In the name of God

Project 1 — Parallel Programming Course

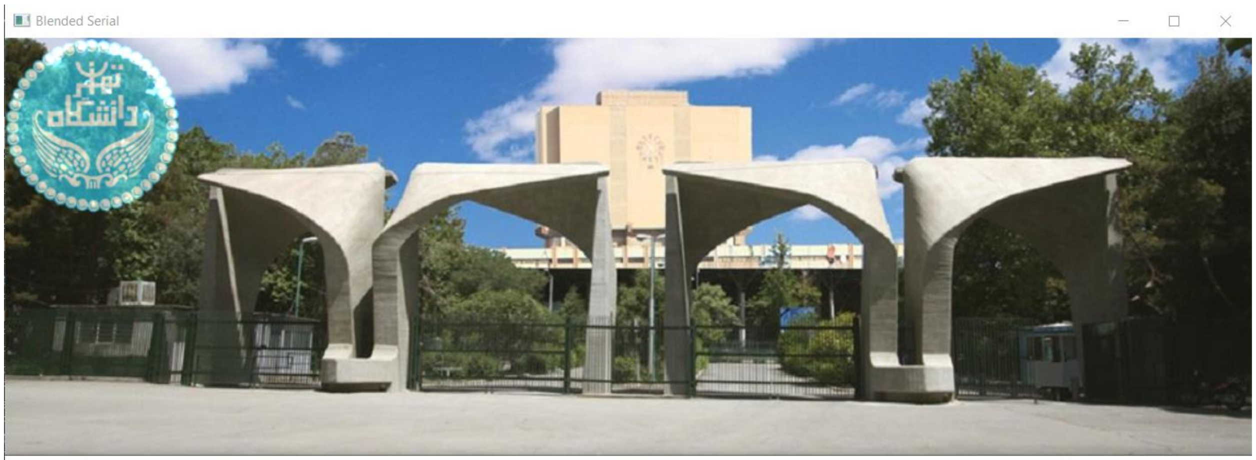
Instructor: **Dr. Safari**

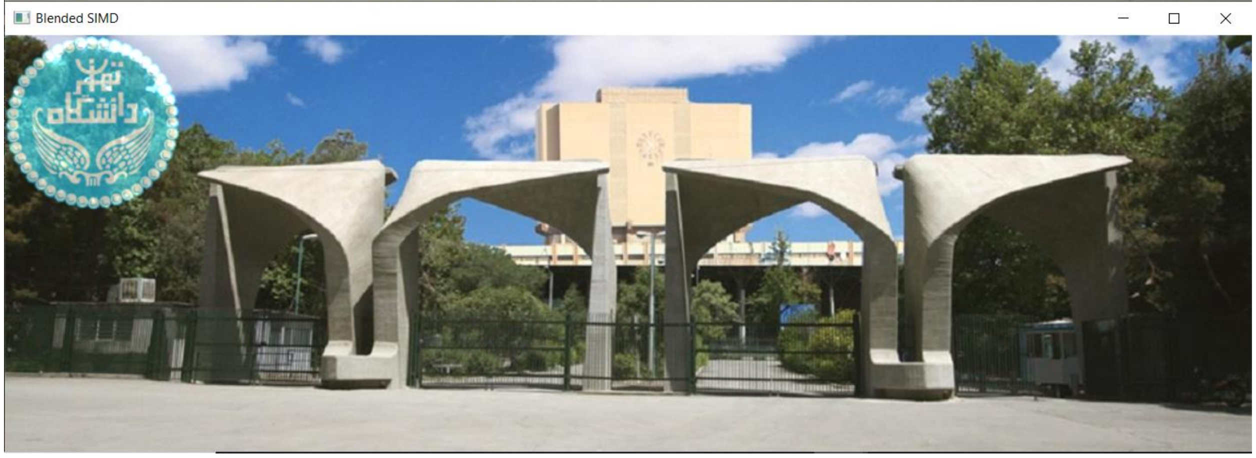
Students:

