

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.

```

8 #include <string>
9 #include <iostream>
10 #include <chrono>
11 #include <type_traits>
12
13 // Class template for aligned array allocation
14 namespace MT
15 {
16     template <typename T>
17     bool IsAligned(const T* ptr, size_t alignment) {
18         return reinterpret_cast<uintptr_t>(ptr) % alignment == 0;
19     }
20
21     template <typename T>
22     class AlignedArray {
23     public:
24         AlignedArray(size_t size, size_t alignment) {
25             data = static_cast<T*>(_aligned_malloc(size * sizeof(T), alignment));
26         }
27         ~AlignedArray() {
28             _aligned_free(data);
29         }
30         T* Data() const {
31             return data;
32         }
33     private:
34         T* data;
35     };
36 }
37

```

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.

```

38 }
39
40 namespace MT {
41     template <typename T>
42     void FillArray(uint8_t* array, size_t size, T minValue, T maxValue, unsigned int seed) {
43         std::mt19937 rng(seed);
44
45         // Use conditional to select the distribution based on the type
46         if constexpr (std::is_integral_v<T>) {
47             std::uniform_int_distribution<T> distribution(minValue, maxValue);
48             for (size_t i = 0; i < size; ++i) {
49                 array[i] = distribution(rng);
50             }
51         }
52         else if constexpr (std::is_floating_point_v<T>) {
53             std::uniform_real_distribution<T> distribution(minValue, maxValue);
54             for (size_t i = 0; i < size; ++i) {
55                 array[i] = distribution(rng);
56             }
57         }
58         else {
59             static_assert(std::is_integral_v<T> || std::is_floating_point_v<T>,
60                 "FillArray supports only integral and floating-point types");
61         }
62     }
63 }
64

```

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.

```

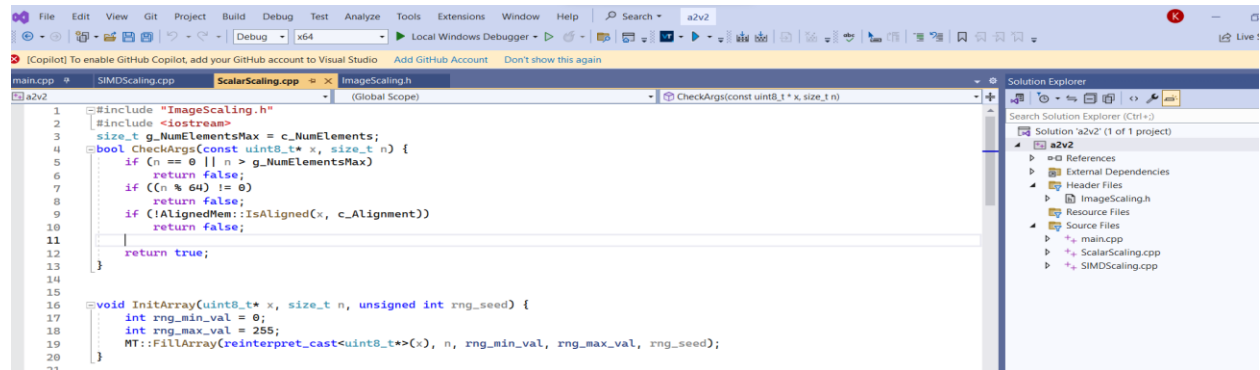
bool Scalarscaling(uint8_t* output, const uint8_t* input, size_t inputWidth, size_t inputHeight);
bool SIMDscaling(uint8_t* output, const uint8_t* input, size_t inputWidth, size_t inputHeight);
void InitArray(uint8_t* x, size_t n, unsigned int seed);
bool CheckArgs(const uint8_t* x, size_t n);
void PrintImage(const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

