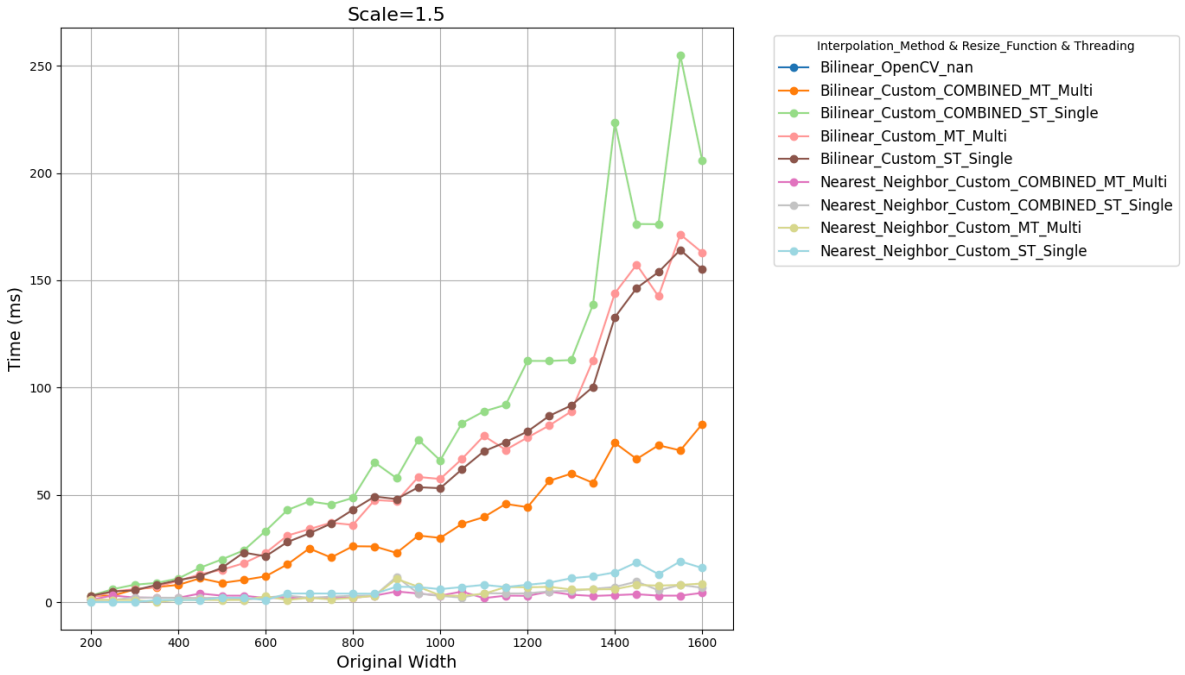
To utilize the custom resize function in your project, follow these steps:

1. **Include the Resize Module:**
   Ensure that the `resize.h` header file is included in your project and that the corresponding `resize.cpp` is compiled alongside your application.
2. **Prepare the Input Image:**
   Load the input image using OpenCV's `imread` function. The image can be single-channel or multi-channel and of supported data types.
3. **Define the Output Size and Interpolation Method:**
   Specify the desired output dimensions and choose the interpolation method (`NEAREST_NEIGHBOR` or `BILINEAR`).**Save or Display the Resized Image:**
   Use OpenCV's `imwrite` or `imshow` functions to save or display the resized image.

```
cv::Mat input = cv::imread("path_to_image.jpg", cv::IMREAD_UNCHANGED);
cv::Size new_size(800, 600);                    //output size
InterpolationMethod method = NEAREST_NEIGHBOR; // or BILINEAR
cv::Mat output;                                //output mat
bool use_multithreading = true; //for single-threaded set false, default:true
resize_custom(input, output, new_size, method, use_multithreading);
cv::imwrite("resized_image.jpg", output);
// or
cv::imshow("Resized Image", output);
```

---

2. Comparison Between Single-threaded and Multi-threaded Implementations



Key Observations:

1. Bilinear Interpolation:

- **Multi-threaded (MT)**: Performs significantly better than single-threaded mode, especially for high-resolution images (e.g., original width > 1000). The performance improvement becomes more evident as image size increases.
- **Single-threaded (ST)**: Execution time increases linearly with resolution. Due to the complexity of bilinear interpolation calculations, single-threaded performance is limited and slower compared to multi-threaded processing.

2. Nearest Neighbor Interpolation:

- **Multi-threaded (MT)**: Shows slight performance improvement over single-threaded mode, but the improvement is less dramatic compared to bilinear interpolation. This is because nearest neighbor interpolation is computationally simpler, leaving less room for optimization through parallelization.
- **Single-threaded (ST)**: Time increases moderately with resolution but remains significantly faster than bilinear interpolation in all cases due to its simplicity.

