Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High-Performance Computing Lab for CSE **2024**

Student: Yannick Ramic Discussed with: Patrick Dowd, Carla Zurita, Arsh Kumbhat

Solution for Project 1

Due date: 11 March 2024, 23:59

HPC Lab for CSE 2024 — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to Moodle (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.

1. Euler warm-up [10 points]

This answers come from the documentation website, hence I won't cite anything properly.

1.1. What is the module system and how do you use it?

While working on the cluster it's necessary to use software or work with programming languages like C++ or Python. To make use of the cluster and use it's potential to work with accelerated GPU and CPU, these dependencies need to be actually installed on the cluster. Thus, to configure the desired environment with a prefered software version, the cluster makes use of modules. For the EULER cluster, two types of modules exist. LMOD modules, which use a hierarchy of modules with three layers (Core -, Compiler - and MPI layer, according to the illustration on the documentation webpage). On the other hand, environment modules have the advantage to make use of pre-installed dependencies by updating the desired software stock. More important information, users are able to install additional applications in their home directory, also there are useful commands to check whether a dependency is installed.

1.2. What is SLURM and its intended function?

SLURM is a workload manager for the management of compute jobs on high-performance computing clusters. It's important to know that users can solely use the cluster ressources through the batch system. During the lecture it was made clear that for each job submitted we need ro make a request at the cluster, which is done by the described batch system! Thus a job submission command consists of three parts:

1. sbatch (SLURM submit command)
2. SLURM options (requesting resources and job-related options)
3. Job (computing job to be submitted)

To summarize, the intended function is to efficiently delegate and utilize computing resources. Further information about how to submit a job, a parallel job, a GPU job, as well as, how to monitor a job and see the job output can be found on the EULER cluster documentation online!

1.3. Write a simple "Hello World" C/C++ program which prints the host name of the machine on which the program is running.

You can find the corresponding C++ file under the name `hello_world.cpp`.

1.4. Write a batch script which runs your program on one node. The batch script should be able to target nodes with specific CPUs

The resulting files are `slurm_job_one-49284262.out` and `slurm_job_one-49284262.err`.

1.5. Write another batch script which runs your program on two nodes

The resulting files are `slurm_job_two-49289041.out` and `slurm_job_two-49289041.err`.

2. Performance characteristics [50 points]

2.1. Peak performance

This task requires the computation of the core, CPU, node and cluster peak performance for the Euler VII (phase 1 and 2) nodes. Relevant information from the provided webpages can be found summarized in table 1. Also from the exercise sheet we get the following formulas for calculating the cluster peak perfromances:

$$\begin{aligned} P_{core} &= n_{super} \cdot n_{FMA} \cdot n_{SIMD} \cdot f \\ P_{CPU} &= n_{core} \cdot P_{core} \\ P_{node} &= n_{sockets} \cdot P_{CPU} \\ P_{cluster} &= n_{node} \cdot P_{node} \end{aligned}$$

In addition I want to note a few things. The value from n_{SIMD} comes from the fact that both AMD processors support AVX2 SIMD instructions with 256-bit wide vector registers. Since we are dealing with double-precision FP numbers the size of the vector register needs to be divided by 64 bits, this will result in the n_{SIMD} value. Also from the given optimization manual for each processor unit we have to get the information if FMA (Floating Multiply Add) is provided. Since both processors support FMA, the fused multiply-add factor $n_{FMA} = 2$ otherwise it would just be 1 and the multiplication and addition can't happen in a single operation. f is the base clock frequency and key characteristic of each CPU. In addition, it should be mentioned that a good approximation for the duration of one clock cycle is given by 1/f. This number helps measuring the execution time of instructions.

From table 1 we get the following theoretical values for the Euler VII Phase 1:

$$\begin{aligned} P_{core} &= 2 \cdot 2 \cdot 4 \cdot 2.6 GHz = 41.6 GFlops/s \\ P_{CPU} &= 64 \cdot 41.6 GFlops/s = 2662.4 GFlops/s \\ P_{node} &= 2 \cdot 2662.4 GFlops/s = 5324.8 GFlopts/s \\ P^{(1)}_{EulerVII} &= 292 \cdot 5324.8 GFlops/s = 1554.8416 TFlops/s \end{aligned}$$

Parameter	Phase 1	Phase 2
n_{nodes}	292	248
$n_{sockets}$	2	2
n_{cores}	64	64
n_{super}	2	2
n_{FMA}	2	2
n_{SIMD}	4	4
f in [GHz]	2,6	2.45

Table 1: CPU Parameter

Analog for the Euler VII Phase 2 these are the results:

$$\begin{aligned} P_{core} &= 2 \cdot 2 \cdot 4 \cdot 2.45 GHz = 39.2 GFlops/s \\ P_{CPU} &= 64 \cdot 39.2 GFlops/s = 2508.8 GFlops/s \\ P_{node} &= 2 \cdot 2508.8 GFlops/s = 5017.6 GFlopts/s \\ P^{(2)}_{EulerVII} &= 248 \cdot 5017.6 GFlops/s = 1244.3648 TFlops/s \end{aligned}$$

2.2. Memory Hierarchies

The result of the command lscpu can be summarized in the following table:

Information	**Resulting Output**
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	4
On-line CPU(s) list	0-3
Thread(s) per core	1
Core(s) per socket	4
Socket(s)	1
NUMA node(s)	1
Vendor ID	GenuineIntel
CPU family	6
Model	71
Model name	Intel(R) Xeon(R) CPU E3-1284L v4 @ 2.90GHz
Stepping	1
CPU MHz	1403.802
CPU max MHz	3800.0000
CPU min MHz	800.0000
BogoMIPS	5799.77
Virtualization	VT-x
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	6144K
L4 cache	131072K
NUMA node0 CPU(s)	0-3

Table 2: Result of the command $ lscpu

The total available main memory with the command cat /proc/meminfo is:

$$MemoryTotal = 32871604 kB$$

Now we want to gain information about the memory hierarchy, which can be achieved by the following command $ hwloc-ls –whole-system –no-io the result is the same as depicted in the exercise sheet. In the end I want to obtain the figure and all necessary commands are described as well in the exercise sheet. The resulting hierarchy is depicted in figure

