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

Vincent Bode, MSc., Dennis-Florian Herr, MSc., Bengisu Elis,MSc.,
Minh Thanh Chung, MSc., Dr. Karl Fürlinger, Amir Raoofy, MSc., Dr. Josef Weidendorfer

Assignment 02 – Due: May 12, 2022, 14:00

Dense Matrix Multiplication

In this assignment, we investigate dense matrix multiplication of square matrices, i.e., matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ with matrix side lengths $\mathbf{N}$. While the memory consumption for the inputs $\mathbf{B}$, $\mathbf{C}$ and the output $\mathbf{A}$ is $O(N^2)$, for computing the matrix matrix multipication $\mathbf{A} = \mathbf{B} * \mathbf{C}$, the number of operations is $O(N^3)$. Unlike the vector triad in the previous assignment, for matrix multiplication, depending on $\mathbf{N}$, the compute requirements are much higher than data transfer requirements from/to main memory, which typically results in a performance more limited by the compute power of a system (assuming a smart ordering of the operations in such a way that data loaded into a processor cache is reused as often as possible).

The code snippet in Listing 1 shows a naive ordering (`ijk`) of loops and operations, i.e., for each element of $\mathbf{A}$, we perform the dot product of the corresponding row of $\mathbf{B}$ and column of $\mathbf{C}$. However, the code snippet is not optimized for cache access at all. In this assignment, we walk you through simple analyses and measurements to better understand why this is the case. The goal of this assignment is to analyze the performance impact of applying different memory access optimizations. By applying these optimizations, you will be able to achieve decent performance for dense matrix multiplication of BEAST systems.

```
for( int i=0; i<N; ++i )
  for( int j=0; j<N; ++j )
    for( int k=0; k<N; ++k )
      a[i][j] += b[i][k] * c[k][j];
```

Listing 1: Code snippet for dense matrix multiplication benchmark.

Base versions of the code is provided. The performance is measured in floating-point operations per second (FLOPs). The function `mm(N, REP, ...)` receives the side length of the matrices `N` and the parameter `REP` which represents the number of repetitions done in an outer loop (r loop) to result in a minimal runtime for stable measurements. It returns the measured megaflop rate. The repetitions are tuned for each N such that one call of `mm()` requires at least 2 billion multiplications and additions. Figure 1 shows an example performance graph varying N (e.g. curve for the naive ordering).

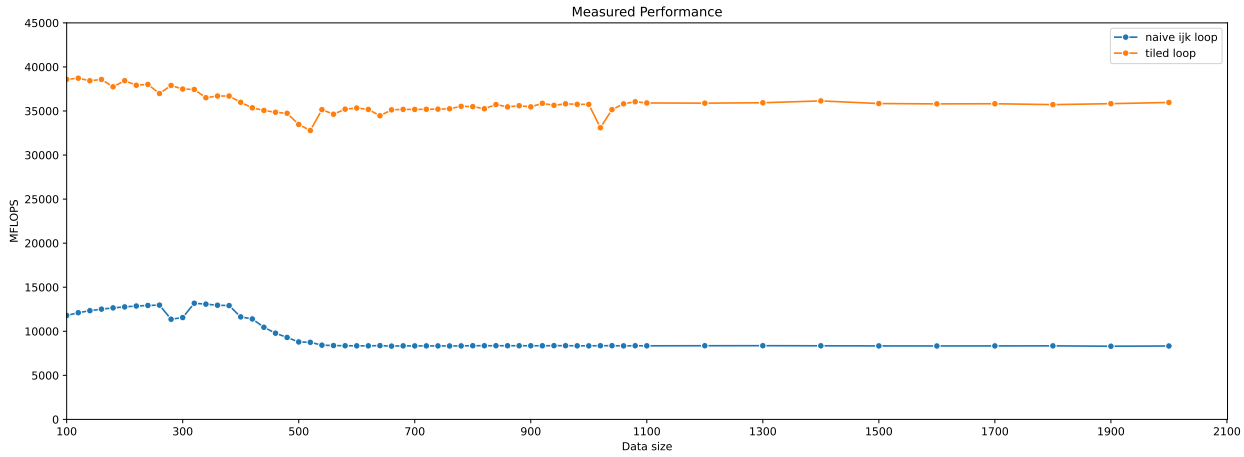

Figure 1: Single-core performance (MFlop/s) of matrix multiplication depending on matrix side length on Icelake system.

