Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Student: Benedict Armstrong Discussed with: Tristan Gabl

Solution for Project 01

Due date: 11 March 2024, 23:59

1. Euler warm-up [10 points]

1.1. Module System

The module system allows Euler users to quickly and easily configure their environment to use centrally installed software package. A detailed description can be found in the Module System documentation.

There are two systems currently in use. The older system is called `Environment Modules` and the newer system is called `LMOD Modules`. All new software installations are done with LMOD Modules.

```
# List all available modules
module avail

# Load a module
module load <module_name>

# list all loaded modules
module list
```

Listing 1: Module System

1.2. SLURM

The Euler cluster uses SLURM to manage and schedule jobs. To run a job on the cluster, you need to submit a job script to the SLURM scheduler. A detailed description can be found in the SLURM documentation.

1.3. Hello Euler!

We start by compiling and running a simple C program on the Euler cluster. The program is called `hello_euler.cpp` and should print "`Host name: <hostname>`" to standard out.

To run the compiled program on the cluster, we need to submit a job script to the SLURM scheduler. The job script is called `hello_euler.slurm` and should look like this:

The job can then be submitted to the SLURM scheduler with the following command:

The code and output can be found in the `hello_euler` directory.

```
#!/bin/bash
#SBATCH --job-name=hello_euler          # Job name    (default: sbatch)
#SBATCH --output=hello_euler.out        # Output file (default: slurm-%j.out)
#SBATCH --error=hello_euler.err         # Error file  (default: slurm-%j.out)
#SBATCH --time=00:01:00                 # Wall clock time limit
#SBATCH --nodes=1                       # Number of tasks
#SBATCH --ntasks=1                      # Number of tasks
#SBATCH --cpus-per-task=1               # Number of CPUs per task
#SBATCH --mem-per-cpu=1024              # Memory per CPU
#SBATCH --constraint=EPYC_9654

srun hello_euler
```

Listing 2: Job script for running hello_euler.cpp

```
    sbatch hello_euler.sh
```

Listing 3: Submitting a job to the SLURM scheduler

```
#!/bin/bash
#SBATCH --job-name=hello_euler_2        # Job name    (default: sbatch)
#SBATCH --output=hello_euler_2.out      # Output file (default: slurm-%j.out)
#SBATCH --error=hello_euler_2.err       # Error file  (default: slurm-%j.out)
#SBATCH --time=00:01:00                 # Wall clock time limit
#SBATCH --nodes=2                       # Number of tasks
#SBATCH --ntasks=2                      # Number of tasks
#SBATCH --cpus-per-task=1               # Number of CPUs per task
#SBATCH --mem-per-cpu=1024              # Memory per CPU

srun hello_euler hello_euler
```

Listing 4: Job script for running hello_euler.cpp on multiple nodes

1.4. Multiple Nodes

We can run the same code on multiple nodes using the following job script:

Where we set the number of nodes to 2 and the number of tasks to 2. The output can be found in the `hello_euler_2.out` file.

2. Performance characteristics [50 points]

2.1. Peak performance

The peak performance of a cluster can be calculated using the following formula:

$$\begin{aligned}
p_{core} &= n_{super} \times n_{FMA} \times n_{SIMD} \times f_{core} \\
p_{CPU} &= n_{core} \times p_{core} \\
p_{node} &= n_{sockets} \times p_{CPU} \\
p_{cluster} &= n_{nodes} \times p_{node}
\end{aligned}$$

The the **Euler VII — phase 1** and **Euler VII — phase 2** nodes use the *EPYC_7H12* and *EPYC_7763* cpus, respectively.

Parameter	**Euler VII — phase 1**	**Euler VII — phase 2**	source
CPU	EPYC_7H12	EPYC_7763	Euler docs
n_{super}	2	2	UOPS Website ($= TP^{-1}$)
n_{FMA}	2	2	UOPS Website
n_{SIMD}	4	4	en.wikichip.org
f_{core}	2.6 GHz	2.45 GHz	Euler docs
n_{core}	64	64	Euler docs
$n_{sockets}$	2	2	Euler docs
n_{nodes}	292	248	Euler docs

Table 1: Parameters of the Euler VII — phase 1 and Euler VII — phase 2 nodes

Using the values from the table 1 we can calculate the peak performance of the Euler VII — phase 1 and Euler VII — phase 2 nodes.