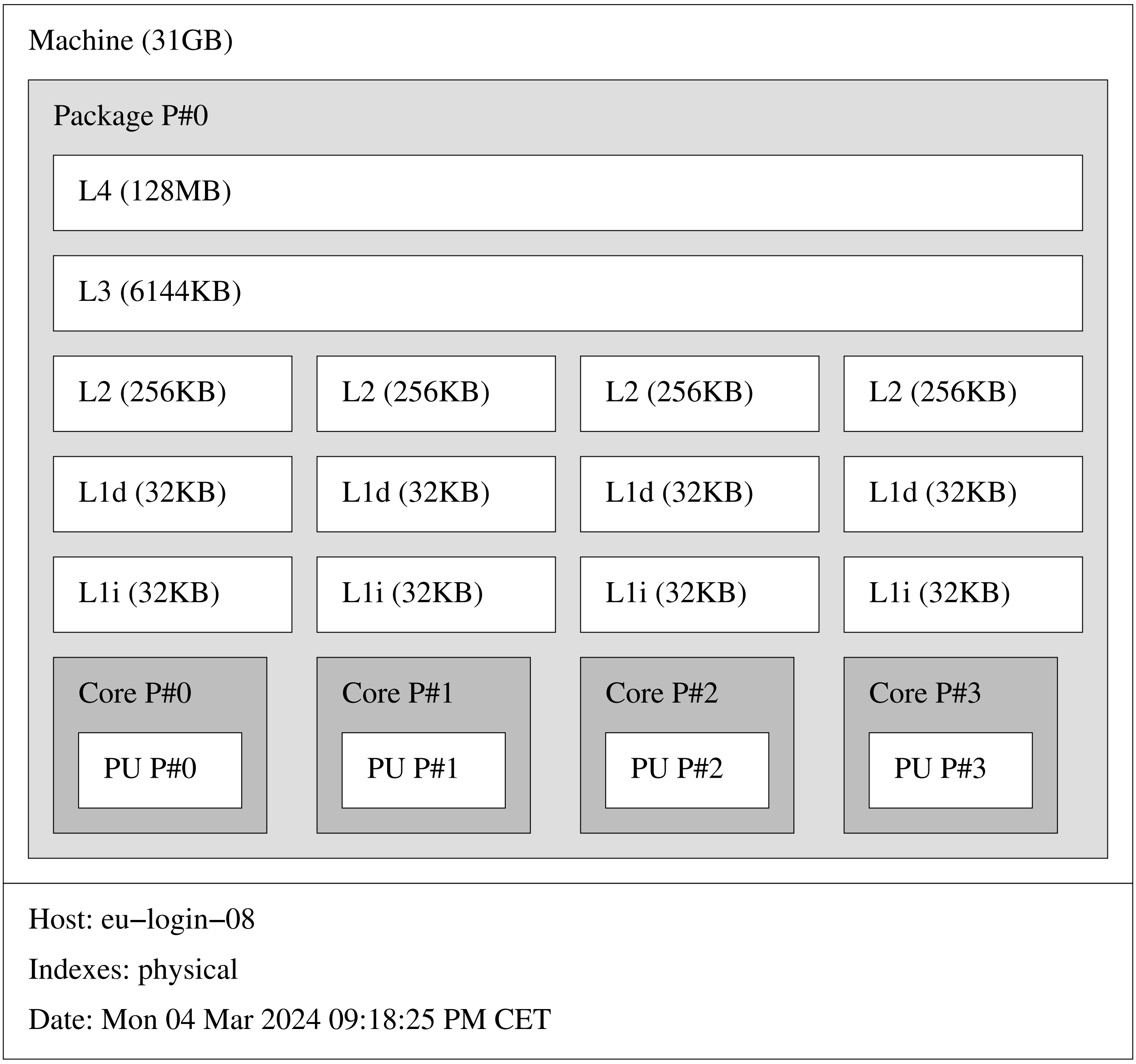

Figure 1: Schematic of an Euler login node with an Intel(R) Xeon(R) CPU E3-1248L v4 2.90GHz.

While the setup for the Intel(R) Xeon(R) CPU E3-1248L v4 2.90GHz is rather simple, lets summarize the result from Figure 2 and 3 in the following Table.

	AMD EPYC 7H12	**AMD EPYC 7763**
Main memory	31 GB	31 GB
L3 cache	16 MB	32 MB
L2 cache	512 KB	512 KB
L1 cache	32 KB	32 KB

Table 3: This table summarizes the information of one node for the underlying processor memory hierarchy.

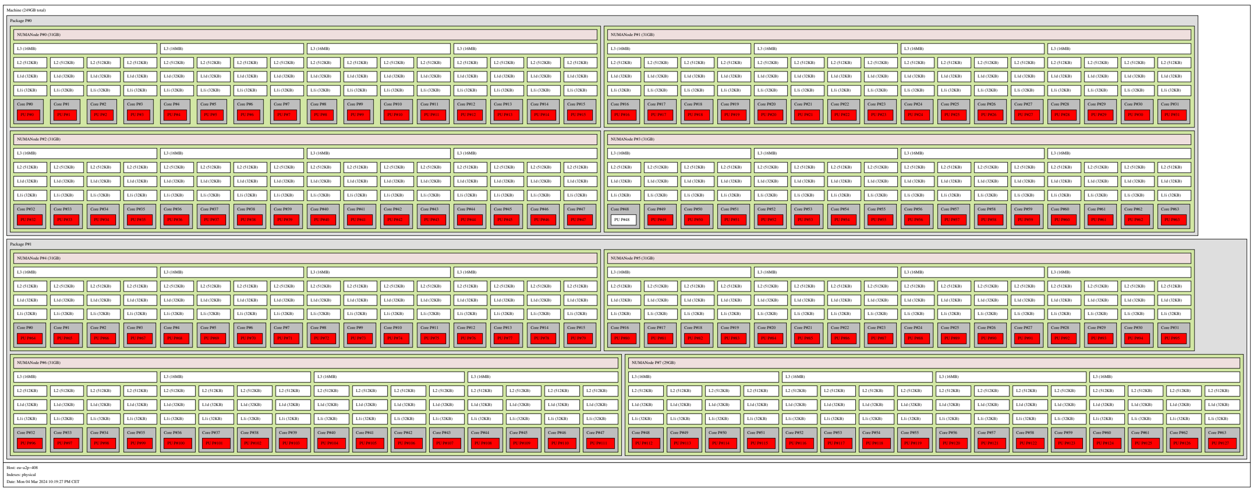

Figure 2: Schematic of an Euler login node with an AMD EPYC 7H12 CPU.

