Centrale Nantes - Faculty of Engineering

COMPUTATIONAL MECHANICS

Domain Decomposition and iterative Solvers

Lab 1: Dense matrix product and BLAS

A report submitted by:

B C Sreekanth Reddy

Submitted to:

Prof. Nicolas Chevaugeon

OBJECTIVES:

To understand the efficient libraries: BLAS, OpenBLAS for linear algebra basic operations which makes use of the available computer architecture effectively.

OUTLINE:

Introduction and brief explanation on code provided.

Implementation of matrix multiplication is performed in a member function: mulV1(), mulV2(), muldgemm.

In first part, I have demonstrated the two ways of implementing the best handmade loop i.e., using the pointer and over load operator () to fetch the data from the memory to the processor.

In the second part, the pointer method of fetching is implemented with different ordering of the loop (second best handmade loop).

In the third part, the implantation using the BLAS library is demonstrated.

In the fourth part, the implementation using the OpenBLAS is demonstrated and concluded with goals achieved from this lab exercise.

Introduction: Ways of storing the data:

- Full storage: Row major, Column major
- Symmetric storage
- Band storage
- Skyline storage
- Co-ordinate storage
- Compressed Row storage/ Column storage

The given code is based on the dense matrix with column storage.

Jargons of performance metrics:

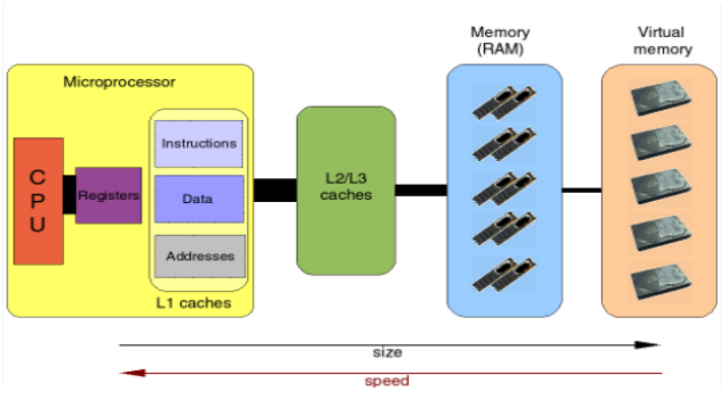


Fig.1 Computer architecture

Memory Latency: A performance metric, measures the time taken for processor to retrieve data or instructions from memory. A very less latency is obtained if the required data is in registers.

Several factors contributing to memory latency:

- **Memory hierarchy:** Modern processors includes multiple levels of memory, such as CPU caches (L1, L2, L3) and main memory (RAM). Each level has different latency characteristics. Registers which sit next to the CPU are the fastest and virtual memory being the slowest.
- **Cache Performance:** CPU caches are designed to reduce the memory latency by storing the frequently accessed data closer to the CPU cores. When the data is present on cache, it can be fetched using low latency.
- **Memory Access Patterns:** As we have discussed earlier the structure of the data storage and structure of data fetching also plays a huge role. Contiguous patterns tend to have a lower latency than the random-access patterns.
- **Memory controller and Bus speed:** These connecting the RAM to the processor can affect the latency. Faster memory controllers and wide buses can reduce the time it takes to transmit data from RAM and CPU.
- **Concurrency:** Memory latency can also be affected by concurrent memory requests from multiple threads or processes. Contentions for memory access can lead to longer latencies as the memory controller schedules and prioritize the requests.

Memory bandwidth: A performance metric, measuring the rate at which the data can be read from or written to a memory, typically measured in bytes per second.

Pipe-Lining: At the processor level, pipe-lining plays an important role. It is quite like the assembly line of car. CPU's use pipelining to overlap the execution of multiple instructions. Each stage in the pipelining is responsible for different parts of execution. The overlapping of instructions allows the processor to execute multiple instructions simultaneously.

The 5-stage pipelining includes:

Instruction fetches→ Instruction decode→ Execute → Memory Access → Write back.

The number of stages in pipelining vary depending on the micro-architecture of the processor. Moder processors have more than 5 stages.

Going deep in to the architecture of the computer is not our objective but to make use of the most efficient ways to get the results efficiently.

Brief explanation on the code:

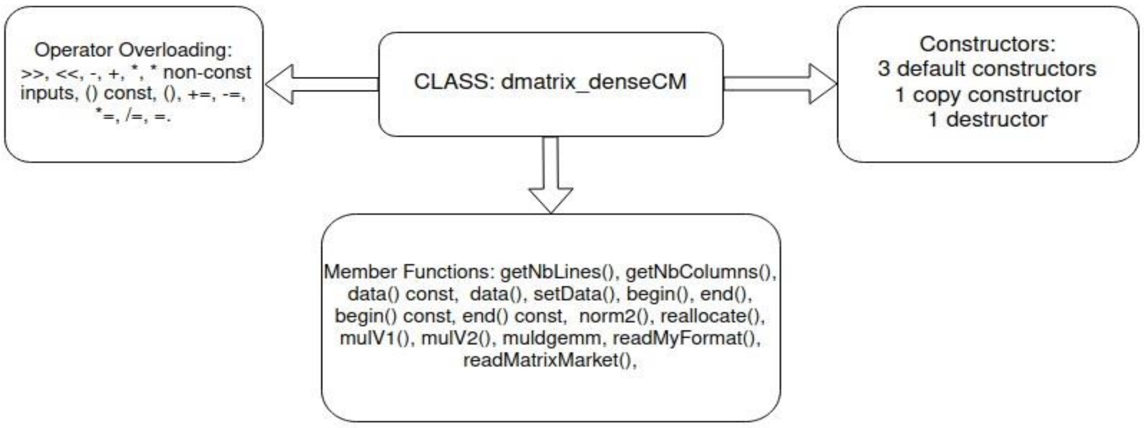

Operator Overloading:
>>, <<, -, +, *, * non-const inputs, () const, (), +=, -=, *=, /=, =.
CLASS: dmatrix_denseCM
Constructors:
3 default constructors
1 copy constructor
1 destructor
Member Functions: getNbLines(), getNbColumns(), data() const, data(), setData(), begin(), end(), begin() const, end() const, norm2(), reallocate(), mulV1(), mulV2(), muldgemm, readMyFormat(), readMatrixMarket(),

Fig.2 Overview of Class

The class dmatrix_denseCM is defined, where the implementations for mulV1, mulV2, muldgemm are added.

There are additional files:

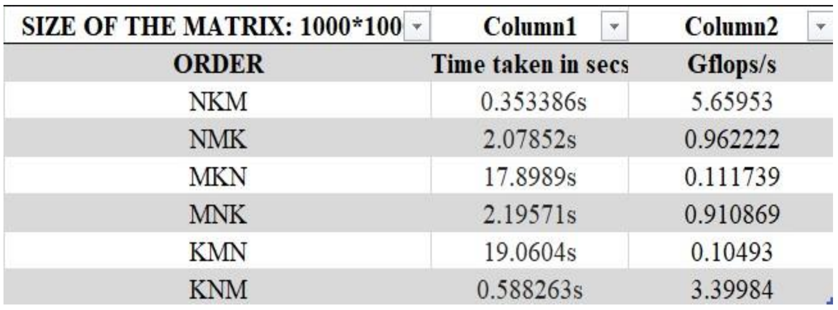
MMIO; input-output file

Main file

Implementation of matrix multiplication

The matrices are stored in column major:

One can construct 6 variants of matrix product:



SIZE OF THE MATRIX: 1000*100	Column1	Column2
ORDER	Time taken in secs	Gflops/s
NKM	0.353386s	5.65953
NMK	2.07852s	0.962222
MKN	17.8989s	0.111739
MNK	2.19571s	0.910869
KMN	19.0604s	0.10493
KNM	0.588263s	3.39984

Table.1 Variants of Algorithm

Matrix size: 1000 x 1000.

Number of floating-point operations: 1000^3 * 2 = 2E9

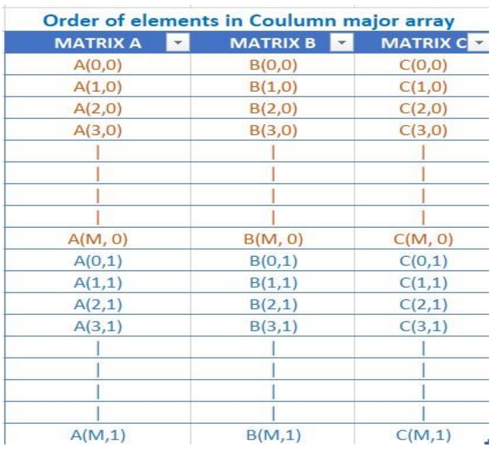
Note that the number of operations is same whatever the algorithm.

We can observe from the table.1 that the time taken and number of floating-point operations performed in a second are the different.

Why is execution time varying?

The way the matrix is stored in the memory, The way the data fetched from memory to processes are one of the factors.

The order of element storage memory is demonstrated in the below figure.3.



Order of elements in Coulumn major array

MATRIX A	MATRIX B	MATRIX C
A(0,0)	B(0,0)	C(0,0)
A(1,0)	B(1,0)	C(1,0)
A(2,0)	B(2,0)	C(2,0)
A(3,0)	B(3,0)	C(3,0)
|	|	|
|	|	|
|	|	|
|	|	|
A(M, 0)	B(M, 0)	C(M, 0)
A(0,1)	B(0,1)	C(0,1)
A(1,1)	B(1,1)	C(1,1)
A(2,1)	B(2,1)	C(2,1)
A(3,1)	B(3,1)	C(3,1)
|	|	|
|	|	|
|	|	|
|	|	|
A(M,1)	B(M,1)	C(M,1)

Fig3. Order of elements in storage

The below chart shows the flow of operations. We can observe that the elements fetching is in order of storage.

