Praktikum im WS 2023
Evaluierung moderner HPC-Architekturen und -Beschleuniger (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)

Sergej Breiter, MSc., Minh Thanh Chung, MSc., MSc., Amir Raoofy, MSc., Bengisu Elis, MSc., Maron Schlemm, MSc., Dr. Karl Fürlinger, Dr. Josef Weidendorfer

Assignment 01 – Due: 26.10.2023, 16:00

Welcome to BEAST Lab! We hope you will learn a lot about modern computer architecture as part of this lab course. You should already have access to the BEAST Lab machines via SSH and 2FA (if that is not the case, contact the advisors). To get started, we have some advice to help you do well:

- Document your steps and findings for each part of the assignment in the `report` folder using Markdown. The reports contribute to your grade.
- Describing and explaining the observed behavior is generally important to us. Try to determine why you observe the things you do. Make sure your explanations are brief and concise but can still explain the observed behavior well enough.

Vector Triad Microbenchmark

This assignment is concerned with analyzing the performance of the vector triad. The benchmark operates on four vectors $\vec{A}$, $\vec{B}$, $\vec{C}$, and $\vec{D}$ of length N (see Listing 1).

```
for (i=0; i<N; i++) {
    A[i] = B[i] + C[i] * D[i];
}
```

Listing 1: Vector triad

The performance of this code can be measured in floating-point operations per second (FLOPs) and is limited by bandwidth to/from memory or cache, depending on the vector length and cache sizes. We provide an implementation of the triad microbenchmark in `src/triad.c`.

1. **General Questions**

   We start with general questions to gain a better understanding of the code.

   (a) Explain the flop calculation. Think about the number of floating-point operations per iteration of the triad.

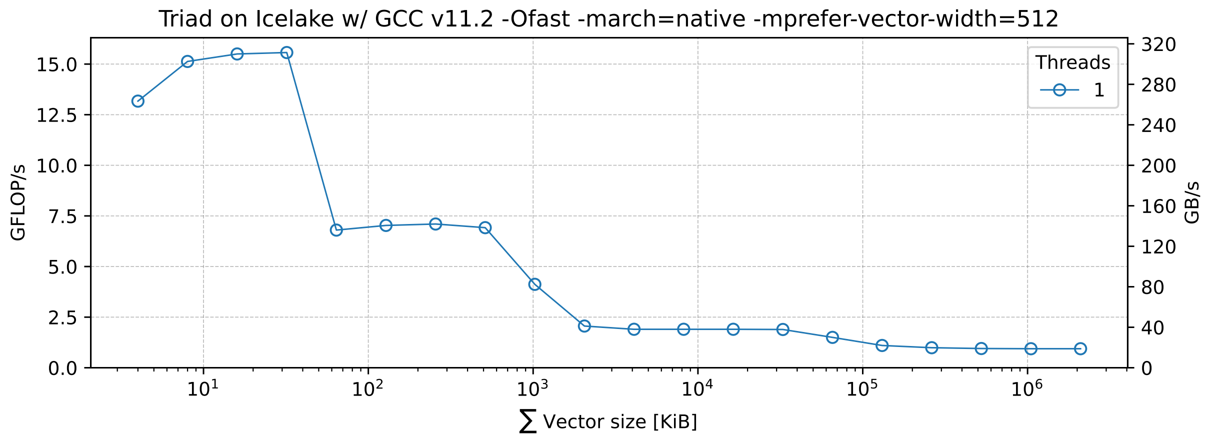

Figure 1: Performance of the sequential vector triad benchmark depending on the vector length. Different levels of caching are visible.

(b) With some simplifications, we can convert FLOPs to bandwidth utilization in bytes/s (memory or cache) by scaling the FLOPs with a factor of 16.

   i. Think about the vector and cache sizes and explain why this is the case in the triad. Hint: calculate the number of bytes transferred by counting the number of loads and stores per iteration.

   ii. Lookup the term *'write allocate'* and explain why the factor 16 may be wrong and give an alternative value. [1]

(c) Code measurement

   i. What is the purpose of the parameter `REP` in the `triad` function?

   ii. What additional effect has the loop for setting the number of repetitions on the subsequent measurement?

   iii. Can you think of any other reason for using the `verify` function?

