Università della Svizzera italiana | **Institute of Computing CI**

High-Performance Computing Lab **Institute of Computing**

Student: ALESSIO BARLETTA Discussed with: PABLO LANDROVE

Solution for Project 1

HPC Lab — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

In this project you will practice memory access optimization, performance-oriented programming, and OpenMP parallelizaton on the Rosa Cluster.

1. Rosa Warm-Up *(5 Points)*

1.1.

A module is a user interface which allows user to switch dev. Environments without having to manually modify their env. Variable (ex. path to the compiler ecc...). It is then used to allow a user to switch between different versions of installed programs and libraries. To use this feature we use the `module` command and to list all available module we can use `module avail`. A comprehensive list of other possible commands can be found on the Rosa's documentation.

1.2.

Slurm is an open-source cluster management and job scheduling system for Linux clusters. It has three different key functions. Firstly, it allocates exclusive access to resources to users for a specified amount of time, so that they can perfor work. It provides a framework for starting, executing, and monitoring work on the set of allocated nodes. Lastly, it arbitrates contention for resources by managing a queue of pending work.

1.3.

To print the hostname of the machine I used the command $gethostname(char * name, size\_tsize)$. I defined $HOST\_NAME\_MAX$ as 255, following the linux man documentation.

1.4.

To target nodes with specific memory I used the `--partition` flag in the sbatch command. I chose to do this because using `sinfo -o "%N %f"` we can see that no features were defined for any node, which makes the `--constraints` flag useless. Instead, using `sinfo -o "%P %N %m"` we can list the amount of memory dedicated to each node in every partition. Thus, by choosing the correct partition we can target node with different memories.

1.5.

The batch script for this exercise was already provided in the assignment, excluding the compilation part. By checking the output file we can see that the job ran on nodes $icsnode18$ and $icsnode19$.

2. Performance Characteristics *(30 Points)*

2.1.

The Rosa cluster comprises 42 compute nodes, as indicated by the Rosa's documentation (i.e., $n_{nodes} = 42$). Each node is equipped with 2 Intel Xeon E5-2650 v2 CPUs (i.e., $n_{sockets} = 2$), each CPU has 10 cores (i.e., $n_{cores} = 10$). Looking into the provided link cpu-world.com we can see that it supports both AVX2 SIMD instructions that use 256bit registers (i.e., $n_{SIMD} = 256/64 = 4$) and FMA3 instructions (i.e., $n_{FMA} = 2$). To get the superscalarity factor we can look at the provided uops.info website, there we can filter by architecture (Haswell) and type of instruction (FMA). We can then see that their throughput TP is equal to 0.5CPI, meaning the CPU can do 2 FMA instructions per cycle (i.e. $n_{FMA} = 2$). We can now compute the theoretical peak performance of the Rosa cluster.

