Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Student: CARLA JUDITH LOPEZ ZURITA

Discussed with: YANNICK RAMIC
ALITZEL MACIAS

Solution for Project 1

Due date: 11 March 2024, 23:59

The aim of this project is to familiarize the students with the Euler cluster and to provide an introduction to performance characteristics, memory hierarchies, and optimization techniques. The project is divided into four sections: Euler warm-up, performance characteristics, auto-vectorization, and matrix multiplication optimization.

1. Euler warm-up [10 points]

The first section is designed to familiarize the user with the basic components of the Euler cluster available to the students and collaborators of ETH. The questions and answers are designed to provide a brief overview of the Euler cluster and its intended function.

1. *What is the module system and how do you use it?* Modules are used to configure the environment for a particular software version. They configure the necesary variables to make sure all required binaries and libraries are found. On the Euler cluster, there are two types of Modules packages:

   a) LMOD Modules

   b) Environment Modules

   To use the modules, on the terminal we write the keyword **module** followed by the commands. For example, to activate the LMOD module and install certain libraries or specify the version, run on the terminal the following commands,

   ```
   module load gcc/6.3.0 python/3.8.5
   ```

2. *What is Slurm and its intended function?* Euler uses Slurm, which stands to Simple Linux Utility for Resource Management, for job management. In order to actually run something, for example a C++ program, the computations must be submitted to the batch system. To submit scripts to Slurm use the following line,

   ```
   sbatch [options] --wrap "command [arguments]"
   ```

   Basically, instead of running an executable using the command **./a.out**, one would run

   ```
   sbatch --wrap "./a.out"
   ```

   To see the job list, run **squeue**. You can also cancel your job with **scancel** plus the job number. To avoid writing the sequence of commands manually and to add more flags or specifications, you can use a batch script, whcih can be then run using,

   ```
   sbatch < my_batch_script.sh
   ```

3. *Write a simple "Hello World" C/C++ program which prints the host name of the machine on which the program is running.* Attached bellow is a C++ program for getting the hostname. I tried using the <limits.h> header file but I couldn't retrieve HOST_NAME_MAX (using the C++ compiler on macOs).

```
#include <iostream>
#include <unistd.h> // For gethostname function

int main()
{
char hostname[256]; // Guess for size of hostname

// Get the host name
if (gethostname(hostname, sizeof(hostname)) == 0)
{
    // Success
    std::cout << "Hello world, this program is running on"
    << hostname << std::endl;
}
else
{
    // Error
    std::cerr << "Failed to get the host name" << std::endl;
}

return 0;
}
```

The output when running this small program was:

```
Hello world, this program is running on Carlas-MacBook-Pro.local
```

4. *Write a batch script which runs your program on one node. The batch script should be able to target nodes with specific CPUs. Hint: Slurm has the option –constraint to select some nodes with specific features. Possible options to select nodes featuring a certain CPU: EPYC_7742, EPYC_7H12, EPYC_7763, EPYC_9654.*

We can use the help of the page LSF/Slurm Submission Line Advisor to write the batch script. For this exercise, I used the skeleton source priveded in the class material. The output was two files, an *out* file and an *err* file, both named *slurm_job_one-48720143*. The *out* file contained the following info,

```
Model name: AMD EPYC 9654 96-Core Processor
eu-g9-046-1
```

5. *Write another batch script which runs your program on two nodes. Check that the Slurm output file slurm.out by default contains two different host names* Once again, I used the template provided in the Moodle of the course and again obtained two files (*out* and *err*), this time named *slurm_job_two-48756918*. The *out* file reads,

```
Model name: AMD EPYC 7763 64-Core Processor
eu-a2p-381
eu-a2p-385
```

2. Performance characteristics [50 points]

In order to understand the performance characteristics of a computing system, we need to know what physical constraints limit data transfer rates and computation speeds within it. In this section, we will analyze the peak performance of the Euler VII (Phase I and II) clusters, the memory hierarchies, and the STREAM benchmark. We will also use the roofline model to analyze the performance of the nodes.

