# High Performance Computing Project - A.A. 2022/2023

Andres Bermeo Marinelli

September 19, 2023

# Contents

# Chapter 1

# Benchmarking MKL, OpenBLAS, and BLIS

## 1.1 Introduction

In this exercise we compare the performance of three High Performance Libraries (HPC): MKL, OpenBLAS, and BLIS. In particular, we focus on the level 3 BLAS function called `gemm`, which multiples an $m \times k$ matrix $A$ times a $k \times n$ matrix $B$ and stores the result in an $m \times n$ matrix $C$.

This function comes in two types, one for single precision (float) and the other for double precision (double). Furthermore, it is capable of exploiting paralellism using OpenMP (OMP) to speed up the calculations, provided that we have the required computational resources.

Using squared matrices only, we perform a scalability study in two scenarios. In the first scenario, we fix the number of cores, and increase the size of the matrices from 2000 to 20000. In the second scenario, we fix the matrix size to 10000 and increase the number of cores that `gemm` can use by modifying the `OMP_NUM_THREADS` environment variable.

In both scenarios, we repeat the measurements for both single and double precision, for both THIN and EPYC nodes, using the maximum number of cores.

Furthermore, for the second scenario, we also modify the thread affinity policy of OMP in order to observe any differences.

## 1.2 Methodology

### 1.2.1 Compiling BLIS and obtaining binaries

We begin by downloading the BLIS library by using the following commands:

```
$git clone https://github.com/flame/blis.git
$cd blis
$srun -p {NODE} -n1 ./configure --enable-cblas --enable-threading=openmp --prefix=/path/to/myblis
$srun -p {NODE} -n 1 --cpus-per-task={P} make -j {P}
$make install
```

Where `NODE` can be specified as either `THIN` or `EPYC` and `P` are the available cores for each node, 24 and 128 respectively.

With these commands, we have compiled the BLIS library for the desired architecture.

Next, we specify the flag in the Makefile to compile for float or double using `DUSE_FLOAT` or `-DUSE_DOUBLE`. Then, we run:

```
$salloc -n {P} -N1 -p {NODE} --time=1:0:0
$module load mkl/latest
$module load openBLAS/0.3.23-omp
$export LD_LIBRARY_PATH=/path/to/myblis/lib:$LD_LIBRARY_PATH
$srun -n1 make cpu
```

Which will generate the binaries for the desired architecture, with floar or double precision, depending on the flag we used.

To run, we use:

```
$srun -n1 --cpus-per-task=128  ./gemm_mkl.x {size_M} {size_K} {size_N}
$srun -n1 --cpus-per-task=128  ./gemm_oblas.x {size_M} {size_K} {size_N}
$srun -n1 --cpus-per-task=128  ./gemm_blis.x {size_M} {size_K} {size_N}
```

At the end of this procedure, we should have the appropriate binaries for each architecture, and for each type of precision, double or float.

We now detail the steps to obtain the measurements for both scenarios.

### 1.2.2 Using a fixed number of cores

For this section, we use all the cores available in a THIN or an EPYC node: 24 and 128, respectively.

Since we only use squared matrices, we can describe the dimensions of the matrices with a single number, which we call "size".

For both architectures, we start with a size of 2000 and end with a size of 20000, with jumps of 2000 for a total of 10 sizes. For each size, we repeat the measurement 10 times and report the average and standard deviation.

Finally, we repeat the measurements for both floating point precision and double point precision.

The scripts that were used can be found in the folder `exercise2/scripts`, under the name `es2_1_thin.sh` and `es2_1_epyc.sh`.

It is important to observe that in this section, since we are using the entire node, there is little possibility to play with combinations of thread affinity.

This will be done for the next section.

Furthermore, contrary to the guidelines for the exercise, we decided to use the entire node to benchmark its full capacity, and also to avoid wasting resources.

In fact, to obtain an accurate benchmark, we need to reserve the whole node, regardless of the number of cores we decide to use. This is because if other people began to use the other half of the node, this could introduce additional workloads which interfere with the benchmark.

