EE4C07 Lab 1

Michiel van der Vlag (4369327), Dirk Vos (4391470), Mark Schrauwen (4419170)

September 27, 2016

1 Introduction

In this report we compare different parallelization techniques for big computational tasks. We analyzed the given code and found that some part(s) could be parallelized. This gives opportunity to increase the performance. We measured the normal execution time of the optimizationable part(s) and compared those to the parallelized execution time.

1.1 Expectations

Assuming ideal executions (e.g. no overhead), we would expect a linear performance improvement calculating bigger matrices. For example, if we do a matrix times vector calculation of a 4-by-4 matrix. We would expect a performance improvement of approximately 4 because the code could multiply the 4 columns times vector in parallel. Of course this can not be scaled indefinitely because there are a finite number of threads. We tested the different parallelization platforms with a matrix range from 1000 to 10.000.

1.1.1 Remarks

The sections in this report reflect the assignments in the lab manual. Some tasks are not presented as a section in this report and are only covered by the code itself.

2 Part A: OpenMP

2.1 Task 1: optimal number of threads, speed-up parallelization

To test the best performance in terms of optimal number of threads we automated a simple test procedure. In a shell script we executed the OpenMP C-code with an arbitrary chosen matrix size of 2000. Every execution was repeated 15 times to get a clear performance pattern. After 15 repetitions the number of threads were incremented to a maximum of 20 threads. The OpenMP C-code was altered in a minimalistic way such that the performance was automatically calculated by dividing the execution time of the *serial execution* by *OMP execution*. A number greater than 1 means that the parallel execution outperforms the serial execution. The optimal number of threads appeared to be 8 which confirms the findings of [1].

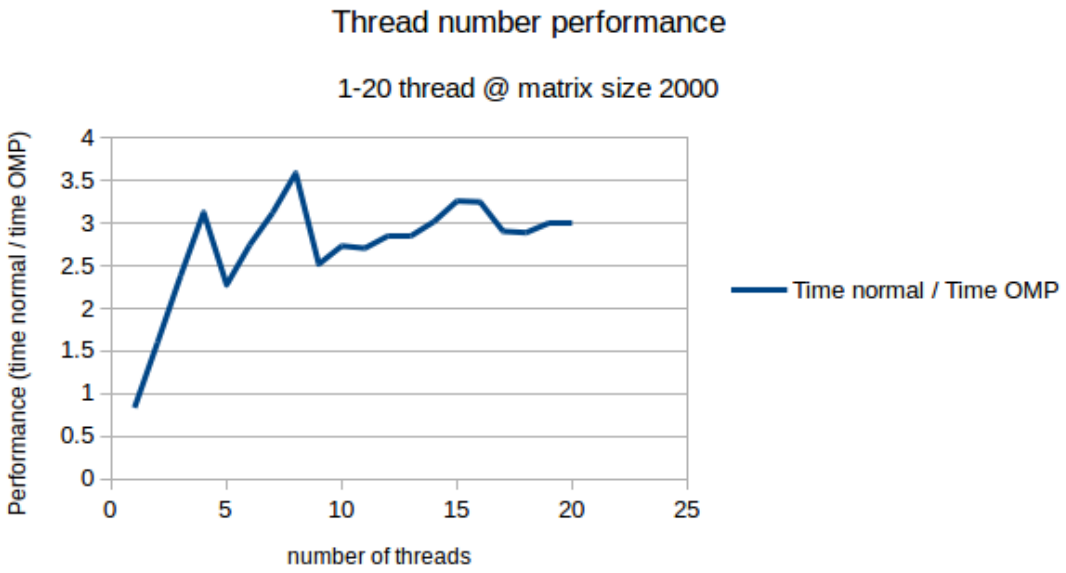

Figure 1: Machine: Intel(R) Xeon(R) CPU W3530 @ 2.80GHz, 8 processors each with 4 cores. We tested the performance by executing the program 15 times. After 15 repetitions the number of threads was incremented. The best performing number of threads is 8. This is the same as the number of cores.

2.2 Task 2: OpenMP Speed-up

After determining the optimal thread number the performance speed-up using the OpenMP library was tested. We limited our maximum matrix size to 10.000 and repeated the performance measurement 3 times in case of high variability. As can be seen in fig. 2 the performance stabilizes with a matrix size of approximately 5000. The OpenMP version has a maximum performance improvement of 3.5. However on average the performance difference is approximately 2.5. This is not the performance improvement we originally expected. These differences are probably due to performance overhead, scheduling and memory swapping. A more course-grained version of the same script as shown in fig. 3 shows that this performance is not stabilized. We suspect the high peeks in performance increase to be due to the peeks in the normal execution (see fig. 4).

