TU Delft

Assignment #1: Report

Advanced Multicore Systems

Tudor Voicu – 4422066

Misael Hernandez – 4423615

May 12th, 2015

Introduction

The first assignment of the Advanced Multicore Systems course consists on optimizing vector multiplications through the use of OpenMP and SIMD extensions for Intel processors. With this purpose, we profiled the parallelization capabilities of our system, and compared the optimum parallel execution times with sequential execution times.

OpenMP is an API for C/C++ with facilitates shared memory management for parallelization purposes. Through the use of compiler directives, OpenMP takes care of thread creation, load distribution and memory management for parallelization of C/C++ code.

SIMD extension sets have been integrated in consumer CPUs since 1996 (MMX in Intel), and have been upgraded ever since to allow an increasing level of parallelism per core. The best available implementation (AVX2) allows for 256-bit wide operations, enabling up to 8x32bit Floating Point operations per clock cycle, per core.

In the first part of the lab we used OpenMP both for optimizing vector multiplication and matrix multiplication. This report includes the tasks required for this, along with a comparison of results and the respective conclusions.

The second half of the assignment, we used SIMD (SSE and AVX) extensions for x86 with the purpose of achieving higher instruction throughput per core, testing both cases of vector-matrix multiplication and matrix-matrix multiplication.

Part 1a) Task 1

The first exercise consisted on obtaining the parallel execution time of the vector-matrix multiplication, keeping the size fixed, but changing the number of threads each time. Because the execution time is also dependent on the number of threads, we can observe the exact number of threads that gives the best speedup in our system. Table 1 shows the results for a vector size of 8000, and a number of threads from 1 to 12. The sequential execution ran at an average of 2.504 seconds, which is the baseline used for all the speedup calculations. Figure 1 shows the parallel execution time for each of the number of threads, and Figure 2 does the same for speedup.

Table 1. Parallel execution time and speedup by number of threads

# Threads	Execution time	Speedup
1	2.755	0.909
2	1.449	1.728
3	0.980	2.555
4	0.810	3.091
5	0.901	2.779
6	0.897	2.792
7	0.797	3.142
8	0.791	3.166
9	0.800	3.130
10	0.828	3.024
11	0.831	3.013
12	0.853	2.936

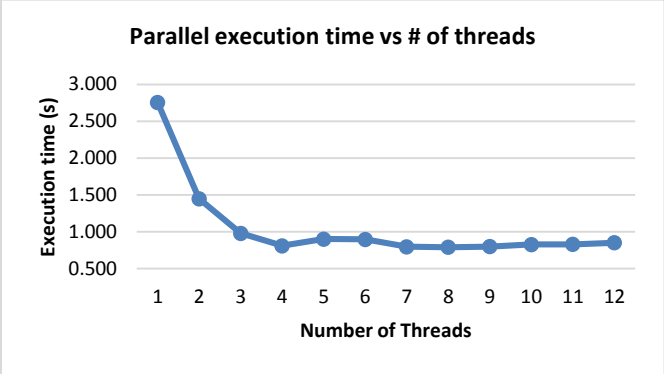

Figure 1. Effect of the number of threads on the execution time

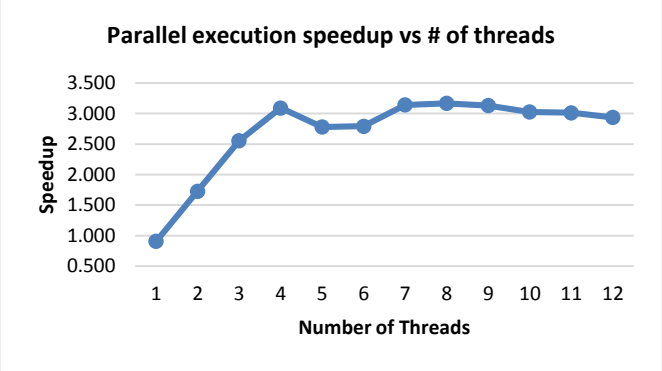

Figure 2. Effect of the number of threads on the speedup achieved by parallelization