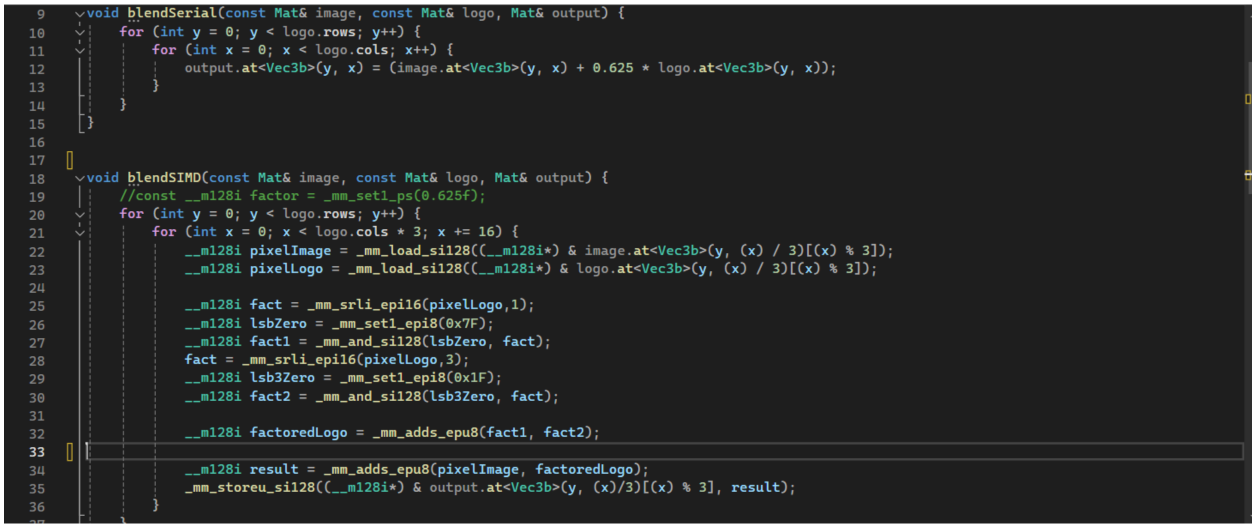
Mohamad Yahyapour

Mohammad Moien Joneidi Jaafari

Part 1





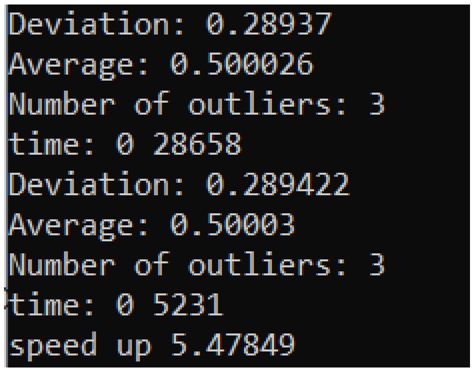


```
void blendSerial(const Mat& image, const Mat& logo, Mat& output) {
    for (int y = 0; y < logo.rows; y++) {
        for (int x = 0; x < logo.cols; x++) {
            output.at<Vec3b>(y, x) = (image.at<Vec3b>(y, x) + 0.625 * logo.at<Vec3b>(y, x));
        }
    }
}


void blendSIMD(const Mat& image, const Mat& logo, Mat& output) {
    //const __m128i factor = _mm_set1_ps(0.625f);
    for (int y = 0; y < logo.rows; y++) {
        for (int x = 0; x < logo.cols * 3; x += 16) {
            __m128i pixelImage = _mm_load_si128((__m128i*) & image.at<Vec3b>(y, (x) / 3)[(x) % 3]);
            __m128i pixelLogo = _mm_load_si128((__m128i*) & logo.at<Vec3b>(y, (x) / 3)[(x) % 3]);

            __m128i fact = _mm_srli_epi16(pixelLogo,1);
            __m128i lsbZero = _mm_set1_epi8(0x7F);
            __m128i fact1 = _mm_and_si128(lsbZero, fact);
            fact = _mm_srli_epi16(pixelLogo,3);
            __m128i lsb3Zero = _mm_set1_epi8(0x1F);
            __m128i fact2 = _mm_and_si128(lsb3Zero, fact);

            __m128i factoredLogo = _mm_adds_epu8(fact1, fact2);

            __m128i result = _mm_adds_epu8(pixelImage, factoredLogo);
            _mm_storeu_si128((__m128i*) & output.at<Vec3b>(y, (x)/3)[(x) % 3], result);
        }
```

In the serial section, one pixel of the logo is read, its **RGB values** are multiplied by **0.625**, and then added to the corresponding pixel of the background image. Finally, the result is written to the output.

