Program 1.A

Student: Anthony Bustamante

Part 0

Write a report including the following:

A. Research BLAS and LAPACK, write a report on what they are and why they are important in HPC.

A.1. What is BLAS?

Basic Linear Algebra Subprograms (BLAS) is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. BLAS implementations take advantage of special floating-point hardware such as vector registers or SIMD instructions [1][2].

A.2. What is LAPACK?

LAPACK ("Linear Algebra Package") is a standard software library for numerical linear algebra. It provides routines for solving systems of linear equations and linear least squares, eigenvalue problems, and singular value decomposition. It also includes routines to implement the associated matrix factorizations such as LU, QR, Cholesky and Schur decomposition. The routines handle both real and complex matrices in both single and double precision. LAPACK relies on an underlying BLAS implementation to provide efficient and portable computational building blocks for its routines[3][4].

A.3. Why are BLAS and LAPACK important for HPC?

Because the BLAS and LAPACK are efficient, portable, and widely available, they are commonly used in the development of high-quality software.
Most of the software programs (if not all) are based on algebra and BLAS and LAPACK help us to get the best performances from them.

B. Investigate the APIs needed to perform vector addition, vector-matrix multiplication and matrix-matrix multiplication. In your own words explain how they work.

To do the operations in mention we can use APIs/functions like <cblas.h> or <mkl.h>. And, according to what I have seen they use Row-major order or Column-major order to optimize the operations between vectors, matrices and matrix-vector [7].

C. Research MKL and explain its difference with BLAS and LAPACK.

C.1. What is MKL?

Intel oneAPI Math Kernel Library (Intel oneMKL; formerly Intel Math Kernel Library or Intel MKL) is a library of optimized math routines for science, engineering, and financial applications. Core math functions include BLAS, LAPACK, ScaLAPACK, sparse solvers, fast Fourier transforms, and vector math[5].

C.2. What is the difference between MKL and BLAS and LAPACK?

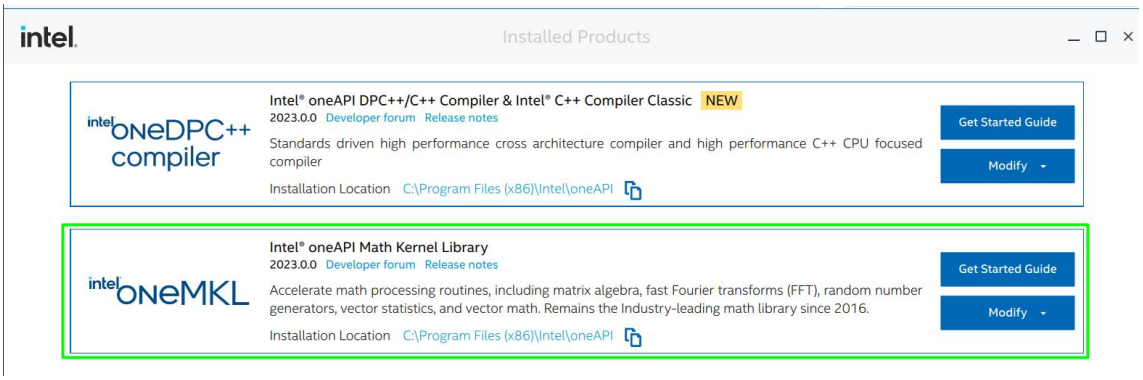
The Intel MKL libraries contain a variety of optimized numerical libraries including BLAS, LAPACK, and some others. The difference is that MKL offers a bigger variety of functions [5].

Also, all the functions in MKL have been optimized for Intel processors, whereas BLAS and LAPACK are also available for other processors [6].

Part 1: Install BLAS or MKL

Download and install the BLAS distribution specific for your operating system. Alternatively, see if you can use Intel's Math Kernel Library.

For this program I worked with MKL for Intel.



Part 2: Write a sequential CPU implementation of the following:

The sequential multiplication code is with name: Sequential-Matrix-Vector-Multiplication.cpp

Note: All the implementation has been done for double precision.

1. Process:

A.1. Coding function for the generation of pseudo-random matrices and vectors with own algorithm, with size equals to M, A[M][M]xB[M]=C[M].

```
double random(double i, double j) {
   return ((double(3.5)*j)/double(2.14253) * i);
}
```

A.2. Coding function for calculating the time it takes for the matrix-vector multiplication, it was done with <chrono> library.

```
auto get_time() {
    return std::chrono::high_resolution_clock::now();
}
```

A.3. Coding function for calculating the matrix-vector multiplication:

```
double sequencial_multiplication(double matA[MAX][MAX],double matB[MAX]){
    for (int i = 0; i < MAX; i++)
      for (int j = 0; j < MAX; j++)
```

```
        matC[i] += matA[i][j] * matB[j];
return 0;
}
```