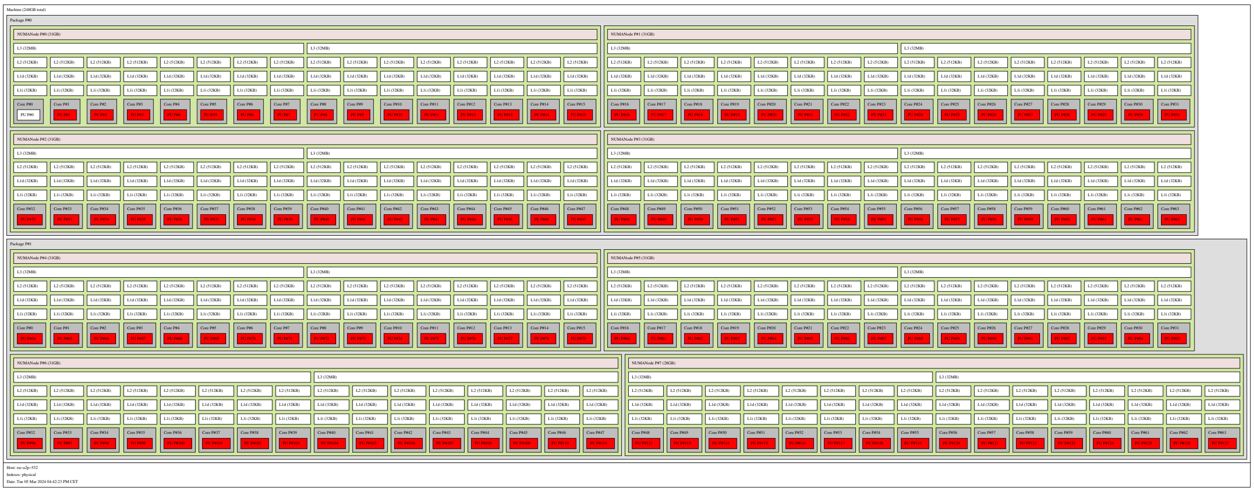

Figure 3: Schematic of an Euler login node with an AMD EPYC 7763 CPU.

2.3. Bandwidth: STREAM benchmark

For the STREAM benchmark we can use the provided file `stream.c`, where we need to adjust the STREAM_ARRAY_SIZE Parameter in the C-file for our system. In particular our two systems are, the Euler VII Phase 1 with the AMD EPYC 7H12 CPU and for Euler VII Phase 2 with the AMD EPYC 7763 CPU. According to the `stream.c` file provided by "Dr. Bandwidth", we have to calculate the STREAM_ARRAY_SIZE by considering the following two criteria.

1. The first criteria is that each array must be at least 4 times the size of the available cache memory. Note that it is important to considering from Figures 2 and 3 only one node.
2. The size should be large enough so that the 'timing calibration' output by the program is at least 20 clock-ticks. Furthermore, according to `stream.c` most versions of Windows have a 10 millisecond timer granularity. This assumption doesn't hold true for the euler cluster!

Retrieving the results from the previous task, namely Table3, we can see that we get for Euler VII Phase 1 the following results:

$$\bar{L}1 = 4 \cdot 32KB = 128KB$$
$$\bar{L}2 = 4 \cdot 512KB = 2.048MB$$
$$\bar{L}3 = 4 \cdot 16MB = 64MB.$$

While we would get for Euler VII Phase 2 these results:

$$\begin{aligned}\bar{L}1 &= 4 \cdot 32KB = 128KB\\ \bar{L}2 &= 4 \cdot 512KB = 2.048MB\\ \bar{L}3 &= 4 \cdot 32MB = 128MB.\end{aligned}$$

By respecting the second criteria the chosen parameter values can be extracted from Figure 4 and 5.

```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 8500000 elements, Offset = 0 elements
Memory per array = 64.8 MiB = 0.1 GiB.
Total memory required = 194.5 MiB = 0.2 GiB.
Each kernel will be executed 20 times.
 The  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 5301 microseconds.
   = 5301 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:           26351.2     0.005233     0.005161     0.005424
Scale:          17845.9     0.007694     0.007621     0.007945
Add:            19750.2     0.010414     0.010329     0.010478
Triad:          20116.1     0.010250     0.010141     0.010537
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

Figure 4: STREAM benchmark result for Euler VII Phase 1

As done in the exercise sheet for a rough estimate we can assume for the Euler VII Phase 1 a maximum bandwidth b_{STREAM} = 17 GB/s, which corresponds to the scale function. Same thing is possible where the Euler VII Phase 2, where we can retrieve from Figure 5 a maximum bandwidth value of b_{STREAM} = 27 GB/s, which corresponds to the add function.

```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 17000000 elements, Offset = 0 elements
Memory per array = 129.7 MiB = 0.1 GiB.
Total memory required = 389.1 MiB = 0.4 GiB.
Each kernel will be executed 20 times.
 The  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 5962 microseconds.
   = 5962 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:           38444.8     0.007299     0.007075     0.007544
Scale:          28665.3     0.009878     0.009489     0.010262
Add:            27657.4     0.015129     0.014752     0.015473
Triad:          28163.1     0.015124     0.014487     0.015558
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

Figure 5: STREAM benchmark result for Euler VII Phase 2

2.4. Performance model: A simple roofline model

As a result, the last question is to answer, at what operational intensity is a kernel or application memory/compute-bound? To answer this question the provided paper by Williams et al. demonstrates the roofline model, which can be described as a visual performance model, that can help in order to optimize code and software for floating point computations. Furthermore, the proposed model combines three important computer metrics which were already computed in the prior tasks. Parameters included are, the floating point performance, the operational intensity and the memory performance. The resulting 2D graph then consists of the following two parts:

1. Horizontal line: represents the peak floating point performance of the core.

2. Linear line: represents the peak memory bandwidth.

Moreover, the intersection between the two lines results in a point, which describes the limits for the underlying system. As a result to determine at what operational intensity a kernel or application is memory-bound or compute-bound, we have to look at where the performance of the kernel intersects (the described critical point) with the memory bandwidth line on the graph. Hence, on one hand, if the kernel's performance is below this line, it's liekly memory-bound. On the other hand, if it's above the line it's compute-bound.

