Università della Svizzera italiana

Institute of Computing CI

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

Student: Jonatan Bella Discussed with: NONE

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. Modules concept is used to allow the user to manage between different versions of software packages without needing to configure the corresponding environment variables that could affect every user of the cluster. For example, by calling *module load modulename/module-version*, the user can load the desired version of the software if available. To check availability is as simple as calling *module avail*. In addition to this commands, it is possible to check the loaded modules *module list*, unload all loaded modules *module purge*, among others. [1]

2. Slurm is a highly scalable cluster management and job scheduling system for Linux clusters (workload manager). As a manager it assigns access between nodes and users for certain time interval given a task (allocation). Furthermore, it provides a framework for starting, executing, and monitoring work (normally a parallel job) on a set of allocated nodes and it does prioritization of the tasks according the queued jobs. [2]

3. The code is provided under the folder "*exercise*1" with the name "*printhost*" with the cited sources used during the development inside the file.

[1] `https://www.ci.inf.usi.ch/research/resources/`
[2] `https://slurm.schedmd.com/overview.html`

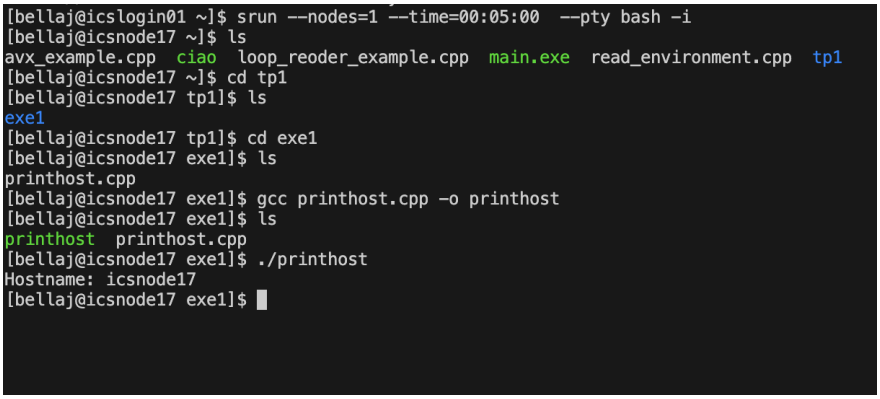

Figure 1: Terminal of Cluster print while testing the code

4. Here i implemented the batch command in the file *printhostbatch.sh* and the output is saved in the file *printhostbatch.txt*. Notice that in the instructions is recommended to use the –constraint option to specify an available feature, however the available features are not set "(*null*)". Therefore, i use the –partition option to specify the partition to run the job as an example.

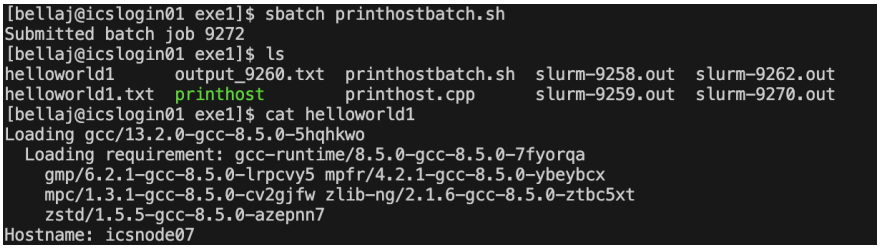

Figure 2: Terminal of Cluster print while testing the code

5. The file *printhostbatch2node.sh* contain the implementation of the batch script that runs the program on two nodes as requested:

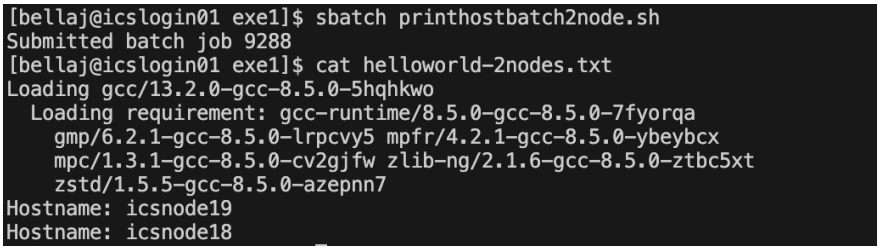

Figure 3: Terminal of Cluster print while testing the code

As in previous exercises, the sources utilized are cited in the file. [3]

2. Performance Characteristics *(30 Points)*

2.1. Peak performance

Following the example, I visit the USI Rosa cluster specifications website [4]. It is possible to find that each node has 2 Intel Xeon E5-2650 v3 (25M Cache, 2.3 GHz).

[3] At the moment I started the assignment the batch files were not included, reason why the naming is different to the provided later on and i added links that i used to understand how to construct such a file to the .sh document

[4] `https://www.ci.inf.usi.ch/research/resources/`

Knowing this, it was possible to find the processor specification in intel's website [5], where it is shown that the processor has 10 cores, a base frequency of 2.3 GHz ($f = 2.30$) and supports AVX2 SIMD instructions, as specified on "Instruction Set Extensions". Since AVX2 uses 256-bit wide vector registers, which can process 4 double-precision (64-bit) numbers at once, then $n_{SIMD} = 4$.