A.4. Showing the outputs, result of the multiplication, C[M] and/or elapsed time for the multiplication:
a. For the matrices generated: E.g. for a A[5][5] and B[5].

```
Matrix A:
0.0450869 0.241117 0.437147 0.633177 0.829207
0.420811 2.25042 4.08004 5.90965 7.73926
0.796535 4.25973 7.72293 11.1861 14.6493
1.17226 6.26904 11.3658 16.4626 21.5594
1.54798 8.27834 15.0087 21.7391 28.4694

Matrix B:
0.212251
0.560205
0.908158
1.25611
1.60406
```

b. For the result of the multiplication C[5]:

```
Result of the multiplication:
2.66708
24.8928
47.1185
69.3442
91.5699
```

c. For the elapsed time of the multiplication:

```
Execution time
**************
Elapsed time for sequential code: 2.76e-05 s
```

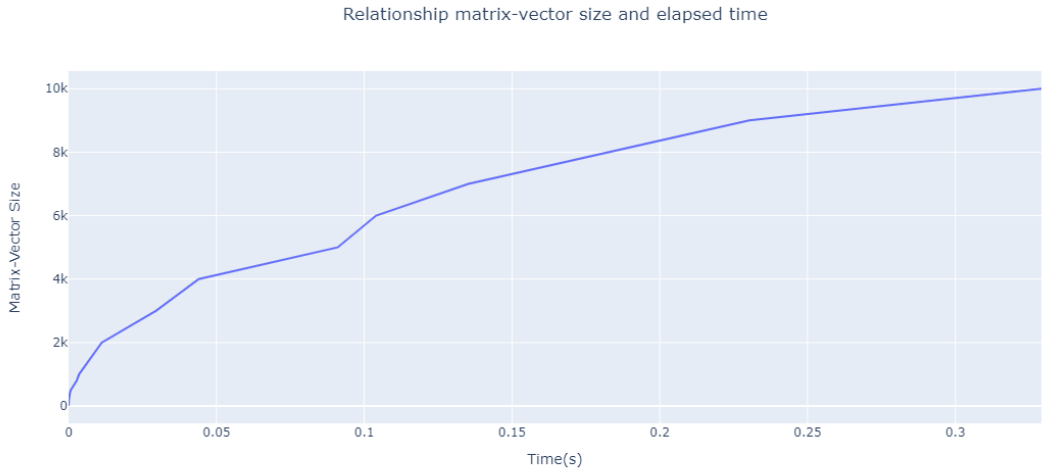
d. For execution rate (operations per second), size=10:

```
double FLOPS=(MAX*MAX)/mkl_time_span.count();
        cout<<"Execution rate (operations per second): "<<FLOPS<<endl;
```

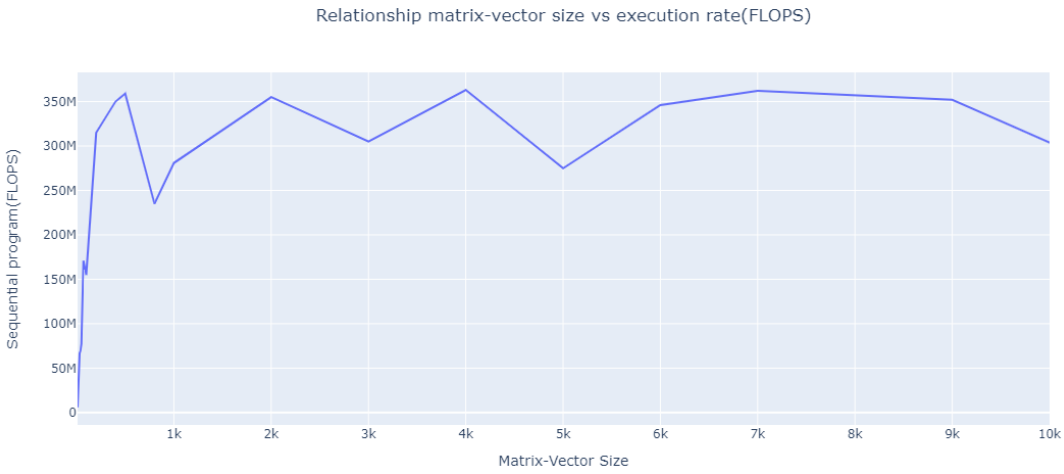
```
Execution rate (operations per second):
143204.926249
```

2. Results:

Relationship between matrix-vector size and elapsed time:



Relationship between matrix-vector size and execution rate (operations per second):



Part 3: Write a parallel CPU implementation using POSIX or Windows threads of the following:

The parallel multiplication code is with name: Sequential-Matrix-Vector-Multiplication.cpp

1. Process:

A.1. Coding function for the generation of pseudo-random matrices and vectors with own algorithm, with size equals to M, A[M][M]xB[M]=C[M]. The code is the same as for part 2.

```
double random(double i, double j) {
   return ((double(3.5)*j)/double(2.14253) * i);
}
```