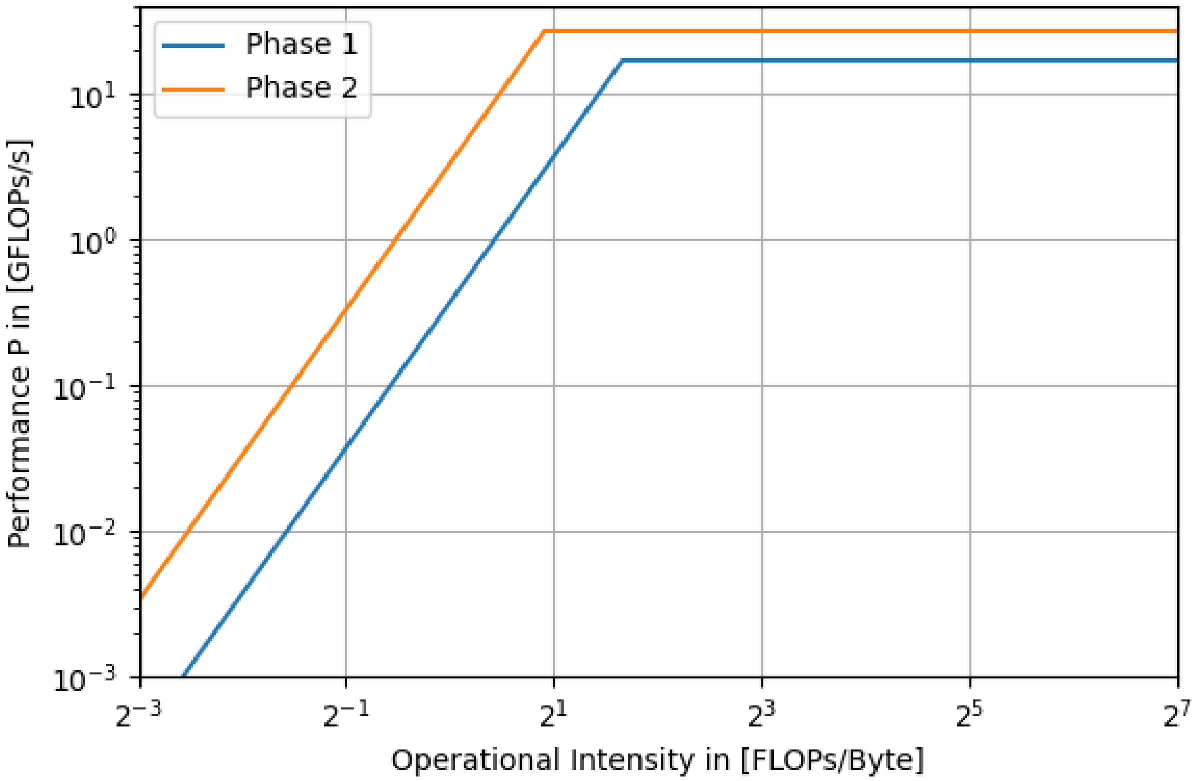

Figure 6: Simple roofline model for a single Euler VII Phase 1 and Phase 2 core.

As a result if we look at Figure 6, it is clear that the kernel or application is memory-bound at operational intensity I values below 3.2 and 1.9 for the Euler VII Phase 1 and 2 core respectively. If values are above this critical point the kernel and application is compute-bound.

Note that on the next page the solution for the Project 1b starts.

3. Auto-vectorization [10 points]

3.1. Why is it important for data structures to be aligned?

From the Intel Compiler Guide [1] we can extract the information that data structure alignment is the adjustment of any data object with relation to other objects. This alignment offers the opportunity to speed up memory access. As a result, it's important to know that misaligned memory accesss can result in performance losses. Hence data structures have to bused in order to store data on a computer so that it can be used efficiently!

3.2. What are some obstacles that can prevent automatic vectorization by the compiler?

Again in the Intel Compiler Guide [1], it's mentioned that in the following cases the compiler decides that vectorization would not be worthwhile.

- **Non-contiguous memory access:** if 4 consecutive integers, floats, or 2 consecutive doubles are not adjacent, they must be loaded separately using multiple instructions, which is considerably less efficient.
- **Data dependencies:** Vectorization results in changes in the order of operations within a loop. This is only possible, if this change of order doesn't change the results of the calculation.

3.3. Is there a way to help the compiler to vectorize and how?

When there is insufficient information for the compiler to decide to vectorize a loop, the following ways presents a solution how the compiler is able to vectorize a given problem with additional information [1].

- **Pragmas**
- **Keywords:** The restrict keword for instance can help the compiler to asswert that the memory referenced by a pointer is not aliased.
- **Options/Switches:** enables different levels of optimizations to achieve automatic vectorization.

3.4. Which loop optimizations are performed by the compiler to vectorize and pipeline loops?

These loop optimizations are typically performed by the compiler:

- **Stop-mining and Cleanup:** Strip-mining also known as loop sectioning is a loop transformation technique for enabling SIMD-encoding of loops. This improves memory performance. Another possibility in this context is loop blocking.
- **Loop Interchange and Subscripts with Matrix Multiply:** This improves memory the memory access patterns. The result of this method can be seen especially in the next task to optimize the DGEMM.

3.5. What can be done if automatic vectorization by the compiler fails or is sub-optimal?

There are different ways to solve this problem, as illustrated in the following points.

- **Manual Vectorization:** As done in the following DGEMM example code can be rewritten to explicitly use vectorization instructions.
- **Loop Unrolling:** This can also be done manually and reduce loop overhead.