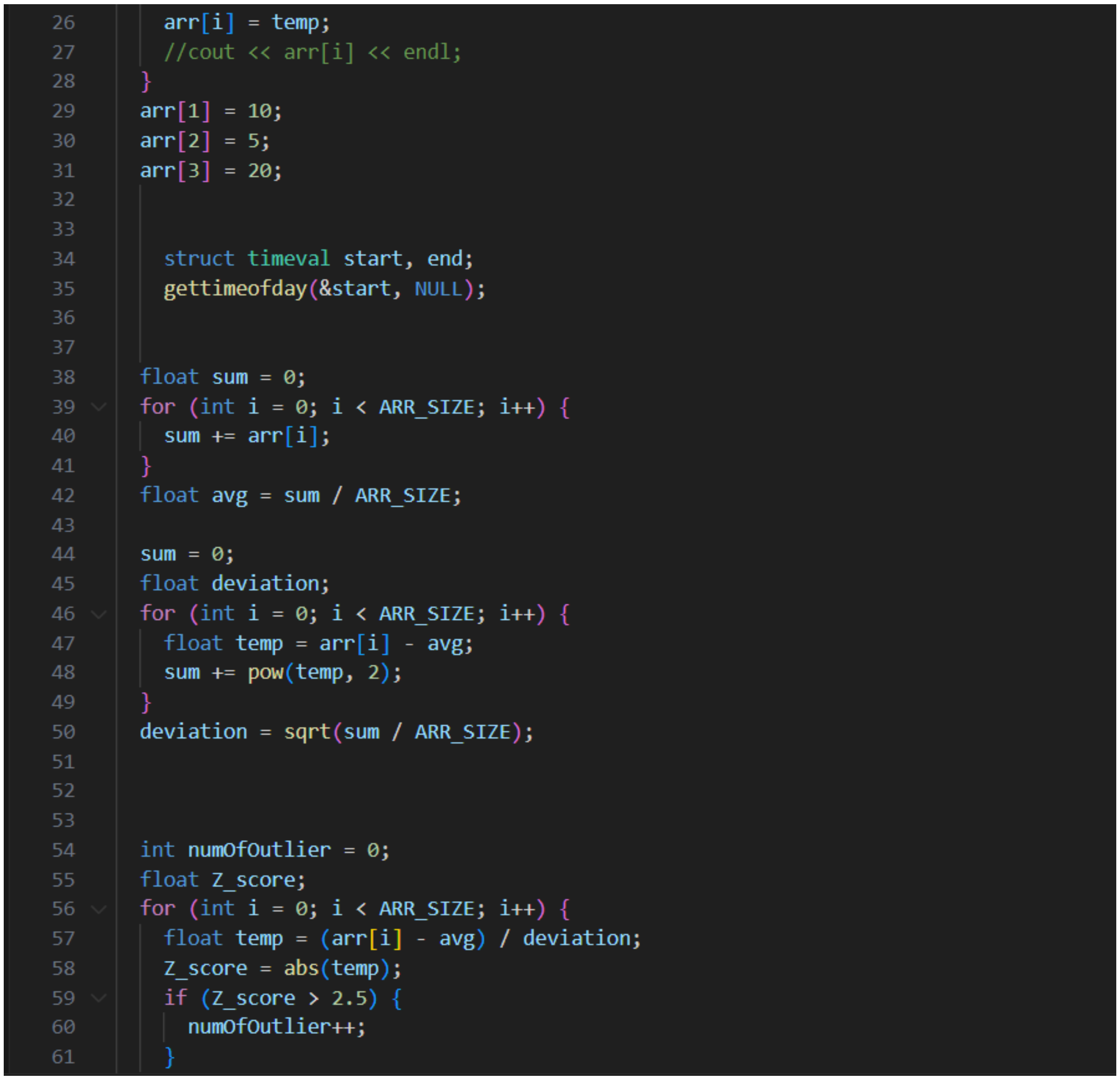
In the parallelized section, **16 bytes** from the pixel vector of the image are read, where every three bytes correspond to one pixel. This means that in a single __m128i, the values of **5 pixels** plus one extra RGB component can be processed. Since this variable is of type **int**, and we want to multiply it by a **floating-point coefficient**, one option would be converting the values to float. However, this would force us to load fewer elements into the register. Instead, the method we chose was to **construct the coefficient using bit shifts**. First, we shifted the logo values one bit to the right and stored them in a 128-bit variable. Then we shifted the values **three bits** to the right and stored them in another variable. Finally, we **added** these results together and then added the final value to the background image. The outcome matched the serial version. If we had converted to float, we would have needed to convert back to int at the end, which would introduce approximation. Here, using shifts also introduces a similar approximation. In this exercise, the **average speedup** is approximately **10 to 11×**.

