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

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

Student: Dennys Mike Huber Discussed with: Alberto Finardi & Leon Ackermann

Solution for Project 1

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)*

Question 1 What is the module system and how do you use it?

The module system allows the user to switch between different versions of pre-installed programs and libraries. Each module comes with its own bundle of programs and libraries, which can be loaded using the `module` command as follows:

```
module load [module-name]
```

Internally the `module` command interprets pre-configured `modulefiles`, which typically instruct the `module` command to adjust or set shell environment variables such as `PATH`, `MANPATH`, etc, giving the user an easy way to access and switch between different compilers, loader, libraries and utilities. In order to discover, which modules are available on the current system, simply run the following command. [1, 7]

```
module avail
```

Further details about a specific module can be displayed by using the following command.

```
module show [module-name]
```

Additionally the following commands can be used to manipulate the current set of loaded modules.

```
module list [module-name] # lists all currently loaded modules
module switch [module-name] # Switches loaded module1 with module2.
module rm [module-name] # unloads module
module purge  # unloads all modules
```

Question 2 What is Slurm and its intended function?

The Simple Linux Utility for Resource Management (Slurm) is a cluster management and job scheduling system with the intend to be fault-tolerant and highly scalable. It is widely used on supercomputers in order to simplify the job execution of parallel jobs by hiding several complexities, including load management, job scheduling etc. Slurm comes with three core functionalities:

1. Allocation of access to resources to users.
2. Framework for starting, executing and monitoring of jobs.
3. Arbitration of contended resources.

Slurm is aware of all the available computing resources of the system, which includes number of cores and nodes, memory of each core, sockets, number of GPUs and number of hyperthreads. After successfully submitting a job, Slurm will place the job in a queue based on the requested resources and priorities. If the resources become available, the job will be executed according to the scheduling policy and can then be monitored using various Slurm utilities.[1, 8]

Question 3 Write a simple "Hello World" C/C++ program which prints the host name of the machine on which the program is running.

The program prints the environment variable `HOSTNAME` of the node it is running on.

```cpp
#include <cstdlib>
#include <iostream>

int main() {

  const char *hostname = std::getenv("HOSTNAME");

  if (hostname) {
    std::cout << "This program is running on " << hostname << "." << std::endl;
  } else {
    std::cout
        << "Environment variable for HOSTNAME was not found on your maschine."
        << std::endl;
  }

  return 0;
}
```

Listing 1: Hello World

Question 4 Write a batch script which runs your program on one node. The batch script should be able to target nodes with specific CPUs with different memories. You can obtain the information on available nodes using the command sinfo.

In order to figure out which specific features are available on the Rosa cluster, after consulting the manual for `sinfo` [9] one can run the following command.

```
# Command
sinfo -o "%N %f"
# Output
NODELIST AVAIL_FEATURES
icsnode[01-39,41-42] (null)
```

Listing 2: Detecting features on Rosa

Here `-o` specifies the output format and `%N %f` specifies the output format to be a string and display the list of nodes and the list of features associate to it. Unfortunatly this cluster does not have any features, as seen by the output of listing 2 therefore the `--constraint` option cannot be applied. The batch file is given in Listing 3, notice that the `--nodes` and the `--ntask` are set to one.

```bash
#!/bin/bash
#SBATCH --job-name=slurm_job_one       # Job name    (default: sbatch)
#SBATCH --output=slurm_job_one-%j.out # Output file (default: slurm-%j.out)
#SBATCH --error=slurm_job_one-%j.err  # Error file  (default: slurm-%j.out)
#SBATCH --nodes=1                      # Number of nodes
#SBATCH --ntasks=1                     # Number of tasks
#SBATCH --cpus-per-task=1              # Number of CPUs per task
#SBATCH --time=00:01:00                # Wall clock time limit

# load some modules & list loaded modules
module load gcc
module list

```

```
# print CPU model
lscpu | grep "Model name"

# run (srun: run job on cluster with provided resources/allocation)
srun ./hello_world
```

Listing 3: single node sbatch script

Question 5 Write another batch script which runs your program on two nodes.

