SIMD Image Scaling Assignment Report

Khizra Hanif

University of Victoria

Introduction:

Creating, implementing, and assessing a SIMD algorithm to reduce the size of an image by 16 times was my aim for this assignment. I've used both SIMD and scalar solutions in my implementations. Based on multiple test cases, I have verified that my code works as expected for both scalar and SIMD input images. However, performance measurement was not successful because of the small amount of data input. Using the <chrono> library, I was able to compare how well both algorithms performed. To assess how well each strategy performed, I compared its execution times.

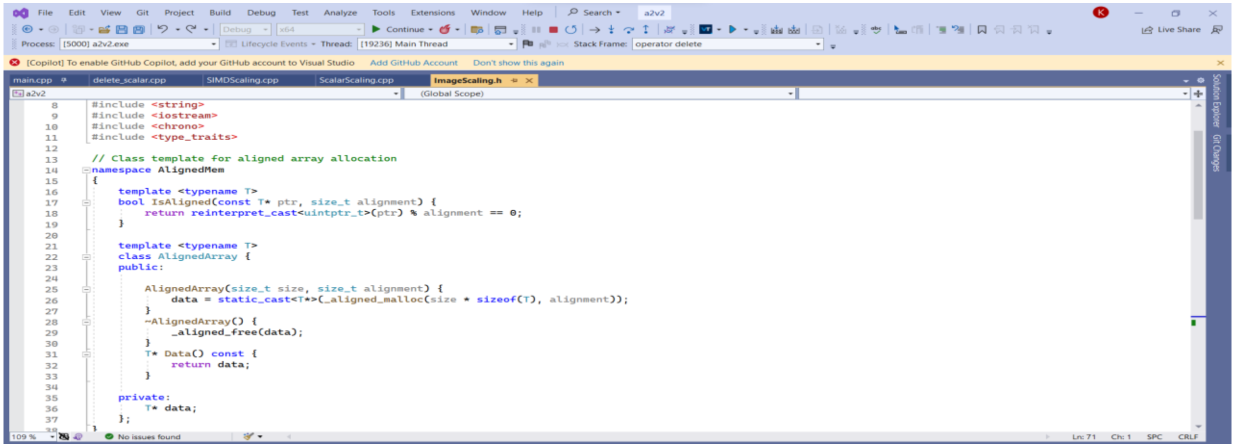
Data Structures:

The same data structures that were used in the Ch02_07 listing have been used by me. I also tried to duplicate the methodology employed in the reference materials. My code structure is the same as the one shown in Ch02_07. Reference material was adhered to by integrating functions like CheckArgs() for argument validation and InitArray() for array initialization. The code includes a class template AlignedMem for aligned array allocation, ensuring proper memory alignment for SIMD operations.Similarly, to fill an array with random values, the MT namespace offers a template function called FillArray.

To reduce the size of an image, two main functions are used: scalar scaling and SIMD scaling, which use scalar and SIMD (AVX2) solutions, respectively. Proper argument validation and alignment checks are provided by the code.

Code explanation:

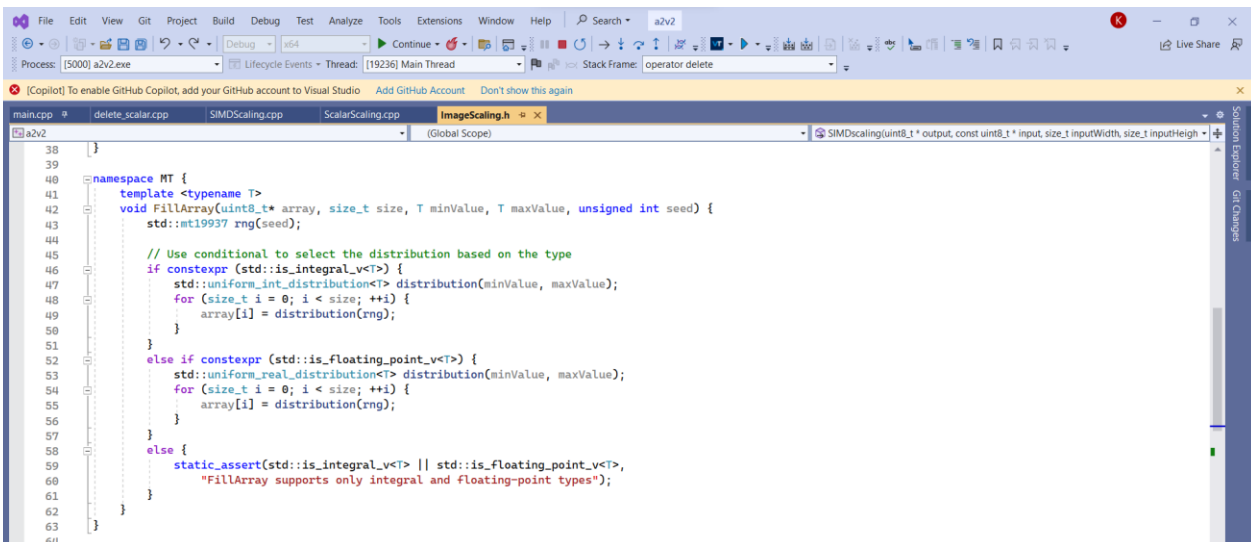
The namespace AlignedMem contains a class template AlignedArray for allocating aligned memory which is use for optimizing SIMD operations, as these operations often require data to be aligned to specific boundaries. The alignment is specified during array creation, ensuring that memory is allocated with the necessary alignment for efficient SIMD processing.



```cpp
#include <string>
#include <iostream>
#include <chrono>
#include <type_traits>

// Class template for aligned array allocation
namespace AlignedMem
{
    template <typename T>
    bool IsAligned(const T* ptr, size_t alignment) {
        return reinterpret_cast<uintptr_t>(ptr) % alignment == 0;
    }

    template <typename T>
    class AlignedArray {
    public:

        AlignedArray(size_t size, size_t alignment) {
            data = static_cast<T*>(_aligned_malloc(size * sizeof(T), alignment));
        }
        ~AlignedArray() {
            _aligned_free(data);
        }
        T* Data() const {
            return data;
        }

    private:
        T* data;
    };
}
```

In above code For allocating and deallocating aligned arrays, AlignedArray class template is used. Memory operations are carried out by _aligned_free and _aligned_malloc.

The MT namespace provides a templated function, FillArray, responsible for populating an array with random values. Instead of using one data input this function can create diverse and randomized input data for performance experiments. Mersenne Twister random number generator (std::mt19937) used to generate unpredictable numbers.



```cpp
}

namespace MT {
    template <typename T>
    void FillArray(uint8_t* array, size_t size, T minValue, T maxValue, unsigned int seed) {
        std::mt19937 rng(seed);

        // Use conditional to select the distribution based on the type
        if constexpr (std::is_integral_v<T>) {
            std::uniform_int_distribution<T> distribution(minValue, maxValue);
            for (size_t i = 0; i < size; ++i) {
                array[i] = distribution(rng);
            }
        }
        else if constexpr (std::is_floating_point_v<T>) {
            std::uniform_real_distribution<T> distribution(minValue, maxValue);
            for (size_t i = 0; i < size; ++i) {
                array[i] = distribution(rng);
            }
        }
        else {
            static_assert(std::is_integral_v<T> || std::is_floating_point_v<T>,
                "FillArray supports only integral and floating-point types");
        }
    }
}
```