What is noticeable from the results is that there are two local minima. The first one occurs at 4 threads and the second one at eight threads. The system in which the program was run has 4 cores, but each one has hyper-threading as well. From 1 thread to 4, the improvement comes from using one more core each time (efficiently). After that, it gets slower because at least one core has to run 2 threads (in average), which comes with a noticeable overhead. However when running 8 threads, the 4 cores with hyper-threading (which allows two threads to be run at each core in a somewhat parallel fashion, for a total of 8) are used most efficiently. More than 8 threads, which is the most that could be run in parallel, the overhead of managing the threads (moving them between the ready queue and execution) affects the performance negatively. So effectively, 8 threads is the most suitable for this system.

Part 1a) Task 2

For the second task we now keep the number of threads constant at the most suitable number, which is 8, and plot the effects of the input size on the execution time. This is done on both the parallel and the serial functions with the purpose of comparing them. Table 1 shows the results. In Figure 3 we can see the comparison of the execution times for the parallel and serial functions in a logarithmic scale. Figure 4 shows the speedup of the parallel version compared to the serial one.

Table 2. Sequential and parallel execution time with speedup by matrix size

Matrix size (N)	Seq Ex	Parallel Ex	Speedup
10	0.000002	0.001697	0.001179
20	0.000006	0.003949	0.001519
40	0.000033	0.007544	0.004374
80	0.000084	0.002200	0.038182
160	0.000327	0.003768	0.086783
320	0.001271	0.002323	0.547137
640	0.007210	0.006441	1.119391
1000	0.019220	0.009433	2.037528
2000	0.111796	0.036506	3.062401
4000	0.540668	0.193470	2.794583
8000	2.712651	0.828773	3.273093
25000	32.525736	13.382240	2.430515