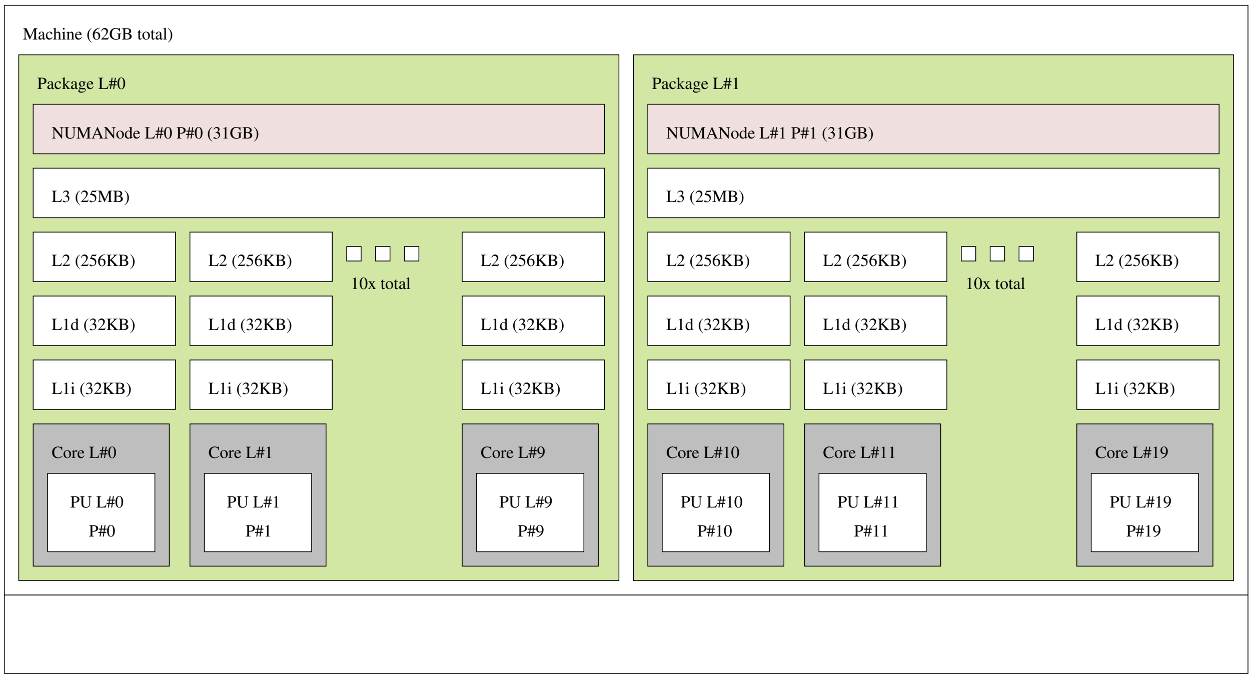
$$\begin{aligned}
P_{core} &= 2 \cdot 2 \cdot 4 \cdot 2.5 \text{ Ghz} = 36.8 \text{ GFlops/s} \\
P_{CPU} &= 10 \cdot P_{core} = 368 \text{ GFlops/s} \\
P_{node} &= 2 \cdot P_{CPU} = 736 \text{ GFlops/s} \\
P_{Rosa} &= 42 \cdot P_{node} = 30912 \text{ GFlops/s} = 30.912 \text{ TFlops/s}
\end{aligned}$$

2.2.

To identify the parameters of the memory hierarchy we can use the `lscpu` command, which provides detailed information about the CPU architecture. From this we obtain that the CPU model is an Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz, as expected from the Rosa's documentation. We can also use the `cat /proc/meminfo` to get an overview of the total available memory.

Using `hwloc` will provide us with how the meomory hierarchy is structured. Using the provided command we can easly have a graphical rappresentation of it and output it to a .fig file. Converting thi file to Pdf gives us a clear image of how the memory is structured.

We can see that each L1 and L2 caches are local among each core while L3 cache is shared among all cores. We can also see the total amount of main memory which differs from the one visualized in the assignment instruction image (31 GB vs 23 GB) per core. This is again consistent with the Rosa's documentation that states the login node has 64 GB of memory 31 GB per core. I could not figure out the discrepancy of 2 GB, but it could be due to approximation.



2.3.

To run the stream benchmark we need to precisely tune the size of the array to match the cache size of the system. These instructions are listed in the stream.c file.

The criteria are two:

1. Each array must be at least 4 times the size of the available cache memory. In our case we have 25 MB L3 cache, so each array need to be at least 100 MB. To do so we have to choose the appropriate $STREAM\_ARRAY\_SIZE$. Each index of array is double so 8 Bytes, so to have a size of 100 MB the minimum of $STREAM\_ARRAY\_SIZE = 100$ MB/8 Bytes = 12.5 M.

