CPD - Matrix Multiplication

Problem description and algorithms explanation

1. Naive matrix multiplication

The first algorithm implemented is the naive matrix multiplication algorithm. In it, each matrix entry is a sum of products (n is the matrix size and we assume that both matrices being multiplied are of size n x n).

- Number of arithmetic operations: $2n^3$
- Time complexity: $O(n^3)$

```
-   for(i=0; i<m_ar; i++)
-       {   for( j=0; j<m_br; j++)
-           {   temp = 0;
-               for( k=0; k<m_ar; k++)
-               {
-                   temp += pha[i*m_ar+k] * phb[k*m_br+j];
-               }
-               phc[i*m_ar+j]=temp;
-           }
-       }
```

This algorithm looks so natural and trivial that it is very hard to imagine a better way to multiply matrices. However, there are more efficient algorithms for matrix multiplication than the naive approach, which we implemented after.

2. Line multiplication

Next up, we implemented the line multiplication algorithm.

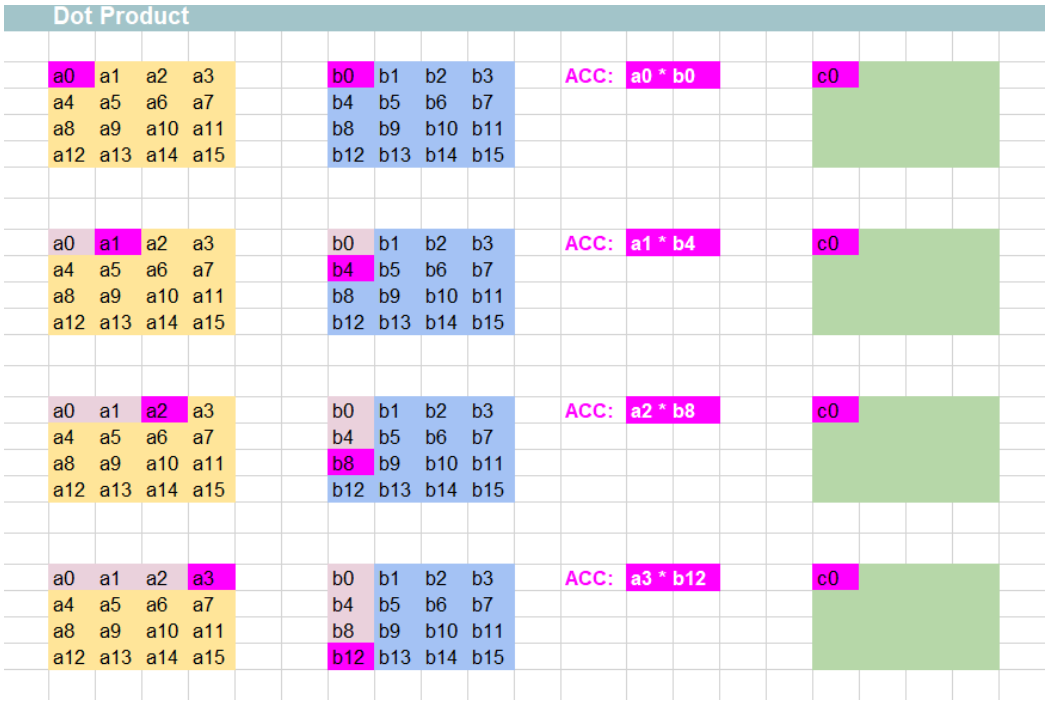
- Number of arithmetic operations: $2n^3$
- Time complexity: $O(n^3)$

```
for(int i = 0; i < m_ar; i++){
        for(int j = 0; j < m_br; j++){
            for(int k = 0; k < m_br; k++){
                // c[i][k] = c[i][k] + ( a[i][j] * b[j][k] );
                c[i * m_ar + k] = c[i * m_ar + k] + (a[i*m_ar + j] * b[j * m_ar +
k]);
            }
        }
    }
```

