Exercises for High Performance Computing
(MA-INF 1108)
WS 2023/2024

E. Suarez, M. Wolter and B. Kostrzewa
Tutors: O. Vrapcani and N. Pillath

---

6 Memory Bandwidth

ATTENTION!: eCampus can become unavailable without previous announcement due to an urgent maintenance. Students are responsible for submitting their checklists enough ahead of time of the deadline. Submissions via Email will not be accepted unless tutors explicitly authorized it beforehand.

This exercise will be performed on the **JURECA-DC system.**

Important recommendations for benchmarking runs (exercises 6.2 and 6.3):

- Try first with just 4-5 different vector lengths until you are sure all your scripts are doing what they should.
- For the final run (after debugging), increase in vector length in factors of 2, starting with a size for which the full problem fits in the L2$, and ending with a size that exceeds the size of the L3$.
- Include in the beginning of your output files a header containing in a few lines all important information of the run conditions, e.g.: name and version of source code, compiler version and flags used, node number on which it run, number of threads and/or processes, etc. See listings example.

```
1 ================================================
2 DAXPY daxpy.c version: 2023
3 gcc -O1 -fopt-info
4 gcc (GCC) 11.3.0
5 Scalar Version (no OpenMP or MPI)
6 Run: Node: jrc0736 ; ntasks: 1
7 ================================================
8         Size     BW[GB/s]   Runtime[ms]
9         2048       39.961         0.001
10        4096       40.289         0.002
11 [...]
```

- Include in your output plots title, axis labels (with units) and legend, so that it is clear what is being plot and how the results have been obtained.

1: Theoretical Peak Performance

Estimate the maximum peak performance of a standard compute node in the JURECA-DC.

a) **[1pt]** Use any of the tools presented in the exercise sheet #3 (*Ex03 Hardware characteristics of the processor*) and/or the JSC-system documentation to find out following parameters:

- Vendor
- CPU model
- Cores per socket
- Threads per core
- Basis clock frequency
- Turbo clock frequency
- Capacity of L1D$ per core
- Capacity of L2$ per core
- Capacity of L3$ per core
- Capacity of main memory

b) **[1pt]** Find out through the chip specifications (e.g. via the vendor website or reliable web sources as `wikichip`):

- Amount of FMA units
- Maximum SIMD vector length supported
- Type of DDR memory
- Maximum memory frequency
- Number of memory channels
- Maximum memory bandwidth

c) **[1pt]** Estimate the maximum performance per core.

d) **[1pt]** Estimate the maximum performance for the whole socket.

e) **[1pt]** Most HPC applications are memory bound. Traditionally, the ideal ratio between memory bandwidth and performance was defined to be 1 Byte/FLOP. Estimate the value for our JURECA-DC CPU (socket).

2: Implement daxpy

You will implement and benchmark `daxpy`, a function that performs following operation on vectors:

$$\vec{z} = a \cdot \vec{x} + \vec{y}$$

where a is a scalar, and $\vec{z}$, $\vec{x}$, and $\vec{y}$ are vectors of a given length N containing double precision numbers.

a) **[2pt]** Solve the `TODOs` in the source code `daxpy.c`.

b) **[2pt]** Compile the source code into an executable. Create a script (`run_daxpy.sh`) to run the executable (on the `dc-cpu-devel` partition of the JURECA-DC cluster) for increasing vector lengths, starting by a value for which all vectors fit in the L1D$, and ending by a value in which they do not fit into the L3$. The output of this script should be saved into the file (`daxpy.txt`) in your solutions folder.

c) **[1pt]** Write a script (e.g. `plot_daxpy.py` in python or matplotlib) that plots the memory bandwidth usage in GB/s vs. the memory footprint of `daxpy`. Hint: the memory footprint is the amount of memory used by the function, which you can calculate from the vector size and amount of reads or writes performed per element. You can neglect the impact of the scalar. Store the plot as `daxpy.png`, in your solutions folder.

d) **[1pt]** Add to the plot (`daxpy.png`, in your solutions folder) vertical lines showing the capacity of the L1D$, L2$, and L3$, plus an horizontal line for the memory bandwidth of this CPU according to its specifications (values from exercise 6.1). What do you observe?

e) **[4pt]** Use the pragma `pragma omp parallel for` before your `for` loops, save the source file as `daxpy_omp.c` and compile it. Write a script (`run_daxpy.sh`) to run `daxpy` for increasing number of OpenMP threads, starting by 1 and ending by the maximum number of hardware threads supported by the CPU, using its SMT capabilities. Store the resulting text files on your solutions folder following the naming convention `daxpy_omp<ThreadNum>.txt` (e.g. `daxpy_omp1.txt`, `daxpy_omp2.txt`, etc.). Plot the curves in (`daxpy_omp.png`) in your solutions folder). What do you observe?