2. The second constraint is that the size should be large enough so that the 'timing calibration' output by the program is at least 20 clock-ticks. Linux usually has a granularity of 1 ms. this CPU has a max declared bandwidth of 68.3 GB/s. 20 tics at 1 ms/tic is equal to 20 ms. This means total array must be the size of 1.37 GB, so $STREAM\_ARRAY\_SIZE\_SUM =$ 1.37 GB/8 Bytes = 171.5 M. Since we have 3 arrays the minimum of $STREAM\_ARRAY\_SIZE = STREAM\_ARRAY\_SIZE\_SUM/3 = 57.17$ M.

Using the provided $STREAM\_ARRAY\_SIZE = 128M$ we are sure to satisfy the above criteria. The reuslt of the test are shown below:

```
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           19132.1     0.107136     0.107045     0.107302
Scale:          11221.1     0.182580     0.182513     0.182755
Add:            12258.6     0.250722     0.250600     0.250854
Triad:          12267.8     0.250536     0.250412     0.250873
-------------------------------------------------------------
```

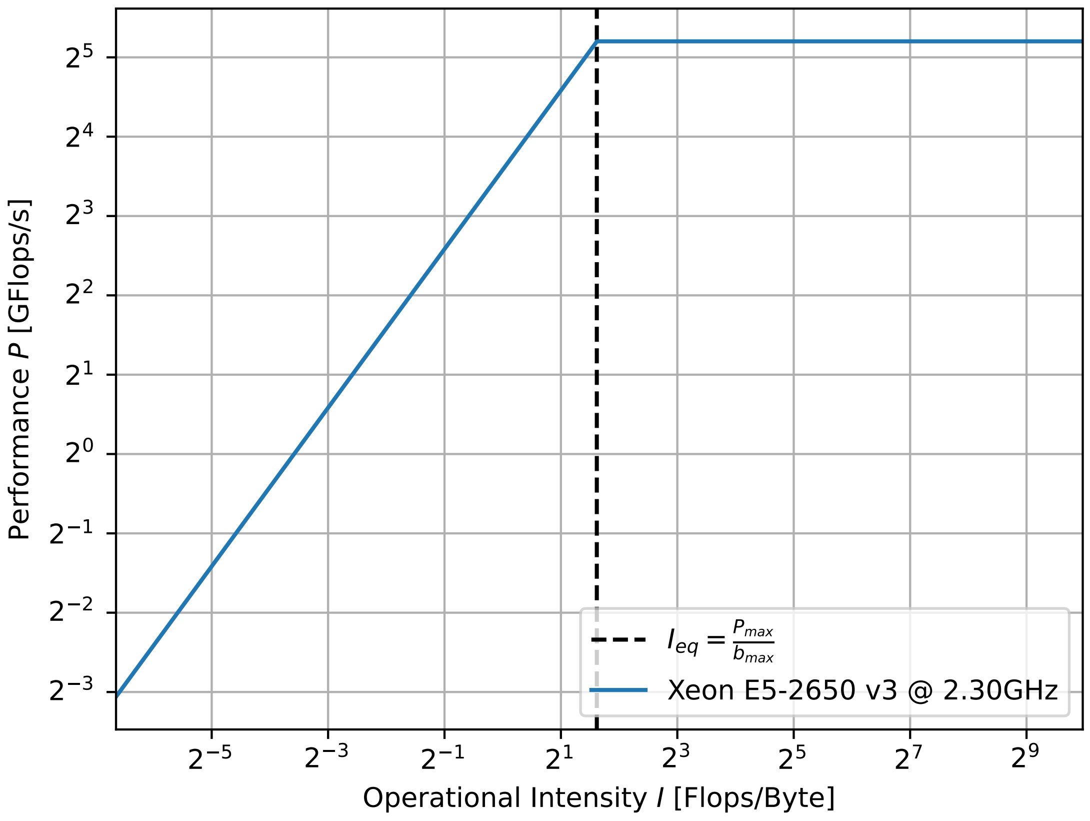
2.4.