Part 2



```
Deviation: 0.28937
Average: 0.500026
Number of outliers: 3
time: 0 28658
Deviation: 0.289422
Average: 0.50003
Number of outliers: 3
time: 0 5231
speed up 5.47849
```

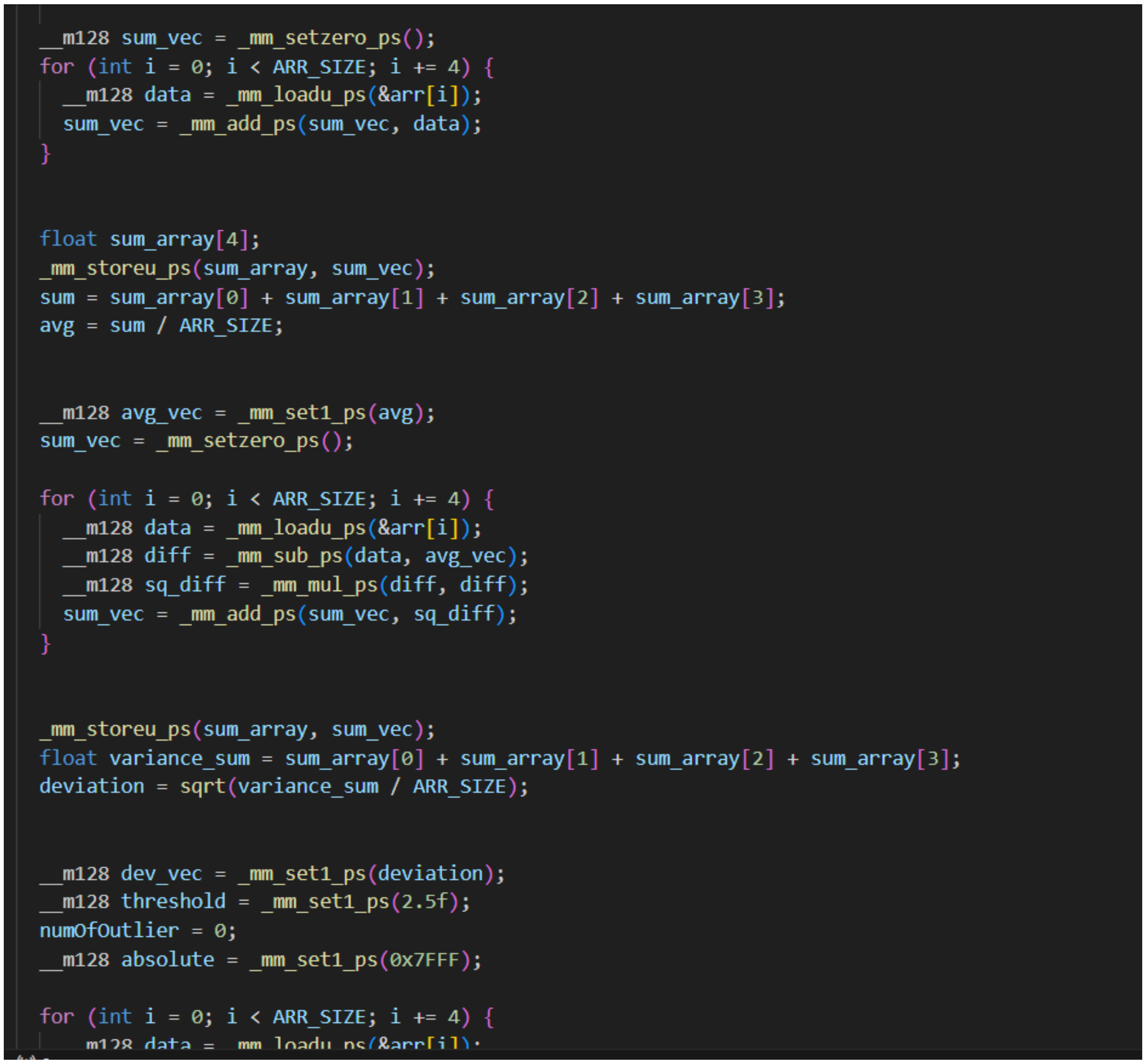
With a large number of tests performed, the **speedup value is approximately 5**.



```
    arr[i] = temp;
    //cout << arr[i] << endl;
  }
  arr[1] = 10;
  arr[2] = 5;
  arr[3] = 20;


    struct timeval start, end;
    gettimeofday(&start, NULL);


  float sum = 0;
  for (int i = 0; i < ARR_SIZE; i++) {
    sum += arr[i];
  }
  float avg = sum / ARR_SIZE;

  sum = 0;
  float deviation;
  for (int i = 0; i < ARR_SIZE; i++) {
    float temp = arr[i] - avg;
    sum += pow(temp, 2);
  }
  deviation = sqrt(sum / ARR_SIZE);



  int numOfOutlier = 0;
  float Z_score;
  for (int i = 0; i < ARR_SIZE; i++) {
    float temp = (arr[i] - avg) / deviation;
    Z_score = abs(temp);
    if (Z_score > 2.5) {
      numOfOutlier++;
    }
```