A.2. Coding function for calculating the time it takes for the matrix-vector multiplication, it was done with <chrono> library. The code is the same as for part 2.

```
auto get_time() {
    return std::chrono::high_resolution_clock::now();
}
```

A.3. Coding function for calculating the matrix-vector multiplication with threads:

a. Threading part: we used a threat for each row in the matrix A that then computes the multiplication with the vector B, which, in turn it is a vector-vector multiplication for each thread. We did not use a static number of threads because in my computer the static number of threads was not working for all the sizes of the matrices from 10 to 10 000.

```
for(i = 0; i < n; i++){
        data[i] = (struct v *)malloc(n * sizeof(struct v));
        for(k = 0; k < n; k++){
            data[i][k].i = i;
            data[i][k].j = k;}
        pthread_create(&threads[i], NULL, multiplication, data[i]);
    }
```

b. Multiplication function:

```
static void * multiplication(void *arg){
    struct v *data = (struct v *)arg;

    size_t l;
    for(l=0; l < N; l++){
        size_t i=(data[l]).i;
        size_t j=(data[l]).j;
        double sum=0;
        size_t d;
        for (d = 0; d < N; d++){
            sum = sum + A[i][d]*B[d];}
        C[i] = sum;
        sum = 0;}
    return 0;}
```

A.4. Showing the outputs, result of the multiplication, C[M] and/or elapsed time for the multiplication:

a. For the matrices generated: E.g. for a A[5][5] and B[5]. The Matrices are the same in part2.

```
Matrix A:
0.0450869 0.241117 0.437147 0.633177 0.829207
0.420811 2.25042 4.08004 5.90965 7.73926
0.796535 4.25973 7.72293 11.1861 14.6493
1.17226 6.26904 11.3658 16.4626 21.5594
1.54798 8.27834 15.0087 21.7391 28.4694

Matrix B:
0.212251
0.560205
0.908158
1.25611
1.60406
```

b. For the result of the multiplication C[5]: The result differs a little bit.

```
The result of the multiplication is:
2.667083
24.892775
47.118467
69.344158
91.569850
```

c. For the elapsed time of the multiplication:

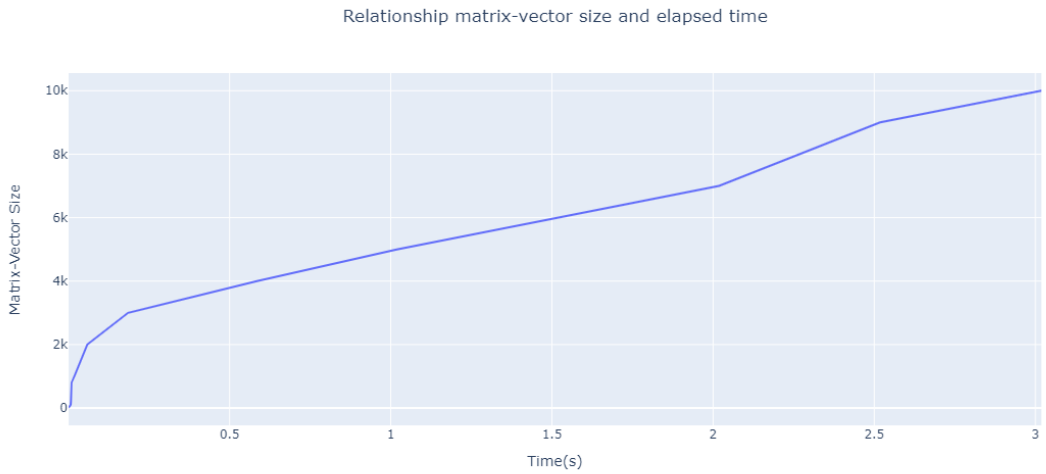
```
Execution time in seconds
0.000510    s
```

d. For execution rate (operations per second), size=10:

```
double FLOPS=(N*N)/mkl_time_span1.count();
    printf("\nExecution rate (operations per second): \n");
    printf("%lf\t", FLOPS);
```

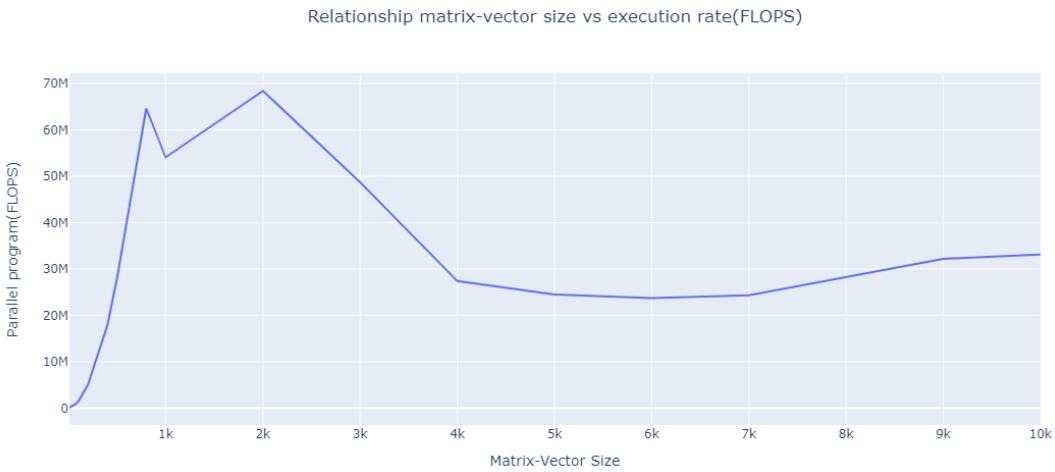
```
Execution rate (operations per second):
143204.926249
```