1. **General Questions**
   (a) Analyze the data dependencies in the loop nest in the code snipped in Listing 1. Can we safely permute the loops (`i`, `j`, `k`)? Explain which of the loops (`i`, `j`, `k`) is parallelizable?
   (b) Explain alternative parallelization schemes (`#pragma` on different loop nest? or nested parallelization?). Do you expect to get benefits from changing the scheduling of the parallel loop?

2. **Experiments and Measurements**
   (a) Check out the provided source code for subtask (a) and understand the source code. Try to find the best set of compiler flags (e.g. `-O3`, `-Ofast`, `-march=native`), and evaluate the performance (FLOP rates) of the matrix multiplication microbenchmark on all platforms of BEAST. Document and analyze the results, and make sure the measurements are stable (limited run-to-run variability).
   (b) As a next step, permutate the loop order (i.e. change the execution order of, `i`, `j`, and `k`), evaluate the performance similar to subtask (a). Repeat your analysis for all the possible 6 cases: `ijk`, `ikj`, `jik`, `jki`, `kij`, and `kji`, and explain the access pattern to each matrix for each permutation, and discuss whether this correlates with the results of your evaluations. In your discussion consider accesses that result in a cache hit or miss (e.g., in L1, L2, L3 caches). Hint: for analyzing the access pattern, focus on the innermost loop for each permutation.
   (c) Check out the provided source code for subtask (c). It applies loop tiling resulting in cache blocking. The default `TILE_SIZE` parameter is set to 10 in the provided code. Conduct experiments with different tile sizes (`TILE_SIZE` $\in \{4, 5, 10, 20, 50\}$). For each architecture of BEAST, check which setting and strategy are the best, and compare it with the naive version. Try to explain your observations.

      Bonus: the provided code only works in case `N % TILE_SIZE == 0`. Extend the implementation to deal with remainder loops in the tiled loops. This helps to conduct more tuning for tile sizes (`TILE_SIZE` $\in \{4, 8, 12, 16, 20, 25\}$).
   (d) Repeat the experiments in part 2.c by using a different compiler on the system. Ideally, select a compiler provided by the vendor of the compute components of the system. Explain your observations. Furthermore, inspect the generated assembly code of the inner-most loop to see whether vector instructions are used. You can use `objdump` for that.

(e) Parallelize your code variants using OpenMP. Test your parallel versions on all platforms with different numbers of cores as given in assignment 1. Run **strong scaling** experiments and create speedup figures (i.e., for all measurements for one curve, use exactly the same workload = same repetition count): core count on X axis, achieved speedup vs. sequential run on Y axis - i.e., point (1/1) always starts the curve.

Consider the following 4 cases for scaling experiments:

- Case 1: N=100, binding=close
- Case 2: N=100, binding=spread
- Case 3: N=1900, binding=close
- Case 4: N=1900, binding=spread

What kind of scaling is visible for all 4 cases? Can you explain why? Specifically, explain the scaling behavior between N=100 and N=1900, and the difference between the scaling curves of 'close' and 'spread' bindings.

(f) Calculate the peak performance of each platform, using clock rate, core count, number of vector units, size and capability of vector units (assuming a throughput of one FMA vector instruction per clock cycle) for each of the platforms. What percentage of the peak performance can your benchmark achieve?

(g) Can you estimate the main memory bandwidth utilization (GB/s) of the code variant with the highest performance without cache blocking using a matrix size for which the matrices do not fit in the L3 cache (e.g., N=4000)? Why is this estimation difficult in comparison to the vector triad? Can you try to find out how to measure it otherwise (we will discuss this at a later lab date)?

(h) Compare the performance results of the matrix multiplication microbenchmark to the vector triad microbenchmark in assignment 1. For this, draw the roofline for each system (as explained in the lecture slides) and mark the points corresponding to the achieved performance.

(i) Try to search on the Internet to find further optimizations and tuning used for matrix multiplication. Such optimizations allow to reduce the gap between the performance you achieved and the theoretical peak performance. The High-Performance Linpack (HPL) benchmark which is used to rank the systems on the Top-500 list consists mostly of dense matrix multiplication. Try to find entries in the Top500 where nodes are similar to the ones used in this lab. What is the ratio of performance achieved with HPL in comparison to the theoretical peak performance?