In comparison to the previous script the important changes are that `--ntasks` and `--nodes` are now set to two. Resulting in the hello_world program being run twice. Depending on the load on Rosa both instances of the program can be executed either on two separate nodes or on the same node. When I ran this sbatch script I encounter the latter scenario.

```
#!/bin/bash
#SBATCH --job-name=slurm_job_two       # Job name    (default: sbatch)
#SBATCH --output=slurm_job_two-%j.out  # Output file (default: slurm-%j.out)
#SBATCH --error=slurm_job_two-%j.err   # Error file  (default: slurm-%j.out)
#SBATCH --nodes=2                      # Number of nodes
#SBATCH --ntasks=2                     # Number of tasks
#SBATCH --cpus-per-task=1              # Number of CPUs per task
#SBATCH --time=00:01:00                # Wall clock time limit

srun ./hello_world
```

Listing 4: two nodes and tasks sbatch script

2. Performance Characteristics *(30 Points)*

2.1. Peak Performance

As stated on the Compute Resources Webpage of the Institute of Computing [1], Rosa is a high performance computing cluster made up of 42 Nodes, each with two Intel XEON E5-2650 v3 CPUs with 10 cores each. To calculate the Peak performance of the cluster, we begin by calculating the peak performance of a single core P_{core} given by the equation 1.

$$\begin{aligned} P_{\text{core}} &= n_{\text{super}} \times n_{\text{FMA}} \times n_{\text{SIMD}} \times f \\ &= 2 \times 2 \times 4 \times 2.3 \text{ GHz} \\ &= 36.8 \text{ GFlops/s} \end{aligned} \tag{1}$$

By consulting Intel's ARK website [4] about the XEON E5-2650 v3 CPUs, we can find out further details about the CPU, firstly that it operates on a base frequency f of 2.3 GHz, secondly that it is part of the *Haswell* microprocessor architecture family and thirdly that the CPU supports Intel's AVX2 SIMD instruction set extension. The AVX2 SIMD instructions [3] uses a 256-bit wide vector registers and because we consider 64-bit double precision float operations to compute the peak performance, each core is able to perform 4 operations simultaneously. Hence leading n_{SIMD} to be 4. Furthermore, by looking at the website uops.info [10] and filtering the table using the keywords `Haswell` and `FMA` we can see that the throughput (TP) is 0.5, this means that a core can execute two float point operations per cycle, resulting in $n_{\text{super}} = 2$. In addition the port column shows 1*p01, indicating that one vector FMA operation can be executed on each port 0 and 1. Hence we can set n_{FMA} to 2 as well.

After successfully calculating the peak performance of a single core, the result can now be used to compute the peak performance of a CPU, given by equation 2.

$$\begin{aligned} P_{\text{CPU}} &= n_{cores} \times P_{core} \\ &= 10 \times 36.8 \text{ GFlops/s} \\ &= 368 \text{ GFlops/s} \end{aligned} \tag{2}$$

The number of physical cores n_{cores} can be found on the previously mentioned Intel ARK website [4]. Now we can compute the peak performance of a single Node on the Rosa cluster given by

$$\begin{aligned} P_{\text{node}} &= n_{\text{sockets}} \times P_{\text{CPU}} \\ &= 2 \times 368 \text{ GFlops/s} \\ &= 736 \text{ GFlops/s.} \end{aligned} \tag{3}$$

n_{sockets} can be set to 2, because for each we Node we have two sockets as stated on the USI's Institute for Computing website [1]. Now we are finally ready to compute the peak performance of the 42 node cluster giving us the final peak performance of

$$\begin{aligned} P_{\text{Rosa}} &= n_{\text{nodes}} \times P_{\text{node}} \\ &= 42 \times 736 \text{ GFlops/s} \\ &= 30912 \text{ GFlops/s} = 30.912 \text{ TFlops/s.} \end{aligned} \tag{4}$$

2.2. Memory Hierarchies

The memory hierarchy of a single node on the Rosa cluster can be seen in Figure 1. It shows two identical CPU's with 10 cores and the corresponding Level of caches including their size.