3. Single-threaded vs Multi-threaded Comparison:

- **High-resolution scenarios**: Multi-threaded processing offers significant advantages, especially for bilinear interpolation. For example, when the original width exceeds 1200, the gap between single-threaded and multi-threaded performance becomes much larger.
- **Low-resolution scenarios**: Multi-threaded advantages are less noticeable, as the overhead of thread management diminishes the benefits of parallelization for smaller image sizes.

4. Nearest Neighbor vs Bilinear Interpolation:

- Nearest Neighbor Interpolation is faster in both single-threaded and multi-threaded modes, as it only requires selecting the nearest pixel value without additional computations.
- Bilinear Interpolation, while slower, delivers smoother results by performing weighted averaging of adjacent pixels, which significantly enhances image quality.

Image Quality Analysis (PSNR Comparison):

To further evaluate the effectiveness of the interpolation methods, we compared them to **OpenCV's built-in `resize` function** using **PSNR (Peak Signal-to-Noise Ratio)** as the metric for image quality.

1. Bilinear Interpolation:
    - Achieved significantly higher PSNR values compared to Nearest Neighbor Interpolation, indicating superior image quality.
    - Particularly effective for large scaling factors, where smoother edges and better preservation of details are required.
2. Nearest Neighbor Interpolation:
    - Delivered lower PSNR values, indicating lower image quality. The results often show pixelation or jagged edges, especially at larger scaling factors.
3. OpenCV's `resize` Function:
    - Performed comparably to the custom bilinear interpolation implementation in terms of PSNR, but with shorter execution times due to better algorithmic optimizations.

Conclusion:

1. **Performance**:
    - Multi-threaded processing significantly reduces execution time for both methods, especially for high-resolution images.
    - Nearest Neighbor Interpolation is ideal for performance-critical tasks due to its low computational cost, but it compromises image quality.
2. **Image Quality**:
    - Bilinear Interpolation offers much better image quality compared to Nearest Neighbor Interpolation, as indicated by higher PSNR values. It is especially recommended for tasks requiring smooth edges and finer details.
    - OpenCV's `resize` function strikes a balance between execution time and image quality, benefiting from optimized implementations.

---

3. Optimization Strategies

3.1. SIMD Optimization with AVX

By leveraging AVX SIMD instructions, the implementation processes multiple pixels simultaneously in a single CPU instruction. Specifically, it handles **8 pixels at a time**, accelerating operations like nearest neighbor and bilinear interpolation. This data-level parallelism reduces loop iterations and speeds up intensive computations.

3.2. Advanced Multi-threading with OpenMP

Using **OpenMP**, the implementation achieves thread-level parallelism by splitting workload across multiple CPU cores. Combined with SIMD optimization, this dual-level parallelism (thread-level + data-level) fully utilizes multi-core architectures, significantly improving performance for high-resolution images.

3.3. Precomputation of Pixel Mappings

Precomputing output-to-input pixel mappings (`x_mapping`, `y_mapping`, and interpolation weights like `dx`, `dy`) minimizes redundant calculations during resize operations. These mappings are calculated once and reused, reducing computational overhead and enhancing cache efficiency.

3.4. Memory Access Optimization

Ensuring the **continuity of the input matrix** (`input.isContinuous()`):

- Optimizes memory access by avoiding scattered memory reads.
- Copies the matrix into a continuous layout when necessary, enabling faster data retrieval and reducing cache misses.

