Praktikum im SS 2022
Quantitative Analyse von Hochleistungssystemen (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)

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

Assignment 01 – Due: 04.05.2022, 14:00

Welcome to BEAST Lab! We hope that 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 (if that is not the case, contact the advisors). To get you started, we have some advice to help you do well:

- Document your steps and your findings for each part of the assignment well. The report makes up the most of your grade. Don't forget to add your explanations in the report especially for the tasks in which explanations are explicitly asked for.
- In general explanation of the observed behavior is important to us. Try to determine why you observe the things you do.Make sure your explanations are brief but can still explain the observed behavior well enough.

We wish you good luck with this first assignment.

Vector Triad Microbenchmark

This assignment is concerned with analyzing the performance of vector triad microbenchmark. The benchmark works with four vectors $\vec{A}$, $\vec{B}$, $\vec{C}$, and $\vec{D}$ of length N and implements an element-wise addition and multiplication according to the following formula:

$$\vec{A} = \vec{B} + \vec{C} \circ \vec{D} \tag{1}$$

This formula is implemented using the following loop:

```
1 for (long i=0; i<N; i++){
2     A[i] = B[i] + C[i] * D[i]
3 }
```

The performance of this code can be measured in floating-point operations per second (FLOPs). This performance depends mostly on bandwidth from memory or cache, depending on the size of the input vectors, i.e., N, see Fig. 1.
This assignment provides a simple implementation of a triad microbenchmark and walks you through various experiments to evaluate and measure its performance. In each subtask, you need to make a small adjustment in the code and therefore we provide some hints to help you.
Document your work in this assignment's `report/` folder using Markdown.

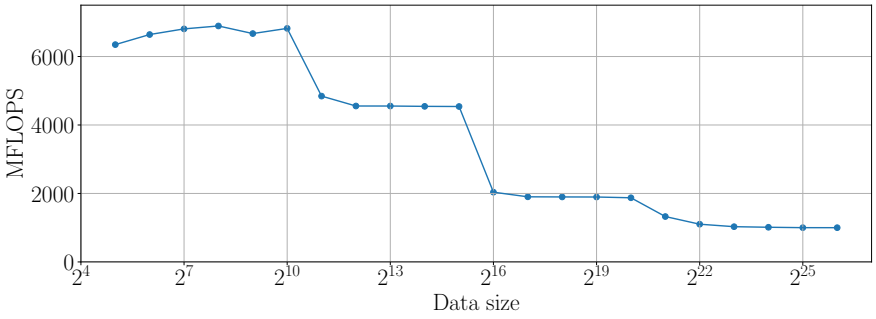

Figure 1: Performance of the sequential vector triad benchmark depending on the vector length on the Icelake machines. Different levels of caching are visible.

1. **General Questions**

   We first start with general questions that will help you to understand the code. Check the source code in `src/triad.c` file.

   (a) Explain the logic of the loop in line 116 and elaborate on how the size of the vectors and iterations are changed in the provided code. Furthermore, explain the flop calculation logic (line 121). Hint: think about the number of floating-point operations (excluding loads and stores) performed for each scalar calculation.

   (b) The microbenchmark is implemented using a function `triad` in line 28. Try to explain the parameters and returning value of this function. Specifically, what is the purpose of parameter `REP`?

   (c) Since in the provided code the function `triad` takes care of all the microbenchmark code including threading, we are querying the number of threads in line 36 using `omp_get_num_threads()`. Explain what the reason for creating a parallel region in line 34 is.

   (d) From lines 40 to 43, Explain why we are using `sysconf(SC_LEVEL1_DCACHE_LINESIZE)` in the first argument of `aligned_alloc`.

   (e) Lines 46-52 initialize all of the vectors (including the output vector $\vec{A}$). Explain why the initialization loop is parallelized and why `schedule(static)` is used.

   (f) While the microbenchmark kernel is implemented in lines 64-70 (wrapped between the time measurement points), we also perform the microbenchmark loop earlier in the code in lines 55-61. Explain what the reason for this is.

   (g) For parallelizing the triad loop (line 68), we use `schedule(static)` and `nowait` clauses. Explain the reason for using these clauses and the implications of using these clauses on the microbenchmark measurements. Furthermore, explain why we are creating a parallel region on line 65 instead of line 68 (i.e., explain why we avoid using `#pragma omp parallel for` in line 68). Test this by changing the code to create a parallel region inside the repetition loop, and explain the performance differences.

   (h) In line 75, we use function `calculateChecksum` to calculate the checksum of vector $\vec{A}$ after running the microbenchmark loops. Apart from verifying the results, can you think of any other reason for using this checksum function?

2. **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, and thread pinning.

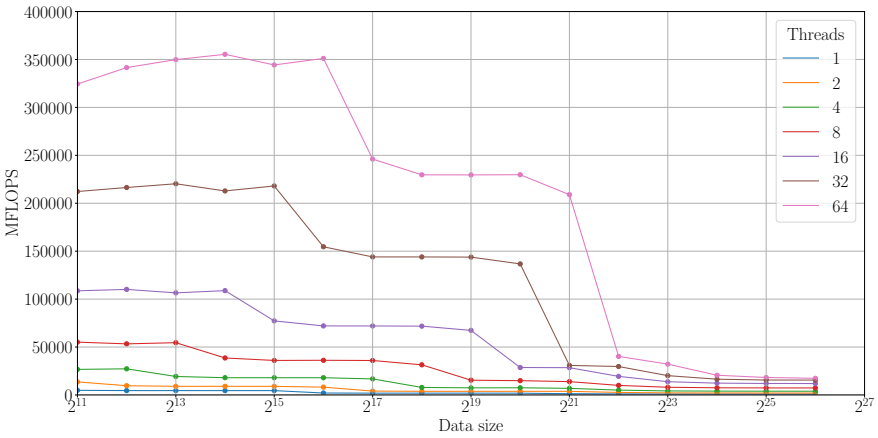

Figure 2: Performance of the parallel vector triad benchmark depending on the vector length on the Icelake machines. Compared to the sequential implementation, there is some overhead for certain dataset sizes (i.e., $N < 1024$ which is not shown in the graph).

