Assignment2 Report

Student name: Yingzheng Pan Student number: 19336862

For this assignment our task is to implement student_conv function and try to make it perform running time faster than multichannel_conv function.
As the Figure 1 shown below, that's the first part of my student_conv, the function definition for student_conv, which takes in an input image, a set of kernels, and an output output. The function also takes in the width, height, nchannels, nkernels, and kernel_order parameters. At the beginning, the first line starts an OpenMP parallel region, which allows the for loop that follows to be executed in parallel across multiple threads. #pragma omp parallel for is a compiler directive for OpenMP, which is used to parallelize for loop and can greatly improve the performance of the code by allowing multiple threads to work on different parts of the computation simultaneously. This will lead to a significant reduction in the running time of the program, particularly when dealing with large data sets. The first for loop iterates over each of the nkernels provided. And then create a new float 3D Array called kernel. For the other three nested for loops copy the current kernel being processed from the kernels input array to the kernel array, converting the elements to float in the process. Transform int16_t array type kernels to store as a float array type kernel[x][y][channel]. The reason for doing this is in the SSE operations, int16_t array type cannot be used to perform SSE operations and store operation results. We need to use the float array format to perform the SSE operations later. These processes are required for efficient SIMD operations.

```
void student_conv(float *** image, int16_t **** kernels, float *** output,
                  int width, int height, int nchannels, int nkernels,
                  int kernel_order)
{

    #pragma omp parallel for
    for (int kernel_idx = 0; kernel_idx < nkernels; kernel_idx++) {
        // Copy kernel to a new buffer with float elements and channel-first layout
        float kernel[kernel_order][kernel_order][nchannels];
        for (int c = 0; c < nchannels; c++) {
            for (int x = 0; x < kernel_order; x++) {
                for (int y = 0; y < kernel_order; y++) {
                    kernel[x][y][c] = (float) kernels[kernel_idx][c][x][y];
                }
            }
        }
```

Figure 1

As the Figure 2 shown below, at the beginning there are two outers for loops and #pragma omp simd. These for loops iterate over each pixel in the output

array, processing each pixel in parallel across multiple threads using SIMD instructions, which can result in faster execution times on processors that support SIMD instructions when processing large data sets. By utilizing SIMD instructions, the #pragma omp simd, the code can take advantage of the parallel processing capabilities of CPUs, which can lead to faster and more efficient execution. This can result in significant speedups.
For the following two lines, we create two sums to initialize these SSE registers to zero, the __m128d data type can hold two 64-bit floating-point values. The reason why we create two-part sum to calculate the operation is for Intel SSE, which can hold 128 bits, four floating point 32 numbers. And then we use two __m128d type SSE registers to store two of the four numbers respectively. SSE 128-bit registers can store 4 single-precision floating-point numbers, which are each 32 bits in size. However, when performing multiplication and addition operations on these numbers using SSE instructions, the result is a pair of 64-bit double-precision floating-point numbers (each 64 bits in size) that cannot be stored in a single 128-bit register. Therefore, to perform SIMD operations on 4 single-precision floating-point numbers using SSE instructions, we need to use two 128-bit and both contains two 64-bit floating-point values registers to store the intermediate results. The first register (sum1) stores the lower 2 double-precision numbers of the 4 partial sums, and the second register (sum2) stores the higher 2 double-precision numbers of the 4 partial sums. By doing this, we can efficiently use SSE instructions to perform 4 multiplication and 4 addition operations simultaneously, which can significantly speed up the computation compared to performing these operations sequentially in the CPU.
After that, there are three nested for loops iterate over each element of the kernel array and corresponding pixels in the image array to compute the convolution. The loop for (int c = 0; c < nchannels; c += 4) is iterating over the channels in increments of 4. This is because in SSE it has four floating-point values, we will process 4 channels at once using SSE. we use SSE instructions to process 4 floats at a time. By incrementing c by 4 in each iteration, the code is able to load 4 input pixels and 4 kernel elements into SSE registers at once, allowing for efficient processing using SSE instructions. The computation in the nested for loop calculates the sum of the products of the kernel and image values for each channel. Each iteration of the innermost loop performs four separate multiplications and additions. For each element, four input pixels are loaded using _mm_loadu_ps, and four kernel elements are loaded using _mm_load_ps. Firstly we use _mm_loadu_ps and _mm_load_ps to load the kernel and image values for each channel and then we need to compute these 4 partial sums, we use _mm_mul_ps to do multiplication between the kernel and image values and store the multiplication result into the SSE register(mul). For the SSE register _m128, they are both have four 32-bit floating-point values. Now we need to add the first two values in the array, and then add the last two values. The