### 1.2.3 Using a fixed matrix size

For this section, we fix the size of the matrices to 10000. Then, we slowly increase the number of cores to be used, until we reach the maximum.

To set the number of cores, we change the environment variable `OMP_NUM_THREADS` to the desired value.

For THIN nodes, which have 24 cores, we start using 1 core, then 2 and then we increase by steps of 2, for a total of 13 points.

For EPYC nodes, which have 128 cores, we start from 1, then 10 and then we increase by steps of 10 until 120. We also use 128 cores, to see what happens at full capacity. We obtain a total of 14 points.

We repeat all measurements 10 times and report the average and standard deviation.

As usual, we repeat this process for both floating and double point precision.

In this section, we have the liberty to explore different thread allocation policies since we are not always using the whole node.

We decided to use following combinations:

1. `OMP_PLACES=cores` and `OMP_PROC_BIND=close`
2. `OMP_PLACES=cores` and `OMP_PROC_BIND=spread`

The scripts that were used can be found in the folder `exercise2/scripts`, under the names `es2_2_close_thin.sh`, `es2_2_close_epyc.sh`, `es2_2_spread_thin.sh`, and `es2_2_spread_epyc.sh`.

| Node Type | Total Cores | IPC (SP) | IPC (DP) | $T_{pp}$ (SP) | $T_{pp}$ (DP) |
|---|---|---|---|---|---|
| THIN | 24 | 64 | 32 | $\sim$ 4 TFLOPS | $\sim$ 2 TFLOPS |
| EPYC | 128 | 32 | 16 | $\sim$ 10.6 TFLOPS | $\sim$ 5.3 TFLOPS |

Table 1.1: Performance table of THIN and EPYC architectures.

## 1.3 Results and Discussion

Before we discuss the results of both exercises individually, we briefly introduce the equation to calculate the theoretical peak performance ($T_{pp}$) of a machine:

$$T_{pp} = \text{Core Count} \times \text{clock freq.} \times \text{IPC} \tag{1.1}$$

Where `IPC` is the instructions per cycle that the architecture is capable of executing.

This equation is very intuitive. The clock frequency tells us how many cycles per second a single core is able to achieve. The ipc factor tells us how many instructions per cycle the core can execute. This number is different for single precision (SP) and double precision (DP) operations. Finally, we need to multiple this by the number of cores that our machine has.

On orfeo, THIN and EPYC nodes are composed of:

- THIN: 24 Intel(R) Xeon(R) Gold 6126 CPU's at 2.60GHz - Skylake
- EPYC: 128 EPYC AMD 7H12 CPU's at 2.60GHz - Zen 2 (7002 a.k.a "Rome")

Skylake architecture is reported[1] to be able to execute 64 SP FLOP per cycle and 32 DP FLOP per cycle. On the other hand, Zen 2 is reported[1] to execute 32 SP FLOP per cycle and 16 DP FLOP per cycle.

Therefore, we obtain:

### 1.3.1 Using a fixed number of cores

As mentioned above, in this section, we keep the number of cores fixed and we slowly increase the size of the matrices being multiplied from $m = 2000$ to $m = 20000$.

#### THIN Nodes

THIN nodes have two sockets each of 12 cores, for a total of 24 cores per node. The cores are Intel(R) Xeon(R) Gold 6126 CPU at 2.60GHz, which are part of the Skylake microarchitecture.