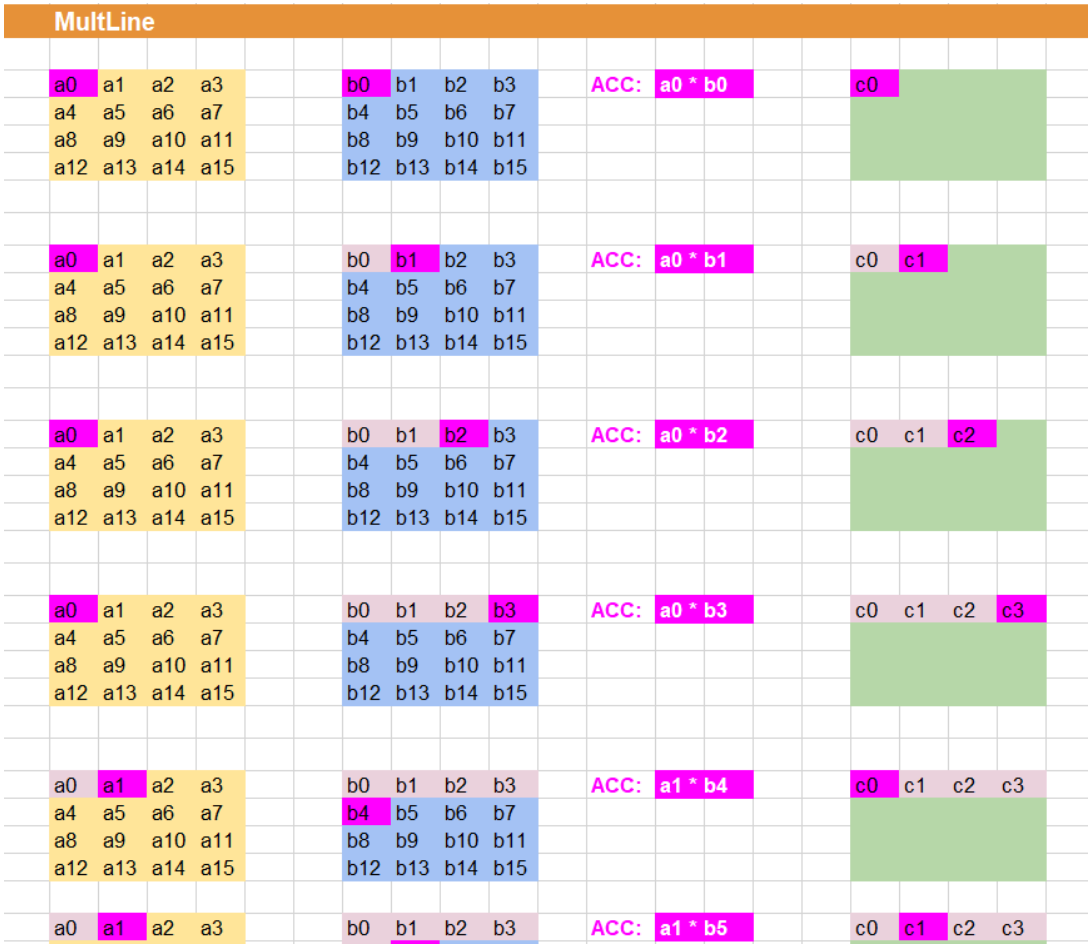
We're still going over every single matrix element three times, so its time complexity is still $O(n^3)$, but this time we initiate the resulting n x n matrix with a 0 in every position and we go on adding and accumulating the values on the respective element of the result matrix.

This approach seems very similar to the one used in the naive algorithm, however as the performance results show, this algorithm heavily outperforms the naive implementation.

Below is the way the naive algorithm goes about calculating each element of the result matrix:



As is visible, it favours calculating each element of the resulting matrix one at a time. Meanwhile, the line multiplication algorithm iterates through all 3 matrices as follows:



In the line multiplication algorithm this way of iterating through the matrices guarantees that read elements are always adjacent elements, which makes the chance of the element already being in cache much greater, when compared to the previous naive implementation. This is reflected in greater cache hit rates in the performance tables, which explains why, even though these algorithms share the same time complexity ($O(n^3)$ as they are both composed of three nested for loops) the line multiplication algorithm runs faster.

Basically, memory access patterns that read from sequential locations in memory can go an order of magnitude faster than naive access patterns due to the fact that sequential accesses can read directly from elements in the cache.

3. Block Matrix Multiplication

Finally, we implemented the block matrix multiplication algorithm.