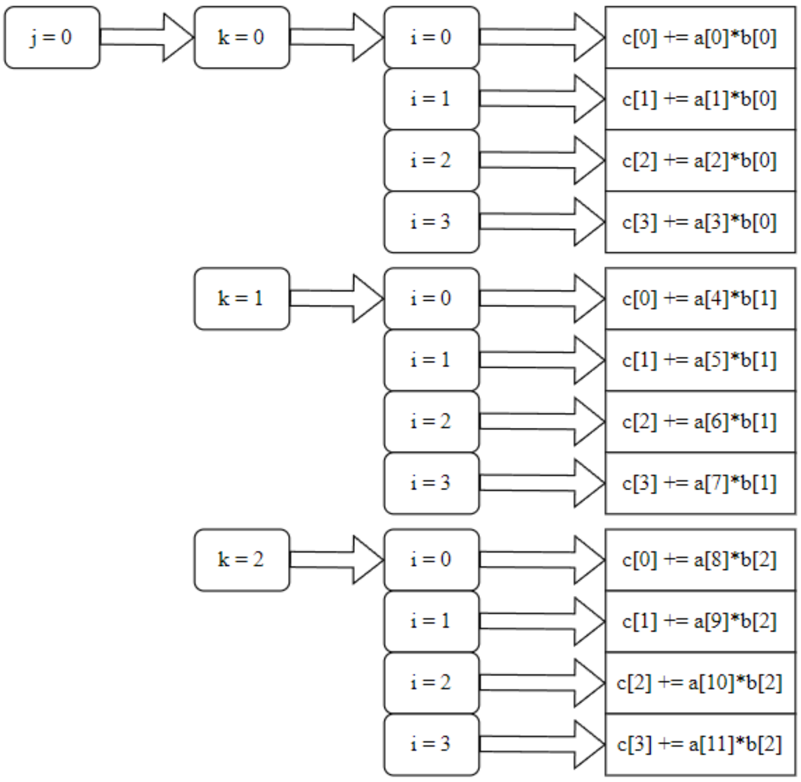


Fig. 4 Flow chart of the algorithm

Two different ways of implementation,

Method-1:

```
for(int j = 0; j < N; ++j)
    for(int k = 0; k < K; ++k)
        for(int i = 0; i < M; ++i)
            C(i,j) += A(i,k)*B(k,j);
```

Method-2:

```
const double * a = A.data();
const double * b = B.data();
double * c = C.data();


for(int j = 0; j < N; ++j)

        for(int k = 0; k < K; ++k)

                for(int i = 0; i < M; ++i)

                        *(c+i+j*M) += (*(a+i+k*M)) * (*(b+k+j*K));
```

Let us see what is the difference between two methods, In the method-1, the overloaded operator () is called to fetch the data every time which is costly and is significant as the size of matrices increases. In the method-2, we created a pointer a, b which are const and c non-const pointing to the initial address of matrix-A, B and C.

Order of fetching the data:

A: 4 x 3 matrix	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
B: 3 x 2 matrix	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]						
C: 4 x 2 matrix	C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]				

In inner loop,

In first iteration: In contiguous manner first column of C, first column of B is fetched. In every inner iteration respective column of A in contiguous manner is fetched.

In second iteration: In contiguous manner second column of C, second column of B is fetched. In every inner iteration respective column of A in contiguous manner is fetched.

.

.

In nth iteration: In contiguous manner last column of C, last column of B is fetched. In every inner iteration respective column of A in contiguous manner is fetched.

We can observe that the fetched data is in-line with the storage, which plays a huge role as the size of the matrix is increasing.

Square Matrix	Time in secs	Time in secs	Gflops	Gflops2
SIZE OF THE MATRIX	Over-load operator ()	Pointer	Over-load operator ()	Pointer
10	1.91E-06	1.13E-06	1.04932	1.77778
100	0.0018398	0.00029	1.08708	6.88148
1000	1.7243	0.356532	1.15989	5.60959
2000	15.3872	4.05523	1.03983	3.94552

Table.2 Comparison of results for implementation method using pointers vs overloading operator ()

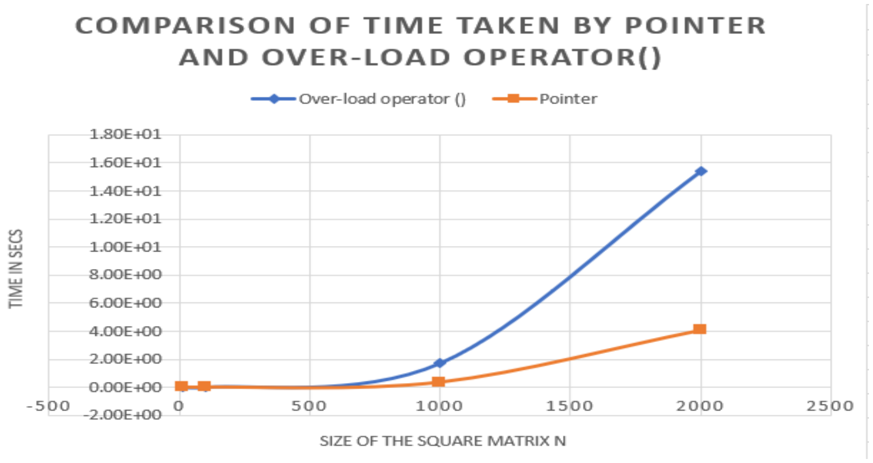


Fig.5 Comparison of execution time b/w pointer and () operator

As explained earlier, if we are fetching the data using over load operator (), member function is called N^3 times if a square matrix. As fetching is taking time taken to perform the operations on data fetched is delayed which can be seen in the graph. The blue line in the above Fig.5 shows the time taken to compute the matrix multiplication is higher compared to the orange line, which is not fetching the data but using the pointer to the initial address and performing the pointer arithmetic to fetch the data.