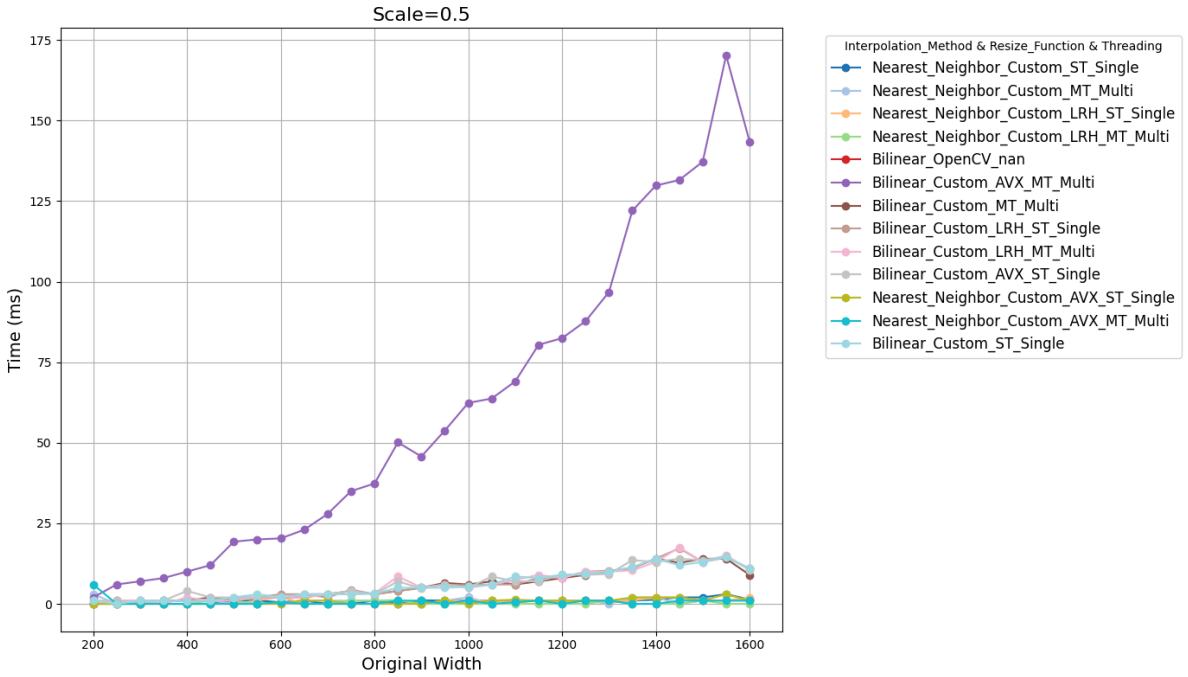
3.5. Optimizations for Nearest Neighbor and Bilinear Interpolation

Both nearest neighbor and bilinear interpolation methods were optimized to achieve high performance while maintaining flexibility for different use cases. The following key techniques were applied:

1. **Multi-threading**:
    - For both interpolation methods, **OpenMP** is used to distribute workload across multiple threads. Each row of the target image is processed independently, enabling efficient utilization of multi-core CPUs and reducing overall processing time.
2. **AVX SIMD Vectorization**:
    - SIMD instructions (AVX) are leveraged to process **8 pixels simultaneously** within a single CPU instruction. This data-level parallelism reduces loop iterations and significantly speeds up pixel-wise computations.
    - For nearest neighbor, source pixel values are batch-loaded and processed in parallel.
    - For bilinear interpolation, the formula:
      $\text{Interpolated Value} = (1 - dx)(1 - dy)I_{11} + dx(1 - dy)I_{21} + (1 - dx)dyI_{12} + dx \cdot dy \cdot I_{22}$
      is computed for **8 pixels at a time**, minimizing computational overhead.
3. **Precomputation of Pixel Mappings and Weights**:
    - To avoid redundant calculations during resizing loops:
        - **Nearest Neighbor**: Output-to-input mappings (`x_mapping`, `y_mapping`) are precomputed and stored in reusable arrays.
        - **Bilinear Interpolation**: Source pixel boundaries (`x1`, `x2`, `y1`, `y2`) and interpolation weights (`dx`, `dy`) are precomputed for each target pixel.
    - These precomputations reduce loop complexity and enhance cache performance by using contiguous memory.
4. **Channel-specific Optimizations**:
    - Separate logic is implemented for single-channel (grayscale) and multi-channel (e.g., RGB) images. This ensures that:
        - For single-channel images, unnecessary channel-handling overhead is eliminated.

- For multi-channel images, `cv::Vec` is used to efficiently process each channel independently while maintaining compatibility with vectorized operations.

Summary



After using these method, we compare to each other to verify if they could speed up our function.