To compute the roofline model we neeed the following parameters:

- P_{max}: Theoretical peak performance of the CPU. From section 2.1 we have $P_{max} = 36.8$ GFlops/s.
- b_{max}: Max memory bandwidth. From the STREAM benchmark we have $b_{max} = 12$ GB/s.

We also need to choose the range of operational intensity I to plot the roofline. We can choose $I_{min} = 10^{-2}$ Flops/Byte and $I_{max} = 10^3$ Flops/Byte.

The frontier between memory-bound and compute-bound is given by the intersection of the two lines, which is given by $I = P_{max}/b_{max} = 36.8/12 = 3.06$ Flops/Byte.



This means that every kernel with operational intensity lower than this 3.06 is memory-bound, otherwise it is compute-bound.

3. Optimize Square Matrix-Matrix Multiplication *(50 Points)*

To implement and optimize the Square matrix multiplication I followed the provided pseudocode but with a strong focus on memory access optimization.

In my first try I just tried to optimize the naive implementation by taking advantage of the fact that the matrices are stored in column-major order. This means that accessing elements in the same column is more cache-friendly than accessing elements in the same row. So I changed the order of the loops to iterate over the columns of the matrices first, instead of going from left to right, going from top to bottom.

This gave me a performance boost of around 5.4x compared to the naive implementation, going from an average peak performance of 2.88% to 15.6% of the theoretical peak performance.

The next step I took was to implement the Square Matrix-Matrix Multiplication. I allocated one block for each matrix and to do so, I used the `malloc` function, in this way each block would be contiguous in memory and when doing operations on the elements of the block, only the blocks would be loaded in cache, instead of trying to load the entire matrix. I also made sure to maintain the column-major order while copying the data from the original matrices to the blocks, so that I could take advantage of my first optimization.

The code for the optimized block multiplication is shown in Listing 1.

```c
for (int j = 0; j < j_max - jj; ++j)
  for (int k = 0; k < k_max - kk; ++k)
    for (int i = 0; i < i_max - ii; ++i)
      block_C[i + j * blockSize] +=
        block_A[i + k * blockSize] * block_B[k + j * blockSize];
```

Listing 1: Optimized block multiplication used in dgemm-blocked.c

This gave me a performance boost of around 2.1x compared to my previous implementation and a total boost of 11.5x compared to the naive implementation, going from an average peak performance of 2.88% to 33.1% of the theoretical peak performance.

Next I focused on choosing the optimal block size. To start, I calculated that the max block size that could fit in the L1 cache was 36, since each block is a square matrix of size $blockSize \times blockSize$ and each element is a double (8 bytes), the total size of a block is $blockSize^2 \times 8$ bytes. The L1 cache size is 32 KB, so we have:

$$blockSize^2 \times 8 \times 3 \leq 32 \times 1024 \implies blockSize \leq 36$$

However, a block size of 36 would not be optimal, since it would not be a multiple of the registers' size. For this reason I firstly tried a block size of 32, but with some trial and error I found that the optimal block size was 64. Moreover, any block size over 128 would not fit in the L2 cache (since the L2 cache size is 256 KB), so it would not be optimal:

$$\text{blockSize}^2 \times 8 \times 3 \leq 256 \times 1024 \implies \text{blockSize} \leq 104$$

During testing I had results that were consistent with this calculation, since the performance started to drop for block sizes over 128.

Finally, I tried more advanced techniques such as loop unrolling, using `restrict` keyword or allocating memory with `aligned_alloc` instead of `malloc`, but none of these gave me a significant performance boost, so I decided to keep my implementation as it is.