Euler VII — phase 1:

$$\begin{aligned}
p_{core} &= 2 \times 2 \times 4 \times 2.6\,\text{GHz} = 41.6\,\text{GFLOPS} \\
p_{CPU} &= 64 \times 41.6\,\text{GFLOPS} = 2662.4\,\text{GFLOPS} \\
p_{node} &= 2 \times 2662.4\,\text{GFLOPS} = 5324.8\,\text{GFLOPS} \\
p_{cluster} &= 292 \times 5324.8\,\text{GFLOPS} = \underline{1554.8\,\text{TFLOPS}}
\end{aligned}$$

Euler VII — phase 2:

$$\begin{aligned}
p_{core} &= 2 \times 2 \times 4 \times 2.45\,\text{GHz} = 39.2\,\text{GFLOPS} \\
p_{CPU} &= 64 \times 39.2\,\text{GFLOPS} = 2508.8\,\text{GFLOPS} \\
p_{node} &= 2 \times 2508.8\,\text{GFLOPS} = 5017.6\,\text{GFLOPS} \\
p_{cluster} &= 248 \times 5017.6\,\text{GFLOPS} = \underline{1244.4\,\text{TFLOPS}}
\end{aligned}$$

2.2. Memory Hierarchies

The output of running `lscpu` and `hwloc-ls` can be found in the `memory_hierarchies` directory. As in the example in the assignment there are also two PDFs detailing the memory hierarchy of the EPYC_7H12 and EPYC_7763 nodes. Both nodes have 8 NUMA nodes, with 8 cores per NUMA node. More information on NUMA can easily be found in the Wikipedia page. Basically it means that the nodes have faster access to their specific part of the shared memory. The rest of the numbers can easily be read out of the two PDFs detailing the memory hierarchy. The main difference between the two nodes is the size of the L3 cache, which is 16MB for the EPYC_7H12 and 32MB for the EPYC_7763. The main memory size is the same for both nodes at 248GB.

2.2.1. Cache and main memory size

Cache	**EPYC_7H12**	**EPYC_7763**
L1d	32KB	32KB
L1i	32KB	32KB
L2	512KB	512KB
L3 (shared)	16MB (4 cores)	32MB (8 cores)
NUMA	31GB	31GB
Total Machine	248GB	248GB

Table 2: Cache and main memory size for both nodes

2.3. Bandwidth: STREAM benchmark

As per the STREAM benchmark documentation we need to set DSTREAM_ARRAY_SIZE to be a four times the L3 cache size.

For the EPYC_7H12 node the cache size (2) is 16MB.

$$16\text{MB} = 1.6e7\text{B}$$

So we set DSTREAM_ARRAY_SIZE to be $1.6e7 \times 4/8 + 2e6 = 10e6$ (we add a little extra as recommended).

```
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           25125.0     0.006416     0.006368     0.006472
Scale:          18591.8     0.008643     0.008606     0.008667
Add:            19702.7     0.012235     0.012181     0.012269
Triad:          20075.2     0.012017     0.011955     0.012046
```

The peak bandwidth for the EPYC_7763 node is around 20GB/s.

The same calculation for the EPYC_7763 node gives us a DSTREAM_ARRAY_SIZE of $32\text{MB} \times 4/8 + 4e6 = 20e6$.

```
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           35390.3     0.009231     0.009042     0.009746
Scale:          25082.3     0.012878     0.012758     0.013321
Add:            25780.4     0.018783     0.018619     0.019019
Triad:          26000.8     0.018654     0.018461     0.019720
```

The peak bandwidth for the EPYC_7763 node is around 25GB/s.

The entire output of the STREAM benchmark for both CPUs can be found in the `stream_benchmark` directory.

2.4. Performance model: A simple roofline model

Using the STREAM benchmark results and the peak performance of the CPUs we can create a simple roofline model. The peak performance of the EPYC_7H12 and EPYC_7763 nodes are 41.6GFLOPS and 39.2GFLOPS, respectively. The peak bandwidth for the EPYC_7H12 and EPYC_7763 nodes are 20GB/s and 25GB/s, respectively.