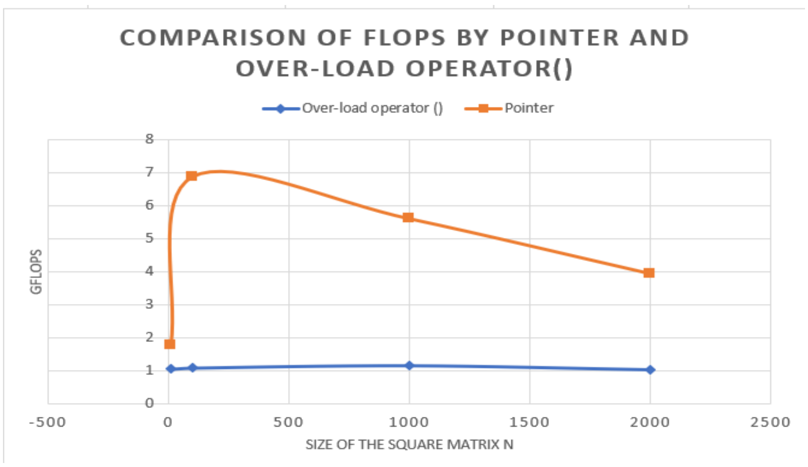


Fig.6 Comparison of flops b/w pointer and () operator

FLOPS = (Total Floating-Point Operations) / (Time in seconds)

No. of operations performed = N^3.

From the above graph it’s evident that the flops using pointer is decreasing, As the size of the matrix increases the time to fetch data from memory also increases, which causes idleness of the Arithmetic Logical Unit. Thereby decreasing the performance of the operation.

From fig.5 blue line shows the Flops by using the over-load operator is constant (decreases with N) because over-load operator () is taking time in calling the function and delay due to fetching the data which increases with increase in size.

Second Implementation: The pointer method is implemented with different ordering of the loop.

```
const double * a = A.data();
const double * b = B.data();
double * c = C.data();


for(int k = 0; k < K; ++k)
        for(int j = 0; j < N; ++j)
                for(int i = 0; i < M; ++i)
                        *(c+i+j*M) += (*(a+i+k*M)) * (*(b+k+j*K));
```

This is the second-best loop from the results.

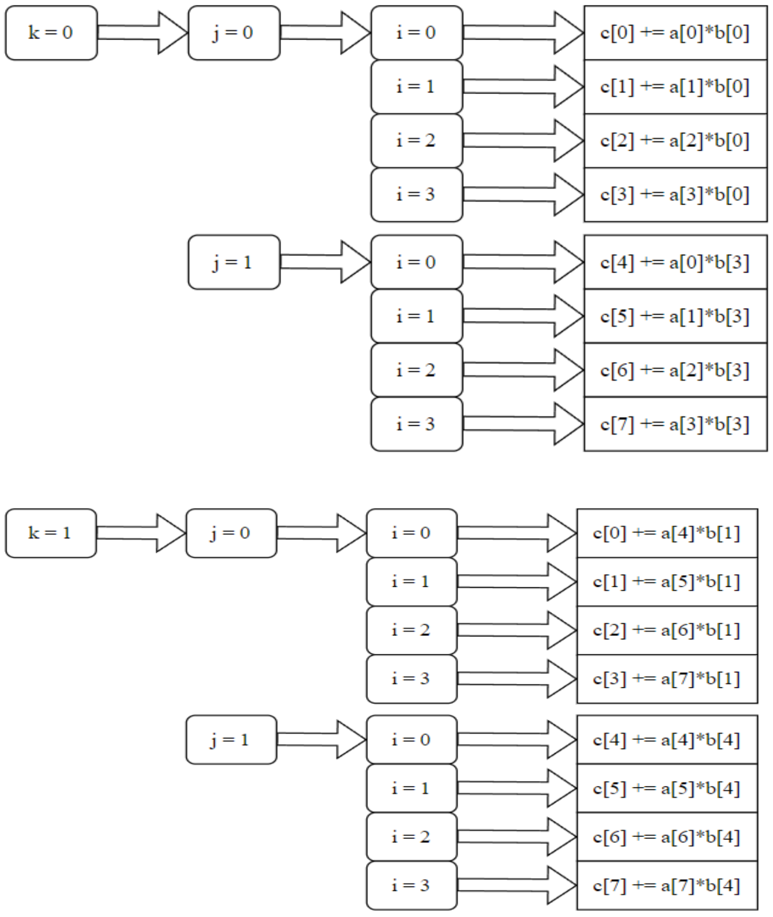


Fig. 7 Flow chart of the algorithm

In first iteration: columns of C in contiguous manner, first column of A, and first row of B are fetched.