- Number of arithmetic operations: $2n^3$
- Time complexity: $O(n\_blocks * n\_blocks * n\_blocks * bkSize * bkSize * bkSize) = O(n^3)$

```
for(int ii = 0; ii < m_ar; ii+=bkSize){
        for(int jj=0; jj< m_ar ; jj+= bkSize){
            for(int kk=0; kk < m_ar; kk+= bkSize){
                for(int i=ii; i< ii + bkSize; i++){
                    for(int j = jj; j < jj+bkSize; j++){
                        for(int k = kk; k<  kk + bkSize ; k++){
                            c[i * m_ar + k]+= a[i*m_ar + j] * b[j * m_ar + k];
                        }

                    }
                }
            }
        }
    }
```

How it works:

- First matrices A and B need to be partitioned in the following way:

$$A = \left(\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right) \Longrightarrow \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$$A_{11} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, A_{12} = \begin{pmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{pmatrix}$$

$$A_{21} = \begin{pmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix}, A_{22} = \begin{pmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{pmatrix}$$

The main goal of partitioning the matrices in this way is built upon the concept of cache coherence described above. Basically, when we use divide-and-conquer in this way, to subdivide the problem of matrix multiplication by multiplying smaller matrices each time, we are loading the smaller matrix into cache where it is likely to fit in its entirety due to its smaller size. By making calculations on these smaller matrices, instead of on the whole matrices at once, we are actively maximizing cache hits which should, in theory, speed up the calculations quite a bit.

However, for this to work all three blocks from A, B and C must fit in cache, so we cannot make these blocks arbitrarily large.

So theoretically, what should happen is an increase in performance, as the block size increases, up to a certain block size, after which, due to the block no longer fitting in cache, the block partition matrix multiplication algorithm should see a decrease in performance.

One possible improvement to this algorithm would be to take advantage of multithreading to process multiple block partitions at the same time.

Performance metrics

To evaluate the performance of the implemented algorithms we took the values of the following metrics:

- In both the **python** and **c++** versions of the program we measured the Processing Time and the number of flops.
    - Processing time refers to the time that the program takes to process a matrix multiplication in the chosen algorithm as well as the
    - Flops refer to the number of floating-point operations made by the program in the time taken to make them. These are calculated according to the formula: $\frac{number\ of\ floating\ point\ OPs}{processing\ Time}$ which for the tree algorithms corresponds to $\frac{2n^3}{processingTime}$.
- In **c++** only with the help of papi to make measurements we calculated the L1 cache hit rate and L2 cache hit rate as well as the Data Prefetch Misses.
    - The L1 cache and L2 cache hit rates refer to the probability of when information is requested from a cache the cache being able to fulfill that request making the program faster since it does not need to access the disk to obtain the information needed making a cache hit rate between 95% and 99% ideal.
    - Rather than waiting for a cache miss to initiate a memory fetch, data prefetching anticipates such misses and issues a fetch to the memory system in advance of the actual memory reference. These operations are useful in order to reduce cache misses.

Results and analysis

We ran the following tests on a computer with the following specs:

- Processor: Intel® Core™ i7-9700 CPU @ 3.00 GHz com 8 cores
- RAM: 16 gb
- Cache size: 12 MB

Furthermore, the number of flops was taken in Mflops for easier visualization since a big number of flops are made per second by all the algorithms.

Results

Matrix Size	Time	Mflops	L1 DCM	L2 DCM	DPM	L1 HitRate	L2 HitRate
600	0.188	2299.14	244770103	41046671	11997198	0.434853	0.905228
1000	1.249	1600.89	1224513219	266557943	77361283	0.38867	0.866923
1400	3.533	1553.34	3487821009	1261135665	369931584	0.365148	0.770448
1800	18.821	619.728	9059568435	7741139669	2800763141	0.223937	0.336877
2200	38.849	548.176	17654260823	23017575412	8583367968	0.171572	-0.0801
2600	70.372	499.52	30881160136	50576622227	19270188636	0.122004	-0.43797
3000	120.325	448.785	50319436725	96561290999	36705309860	0.068625	-0.78728