- **Pragma Directives:** Further pramga directives such as #pragma omp simd.
- **Keywords:** Additional keywords like restrict.
- **Compiler Flags:** Additional or more aggresive compiler optimization flags can resolve this problem.
- **Profiling:** Analysis of the generated assembly code to identify bottlenecks and areas of improvement.
- **Software Pipelining:** This technique allows overlapping of loop iterations and improve vectorization.

Which one of the mentioned approaches actually helps, depends ont the specific code and architecture. It could be seen from the next example the optimization of the DGEMM problem. It becomes evident that not all of the mentioned optimization techniques to assure vectorization help.

4. Matrix multiplication optimization [30 points]

The goal of this task is to optimize the naive matrix multiplication through replacing it with a blocking optimization. Also for further optimization its performance, we should consider various additional strategies. In this part of the report I will present the evolution of how I reached better results through trying out different optimization strategies.
I started off by looking at the initial output first, without any optimization and only the comparison between the DGEMM (Double Precision GEneral Matrix Multiplication) implementation of the BLAS library, which is highly optimized and the naive matrix multiplication. The result can be seen in the Figure 7

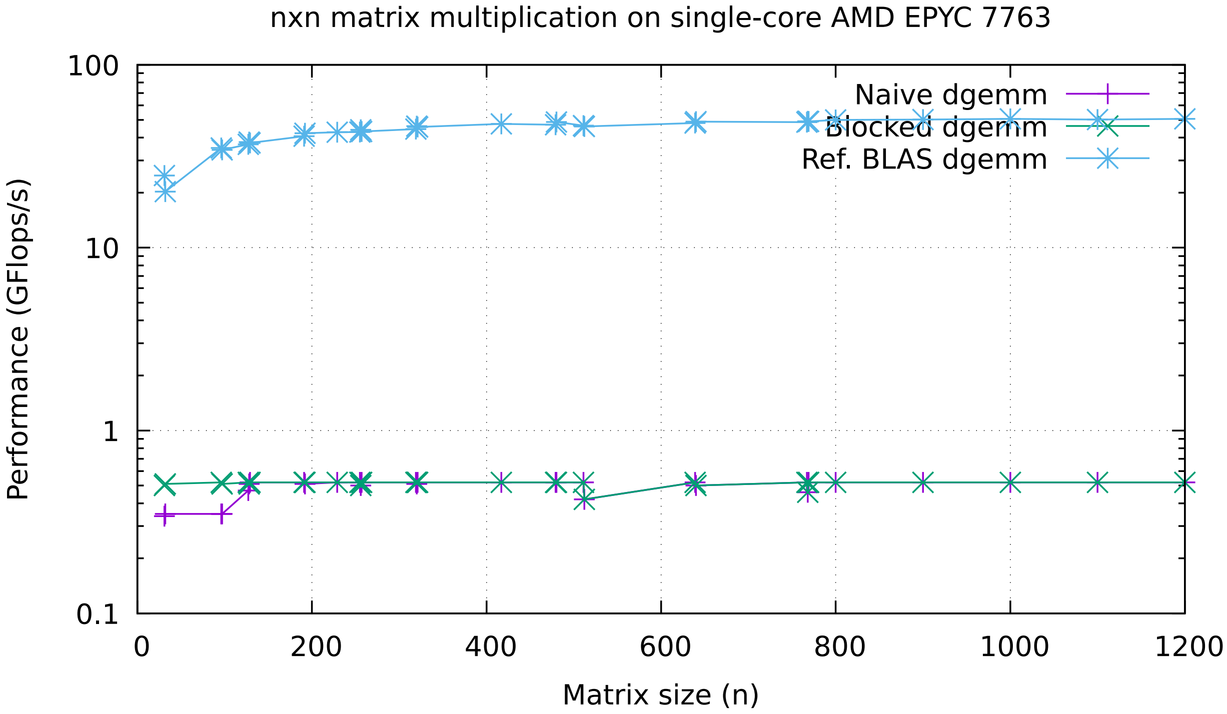

Figure 7: Results of the DGEMM without any optimization.

From this image, it's evident, as described in the exercise sheet that there is a huge optimization potential with the goal to come as close as possible to the BLAS implementation. First step I did to improve the results was to add optimization flags to push the results of the naive approach a little closer.