```

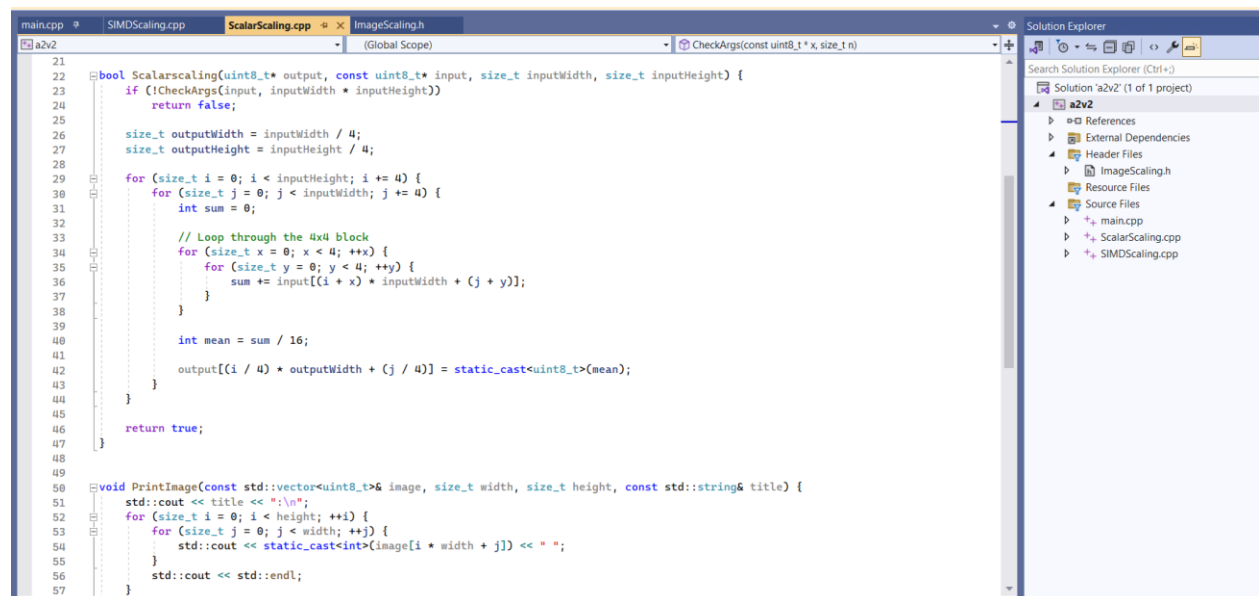
The `CheckArgs()` function checks the validity of the function arguments before performing image scaling. It verifies the following conditions:

- `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.