2.1. Peak performance

In this task, we compute the peak performance of a core, CPU, node and cluster for the Euler VII (Phase I and II) nodes. The peak performance P of each of the elements is computed as follows

$$P_{core} = n_{super} \times n_{\text{FMA}} \times n_{\text{SIMD}} \times f,$$

$$P_{CPU} = n_{cores} \times P_{core},$$

$$P_{node} = n_{sockets} \times P_{CPU},$$

$$P_{cluster} = n_{nodes} \times P_{node},$$

where n_{super} is the superscalarity factor, n_{FMA} is the fused multiply-add factor, n_{SIMD} is the SIMD (Single Instruction, Multiple Data) factor, and f is the (base) clock frequency. n_{cores} is the number of cores per CPU, $n_{sockets}$ is the number of sockets per node, and n_{nodes} is the number of nodes in the cluster.

For Phase I, we find the corresponding values in the literature and resources available online.

- n_{super}: 2 [?]
- n_{FMA}: 2 [?]
- n_{SIMD}: SIMD width supported by AMD Zen2 is 256 bits per SIMD register. [?]
- f: 2.6 GHz nominal, 3.3 GHz peak [?]
- n_{cores}: 64-core AMD EPYC 7H12 processors per CPU (Product Line: AMD EPYC™ 7002 Series) [?]
- $n_{sockets}$: each node has 2 AMD EPYC 7H12 CPUs [?]
- n_{nodes}: 292 compute nodes — HPE ProLiant XL225n Gen10 Plus [?]

The peak performance is computed as follows

$$P_{core} = 2 \times 2 \times 4 \times 2.6 \text{ GHz} = 41.6 \text{ GFlops/s}$$

$$P_{CPU} = 64 \times 41.6 = 2662.4 \text{ GFlops/s}$$

$$P_{node} = 2 \times 2662.4 = 5324.8 \text{ GFlops/s}$$

$$\boxed{P_{\text{Euler VII.I}} = 292 \times 5.325 \text{ TFlops/s} = 1,554.8416 \text{ TFlops/s}}$$

For Phase II:

- n_{super}: 2 [?]
- n_{FMA}: 2 [?]
- n_{SIMD}: 4 double-precision (AMD Zen3) [?]

- f: 2.45 GHz nominal, 3.5 GHz peak [?]
- n_{cores}: 64-core AMD EPYC 7763 processors per CPU (Product Line: AMD EPYC™ 7003 Series) [?]
- $n_{sockets}$: each node has 2 AMD EPYC 7763 CPUs [?]
- n_{nodes}: 248 compute nodes — HPE ProLiant XL225n Gen10 Plus [?]

The peak performance is computed as follows

$$P_{core} = 2 \times 2 \times 4 \times 2.45 \text{ GHz} = 39.2 \text{ GFlops/s}$$

$$P_{CPU} = 64 \times 39.2 = 2508.8 \text{ GFlops/s}$$

$$P_{node} = 2 \times 2508.8 = 5017.6 \text{ GFlops/s}$$

$$\boxed{P_{\text{Euler VII.II}} = 248 \times 5.018 \text{ TFlops/s} = 1,244.365 \text{ TFlops/s}}$$

From the calculatiosn, we can see that the peak performance of the Euler VII (Phase I) cluster is greater than the peak performance of the Euler VII (Phase II) cluster. This is due to the higher base and peak clock frequency of the AMD EPYC 7H12 processor compared to the AMD EPYC 7763 processor.

2.2. Memory Hierarchies

The memory hierarchy of a computing system is a pyramid of storage devices with different capacities, access times, and costs. In this section, we identify the parameters of the memory hierarchy on a node of the Euler VII (Phase I and II) cluster.

2.2.1. Cache and main memory size

After logging into the Euler VII (Phase I and II) cluster, we can use the **lscpu** command to obtain information about the CPU architecture. We also read the file **/proc/meminfo** to obtain information about the main memory. Finally, **hwloc-ls** can be used to obtain information about the cache sizes. The results are shown in Table **??**. Each core has its own L1 and L2 cache. The retrived information reveals that the L3 cache is shared among eight cores for the AMD EPYC 7H12 and four cores for the AMD EPYC 7763. The schemas of the memory hierarchy of both processors can be found in Figure **??** and Figure **??** included in the Appendix.

Table 1: Memory hierarchy of Euler VII login nodes Phases I and II.