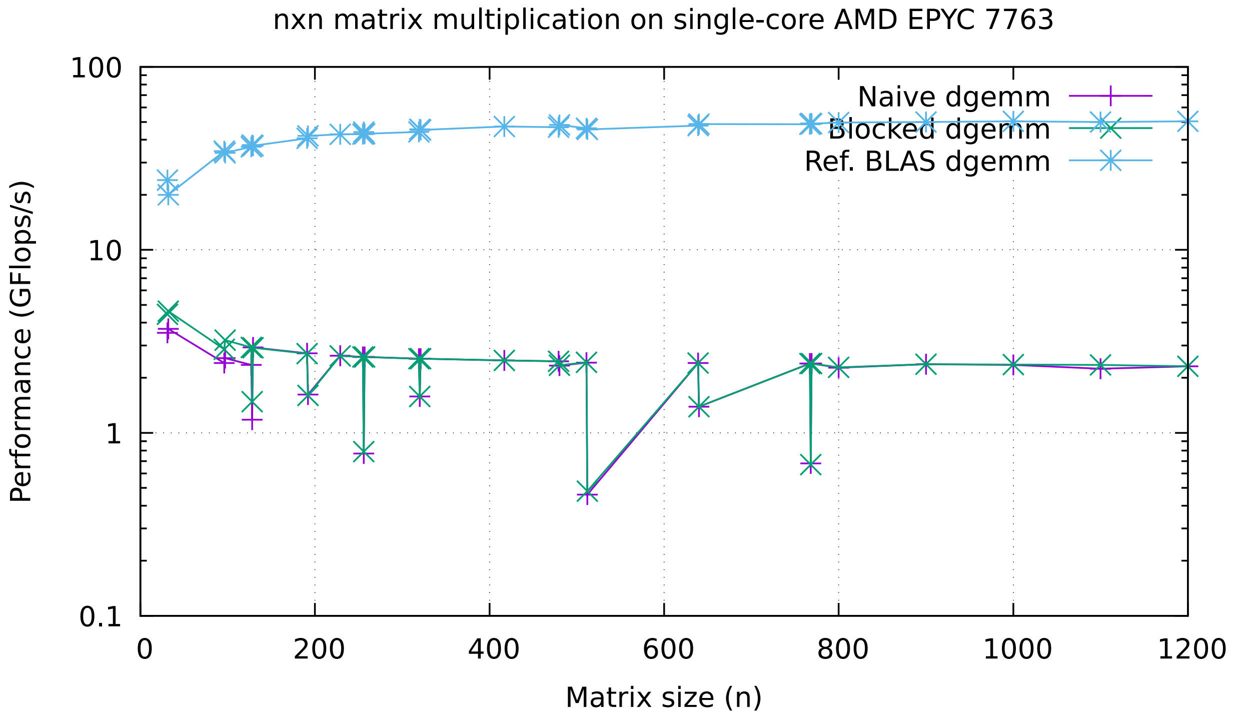

Figure 8: Results of the DGEMM with compiler optimization.

As a result, Figure 8 shows that just by adding some compiler flags can improve the results dramastically. I will briefly summarize the utilized compiler flags:

- **-03**: This flag represents the optimization level, where I used an aggressive approach that even outperforms -Ofast.
- **-march=native**: This gives the compiler the instruction to generate code for the host machine's architecture.
- **-funroll-loops**: This compiler flag enables loop unrolling.

It becomes evident from Figure 8 that only these few flags have a huge effect on the overall performance of the DGEMM task. A compiler optimization guide provided specifically for the used AMD EPYC processor series offers more insights in how to tune a GNU compiler for even more optimized results [2], [3]. In [3] the suggested compiler flags for a GNU compiler area as follows:

$$OPT = -O3 - march = znver1 - mtune = znver1 - mfma \\ - mavx2 - m3dnow - fomit - frame - pointer$$

This should lead to a god optimized result and in the end should generate optimal code for the AMD EPYC processor [3]. Unfortunatelly this wasn't the case for me, I think it is due to #pragma intrinsicts that should have been placed more properly to make use of SIMD. That's why I used the following compiler flags, as already presented:

$$OPT = -O3 - march = native - funroll - loops$$

Also to mention just a few, I tried other compiler flags such as -ftree-vectorize -fopt-info-vec to further optimize DGEMM, but couldn't reach better results. A list of all possible compiler intrinsics for the underlying AMD EPYC processor can be found on Microsoft's webpage [4].
In the next approach, since in the exercise sheet it was stated that the provided template uses the column major order storage scheme, I looked at the naive code closer and found that the loop

hierarchy could be handled better. Through changing it I reached the following results, illustrated in Figure 9. Note that all compiler optimization described from before are utilized for all the following explanations.

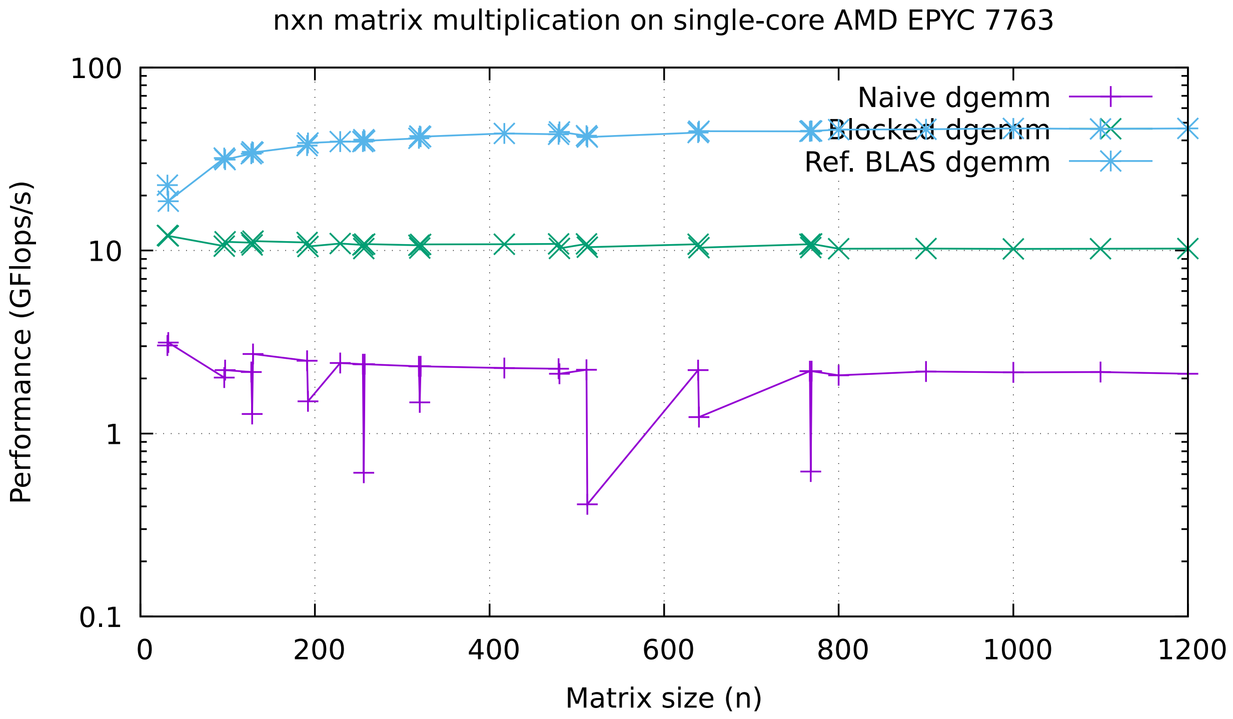

Figure 9: Results of the DGEMM with a loop rearrangement.