3: The Stream benchmark

With this exercise you will learn to run the `stream` benchmark, developed by John D. McCalpin. This benchmark suite has been designed to measure the memory bandwidth on almost any computer platform. It calculates three different vector operations:

$$\begin{aligned} \text{Copy: } & \vec{y} = \vec{x} \\ \text{Scale: } & \vec{y} = c \cdot \vec{x} \\ \text{Add: } & \vec{y} = \vec{x} + \vec{z} \\ \text{Triad: } & \vec{y} = \vec{x} + c \cdot \vec{z} \end{aligned}$$

where $\vec{x}$, $\vec{y}$, and $\vec{z}$ are vectors in double precision of a given length, and c is a scalar.

First of all: Download the stream benchmark from its original website (`https://www.cs.virginia.edu/stream/`) or the git repository:
`https://github.com/jeffhammond/STREAM.git`

a) **[1pt]** Read the provided `README` file and adapt the source file `stream.c` to make sure that you run the benchmark under its defined specifications. Compile it also according to the description.

b) **1pt** Write a script (`run_stream.sh`) to run the `stream` benchmark on a **standard compute node of JURECA-DC** for growing vector sizes (vector length increasing in factors of 2), from a size fitting in the L1D$ up to sizes that do not fit in the L3$. Follow the recommendations at the top of the exercise sheet.

c) **[1pt]** Write a script that extracts the following from the standard output files of `stream`: the vector size, and the `best rate` for the three benchmarks contained in the suite. Print these four values into four columns in a text file (`stream_bw.txt`) in your solutions folder.

d) **[1pt]** Write a script (e.g. `plot_stream.py` in python or matplotlib) that plots (`stream.png`, in your solutions folder) the memory bandwidth (in GB/s) measured by the four `stream` benchmarks vs. their memory footprint. Include in the plot vertical lines marking the physical capacity of the L1D$, L2$, and L3$ caches (values from exercise 6.1), and an horizontal line for the memory bandwidth as given in the chip specifications.

e) **[1pt]** What are the bandwidths measured at the L1D$, L2$, L3$, and main memory? Do the latter match with the specifications of memory bandwidth that you found out in exercise 6.1?

f) **[3pt]** Run now the `stream` benchmarks using OpenMP, following the indications given in the source file. Redo the previous steps, saving the `best rate` for increasing number of OpenMP threads in text files (`stream_omp1.txt`, `stream_omp2.txt`, etc.). Plot the copy, scale, add, and triad results for the run with 64 OpenMP threads (stream_omp64.png), including vertical lines for the L1$, L2$, and L3$ capacities, and an horizontal line for the memory bandwidth given by the chip specifications (see 6.1). Compare your result with the plot you previously obtained with the sequential version.

g) **[1pt]** Produce a plot (`triad_omp.png`) of bandwidth vs. memory footprint with the stream triad results alone, but for increasing number of OpenMP threads.

4: Memory access time

Consider a computer in which the hit time is 1 clock cycle, the miss penalty is 200 clock cycles, the miss rate is 2%, and in average 1.5 memory accesses are done per instruction. Perform following calculations and reply the questions on the eCampus checklist.

a) **[1pt]** How much more time do memory access take due to cache misses? (compared to when there are no misses)

b) **[1pt]** How much slower (`CPU_time`) does the program become?

Consider now a multi-level cache system, in which the base CPI is 1, the clock rate is 4 GHz, the L1$ miss rate is 2%, the L2$ access time is 5 ns, the L2$ miss rate is 0.5%, and the time to access main memory is 100 ns. The effective CPI (CPI_{eff}) can be calculated as:

$$CPI_{eff} = CPI_{base} + (\text{Miss Rate} \times \text{Miss Penalty})$$

c) **[1pt]** Which is the effective CPI when having only L1$?

d) **[1pt]** Which is the performance improvement (based on CPI) when adding the L2$ to the design?

Commit your solutions to the GitHub Classroom.
If you have used Jupyter, close your Jupyter session and stop JupyterLabs.