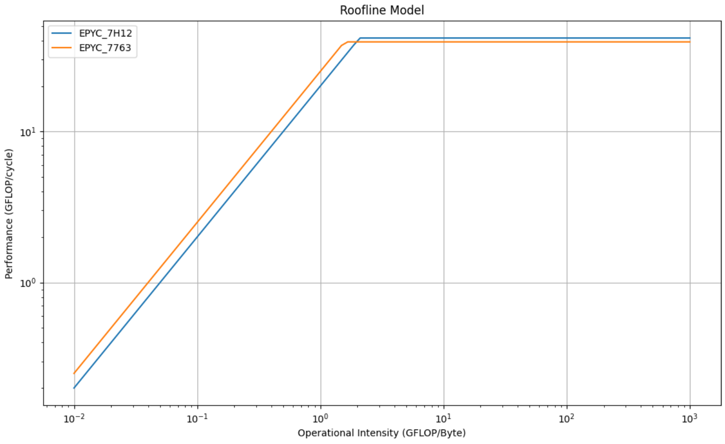

Figure 1: Roofline model for the EPYC_7H12 and EPYC_7763 nodes

The ridge point for the EPYC_7H12 is around $I_{ridge} = 2$, and for the EPYC_7763 node is around $I_{ridge} = 1.6$.

3. Auto-vectorization

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

   It is important for data structures to be aligned because the CPU can only load and store data from memory in chunks of a certain size. If the data is not aligned, the CPU will have to load and store the data in multiple chunks, which is less efficient. For example, if the CPU can load 128 bits at a time, and the data is not aligned, the CPU will have to load 64 bits, then 64 bits, which is less efficient than loading 128 bits at once.

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

   Some obstacles that can prevent automatic vectorization by the compiler are:

   - The code is not written in a way that the compiler can understand and optimize.
   - The code contains dependencies that prevent the compiler from reordering instructions.
   - Complex loop terminations
   - The code contains loops that the compiler cannot unroll.
   - Using pointers instead of arrays

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

   Yes, there are ways to help the compiler to vectorize. For example, you can use compiler directives to give the compiler hints about how to vectorize the code. You should also write code in a way that the compiler can understand and optimize, for example by using simple loops, avoiding dependencies and using arrays instead of pointers.

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

   The compiler can perform several loop optimizations to vectorize and pipeline loops. For example, the compiler can unroll loops to expose more instruction-level parallelism, and it can reorder instructions to eliminate dependencies. The compiler can also use loop interchange to improve data locality, and it can use loop fusion to combine multiple loops into a single loop. The compiler can also use loop tiling to break up large loops into smaller loops that fit in the cache.

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

   If automatic vectorization by the compiler fails or is sub-optimal, you can try to rewrite the code in a way that the compiler can understand and optimize (See previous points). You can also use compiler directives to give the compiler hints about how to vectorize the code. Examples are `#pragma ivdep` and `#pragma vector align`.

4. Matrix multiplication optimization

The goal of this task was to improve the performance of a simple matrix multiplication program. To do this I implemented blocked matrix multiplication.

```c
void square_dgemm(int n, double *A, double *B, double *C) {

  for (int i = 0; i < n; i += BLOCK_SIZE) {
    for (int j = 0; j < n; j += BLOCK_SIZE) {
      for (int k = 0; k < n; k += BLOCK_SIZE) {
        // Correct block dimensions if block "goes off edge of" the matrix
        int M = (i + BLOCK_SIZE < n) ? BLOCK_SIZE : n - i;
        int N = (j + BLOCK_SIZE < n) ? BLOCK_SIZE : n - j;
        int K = (k + BLOCK_SIZE < n) ? BLOCK_SIZE : n - k;
        // Perform individual block dgemm
        for (int i2 = 0; i2 < M; ++i2) {
          for (int j2 = 0; j2 < N; ++j2) {
            for (int k2 = 0; k2 < K; ++k2) {
              C[(i + i2) + (j + j2) * n] +=
                  A[(i + i2) + (k + k2) * n] * B[(k + k2) + (j + j2) * n];
            }
          }
        }
      }
    }
  }
}
```