From the image it's evident that rearranging the order of the loops has a huge impact on the overall performance since we can utilize a unity stride in a better way, as explained in the previous chapter where the concept of vectorization was discussed. Note that the underlying output files can be found in the Appendix of this chapter. From Figure 12 we can see that the average percentage of the peak performance is around 27,38%. In comparison the so far best possible result, the BLAS library, has according to Figure 13 an average percentage of the peak performance around 102,02%. Unfortunatelly, this was already my best result I could achieve.
Nevertheless I will present further optimization trials. Since the main task was to implement a blocked matrix DGEMM approach, I briefly want to adress this task as well. The resulting code for this task can be found in the `dgemm-blocked.c`, where both versions, my optimized naive approach as well as the tiled matrix DGEMM are available. From the exercise sheet it becomes evident that for this approach we need a block size to split the problem up and this size actually depends on the underlying processor architecture. From previous tasks we know that we want to store frequently used data inside the L1 cache, as close as possible to the CPU. The memory capacity of the L1 cache for the AMD EPYC 7763 processor is 32KB, we can utilize the conversion that 1KB = 2024 Bytes and get in the end the following result for an upper bound of the block size:

$$s \leq \sqrt{\frac{32 \cdot 1024}{3 \cdot 8}} = 36$$

The product of $32 \cdot 1024$ has to be divided through $3 \cdot 8$ this is the case because the size of a double precision floating point is 8 Bytes. Also, as stated and explained in the exercise sheet s should be as large as possible to gain best results so in this case, the best result for the matrix blocking should be with a blocking size of 36. But since this is only a theoretical upper bound there is still some adjustment necessary. In my case, as evident from Figure 11, I got the best results with a block size value of 26 instead of 36.

Initially, I had an average percentage of the peak performance around 22%. Through further pragma directives I was able to tweak it at least up to a value of 24,66%. The most important one is #pragma unroll which tells the compiler to unroll the following loop. Another used pragma directive is #pragma omp atomic, which helped eventhough there was no multithreading involved. For some strange reason also the second used pragma directive is just for parallel computation, but it still improved my results a bit, which doesn't make any sense to be honest. #pragma omp parallel for collapse(2) enables parallelization across the following two loop dimensions for improved performance in a parallel computing environment. Also I want to mention another rather strange observation if the block size was defined with a preprocessor directive, more specifically the macro named BLOCK_SIZE, instead of initializing it with an integer, the performance dropped rapidly from 24% to 10%. Hence I stayed with the implementation provided in the code.
To summarize my result briefly, eventhough I tried out all mentioned optimization techniques and did even further investigation I wasn't able to tweak the tiled matrix multiplication to a better result than the naive approach.

4.1. Appendix

The following Figures 10, 11, 12 and 13 are the outputted results for the AMD EPYC 7763 processor on the Euler cluster.

```
# Description:        Naive, three-loop dgemm.

Size:       31        Gflop/s:     2.53        Percentage:    6.46
Size:       32        Gflop/s:     2.59        Percentage:    6.60
Size:       96        Gflop/s:     2.02        Percentage:    5.16
Size:       97        Gflop/s:     2.16        Percentage:    5.50
Size:      127        Gflop/s:     2.05        Percentage:    5.23
Size:      128        Gflop/s:     1.21        Percentage:    3.08
Size:      129        Gflop/s:     2.70        Percentage:    6.88
Size:      191        Gflop/s:     2.49        Percentage:    6.36
Size:      192        Gflop/s:     1.49        Percentage:    3.81
Size:      229        Gflop/s:     2.42        Percentage:    6.18
Size:      255        Gflop/s:     2.39        Percentage:    6.09
Size:      256        Gflop/s:     0.60        Percentage:    1.53
Size:      257        Gflop/s:     2.39        Percentage:    6.08
Size:      319        Gflop/s:     2.32        Percentage:    5.93
Size:      320        Gflop/s:     1.44        Percentage:    3.68
Size:      321        Gflop/s:     2.33        Percentage:    5.94
Size:      417        Gflop/s:     2.28        Percentage:    5.82
Size:      479        Gflop/s:     2.26        Percentage:    5.76
Size:      480        Gflop/s:     2.11        Percentage:    5.38
Size:      511        Gflop/s:     2.23        Percentage:    5.68
Size:      512        Gflop/s:     0.41        Percentage:    1.04
Size:      639        Gflop/s:     2.21        Percentage:    5.65
Size:      640        Gflop/s:     1.22        Percentage:    3.12
Size:      767        Gflop/s:     2.19        Percentage:    5.60
Size:      768        Gflop/s:     0.63        Percentage:    1.62
Size:      769        Gflop/s:     2.20        Percentage:    5.60
Size:      800        Gflop/s:     2.09        Percentage:    5.33
Size:      900        Gflop/s:     2.18        Percentage:    5.56
Size:     1000        Gflop/s:     2.16        Percentage:    5.50
Size:     1100        Gflop/s:     2.17        Percentage:    5.53
Size:     1200        Gflop/s:     2.13        Percentage:    5.42
# Average percentage of peak performance = 5.06834
```

Figure 10: This is the result without any optimization of the basic/naive DGEMM.

```
# Description:       Blocked dgemm.

Size:      31       Gflop/s:     9.30       Percentage:   23.73
Size:      32       Gflop/s:     9.57       Percentage:   24.42
Size:      96       Gflop/s:    10.88       Percentage:   27.74
Size:      97       Gflop/s:    10.24       Percentage:   26.12
Size:     127       Gflop/s:    10.67       Percentage:   27.22
Size:     128       Gflop/s:    10.17       Percentage:   25.94
Size:     129       Gflop/s:     9.82       Percentage:   25.06
Size:     191       Gflop/s:    10.28       Percentage:   26.24
Size:     192       Gflop/s:    11.19       Percentage:   28.54
Size:     229       Gflop/s:    10.38       Percentage:   26.48
Size:     255       Gflop/s:     8.47       Percentage:   21.61
Size:     256       Gflop/s:     8.16       Percentage:   20.82
Size:     257       Gflop/s:     7.85       Percentage:   20.03
Size:     319       Gflop/s:    10.22       Percentage:   26.07
Size:     320       Gflop/s:    11.46       Percentage:   29.24
Size:     321       Gflop/s:    10.44       Percentage:   26.64
Size:     417       Gflop/s:    10.49       Percentage:   26.76
Size:     479       Gflop/s:    10.67       Percentage:   27.23
Size:     480       Gflop/s:    10.95       Percentage:   27.94
Size:     511       Gflop/s:     6.14       Percentage:   15.65
Size:     512       Gflop/s:     4.73       Percentage:   12.06
Size:     639       Gflop/s:     9.62       Percentage:   24.54
Size:     640       Gflop/s:    10.07       Percentage:   25.70
Size:     767       Gflop/s:     7.90       Percentage:   20.16
Size:     768       Gflop/s:     7.77       Percentage:   19.81
Size:     769       Gflop/s:     8.08       Percentage:   20.62
Size:     800       Gflop/s:    10.78       Percentage:   27.50
Size:     900       Gflop/s:    11.25       Percentage:   28.71
Size:    1000       Gflop/s:    10.56       Percentage:   26.93
Size:    1100       Gflop/s:    10.47       Percentage:   26.72
Size:    1200       Gflop/s:    11.13       Percentage:   28.38
# Average percentage of peak performance = 24.6644
```