2. Results: Relationship between matrix-vector size and elapsed time:



For threading, in parallel multiplication, we did not see a notorious improvement, in fact, for arrays and vectors superior to 2000 the time for the multiplication got increased as the graph can show. However, we did try it in another computer (CPU) and in there we saw an improvement, therefore, even though the correctness is reliable and "correct" (results same as sequential code), the efficiency was not superior for my personal computer for all sizes.

Relationship between matrix-vector size and execution rate (operations per second):



Part 4: Compare correctness against BLAS and/or MKL

The MKL multiplication code is with name: MKL-Multiplication.cpp

1. Implementation in MKL:

A.1. Coding function for the generation of pseudo-random matrices and vectors with own algorithm, with size equals to M, A[M][M]xB[M]=C[M]. The code is similar to part 2.

```
double random(double i, double j) {
   return ((double(3.5)*j)/double(2.14253) * i);
}

template <class T>
void init_matrix(T* a, T value) {

	if (value == 1) {
		for (int i = 0; i < MAX; i++) {
			a[i] = random(i + 0.61, 0.213);
		}

	}
	if (value == 0) {
		for (int i = 0; i < MAX; i++) {
			for (int j = 0; j < MAX; j++) {
				matA[i][j] = random(i + 0.12, j + 0.23);
			}
		}
		int k = 0;
		for (int i = 0;i < MAX;i++) {
			for (int j = 0;j < MAX;j++) {
				a[k] = matA[i][j];
				k++;
			}

		}

	}
```

A.2. Coding function for calculating the time it takes for the matrix-vector multiplication, it was done with <chrono> library. The code is the same as for part 2.

```
auto get_time() {
```

```
    return std::chrono::high_resolution_clock::now();
}
```

A.3. Coding function for calculating the matrix-vector multiplication with <MKL.h> in C++:

a. Allocating memory:

```
A = (double*)mkl_malloc(m * k * sizeof(double), 64);
B = (double*)mkl_malloc(k * n * sizeof(double), 64);
C = (double*)mkl_malloc(m * n * sizeof(double), 64);
```

b. Multiplication function with cblas_dgem:

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k, alpha, A, k,
B, n, beta, C, n);
```

A.4. Showing the outputs, result of the multiplication, C[M] and/or elapsed time for the multiplication:

a. For the matrices generated: E.g. for a A[5][5] and B[5]. The Matrices are the same in part2.

Note: Matrix A for cblas_dgemm is introduced as an array(1D), that's why we see Matrix A as an array below:

```
Initialize the matrix A
0.0450869 0.241117 0.437147 0.633177 0.829207 0.420811 2.25042 4.08004 5.90965 7.73926 0.796535 4.25973 7.72293
11.1861 14.6493 1.17226 6.26904 11.3658 16.4626 21.5594 1.54798 8.27834 15.0087 21.7391 28.4694

Initialize the matrix B
0.212251
0.560205
0.908158
1.25611
1.60406
```

b. For the result of the multiplication C[5]: The result differs a little bit.

```
Result of the multiplication
2.66708
24.8928
47.1185
69.3442
91.5699
```

c. For the elapsed time of the multiplication:

```
Elapsed time MKL: 0.0036123 s
```

d. For execution rate (operations per second), size=10:

```
double FLOPS = (MAX * MAX) / mkl_time_span.count();
      cout << "Execution rate (operations per second): " << FLOPS << endl;
```

