Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Student: Noah Gigler Discussed with: Felicia Scharitzer, Luis Wirth, Ankush Majmudar, Ben Armstrong

Solution for Project 1a

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]

1. The module system is a tool used to manage software environments on a Euler. It allows us to configure their environment by dynamically loading or unloading software modules. These modules adjust system variables to ensure that the necessary binaries and libraries are accessible. You use it by loading specific software versions with module load and unloading them with module unload when done.
2. Slurm is a tool used in big computer clusters to help manage who gets to use the computers and when. It schedules tasks and makes sure everything runs smoothly by allocating resources like processors and memory. It's like the traffic controller for a cluster of computers.
3. see hostname.cpp
4. see slurm_job_one.sh
5. see slurm_job_two.sh

2. Performance characteristics [50 points]

2.1. Peak performance

Source: `https://scicomp.ethz.ch/wiki/Euler#Euler_VII_.E2.80.94_phase_1`

Table 1: Euler VII Phase 1 and Phase 2 Specifications

Phase	Compute Nodes	CPUs per Node	CPU	Clock Speed (GHz)
Phase 1	292	2	AMD EPYC 7H12	2.6
Phase 2	248	2	AMD EPYC 7763	2.45

$$n_{\text{super}} = \frac{1}{TP} = 2$$
$$n_{\text{FMA}} = 2$$
$$n_{\text{SMID}} = 4$$

Values are the same for both Euler VII Phase 1 and 2.
Source for FMA, TP and : `https://uops.info/table.html`
Source for the SIMD values: "Software Optimization Guide for AMD EPYC™ 7002 Processors" and "Software Optimization Guide for AMD EPYC™ 7003 Processors"

$$P_{\text{core}} = n_{\text{super}} \cdot n_{\text{FMA}} \cdot n_{\text{SMID}} \cdot f$$
$$P_{\text{CPU}} = P_{\text{core}} \cdot \#\text{Cores}$$
$$P_{\text{node}} = P_{\text{core}} \cdot \#\text{CPUs}$$
$$P_{\text{EulerVII}} = P_{\text{node}} \cdot \#\text{Nodes}$$

Table 2: Peak Performance Comparison

Metric	Phase 1	Phase 2
P_{core}	41.6 GFLOP/s	39.2 GFLOP/s
P_{CPU}	2.66 TFLOP/s	2.51 TFLOP/s
P_{node}	5.32 TFLOP/s	5.02 TFLOP/s
P_{EulerVII}	1.55 PFLOP/s	1.24 PFLOP/s

2.2. Memory Hierarchies

2.2.1. Cache and main memory size

Table 3: Cache and Main Memory Sizes

Phase	L1 (KB)	L2 (KB)	L3 (MB)	Main Memory (GB)
Phase 1	32	512	16	256
Phase 2	32	512	32	256

The only difference between the cache sizes of Phase 1 and Phase 2 is the L3 cache size. Phase 2 has twice the L3 cache size of Phase 1.

2.3. Bandwidth: STREAM benchmark

As we can see, the bandwidth of the AMD EPYC 7763 is higher than the AMD EPYC 7H12 for all functions. The Copy function generally has the highest bandwidth. The other functions have similar bandwidths. We can assume that the bandwidth of the AMD EPYC 7763 is 25Gb/s and the bandwidth of the AMD EPYC 7H12 is 20Gb/s.

Table 4: Memory Bandwidth Comparison

Function	AMD EPYC 7H12	AMD EPYC 7763
Copy	30804.4 MB/s	34873.5 MB/s
Scale	19134.3 MB/s	24681.9 MB/s
Add	21820.0 MB/s	25473.4 MB/s
Triad	21891.7 MB/s	25710.9 MB/s

2.4. Performance model: A simple roofline model

The formula for the naive roofline model is:

$$P = \begin{cases} \pi & \pi < \beta \cdot I \\ \beta \cdot I & \text{else} \end{cases}$$

Where π is the peak performance and I is the operational intensity and β is the peak bandwidth. Here we are looking at the performance of a single core so we can use the peak performance we calculated in task 2.1. The peak bandwidth was calculated in task 2.3.