Figure 11: This is the result of the implemented and even more optimized matrix blocking/tiling approach.

```
# Description:        Blocked dgemm.

Size:       31        Gflop/s:    12.10        Percentage:   30.87
Size:       32        Gflop/s:    12.01        Percentage:   30.64
Size:       96        Gflop/s:    10.57        Percentage:   26.98
Size:       97        Gflop/s:    11.15        Percentage:   28.44
Size:      127        Gflop/s:    11.03        Percentage:   28.14
Size:      128        Gflop/s:    10.73        Percentage:   27.38
Size:      129        Gflop/s:    11.25        Percentage:   28.71
Size:      191        Gflop/s:    11.07        Percentage:   28.24
Size:      192        Gflop/s:    10.53        Percentage:   26.87
Size:      229        Gflop/s:    10.91        Percentage:   27.84
Size:      255        Gflop/s:    10.73        Percentage:   27.38
Size:      256        Gflop/s:    10.25        Percentage:   26.14
Size:      257        Gflop/s:    10.84        Percentage:   27.66
Size:      319        Gflop/s:    10.69        Percentage:   27.26
Size:      320        Gflop/s:    10.31        Percentage:   26.29
Size:      321        Gflop/s:    10.79        Percentage:   27.52
Size:      417        Gflop/s:    10.82        Percentage:   27.61
Size:      479        Gflop/s:    10.88        Percentage:   27.76
Size:      480        Gflop/s:    10.28        Percentage:   26.22
Size:      511        Gflop/s:    10.87        Percentage:   27.73
Size:      512        Gflop/s:    10.42        Percentage:   26.58
Size:      639        Gflop/s:    10.83        Percentage:   27.64
Size:      640        Gflop/s:    10.37        Percentage:   26.44
Size:      767        Gflop/s:    10.82        Percentage:   27.61
Size:      768        Gflop/s:    10.38        Percentage:   26.49
Size:      769        Gflop/s:    10.88        Percentage:   27.77
Size:      800        Gflop/s:    10.23        Percentage:   26.11
Size:      900        Gflop/s:    10.25        Percentage:   26.15
Size:     1000        Gflop/s:    10.21        Percentage:   26.05
Size:     1100        Gflop/s:    10.22        Percentage:   26.07
Size:     1200        Gflop/s:    10.26        Percentage:   26.17
# Average percentage of peak performance = 27.3785
```

Figure 12: This is the result of the fine tuned and optimized naive DGEMM.

```
# Description:        Reference dgemm.

Size:       31        Gflop/s:    22.74        Percentage:   58.02
Size:       32        Gflop/s:    18.58        Percentage:   47.39
Size:       96        Gflop/s:    32.10        Percentage:   81.88
Size:       97        Gflop/s:    31.43        Percentage:   80.19
Size:      127        Gflop/s:    33.86        Percentage:   86.38
Size:      128        Gflop/s:    34.70        Percentage:   88.51
Size:      129        Gflop/s:    34.35        Percentage:   87.63
Size:      191        Gflop/s:    37.55        Percentage:   95.80
Size:      192        Gflop/s:    38.81        Percentage:   99.02
Size:      229        Gflop/s:    39.49        Percentage:  100.75
Size:      255        Gflop/s:    39.47        Percentage:  100.68
Size:      256        Gflop/s:    40.49        Percentage:  103.29
Size:      257        Gflop/s:    39.78        Percentage:  101.47
Size:      319        Gflop/s:    41.02        Percentage:  104.64
Size:      320        Gflop/s:    42.31        Percentage:  107.93
Size:      321        Gflop/s:    41.92        Percentage:  106.93
Size:      417        Gflop/s:    43.67        Percentage:  111.41
Size:      479        Gflop/s:    43.30        Percentage:  110.46
Size:      480        Gflop/s:    44.56        Percentage:  113.68
Size:      511        Gflop/s:    42.46        Percentage:  108.31
Size:      512        Gflop/s:    41.74        Percentage:  106.47
Size:      639        Gflop/s:    44.23        Percentage:  112.83
Size:      640        Gflop/s:    44.88        Percentage:  114.49
Size:      767        Gflop/s:    44.74        Percentage:  114.14
Size:      768        Gflop/s:    45.16        Percentage:  115.20
Size:      769        Gflop/s:    44.99        Percentage:  114.77
Size:      800        Gflop/s:    45.88        Percentage:  117.04
Size:      900        Gflop/s:    46.09        Percentage:  117.57
Size:     1000        Gflop/s:    46.54        Percentage:  118.72
Size:     1100        Gflop/s:    46.28        Percentage:  118.07
Size:     1200        Gflop/s:    46.60        Percentage:  118.87
# Average percentage of peak performance = 102.017
```

Figure 13: This is the result of the the BLAS library for the DGEMM task.

References

[1] Intel oneapi dpc++/c++ compiler developer guide and reference. [online]. available: https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2024-0/overview.html. accessed: 2024-03.

[2] Amd epyc series processors: Compiler options quick reference guide. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/user-guides/compiler-options-guide-amd-epyc-7xx1-series-processors.pdf. AMD, Latest Release 2018-07, Accessed: 2024-03.

[3] Ole Widar Saastad Xu Guo. Best practice guide: Amd epyc. *University of Oslo*, 2019.

[4] x64 (amd64) intrinsics list. [online]. available: https://learn.microsoft.com/en-us/cpp/intrinsics/x64-amd64-intrinsics-list?view=msvc-170. accessed: 2024-03.