With respect to FMA instructions, on Chapter 15 - section 15 of Intel's Optimization Reference Manual Volume 1, it is mentioned that Haswell microarchitectures implements FMA instructions with execution units on port 0 and 1 and 256 - bit data paths. "Dot product, matrix multiplication and polynomial evaluations are expected to benefit from the use of FMA, 256 - bit data path and the independent executions on two ports. The peak throughput of FMA from each processor core are 16 single-precision and 8 double-precision results each cycle". Therefore, $n_{FMA} = 2$. In addition, it also tells us that since "...the independent executions on two ports", we can expect that the number of FP operations the FPU of a core can execute in parallel within a single clock cycle is 2 (n_{super}).[6]

With this information we can compute the peak performance of a single core:

$$\begin{aligned} P_{\text{core}} &= f \times n_{\text{SIMD}} \times n_{\text{FMA}} \times n_{\text{super}} \\ &= 2.3 \times 4 \times 2 \times 2 \\ &= 36.8 \text{ GFLOP/s} \end{aligned}$$

As i mentioned before, each processor has 10 cores, therefore each CPU has a peak performance of 368 GFLOP/s.

$$\begin{aligned} P_{\text{CPU}} &= n_{\text{cores}} \times P_{\text{core}} \\ &= 10 \times 36.8 \text{ GFLOP/s} \\ &= 368 \text{ GFLOP/s} \end{aligned}$$

Since in the cluster website it is specified that there are two sockets per node, then the peak performance of a node is 736 GFLOP/s:

$$\begin{aligned} P_{\text{node}} &= n_{\text{sockets}} \times P_{\text{CPU}} \\ &= 2 \times 368.0 \text{ GFLOP/s} \\ &= 736 \text{ GFLOP/s} \end{aligned}$$

Finally, it is also specified that there are 42 compute nodes in total in the cluster:

$$\begin{aligned} P_{\text{cluster}} &= n_{\text{nodes}} \times P_{\text{node}} \\ &= 42 \times 736.0 \text{ GFLOP/s} \\ &= 30.912 \text{ TFLOP/s} \end{aligned}$$

Which means that the peak performance of the cluster is 30.912 TFLOP/s.

2.2. Memory Hierarchies

I present each terminal output after following the instructions:

[5] `https://ark.intel.com/content/www/us/en/ark/products/81705/intel-xeon-processor-e5-2650-v3-25m-cache-2-30-gh` `html`

[6] It is also possible to check on `https://uops.info/table.html`, however the results are quite extensive even after filtering by Haswell architecture and instruction set categories.

Figure 4: CPU model and specificiations

Here we can observe the CPU model and specifications of the cluster. Which confirms the information we could obtain from the cluster website and the Intel's website.

We proceed to obtain the main memory information as indicated in the commands:

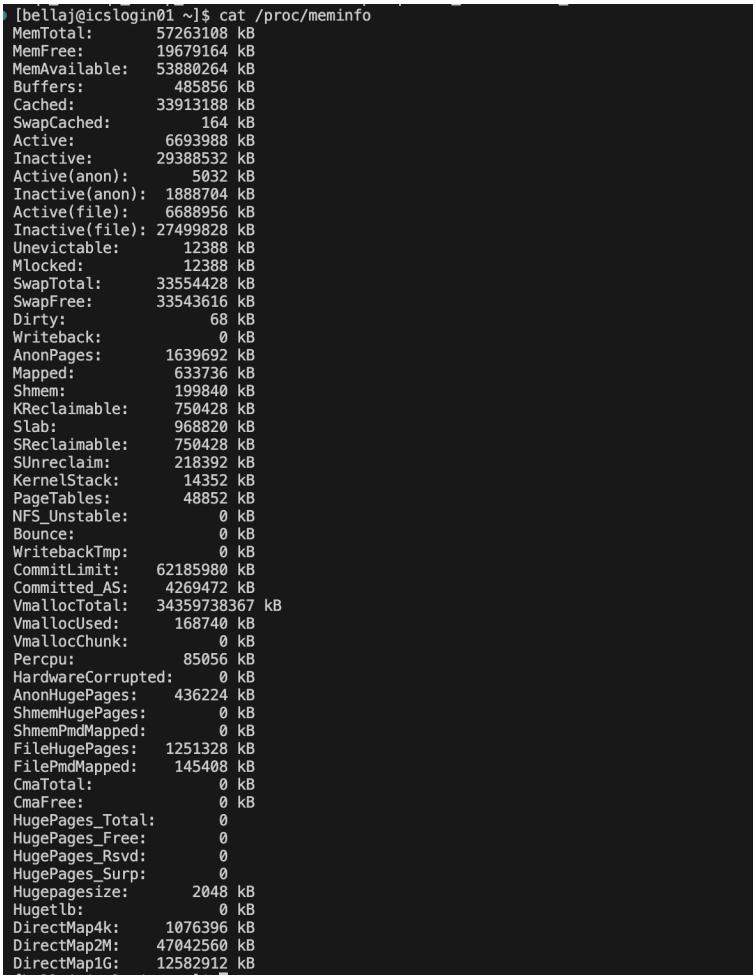

Figure 5: Main memory information

Now we proceed with the cache information:

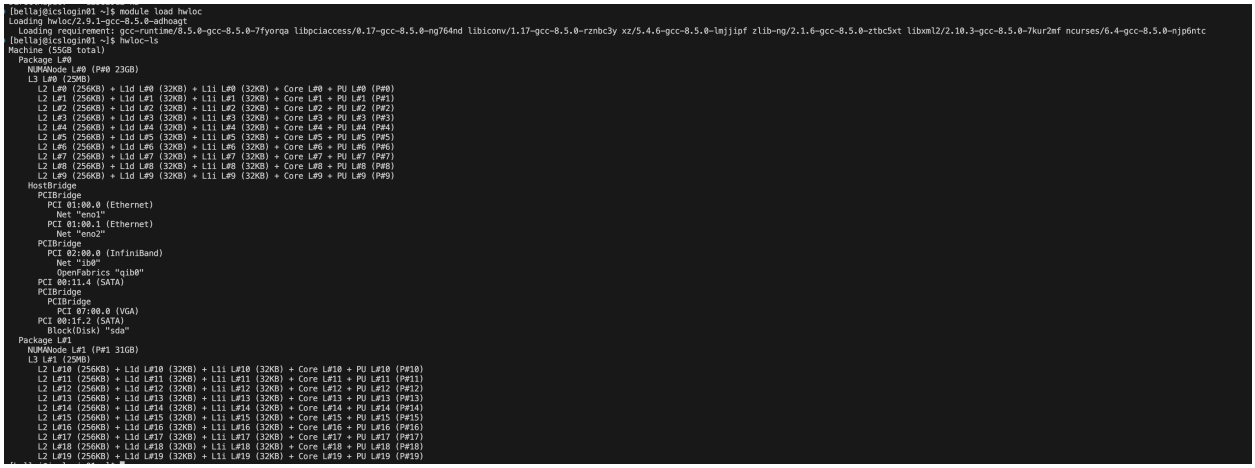

Figure 6: Cache memory information

Where it is possible to see that, since we are reporting about the login node, it has 55 GB of total memory, and it is composed of 2 "NUMA" nodes with 10 cores each. Its total memory is distributed such as 23 GB of RAM memory belonging to the first package and 31 to the second one. Then, each package has an L3 cache memory of 25 MB each, and it is shared among all the cores each of the packages. The L2 cache memory has 256 KB and L1 cache memory has 32 KB (for each L1i - instruction and L1d - data) on each package too.

This can be visualized in the downloaded figure as required:[7]

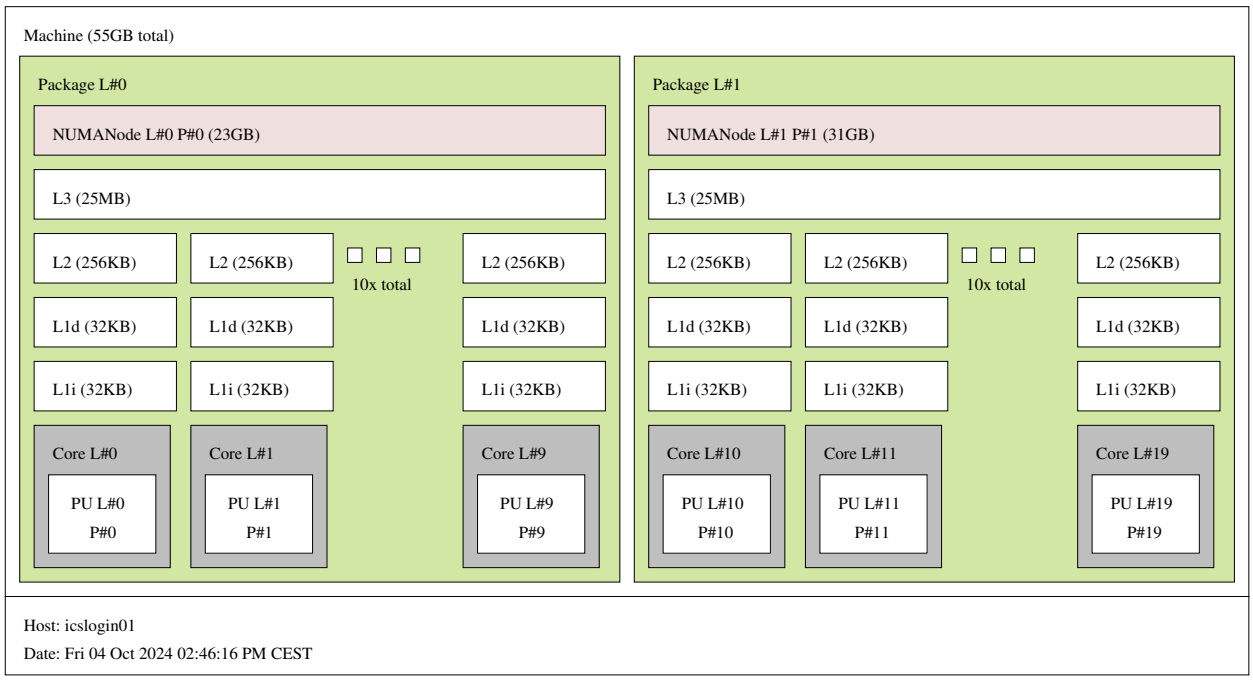

Figure 7: Resulting PDF figure from ROSA Cluster - memory Hierarchies

2.3. Bandwidth: STREAM benchmark

After downloading the proper files (stream.c and mysecond.c), I proceed with the calculations required to run properly the file. In particular, as it is mentioned in the instructions, each array must be at least four times the size of the L3 cache memory. Therefore, since each L3 cache memory is 25 MB, then the size of each array must be at least 100 MB. In line 176 of the stream.c file, it is

[7] I had troubles downloading it through the terminal with the provided command given permission issues, but I could download it through VS Code.

defined that the stream type is a double and since a double in C is 8 bytes (64 bits), then 100 MB / 8 = 12.5 M doubles (12500000). To approximate it into a power of two number, we can choose 12800000. Notice that this number is one magnitude smaller than the provided in the example, but it can be seen that both provide similar results, since, as calculated, the size of the array is at least four times the size of the L3 cache memory in both cases.

Now, I run the commands with the specified changes:

```
[bellaj@icslogin01 exe2]$ module load gcc
[bellaj@icslogin01 exe2]$ gcc -O3 -march=native -DSTREAM_TYPE=double -DSTREAM_ARRAY_SIZE=128000000 -DNTIMES=20 stream.c -o stream_c.exe
[bellaj@icslogin01 exe2]$ ls
mysecond.c  stream.c  stream_c.exe  stream.f
[bellaj@icslogin01 exe2]$ sbatch --mem-per-cpu=4G --wrap "./stream_c.exe"
Submitted batch job 9844
[bellaj@icslogin01 exe2]$ ls
mysecond.c  slurm-9844.out  stream.c  stream_c.exe  stream.f
```

Figure 8: Terminal commands on ROSA cluster to obtain the STREAM benchmark - same I did with 12800000