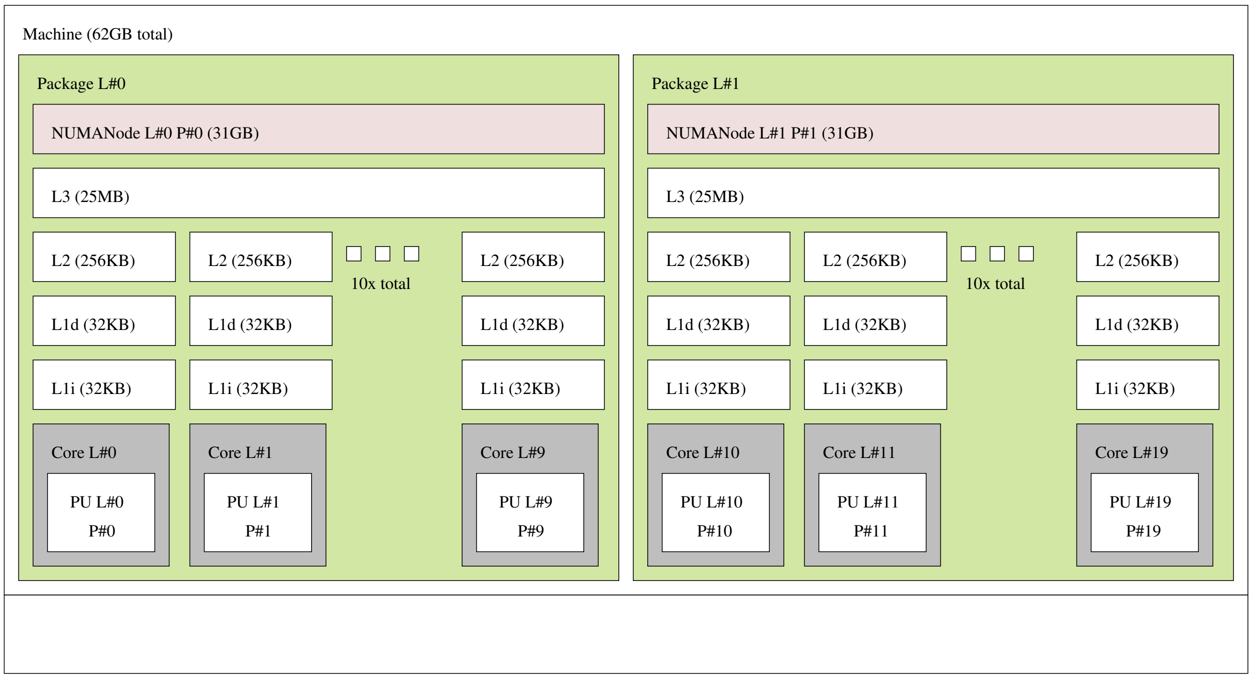

Figure 1: 2 x Intel Xeon E5-2650 v3 @ 2.30GHz, 20 (2 x 10) cores

For single core the memory hierarchy is summarized in Table 1.

Main Memory	31 GB
L3 cache	25 MB
L2 cache	256 KB
L1 cache	32 KB

Table 1: Memory hierarchy of a Rosa node with an Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz

2.3. Bandwidth: STREAM Benchmark

After illustrating the topology of single node, we are now interested in the speed of the memory system. This speed is measured in terms of bandwidth, indicating the how much data can be transfer within a specific time period. The STREAM is a common benchmark to assess the bandwidth. It uses four operations: Copy, Scale, Add and Triad to evaluate the bandwidth and the rate of computing. In order to properly setup benchmark we need to compute the number of element, which is, according to instructions given on the STREAM benchmark website [6], four times the sum of all last-level cache. This results in the following array size n_{arr}:

$$\begin{aligned} n_{\text{arr}} &= 4 \times n_{\text{cores}} \times \text{L1 cache} \\ &= 4 \times 10 \times 32 \text{ KB} \times 1000 \\ &= 1'280'000 \end{aligned} \tag{5}$$

```
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           19199.2     0.106757     0.106671     0.106884
Scale:          11241.1     0.182266     0.182188     0.182441
Add:            12299.4     0.249855     0.249768     0.250008
Triad:          12294.1     0.249960     0.249875     0.250129
-------------------------------------------------------------
```

