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

CSU33014 Concurrent Systems I
Assignment #2 Report

Mykhailo Bitiutskyy, Anand Sainbileg, Andrii Yupyk

April 4, 2024

Contents

1 Introduction

The challenge this assignment poses is to optimise an already existing solution for image processing algorithm. This algorithm involves a convolution of a 3D image matrix with a set of 4D kernels matrices and channels in order to produce a 3D output matrix. The main goal of the assignment is to parallelise the algorithm in order to achieve the best possible performance. The following sections will explain the implementation of the solution, the steps taken to achieve the best performance and the benchmarks of the solution.

2 Solution Development

The development of the fast solution involved a number of steps. The first step was to understand the existing solution and the algorithm it implements. The second step was to identify the bottlenecks in the existing solution and to come up with a plan to parallelise the algorithm. The third step was to try different approaches to parallelisation and vectorisation and to benchmark them. The following sections will explain the steps taken to achieve the best performance.

2.1 Step 1 - Understanding the Existing Solution

The existing solution is a simple implementation of the image processing algorithm. The algorithm involves a convolution of a 3D image matrix with a set of 3D kernels matrices and channels in order to produce a 3D output matrix. Basically, for each kernel and channel, the algorithm convolves the image with the kernel and channel and produces the output.

2.2 Step 2 - Identifying the Bottlenecks and Planning the Optimisation

The algorithm is implemented in a single thread and is not optimised for multiprocessor systems. It is neither parallelised nor vectorised. It also has sub-optimal spatial locality, which results in a lot of cache misses. When it comes to nested "for" loops, it makes the most sense in OpenMP to parallelise the outermost loop. This is because typically the outermost loop is the one that is executed the most times and is the most time-consuming. It is also worth noting, that parallelisation in OpenMP generates a significant overhead, so it is important to choose wisely which loop to parallelise. Another point worth mentioning is that the only complex computation lies in the innermost loop. It is an interesting subject for vectorisation, since the operation remains the same for all the elements in the loop and there is no data dependency either.

2.3 Step 3 - Trying Different Approaches to Parallelisation and Vectorisation

2.3.1 Parallelisation

As it was mentioned earlier, we proceeded to parallelise the outermost loop in OpenMP. The parallelisation was done in a simple manner. We used the "#pragma omp parallel for" directive to parallelise the loop. The results were promising. The performance increased significantly, but there was still room for improvement. That immediately yielded a x20 speedup.

```
#pragma omp parallel for
for (m = 0; m < nkernels; m++) {
    for (w = 0; w < width; w++) {
        for (h = 0; h < height; h++) {
            // ...
        }
    }
}
```

As we discovered later, specifying the private and shared variables can result in a small, yet noticeable and stable performance increase. This is because the OpenMP runtime does not have to guess which variables are shared and which are private, so it saves time and memory by not having to do so. We have also attempted to use different scheduling types with different chunk sizes, but the results were either on par with the default static scheduling or worse. This makes sense because the amount of work in each iteration of the outermost loop is identical, since we iterate over arrays with equal sizes, not dynamic data structures. Another attempt at optimisation of the parallelisation was to use the collapse clause. This clause allows to collapse multiple loops into one, which can be beneficial in terms of performance. However, the results were not as good as expected. The performance was on par with the default parallelisation.

```
#pragma omp parallel for private(m, w, h, c, x, y) shared(image, _kernels, output) collapse(3)
for (m = 0; m < nkernels; m++) {
    for (w = 0; w < width; w++) {
        for (h = 0; h < height; h++) {
            // ...
        }
    }
}
```

As it can be seen from the code snippet above, the collapse clause is used to collapse the 3 outer loops into one. We believe the reason it did not improve performance is because the workload is already equally distributed, so the result was not affected. Final parallelisation implementation is shown below.

```
#pragma omp parallel for private(m, w, h, c, x, y) shared(image, _kernels, output)
for (m = 0; m < nkernels; m++) {
    for (w = 0; w < width; w++) {
        for (h = 0; h < height; h++) {
            // ...
        }
    }
}
```

2.3.2 Vectorisation

The next step was to vectorise the calculations in the innermost loop. Those operations are repetitive and do not have any data dependencies. This makes them a perfect candidate for vectorisation. However, there was one big issue. The structure of the "kernels" 4D array was not vectorisable in its initial state. The problem lied in the order of the dimensions of "kernels". The dimensions were ordered as follows: nkernels, nchannels, kernel_order, kernel_order. Keeping in mind that we can only load 4 float-type values from the last dimension, the issue is clear now. "kernel_order"'s highest value is 7, which practically makes the vectorisation useless. This made us think about reordering of the dimensions, so the last dimension is the one with a significantly higher value than the rest and in this case matches the last dimension of "image" array. The new order of the dimensions is as follows: nkernels, kernel_order, kernel_order, nchannels. Since we cannot modify the original code, we had to create a new array and copy the values from the original array to the new one. We also did not forget to parallelise the copying process to lose as less time as possible on the conversion process. It is worth noting that such rearrangement improves the spatial locality as well, so not only the solution becomes vectorisable, but we lose less time on memory access. Lastly, we converted the "kernels" array of type 'int16_t' into 'float' in order to prepare it for SIMD instructions. The code below shows the copying and reordering process.

```
#pragma omp parallel for private(m, c, x, y) shared(kernels, _kernels)
for (m = 0; m < nkernels; m++) {
    for (x = 0; x < kernel_order; x++) {
        for (y = 0; y < kernel_order; y++) {
            for (c = 0; c < nchannels; c++) {
                _kernels[m][x][y][c] = (float)kernels[m][c][x][y];
            }
        }
    }
}
```