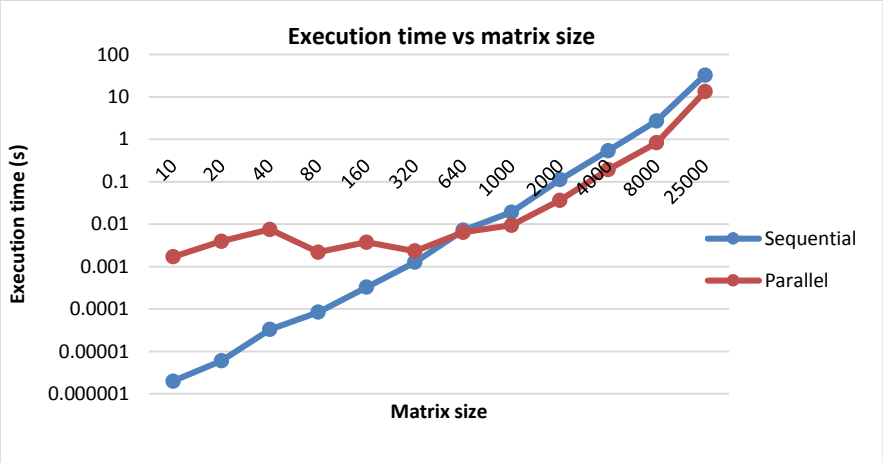

Figure 3. Effect of matrix size on sequencial and parallel execution time

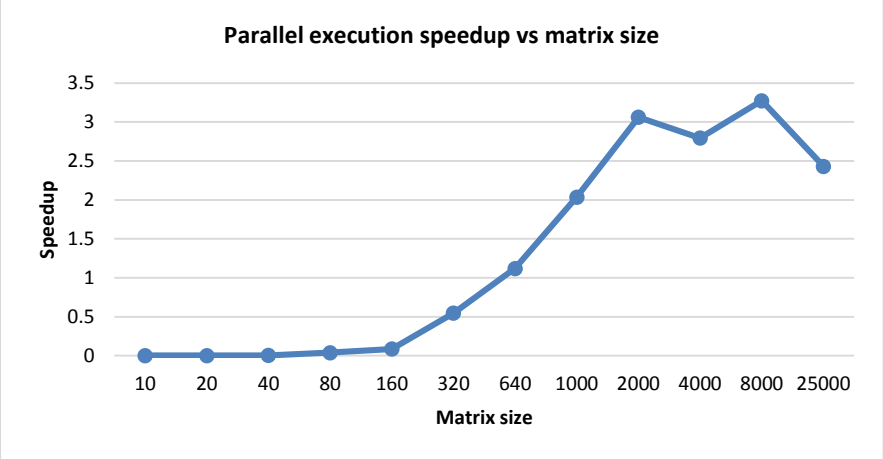

Figure 4. Effect of matrix size on speedup

As seen on Figure 3, the parallel execution time is higher for matrix sizes smaller than 640x640. This can be explained by the overhead involved in parallelization. This involves the creation of the 8 threads and the management of these threads by the operating system and the hardware itself. For small sizes, the overhead is big relative to the calculations, so the execution time ends up being bigger. However, the overhead can be considered more or less constant relative to the input size. This means that as the input grows, the effect of the overhead becomes negligible and the parallel execution becomes faster than the sequential.

Part 1a) Task 3

This task consisted on modifying the current program to execute matrix multiplication. The code for executing the multiplication itself is shown below. Refer to the source code packaged with this report for the rest.

```
void matrix_mult_sq(int size, double *matrix_in,
                    double *matrix_in2, double *matrix_out){
  int rows, cols;
  int j;

  for(rows=0; rows<size; rows++)
    for(cols=0; cols<size; cols++){
      matrix_out[rows*size + cols] = 0.0;
      for(j=0; j<size; j++)
       matrix_out[rows*size + cols] += matrix_in[rows*size + j] * matrix_in2[j*size + cols];
    }
```

Figure 5. Matrix multiplication function

The code as it is has a slight problem with the cache. The pointer for the second input matrix, matrix_in, increases by size, which means that if size is big, it is very possible that the next data is not found in the same cache line, causing more cache misses per inner loop execution. If the output matrix is initialized with zeros beforehand (before the start of the outermost loop), this problem can be solved by exchanging the innermost loop with the middle one. In this way the pointers to the three matrices only increase by one, causing less cache misses per innermost loop execution. For the purpose of this assignment, the code was left as shown, to be faithful to the original code.

Part 1a) Task 4

The last task consisted on improving the performance of the matrix multiplication code using techniques such as OpenMP, SSE or such. For further improvements over the OpenMP implementation we have used the AVX extension, which supports simultaneous 4 operations on double-precision numbers (64 bit floating point numbers) per CPU core. The application to be benchmarked is the matrix-matrix multiplication as requested at exercise 1a.3).

The theoretical improvement should be 4 over the OpenMP configuration (in our case, 8 threads achieved highest performance, running on a 4-core/8 threads Intel CPU). However, from Figure 6 it can be seen that the maximum speed-up is slightly above 2 over the parallel implementation and aprox. 6.3 over the sequential one.

The implementation details and the code are provided in the updated matrix3.c file. The application was running on an Intel i7-4700MQ quad-core mobile (laptop) CPU, with tuned frequency at 2.5 Ghz for any core count (we had to bypass the Intel TurboBoost technology in order to get relevant results) and 8 GB of dual-channel DDR3 RAM (2x4GB).

Table 3. Results in seconds for running different size (N) matrix x matrix multiplication in the three mentioned implementations. Speed-up results are shown in the two left-most columns.

Matrix size (N)	Sequential (sec)	Parallel (sec)	AVX + FMA (sec)	Speed-up over OpenMP	Speed-up over sequential
384	0.5229	0.1807	0.0972	1.8599	5.3806
480	0.8463	0.2772	0.1362	2.0342	6.2115
512	1.2852	0.4540	0.2044	2.2213	6.2884
640	2.4994	0.8280	0.6106	1.3560	4.0933
720	2.9814	0.9706	0.7355	1.3195	4.0533
800	4.4947	1.4029	1.1412	1.2293	3.9386
1024	17.6088	4.4478	3.3742	1.3182	5.2187
2048	217.2149	64.9183	48.7952	1.3304	4.4516