Listing 5: Output of STREAM benchmark

The Copy function is ignored and significantly higher due to several factors, which is beyond the scope of this report. The observed best rate for Scale, Add and Triad are approximately similar, therefore we can take as a rough estimate the maximum bandwidth as

$$b_{\text{STREAM}} \approx 12\text{GB/s}. \tag{6}$$

2.4. A simple roofline model

As a final step the values computed in previous section are now coming together under one roof. Using the peak performance of a single core, as seen in equation 1 and the maxium bandwidth, calculated in 6, we can construct a simple roofline model. The performance potential of Rosa is shown in 2 with the ridge point being located at $I_{\text{ridge}} \approx 3$, separating the Bandwidth bound region to the left and the compute bound region to the right. This roofline model can now provided us a reference guide when improving the performance of our code on Rosa.

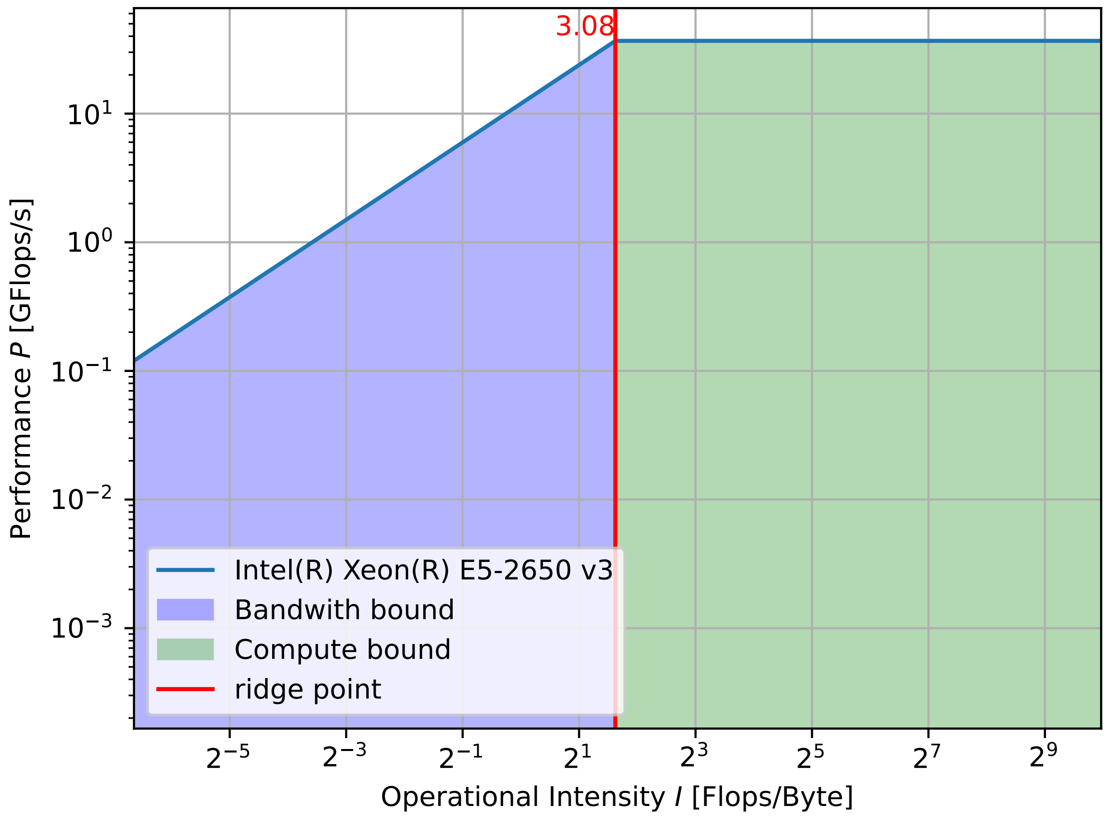

Figure 2: Naive roofline model for Intel Xeon E5-2650 v3 @ 2.30GHz

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

3.1. Implementation

As given in the exercise, the optimal block size is

$$s = \sqrt{\frac{M}{3}} \tag{7}$$

where M is the size of the fast memory. In our case this is the size of the L1 cache, which is 32KB as seen in table 1. In our dgemm-blocked implementation we use the double type, which has size 64 bit or 8 byte. Therefore we can convert our L1 cache size into bytes and divide it by the size of the double type in order to get M.

$$M = \frac{32000}{8} = 4000 \text{ doubles} \tag{8}$$

After finding M, we can now calculate s:

$$s = \sqrt{\frac{4000}{3}} = 36.5148 \approx 36 \text{ blocks} \tag{9}$$

We can now insert this result into the blocked dgemm code given in Listing 6. Shifting our attention to the actual implementation of Blocked dgemm. The goal is to improve cache utilization by optimizing the memory access pattern and reduce cache misses.
The two outer loops split the matrix up into smaller blocks of the previously calculated block sizes s. The next two inner loops go through the individual entry of each block and make sure if the final block cannot be completely filled due to the matrix dimension, that there's not an out-of-bounds error. The innermost loop iterates over the shared dimension. Finally the computation is then performed according to the column-major format. Notice that an additional variable sum is introduced, which reduces the times we have to write to the C matrix.

```c
 const char *dgemm_desc = "Blocked dgemm.";

// Optimize for single core
#define BLOCK_SIZE 36

/* This routine performs 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.
 */
void square_dgemm(int n, double *A, double *B, double *C) {
  double sum;
  for (int bi = 0; bi < n; bi += BLOCK_SIZE) {
    for (int bj = 0; bj < n; bj += BLOCK_SIZE) {
      for (int i = bi; i < bi + BLOCK_SIZE && i < n; i++) {
        for (int j = bj; j < bj + BLOCK_SIZE && j < n; j++) {
          sum = C[i + j * n];
          for (int k = 0; k < n; k++) {
            sum += A[i + k * n] * B[k + j * n];
          }
          C[i + j * n] = sum;
        }
      }
    }
  }
}

```

Listing 6: Dgemm blocked

This initial dgemm block implementation shows a significant improvement compared to the naive dgemm implementation as depicted in Figure 3