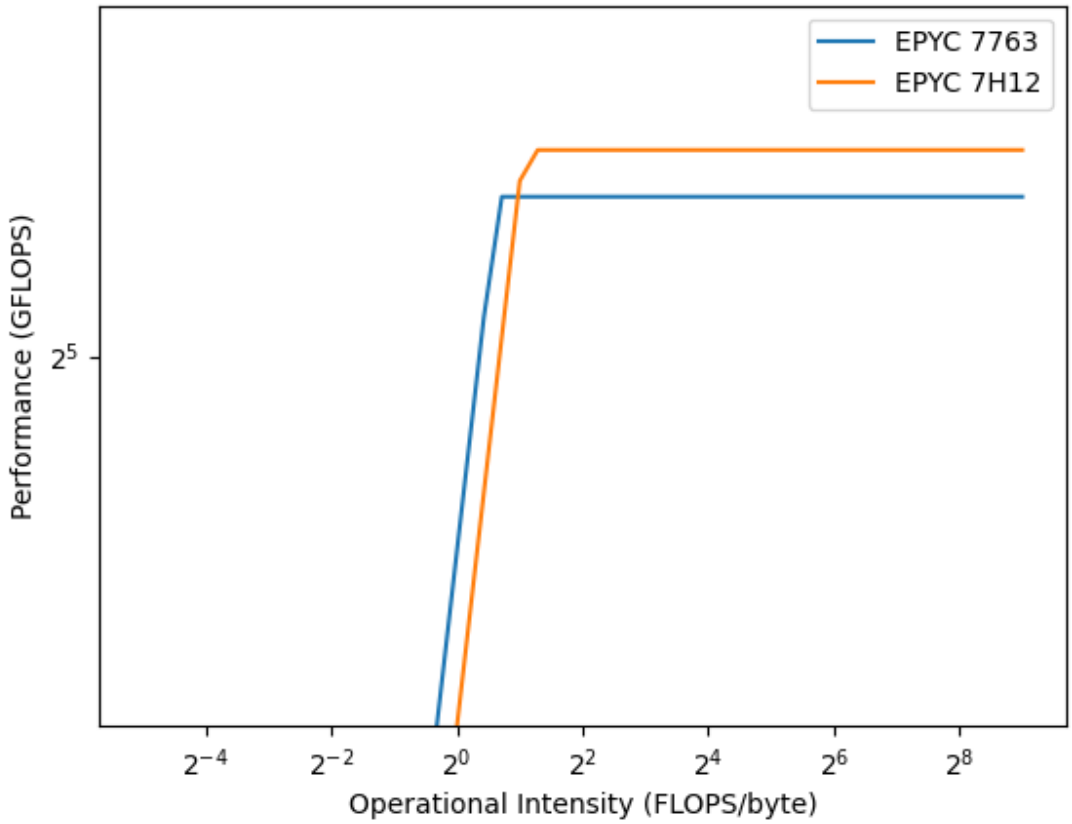

Figure 1: Roofline Model

As we can see from the formula, the performance is limited by the bandwidth when $\pi < \beta \cdot I$ and by the peak performance when $\pi \geq \beta \cdot I$. In the graph this can be seen visually. As long as the performance is still increasing we are limited by the bandwidth and once the performance plateaus we are limited by the peak performance.

3. Auto-vectorization [10 points]

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

Alligned data structures are important because the vectorization process requires that the data be contiguous in memory and alligned. Unalligned memory access can cause a performance penalty. If allignment is not specified, the compiler will be forced to generate runtime alignment optimization in order to be able to vectorize the code, which will cause unecessary overhead or even prevent the vectorization.
2. What are some obstacles that can prevent automatic vectorization by the compiler?

- Function calls
- Data dependencies
- Data-dependent loop exit conditions
- Non-contiguous data

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

Yes, we can provide the compiler with extra information in the form of pragmas or compiler flags.