Such reordering made the vectorisation process possible. There was one additional small rearrangement that had to be done. Even though the order of the "kernels" array was changed, the order of loops was the same. Since we decided to vectorise the iteration over "nchannels", we made the iteration over that variable as the innermost loop. The snippet below shows the new order of the loops.

```
for (x = 0; x < kernel_order; x++) {
    for (y = 0; y < kernel_order; y++) {
        for (c = 0; c < nchannels; c += 4) {
            // Vectorised operations
        }
    }
}
```

Finally, the vectorisation itself. Our initial approach was to load the values from "image" and converted "_kernels" array into the corresponding vectors. Then we quickly noticed that the "sum" is of type 'double' and the values in the vectors are of type 'float'. That made us think that the product of 'image[w+x][h+y][c]' and '_kernels[m][x][y][c]' would mean losing that valuable double-precision. This is why we decided to convert the values in the vectors into 'double' before calculating the product. That process involved the following steps:

1. Load the values from "image" and "_kernels" into the corresponding vectors.
2. Convert the first 2 values in the vectors into 'double'.
3. Calculate the product of the vectors.
4. Accumulate the result in the "sum2" vector.
5. Shuffle the vectors to get the 3rd and 4th values into the first 2 positions.
6. Convert the 3rd and 4th values into 'double'.
7. Calculate the product of the vectors.
8. Accumulate the result in the "sum2" vector.

Although it was a completely viable solution with significant performance increase (up to x55 with medium-sized parameters), we did not take into account one small detail. As it was mentioned earlier, the "sum" variable is of type 'double'. And we know that we cannot ensure the precision when multiplying 'float' values. However, 'float' value we get from the "_kernels" array is not really 'float'. It is still 'int16_t' value, but casted to 'float'. After we noticed that, we realised that multiplication of 'float' and any integer type would not affect the precision, thus fits into that same 'float' type boundaries. So, if the precision is not affected, we can convert "image" and "_kernels" values into 'double' only after the multiplication. This way we removed 50% of the SIMD instructions and saved an impressively large amount of time. The final solution's gain can be up to x105 with medium-sized parameters. The final vectorisation implementation is shown on the next page.

```
#pragma omp parallel for private(m, w, h, c, x, y) shared(image, _kernels, output)
for (m = 0; m < nkernels; m++) {
    for (w = 0; w < width; w++) {
        for (h = 0; h < height; h++) {
            double sum = 0.0;
            __m128d sum2 = _mm_setzero_pd();

            for (x = 0; x < kernel_order; x++) {
                for (y = 0; y < kernel_order; y++) {
                    for (c = 0; c < nchannels; c += 4) {
                        // Load 4 elements from kernels and image
                        __m128 image4 = _mm_load_ps(&image[w+x][h+y][c]);
                        __m128 kernel4 = _mm_load_ps(&_kernels[m][x][y][c]);

                        // Get products of image[n] and kernel[n]
                        __m128 prod4 = _mm_mul_ps(image4, kernel4);

                        // Convert prod4[0], prod4[1] to double
                        // Add prod4[0], prod4[1] to sum2
                        __m128d prod2 = _mm_cvtps_pd(prod4);
                        sum2 = _mm_add_pd(sum2, prod2);

                        // Move prod4[2], prod4[3] to the start of the vector
                        // Convert prod4[2], prod4[3] to double
                        // Add prod4[2], prod4[3] to sum2
                        prod4 = _mm_shuffle_ps(prod4, prod4, _MM_SHUFFLE(1,0,3,2));
                        prod2 = _mm_cvtps_pd(prod4);
                        sum2 = _mm_add_pd(sum2, prod2);
                    }
                }
            }

            // Horizontal add on sum2 elements
            sum2 = _mm_hadd_pd(sum2, sum2);
            // Store sum
            _mm_store_sd(&sum, sum2);
            // Update matrix element value
            output[m][w][h] = (float)sum;
        }
    }
}
```

3 Benchmarks

Test 1 (small-sized parameters): ./a.out 100 100 5 128 128
COMMENT: sum of absolute differences (0.000000) within acceptable range (0.062500)
David's conv time: 8558409 microseconds
Student's conv time: 168794 microseconds
Gain over David's conv time: 50.70 times

Test 2 (medium-sized parameters): ./a.out 128 128 7 256 256
COMMENT: sum of absolute differences (0.000000) within acceptable range (0.062500)
David conv time: 134243967 microseconds
Student conv time: 1277308 microseconds
Gain over David's code: 105.10 times

As it can be seen from the benchmarks, with the increase of the dataset size, the performance gain increases as well because the parallelisation and vectorisation show more efficiency with large parameters than sequential code. Provisionally, the gain can be even more impressive with larger parameters.

4 Sources

1. Amazing lectures by David Gregg
2. Intel Intrinsics Guide. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`