```
Execution rate (operations per second): 24339.2
```

2. Showing the correctness(results) of part2 and part 3 with the results from MKL:

a. For a matrix-vector multiplication n=5(size): We can see that the results are really similar if not the same.

For size n=5		
Sequenctial Multiplication	**Parallel Multiplication**	**Multiplication with MKL**
2.66708	2.667083	2.66708
24.8928	24.892775	24.8928
47.1185	47.118467	47.1185
69.3442	69.344158	69.3442
91.5699	91.56985	91.5699

b. For a matrix-vector multiplication n=10: We can see that the results are really similar if not the same.

For size n=10		
Sequenctial Multiplication	**Parallel Multiplication**	**Multiplication with MKL**
22.1136	22.113635	22.1136
206.394	206.393930	206.394
390.674	390.674225	390.674
574.955	574.954519	574.955
759.235	759.234814	759.235
943.515	943.515109	943.515
1127.8	1127.795403	1127.8
1312.08	1312.075698	1312.08
1496.36	1496.355993	1496.36
1680.64	1680.636287	1680.64

c. For a matrix-vector multiplication n=50: We can see that the results are really similar if not the same.

For size n=50		
Sequenctial Multiplication	**Parallel Multiplication**	**Multiplication with MKL**
2828.02	2828.023699	2828.02
26394.9	26394.887859	26394.9
49961.8	49961.752019	49961.8
73528.6	73528.616180	73528.6
97095.5	97095.480340	97095.5
120662	120662.344500	120662
144229	144229.208660	144229
167796	167796.072820	167796
191363	191362.936980	191363
214930	214929.801140	214930

238497	238496.665301	238497
262064	262063.529461	262064
285630	285630.393621	285630
309197	309197.257781	309197
332764	332764.121941	332764
356331	356330.986101	356331
379898	379897.850261	379898
403465	403464.714421	403465
427032	427031.578582	427032
450598	450598.442742	450598
474165	474165.306902	474165
497732	497732.171062	497732
521299	521299.035222	521299
544866	544865.899382	544866
568433	568432.763542	568433
592000	591999.627702	592000
615566	615566.491863	615566
639133	639133.356023	639133
662700	662700.220183	662700
686267	686267.084343	686267
709834	709833.948503	709834
733401	733400.812663	733401
756968	756967.676823	756968
780535	780534.540984	780535
804101	804101.405144	804101
827668	827668.269304	827668
851235	851235.133464	851235
874802	874801.997624	874802
898369	898368.861784	898369
921936	921935.725944	921936
945503	945502.590104	945503
969069	969069.454265	969069
992636	992636.318425	992636
1.02E+06	1016203.182585	1.02E+06
1.04E+06	1039770.046745	1.04E+06
1.06E+06	1063336.910905	1.06E+06
1.09E+06	1086903.775065	1.09E+06
1.11E+06	1110470.639225	1.11E+06
1.13E+06	1134037.503386	1.13E+06
1.16E+06	1.16E+06	1.16E+06

3. Compute the residual by obtaining the difference between your results and the results from BLAS/MKL for the same inputs.

a. For a matrix-vector multiplication n=5: We can see that the residual is "0" for MKL – Sequential multiplication, showing in this way that our code was correct. For MKL-parallel, there is a difference, but this is really small as you can see below.

For size n=5	
Residual 1: MKL - Sequential	**Residual 2: MKL - Parallel**
0	-3E-06
0	2.5E-05
0	3.3E-05
0	4.2E-05
0	5E-05