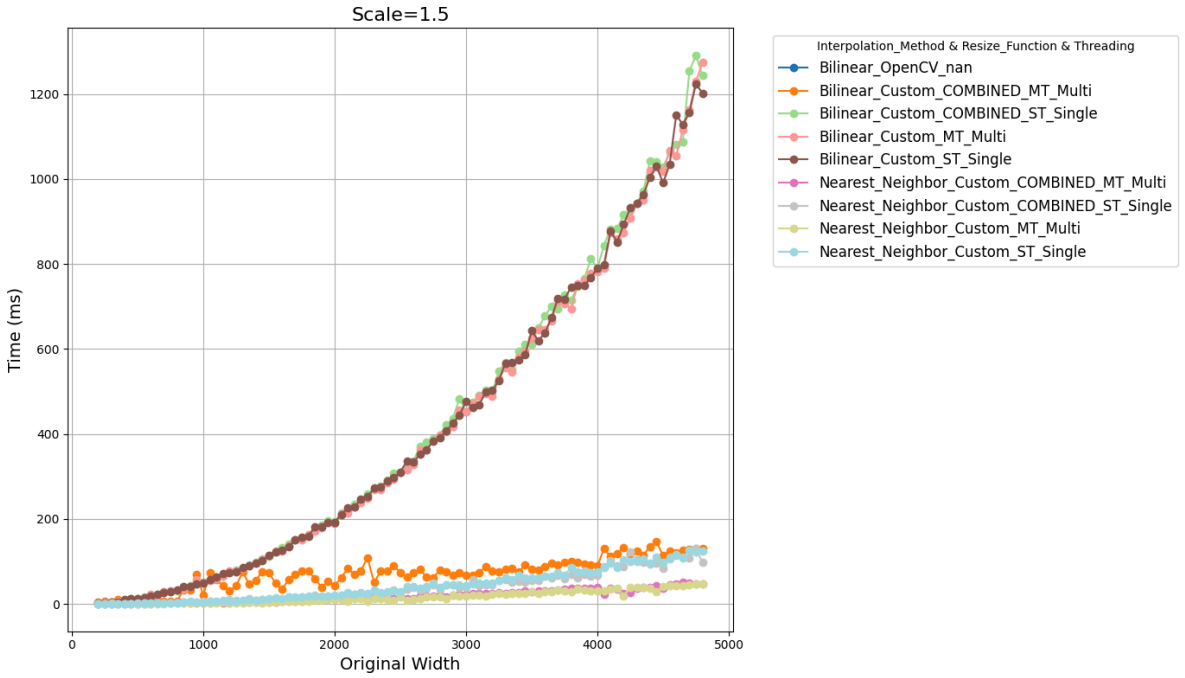
For Maximum Speed: Nearest Neighbor methods, particularly **AVX_MT_Multi**, provide the fastest performance due to their simplicity and optimized instructions.

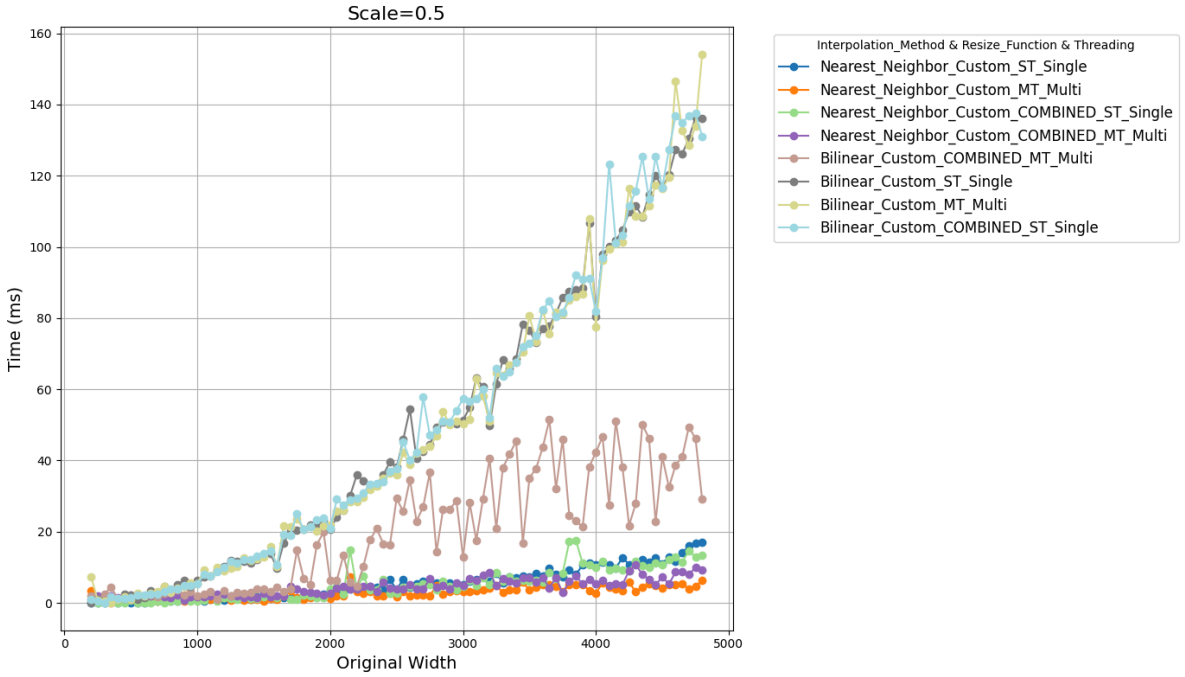
For Quality and Speed Balance: Bilinear methods with AVX and multithreading (e.g., **Bilinear_Custom_AVX_MT_Multi**) offer a good trade-off, delivering better image quality while keeping processing times reasonable.