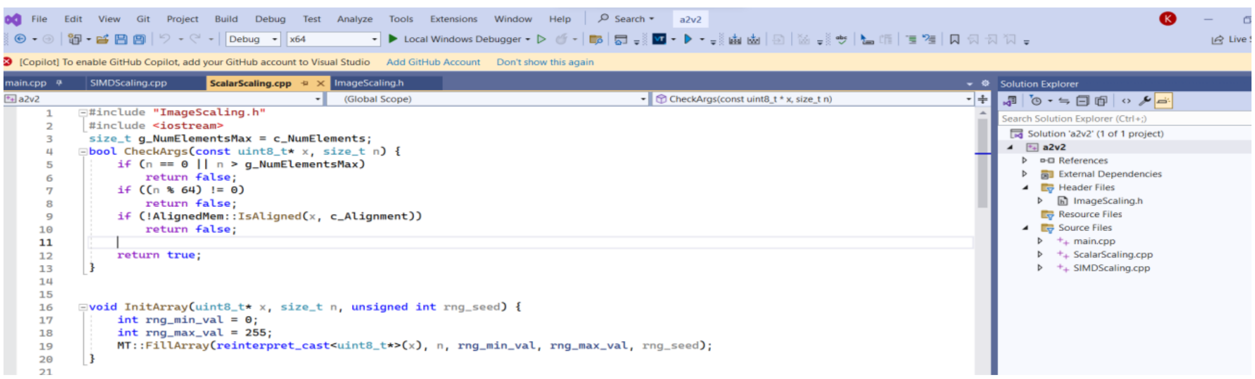
These are the functions below in the code which i have defined in Header file for scalar and SIMD operations. Scalarscaling function calculates the mean of each 4x4 pixel block and assigns the result to the corresponding pixel in the output array. Meanwhile Image scaling is accomplished by the SIMDscaling function using SIMD instructions (AVX2). Performance is enhanced by its effective simultaneous processing of multiple pixels using Intel Intrinsics.

```cpp
bool Scalarscaling(uint8_t* output, const uint8_t* input, size_t inputWidth, size_t inputHeight);
bool SIMDscaling(uint8_t* output, const uint8_t* input, size_t inputWidth, size_t inputHeight);
void InitArray(uint8_t* x, size_t n, unsigned int seed);
bool CheckArgs(const uint8_t* x, size_t n);
void PrintImage(const std::vector<uint8_t>& image, size_t width, size_t height, const std::string& title);
```

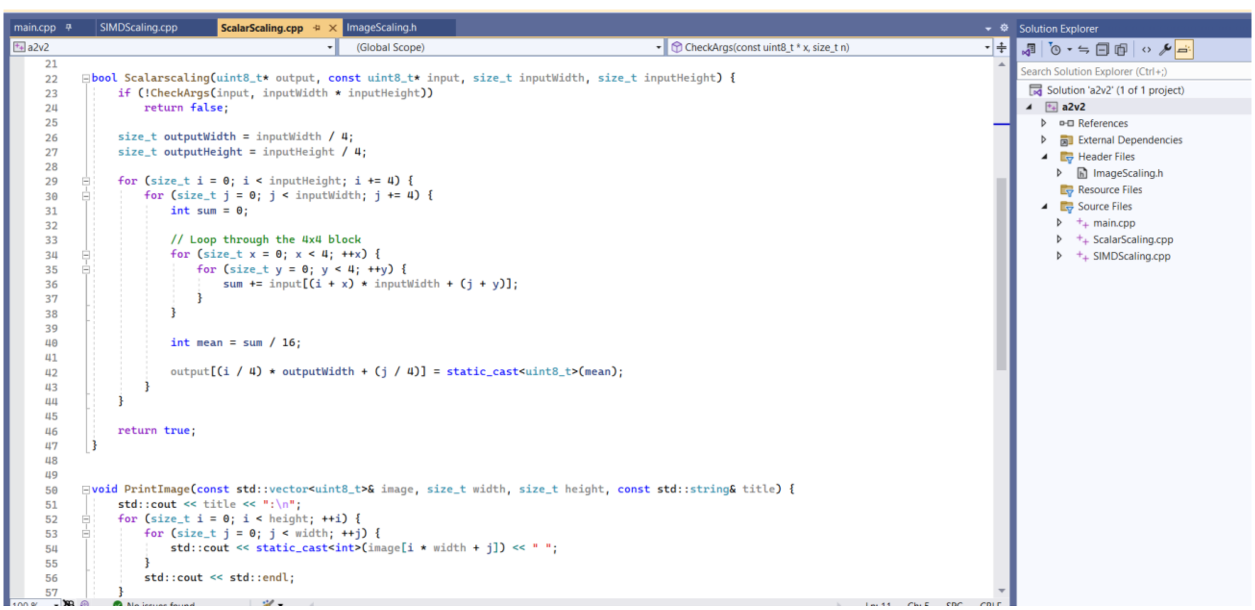
The CheckArgs() function checks the validity of the function arguments before performing image scaling. It verifies the following conditions:

- n (number of elements) is not zero and does not exceed the maximum allowed (g_NumElementsMax).
- n is a multiple of 64, which aligns with SIMD processing requirements.