With the resulting benchmarks as shown in the file *slurm*9844.*out* and *slurm*9846.*out*:

```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 128000000 (elements), Offset = 0 (elements)
Memory per array = 976.6 MiB (= 1.0 GiB).
Total memory required = 2929.7 MiB (= 2.9 GiB).
Each kernel will be executed 20 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 117053 microseconds.
   (= 117053 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           19178.7     0.106838     0.106785     0.107000
Scale:          11218.7     0.182619     0.182553     0.182755
Add:            12294.0     0.249960     0.249878     0.250121
Triad:          12288.2     0.250086     0.249996     0.250206
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

Figure 9: STREAM benchmark results

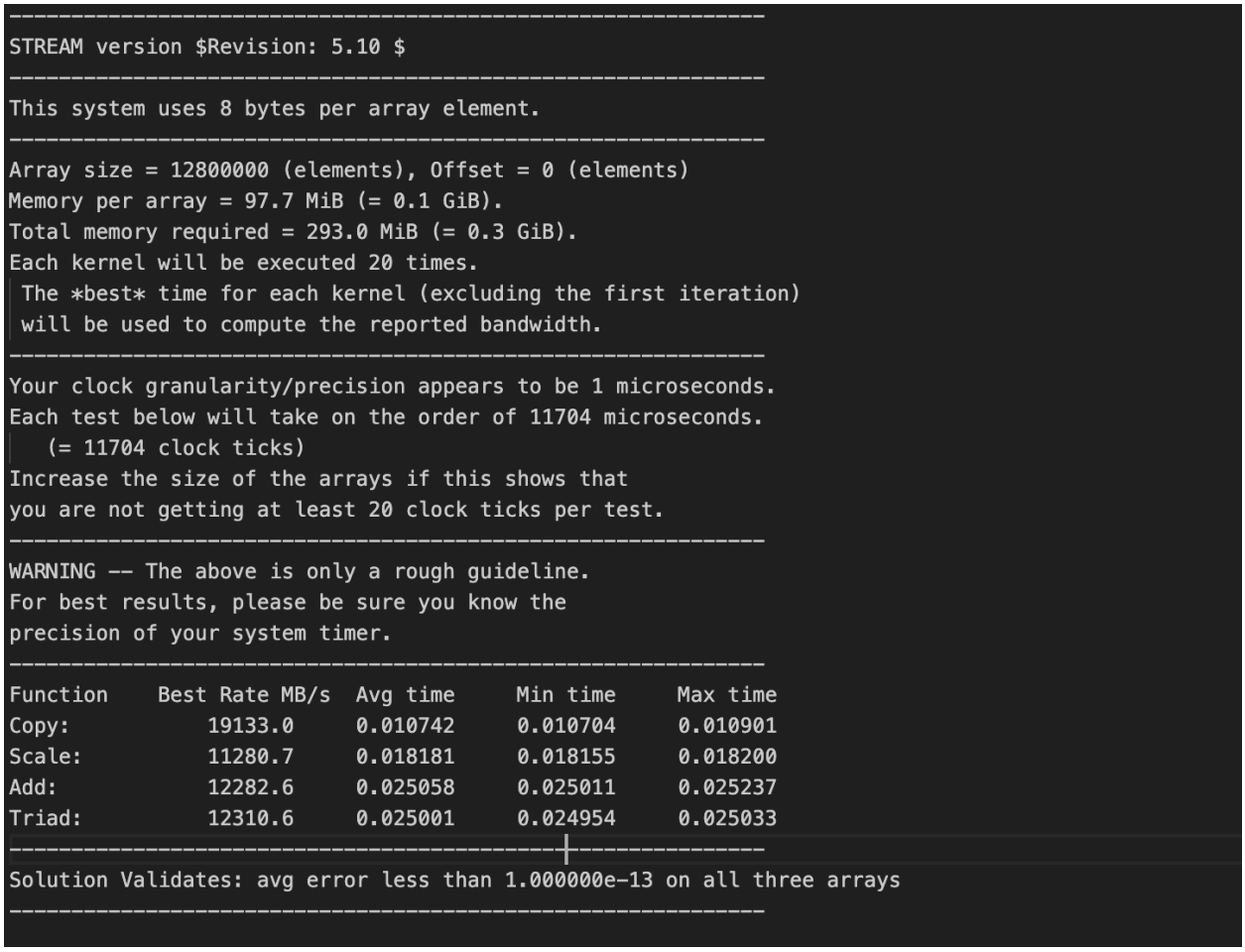

Figure 10: STREAM benchmark results - with 12800000 array size as calculated

2.4. Performance model: A simple roofline model

Let's put together all the information we gathered to build a simple roofline model as in the provided example. On one side, we know that the y-axis of the roofline model measures the attainable floating-point performance, as gigaflops per second (Giga Floating-Point Operations per Second - GFLOP/s). On the other side, the x-axis measures the operational intensity as flops per byte of DRAM traffic [8] (the number of floating-point operations per byte of data moved from memory to the processor - Operational Intensity / [Flops/byte]). Therefore, low operational intensity means that the system is moving a lot of data but doing little computation - memory bound. whereas high operational intensity means that the system is performing a lot of computations with the data it moves.

As we computed in 2.1, we know that the theoretical peak performance of a core with the described characteristics in these exercises is 36.8 GFLOP/s. Whereas, using the same approach that in the provided example of taking the STREAM benchmark results of Scale, Add, Triad (treating Copy as an exception), we can average the results of these three:

$$\begin{aligned}\text{Average maximum bandwidth} &= \frac{11280.7 + 12282.6 + 12310.6}{3} \\ &= 11957.97 \text{ MB/s} \\ &= 11.96 \text{ GB/s}\end{aligned}$$

[8] Based on Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM, 52(4):65, April 2009. `https://dl.acm.org/doi/10.1145/1498765.1498785` and class slide.

Following the assignment explanation and the mentioned paper, the sloped line in the graph represents the memory bound region and by consequence, the performance there is restricted by how fast the system can move data between memory and the processor (bandwidth - slope). Then, the machine balance is the operational intensity at which the memory bandwidth is saturated. This is the point where the performance is limited by the memory bandwidth and the peak performance of the processor is reached, the memory bound region meets the peak performance line. To put this into math terms since it makes it easier to understand:

$$\begin{aligned}\text{Attainable GFlops/sec} = \min(&\text{Peak Floating-Point Performance},\\ &\text{Peak Memory Bandwidth} \times \text{Operational Intensity})\end{aligned}$$