The image above shows the **serial code**. First, all values are summed and then divided by the total count to compute the **mean**. Next, the squared differences from the mean are summed and divided by the total count to compute the **variance**. Finally, the square root is taken to obtain the **standard deviation**.

Finally, the **z-score** is computed, and if it is greater than **2.5**, the value is counted as an outlier by incrementing **numOutlier**.

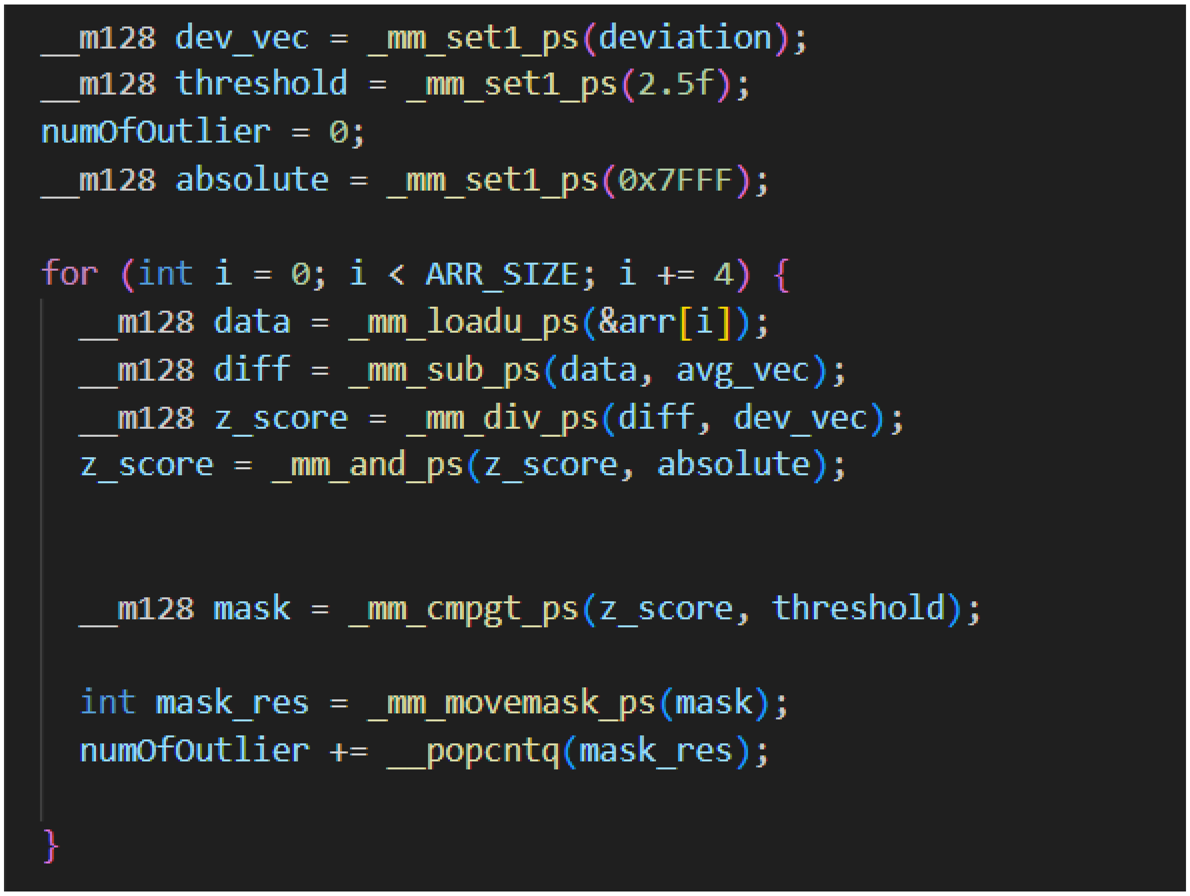


```
__m128 sum_vec = _mm_setzero_ps();
for (int i = 0; i < ARR_SIZE; i += 4) {
  __m128 data = _mm_loadu_ps(&arr[i]);
  sum_vec = _mm_add_ps(sum_vec, data);
}


float sum_array[4];
_mm_storeu_ps(sum_array, sum_vec);
sum = sum_array[0] + sum_array[1] + sum_array[2] + sum_array[3];
avg = sum / ARR_SIZE;


__m128 avg_vec = _mm_set1_ps(avg);
sum_vec = _mm_setzero_ps();

for (int i = 0; i < ARR_SIZE; i += 4) {
  __m128 data = _mm_loadu_ps(&arr[i]);
  __m128 diff = _mm_sub_ps(data, avg_vec);
  __m128 sq_diff = _mm_mul_ps(diff, diff);
  sum_vec = _mm_add_ps(sum_vec, sq_diff);
}


_mm_storeu_ps(sum_array, sum_vec);
float variance_sum = sum_array[0] + sum_array[1] + sum_array[2] + sum_array[3];
deviation = sqrt(variance_sum / ARR_SIZE);


__m128 dev_vec = _mm_set1_ps(deviation);
__m128 threshold = _mm_set1_ps(2.5f);
numOfOutlier = 0;
__m128 absolute = _mm_set1_ps(0x7FFF);

for (int i = 0; i < ARR_SIZE; i += 4) {
  __m128 data = _mm_loadu_ps(&arr[i]);
```