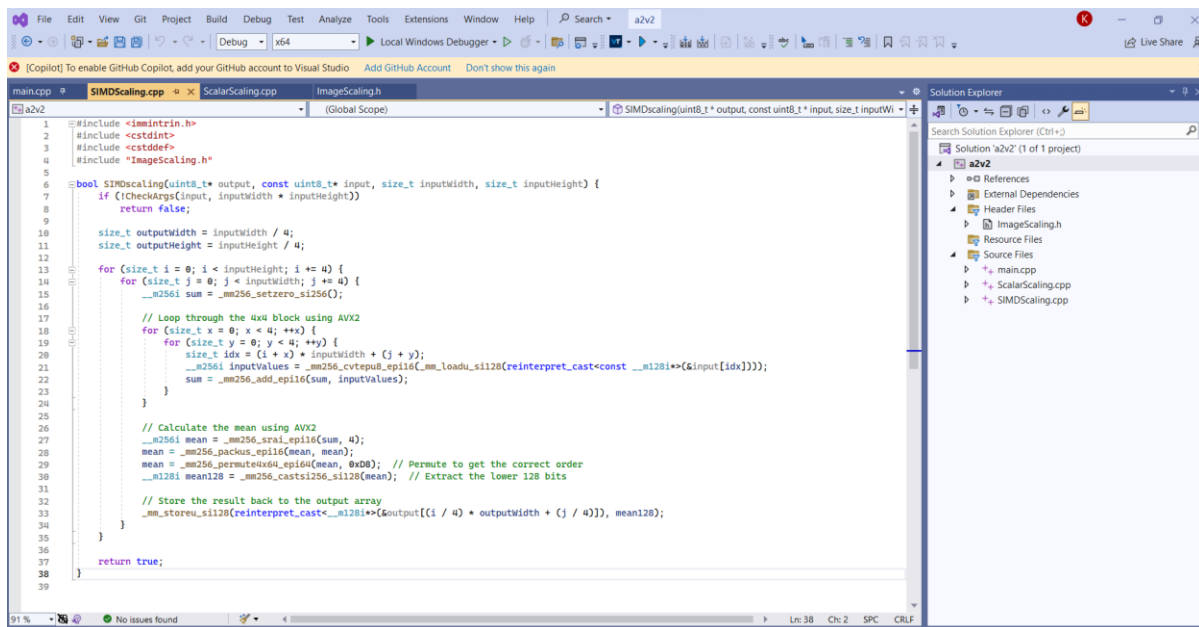


ScalarScaling function:



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:

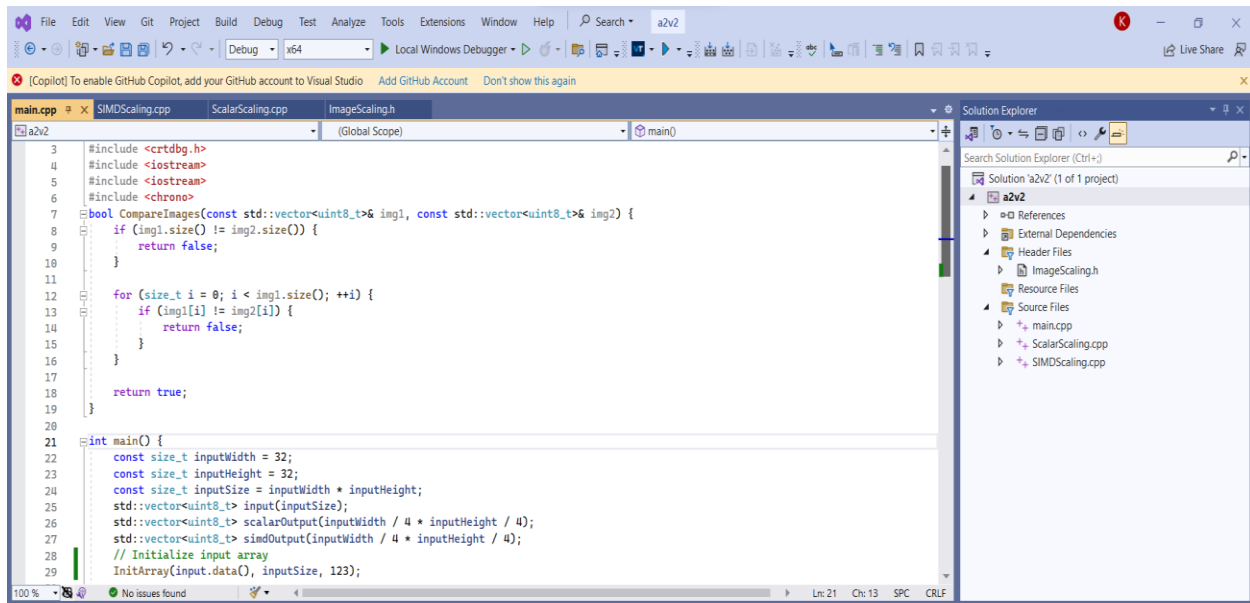


```
1 #include <immintrin.h>
2 #include <stdint.h>
3 #include <cstring>
4 #include "ImageScaling.h"
5
6 bool SIMDScaling(uint8_t* output, const uint8_t* input, size_t inputWidth, size_t inputHeight) {
7     if (!CheckArgs(input, inputWidth * inputHeight))
8         return false;
9
10    size_t outputWidth = inputWidth / 4;
11    size_t outputHeight = inputHeight / 4;
12
13    for (size_t i = 0; i < inputHeight; i += 4) {
14        for (size_t j = 0; j < inputWidth; j += 4) {
15            __m256i sum = _mm256_setzero_si256();
16
17            // Loop through the 4x4 block using AVX2
18            for (size_t x = 0; x < 4; ++x) {
19                for (size_t y = 0; y < 4; ++y) {
20                    size_t idx = (i + x) * inputWidth + (j + y);
21                    __m256i inputValues = _mm256_cvtepu8_epi16(_mm_loadu_si128(reinterpret_cast<const __m128i*>(&input[idx])));
22                    sum = _mm256_add_epi16(sum, inputValues);
23                }
24            }
25
26            // Calculate the mean using AVX2
27            __m256i mean = _mm256_srli_epi16(sum, 4);
28            mean = _mm256_packus_epi16(mean, mean);
29            mean = _mm256_permute4x64_epi64(mean, 0x008); // Permute to get the correct order
30            __m128i mean128 = _mm256_castsi256_si128(mean); // Extract the lower 128 bits
31
32            // Store the result back to the output array
33            _mm_storeu_si128(reinterpret_cast<__m128i*>(&output[(i / 4) * outputWidth + (j / 4)]), mean128);
34        }
35    }
36
37    return true;
38 }
39
```