Furthermore, we know that the peak performance of any machine is given by the equation:

#### EPYC Nodes

### 1.3.2 Using a fixed matrix size

## 1.4 Conclusion

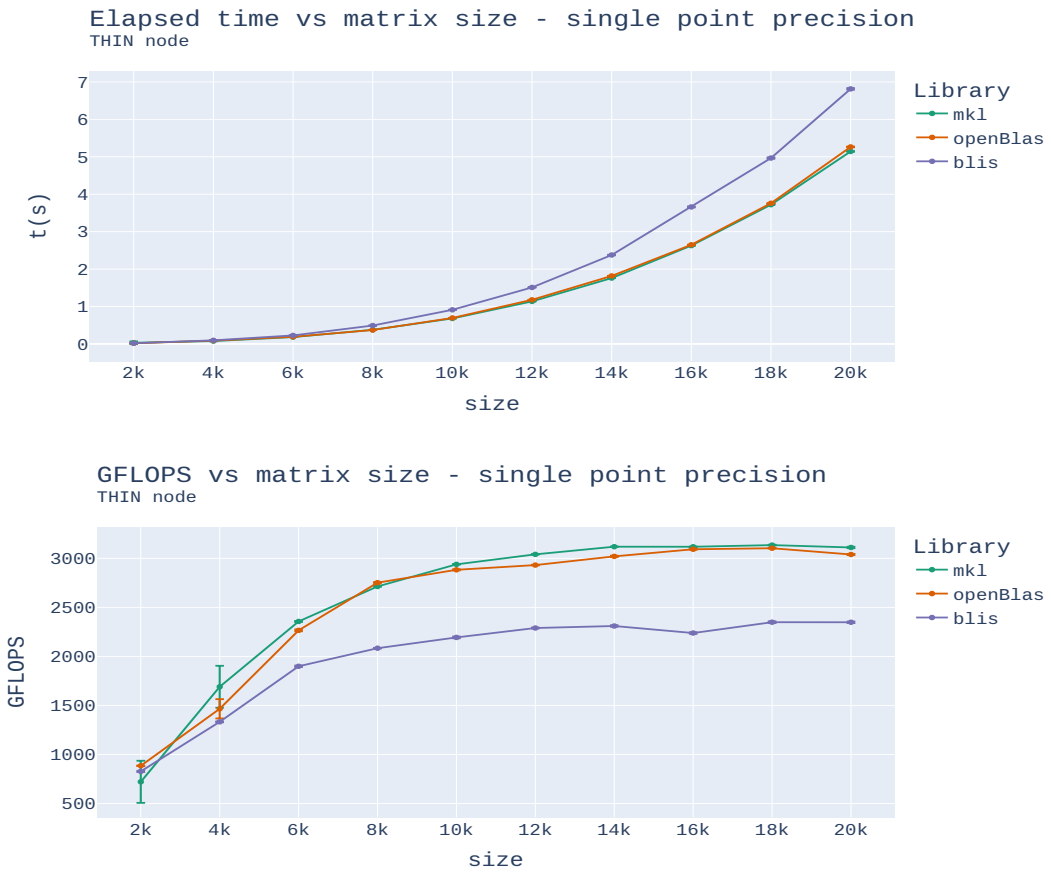
Elapsed time vs matrix size - single point precision
THIN node
t(s)
0
1
2
3
4
5
6
7
2k
4k
6k
8k
10k
12k
14k
16k
18k
20k
size
Library
mkl
openBlas
blis
GFLOPS vs matrix size - single point precision
THIN node
GFLOPS
500
1000
1500
2000
2500
3000
2k
4k
6k
8k
10k
12k
14k
16k
18k
20k
size
Library
mkl
openBlas
blis

Figure 1.1:

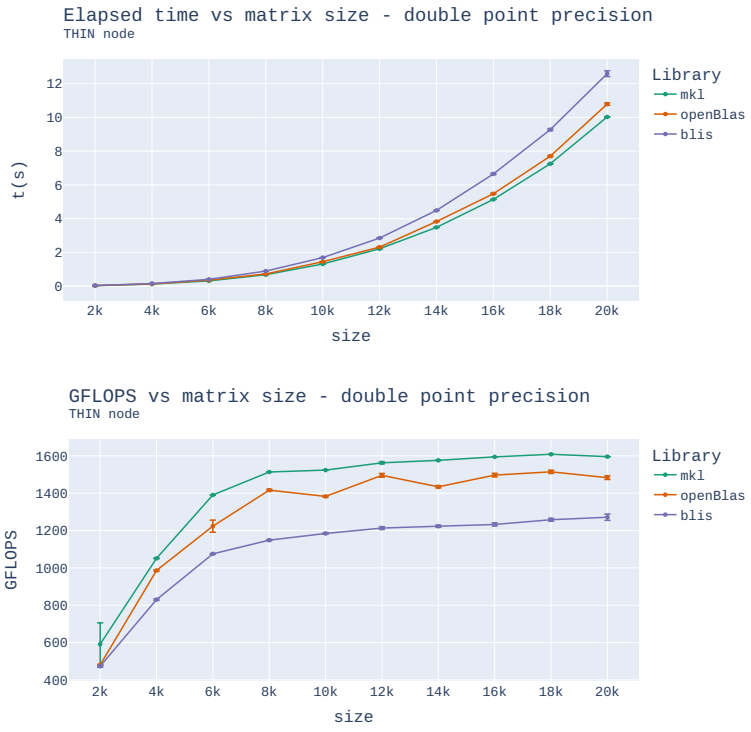


Figure 1.2: .

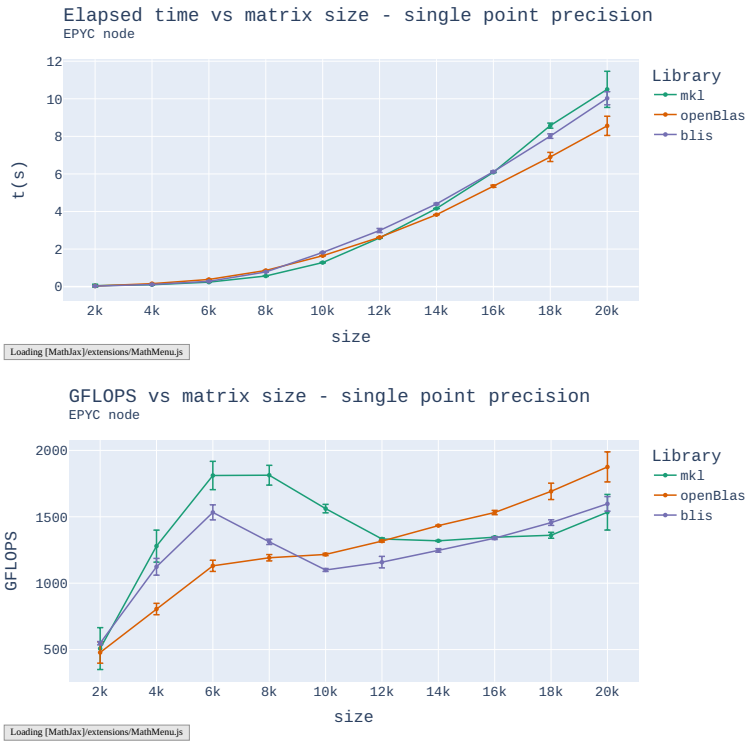


Figure 1.3: .


Elapsed time vs matrix size - double point precision
EPYC node
t(s)
0
5
10
15
20
25
2k
4k
6k
8k
10k
12k
14k
16k
18k
20k
size
Library
mkl
openBlas
blis
Loading [MathJax]/extensions/MathMenu.js
GFLOPS vs matrix size - double point precision
EPYC node
GFLOPS
200
400
600
800
1000
2k
4k
6k
8k
10k
12k
14k
16k
18k
20k
size
Library
mkl
openBlas
blis
Loading [MathJax]/extensions/MathMenu.js

Figure 1.4: .

# Chapter 2

# Conway's Game of Life

## 2.1 Introduction

This exercise is devoted to implementing a scalable version of Conway's Game of life[2]. The game consists of a $k \times k$ grid, where each cell can be "alive" or "dead".

The grid evolves over time by looking at a cell's $\mathcal{C}$ eight nearest neighbors and observing the following simple rules:

- A dead cell with exactly three live neighbors comes to live (*birth*).
- A live cell with two or three neighbors stays alive (*survival*).
- A dead or live cell less than two or more than three neighbors dies or stays dead by under or overpolulation, respectively (*death*).