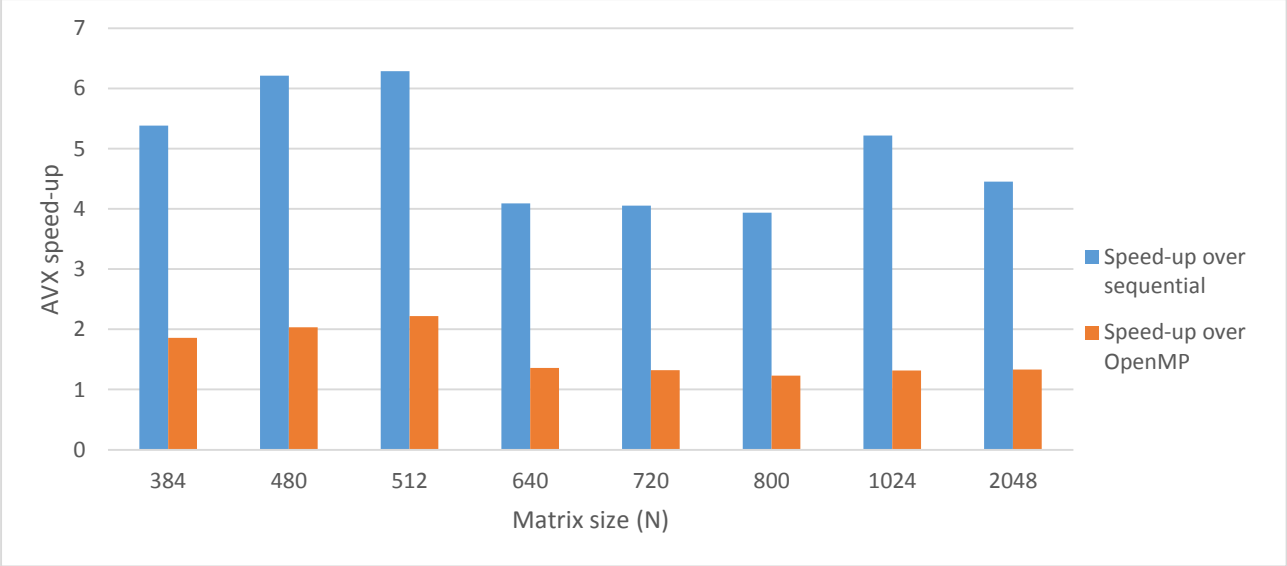

Figure 6. Computed speed-ups where the reference is the AVX with FMA implementation over parallel and sequential code

Part 1b) Task 1 and Task 2

The SSE (Streaming SIMD Extensions) instruction set extension for the x86 architecture enables 128-bit wide SIMD operations. These operations enable the use of 4x32 bit/2x64 bit special purpose registers for performing the same operation on different data, in parallel. Therefore, using this approach should bring a theoretical improvement of 4 times in mathematical computations, when using single-precision floating-point or 2 times improvement when using double precision.

From our measurements (presented in Table 4), the speed-up increases towards 3.4 when we reach the limit of RAM use (over 4 GB).

Matrix x vector - original implementation				**Implementation with arbitrary N**			
N (matrix / vector size)	Sequential	SSE implementation (seconds)	Speed-up	N (matrix / vector size)	Sequential (seconds)	SSE implementation (seconds)	Speed-up
500	0.001741	0.0007	2.487143	499	0.0018	0.000953	1.888772
1000	0.006311	0.003036	2.078722	1001	0.006853	0.003433	1.996213
2000	0.04043	0.014827	2.726782	2002	0.048672	0.021972	2.215183
4000	0.200552	0.069973	2.866134	3999	0.237666	0.086238	2.755931
8000	1.030997	0.339643	3.035531	8001	1.070841	0.369465	2.898356
16000	6.107547	1.939832	3.148493	16002	4.616019	1.720859	2.682392
32000	30.813303	9.363833	3.290672	32000	34.879671	10.877502	3.206588