Some examples:

```
#pragma vector align
```

for example can assert that the data is alligned in memory which saves the compiler from having to generate runtime alignment optimization.

```
#pragma ivdep
```

Tells the compiler to ignore potential dependencies in the loop and vectorize it anyway. This can be helpful if the compiler is not able to detect that the loop is safe to vectorize but we insure that it is in some other part of our code.

4. Which loop optimizations are performed by the compiler to vectorize and pipeline loops?
Loop unrolling: The compiler will generate code that executes multiple iterations of the loop at once. This is done to increase the number of iterations that are executed in parallel.
Strip mining: The compiler will split the loop into smaller loops that are executed in parallel. This can increase the temporal and spacial locality of the data.

5. What can be done if automatic vectorization by the compiler fails or is sub-optimal? First we can try to provide the compiler with more information to help it vectorize the code. If it still fails we will have to manually vectorize the code. This can be done by using intrinsics or by using a library that provides vectorized operations.

4. Matrix multiplication optimization [30 points]

The first optimization we do is to use blocking to improve the temporal and spacial locality of the data. Here we split the matrix into smaller blocks and multiply these blocks instead of the entire matrix. This will increase the temporal and spacial locality of the data and reduce the number of cache misses.
It is important to maximize the stride one access pattern to maximize the performance of the cache. This leads us to changing the order of the loops from the naive i,j,k to j,k,i. We then notice that in the last loop where we iterate over i we always multiply the same elements of the matrix B. We can therefore load this element into memory only once before each iteration of the innermost loop and save a lot of memory accesses.

In order to find the optimal blocksize we have to look at the cache sizes. We can then use the formula $blocksize = \sqrt{\frac{\text{cache size}}{3 \times \text{sizeof(double)}}}$ as we have to fit 3 blocks into the cache at the same time. Plugging the L1 cache size of 32KB into the formula we get a blocksize of 36 which is way to small to be efficient. Using the L2 cache size of 512KB we get a blocksize of 146 which is a good starting point. Through test we see that the performance difference between 100 and 150 is not very big. Anything above 150 will start to decrease the performance so we go with the theoretical optimal blocksize of 146.