These seemingly simple rules give rise to many interesting behaviors and patterns.

Depending on how we update the cells, there exist two methods to evolve the grid: `static` and `ordered`. In `static` evolution, we freeze the state of the grid $\mathcal{G}_t$ at time step $t$, and compute $\mathcal{G}_{t+1}$ separately, while looking at $\mathcal{G}_t$.

On the other hand, in `ordered` evolution, we start from a specific cell, usually in position $(0, 0)$ (top left), and update the elements inplace. In this scenario, the state of each cell depends on the evolution of all the cells before it.

Our implementation must satisfy the following requirements:

1. Randomly initialize a square grid ("playground") of size $k \times k$ with $k \geq 100$ and save it as a binary PGM file.
2. Load a binary PGM file and evolve for $n$ steps.
3. Save a snapshot during the course of evolution with frequency $s$ ($s = 0$ means save at the end).
4. Support both `static` and `ordered` evolution.

Lastly, it must use both MPI[3] and OpenMP[4] to paralellize the computations and be able to process grids of considerably high dimensions.

## 2.2 Methodology

Since programs in MPI need to be rewritten completely from their serial counterparts, we must begin to conceptualize the problem in an encapsulated manner from the start.

At a high abstract level, we must make two important choices:

1. How we will decompose the problem.
2. How we will perform the IO (this has important consequences on the organizational paradigm we will use).

We briefly discuss both of these topics more in depth in the following subsections.

### 2.2.1 Problem decomposition

To exploit paralellism, we must first identify the concurrency in our application and then apply some form of decomposition. The two most important types of decomposition are `domain` and `functional` decomposition.

In domain decomposition, multiple workers are performing the same set of instructions on different portions of data (SIMD). In functional decomposition, workers are performing different instructions on possibly different data (MIMD).

In the case of static evolution, each cell can be updated independently from each other, as long as we have acccess to its neighbors. Therefore, one immediately obviously form of paralellism is for each MPI process to update their "part" of the whole grid.

For this, we need to decide how to split the grid since we can do both a 1D or 2D decomposition of our grid. It is known that a 2D decomposition is more efficient as it can exploit more bandwidth (more workers will send shorter messages at the same time). However, it is more complicated both from an implementation point of view, as well as it is worse for memory access.

Let's talk briefly about this second aspect. Although we will talk about our grid as a 2D array, internally, for efficiency, it will be represented as a 1D array. Therefore, each process will work on a strip of continuous data as shown in the figure below.

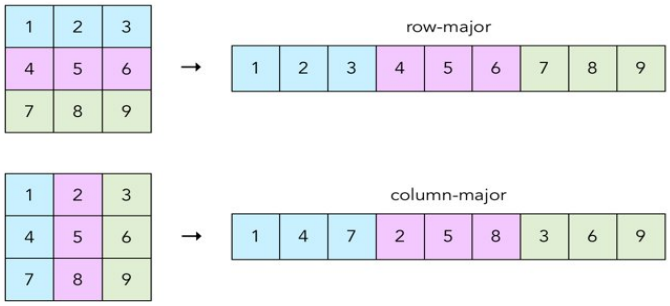


Figure 2.1: .

Fas one long array, in a 1D split, each process will need to work on a continuous "strip" of this memory.

## 2.3 Implementation

## 2.4 Results and Discussion

## 2.5 Conclusions

### 2.5.1 How to add Tables

# Bibliography


[1] wikichip. "Floating-point operations per second (flops)." (), [Online]. Available: https://en.wikichip.org/wiki/flops.

[2] LifeWiki. "Conway's game of life." (), [Online]. Available: https://conwaylife.com/wiki/Conway%27s_Game_of_Life.

[3] OpenMPI. "Openmpi." (), [Online]. Available: https://www.open-mpi.org/.

[4] OpenMP. "Openmp." (), [Online]. Available: https://www.openmp.org/.