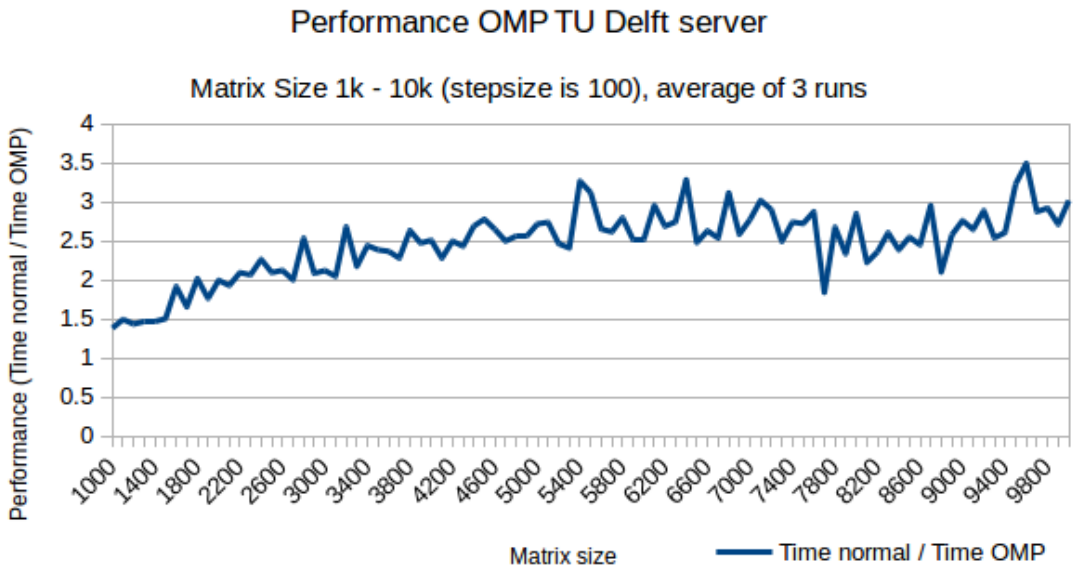

Figure 2: The performance of an increasing matrix size by dividing the normal execution time by the OpenMP execution time.

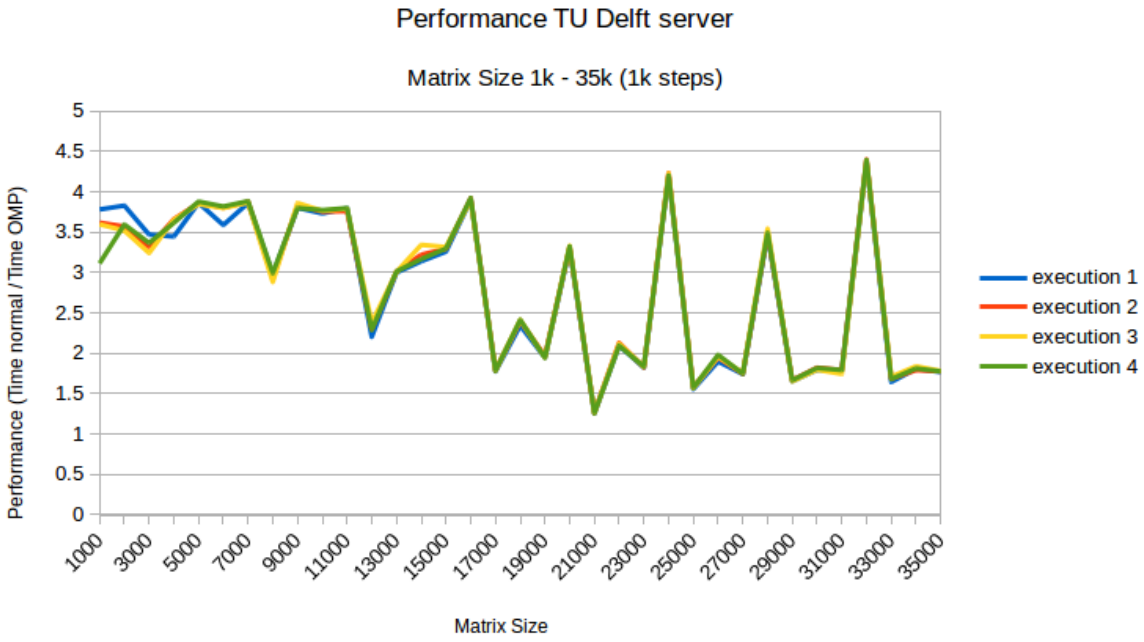

Figure 3: The performance of an increasing matrix size with matrix size steps of 1000 to 35k. The measurements are repeated 4 times.