In this section, the sum is computed using **packed single-precision operations**, meaning the calculations are performed on **four**

floating-point numbers simultaneously. Then, the results of these four numbers are stored in the variable **sum_array** and added together.

Then the mean is obtained using a division, and its value is placed into all four elements of a SIMD variable using the **set1** instruction. This SIMD variable, containing four copies of the mean, is then used to compute the **variance**. Afterward, the four variance values are added together, and finally, using the resulting variance, the **standard deviation** is computed.

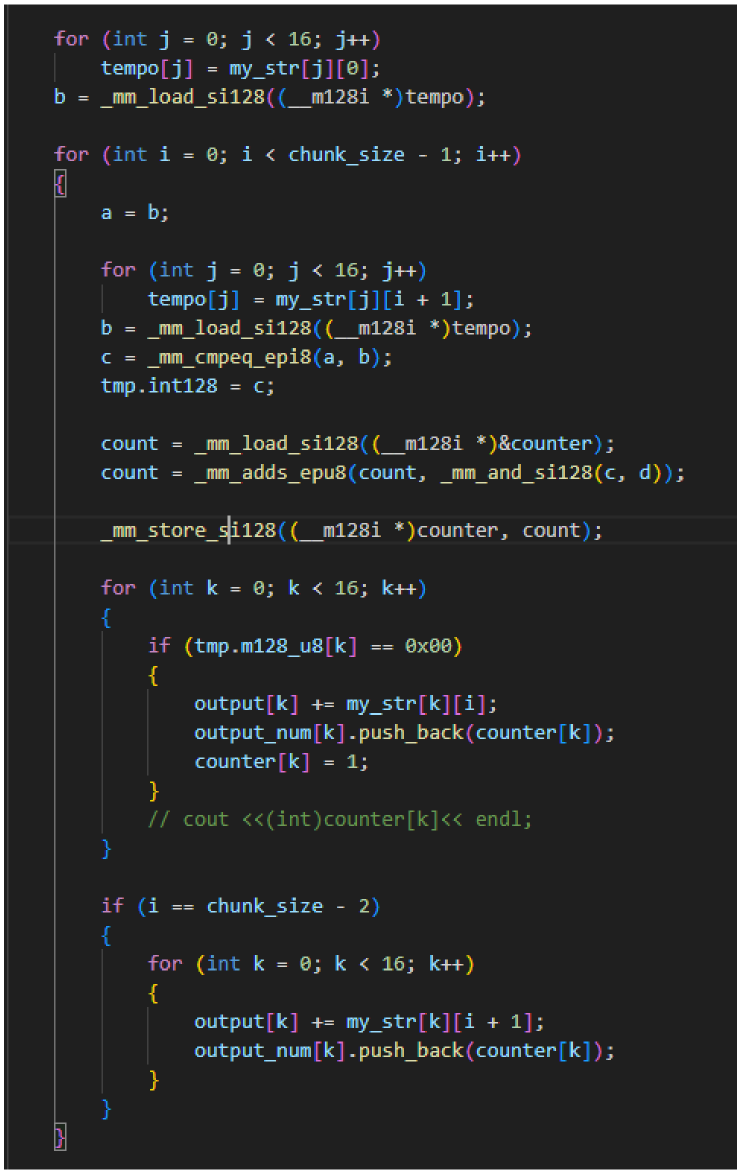


```
__m128 dev_vec = _mm_set1_ps(deviation);
__m128 threshold = _mm_set1_ps(2.5f);
numOfOutlier = 0;
__m128 absolute = _mm_set1_ps(0x7FFF);

for (int i = 0; i < ARR_SIZE; i += 4) {
  __m128 data = _mm_loadu_ps(&arr[i]);
  __m128 diff = _mm_sub_ps(data, avg_vec);
  __m128 z_score = _mm_div_ps(diff, dev_vec);
  z_score = _mm_and_ps(z_score, absolute);


  __m128 mask = _mm_cmpgt_ps(z_score, threshold);

  int mask_res = _mm_movemask_ps(mask);
  numOfOutlier += __popcntq(mask_res);

}
```