Table 1 - Performance results of the matrix multiplication algorithm in c++

Matriz Size	Time	Mflops	L1 DCM	L2 DCM	DPM	L1 hit rate	L2 hit rate
600	0.099	4354.84	27109126	58188143	35501377	0.95826	0.910407
1000	0.493	4060.87	125770091	261135274	162543375	0.958133	0.913072
1400	1.537	3570.12	346234991	699195496	447528117	0.957981	0.915145
1800	3.754	3106.81	745986574	1422409319	981669909	0.957394	0.918761
2200	6.234	3416.33	2075346547	2594475727	1790790333	0.935071	0.91883
2600	10.413	3375.71	4413604010	4207079803	2971888023	0.916338	0.920253
3000	16.01	3372.95	6781560353	6402908989	4569134552	0.916314	0.920987
4096	41.359	3323.1	17547772388	16288949276	11670739563	0.91491	0.921014
6144	138.141	3357.84	59125004844	53396311205	39870892431	0.915042	0.923274
8192	331.256	3319.22	1.40123E+11	1.27236E+11	94233107628	0.915053	0.922866
10240	656.883	3269.2	2.73778E+11	2.54808E+11	1.84837E+11	0.915019	0.920907

Table 2 - Performance results of the line multiplication algorithm in c++

Matriz Size	Time	Mflops	L1 DCM	L2 DCM	DPM	L1 HitRate	L2 HitRate
4096	32.866	4181.79	9514937939	32295064861	17052923875	0.953982	0.843809
6144	107.055	4332.89	32626020287	1.10995E+11	58237190351	0.953243	0.840932
8192	257.174	4275.36	75569778734	2.56902E+11	1.35164E+11	0.954309	0.844672
10240	499.075	4302.93	1.51086E+11	5.08486E+11	2.69772E+11	0.953228	0.842587

Table 3a - Performance results of the block multiplication algorithm with blocks of 128 in c++

Matriz Size	Time	Mflops	L1 DCM	L2 DCM	DPM	L1 HitRate	L2 HitRate
4096	27.524	4993.48	9093214405	23124950295	13471834097	0.955961	0.888005
6144	93.219	4975.99	30613979972	77852598571	45524929699	0.956066	0.888275
8192	415.6	2645.6	72083984532	1.66106E+11	1.08771E+11	0.956357	0.899431
10240	422.044	5088.29	1.41974E+11	3.57393E+11	2.10363E+11	0.955988	0.889208

Table 3b - Performance results of the block multiplication algorithm with blocks of 256 in c++

Matriz Size	Time	Mflops	L1 DCM	L2 DCM	DPM	L1 hit rate	L2 hit rate
4096	37.519	3663.14	8809107058	19260003701	11940059220	0.957309	0.906661
6144	92.716	5002.98	29679340962	66249893399	40371210498	0.957379	0.904862
8192	344.197	3194.42	70737650974	1.46826E+11	98122163056	0.957143	0.911044
10240	429.871	4995.64	1.37234E+11	3.11477E+11	1.87182E+11	0.957429	0.903378

Table 3c - Performance results of the block multiplication algorithm with blocks of 512 in c++

Matriz Size	Time	Mflops
600	28.61143	15.09885
1000	140.0721	14.27836
1400	391.5382	14.01651
1800	836.5472	13.94303
2200	1547.437	13.76211
2600	2578.061	13.63506
3000	4046.006	13.3465

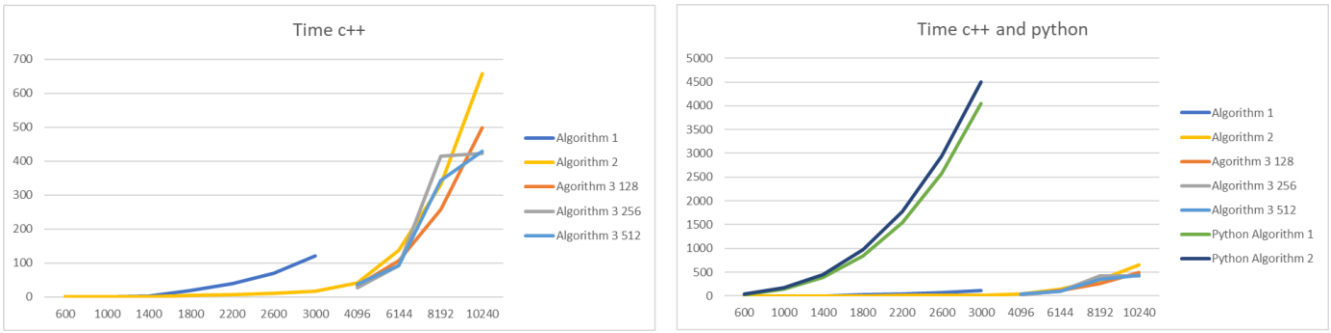
Table 4a - Performance results of the matrix multiplication algorithm in Python