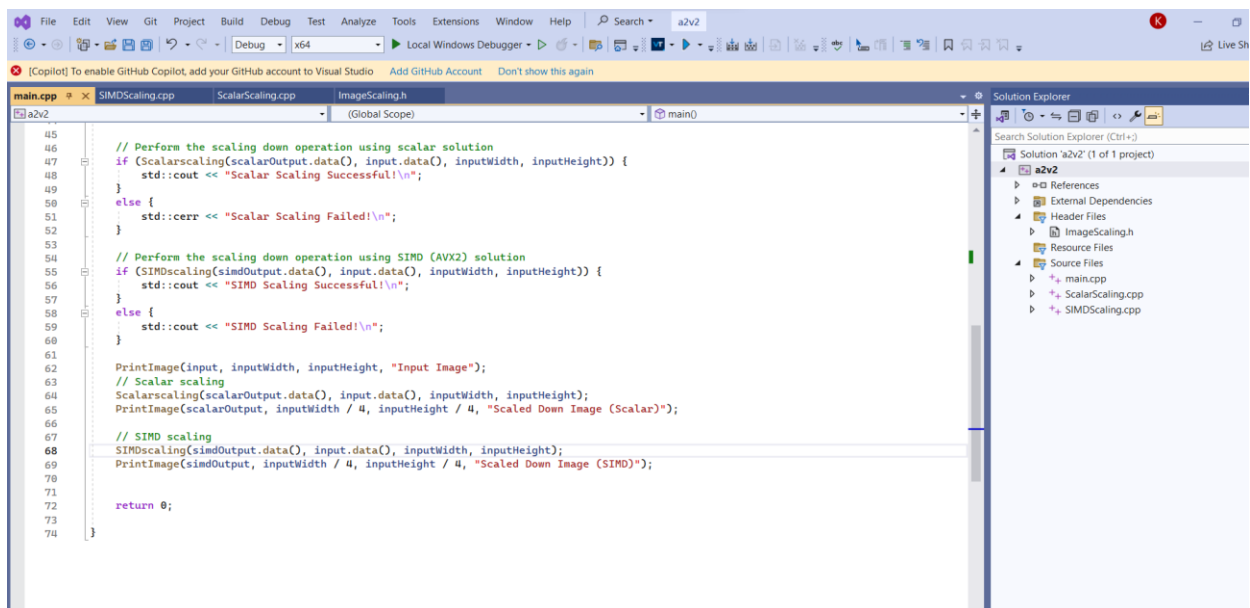
The first step of checking argument validation is same as previous `Scalar` function then the dimensions of the output array are calculated based on the input dimensions, effectively scaling down the image by a factor of 4. Same as previous scalar function The `SIMD` function enters a nested loop that 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:



```
3 #include <cstdlib>
4 #include <iostream>
5 #include <iostream>
6 #include <chrono>
7
8 bool CompareImages(const std::vector<uint8_t>& img1, const std::vector<uint8_t>& img2) {
9     if (img1.size() != img2.size()) {
10         return false;
11     }
12     for (size_t i = 0; i < img1.size(); ++i) {
13         if (img1[i] != img2[i]) {
14             return false;
15         }
16     }
17     return true;
18 }
19
20 int main() {
21     const size_t inputWidth = 32;
22     const size_t inputHeight = 32;
23     const size_t inputSize = inputWidth * inputHeight;
24     std::vector<uint8_t> input(inputSize);
25     std::vector<uint8_t> scalarOutput(inputWidth / 4 * inputHeight / 4);
26     std::vector<uint8_t> simdOutput(inputWidth / 4 * inputHeight / 4);
27     // Initialize input array
28     InitArray(input.data(), inputSize, 123);
29 }
```