At first glance the performance characteristic in fig. 2 is expected to be related to a multiple of 4000. But a closer inspection learns that the normal execution time spikes every other 4000th input size (see fig. 4). We

suspect this behavior originates from the memory / cache swapping or a low cache associativity.

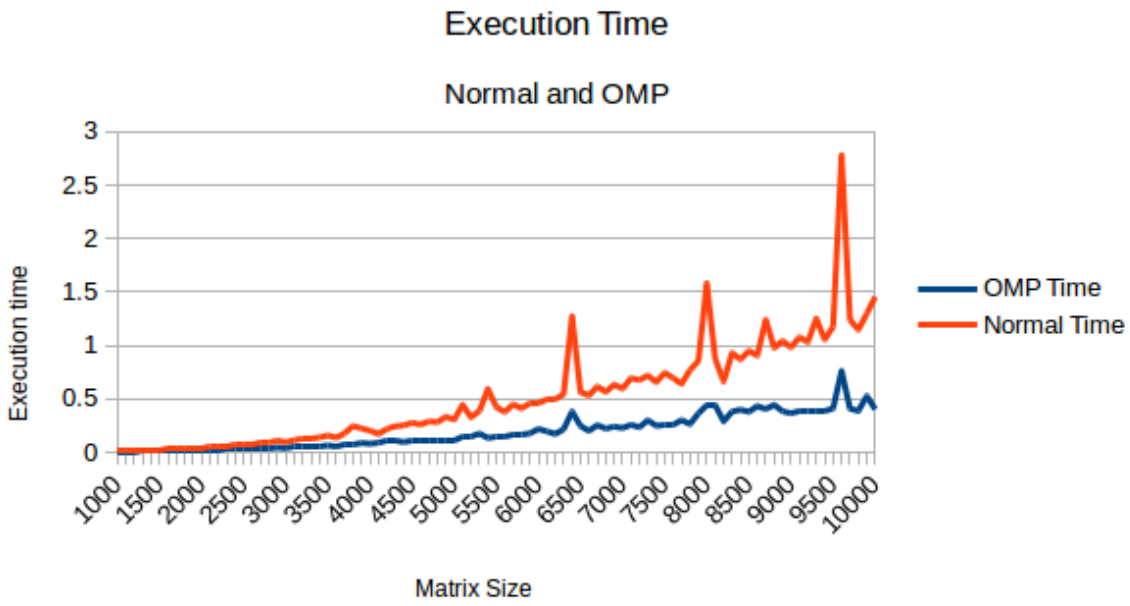

Figure 4: Spiking of the execution time at different Matrix Sizes.

3 Part B: SSE

Streaming SIMD Extensions is an extension to the instruction set which makes it possible to execute the same operations on multiple data. Data is organized in vectors. In this part the speedup is sketched using different and arbitrary matrix sizes and using double precision floating point data type.

3.1 Task 1: Speed-Up - Fixed Size

For task 1 the speed-up of SSE is sketched compared to the normal execution with different input matrix and vector size. The size is varied starting from 1000 with steps of 100 until 10000 is reached.

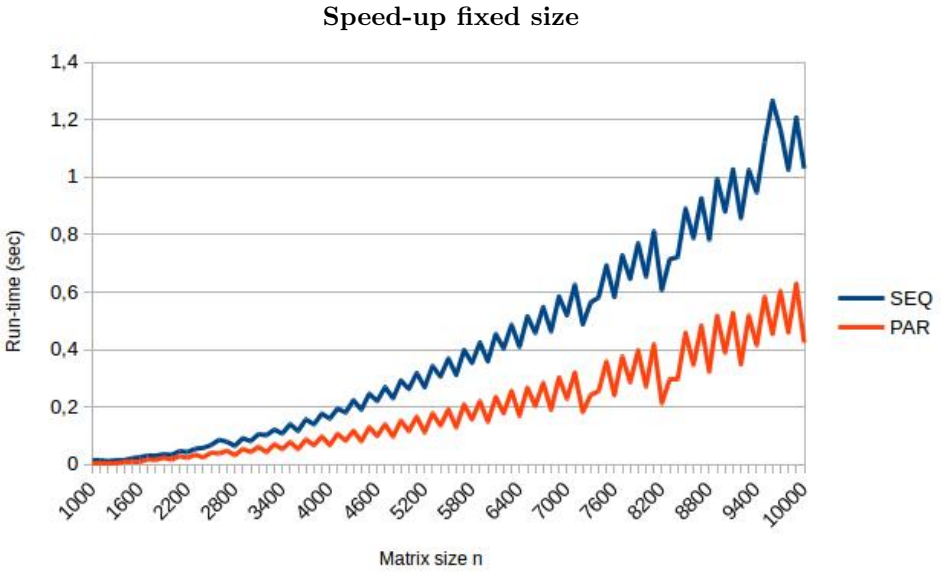

Figure 5: Speed-up for matrix size of multiple of 4

In fig. 5 the speed-up that is reached for above mentioned input sizes is visualized. As matrix sizes become larger the speed-up increases. Eventually, the sequential part needs more time to execute than the parallel part. The term parallel was used here to refer to executing a single instruction on multiple data in parallel (vector).