Matriz Size	Time	Mflops
600	35.20787	12.26998
1000	166.9513	11.97954
1400	455.0101	12.06127
1800	968.1767	12.04739
2200	1775.698	11.99303
2600	2937.003	11.96866
3000	4497.744	12.00602

Table 4b - Performance results of the line multiplication algorithm in Python

Analysis

Firstly, across all c++ algorithms, we see a proportional increase of data prefetch misses with matrix size, simply because we are accessing more data, and therefore the amount of data prefetching being done is greater, so the number of misses also goes up.



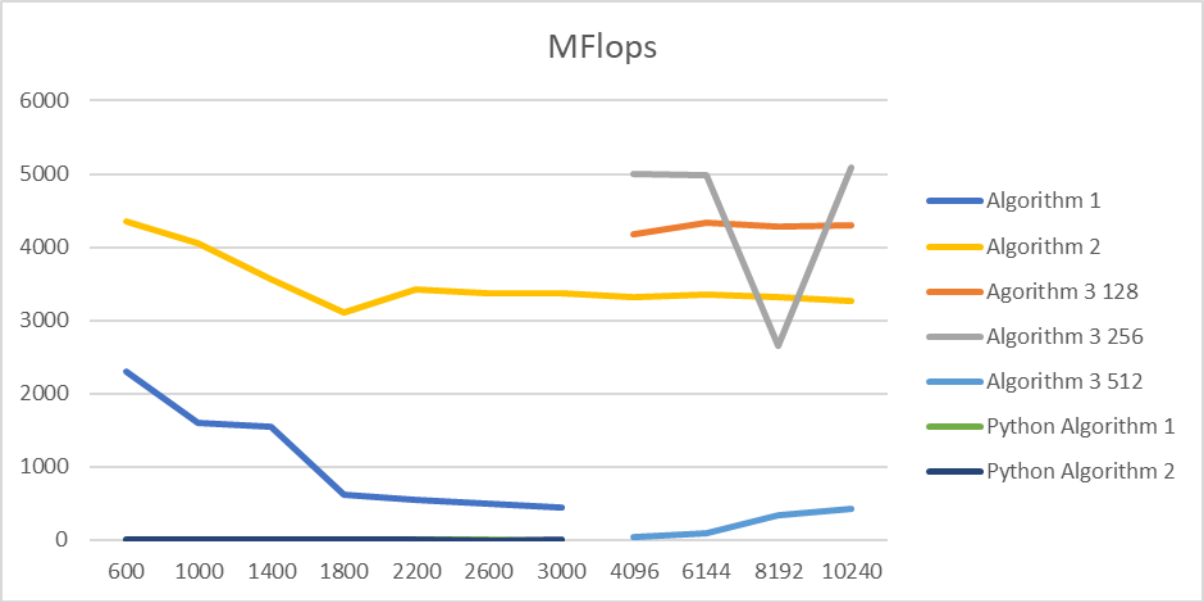
Graph 1 – Graphs of the time taken by the algorithms by the size of the matrix.

With Graph 1 made from the tables above we can see, as expected, that python's version of the algorithms take much longer than their c++ counterparts which is mainly explained by the fact that Python is a higher level language than c++, which means it abstracts certain aspects like memory management, pointers, …. Adding to this, Python is an interpreted language which means that code executes slower as code must be interpreted at runtime instead of being compiled to machine code at compile time making it much slower than c++.

We also verified that in Python, the algorithm of line multiplication (**Algorithm 2**) had a slower performance relative to the naive implementation. Asking our colleagues that had chosen Python as their second language revealed that they were facing the exact same dilemma. This might be once again explained by python's unorthodox memory management rendering out attempt to optimize the use of the cache memory useless.

In general, our assumptions from the algorithm description sections were true and the block multiplication algorithm (**Algorithm 3**) has a better performance than the line multiplication algorithm (**Algorithm 2**) which has an even better performance than our naïve multiplication algorithm (**Algorithm 1**).