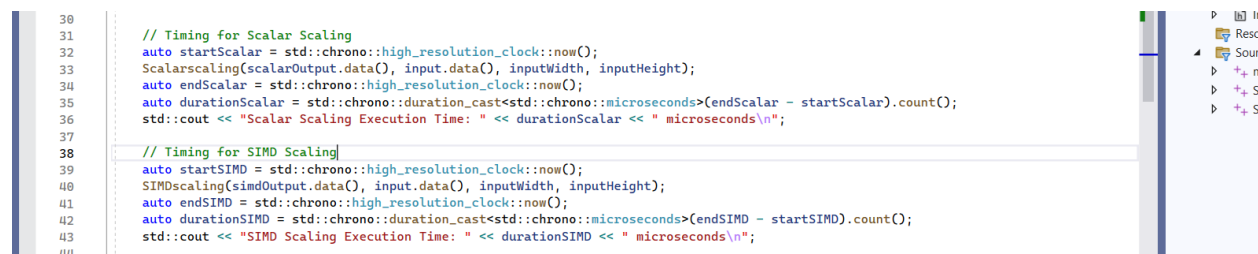


```
45 // Perform the scaling down operation using scalar solution
46 if (Scalarscaling(scalarOutput.data(), input.data(), inputWidth, inputHeight)) {
47     std::cout << "Scalar Scaling Successful!\n";
48 }
49 else {
50     std::cerr << "Scalar Scaling Failed!\n";
51 }
52
53 // Perform the scaling down operation using SIMD (AVX2) solution
54 if (SIMDscaling(simdOutput.data(), input.data(), inputWidth, inputHeight)) {
55     std::cout << "SIMD Scaling Successful!\n";
56 }
57 else {
58     std::cout << "SIMD Scaling Failed!\n";
59 }
60
61 PrintImage(input, inputWidth, inputHeight, "Input Image");
62 // Scalar scaling
63 Scalarscaling(scalarOutput.data(), input.data(), inputWidth, inputHeight);
64 PrintImage(scalarOutput, inputWidth / 4, inputHeight / 4, "Scaled Down Image (Scalar)");
65
66 // SIMD scaling
67 SIMDscaling(simdOutput.data(), input.data(), inputWidth, inputHeight);
68 PrintImage(simdOutput, inputWidth / 4, inputHeight / 4, "Scaled Down Image (SIMD)");
69
70 return 0;
71
72
73
74 }
```

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.