Table 4 Comparison for the results obtained for the original SSE matrix x vector implementation and the arbitrary-size-matrix implementation

```
void matrix_mult_sse(int size, double *vector_in,
        double *matrix_in, double *vector_out){
                __m128d a_line, b_line, r_line;
                int i, j, idx;
                int size_trunc = size - size%2;
                for (i=0; i<size_trunc; i+=2){
                        j = 0;
                        b_line = _mm_load_pd(&matrix_in[i]);
                        a_line = _mm_set1_pd(vector_in[j]);
                        r_line = _mm_mul_pd(a_line, b_line);
                        for (j=1; j<size; j++) {
                                b_line = _mm_loadu_pd(&matrix_in[j*size+i]);
                                a_line = _mm_set1_pd(vector_in[j]);
                                r_line = _mm_add_pd(_mm_mul_pd(a_line, b_line), r_line);
                        }
                        for (j=1; j<size; j++) {
                                _mm_store_pd(&vector_out[i], r_line);
                        }
                }
                if(size!=size_trunc) {
                        j = 0;
                        b_line = _mm_loadu_pd(&matrix_in[size_trunc]);
                        a_line = _mm_set1_pd(vector_in[j]);
                        r_line = _mm_mul_pd(a_line, b_line);
                        for (j=1; j<size; j++) {
                                b_line = _mm_loadu_pd(&matrix_in[j*size+size_trunc]);
                                a_line = _mm_set1_pd(vector_in[j]);
                                r_line = _mm_add_pd(_mm_mul_pd(a_line, b_line), r_line);
                        }
                        for (j=1; j<size; j++) {
                                double rest[2];
                                _mm_storeu_pd(rest, r_line);
                                vector_out[size_trunc] = rest[0];
                        }
                }
}
```

Code snippet 1 Function for SSE implementation for an arbitrary matrix size (not a multiple of the SIMD)

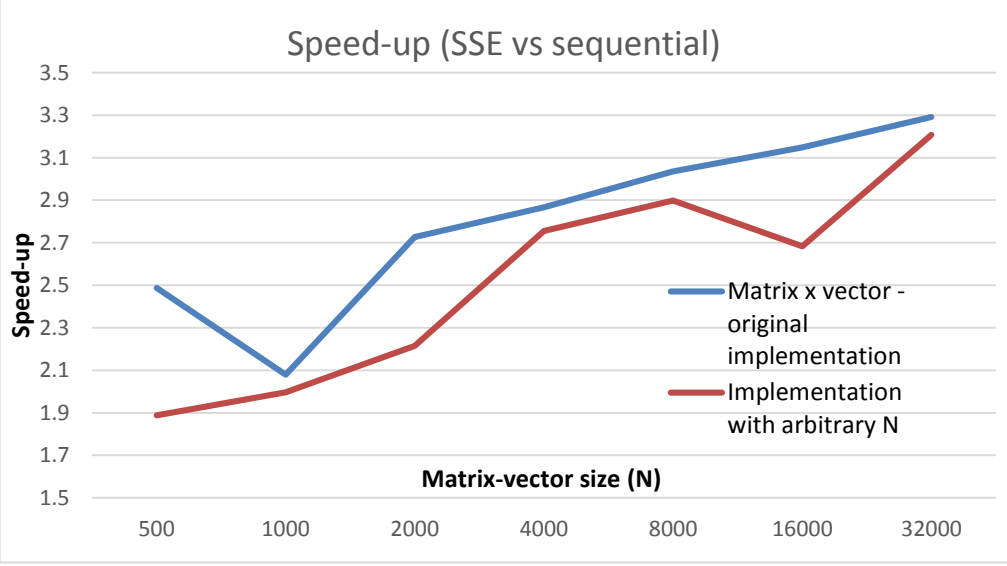

Figure 7. Optimized matrix size vs arbitrary matrix size comparison in terms of parallel speed-up over the sequential code

The implementation with the arbitrary N size (not multiple of 4) can be viewed in matrix-arbVal.c file and run via run_sse_arb_float.sh. In Figure 7, it can be seen that the handling of corner cases of the arbitrary matrix size adds overhead to the execution (condition checks and/or branches inside the loops), reducing the speed-up over the sequential counterpart.