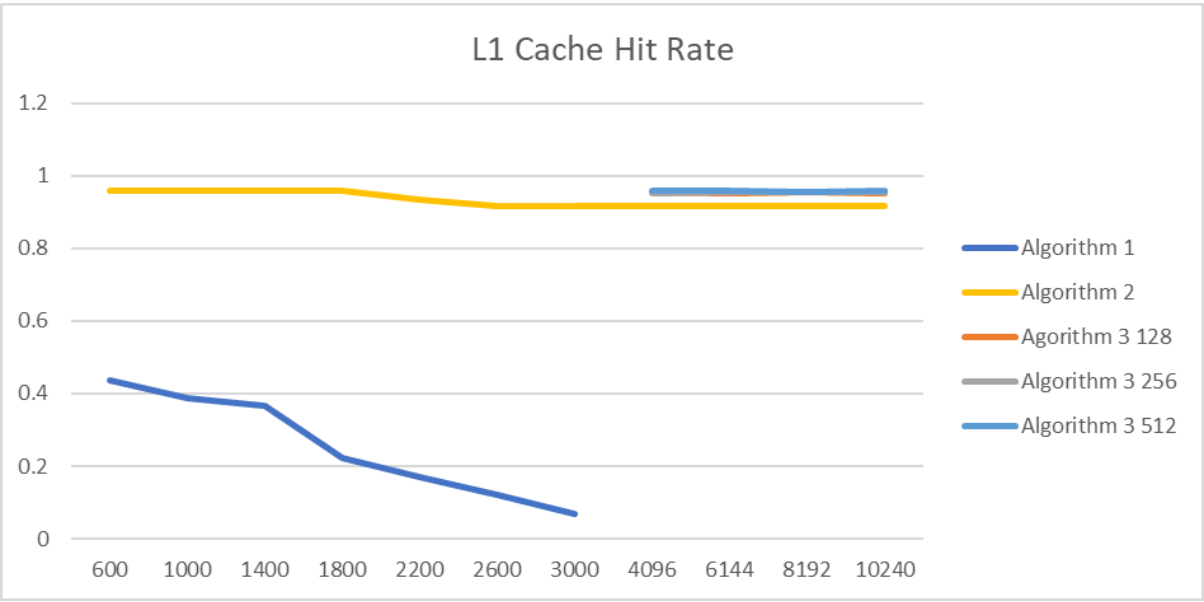
We can also see the efficiency of the block multiplication algorithm increasing until a certain point with the increase of the size of the blocks before stagnating.



Graph 2 – Graph of the number of Mflops per matrix size.

In general, we see a decrease in Mflops as the matrix size increases because our cache can only hold so many matrix elements, thus, as the matrix size gets bigger, the chances of our algorithm finding the element it's looking for in fast memory decreases.

We can also see that the line matrix multiplication (**Algorithm 2**) has more FLOPs for the same matrix size which means it is able to do the same number of floating-point operations in a shorter amount of time. This is because it maximizes cache hits reading sequential elements from the matrices thus being able to read values from the matrix at faster rates.



Graph 3 – Graph of the L1 Cache Hit Rate per matrix size.

As was assumed previously, the L1 Cache Hit Rate is improved using the line multiplication matrix (**Algorithm 2**) and the block multiplication matrix (**Algorithm 3**) since the accessing of adjacent elements makes it more likely for the needed elements to already be in cache.

Conclusions

This practical assignment has made us realize the importance of having good memory management in our algorithms and not only good time complexity, as we were faced with three algorithms possessing the same time complexity but with very different processing times.

We conclude that locality of data is at least as important as computation be it temporal, as in the re-use of data recently used, or spatial, as in using data nearby that was recently used.

Furthermore, the use of python as our second programming language in this project has helped us understand better the advantage of lower-level languages when it comes to optimization since in python, the abstraction of memory management rendered the two algorithms used very similar in terms of performance.

Group 3:

- Miguel Norberto Costa Freitas (up201906159@edu.fe.up.pt)
- Joana Teixeira Mesquita (up201907878@edu.fe.up.pt)
- Diogo Miguel Chaves dos Santos Antunes Pereira (up201906422@edu.fe.up.pt)