A screenshot of a code editor showing C++ code for image scaling performance evaluation. The code is divided into two sections: 'Timing for Scalar Scaling' and 'Timing for SIMD Scaling'. The scalar section uses std::chrono::high_resolution_clock to measure the time taken by Scalarscaling, resulting in a durationScalar of 19 microseconds. The SIMD section uses the same clock to measure the time taken by SIMDscaling, resulting in a durationSIMD of 2 microseconds. The code also includes function calls for CheckArgs, InitArray, and the scaling functions themselves. The output is printed to the console using std::cout.

```
30
31 // Timing for Scalar Scaling
32 auto startScalar = std::chrono::high_resolution_clock::now();
33 Scalarscaling(scalarOutput.data(), input.data(), inputWidth, inputHeight);
34 auto endScalar = std::chrono::high_resolution_clock::now();
35 auto durationScalar = std::chrono::duration_cast<std::chrono::microseconds>(endScalar - startScalar).count();
36 std::cout << "Scalar Scaling Execution Time: " << durationScalar << " microseconds\n";
37
38 // Timing for SIMD Scaling
39 auto startSIMD = std::chrono::high_resolution_clock::now();
40 SIMDscaling(simdOutput.data(), input.data(), inputWidth, inputHeight);
41 auto endSIMD = std::chrono::high_resolution_clock::now();
42 auto durationSIMD = std::chrono::duration_cast<std::chrono::microseconds>(endSIMD - startSIMD).count();
43 std::cout << "SIMD Scaling Execution Time: " << durationSIMD << " microseconds\n";
44
```