By consequence, on the left side of the graph, the expression summarize that the attainable performance is proportional to how fast it can transfer data from memory (bandwidth) and the FLOPs it can perform per byte of data (Operational Intensity). Another way of seeing this is that for every additional FLOP per byte of data transfer, the performance increases by a factor equal to the memory bandwidth.

Now, in the compute-bound region, the performance is capped by the peak core performance (independently of the data transfer)

$$\text{Attainable performane (GFLOP/s)} = \text{Peak performane (GFLOP/s)}$$

Then, the ridge, as defined in the assignment, is the point where the two regions meet:

$$\begin{aligned}\text{Peak performance (GFLOP/s)} &= \text{Peak Memory Bandwidth (GB/s)} \times \text{OI (FLOP/Byte)}\\ 36.8\ \text{GFLOP/s} &= 11.96\ \text{GB/s} \times \text{Operational Intensity (FLOP/Byte)}\\ \text{Operational Intensity (FLOP/Byte)} &= \frac{36.8\ \text{GFLOP/s}}{11.96\ \text{GB/s}}\\ &= 3.08\ \text{FLOP/Byte}\end{aligned}$$

Finally, the ridge point is at 3.08 FLOP/Byte. The following graph summarize the calculations as required: [9]

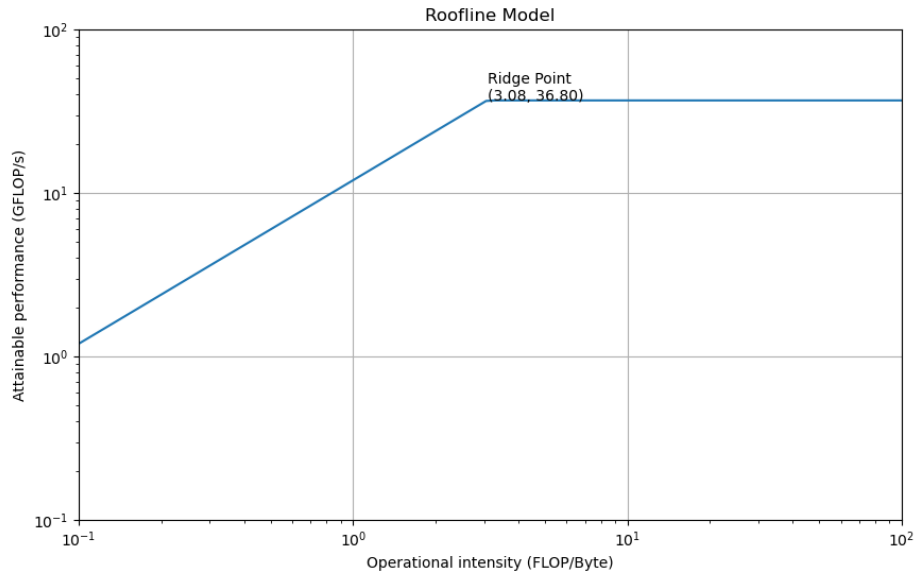

Figure 11: Resulting Roofline plot

[9] The code of the plot is named *plot.py* and the resulting plot is named *roofline_plot.png*

[10]

In conclusion, given the data, kernels with operational intensity with less 3.08 FLOP/Byte will benefit the most from optimizations that improve memory access patterns or reduce memory traffic by restructuring loops for unit stride accesses, ensuring memory affinity, and using software prefetching. [11] Whereas, kernels with operational intensity with more than 3.08 FLOP/Byte could improve its computational efficiency by improving instruction-level parallelism (ILP) and applying SIMD.

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

3.1. Blocked matrix multiplication - Optimization

To implement blocked matrix multiplication I started by computing the maximum size of each block such that it fits on the L3 cache memory. [12]

Since the L3 cache memory is 25 MB which is equivalent to 26214400 bytes [13]

Then the maximum size of each block can be calculated as $\sqrt{\frac{26214400}{3\times 8 \text{ bytes}}}$ since we are working with 3 matrices (A, B, C) that are symmetric and that contains double values (8 bytes). Therefore, the $(block\_size)^2 = (\frac{26214400}{3\times 8 \text{ bytes}})$.

After this calculation, on the main function *square_dgemm*, I implement the block as follows:

First, I check if the block size is bigger than the matrix size, since in that case then we should store the matrix completely in L3 and avoid indexing errors. Then, each of the three loops increments by calculated block size, taking into account the edge case that emerge when one of the loops block jumps can surpass the length of the matrix. If that occurs, the faction that computes the actual multiplication will receive the appropriate size of the corresponding block to not surpass the matrix size when indexing.

Most importantly, the order of indexing follows the column-major order as specified in the assignment. Citing from the main class book [14] "If an inner loop variable is used as an index to a multidimensional array, it should be the index that ensures stride-one access".