3.2 Task 2: Speed-Up - Arbitrary Size

The program has been enhanced to calculate arbitrary matrix sizes. Matrices are still squared but any squared size can be entered for calculation with the vector.

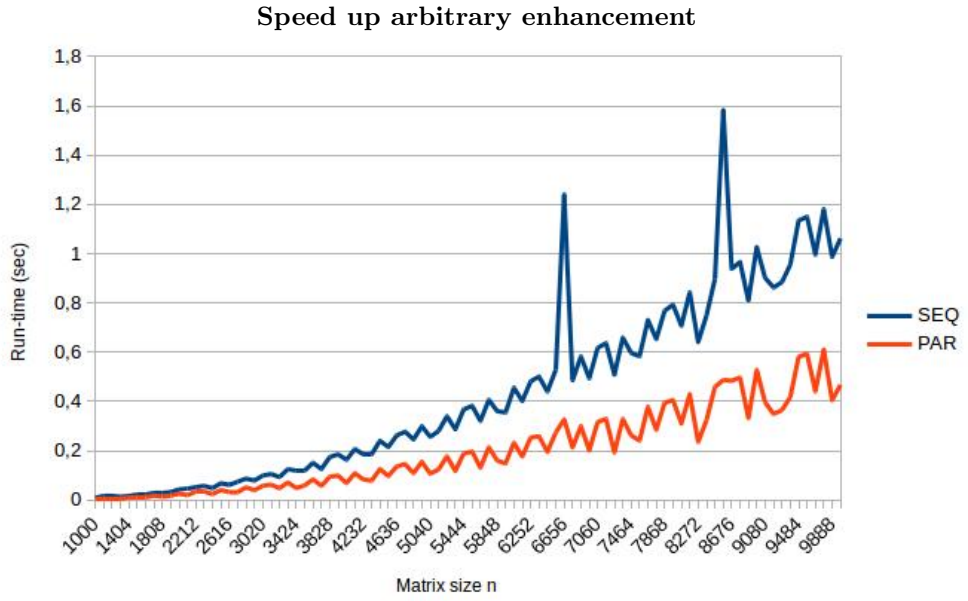

Figure 6: Speed-up for arbitrary matrix sizes

In fig. 6 the speedup is sketched. The arbitrary matrix sizes used, run from 1000 - 10000 with steps of 101 to show that arbitrary matrix sizes can be calculated. Just as with OpenMP some spikes in run-time can be observed.

3.3 Task 3 - Double Precision Floating Point

To enable the double precision for the SSE algorithm the double data type and the corresponding functions for SSE are used. The outer for loop of the SSE matrix times vector algorithm needs to be adjusted to half the size that was taken for the floating point calculation. This is because the double data type only loads 2 elements per execution, since the register space of 128 bits stays the same. So instead of loading 4 elements at the same time, now you can only calculate 2 elements in the same time. To keep things comparable only square matrices are sketched for the speed-up with the same resolution as mentioned in task 1. The expectation is that executing for the same size of the matrices the run-time of the application will double.

```
//Memory allocation and alignment for double
  double *matrix = (double *)memalign(sizeof(double)*2, sizeof(double)*size*size);

//SSE Double datatype declaration
    __m128d a_line, b_line, r_line;

//SSE Functions used
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);
```

The above listed code was used. Memory allocation, data type declaration and the SSE functions used are all tuned for the double data type.