Finally, the **z-scores** are computed in parallel, and each value is **ANDed with 0x7FFF** so that they all become positive. These values are then compared with the **threshold** (which is 2.5), and the results are stored in a packed variable containing either **0xFFFF** or **0x0000** for each element. In the end, the number of **0xFFFF** entries is counted, and that amount is added to **numOfOutlier**.

Part 3

The algorithm used in this section works by dividing the string into **16 parts**, with each part assigned to an index of the SIMD. Finally, these 16 parts are **merged** together.

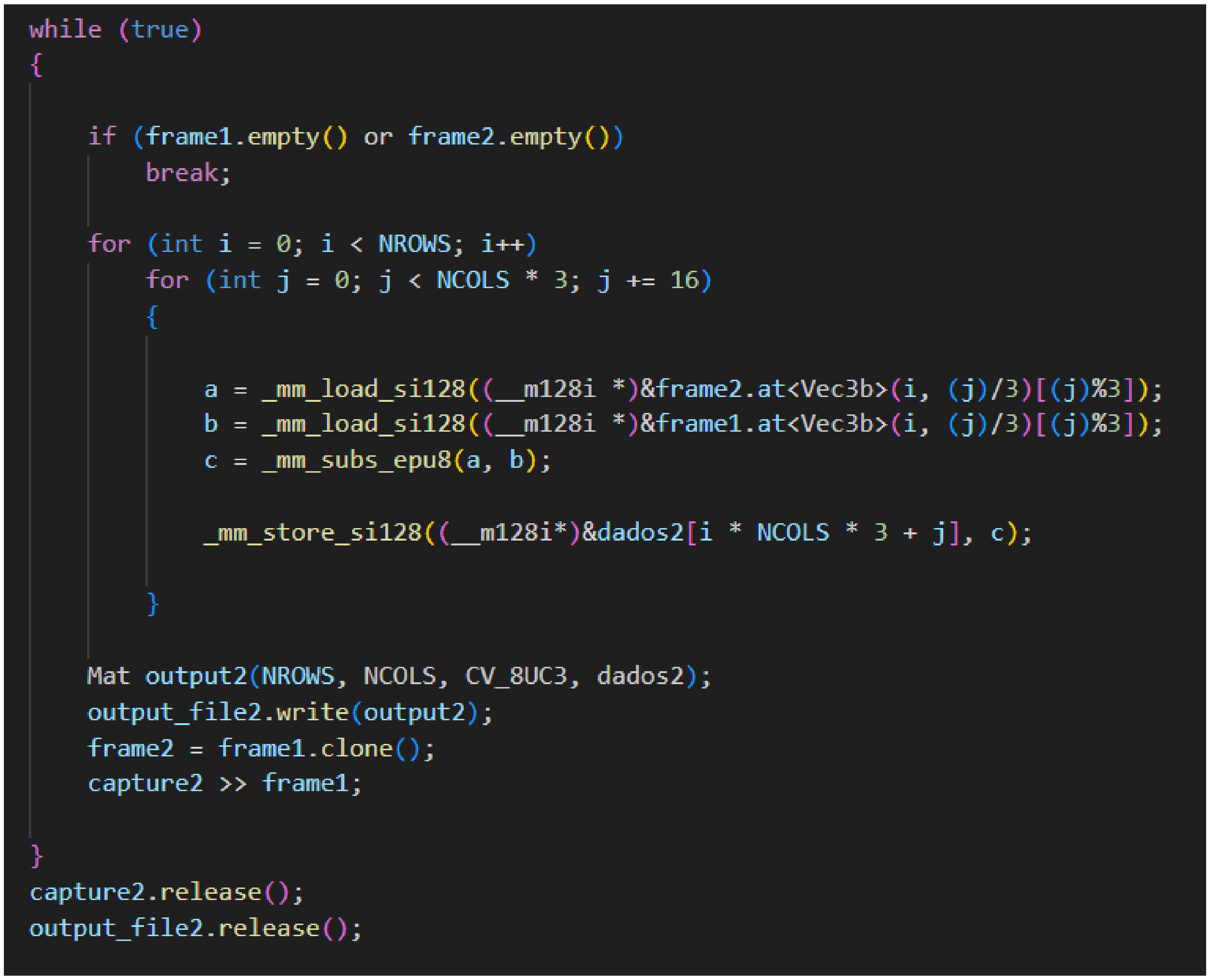


```
for (int j = 0; j < 16; j++)
    tempo[j] = my_str[j][0];
b = _mm_load_si128((__m128i *)tempo);

for (int i = 0; i < chunk_size - 1; i++)
{
    a = b;

    for (int j = 0; j < 16; j++)
        tempo[j] = my_str[j][i + 1];
    b = _mm_load_si128((__m128i *)tempo);
    c = _mm_cmpeq_epi8(a, b);
    tmp.int128 = c;

    count = _mm_load_si128((__m128i *)&counter);
    count = _mm_adds_epu8(count, _mm_and_si128(c, d));

    _mm_store_si128((__m128i *)counter, count);

    for (int k = 0; k < 16; k++)
    {
        if (tmp.m128_u8[k] == 0x00)
        {
            output[k] += my_str[k][i];
            output_num[k].push_back(counter[k]);
            counter[k] = 1;
        }
        // cout <<(int)counter[k]<< endl;
    }

    if (i == chunk_size - 2)
    {
        for (int k = 0; k < 16; k++)
        {
            output[k] += my_str[k][i + 1];
            output_num[k].push_back(counter[k]);
        }
    }
}
```