Euler VII	Phase I	Phase II
CPU	AMD EPYC 7H12	AMD EPYC 7763
Main memory	31 GB	31 GB
L3 cache	16 MB	32 MB
L2 cache	512 KB	512 KB
L1d cache	32 KB	32 KB
L1i cache	32 KB	32 KB

2.3. Bandwidth: STREAM benchmark

We now consider in our analysis th speed of the memory system, quantitatively measured using its bandwith, which can be obtained using the STREAM benchmark. STREAM uses four different kernels (or operations) to measure the sustainable memory bandwidth of a system and its corresponding computation rate. These kernels are:

- **Copy**: Copying elements from one array to another.
- **Scale**: Multiplying each element of an array by a scalar.
- **Add**: Adding elements of two arrays and storing the result in a third array.
- **Triad**: Combining the scale and add operations.

The STREAM benchmark for Euler VII Phase I (AMD EPYC 7H12) was executed with the following parameters:

- Array size: 8,500,000 elements
- Offset: 0 elements
- Memory per array: 64.8 MiB (= 0.1 GiB)
- Total memory required: 194.5 MiB (= 0.2 GiB)
- Each kernel was executed 20 times

The results of the analysis are shown in Table **??**. We observe that the bandwidths resulting from the Scale, Add, and Triad kernels are roughly consistent, while the bandwidth for the Copy kernel appears to be substantially higher. For a rough estimate, we can assume a maximum bandwidth **bSTREAM = 27 GB/s**.

Table 2: Performance Metrics for Euler VII Phase I (AMD EPYC 7H12).

Function	Best Rate (MB/s)	Avg time (s)	Min time (s)	Max time (s)
Copy	38822.9	0.003888	0.003503	0.004435
Scale	29254.1	0.005408	0.004649	0.005897
Add	27327.9	0.008317	0.007465	0.009960
Triad	27991.3	0.008281	0.007288	0.010853

The STREAM benchmark for Euler VII Phase II (AMD EPYC 7763) was executed with the following parameters:

- Array size: 17,000,000 elements
- Offset: 0 elements
- Memory per array: 129.7 MiB (= 0.1 GiB)
- Total memory required: 389.1 MiB (= 0.4 GiB)
- Each kernel was executed 20 times

The results of the analysis are shown in Table **??**. We observe that the bandwidths resulting from the Scale, Add, and Triad kernels are roughly consistent, while the bandwidth for the Copy kernel appears to be substantially higher. For a rough estimate, we can assume a maximum bandwidth **bSTREAM = 34 GB/s**.

Table 3: Performance Metrics for Euler VII Phase II (AMD EPYC 7763).

Function	Best Rate (MB/s)	Avg time (s)	Min time (s)	Max time (s)
Copy	45822.8	0.006177	0.005936	0.006613
Scale	34862.8	0.008157	0.007802	0.009234
Add	33923.6	0.012687	0.012027	0.015450
Triad	33951.9	0.012721	0.012017	0.014769

2.4. Performance model: A simple roofline model

The roofline model offers valuable insights into a system's performance characteristics by graphing achievable performance against the operational intensity of a kernel. Operational intensity, defined as the ratio of floating-point operations to bytes transferred from memory, guides this analysis.

The model delineates two key regions: memory-bound and compute-bound. In the memory-bound segment, performance is constrained by the system's memory bandwidth, whereas the compute-bound area is limited by the processor's peak computational capacity. This is useful for identifying performance bottlenecks within the system.

We present the roofline models for single cores of Euler VII (Phases I and II) nodes, as well as the roofline model for Euler III, presented in class, for reference purposes. The model was calculated using the values for the peak performance (P_{Peak}) presented in Section **??** and the memory bandwidths (b_{STREAM}) found in Section **??**. We can identify the ridge point of the roofline model, which represents the transition point between the memory-bound and compute-bound regions in terms of its operational intensity I_{ridge}. Table **??** summarizes these three values for the Euler III and Euler VII (Phase I and II) cores. The results are shown in Figure **??**.

Table 4: Peak performance (P_{Peak}), memory bandwith (b_{STREAM}) and ridge point (I_{ridge}) for different Euler cores.

