Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Lab 2: Parallel Multichannel Multikernel Convolution

Concurrent Systems - CSU33014

Group Submission by:

Prathamesh Sai Sankar (Student ID: 19314123)

Tomasz Bogun (Student ID: 19335070)

School of Computer Science and Statistics,
Trinity College Dublin

Contents

1 Assignment Description

In this assignment, we are trying to write an efficient multichannel multikernel convolution routine. We are provided with some sample code which is as follows:

```
void multichannel_conv(float ***image, int16_t ****kernels,
                       float ***output, int width, int height,
                       int nchannels, int nkernels, int kernel_order)
{
int h, w, x, y, c, m;

  for ( m = 0; m < nkernels; m++ ) {
    for ( w = 0; w < width; w++ ) {
      for ( h = 0; h < height; h++ ) {
        double sum = 0.0;
        for ( c = 0; c < nchannels; c++ ) {
          for ( x = 0; x < kernel_order; x++) {
            for ( y = 0; y < kernel_order; y++ ) {
              sum += image[w+x][h+y][c] * kernels[m][c][x][y];
            }
          }
          output[m][w][h] = (float) sum;
        }
      }
    }
  }
}
```

Listing 1: The slow but correct version of matmul provided by Prof. David Gregg.

Our aim is to provide code optimization and speedup this functionality using parallelization and vectorization, and write it inside a function called "student_conv". The target machine is stoker.scss.tcd.ie. This machine contains four processors, with each processor containing eight out-of-order pipelined, superscalar cores. Each core has two-way simultaneous multithreading, supporting Intel SSE4 vector instructions.

2 Optimization strategy

Our optimisation strategy was to mainly use OpenMP to vectorize and parallelize our function. Since the Stoker machine has 4 processors, with 8 cores per processor, and 2 execution threads per code, it means that we can utilise up to 32 execution threads. If we can parallelize the code, there will be major performance improvements. There will also be additional improvements if we are able to vectorize the code.

Figure 1: OpenMP is an application programming interface that supports multi-platform shared-memory multiprocessing programming in C [1].

Initially, we were keen on vectorizing using _m128d but because of overhead issues between SSE and OpenMP, the speedup actually decreased overall. We decided that it was more efficient to do the vectorization with OpenMP. OpenMP provides vectorization that is very efficient and simple to use, which we liked.

3 Standard input sizes we used

Set of standard input sizes					
Set	Image Width	Image Height	Kernel Order	Number of Channels	Number of Kernels
1	64	64	3	32	32
2	128	128	5	64	64
3	256	256	7	128	128

For this document we will be using three standard testing size sets which are shown above. This set of standard input sizes will be used later on to list the execution times we have achieved along with the appropriate speedup.

4 Our optimizations and their execution time

```
int h, w, x, y, c, m;

for ( m = 0; m < nkernels; m++ ) {
  for ( w = 0; w < width; w++ ) {
    for ( h = 0; h < height; h++ ) {
      double sum = 0.0;
      for ( c = 0; c < nchannels; c++ ) {
        for ( x = 0; x < kernel_order; x++) {
          for ( y = 0; y < kernel_order; y++ ) {
            sum += image[w+x][h+y][c] * kernels[m][c][x][y];
          }
        }
        output[m][w][h] = (float) sum;
      }
    }
  }
}
```

Figure 2: OpenMP is an application programming interface that supports multi-platform shared-memory multiprocessing programming in C [1].

If we inspect the original unoptimised code above, we discover that it consists of two parts. The first part is the outer for-loops (first blue box)that don't have a data dependency, and the second part is the inner for-loops (second blue box), that do contain a data dependency (in red). This data dependency is the sum variable. Since there are no data dependencies between each iteration of the outer 3 loops, it means that they can be executed in any order and can be parallelized because each iteration does not depend on the iteration of another. We parallelized the outer 3 loops using OpenMP as show below:

```
//Parallelize 3 nested for-loops, with variables m,w,h,x,y,c being shared
amongst the execution threads
#pragma omp parallel for collapse(3) shared(m,w,h,x,y,c)
for (m = 0; m < nkernels; m++){
  for (w = 0; w < width; w++){
    for (h = 0; h < height; h++){
      double sum = 0.0;
      .....
```

Listing 2: Parallelizing the outer 3 loops using OpenMP

The **"#pragma omp parallel for"** tells the compiler to parallelize the outer for-loops by running each iteration on a separate execution thread. The **"collapse(3)"** part tells the compiler that the outer 3 loops should be parallelized too, not just the outer loop. We can do this since there are no data dependencies in the 3 outer loops. The **"shared(m,w,h,x,y,c)"** part tells the compiler that the variables are shared amongst the execution threads. Using just this line, we observed an average speedup for out input sizes as follows:

```
saisankp@stoker:~/CSU33014/lab2$ ./a.out 64 64 3 32 32

The original conv time: 212470 microseconds
The student conv time (improved): 17611 microseconds
The speedup: x12

COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
```