(d) Optimization

   i. Why we are using `sysconf(_SC_LEVEL1_DCACHE_LINESIZE)` in the aligned memory allocation?

   ii. We use the `restrict` keyword in the declaration of the pointers for the vectors (although not necessary in this code). Explain what the `restrict` qualifier asserts the compiler and why this can be important for vectorization.

(e) Parallelization

   i. Explain the reason and effect of using the `schedule(static)` and `nowait` clauses in the `triad` function. Furthermore, explain why we create the parallel region outside of the repetition loop instead of using `#pragma omp parallel for` on top of the inner loop.

   ii. Why is the initialization parallelized and why do we use `schedule(static)` here?

Experiments and Measurements

In this part, we conduct various experiments and look at the scaling behavior of the triad benchmark on different architectures. To understand and discuss the tasks in this part, you need to read up on *NUMA effects*, *first touch policy*, and *thread pinning*. See the provided

[1]We will measure the actual factor by measuring bandwidth with *performance events* in a future assignment.

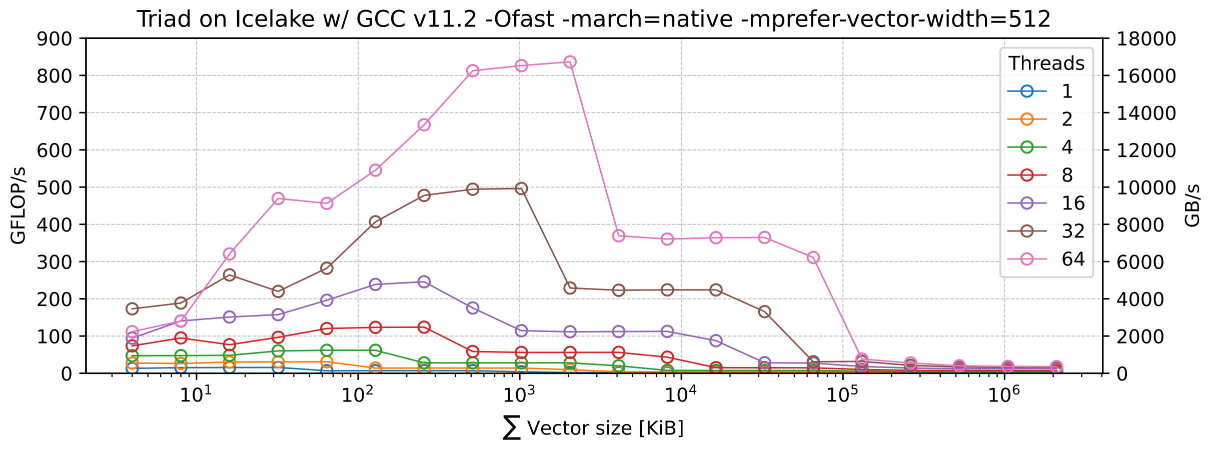

Figure 2: Performance of the parallel vector triad.

`README.md` file for how to build and run the microbenchmark. You can generate graphs using the output of the benchmark.

For the following tasks, we assign a particular architecture to each group according to Table 1. Document the system, environment, compilers, and flags you are using in all the experiments.

GroupID % 4	System	CPU Model
0	milan2	AMD EPYC 7773X
1	thx2	Marvell Thunder-X2
2	ice1	Intel Xeon Platinum 8360Y
3	cs2	Fujitsu A64FX

Table 1: Architecture/system assignment to each group

2. **Sequential Performance**

   First, we look at sequential performance. Set the number of threads to 1, thread binding to `spread` and `N`=2^{26}. Use a **recent** version (GCC 10+) of `gcc` installed on your system, and a compiler provided by the vendor for the architecture assigned to your group. Lookup the cache sizes of your system before you interpret the results.

   (a) Run the triad benchmark to generate plots similar to Fig. 2. We plot the vector length N, or the sum of the vector sizes, on the x-axis and the performance in FLOPs on the y-axis. Add a second axis to your plot showing bandwidth utilization in GB/s (see Task 1 b).

   (b) Experiment with the compiler and optimization flags.

      i. Set the optimization level to `-O2` using `gcc` and compare the performance to a higher optimization level (`-O3` or `-Ofast`). What is the main reason for the performance difference? You can query the set of optimizations enabled at each optimization level with `gcc <-O flag> -Q --help=optimizers` to compare the optimizations.

      ii. Try another compiler and optimization flags. Select the compiler and optimization flags with best performance for subsequent tasks.