b. For a matrix-vector multiplication n=10: We can see that the residual is "0" for MKL – Sequential multiplication, showing in this way that our code was correct. For MKL-parallel, there is a difference, but this is really small as you can see below.

For size n=10	
Residual 1: MKL - Sequential	**Residual 2: MKL - Parallel**
0	-3.5E-05
0	7E-05
0	-0.000225
0	0.000481
0	0.000186
0	-0.000109
0	0.004597
0	0.004302
0	0.004007
0	0.003713

c. For a matrix-vector multiplication n=50: We can see that the residual is "0" for MKL – Sequential multiplication, showing in this way that our code was correct. For MKL-parallel, there is a difference, that becomes bigger as the size of matrices and vectors increase.

For size n=50	
Residual 1: MKL - Sequential	**Residual 2: MKL - Parallel**
0	-0.003699
0	0.012141
0	0.047981
0	-0.01618
0	0.01966
0	-0.3445
0	-0.20866
0	-0.07282
0	0.06302
0	0.19886
0	0.334699
0	0.470539
0	-0.393621
0	-0.257781
0	-0.121941
0	0.013899

0	0.149739
0	0.285579
0	0.421418
0	-0.442742
0	-0.306902
0	-0.171062
0	-0.035222
0	0.100618
0	0.236458
0	0.372298
0	-0.491863
0	-0.356023
0	-0.220183
0	-0.084343
0	0.051497
0	0.187337
0	0.323177
0	0.459016
0	-0.405144
0	-0.269304
0	-0.133464
0	0.002376
0	0.138216
0	0.274056
0	0.409896
0	-0.454265
0	-0.318425
0	-3.182585
0	-0.046745

Part 5: Compare performance against BLAS and/or MKL

1. Research your hardware specifications, for your CPU, find out clock speeds, instructions per clock cycle, caches and sizes, number of cores, memory, etc.

CPU: Intel® Core™ i7-7500U
CPU speed: 2.70 GHz
Base Speed: 2.90 GHz
Cores: 2
Sockest: 1
Logical processors: 4
L1 cache: 128KB
L2 cache: 512KB
L3 cache: 4.0MB
This processor counts on a Kaby Lake microarchitecture [8], therefore, for double precision that architecture offers 16 FLOPs/cycle[9] which is going to be used to calculate the **theoretical peak performance.**

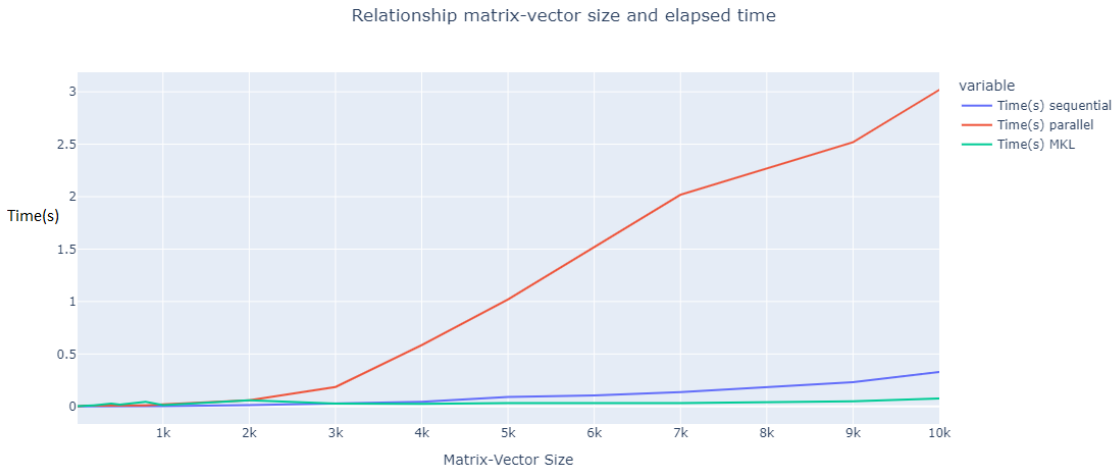
FLOPs by microarchitecture [edit]

x86 [edit]

Microarchitecture	FLOPs			ISA
Intel Microarchitectures				
Core Penryn Nehalem	**EUs**	1 × 128-bit Multiplication + 1 × 128-bit Addition		SSE (128-bit)
	DP	4 FLOPs/cycle	2 FLOPs + 2 FLOPs	
	SP	8 FLOPs/cycle	4 FLOPs + 4 FLOPs	
Sandy Bridge Ivy Bridge	**EUs**	1 × 256-bit Multiplication + 1 × 256-bit Addition		AVX (256-bit)
	DP	8 FLOPs/cycle	4 FLOPs + 4 FLOPs	
	SP	16 FLOPs/cycle	8 FLOPs + 8 FLOPs	
Haswell Broadwell Skylake	**EUs**	2 × 256-bit FMA		
Kaby Lake	**DP**	16 FLOPs/cycle	2 × 8 FLOPs	AVX2 & FMA (256-bit)
Amber Lake Coffee Lake Whiskey Lake	**SP**	32 FLOPs/cycle	2 × 16 FLOPs	

2. Performance of sequential and parallel implementations against MKL:

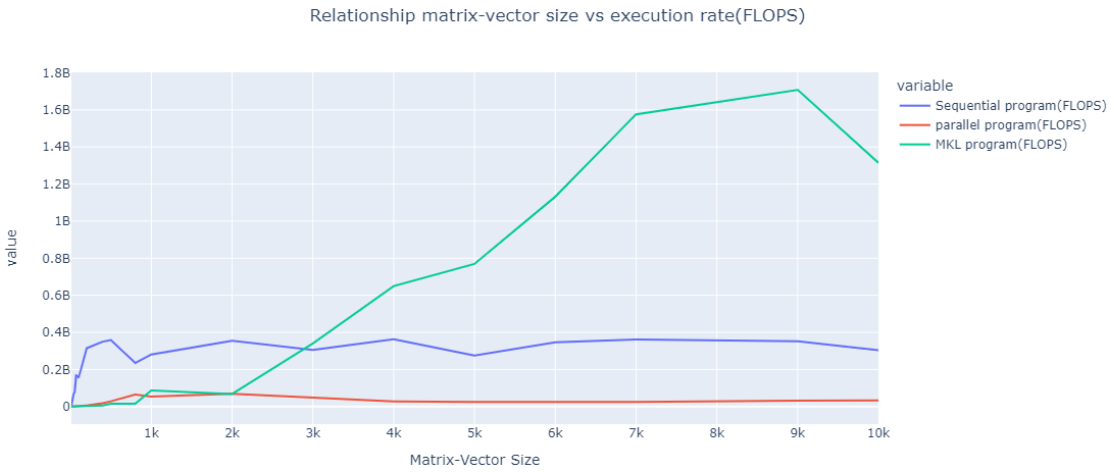
Overall, we saw the best performance with MKL, the parallel implementation was also good until it reached a matrix-vector size of 2K, and from that point on it had more delay than the other solutions. The sequential solution was also good and exact. The figure below illustrates our finding in a comparison of size of the matrix-vector vs the seconds it took the program to execute.



Some sample times that were graphed are:

Matrix-Vector Size	Time(s) sequential	Time(s) parallel	Time(s) MKL
10	1.70E-05	0.000679	0.0034041
20	1.29E-05	0.001487	0.00376692
30	1.35E-05	0.002148	0.0040086
40	2.30E-05	0.003868	0.0069707