Figure 3: Set 1 had a speedup of 12x

```
saisankp@stoker:~/CSU33014/lab2$ ./a.out 128 128 5 64 64

The original conv time: 5052817 microseconds
The student conv time (improved): 251936 microseconds
The speedup: x20

COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
```

Figure 4: Set 2 had a speedup of 20x

```
saisankp@stoker:~/CSU33014/lab2$ ./a.out 256 256 7 128 128

The original conv time: 119921284 microseconds
The student conv time (improved): 4469528 microseconds
The speedup: x26

COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
```

Figure 5: Set 3 had a speedup of 26x

These was a significant improvement compared to the original unoptimized code. If we want a larger speedup, we can utilise vectorization for the inner loop. We attempted to use Intel's SSE instructions to perform this vectorization, but we got a slow down and inconsistent answers due to overheads and dependencies between OpenMP and SSE, so after some research we decided to use **OpenMP for vectorization too**. Before attempting vectorization, we need to ensure that the types of the image matrix and the kernel matrix are the same, because they are involved with each other in the following multiplication operation:

```
sum += image[w+x][h+y][c] * kernels[m][c][x][y];
```

Figure 6: The image and kernel matrix are used to get the sum.

We cannot perform operations of two vectors that contain values of different sizes because they are not aligned in memory. **The passed in image matrix contains floats whereas the passed in kernels matrix contains int16_t values (i.e short ints)**. We can solve this by introducing some overhead that involves creating a new kernels matrix before the main function and copying values while casting them to the type we want. The following nested loops copy values from the int16_t kernel to the float kernel:

```
//Parallelize and vectorize nested 4 loops
#pragma omp parallel for simd collapse(4)
for (int a = 0; a < nkernels; a++){
  for (int b = 0; b < nchannels; b++){
    for (int d = 0; d < kernel_order; d++){
      for (int e = 0; e < kernel_order; e++){
        kernelsAsFloats[a][d][e][b] = kernels[a][b][d][e];
      }
    }
  }
}
```

Listing 3: Convert the kernels vector (short int) to a float vector.

This copying is also parallelized and vectorized using OpenMP to maximise performance. The parallelization occurs from the **"#pragma omp parallel for"** which tells the compiler to parallelize the loop by running each iteration on a separate execution thread. The vectorization occurs from the **"simd"** which is a directive is applied to a loop to indicate that multiple iterations of the loop can be executed concurrently by using SIMD instructions [2]. The **"collapse(4)"** tells the compiler that the outer 4 loops should be parallelized as well. Because this overhead is introduced, the performance difference will be greater with larger input sets than smaller ones, because the overhead will be less significant.

Now it is possible to start vectorizing the main loop since the data types of the two matrices match and are aligned in memory. The below code is the first stage of the vectorization of the inner **"nchannels"** loop:

```
#pragma omp simd
for (c = 0; c < nchannels; c += 2){
  sum += image[w+x][h+y][c] * kernelsAsFloats[m][x][y][c];
  sum += image[w+x][h+y][c+1] * kernelsAsFloats[m][x][y][c+1];
}
```

Listing 4: Initially trying to vectorize the inner loop.

The **"pragma omp simd"** part tells the compiler to vectorize the inner loop. In this example, only two iterations will be vectorized into one, but we can test multiple vector sizes that contain values such as 1, 2, 4, 8 ...(the powers of 2 up to 32 since nchannels has a minimum value of 32).