The **speedup** achieved is less than 1, around **0.7**, and the reason for this could be that this problem is not well-suited for **SIMD**, since the entire string is interdependent—unlike Question 4, where each pixel only interacts with its corresponding pixel in the next frame. Additionally, the parallel algorithm must traverse the string multiple times to **merge the 16 separate parts**, which incurs extra overhead. The overall **serial algorithm** has complexity **O(n)**, and in the parallel implementation, it has often been necessary to traverse the string multiple times to take advantage of SIMD capabilities. For this reason, there is **little gain** from the parallel implementation of this problem, and it can even result in **longer execution time**.

Part 4

The algorithm used for this question works as follows: **16 bytes** from the image are taken, where each byte corresponds to an **RGB channel**, and subtracted from the corresponding bytes in the next frame. The results are then stored, and ultimately a frame of the image is reconstructed. This process is repeated until the end.



```
while (true)
{

    if (frame1.empty() or frame2.empty())
        break;

    for (int i = 0; i < NROWS; i++)
        for (int j = 0; j < NCOLS * 3; j += 16)
        {

            a = _mm_load_si128((__m128i *)&frame2.at<Vec3b>(i, (j)/3)[(j)%3]);
            b = _mm_load_si128((__m128i *)&frame1.at<Vec3b>(i, (j)/3)[(j)%3]);
            c = _mm_subs_epu8(a, b);

            _mm_store_si128((__m128i*)&dados2[i * NCOLS * 3 + j], c);

        }

    Mat output2(NROWS, NCOLS, CV_8UC3, dados2);
    output_file2.write(output2);
    frame2 = frame1.clone();
    capture2 >> frame1;

}
capture2.release();
output_file2.release();
```

The **speedup** achieved is approximately **3**. The reason this value is far from the maximum possible speedup of 16 is that the portion that takes the most time is **memory reading**, which can only be performed **serially** and operates much slower compared to the processor's computation speed.