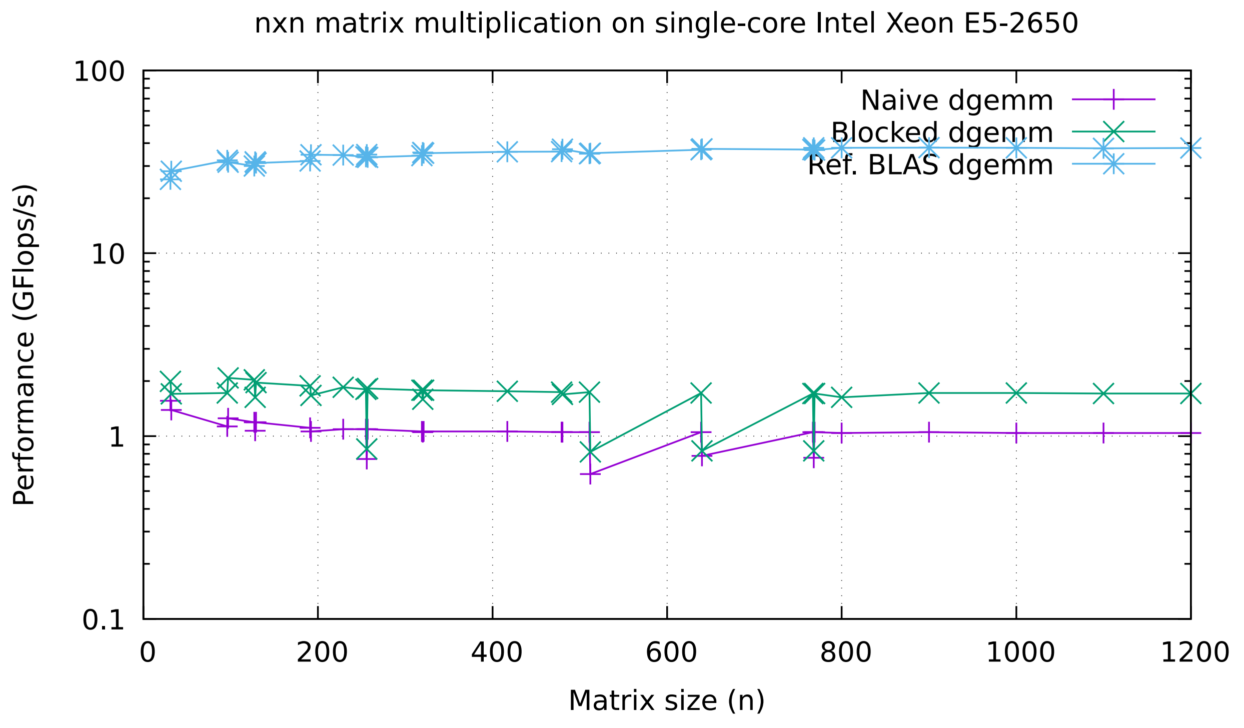

Figure 3: Timing of initial block matrix implementation

3.2. Unrolling Loops and Pragma statements

As an attempt of improving the performance further, the innermost loop was replaced by the unrolled loop seen in Listing 7. This loop is unrolled by a factor of 4, because this is equivalent to the number of doubles the FPU can process concurrently with a single SIMD instruction.

```
int k;
for (k = 0; k <= n - 4; k += 4) {
  sum += A[i + k * n] * B[k + j * n];
  sum += A[i + (k + 1) * n] * B[(k + 1) + j * n];
  sum += A[i + (k + 2) * n] * B[(k + 2) + j * n];
  sum += A[i + (k + 3) * n] * B[(k + 3) + j * n];
}
// For the final block if the number of elements is not divisible by 4
for (; k < n; k++) {
  sum += A[i + k * n] * B[k + j * n];
}
```

Listing 7: Unrolled loop

As a second approach the `#pragma gcc ivdep` [5] directive was introduced in order to force the compiler to unconditionally vectorize the innermost loop. Both approach did not show any improvement in performance compared to the initial block matrix implementation in Figure 3. Hence these approaches were subsequently removed from my final blocked dgemm implementation, but the output file for this option can be found in the submission. These approaches were most likely unsuccessful due to the compiler being able of doing these optimization on its own. This theory is reinforced by having a look at the provided compiler flags in the `Makefile` this includes flags such as `-funroll-loops` and `-march=native`.

3.3. Compiler Flags

By introducing further compiler flags, it is possible to increase the performance even more. Especially the addition of `-ffast-math` [2] resulted in an significant uptick in performance, see Figure 4. It enables a set of flag that enables aggressive floating-point optimizations, but be aware `-ffast-math` is not always safe.