In second iteration: columns of C in contiguous manner, second column of A, and second row of B are fetched.

.

.

.

In nth iteration: Columns of C in contiguous manner, nth column of A, and nth row of B are fetched.

As the size of matrix increases, fetching rows of B is costly. As we must skip from 1st → K*1 → K*2 → K*3 → ……… →K*N. (K, N: No. of rows & columns of B)

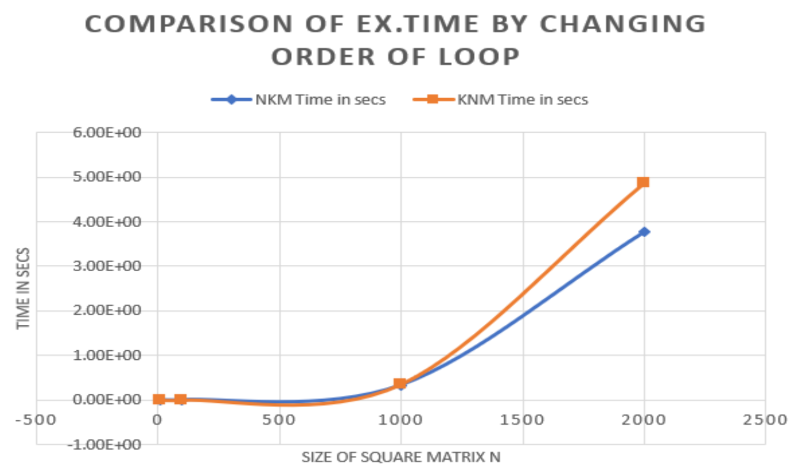


Fig.8 Comparison of execution time b/w two loops

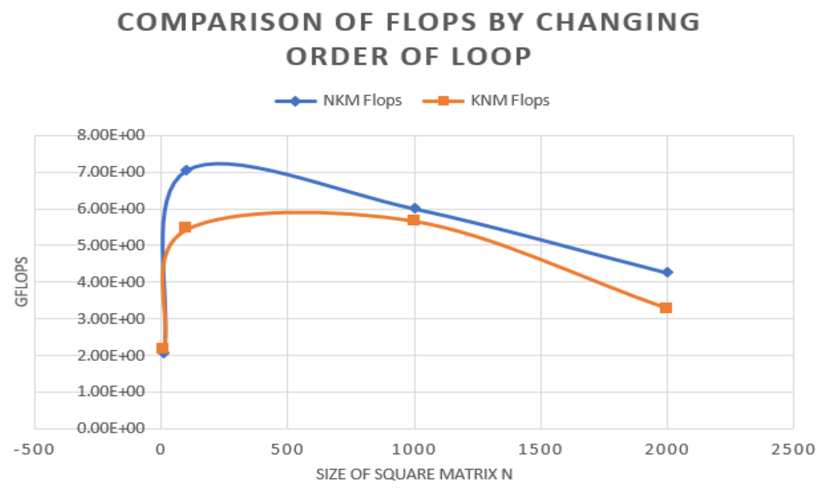


Fig.9 Comparison of FLOPS b/w two loops

The main difference between the two loops is fetching the data. In the NKM, the data is fetched in the column wise, the cost to fetch the data is approximately equal to fetching the first data onto the cache line. But in the second loop KNM, the array-B is fetched in row wise which is costly to fetch as explained earlier. The above fig.8 & 9 shows the significance of the order of fetching the data.

Using BLAS library:

In the above part, we have seen how to make use of the storage of the matrix and produce the effective result. This is not the only factor. The efficient use of processor can only be seen if all the data is in register.

There are certain libraries to consider to make use of the resources available efficiently. They are Optimized Libraries and Compiler optimizations.

Optimized library: BLAS (Basic Linear Algebra Subprograms)

BLAS are optimized for various hardware architectures. They take advantage of memory hierarchy to maximize the performance. They also take advantage of the multiple CPU cores or threads for performance gains. BLAS plays a huge role in numerical computing, providing high performance implementation of essential linear algebra operations.

Compiler optimizations: This refers to specific techniques the compiler applies to the code to make it run faster or use less memory. These optimizations are controlled by flags provided to compiler. The flags specifically target improving the generated machine code's efficiency.

In our program, we used the flag -O3 for optimizing the compiler to provide us the best efficiency.

BLAS OVERVIEW

BLAS is divided in 3 levels:

Level-1: Vector operation (xCOPY, xDOT, xNRM2, xSCAL, xAXPY…)

Level-2: Matrix-Vector operation (xGEMV, xGBMV, xSYMV, xSBMV,..)

Level-3: Matrix-Matrix operation (xGEMM, xSYMM, xTRMM, …)

For matrix multiplication, A and B being general dense matrix, we used dGEMM.

'x' in front of BLAS is denoting the datatype.