The InitArray function initializes an array (x) with random values based on the provided seed (rng_seed). It uses the FillArray function from the MT namespace, which fills the array with random values within the specified range.

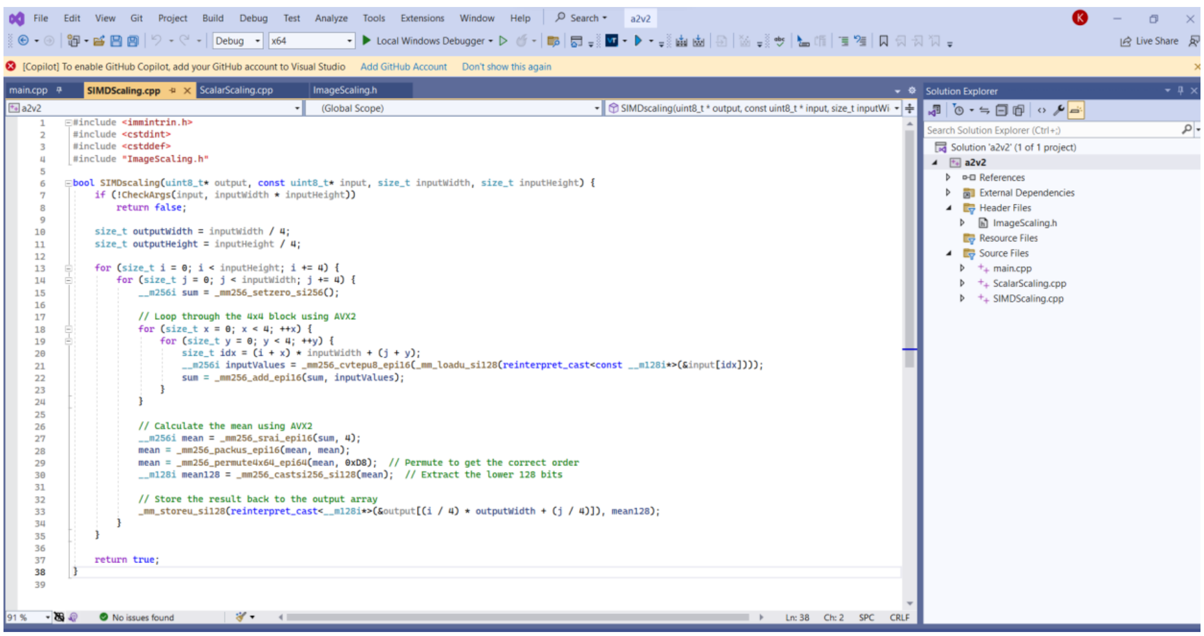


ScalarScaling function:



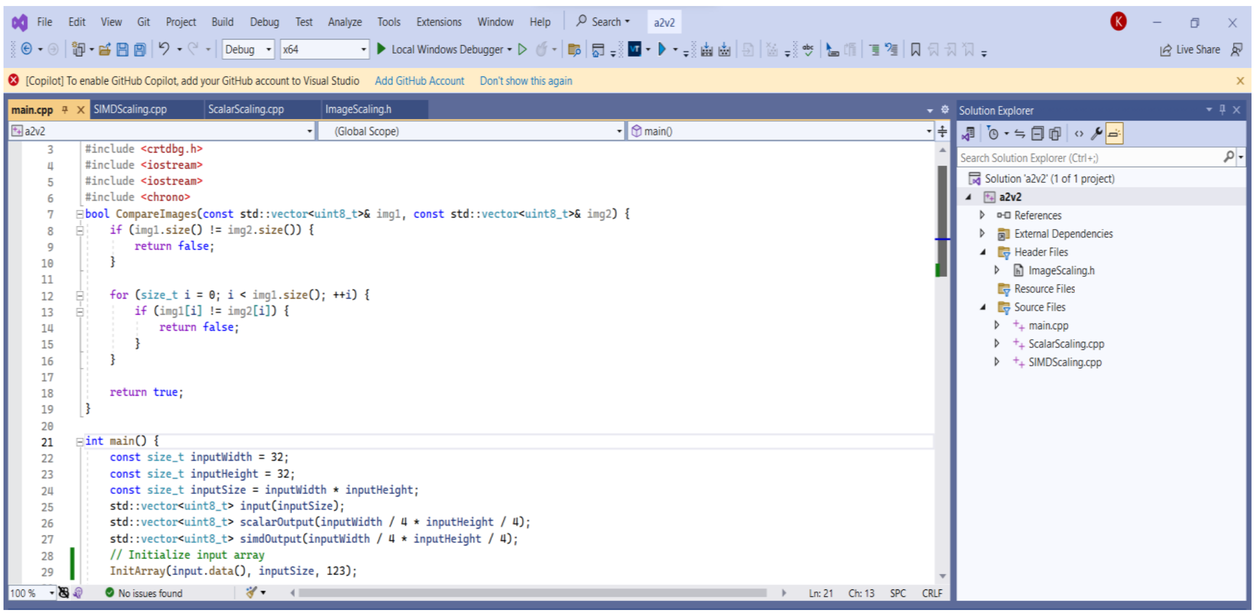
I have used the Scalarscaling function to reduce the dimensions by a factor of 4 using a mean calculation in 4x4 blocks. This function implements a basic image scaling operation. The function uses the CheckArgs function to verify the input arguments before starting the scaling process. This involves verifying that the array is aligned, has a non-zero size, and follows certain size restrictions. After that, the dimensions of the output array are calculated based on the input dimensions, effectively scaling down the image by a factor of 4. Next, a nested loop that iterates through the input array in 4x4 blocks is entered by the function. Rows are denoted by the outer loop (i), and columns by the inner loop (j). Within each 4x4 block, another pair of nested loops (x and y) iterates through each element, accumulating their values into the sum variable. Another set of nested loops (x and y) iterates through each element in each 4x4 block, adding the values of each iteration to the sum variable. The corresponding position in the output array is assigned to the computed mean value. The index computation guarantees correct placement in the output array that has been shrunk.

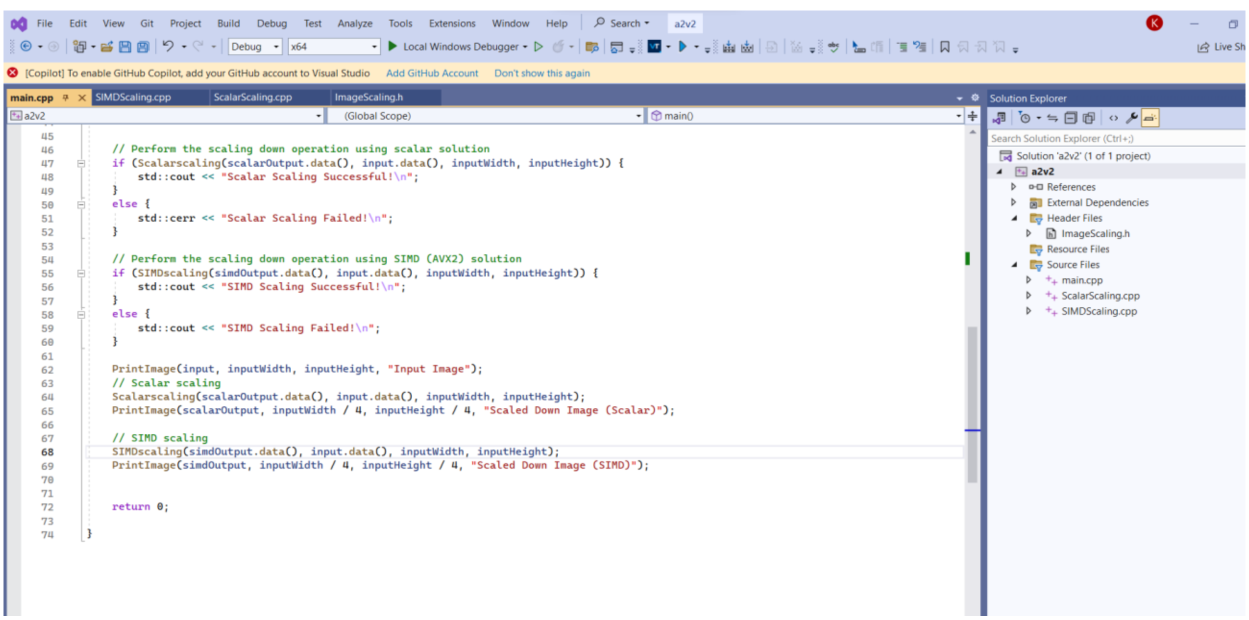
SIMDScaling Function:



The first step of checking argument validation is same as previous Scalar function then the dimensions of the output array are calculated based on the input dimensions, effectively scaling down the image by a factor of 4. Same as previous scalar function The SIMD function enters a nested loop that iterates through the input array in 4x4 blocks. The outer loop (i) represents the rows, and the inner loop (j) represents the columns. Each 4x4 block contains 8-bit unsigned integer values that are loaded using SIMD instructions, expanded to 16 bits. Specifically, AVX2 (Advanced Vector Extensions 2) SIMD (Single Instruction, Multiple Data) instructions are used by the SIMDscaling function in my code to increase the efficiency of image scaling operations. The function uses AVX2 instructions to execute parallelized operations on 256-bit vectors within the main processing loop. Each 4x4 block's pixel values are added together using the function _mm256_add_epi16, which accumulates 16-bit integer values. After the accumulated values are right-shifted, _mm256_srai_epi16 is used to calculate the mean. Other AVX2 instructions, such as _mm256_packus_epi16 and _mm256_permute4x64_epi64, are then used to complete the mean calculation and guarantee proper ordering. _mm_storeu_si128 is then used to store the outcome back in the output array. In general, these SIMD instructions allow for the processing of several data elements at once, enhancing the efficiency of image scaling operations compared to a scalar approach.

Main function:





The purpose of the above shared code is to assess and contrast the image scaling performance of scalar and SIMD (AVX2) implementations. Utilising custom functions from the "ImageScaling.h" header, the code first applies Scalarscaling for scalar image scaling, CheckArgs for function argument validation, InitArray for random array initialization, and SIMDscaling with AVX2 instructions. The main function sets up an input image, calculates and shows the times it takes to execute scalar and SIMD scaling operations, and prints the original, scalar-scaled, and SIMD-scaled images to create visual comparisons. The is used as a standard to evaluate how well SIMD instructions perform image scaling tasks in comparison to a traditional scalar method.

Performance Evaluation:

I have measured the execution time in order to perform performance analysis. The time taken by each scaling operation is measured using the std::chrono library. Specifically, high-resolution time points are recorded before and after the image scaling functions are executed using the std::chrono::high_resolution_clock.

```cpp

// Timing for Scalar Scaling
auto startScalar = std::chrono::high_resolution_clock::now();
Scalarscaling(scalarOutput.data(), input.data(), inputWidth, inputHeight);
auto endScalar = std::chrono::high_resolution_clock::now();
auto durationScalar = std::chrono::duration_cast<std::chrono::microseconds>(endScalar - startScalar).count();
std::cout << "Scalar Scaling Execution Time: " << durationScalar << " microseconds\n";

// Timing for SIMD Scaling
auto startSIMD = std::chrono::high_resolution_clock::now();
SIMDscaling(simdOutput.data(), input.data(), inputWidth, inputHeight);
auto endSIMD = std::chrono::high_resolution_clock::now();
auto durationSIMD = std::chrono::duration_cast<std::chrono::microseconds>(endSIMD - startSIMD).count();
std::cout << "SIMD Scaling Execution Time: " << durationSIMD << " microseconds\n";