[10] I used python to generate the graph since, according to what I could found about graph generation in C++, the most common libraries used as matplotlib-cpp calls for python during compilation anyway. `https://stackoverflow.com/questions/63667255/plotting-graphs-in-c`

[11] Based on mentioned paper.

[12] I wrote the majority of my code based on the pseudocode from the last slide of HPC - "Blocked matrix multiplication", Chapter 3 of Georg Hager and Gerhard Wellein. Introduction to high performance computing for scientists and engi- neers; and the example code provided about AVX on previous to last class.

[13] https://www.gbmb.org/mb-to-bytes

[14] Introduction to High Performance Computing for Scientists and Engineers by Georg Hager and Gerhard Wellein, chapter 3, page 70

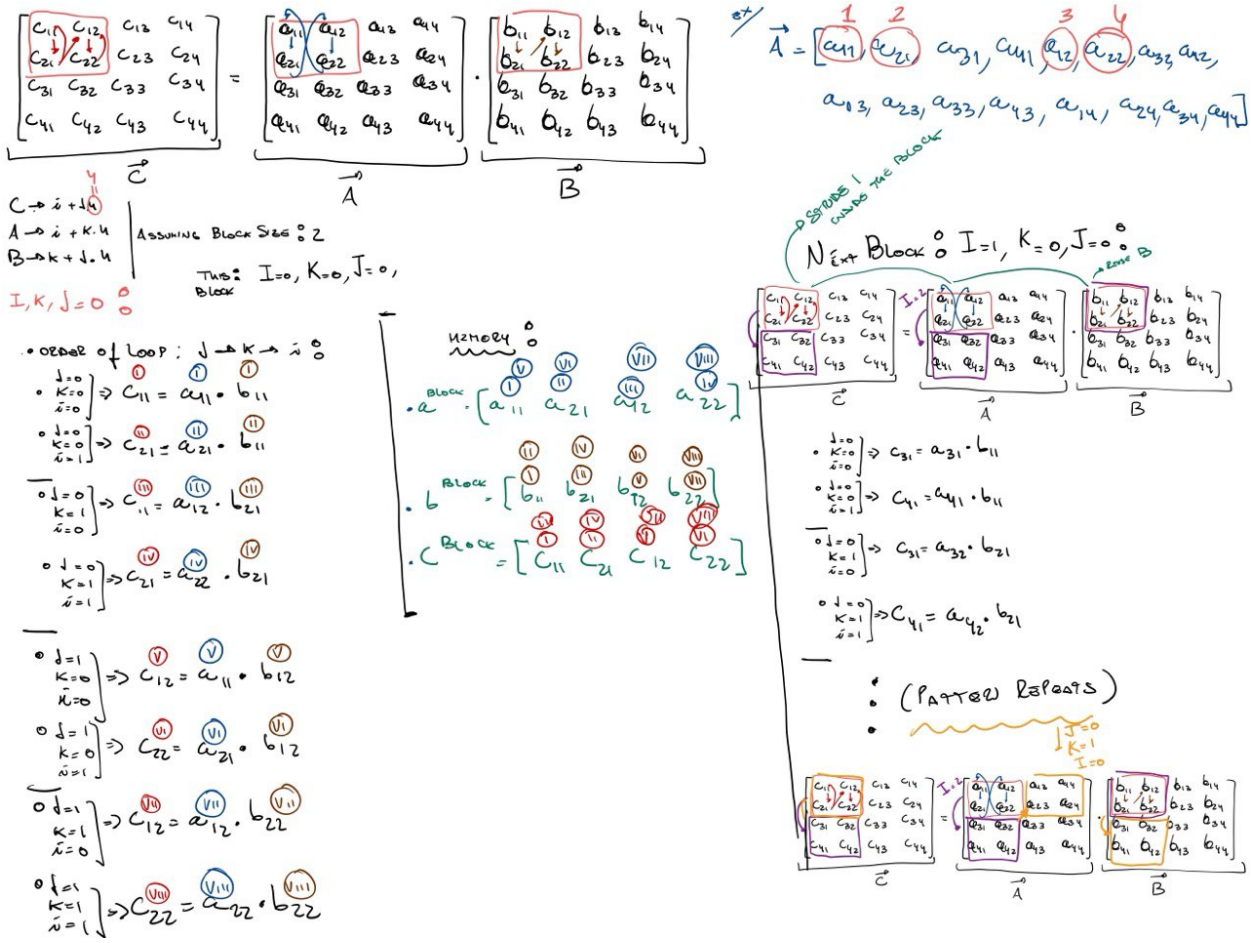

Figure 12: An illustration and summary of how I thought the implementation

Since we pass a pointer with the address of the beginning of each matrix (double* A , as we saw in previous assignment) then, we need to move in each block iteration the pointer to the right position of the block. This means, as an example, to take the first block, process it in the way I will describe below and then move the pointer to the beginning of the next block.

The block multiplication function call *matrix_multiply_block* receives the matrix size, the block size of each matrix (by debugging the edge cases), and the matrices. Following again the column major order as described in the book, which means that if a matrix, for example, of 3x3 is store in memory as:

$[a11, a21, a31, a12, a22, a32, a13, a23, a33]$

To index for a particular position, we can think of "jumping" to the right column j*n and then move I step into the column row that is expected. However, we need to do this in a way that we can access the memory efficiently. Remembering how our matrices are stored, then we should make operations such that each movement is done as going down in rows into a column and then moving to the next one such that I access the C and A consecutive elements in column (stride one access) while keeping B elements constant (same element - repeatedly) until that element will not be necessary anymore for operations.

As we can see in my handwritten illustration (Figure 12), which shows the process that my implementation follows and its benefits over the naive implementation, we can see that takes into advantage two main things:

Loading a block into cache once and reusing it multiple times - temporal locality. As it is the case with the B matrix. And stride one access to the elements in the corresponding block of A and C - spatial locality. Iterating on each block in the column-major order.

This provided a substantial increase over the naive implementation but still far away with the provided by blas library.

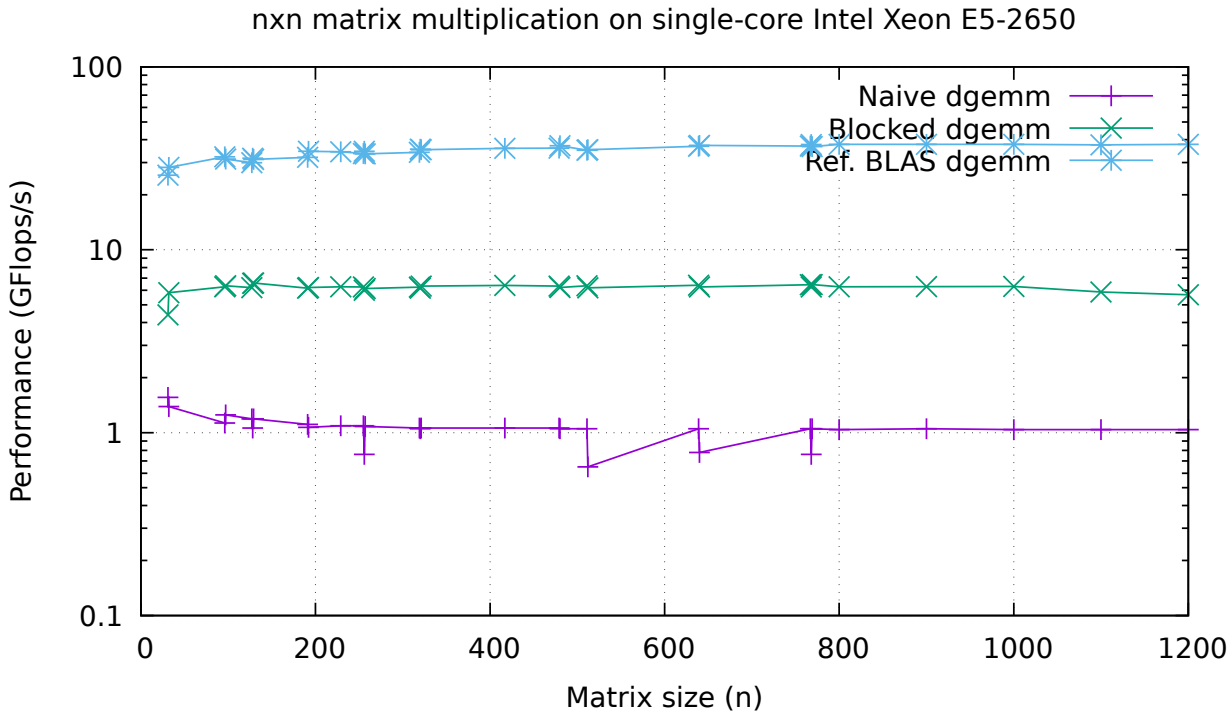

Figure 13: Performance of the different implementations - Average percentage of peak performance = 16.8045