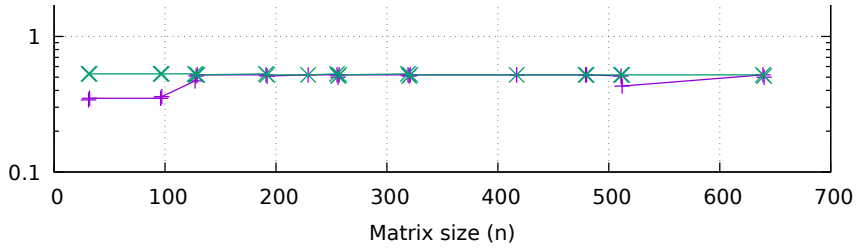

Figure 2: Block size 64

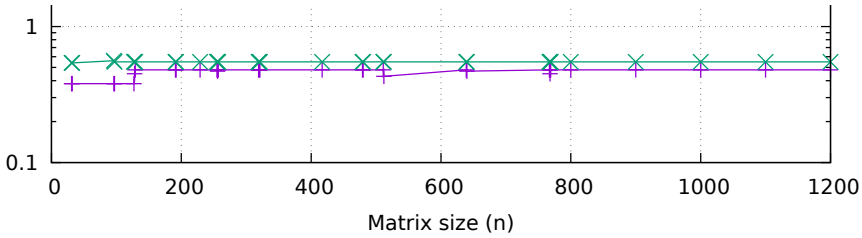

Figure 3: Block size 146

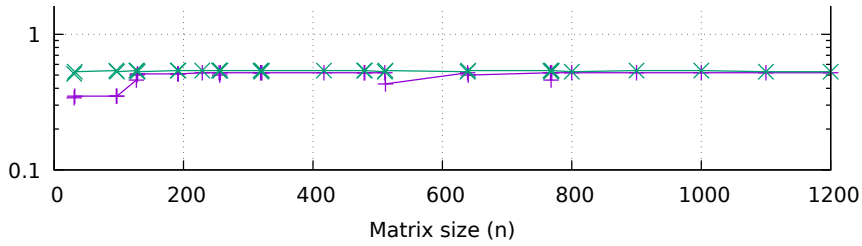

Figure 4: Block size 200

The pragma vector aligned is used to assert that the data is alligned in memory which saves the compiler from having to generate runtime alignment optimization. Another pragma that is used is the ivdep pragma which tells the compiler to ignore potential dependencies in the loop. The same effect can be used by using the restrict keyword to tell the compiler that the pointers are not aliased. This combined with setting the optimization level to -O3 will allow the compiler to perform more aggressive optimizations.
The final simple optimization we do is to use the -march=native flag in order to allow the compiler to generate code that is optimized for the specific architecture that we are running on. This alone adds a significant performance improvement.
After all these optimizations we can see a speedup from about 0.5 GFLOPS to 10 to 15 GFLOPS which is a significant improvement. This is still slower than the performance of specialized libraries such as BLAS which reach speeds of 20 to 30 GFLOPS.

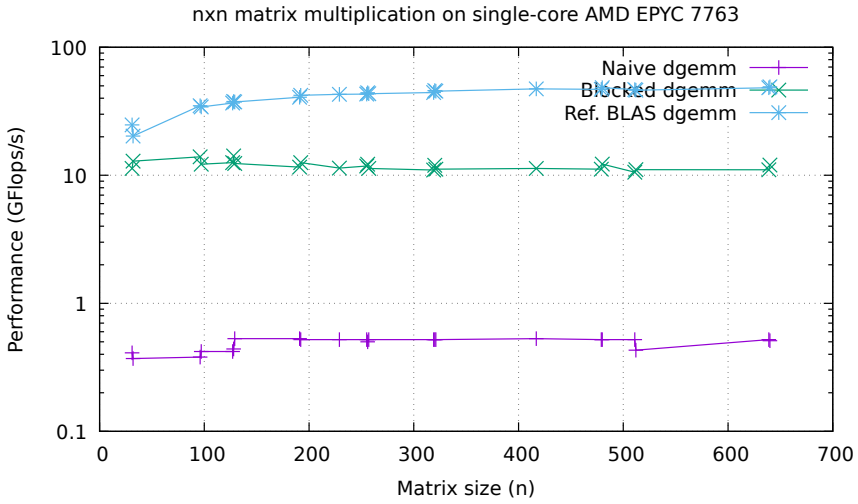

Figure 5: After the above optimizations

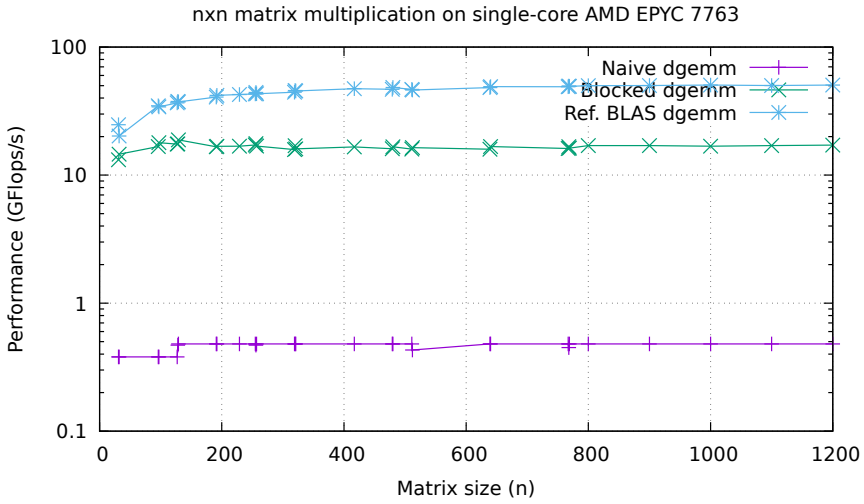

Figure 6: With architecture specific optimizations