We provide some hints in a `README.md` file next to the source code to help you build and run the microbenchmark. A simple example on the Icelake system is the following:

```
1 make CC=cc
2 export OMP_NUM_THREADS=2 OMP_PLACES=sockets OMP_PROC_BIND=spread
3 ./triad 1024 1024
```

With this configuration, we use the default SUSE compiler v7.5.0 for building. Furthermore, we use 2 threads and specify `spread` as binding which results in running 1 thread on each socket. The vector size and repetition are both set to 1024 and the output of the code looks like the following:

```
N = 1024 and REP = 1024. Performance in MFLOPS.
| Dataset size |       Threads |        MFLOPS |        Cycles |
|           32 |             2 |        355.43 |            32 |
|           64 |             2 |        361.32 |            16 |
|          128 |             2 |        291.44 |             8 |
|          256 |             2 |        322.26 |             4 |
|          512 |             2 |        285.75 |             2 |
|         1024 |             2 |        315.85 |             1 |
```

Using the reported outputs, you can generate graphs similar to what we present in Fig. 2.

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

(a) In the first step, run the triad benchmark by setting the number of threads to 1, thread binding to `spread` and `N`=2^{26} and `REP`=2^{26}. Visualize the results and generate similar plots to Fig. 1 (You should even observe higher performance!).

(b) With a lot of simplifications, for *triad* we can convert the GFLOPS to memory bandwidth utilization (GB/s) by simply scaling the GFLOPS values with a factor of 16.0. Explain why this is the case, and estimate and plot the bandwidth utilization of the triad microbenchmark. Hint: check the hint in question 1.a. Plot a bandwidth utilization graph with the

GroupID % 4	Architecture/System
0	rome2
1	thx2
2	ice2
3	cs2

Table 1: Architecture/system assignment to each group

data size (N) as the parameter. Here you need to count the number of loads and stores instead of the arithmetic operations.

(c) Compute the maximum theoretical memory bandwidth of the BEAST system assigned to your group and compare the results of your bandwidth measurements in 2.b to the maximum theoretical memory bandwidth. Bonus: In case you plan to work on part 2.k (the other bonus part), compare the percentage of peak bandwidths among the systems in BEAST can get with 1 thread. What may be the reason for the difference? What consequence does this have for parallel code?

(d) Repeat the experiments in part 2.a with different numbers of threads up to the number of cores in the system and plot similar graphs to Fig. 2. Also similar to 2.b plot a bandwidth utilization graph with the data size, N, as the parameter.

(e) Modify the provided triad code to allow for strong scaling experiments: In such experiments, we use a constant data set size (`N=constant`). We then use the number of threads as the parameter and evaluate the FLOPS and bandwidth utilization (you still need to adjust the repetition parameter properly). Plot the achieved FLOPS rate and bandwidth utilization as a function of the number of threads, and explain the results. Use different numbers of threads including $1, 2, \ldots, N, C, T$, where N is the number of NUMA domains of the system, C is the number of physical cores, and T is the number of virtual cores (taking SMT into account). Note that for this task you need to set the dataset size to a constant value which is big enough to cause the data to be loaded from the main memory. For example, on Icelake system (again see Fig. 2) we can simply set N=2^{27}, so that for all different number of threads used data will be loaded/stored from/to the main memory. *Hint: When interpreting the observed results, think about NUMA and thread affinity.*

(f) Repeat the scaling experiments in part 2.e by disabling the parallelization of the loop in line 45. Compare the results with the results you achieved in 2.e and explain your observations.

(g) Change the scheduling policy in lines 46, 58, and 68 of the code and repeat the experiments in part 2.e (e.g., use `schedule(dynamic)` or `schedule(static,1)`). Compare the results with the results you achieved in 2.e and explain your observations.

(h) Remove the `nowait` clauses in lines 58 and 68, and repeat the experiments in part 2.e. Compare the results with the what you achieved in 2.e and explain your observations. Repeat this experiment with a smaller data set size (e.g., residing on L3), and explain your observations.

(i) Consider the following 4 cases and repeat the experiments in 2.e for all of them.

- Case 1: N=2^{17}, binding=close
- Case 2: N=2^{17}, binding=spread
- Case 3: N=2^{27}, binding=close
- Case 4: N=2^{27}, binding=spread

What kind of scaling is visible for all 4 cases? Can you explain why? Specifically, explain the scaling behavior between N=2^{17} and N=2^{27}, and the difference between the scaling curves of 'close' and 'spread' bindings.

(j) Repeat the experiments in part 2.i by using a different compiler on the system. Ideally, one of the selected compilers should be a native compiler provided by the vendor for the architecture assigned to your group (e.g., `icc` for the Intel systems). Explain your observations.

(k) Bonus: repeat previous experiments 2.i and 2.j on all platforms of BEAST (rome2, thx2, ice2, and cs2). Make a comparison between the scaling plots of different systems and explain your observations.