Core	P_{Peak}	b_{STREAM}	I_{ridge}
Euler VII (Phase I)	41.6 (GFlops/s)	27 (GB/s)	1.5407 (Flops/B)
Euler VII (Phase II)	39.2 (GFlops/s)	34 (GB/s)	1.1529 (Flops/B)
Euler III	48 (GFlops/s)	12 (GB/s)	4.0000 (Flops/B)

3. Auto-vectorization [10 points]

After reading the Chapter "Automatic Vectorization" of the *Intel oneAPI DPC++/C++ Compiler Developer Guide and Reference* [**?**], we provide a brief answer to the following questions:

1. *Why is it important for data structures to be aligned?* Data structure alignment referts to the organization and retrieval of data stored in memory. It involves adjusting data objects in relation to each other. For example, the Intel oneAPI DPC++/C++ compiler can arrange variables to begin at specific memory addresses, improving memory access speed. Misaligned memory accesses may result in significant performance declines on certain processors that lack hardware support for them.

2. *What are some obstacles that can prevent automatic vectorization by the compiler?* Some obstacles that may cause the compiler not to vectorize include non-conntiguous memory access and data dependencies. Non-contiguous memory access occurs when the data is not stored in a contiguous block of memory and therefore leads to inneficcient memory access. The most prominent examples are loops with non-unit stride or indirect addressing. On the other hand, data dependencies occur when the result of changing the order of operations necessary for vectorization would change the result of the program. This can be caused by loop-carried dependencies, anti-dependencies, and output dependencies.

3. *Is there a way to help the compiler to vectorize and how?* Sometimes, the compiler is unsure as to whether or not it is safe to vectorize a loop. You can help the compiler vectorize by the use of pragmas, keywords and options/switches.

   Pragma directives are used to provide additional information to the compiler about how to optimize the code. For example, whether it is safe to ignore any potential data dependencies, give hints about the loop structure, ask the compiler to vectorize or nor certain loops, assert that the data is aligned, and many others.

../task_2_4/roofline.pdf

Figure 1: Log-Log Roofline Model for Euler III and Euler VII Nodes (Phase I and II).

Certain keywords can help the compiler with optimization. Specifically, the *restrict* keyword can be used to tell the compiler that the pointer is the only pointer that can access the data it points to. It is similar to the *const* keyword, but for pointers, and the pragma *ivdep*. The drawback is that not all compilers support keywords, which reduces portability.

Finally, options/switches can be used to enable different levels of optimizations, such as Interprocedural Optimization (IPO) and High-Level Optimizations (HLO).

4. *Which loop optimizations are performed by the compiler to vectorize and pipeline loops?* Loop vectorization transforms scalar loops into vectorized loops that can process multiple data elements in parallel using SIMD (Single Instruction, Multiple Data) instructions. This optimization can significantly improve performance by utilizing the full power of vectorized hardware.

   In order to efficienly vectorrize loops, the compiler does some optimizations automatically, such as loop unrolling, loop fusion, loop interchange, and loop sectioning (also known as stip-mining). Loop unrolling means that the compiler replicates the loop body. Loop fusion combines multiple loops into a single loop, while loop interchange changes the order of nested loops. Loop sectioning divides a loop into smaller blocks. All of these techniques aim to reduce loop overhead and improve data temporal and spatial locality, allowing the process of vectorization to be more efficient.

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

   There are a number of strategies that can be used in case automatic vectorization by the compiler fails or is sub-optimal. Manual vectorization involves using explicit vector programming models, such as SIMD intrinsics or compiler-specific vectorization directives. You can also try restructuring the code to improve data locality and reduce data dependencies, for example try different data structures such as structures of arrays instead of arrays of structures as was discussed in the chapter. YOu can also try vector blocking, which is a technique that divides the data into smaller blocks that fit into the cache. Finally, you can use profiling tools and performance analysis to identify vectorization bottlenecks and understand where the issue might be. Trying out differente compiler flags and options can also help.

4. Matrix multiplication optimization [30 points]

In this section, we implemented the matrix-matrix multiplication using three algorithms, the naive approcah the blocked algorithm, and using the OpenBLAS library when running it on an **AMD EPYC_7763 CPU**. Afterwards, we optimized the blocked algorithm to maximize its performance using different techniques and strategies that will be discussed in the following paragraphs. We use the OpenBLAS implementation as benchmark, and can be also refered to in the text as *reference*.