Part 1b) Task 3

For this task we have assumed that the change should be done having the reference the code modified for Task 2. Therefore, for good means of comparison, the code has been adapted to perform computation on *double* type, for matrix x vector multiplication with arbitrary N size. The changes can be viewed in matrix-arbVal-double.c file and run via run_sse_arb_double.sh.

We can observe that the speed-up is (as expected) twice as lower (Figure 8), because the level of parallelism decreases tot 2 FP instructions per core (as opposed to 4 for single-precision SSE). The best results are seen for matrix size of aprox 4000.

The last row (marked with red) shows significant performance degrade, when virtual memory is expanded into the disk (exceeding the RAM size because of the matrix size – 8 GB only for the input matrix). It seems that the parallel implementation is heavily impacted by the page switching mechanism between the RAM and the disk, therefore making this particular implementation worse than the sequential code. The algorithm may need to be improved (also using aligned data load would probably help), in order to reduce the overhead created by the memory access.

Double-precision floating-point implementation - arbitrary matrix size

N (matrix / vector size)	Sequential (seconds)	SSE implementation (seconds)	Speed-up
499	0.02475	0.0192	1.2890625
1001	0.011723	0.01016	1.15383858
2002	0.062154	0.059221	1.04952635
3999	0.293791	0.199865	1.46994721
8001	1.226782	0.855995	1.43316491
16002	5.808485	3.982106	1.45864651
32000	254.063376	378.784792	0.67073278

Table 5 Results for in seconds and speed-up comparison for SSE double vs sequential execution for Task 3 (arbitrary matrix size)

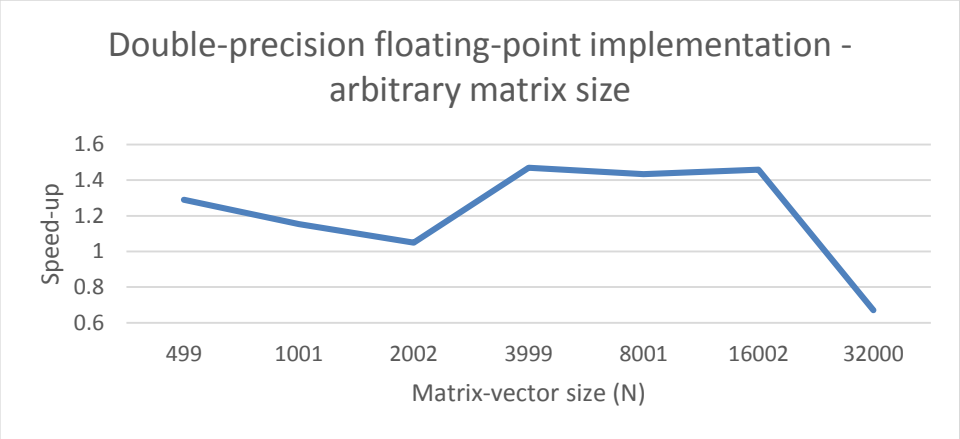

Figure 8. Speed-up sketch for double-precision floating-point SSE operation vs sequential code for different matrix size (arbitrary size)

Part 1b) Task 4

The matrix times matrix implementation has as reference the implementation from Task 2 (vector times matrix – single-precision floating point), for means of computing efficiency, because double-precision would have doubled the compute time that is already $O(N^3)$ complex.

```
$ ./run_sse_matrix.sh 501
compile application
executing the application
SSE enabled execution: 0.428576 (sec)
SEQUENTIAL EXECUTION : 0.718477 (sec)
SEQ: 20570871693312.000000 AV: 20570867499008.000000
wrong at position 0
```