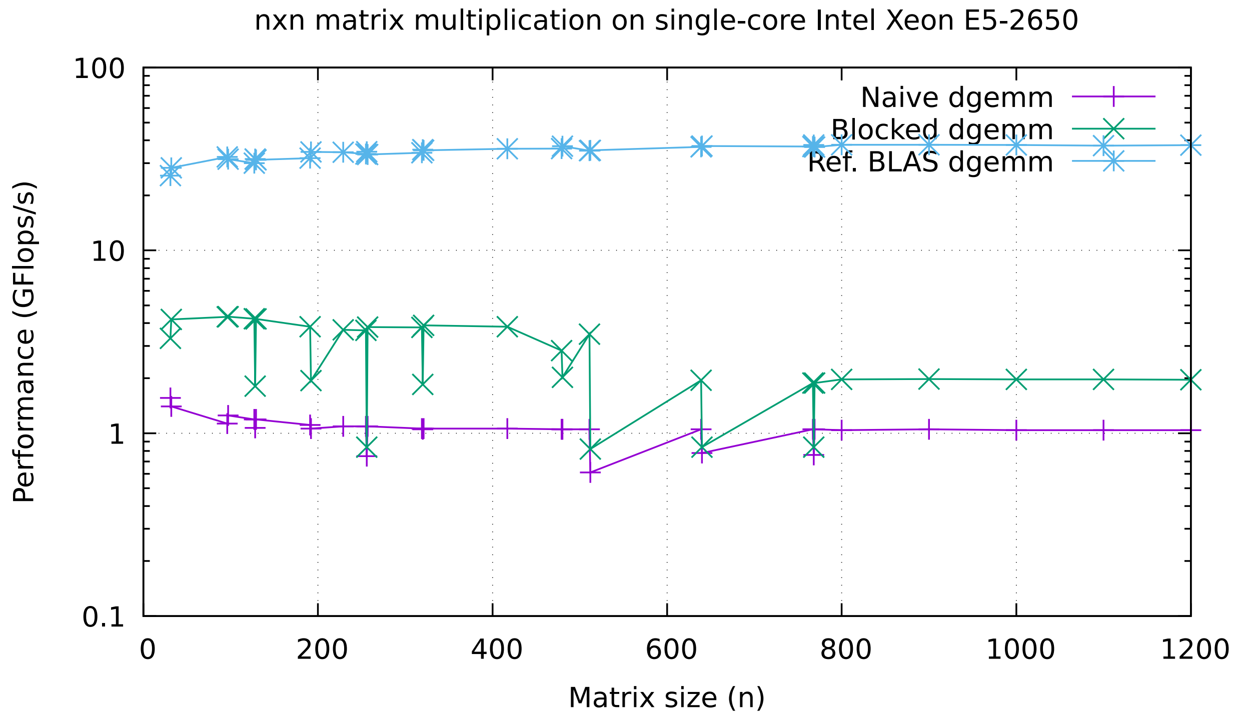

Figure 4: Timing of dgemm using ffast-math

References

[1] *Compute Resources – Institute of Computing USI.* en-US. URL: https://www.ci.inf.usi.ch/research/resources/ (visited on 10/05/2024).

[2] *FloatingPointMath - GCC Wiki.* URL: https://gcc.gnu.org/wiki/FloatingPointMath (visited on 10/07/2024).

[3] *Intel® Intrinsics Guide.* en. URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html (visited on 10/09/2024).

[4] *Intel® Xeon® Processor E5-2650 v3 (25M Cache, 2.30 GHz) Product Specifications.* en. URL: https://www.intel.com/content/www/us/en/products/sku/81705/intel-xeon-processor-e5-2650-v3-25m-cache-2-30-ghz.html (visited on 09/29/2024).

[5] *Loop-Specific Pragmas - Using the GNU Compiler Collection (GCC).* URL: https://gcc.gnu.org/onlinedocs/gcc-4.9.0/gcc/Loop-Specific-Pragmas.html (visited on 10/08/2024).

[6] *MEMORY BANDWIDTH: STREAM BENCHMARK PERFORMANCE RESULTS.* URL: https://www.cs.virginia.edu/stream/ (visited on 10/08/2024).

[7] *module(1): command interface to Modules package - Linux man page.* URL: https://linux.die.net/man/1/module (visited on 10/05/2024).

[8] *Slurm Workload Manager - Overview.* URL: https://slurm.schedmd.com/overview.html (visited on 10/05/2024).

[9] *Slurm Workload Manager - sinfo.* URL: https://slurm.schedmd.com/sinfo.html (visited on 10/09/2024).

[10] *uops.info - Table.* URL: https://uops.info/table.html (visited on 09/29/2024).