   (c) Compute the maximum theoretical memory bandwidth of your system and compare it to your measurements. What may be the reason for the difference? What consequence does this have for parallel code?

   (d) Measure performance without specific memory alignment (using `malloc`). Can you measure a difference on your system?

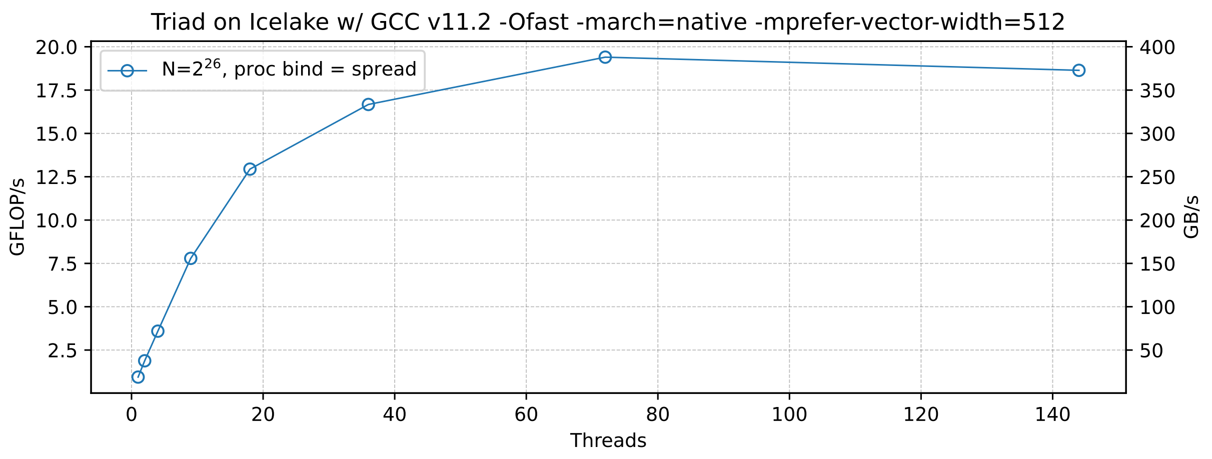

Figure 3: Vector triad strong scaling graph with N=2^{26}.

3. **Parallel Performance**

   Lookup the number of NUMA domains (N) of your system, the number of physical cores (C), and the number of virtual cores (T) (taking SMT into account). Use different numbers of threads $\in \{1, 2, 4, \ldots, N, C, T\}$ in the following tasks. When interpreting the observed results, think about NUMA and thread affinity. [2]

   (a) Measure the triad performance with different numbers of threads and plot similar graphs to Fig. 2. Add a second axis for the bandwidth utilization. Describe your observations. What is the maximum memory bandwidth utilization measured? Compare it to the theoretical peak bandwidth from your earlier calculation.

   (b) In this part, we perform strong scaling experiments measuring memory bandwidth utilization. In such experiments, we use a constant data set size and vary the number of threads as parameter instead (see Fig. 3).

   Set the vector length to a constant value large enough such that data is always transferred to / from main memory (e.g. N=2^{27}). Use the provided code and environment for the first subtask. In the following subtasks, you will repeat the experiment with minor modifications to the code or environment. Compare the results to the first subtask and explain the difference between the scaling curves and the effect of the modifications.

      i. Modify the provided triad code to allow for strong scaling experiments. Plot the FLOPs rate and bandwidth utilization as a function of the number of threads. Describe the scaling behavior.
      ii. Change the thread binding from `spread` to `close`.
      iii. Disable the parallelization of the initialization.
      iv. Change the scheduling policy to `schedule(dynamic)` or `schedule(static,1)`.
      v. Remove the `nowait` clause in the triad.

   (c) Now we set the vector length small enough such that the vectors fit in cache (e.g. N=2^{17}). Repeat the scaling experiments (b) i-iii with the smaller vectors. Additionally, compare the scaling behavior to the previous task and explain the difference.

   (d) Bonus: repeat previous experiments on other platforms of BEAST. Make a comparison between the scaling plots of different systems.

[2]You can check the thread affinity by setting `OMP_DISPLAY_AFFINITY=true`.