Figure 9 Sample result for matrix-matrix multiplication with N = 501

Interestingly, because the partial sums of the algorithm lead to intermediate results several times bigger than the dynamic range of the single-precision floating point representation, the program shows slight differences between the sequential code and the SSE implementation. This issue comes from the fact that the x87 FPU used in the SSE extension doesn't use any additional guard bits (like the normal FPU in the pipeline which uses 80 bit FP representation for intermediate

```
void matrix_matrix_mult_sse(int size, float *matrix_1,
        float *matrix_2, float *matrix_out){
                int rows, cols;
                int j, k;
                __m128 in1,in2,out;
                float result[4];
                int  rest =  size%4, size_trunc = size - rest;

                for(cols=0; cols<size; cols++)
                        for(rows=0; rows<size; rows++){
                                matrix_out[rows*size + cols] = 0.0;
                                out = _mm_set1_ps(0.0);
                                for(j=0; j<size_trunc; j+=4){
                                        in1 = _mm_loadu_ps(&matrix_1[rows*size+j]);
                                        in2 = _mm_set_ps(matrix_2[cols+(j+3)*size],
matrix_2[cols+(j+2)*size], matrix_2[cols+(j+1)*size],matrix_2[cols+(j)*size]);
                                        out = _mm_add_ps(_mm_mul_ps(in1, in2), out);
                                }
                                if(size!=size_trunc) {
                                        float aux[4] = {0}, aux2[4] = {0};
                                        for(k=0; k<rest; k++) {
                                                aux[k]=matrix_1[rows*size+j];
                                                aux2[k]=matrix_2[cols+(j++)*size];
                                        }
                                        in1 = _mm_load_ps(aux2);
                                        in2 = _mm_set_ps(aux[3], aux[2], aux[1], aux[0]);
                                        out = _mm_add_ps(_mm_mul_ps(in1, in2), out);
                                }
                                _mm_storeu_ps(result, out);
                                matrix_out[rows*size + cols] = result[0]+result[1]+result[2]+result[3];
                        }
}
```

Code snippet 2 SSE matrix-matrix multiplication function for aribitrary sized matrices with single-precision FP element representation

values), which can be confirmed by the fact that the numbers differ on their Least Significant Bit.

The function presented in *Code Snippet 3* is also included in matrix-matrix.c file that can be run via run_sse_matrix.sh.

Part 1b) Task 5

For the purpose of further accelerating the matrix-matrix multiplication (with arbitrary matrix size), we have adapted the code for using the latest instruction set extensions from Intel, implemented in the 4th generation of Core processors (Haswell). The used extensions that came in handy were AVX2 and FMA. By these means, the application would have a theoretical throughput of 32 single-precision FLOP per cycle (8 FLOPs per core), with the biggest improvement coming from the fused-multiply-add block that accelerates the intermediate sums computation.

Improvement of AVX2+FMA over SSE			
N (matrix size)	**SSE implementation (seconds)**	**AVX2 + FMA implementation (seconds)**	**Speed-up**
399	0.232097	0.162034	1.4323969
501	0.477164	0.338361	1.41022163
602	0.861006	0.620024	1.3886656
703	1.392	1.000719	1.39099987
804	2.04744	1.498357	1.36645673

Table 6 Comparison between the SSE implementation and AVX2 + FMA implementation

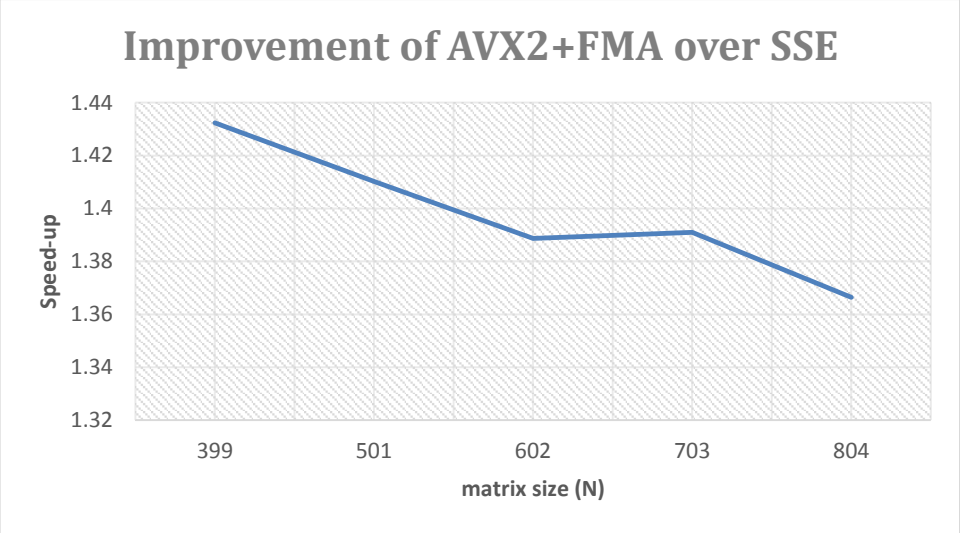

Figure 10 Speed-up of the AVX2+FMA over the SSE implementation for a matrix-matrix multiply application

