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., Amir Raoofy, MSc., Dr. Karl Fürlinger, Dr. Josef Weidendorfer

Assignment 04 – Due: 16.11.2023

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 multiplication $\mathbf{A} = \mathbf{B} * \mathbf{C}$, the number of operations is $O(N^3)$. Therefore, unlike the vector triad in the previous assignment, for matrix multiplication, the number of floating point operations is much higher than the data transfer requirements ($O(N^3)$ vs $O(N^2)$). However, this statement is only valid when we assume a smart ordering of the operations in such a way that data loaded into a processor cache is reused as often as possible. Thus, given the proper ordering and access pattern, the performance of matrix multiplication kernel is by the compute power of a system,

The goal of this assignment is to analyze the performance impact of applying various optimizations and discuss the different performance behavior on BEAST. By applying these optimizations, you will be able to achieve decent performance for dense matrix multiplication of BEAST systems. Note that the intention for this assignment is **not** to optimize the matrix multiplication kernel exhaustively. Such primitive kernels are often implemented by vendor performance libraries, and users are advised to leverage those libraries in the code to achieve the best performance in their code.

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 proper access. In this assignment, we walk you through simple analyses and measurements to better understand why this is the case.

```
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

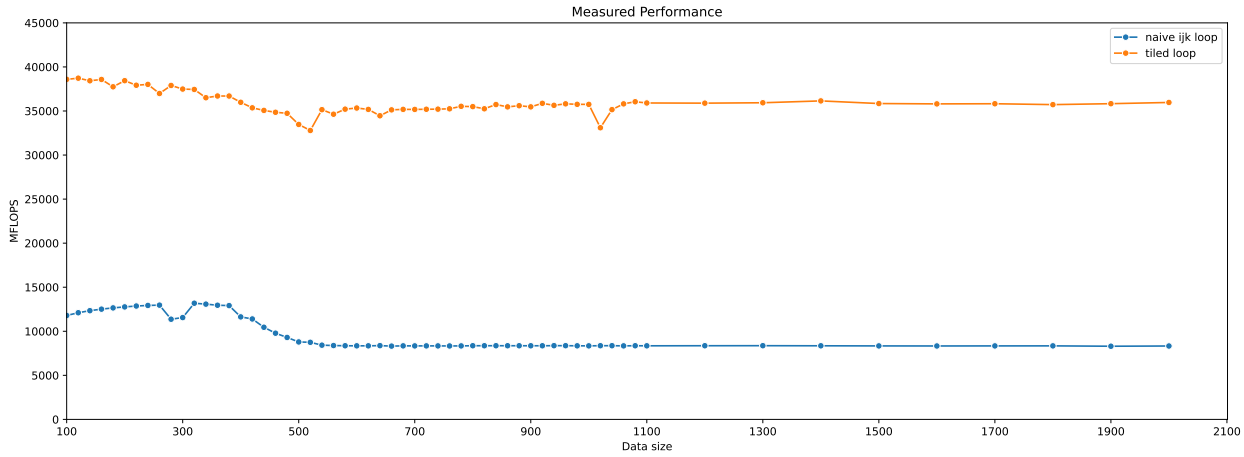

Figure 1: Single-core performance (MFlop/s) of matrix multiplication depending on matrix side length on Icelake system.

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).

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 the possibility of various parallelization schemes, e.g., `#pragma` on different loop levels. Is nested parallelization possible? 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 and understand the code. Try to find the best combination 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. Select compilers from the following list (See Table 1).

System	compiler 1	compiler 2
ice1	icx & icpc (module load compiler)	g++ (module load gcc)
thx2	clang++ (module load llvm)	g++ (module load gcc)
milan2	clang++ (system compiler)	g++ (module load gcc)
cs2	CC	g++ (module load gcc)

Table 1: Compilers to use on different systems.

   Document, analyze the results, and make sure the measurements are stable (i.e., you don't observe significant run-to-run variability). Make sure to report the compiler versions you are using as well. Select the best combination of the flags and compilers and use this combination for the rest of the tasks.

   (b) Inspect the generated assembly code of the inner-most loop to see whether vector instructions are used. You can use `objdump` for that.

(c) As a next step, permute the loop order (i.e., change the execution order of `i`, `j`, and `k`), i.e., `ijk`, `ikj`, `jik`, `jki`, `kij`, and `kji`, and analyze the access pattern to each matrix $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ for each permutation. 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. Now, evaluate these permutations on BEAST systems and discuss whether the results of your evaluations match your expectations from the analysis of access patterns to matrices. Select and use the best loop permutation for the rest of the tasks.

(d) Check out the provided source code with the comment for a hint of the tiling version. It applies loop tiling to the loop nest. The default `TILE_SIZE` parameter can be set to 10. By running this variant, you may get similar performance improvements to what is shown in Fig. 1.

Conduct experiments with different tile sizes (`TILE_SIZE` $\in \{4, 5, 10, 20, 50\}$) on BEAST platforms, and explain your observations. Select and use the best `TILE_SIZE` for the rest of the tasks.

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` $\in \{4, 8, 12, 16, 20, 25\}$).

(e) Parallelize the tiled code using OpenMP. Test your parallel versions on all platforms with different numbers of cores. Run **strong scaling** experiments and create speedup figures, i.e., run experiments with different number of threads while maintaining the size of the matrix constant: core count on X-axis, achieved speedup vs. sequential run on Y-axis – i.e., point (1/1) always start 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? 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[1]) for each platform. 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, try to draw the roofline[2] for each system (as explained in the lecture slides about memory hierarchy) and mark the points corresponding to the achieved performance.

[1] A useful link to lookup instruction throughput and latencies:`https://uops.info/table.html`

[2] You can get some hint about how to plot the roofline here: `https://github.com/RRZE-HPC/likwid/wiki/Tutorial%3A-Empirical-Roofline-Model`

(i) **Performance events:** Show the effect of code optimization(s) (e.g., loop tiling, loop reordering) by comparison of suitable performance metrics obtained from LIKWID.

(j) Search on the Internet for further optimizations and tuning used for matrix multiplication. Such optimizations allow you to reduce the gap between your achieved performance 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 performance ratio achieved with HPL compared to the theoretical peak performance?