The various results of my implementations are shown in the respective .out file, and in the table below:

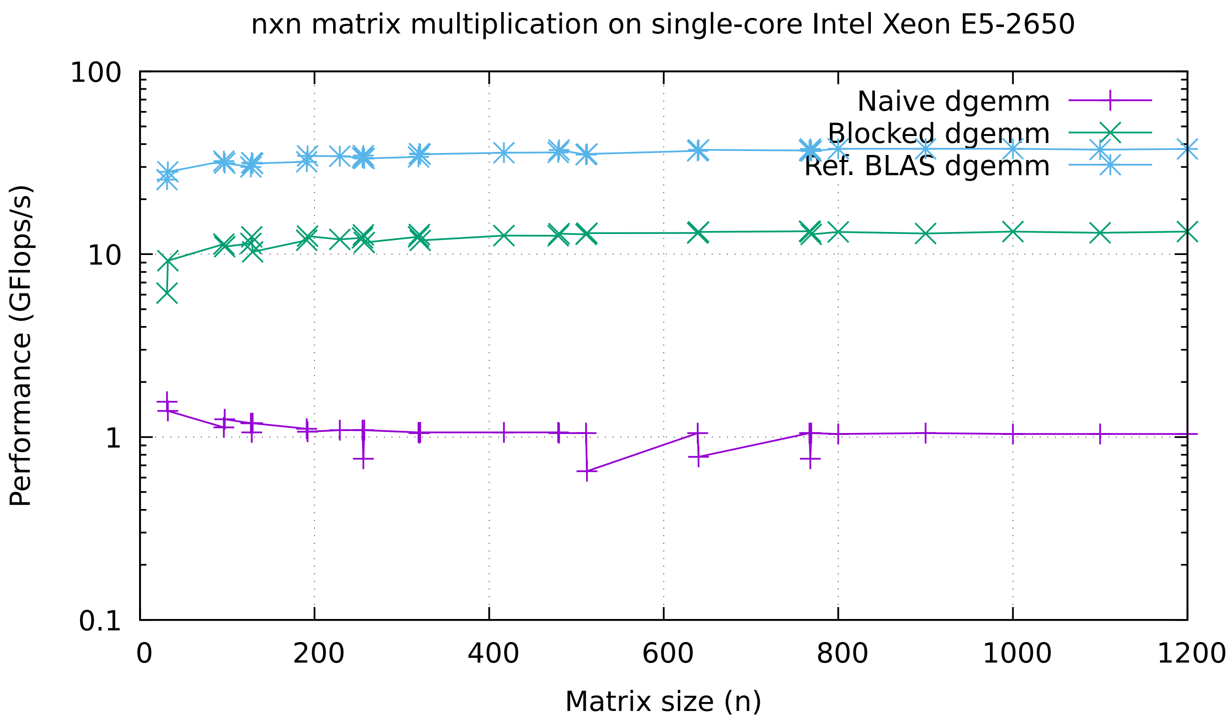
Method	**Average % of Peak Performance**	**Speedup vs NAIVE**
NAIVE	2.88057 %	1.00×
COLUMN MAJOR OPTIMIZED	15.6413 %	5.43×
BLOCKING_64	33.1122 %	11.49×
BLOCKING_32	29.2751 %	10.16×
ADVANCED OPTIMIZATION	24.517 %	8.51×

Table 1: Average percentage of peak performance and speedup relative to NAIVE method.

To check if my program was actually using vectorization and/or FMA instructions I produced the assembly code using the command: `gcc -S -O3 -march=native -ffast-math dgemm-blocked.c` and then `grep -A 20 "vmulpd |vfmadd" dgemm-blocked.s` allowed me to confirm that these instructions were actually being used in the assembly code.

The final version of my optimized function (the one present in the submission) was compiled using the following flags: `-O3 -march=native -funroll-loops -ftree-vectorize`, I also tried adding other flags such as `-ffast-math` or changing `-O3` to `-Ofast`, but they did not give me any significant performance boost.

The following graph illustrates the performance of the naive vs the BLAS implementation vs my optimized implementation with block size 64.



To write my code I used as reference the provided pseudocode and the LAFF-On Programming for High Performance online resource. Mainly for the optimization part, I referred to the C++ Reference online manual as well as lecture notes.

4. Quality of the Report *(15 Points)*