50	3.23E-05	0.003898	0.0042438
70	2.86E-05	0.005958	0.005532
100	6.47E-05	0.006958	0.0049193
200	0.0001269	0.007958	0.0090856
400	0.0004573	0.008958	0.0265631
500	0.000697	0.0089	0.0165324
800	0.0027223	0.0099	0.0441606
1000	0.0035567	0.018474	0.0115874
2000	0.0112555	0.058474	0.0587818
3000	0.029508	0.18474	0.0264683
4000	0.0440385	0.58474	0.0246241
5000	0.0910504	1.018474	0.0325062
6000	0.104012	1.518474	0.0317976
7000	0.135178	2.018474	0.0310873
9000	0.230302	2.518474	0.0474314
10000	0.329083	3.018474	0.0760784

Additionally, below we can see the performance comparing the relationship matrix-vector size vs execution rate (operations per second):



Some sample times that were graphed are:

Matrix-Vector Size	Sequential program(FLOPS)	Parallel program(FLOPS)	MKL program(FLOPS)
10	5.88E+06	1.47E+05	29376.3403
20	3.10E+07	2.69E+05	106187.5484
30	6.67E+07	4.19E+05	224517.2878
40	6.96E+07	4.14E+05	229532.1847

50	7.74E+07	6.41E+05	589094.6793
70	1.71E+08	8.22E+05	885755.6038
100	1.55E+08	1.44E+06	2032809.546
200	3.15E+08	5.03E+06	4402571.102
400	3.50E+08	1.79E+07	6023393.354
500	3.59E+08	2.81E+07	15121821.39
800	2.35E+08	6.46E+07	14492556.71
1000	2.81E+08	5.41E+07	86300636.9
2000	3.55E+08	6.84E+07	68048273.45
3000	3.05E+08	4.87E+07	340029393.7
4000	3.63E+08	2.74E+07	649769940.8
5000	2.75E+08	2.45E+07	769084051.7
6000	3.46E+08	2.37E+07	1132160918
7000	3.62E+08	2.43E+07	1576206361
9000	3.52E+08	3.22E+07	1707729479
10000	3.04E+08	3.31E+07	1314433532

3. Theoretical peak performance for the processor:

TPP is defined as = (CPU speed in GHz)x(number of CPU cores)x(CPU instruction per clock cycle)x(number of CPUs per node)
TPP= (2.70 GHz)x(2)x(16 FLOPs/cycle)x(1)
TPP= 86.4Gflops/s

Optional: (up to 8 bonus points) try and analyze your code on other computers with different processors. You must include the specifications of each architecture used.

We analyzed our code in one other computer.

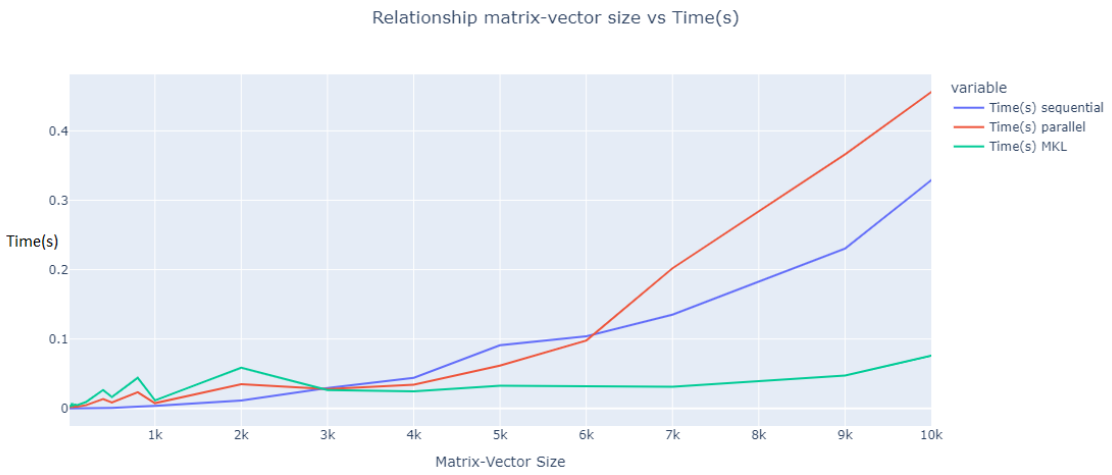
1. Specifications:

CPU: Intel® Core™ i7-1255U
CPU speed: 4.70 GHz
Base Speed: 4.70 GHz
Cores: 10
L1 cache: 928KB
L2 cache: 9.5MB
L3 cache: 12.0MB

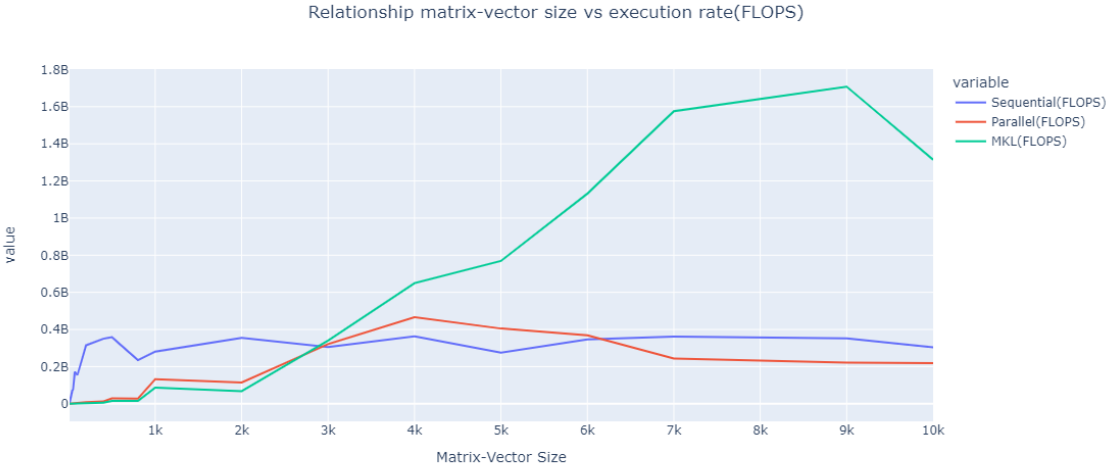
IDE: Visual Studio 2022
Also, Visual Studio Code

2. Performance of sequential and parallel implementations vs MKL:

As described before, with this computer we got better results for our parallel implementation, obtaining the graph below:



Additionally, below we can see the performance comparing the relationship matrix-vector size vs execution rate (operations per second):



CONCLUSION:

Overall, we have implemented and compared matrix-vector multiplication with sequential, parallel and MKL implementations. We have seen that MKL offers that best performance, followed by a parallel implementation, and lastly by the sequential implementation. However, the results also depend on the type of processors our computers have, because an implementation can perform better or worse depending on the characteristics of our CPU.

How to execute the code for sequential, parallel and MKL programs?

1. For sequential code:

Modify the constant MAX to calculate the multiplication of the size you wrote for matrix A and vector B, by default is 10:

```
const int MAX=10;
```

As result you are going to see C[MAX](result of the multiplication) and the elapsed time.

2. For parallel code:

Modify the constant N to calculate the multiplication of the size you wrote for matrix A and vector B, by default is 10:

```
#define N 10
```

As result you are going to see C[N](result of the multiplication) and the elapsed time.

3. For MKL code:

Modify the constant MAX to calculate the multiplication of the size you wrote for matrix A and vector B, by default is 10:

```
#define MAX 10
```

As result you are going to see C[MAX](result of the multiplication) and the elapsed time.

REFERENCES

[1] https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

[2] https://netlib.org/blas/

[3] https://en.wikipedia.org/wiki/LAPACK

[4] https://netlib.org/lapack/

[5] https://en.wikipedia.org/wiki/Math_Kernel_Library

[6] https://community.intel.com/t5/Intel-oneAPI-Math-Kernel-Library/Difference-between-Intel-MKL-LAPACK-and-Netlib-LAPACK-methods/td-p/1266354

[7] https://en.wikipedia.org/wiki/Row-_and_column-major_order

[8] https://en.wikichip.org/wiki/intel/core_i7/i7-7500u

[9] https://en.wikichip.org/wiki/flops

APPENDICES

Code to graph our results in python:

```
import pandas as pd
import plotly.express as px

#df = pd.read_csv('graphs.csv')
#df1 = pd.read_csv('graphs_parallel.csv')
#df2 = pd.read_csv('MKL_times.csv')
df3 = pd.read_csv('Times2_F.csv')
```

```
fig = px.line(df3, x = 'Matrix-
Vector Size', y =df3.columns[1:], title='Relationship matrix-
vector size vs execution rate(FLOPS)')
#fig = px.line(df3, x = 'Matrix-
Vector Size', y =df3.columns[1:], title='Relationship matrix-
vector size vs Time(s)')
#fig = px.line(df3, x = 'Matrix-
Vector Size', y ='Parallel program(FLOPS)', title='Relationship mat
rix-vector size vs execution rate(FLOPS)')
fig.update_layout(title_x=0.5)

fig.show()
```