One important parameter is the block size. To evaluate different block sizes I added a macro `BLOCK_SIZE` to the code. I then compiled the code with different block sizes from 2 to 256. Plotting the performance of the blocked matrix multiplication for different block sizes we see that we get the best performance for a block size of around 10. This result was not expected, as the L1 cache size is 32KB, which using the calculation from the task description

$$32\text{KB} = 32000\text{B} \rightarrow 32000\text{B}/8 = 4000\,\text{doubles} \rightarrow \sqrt{\frac{4000}{3}} \approx 36$$

would give us a theoretical ideal block size of 36.

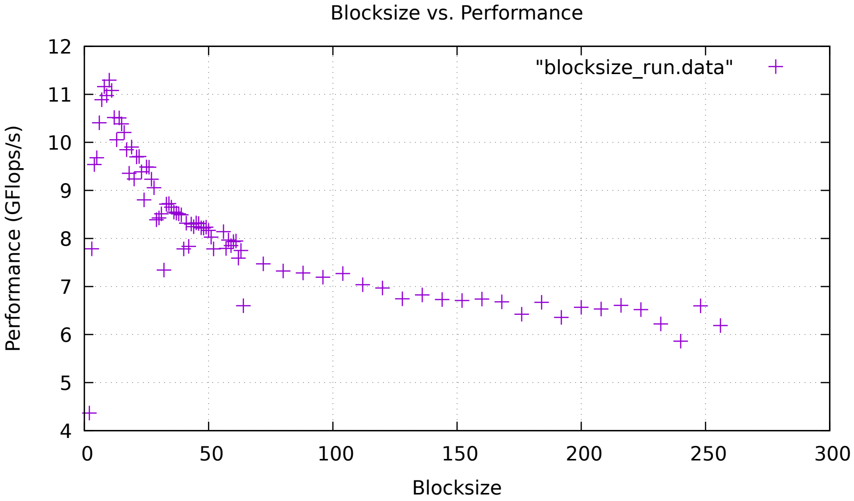

Figure 2: Performance of blocked matrix multiplication for different block sizes

The entire output of the blocked matrix multiplication for different block sizes can be found in the `BLOCKSIZE_RUNS` directory.

Running the benchmark against the naive and BLAS implementation we see that the blocked matrix multiplication is faster than the naive implementation, but slower than the BLAS implementation. To evaluate the performance fairly I compiled all three versions with the `-O3` flag.

The naive implementation hovers at around 2GFLOPS, the blocked implementation at around 4GFLOPS and the BLAS implementation goes up to around 50GFLOPS.

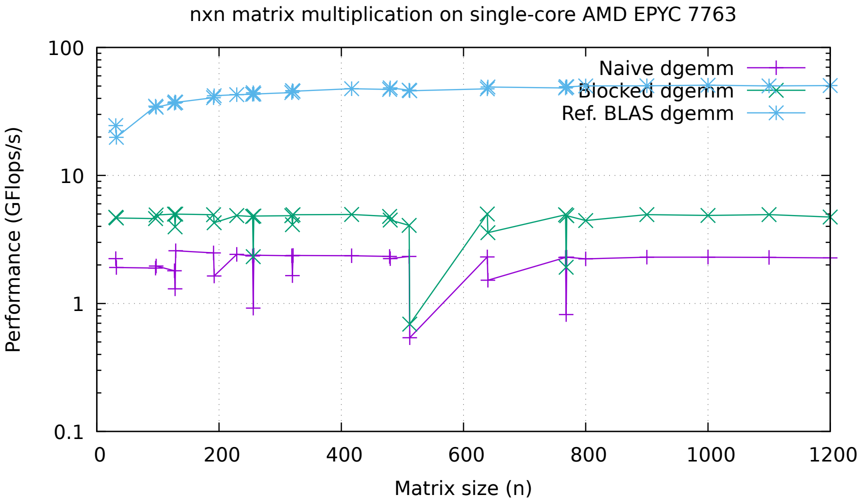

Figure 3: Performance of naive, blocked and BLAS matrix multiplication