The routines perform a *dgemm* operation, $C := C + A * B$ where A, B, and C are lda-by-lda matrices stored in column-major format. On exit, A and B maintain their input values. The implementation of the naive approach is shown in the following code snippet.

```c
const char *dgemm_desc = "Naive, three-loop dgemm.";

void square_dgemm(int n, double *A, double *B, double *C)
{
  for (int i = 0; i < n; ++i)
  {
    for (int j = 0; j < n; ++j)
    {
      for (int k = 0; k < n; ++k)
      {
        C[i + j * n] += A[i + k * n] * B[k + j * n];
      }
    }
  }
}
```

Similarly, the implementation of the blocked algorithm is shown in the following code snippet. The L1 cache size for the AMD EPYC 7763 is 52 KB, which can be written down using the bitwise left shift operator. The block size is calculated as $s = \sqrt{M/3}$, which is then used to divide the matrix into blocks of size $s \times s$. Finally, the macro-kernel is used to perform the multiplication of the blocks.

```
const char *dgemm_desc = "Blocked dgemm.";
#include <math.h>

void square_dgemm(int n, double *A, double *B, double *C)
{
  const unsigned int L1 = 1 << 15; // L1 cache size 52 KB
  unsigned int n_ = (unsigned int)sqrt(L1 / (3.0 * sizeof(double))); // block
      size

  for (unsigned int j = 0; j < n; j += n_)
  {
    for (unsigned int k = 0; k < n; k += n_)
    {
      for (unsigned int i = 0; i < n; i += n_)
      {
        // Macro-kernel
        for (unsigned int jj = j; jj < (j + n_ > n ? n : j + n_); ++jj)
        {
          for (unsigned int kk = k; kk < (k + n_ > n ? n : k + n_); ++kk)
          {
            double b = B[kk + jj * n];
            for (unsigned int ii = i; ii < (i + n_ > n ? n : i + n_); ++ii)
            {
              C[ii + jj * n] += A[ii + kk * n] * b;
            }
          }
        }
      }
    }
  }
}
```

The summary of the results of the performance of the three algorithms using no compilation flags is shown in Table ??. Afterwards, we tried several compiler flags to optimize the performance of both the naive and the blocked algorithms. The compiler flags added were:

Table 5: Performance summary using no compilation flags.

Algorithm	Avg Gflop/s	Avg Percentage
Naive, three-loop dgemm	0.37	1.16%
Reference dgemm	40.65	105.61%
Blocked dgemm	0.54	1.37%

```
-march=native -mtune=native -ffast-math -funroll-loops -ftree-vectorize -fopenmp
```

The compiler flags **-march=native** and **-mtune=native** optimize code generation for the host machine's CPU architecture, leveraging its specific instruction set and tuning parameters for performance. The **-ffast-math** flag enables aggressive optimizations on floating-point arithmetic, potentially sacrificing strict IEEE compliance for speed. With **-funroll-loops**, the compiler replicates loop bodies to reduce loop overhead and increase parallelism. **-ftree-vectorize** automatically transforms scalar operations into SIMD instructions, enhancing performance on SIMD architectures. Finally, **-fopenmp** enables OpenMP support, allowing developers to parallelize code easily for multi-core processors. Together, these flags streamline code execution and leverage hardware capabilities to maximize performance. The results of the performance of the three algorithms using these compilation flags is described in Table **??**. From the results, we can see that the performance of the naive algorithm improved slightly but the blocked algorithm remained the same. We then tried adding the **-O3** flag, which is the highest stable (or trusted) optimization level, to the already discussed flags. The results are shown in Table **??**. Using this flag, we can see a significant improvement in the performance of the naive and blocked algorithms. The performance of the naive algorithm improved to 5.08 Gflop/s, and the performance of the blocked algorithm improved to 10.14 Gflop/s, almost doubling the performance of the naive implementation. We can note that the performance of the reference algorithm also improved to 102.01 Gflop/s. Additionally, we tried using the **-Ofast** flag, but did not encounter any significant improvements compared to the **-O3** flag and decided not to include the results in the report.

Table 6: Performance summary using some compilation flags.

Algorithm	Avg Gflop/s	Avg Percentage
Naive, three-loop dgemm	0.46	1.16%
Reference dgemm	40.96	101.91%
Blocked dgemm	0.53	1.35%

Table 7: Performance summary using -O3 compilation flag.

Algorithm	Avg Gflop/s	Avg Percentage
Naive, three-loop dgemm	2.08	5.08%
Reference dgemm	40.01	102.01%
Blocked dgemm	3.98	10.14%

Afterwards, we tried using some pragma directives to optimize the performance of the blocked algorithm. OpenMP pragmas to potentially parallelize the outer loops and utilize SIMD (Single Instruction, Multiple Data) parallelism within the inner loops. We used the following pragmas:

1. **#pragma omp parallel for collapse(3) schedule(static)**: This pragma hints the compiler to parallelize the three outer loops (j, k, and i) using OpenMP. The *collapse(3)* directive collapses the three loops into one parallel loop. The *schedule(static)* clause ensures a static scheduling strategy for the iterations.