However, there is also an additional slight improvement exist if we choose to keep fixed C block instead of B as we saw in class. This means that the loop order of the blocks is changed to J - I - K, where K is the innermost one. Given that in my code K index for the columns of A and J for the rows of B (as we saw in last class example, slide 44 - "*blocked_matrix_multiply*"). Notice that for the multiplication itself, the loop order does not change since we need to keep stride one access for column-major.

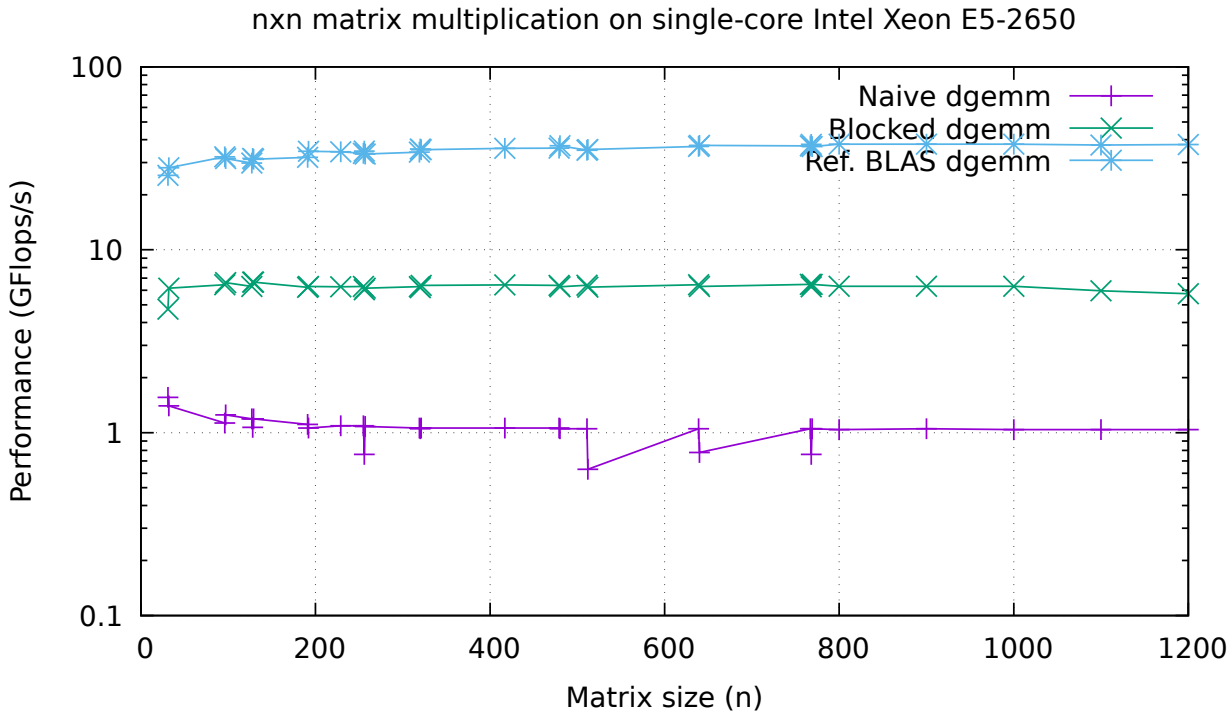

Figure 14: Performance of the different implementations - Average percentage of peak performance = 17.0158

The corresponding codes can be found in the folder *exercise*3, where, in *block* folder it is the B fixed block implementation of the illustration whereas in $blockCfixed$ can be found the C fixed implementation.

3.2. AVX optimization

Considering these results, I tried to improve my performance by using the AVX instructions. [15]

Notice that the same illustration (Figure 12) allow us to visualize how to construct this. As i mentioned before, i constructed *square_dgemm* such that it is not necessary to modify it later on, since the later optimization can be implemented directly on the multiplication related loops, not on the blocks.

With respect to *matrix_multiply_block*, the loops are almost unchanged, however, there are two main differences. Since we can process 4 doubles at a time with AVX (256-bits), we need to take this into account. First, as i noted when i built the previous one, B (or C - with loop order change as in previous implementation) matrix element will be repeated until passing to the next j or k index. This pattern emerges since the independence of i which is the inner loop. Then, we can broadcast the value of B (or C) into 4 and keeping it constant through the loop over C and A sub-blocks respectively.

The second main difference, is that we need to account for 4 loads at once in the internal loop. Which means that we loop over the index on jumps of 4. Which also means that we should account for the edge case where the block is not divisible by 4. This explains why I define i outside I loop, such that when the "expected" case is finish, it keeps the value where it finishes and pass to the next loop that will manage that fragment of maximum 3 elements each in the "normal" way. Notice that the loading of the C and A value follows the same that in the class example of AVX. The difference here is on $\_mm256\_fmadd\_pd$, since we need to multiply A and B sub-blocks and then add the result to the C sub-block. The function that does this, thanks to ETH slide set [16], is the

[15] For this implementation i based on the following documents: Class code example on avx, `https://acl.inf.ethz.ch/teaching/fastcode/2021/slides/07-simd-avx.pdf`, and `http://const.me/articles/simd/simd.pdf`

[16] https://acl.inf.ethz.ch/teaching/fastcode/2021/slides/07-simd-avx.pdf

previously mentioned one. Finally, I store the results in the same manner as class example.

It is finally not that exiting to report that the result was not as expected, since the implementation provided the same CPU usage as the previous one. I am still thinking why this could be. I suspect that in part I could be using the wrong compiling configuration:

Listing 1: Makefile example