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.

```
Microsoft Visual Studio Debu...
Scalar Scaling Execution Time: 19 microseconds
SIMD Scaling Execution Time: 2 microseconds
Scalar Scaling Successful!
SIMD Scaling Successful!
Input Image:
178 182 73 109 58 176 141 184 184 125 108 199 251 105 175 148 123 35 100 102 87 160 186 82 112 62 15 177 101 152 188 161
46 112 44 21 136 182 136 109 162 76 217 125 185 189 156 91 184 186 82 167 92 95 58 60 75 252 161 196 23 198 111 7
110 44 126 39 109 19 79 227 109 192 228 177 241 131 128 118 159 145 29 77 81 127 106 174 221 234 64 27 123 126 252 59
132 111 156 192 30 123 211 204 154 72 139 110 87 2 77 87 106 176 174 222 224 68 130 116 171 68 149 214 159 69 172 135
215 44 21 80 195 228 62 46 49 126 146 54 24 133 226 40 160 235 185 254 4 122 152 79 142 129 40 71 39 195 178 27
81 130 177 232 141 55 99 92 236 218 215 182 91 100 11 59 78 97 101 140 180 142 254 1 91 163 195 14 151 11 177 224
38 74 182 195 61 94 87 223 131 7 170 141 27 61 33 226 82 117 169 49 216 75 141 209 218 143 98 173 81 207 90 222
43 107 212 15 86 122 141 133 148 148 133 79 0 235 253 167 231 89 53 138 74 114 133 72 230 75 251 144 65 183 144 132
206 90 100 161 187 223 41 42 153 252 221 89 251 211 20 16 109 1 52 227 115 233 140 51 23 7 75 209 237 61 145 14
117 17 192 121 189 130 12 106 181 65 214 97 42 206 199 207 73 175 78 254 170 103 28 48 170 124 227 128 178 103 112 46
112 12 195 154 144 68 21 225 149 149 208 122 86 43 237 14 192 40 146 86 192 112 20 175 220 52 210 157 232 236 32 20
20 124 35 178 182 20 108 222 143 110 31 126 51 126 207 94 119 236 206 215 1 167 141 118 238 144 149 108 52 73 183 67
97 176 171 213 7 151 162 119 8 172 190 100 121 179 31 247 138 23 17 14 167 150 254 192 196 130 146 235 26 77 179 126
169 250 12 82 202 48 132 78 169 88 201 173 185 13 123 206 46 154 82 167 216 191 47 137 106 27 253 127 60 44 234 175
235 135 23 160 118 80 128 125 80 209 12 33 61 31 24 114 60 10 206 76 229 104 11 95 77 87 251 208 138 135 160 141
1 30 124 200 253 111 96 19 24 46 118 133 246 122 87 80 204 25 204 152 53 63 113 154 183 78 105 229 48 148 247 182
166 175 221 242 6 233 68 13 128 97 17 144 254 33 60 147 95 82 54 251 26 147 59 161 76 102 162 190 71 102 92 195
1 251 93 43 136 121 41 140 152 111 75 75 161 100 6 214 227 166 4 244 32 122 198 224 11 202 182 67 248 195 223 210
181 90 245 249 110 172 223 179 91 117 238 17 38 255 240 163 213 17 216 37 31 214 152 70 4 138 184 20 1 144 21 92
57 81 224 167 93 90 138 183 145 160 57 247 146 72 169 63 76 19 107 228 115 8 238 125 150 0 242 227 142 86 128 161
0 84 123 227 237 186 50 24 13 180 104 225 95 138 219 249 6 120 235 10 174 131 231 133 155 148 207 214 85 89 89 233
99 102 193 99 94 102 62 26 240 119 232 165 89 76 162 228 70 122 52 72 86 107 83 106 225 197 210 188 181 121 245 128
188 213 62 184 30 130 77 190 37 85 23 138 154 148 93 18 144 173 48 184 173 141 55 129 71 21 189 148 143 161 85 150
139 185 177 21 233 251 108 31 59 28 191 129 199 113 51 46 210 107 119 222 199 103 60 34 85 45 244 141 168 143 197 110
176 92 52 132 120 105 207 7 172 13 1 29 22 160 88 109 241 33 125 95 69 190 92 100 53 17 107 180 55 81 216 251
116 40 71 52 238 123 80 122 232 17 11 134 181 210 123 108 113 148 9 192 10 101 85 119 242 202 158 206 94 42 156 197
52 120 42 236 92 84 221 78 130 123 76 236 243 37 208 174 82 206 248 222 252 22 104 172 167 156 103 166 65 14 21 98
67 187 69 34 102 145 47 5 244 4 26 121 160 216 113 227 108 76 95 240 222 150 71 25 5 24 235 144 221 193 70 35
134 95 27 253 23 231 214 89 105 125 169 155 241 1 62 16 3 168 6 40 181 232 236 101 119 120 96 156 138 210 219 123
166 63 59 103 198 52 34 197 42 30 156 252 61 65 180 101 89 163 71 154 255 232 10 245 165 141 9 95 194 158 58 84
22 25 166 13 187 234 173 137 13 52 75 145 115 244 73 63 207 157 33 133 156 13 252 130 231 0 56 47 0 196 251 85
225 72 235 99 106 145 190 144 54 47 100 27 217 189 32 204 228 208 127 252 109 198 78 200 234 224 132 220 205 104 219 77
Scaled Down Image (Scalar):
104 132 148 135 124 115 137 127
110 116 136 105 136 123 136 132
114 115 144 125 138 113 140 111
129 114 106 111 98 136 152 132
155 121 116 132 127 120 122 131
126 116 123 129 118 127 155 145
96 111 98 148 139 117 135 113
109 147 96 116 127 164 127 145
Scaled Down Image (SIMD):
104 132 148 135 124 115 137 127
110 116 136 105 136 123 136 132
114 115 144 125 138 113 140 111
129 114 106 111 98 136 152 132
155 121 116 132 127 120 122 131
126 116 123 129 118 127 155 145
96 111 98 148 139 117 135 113
109 147 96 116 127 164 127 145
C:\Users\khizr\source\repos\av2v2\x64\Release\av2v2.exe (process 6244) exited with code 0.
Press any key to close this window . . .]
```

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 than 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.