2. **#pragma omp simd**: This pragma suggests that the loop can be vectorized by the compiler. It allows the compiler to generate SIMD instructions for the loop iterations, potentially improving performance.

The exact implementation of the blocked algorithm using these pragmas can be found in the code attached to the report. The results of the performance of the blocked algorithm using these pragmas is shown in Table **??**. From the results we can see a slight improvement in the performance of the blocked algorithms, and no imprevemnet on the rest since no change was made in this regard. Slight variations over the previous results can be attributed to the randomness of the execution of the code.

Table 8: Performance summary using pragma directives.

Algorithm	Avg Gflop/s	Avg Percentage
Naive, three-loop dgemm	2.14	5.24%
Reference dgemm	40.69	101.69%
Blocked dgemm	4.33	10.93%

Table 9: Performance summary using mavx2 compilation flag and pragma directives.

Algorithm	Avg Gflop/s	Avg Percentage
Naive, three-loop dgemm	2.01	5.13%
Reference dgemm	43.72	102.02%
Blocked dgemm	4.56	11.68%

Finally, we tried usign the **-mavx2** flag to enable the use of the AVX2 instruction set, which is supported by the AMD EPYC 7763 CPU. The results of the performance of the blocked algorithm using this flag is shown in Table **??**. Figure **??** shows the performance of the three algorithms using the previously discussed compiler flags and pragma directives. Using the graph, we can analyze more quantitatively the performance of the three algorithms. We can see that the performance of the naive algorithm is significantly lower than the performance of the reference and blocked algorithms. We didn't expect the blocked algorithm to perform better than the reference algorithm since the OpenBLAS library is highly optimized for matrix-matrix multiplication. However, we can see that the performance of the blocked algorithm performed significantly better than the naive algorithm and doesn't suffer from dips in performance as the size of the matrix increases. This is due to the blocked algorithm's ability to utilize the cache more effectively and reduce the number of cache misses. We tried extending the blocked matrix algorithm to include the L2 chache using the same logic and structure but failed in the implementation. Looking back at the original implementation of the blocked algorithm, we can see that its performance improved significantly by using the optimization flags and pragma directives. The final result of 11.68% shows a 852% improvement over the first block algorithm implementation.

References

[1] uops.info. `https://uops.info/table.html`. [Online; accessed 28-February-2024].

[2] Advanced Micro Devices, Inc. AMD EPYC 7H12 Processor. `https://www.amd.com/en/products/cpu/amd-epyc-7H12`, 2019. [Online; accessed 27-February-2024].

[3] Advanced Micro Devices, Inc. AMD EPYC 7763 Processor. `https://www.amd.com/en/products/cpu/amd-epyc-7763`, 2021. [Online; accessed 27-February-2024].

[4] Georg Hager. Is AMD's EPYC the right architecture for extreme-scale computing? `https://blogs.fau.de/hager/files/2021/10/sc21-01_architecture.pdf`, 2021. [Online; accessed 27-February-2024].

[5] Intel. Intel oneAPI DPC++/C++ Compiler Developer Guide and Reference. `https://cdrdv2.intel.com/v1/dl/getContent/792222?fileName=dpcpp-cpp-compiler_developer-guide-reference_2024.0-767253-792222.pdf`, 2024. [Online; accessed 5-Mars-2024].

[6] ETH Zurich. Euler. `https://scicomp.ethz.ch/wiki/Euler`. [Online; accessed 28-February-2024].

../task_4/timing_mavx2.pdf

Figure 2: Performance of the three algorithms using several compiler flags including mavx2 and pragma directives.

Appendix

I apologize for the inconvenience of the image sizes and the orientation. The output of the image was given by the software, and I had to rotate them to fit the page and become most readable.

../task_2_1/eulerVII_I/EPYC_7H12.pdf

Figure 3: Memory Hierarchy of the AMD EPYC 7H12 Processor.

../task_2_1/EulerVII_II/EPYC_7763.pdf

Figure 4: Memory Hierarchy of the AMD EPYC 7763 Processor.