'S' - Single, 'D' - Double, 'C' – Complex, 'Z' – Double Complex.

xGEMM perform the operation $C = \alpha AB + \beta C$ needs 13 parameters.

dGEMM (TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC);

dGEMM stands for "Double precision General Matrix Multiply."

- TRANSA, TRANSB specifies whether the matrix A should be transposed before multiplication.

  It can take one of the following values:

  N or n: No transpose, T or t: Transpose, C or c: Conjugate transpose
- M: An integer that represents the number of rows in the resulting matrix C.
- N: An integer that represents the number of columns in the resulting matrix C.
- K: An integer that represents the number of columns in matrix A.
- ALPHA: A double precision scalar that scales the input matrices A and B.
- A, B, C: An array representing matrix A, B, C.
- LDA, LDB, LDC is the leading dimension of A, B, C. An integer representing the leading dimension of matrix `A`. It specifies how the elements of A are stored in memory.
- BETA: A double precision scalar that scales the input matrix C before adding the result of the multiplication.
- The leading dimensions are used to control how the data is stored in memory and accessed during the multiplication.

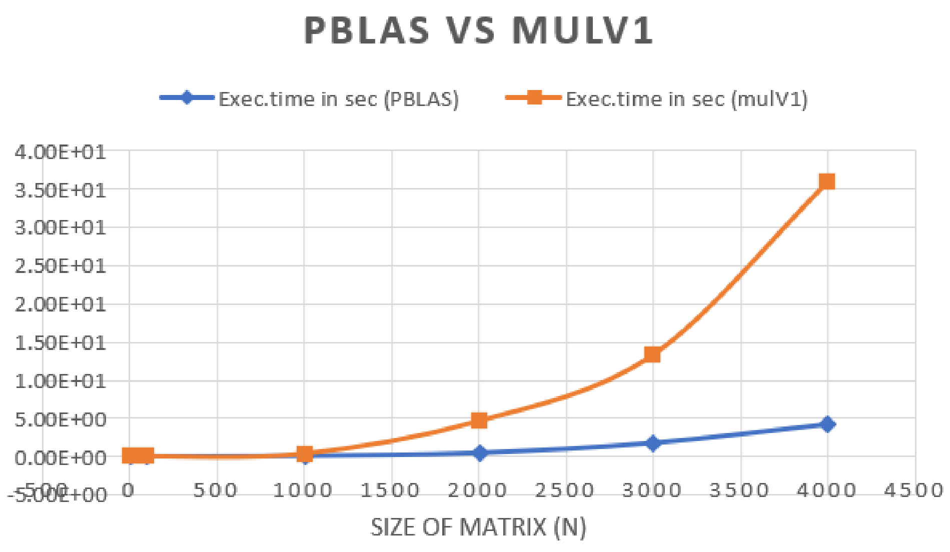


Fig.10 Comparison of execution time b/w PBLAS and Efficient handmade loop

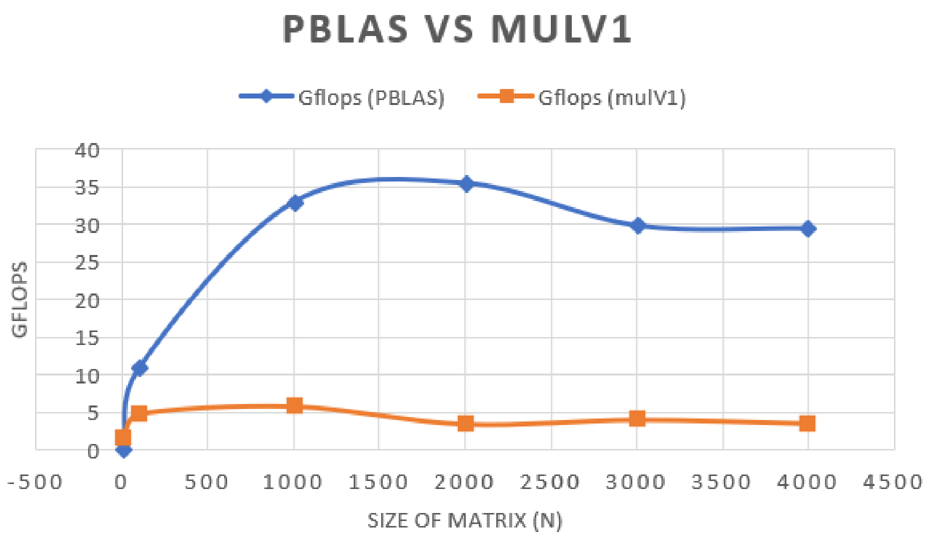


Fig.11 Comparison of FLOPS b/w PBLAS and Efficient handmade loop

In the Case-1, we made use of the efficient loop and we didn't consider the computer architecture to enhance the efficiency. Using BLAS library, we are making use of sub-routine and compiling using the blas flag, which is trying to make use of the efficient fetching through cache lines, making use of the efficient pipe-lining and the architecture to provide us the best it can.