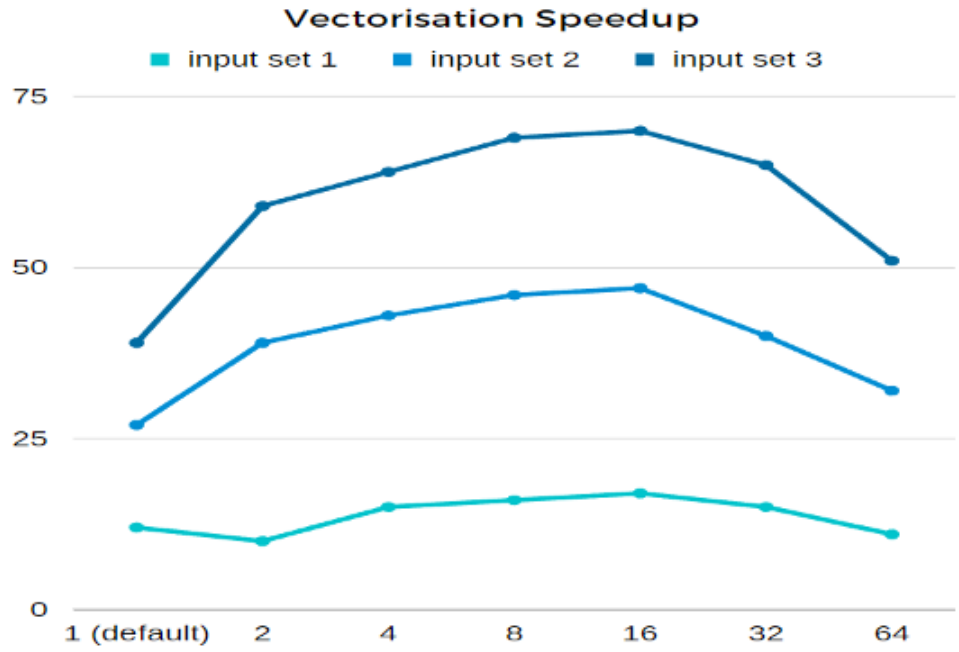

Figure 7: Size of the vector we use (x-axis) compared to the speedup (y-axis) we get for each of the input sizes

As we can see, **the optimal vector size of 16 floats** or equivalent to a _m512 vector. After this size, the performance decreases. Using vectorization in addition to parallelization, we obtain improved speedups:

```
saisankp@stoker:~/CSU33014/lab2$ ./a.out 64 64 3 32 32

The original conv time: 212228 microseconds
The student conv time (improved): 9916 microseconds
The speedup: x21

COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
```

Figure 8: Set 1 had a speedup of 21x

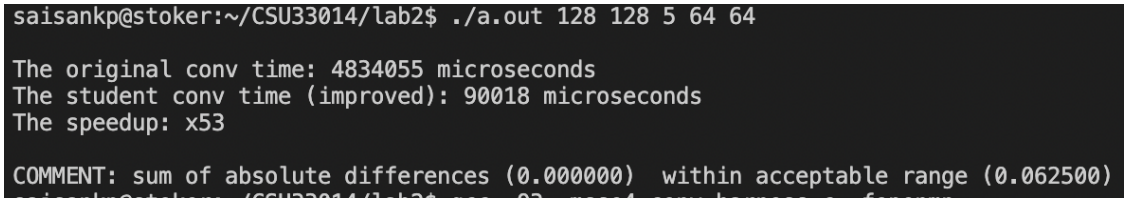

Figure 9: Set 2 had a speedup of 53x

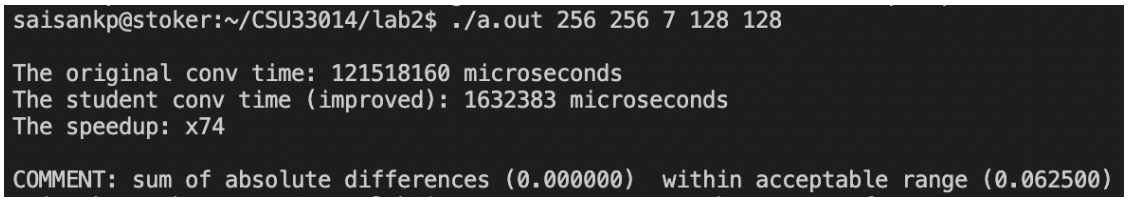

Figure 10: Set 3 had a speedup of 74x

We wanted to test **other inputs** other than the ones in the standard testing size sets to ensure that our code was also optimizing other inputs as well, and we still achieved very high speedups which we were very happy with.

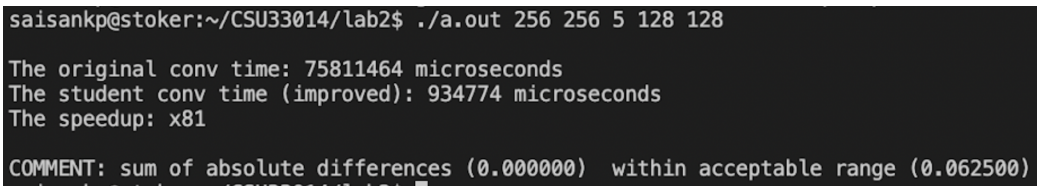

Figure 11: A x81 speedup with different inputs

Overall, we are very happy with how the assignment turned out. There were some setbacks such as spending time on vectorizing with _m128d which ended up slowing down our speedup, but we eventually got through it and used OpenMP for the vectorization as well. We learned a lot during this assignment and gained experience in writing an efficient multichannel multikernel convolution routine for a target machine, reinforcing what we learnt in the module with parallelization and vectorization.

5 References

1. Wikipedia. 2022 *OpenMP* [Online]. Available from: https://en.wikipedia.org/wiki/OpenMP [accessed 1st April 2022].

2. IBM. 2022 *#pragma omp simd* [Online]. Available from: https://www.ibm.com/docs/en/xl-c-and-cpp-linux/16.1.0?topic=pdop-pragma-omp-simd [accessed 2nd April 2022].