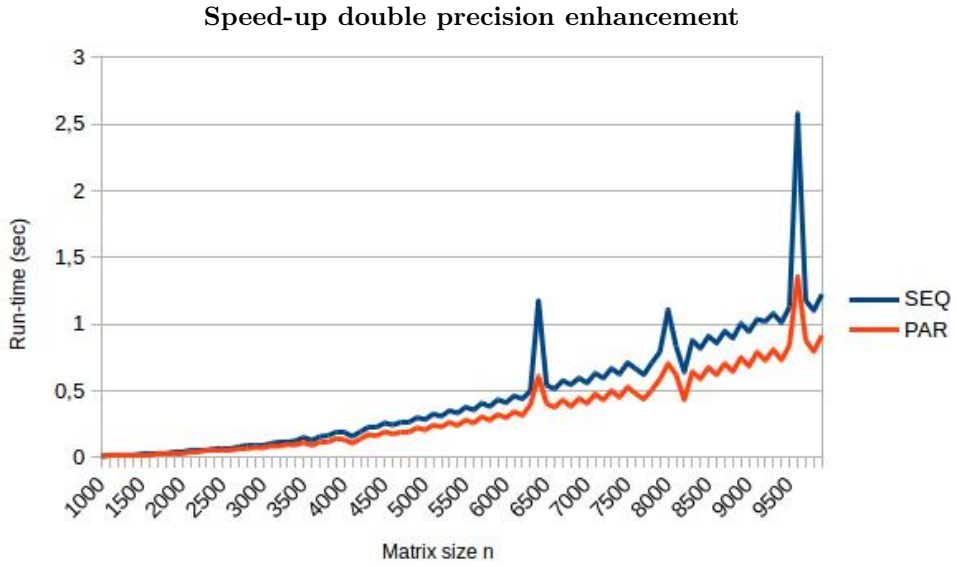

Figure 7: Speed-up double precision data type for SSE

Figure 7 shows the result for double precision SSE implementation versus the regular execution of matrix multiplication. If we compare the graph from this graph to fig. 6 than we obtain the following graph:

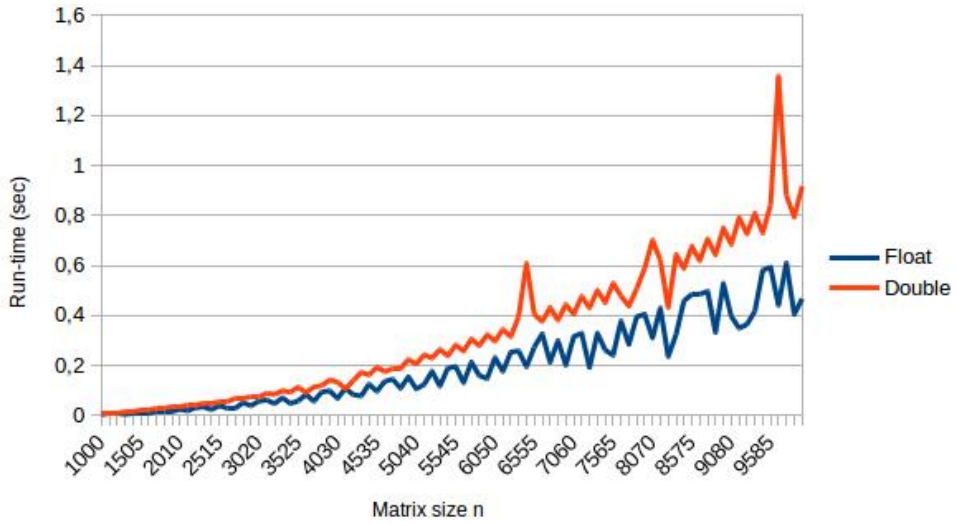

Figure 8: Speed-up double precision vs float data types using SSE

In Figure 8 the expected doubling of the run-time can be observed. The float algorithm is executed in shorter run-time then the double algorithm is.

3.4 Task 5: bonus

In this section we combined OpenMP with SSE. As can be seen from fig. 9 the combination of OpenMP with SSE gives the best execution time. If we relate normal execution time (SER) to other parallelization techniques we get fig. 10. On average the combination of OpenMP and SSE gives a speed-up (compared to the normal execution) of 8.

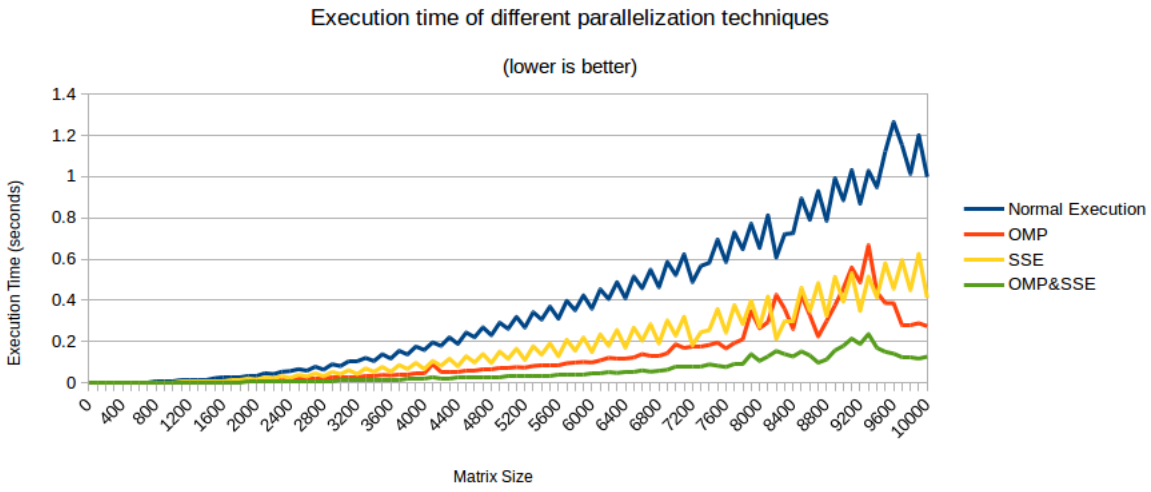

Figure 9: The execution time versus the matrix size of the normal execution, OMP, SSE and OMP + SSE