For assessing the implementation, we have compared it with the SSE code presented in ***Task 4***. The improvements are between 30-40 % (observable in *Figure 10*). We feel that the implementation can be further improved by increasing cache locality and, therefore, offloading the memory lane to the RAM, which seems to be the biggest bottleneck for this data-hungry application. The function is presented in *Code Snippet 4*.

```
void matrix_matrix_mult_avx(int size, float *matrix_1,
        float *matrix_2, float *matrix_out){
                int rows, cols;
                int j, k;
                __m256 in1,in2,out;
                float result[8];
                int  rest =  size%8, size_trunc = size - rest;

                for(cols=0; cols<size; cols++)
                        for(rows=0; rows<size; rows++){
                                matrix_out[rows*size + cols] = 0.0;
                                out = _mm256_set1_ps(0.0);
                                for(j=0; j<size_trunc; j+=8){
                                        in1 = _mm256_loadu_ps(&matrix_1[rows*size+j]);
                                        in2 = _mm256_set_ps(matrix_2[cols+(j+7)*size],
matrix_2[cols+(j+6)*size], matrix_2[cols+(j+5)*size], matrix_2[cols+(j+4)*size],
matrix_2[cols+(j+3)*size],matrix_2[cols+(j+2)*size],matrix_2[cols+(j+1)*size],matrix_2[cols+(j)*size]);
                                        out = _mm256_fmadd_ps(in1, in2, out);
                                }
                                if(size!=size_trunc) {
                                        float aux[8] = {0}, aux2[8] = {0};
                                        for(k=0; k<rest; k++) {
                                                aux[k]=matrix_1[rows*size+j];
                                                aux2[k]=matrix_2[cols+(j++)*size];
                                        }
                                        in1 = _mm256_load_ps(aux2);
                                        in2 = _mm256_set_ps(aux[7], aux[6], aux[5], aux[4], aux[3], aux[2],
aux[1], aux[0]);
                                        out = _mm256_fmadd_ps(in1, in2, out);
                                }
                                _mm256_storeu_ps(result, out);
                                matrix_out[rows*size + cols] = result[0] + result[1] + result[2] + result[3]
+ result[4]+result[5]+result[6]+result[7];
                        }
}
```

Code snippet 3 matrix_matrix_mult_avx function which uses the new vector extensions (AVX2 + FMA)

Conclusion

Due to the direction taken in modern computer architectures, an optimal code now requires the use of parallelization to efficiently use the capabilities provided by the hardware. However, this method requires thorough consideration.

As easy as it is to run C code in different threads through the use of OpenMP, one should also keep in mind the burden that comes included with this commodity. As seen from the results of the assignment, making the most of parallelized code involves a deep understanding of the underlying hardware, such as number of cores, threads and memory organization.

It has been proven that further parallelization improvements can be achieved if SIMD (AVX, SSE) extensions are used. Since the majority of the consumer CPUs used today include SIMD FPU blocks, this optimization can be implemented whenever the application is suited. However, as there is no free lunch, using SIMD comes with its sensitive issues (lower energy efficiency, parallelization overhead, higher memory demand) and it should be considered thoroughly if it should be used in an algorithm.

But not only is broader hardware knowledge needed, but also an understanding of the limitations of parallelization. In the results, when using OpenMP we could also see that the overhead caused by the creation and maintenance of more than one thread could actually have a negative impact on the performance. Similar, SIMD execution has big latency when loading unaligned data and can be even slower than sequential code.

As such, these method of improving performance can be a double-edged sword when thorough care is not given into the details of the underlying hardware and the application itself.