OpenCV as a Baseline: OpenCV remains a reliable choice for interpolation, offering strong performance without requiring extensive custom optimization.

---

4. Performance and Review





Program Performance

- Our work has achieved **strong performance results** with the custom **multi-threaded implementations of Nearest Neighbor and Bilinear Interpolation**. According to this chart, Especially in Muti-Threaded case, our function only take **time less than 100ms**.
- Thanks to multi-threading and AVX2 optimization, our function's execution time does not exhibit exponential growth like single-threaded implementations as the **target image size increases**. Instead, it grows slowly in a linear manner. We have also provided two interpolation methods: **bilinear interpolation** (which offers better image quality) and **nearest neighbor interpolation** (which is faster).

Our weakness

- **Lacks optimizations**: The custom `resize` implementation is likely to be slower due to the absence of advanced optimizations like multi-threading or SIMD instructions.
- **No hardware acceleration**: The current solution does not leverage GPU acceleration or other hardware-specific features, leading to potentially higher processing times for large images.
- **Scalability issues**: As the image size increases, the custom method may face performance bottlenecks, especially when scaling up or down by large factors.
- **Limited interpolation methods**: The custom implementation might support only basic interpolation techniques, such as linear or nearest-neighbor interpolation, limiting its flexibility.
- **Basic performance**: More advanced algorithms (like bicubic or Lanczos) are not present in the custom function, which may lead to lower-quality resized images, especially when upscaling or downscaling by significant amounts.
- **Platform-dependent**: The custom function might not be as robust or portable across different platforms, and it could require additional adjustments or optimizations for different operating systems.
- **Potential for bugs**: As the custom solution is not as widely tested as OpenCV, it might be more prone to bugs or inconsistencies when used in diverse environments.
-

Optimize for Performance

- **Multi-threading**: Implementing multi-threading can significantly improve the performance of the resizing operation, particularly when processing large images.
- **SIMD Support**: Adding SIMD (Single Instruction, Multiple Data) instructions, such as AVX2 or SSE, would optimize the resizing process by processing multiple pixels in parallel, resulting in faster execution.
- **GPU Acceleration**: Leveraging GPU libraries like CUDA or OpenCL could drastically speed up the resizing operation, especially when dealing with large images or videos.
- To ensure broader compatibility, we can refactor the code to make it more cross-platform. This may involve using platform-agnostic libraries or adding conditional code to handle different environments (e.g., Windows, Linux, macOS).
- Improving how edge cases (e.g., very small images, non-square aspect ratios) are handled would enhance the robustness of the custom function.
- Ensuring that the resized images maintain good visual quality without unexpected artifacts (such as pixelation or blurring) in edge regions would improve the overall user experience.

5. Conclusion

This project successfully developed a custom image resize function inspired by OpenCV's native `resize` function. By implementing both nearest neighbor and bilinear interpolation methods and enhancing them with multi-threading and SIMD optimizations, the custom resize function achieves high performance and accuracy. The support for multiple data types and multi-channel images broadens its applicability across various computer vision tasks. Comparative analysis demonstrates that while the custom implementation performs admirably, OpenCV's native functions still hold advantages in terms of extensive optimizations and broader feature support. Nonetheless, the custom resize function serves as an excellent educational tool for understanding image processing fundamentals and optimization techniques.

References

- **OpenCV Documentation:** https://docs.opencv.org/
- **Bilinear Interpolation:** https://en.wikipedia.org/wiki/Bilinear_interpolation
- **SIMD Optimization:** https://en.wikipedia.org/wiki/SIMD
- OpenCV图像缩放resize各种插值方式的比较实现 / 张生荣

Explanation:

1. **Loading the Image:**
   The input image is loaded using OpenCV's `imread` function. Ensure that the image path is correct and that the image is successfully loaded.
2. **Defining Output Size:**
   Specify the desired dimensions for the resized image. In this example, the output size is set to 800x600 pixels.
3. **Choosing Interpolation Method:**
   Select between `NEAREST_NEIGHBOR` and `BILINEAR` interpolation methods based on the quality and performance requirements.
4. **Performing Resize Operation:**
   Call the `resize_custom` function, passing in the input image, an output image container,