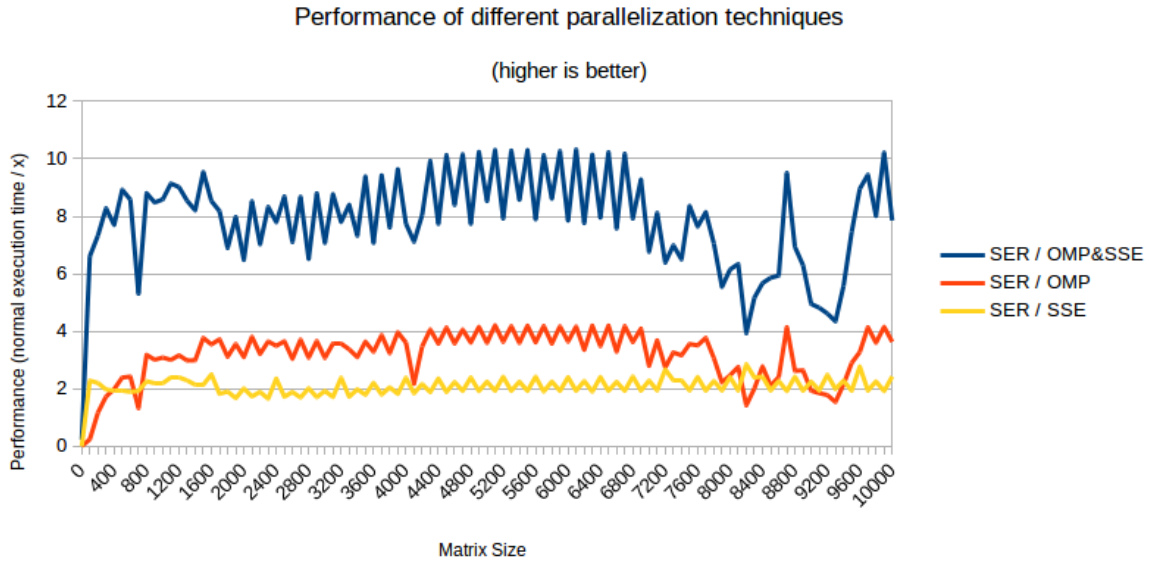

Figure 10: The performance of matrix time vector calculation by implementing OMP and SSE

4 Part C: OpenCL

In order to test the OpenCL functionality, we used the TUDelft server for maximum comparability with the previous tasks. The two most important files are: matrix_cl.c and vectorMatrixMUl.cl. The file matrix_cl.c includes the sequential matrix*vector code and the initialization code for OpenCL device and all the code needed to measure execution time, average multiple runs (a test is done a couple of times and the result is the average of those runs) and iterate over matrix size or local workgroup size. We also altered the example code to move all the malloc functions and not execution related OpenCL code to the start of the main function because it can be done up front and will tamper with execution time measurements otherwise. The vectorMatrixMul.cl file holds the actual OpenCL code, the code that is done in parallel. So if multiple work items are created, then each workitem will contain the code that is in vectorMatrixMul.cl, using global ID's the workitems know which data to use and where to put it.

4.1 Task 1

The first task was researching the execution time for sequential and parallel execution (parallel being on an OpenCL device) for different matrix sizes. To do this a loop was created inside the matrix_cl.c code to iterate over different matrix sizes going from 1000 to 10.000 with corresponding vector sizes, the local workgroupsize was fixed at 100 for now (task 2 will investigate the effect of the workgroupsize further). The measurements are put in the graph below(11), the OpenCL code is the following:

```
__kernel void mul_kernel(global double *vector_in, global double *matrix_in,
        global double *vector_out, int size){
        int id = get_global_id(0);
        double value = 0;
    int k;
    for(k = 0; k < size; k++){
        value += matrix_in[(k* size) + id] * vector_in[k];
    }
    vector_out[id] = value;
}
```

We can conclude from this graph fig. 11 that the parallel execution on the OpenCL device is around 2 times faster the the sequential one.