From the fig.10 & 11, we can see the architecture of the computer is one of factors in being efficient. As the size of the matrix increases the execution time is quadratically increasing in handmade loop, while it is linear for BLAS. The FLOPS after certain size remains constant, As the fetching of data is efficient through efficient pipelining, cache lines, etc. using blas, it is evident from the plot the peak of 36 Giga flops is attained where only 5 Giga flops is attained using the hand-made loop.

Using OpenBLAS

OpenBLAS is a very fast implementation open source of BLAS. It includes architecture-specific optimizations that take advantage of the features and capabilities of modern CPUs, such as SIMD (Single Instruction, Multiple Data) instructions, multiple cores, and cache hierarchies, to achieve high-performance linear algebra operations. OpenBLAS supports multi-threaded execution, allowing it to take advantage of multiple CPU cores for parallel computation. This can significantly improve performance on multi-core processors. OpenBLAS also offers optional support for using GPUs (Graphics Processing Units) for accelerating certain linear algebra operations. This feature can provide substantial speedup for specific workloads when compatible GPU hardware is available. OpenBLAS is often used as a backend for numerical and scientific computing libraries like NumPy in Python and SciPy. This integration allows these libraries to benefit from the performance enhancements provided by OpenBLAS.

The implementation is similar to BLAS, while compiling we choose the flag for OpenBLAS.

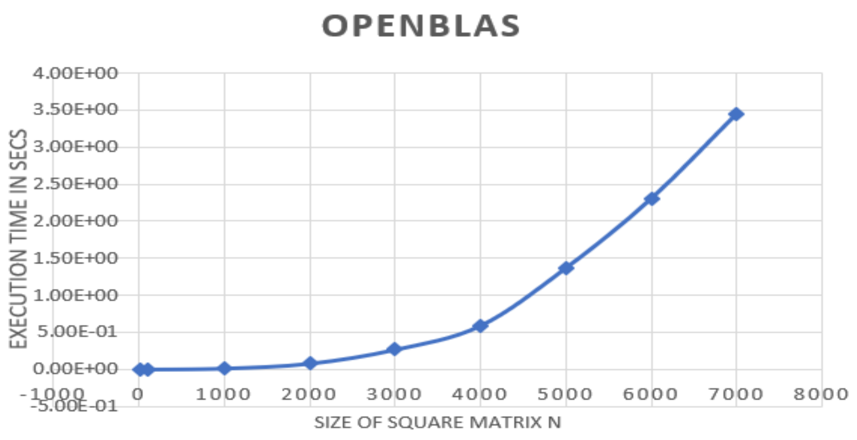


Fig.12 Execution time vs Size (N) using OpenBLAS

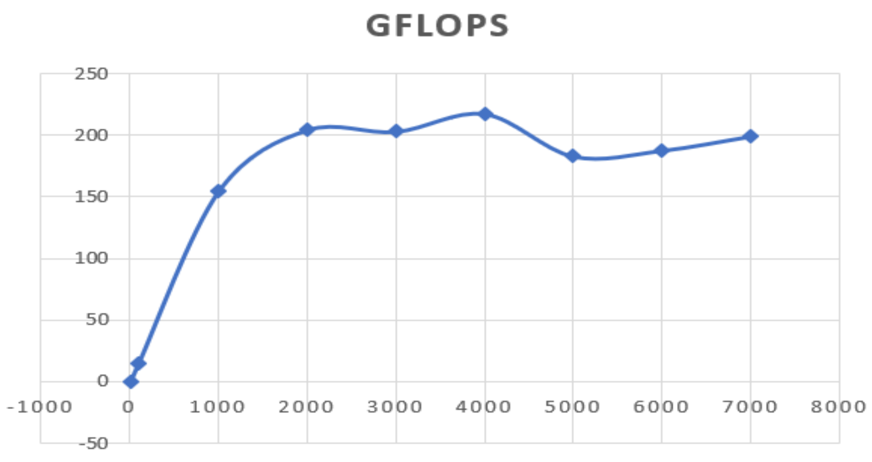


Fig.13 FLOPS vs Size (N) using OpenBLAS

As explained the best open-source available, it can be seen form the above fig. 12 & 13, The execution time for 4000*4000 matrix is 0.6 secs and the peak performance of 220 GFLOPS attained. The efficient use of the available resources is evident compared to hand-made algorithm.

Comparison between three algorithms:

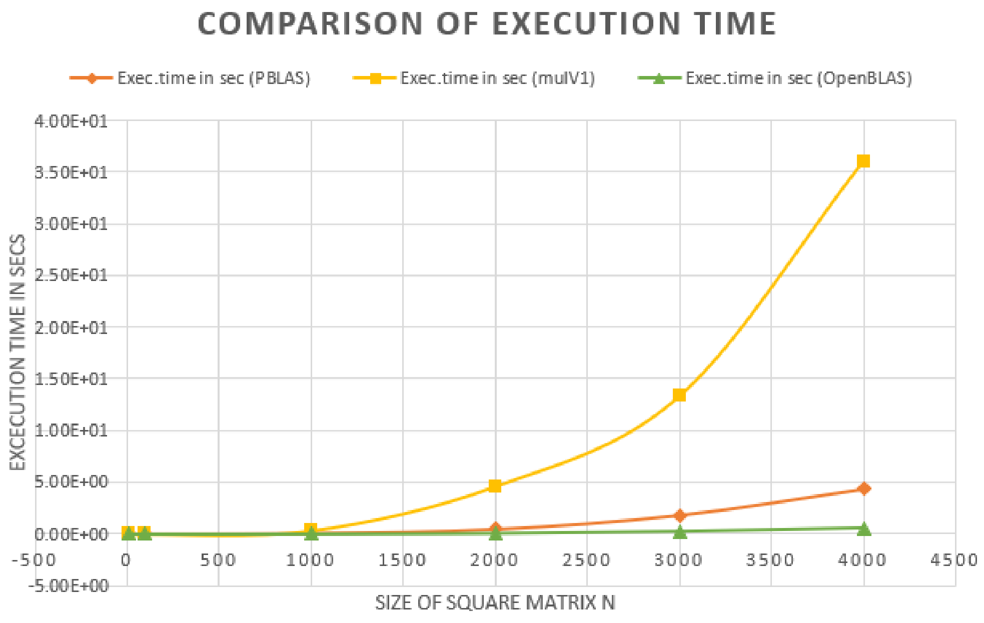


Fig.14 Comparison of Execution time vs Size (N)

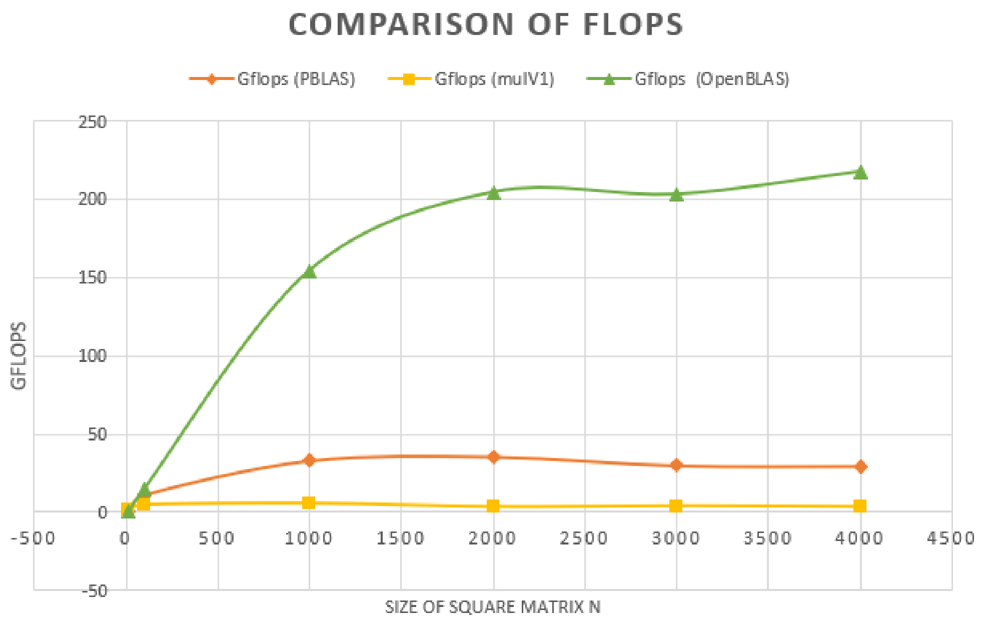


Fig.15 Comparison of FLOPS vs Size (N)

From the above fig. 14, 15 it can be seen the algorithm of the solution is only one of the factors which influences the execution time, the system architecture: the clock speed, number of cores, cache hierarchy, pipelining, memory hierarchy, bus architectures, instruction level parallelism, software optimisation (-O3), memory access patterns. Etc plays a major role in providing the best efficiency. Thanks to OpenBLAS library which provides the best possible efficiency based on the computer architecture available.

The execution time of OpenBLAS seems constant when compared to handmade loop and is linear compared to BLAS.

The peak performance of 220 Giga flops is attained and the performance not changes much and remains almost constant with increase in size.

Conclusion:

1. The main objective is to understand that there are efficient libraries developed which will make use of the computer architecture to the best.
2. Yes, we can find the efficient way of computing any algorithm based on the method of storage of the elements in the memory. But we can’t make use of the resources available efficiently like cache lines, threads, memory lines, pipe-lining etc. (We can but a tedious task)
3. The problem we have encountered is a dense matrix and we knew the storage of the matrix are in column major. But, if matrix is symmetric, triangular, banded, symmetric band... etc. The complexity of the algorithm increases and finding the efficient algorithm is expensive and time intensive.
4. From the above plots and explanation, it is evident that the OpenBLAS is way more efficient compared to the BLAS and handmade loop. OpenBLAS is known for its focus on performance and optimization. It aims to provide high-performance BLAS routines that are optimized for a wide range of CPU architectures.
5. Next time when we want to compute any basic linear algebra operation, we have a best available open library “OpenBLAS”, to make use of the available architecture efficiently.