```
CC = gcc                                # compiler (& linker)
OPT = -O3 -march=core-avx2 -mavx \
      -funroll-loops      \
      -ftree-vectorize                  # optimization flags (you may add more)
CFLAGS = -Wall -std=gnu99 $(OPT) # standard compiler flags
```

I tried using the commented Intel config for this, however, I keep getting an error related to not being able to load icx or icc. I tried to compensate this by adding the flag -mavx as it was in the AVX class example but still I do not see any improvement, which could also mean that my AVX implementation is not correct. [17]

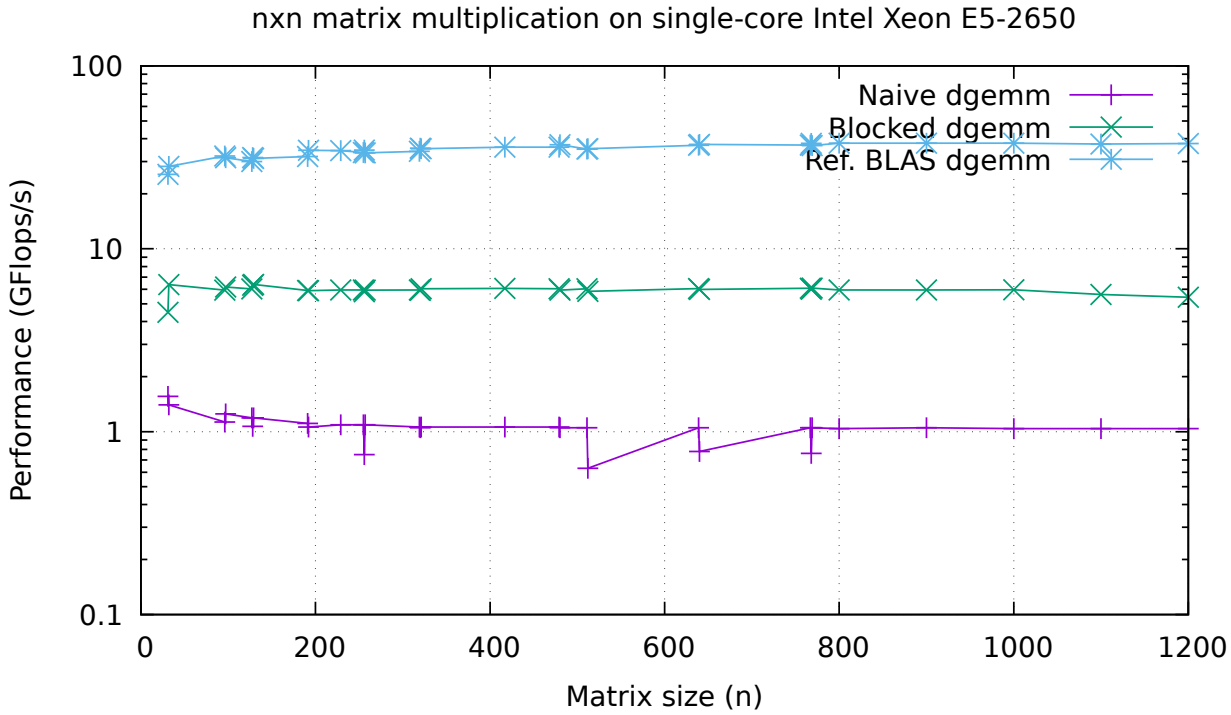

Figure 15: Performance of the different implementations with AVX - Average percentage of peak performance = 16.1474

The corresponding code can be found in the folder *exercise*3, where, in avx subfolder you will find the AVX related code whereas in block the original blocked matrix multiplication implementation. The "*provided*" subfolder contains the rest of the code provided by the assignment.

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

[17] To run this code, I worked in a separate file call *dgemmblockedavx*, however, for running I copy and paste it inside the *dgemmblocked* file to not modify the *run_matrixmult.sh*.