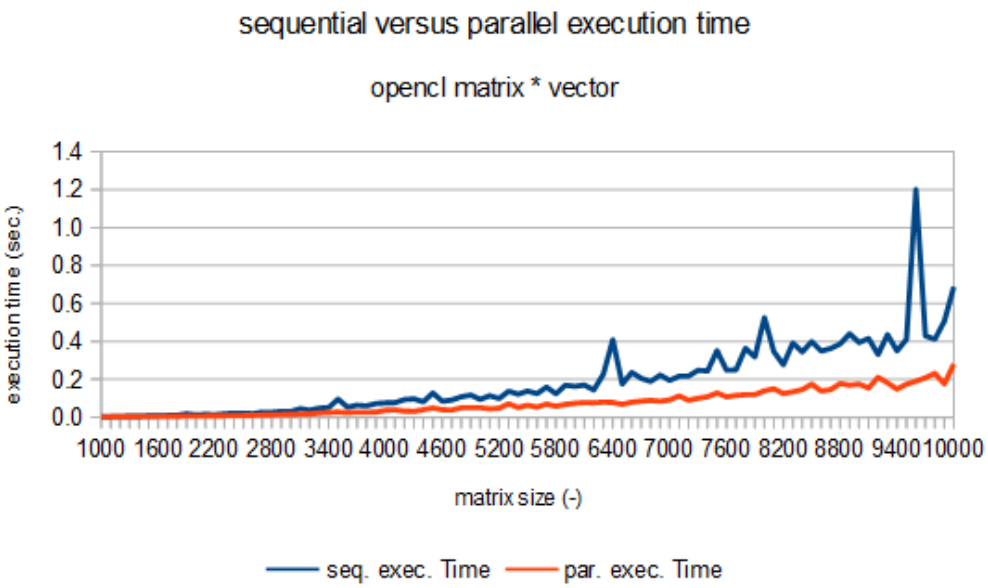

Figure 11: iterating over different matrix sizes, sequential and with OpenCL

4.2 Task 2

The second task was checking execution time for sequential and parallel execution (parallel being on an OpenCL device) for different *local workgroup sizes*. Local workgroup size means how much each computation unit will have to do in parallel. To do this a loop was created inside the matrix_cl.c code to iterate over different local workgroup sizes going from 1 to 1000, after sizes of 1000, OpenCL could no longer divide the work on the connected OpenCL device (the one used on the TU Delft server used in the lab). Only if the matrix size is dividable by the *local workgroup size* then a run is made otherwise the result is the same as before (empty threads are created otherwise, which do not contribute to the final result). The results are shown in the graph below(12), the OpenCL code is the same as the code used in task 1. as can be seen from the graph, the *optimal workgroup size* for matrices of size 3000 x 3000 is around 30. After that the parallel execution time will no longer decrease.

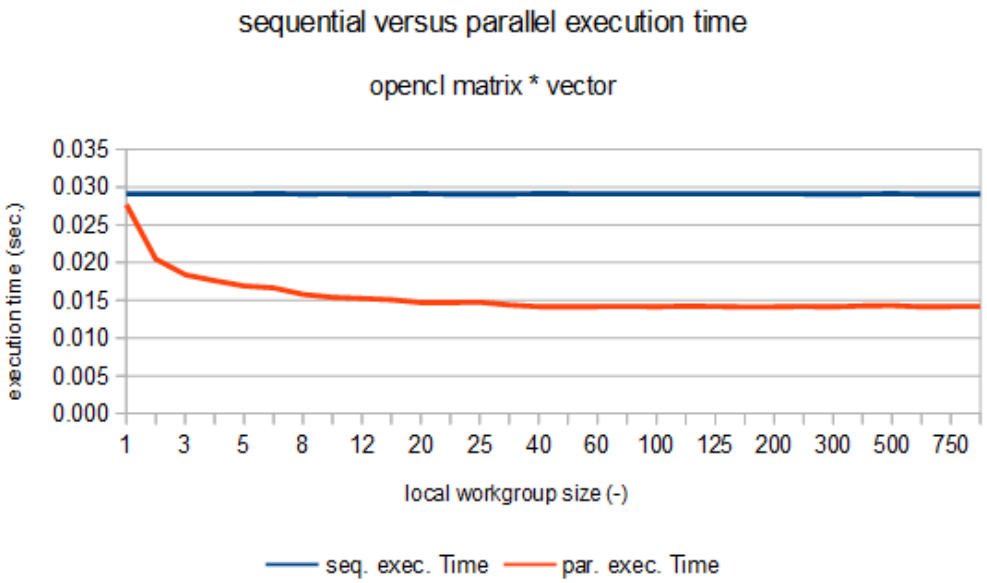

Figure 12: iterating over different local workgroup sizes, sequential and with OpenCL