```

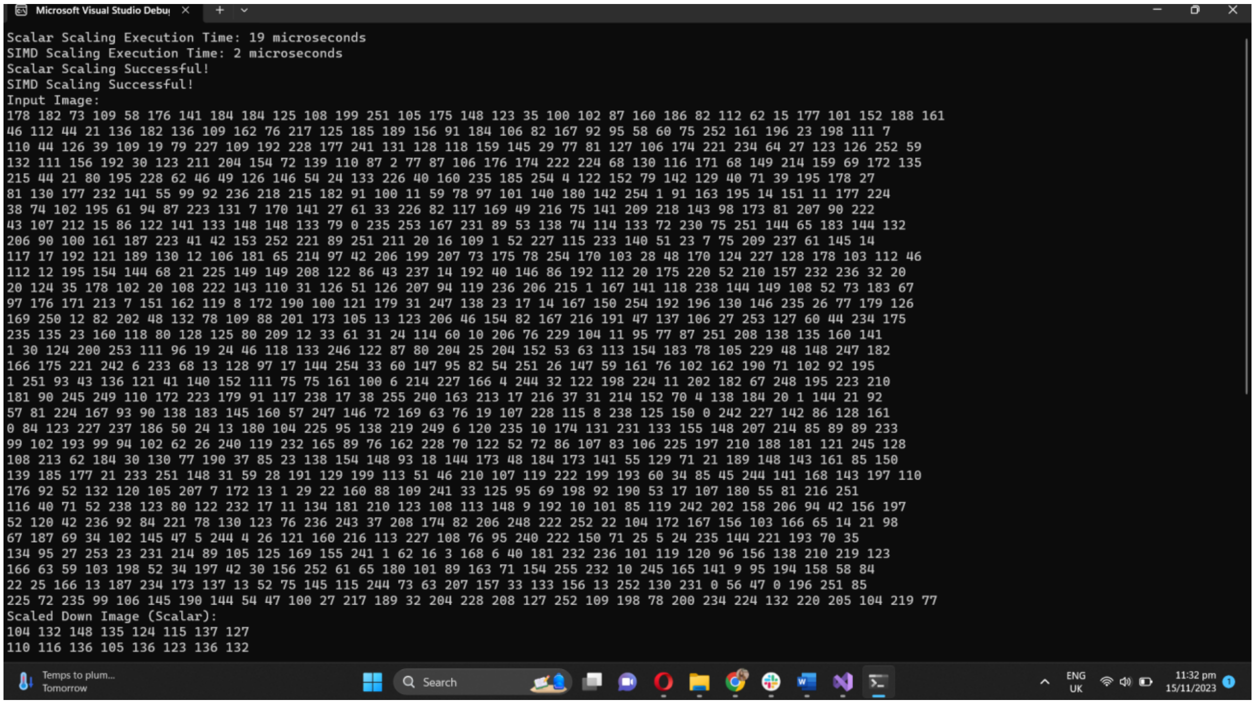
The execution time is then computed as the interval between these time points. A numerical evaluation of each implementation's effectiveness is provided to the console by printing the results, which are expressed in microseconds. Success messages are indicating if the scaling operations went smoothly. A qualitative comparison is also provided by presenting visual representations of the input image and the scaled-down images (both scalar and SIMD) that are produced. Notably, the execution time of the SIMD implementation is shorter than that of the scalar implementation, indicating that it performs better. For the purpose of optimising code and choosing appropriate algorithms and implementations, this performance analysis is essential.

The reported execution times are essential metrics for assessing the efficiency of each implementation. In one of the results:

- **Scalar Scaling Execution Time: 19 microseconds**
- **SIMD Scaling Execution Time: 2 microseconds**

These numbers show that the SIMD implementation performs noticeably better than the scalar one. The faster scaling operation completion time of the SIMD version highlights the

improved parallel processing capabilities of SIMD instructions.



Variability problem in Execution Times:

Performance evaluation of my code reveals a certain degree of variability in execution times upon multiple runs. In one case, the SIMD implementation outperforms the scalar counterpart, resulting in shorter execution times and in other scalar is performing better that SIMD.

However, it is important to note that the execution times are not strictly deterministic. Various factors, such as system load, background processes, and resource availability, can introduce variability in the performance measurements.