_mm_cvtps_pd function converts each partial sum from single-precision floating point to double-precision floating point. The result of each multiplication is a 32-bit float value, so the _mm_cvtps_pd intrinsic is used to convert it to a 64-bit double precision value, which can be added to the 128-bit _m128d data type register using _mm_add_pd. mul is an SSE register that contains four single-precision (32-bit) floating-point numbers. mul is the result of multiplying four input pixel values with their corresponding kernel elements. The _m128d sum1 is the summation of the first two value of the 4 partial values in multiplication result(mul). For the second summation(sum2), because we have already add the first two values and then we need to sum the last two values of the 4 partial values in multiplication result(mul). The _mm_movehl_ps function swaps the high and low halves of the mul register. This is done to add together the upper and lower halves of the vector separately. And then we add the last two values and store in _m128d sum2 SSE register. Since the _mm_add_pd function can only add two values at a time. In particular, the first _mm_add_pd(sum1, _mm_cvtps_pd(mul)) adds the first two 32-bit floating point values in the mul register (which has four such values) to the sum1 register. The second _mm_add_pd(sum2, _mm_cvtps_pd(_mm_movehl_ps(mul, mul))) adds the last two 32-bit floating point values in the mul register to the sum2 register. In order to add the last two values in mul, we use the _mm_movehl_ps(mul, mul) instruction shuffles the mul register such that its high 64 bits (corresponding to the third and fourth 32-bit floating point values) are moved to the low 64 bits of register, and the low 64 bits (corresponding to the first and second 32-bit floating point values) are moved to the high 64 bits of the register. Now the last two values are moved to the front position, and then use _mm_add_pd to add the front position two values which was originally the last two elements. _mm_cvtps_pd then converts these values to the double precision value, which is added to sum2.

For the final part, __m128d sums = _mm_add_pd(sum1, sum2). This line adds the values in sum1 and sum2 element-wise and stores the result in a new 128-bit register sums. And then _m128d sum = _mm_hadd_pd(sums, sums). This line horizontally adds the elements in sums, _mm_hadd_pd intrinsic function horizontally adds adjacent double-precision floating-point values in the 128-bit SSE register sums, producing two new double-precision floating-point values, which adds the lower and upper 64-bit values together, and stores the result in a new 128-bit register sum. So, after the call to _mm_hadd_pd(sums, sums), the sum SSE register will contain two identical double-precision floating-point values. This is intentional, since the final result is a single-precision floating-point value that can be stored in a 32-bit memory location. The two identical double-precision floating-point values are converted to a single-precision floating-point value using the _mm_cvtpd_ps intrinsic function before being stored in memory using _mm_store_ss. Because they are two same double floating-point numbers, it is okay to

convert them into single floating-point numbers. Finally we use _mm_store_ss to store the output final result into the output array. when we convert the double floating-point value to a single floating-point value, which is then stored in the output array. And then our output array have our final result.

```
        // Process each pixel in the output image
        #pragma omp simd
        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {
                // Initialize SSE registers to zero
                __m128d sum1 = _mm_setzero_pd();
                __m128d sum2 = _mm_setzero_pd();

                // Process the kernel for the current pixel
                for (int kx = 0; kx < kernel_order; kx++) {
                    for (int ky = 0; ky < kernel_order; ky++) {
                        for (int c = 0; c < nchannels; c += 4) {
                            // Load 4 input pixels and kernel elements
                            __m128 img = _mm_loadu_ps(&image[x + kx][y + ky][c]);
                            __m128 krn = _mm_load_ps(&kernel[kx][ky][c]);

                            // Compute 4 partial sums
                            __m128 mul = _mm_mul_ps(img, krn);
                            sum1 = _mm_add_pd(sum1, _mm_cvtps_pd(mul));
                            sum2 = _mm_add_pd(sum2, _mm_cvtps_pd(_mm_movehl_ps(mul, mul)));
                        }
                    }
                }

                __m128d sums = _mm_add_pd(sum1, sum2);
                __m128d sum = _mm_hadd_pd(sums, sums);
                _mm_store_ss(&output[kernel_idx][x][y], _mm_cvtpd_ps(sum));
            }
        }
    }
}
```

Figure 2

As the Figure 3 shown below, we use openMP and SSE methods to make our function perform running time become faster and faster. We use three examples here to see the changing of the performance running time. Firstly use 16 16 3 32 32. My student_conv function is running 2 times faster than multichannel conv function. Secondly we use 32 32 5 128 128. My student_conv function is running 57 times faster than multichannel conv function. Finally, we use 128 128 7 256 256. My student_conv function is running 98 times, which is approximately 100 times faster than multichannel conv function. By implementing openMP and SSE, they can significantly improve the running time of the project function. As the number of data increases, the running time speed increase becomes faster and faster.

```
panyi@macneill:~/lab2$ gcc -O3 -msse4 -fopenmp conv-harness.c -o conv
panyi@macneill:~/lab2$ ./conv 16 16 3 32 32
Multichannel conv time: 15321 microseconds
Student conv time: 7124 microseconds
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.0625
00)
panyi@macneill:~/lab2$ ./conv 128 128 7 256 256
Multichannel conv time: 266408246 microseconds
Student conv time: 2732397 microseconds
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.0625
00)
```

```
panyi@macneill:~/lab2$ ./conv 32 32 5 128 128
Multichannel conv time: 1794624 microseconds
Student conv time: 31205 microseconds
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.0625
00)
```

$$\frac{15321}{7124} = 2.150617631$$

$$\frac{1794624}{31205} = 57.51078353$$

$$\frac{266408246}{2732397} = 97.4998311$$

Figure 3