4.3 Task 3

The third task was to change the original OpenCL code to do a matrix times matrix multiplication which has a lot more calculations. To make it interesting we used the optimal *local workgroup size* as discovered in task 2: size 30. Furthermore we decided to iterate over different matrix sizes to investigate the effects of different sizes. A loop was created inside the matrix_cl.c code to iterate over different matrix sizes going from 120 to 1170 (matrix sizes should be multiples of the local workgroup size). The measurements are put in the graph below(13), the OpenCL code is the following:

```
__kernel void mul_kernel(global double *matrix1_in, global double *matrix2_in,
        global double *matrix_out, int size){
        int rows1 = get_global_id(0);
        int cols = get_global_id(1);
        double value = 0.0;
        int rows2;

        for(rows2=0; rows2<size; rows2++)
                value += matrix1_in[rows1*size + rows2] * matrix2_in[rows2*size + cols];

        matrix_out[rows1*size + cols] = value;
}
```

As can be seen in this code, we used two-dimensional workgroups: two global ID's. This means we eliminated two loops (one iterating over rows and the other over columns). As can be seen from the graph 13, this results in a massive gain in execution speed. The time to execute on the OpenCL device stays nearly the same (around 0.28 seconds at a matrix size of 1170) while the sequential execution grows very fast to even 8 seconds at a matrix size of 1170. This means the parallel execution is almost 30 times faster then the sequential one.

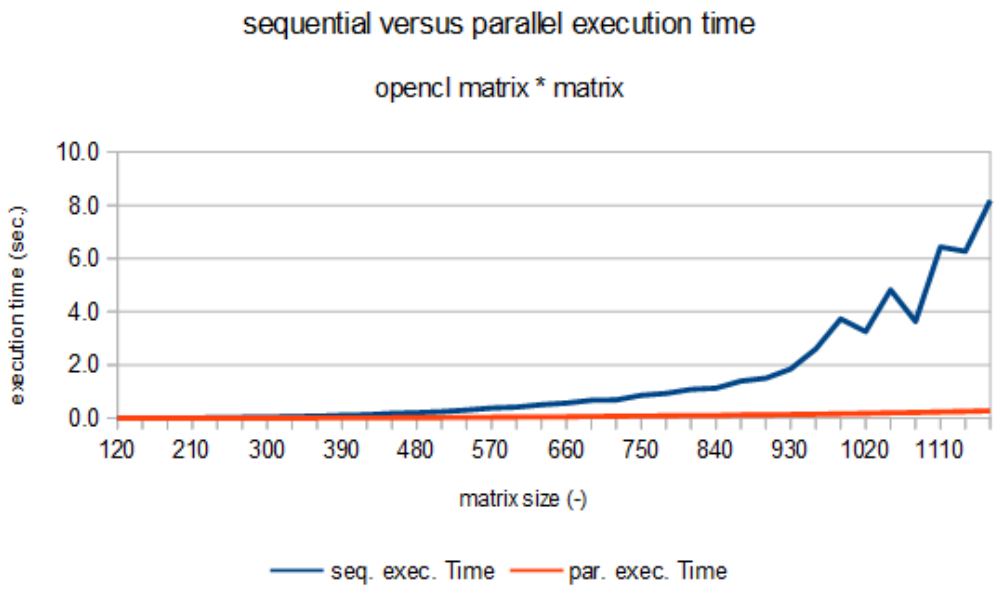

Figure 13: iterating over different matrix sizes, sequential and with OpenCL

5 Comparison

We tried to compare the different parallization platforms. We found that the normal execution time differed per platform. This means that the given enhancement can not be compared correctly. We found that the different compiler optimization flags heavily influenced the normal execution time of the different parallelization platforms. Note that the OpenCL gives the best execution time of the matrix time vector calculation compared to the execution time of OpenMP (OMP) and SSE, but also comes with the most overhead.

6 Conclusions

Concluding from the comparison of different techniques to run code in a parallel way, we can say that OpenMP surely has the benefit of simplicity. Very low overhead but good results if used properly.

OpenCL has a lot of overhead (building OpenCL program on the fly, searching for platforms, searching for OpenCL devices, initializing a communication channel to the OpenCL device and etc.), but OpenCL does bring the highest speedup of the three methods discussed in this report. As seen in the matrix times matrix calculations: OpenCL really pushes performance to its limit by working with more dimensions, which suggests OpenCL is a very good solution for very big computational problems.

SSE is kind of a trade off between OpenMP and OpenCL, it has a little bit of overhead (dealing with vector sizes which are bigger then 1x1) but it gives a nice speedup. Potentially, when using an automatic vectorizing tool, SSE could be a very good solution for medium sized computational problems.

References

[1] Jie Shen, Ana Lucia Varbanescu *A Detailed Performance Analysis of the OpenMP Rodinia Benchmark.* Delft University of Technology Parallel and Distributed Systems Report Series