High Performance Computing. Assignment 1 Report

Lukas Valatka

March 15, 2018

1 Cache aware programming

50 iterations for averaging metrics were used instead of 5 for more correct (concise) results.

1.1 Explanation

The idea is to fit slice matrices into blocks that would all together fit in processor cache at one instance of time. Simulation was run by splitting matrices into blocks of 200 - maximum common divisor of test data matrix sizes. 200*200*8 = 0.32 MB per matrix, which fits in miniHPC Xeon cache (0.32MB*3 < 25MB).

Results show increase of performance starting from 400. This justifies the block size is correct (fits in cache).

Blocking (or tiling) reduces cache misses, therefore increasing performance. Cache misses are reduced by changing the way matrices are accessed. Algorithm calculates matrix C parts not line by line (naive example) but by by b*b (b - block size) blocks. Such way, A and B matrices are multiplied by b*n slices (A - b rows, n columns, B - n rows, b columns). One major difference is that each line of A does not have to multiplied at one with whole matrix B (instead, with slice of it). It means following matrix A lines get cached versions of current B block (otherwise, would not).

1.2 Performance Plot

200 element block size was chosen. It is the highest common divisor of given matrices sizes. 3 matrices of such size do fit in cache of miniHPC.

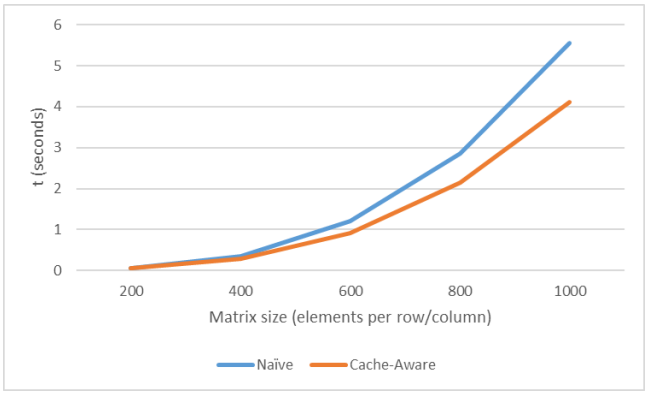

Figure 1: Comparison of matrix multiplication for naive and blocked (tiled) algorithm regarding runtime

1.3 Cache Misses Report

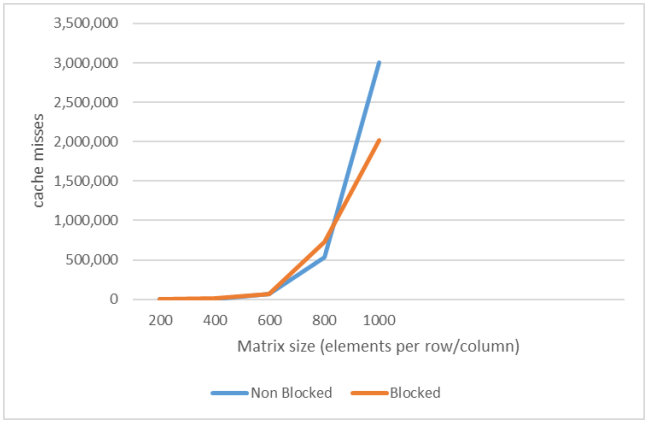

Figure 2: Comparison of matrix multiplication for naive and blocked (tiled) algorithm regarding cache misses

It has to be noted that cache misses were not averaged (as specified by the conditions of the assignment). For this reason, there are fluctuations.

2 Memory access patterns

5 iterations were used (as specified initially) to average results.

2.1 Performance Report

Non-optimized: 0.136634 GFLOPS/s Optimized: 0.338186 GFLOP/s

2.2 Explanation

GFLOP/s value is calculated by dividing matrix size in bytes by full run time. Optimized version runs faster, therefore we receive higher GFLOP/s value.

Loop reorganization increases performance because matrix B is accessed row by row instead of column by column when multiplied with matrix A rows in the inner most loop.

C programs store matrices (arrays of numbers) in a Row-major order, that is, performance is increased (cache misses are decreased) if programs access data row by row.

3 Data organization in the memory

3.1 Performance Plot

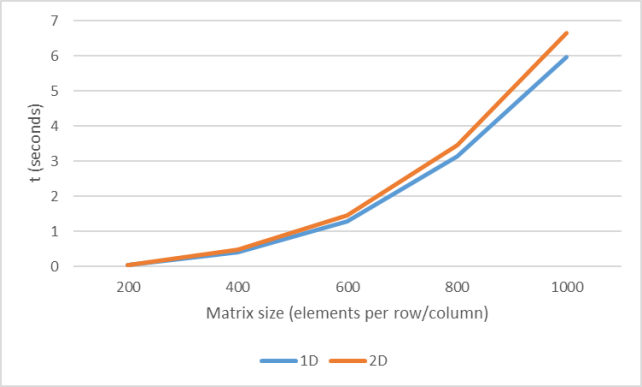

Figure 3: Comparison of runtime for matrix multiplication with C 1D and 2D matrices

4 Auto-vectorization

4.1 Optimization Reports

report-non-vector.optrpt contains non vectorized report. report.optrpt contains vectorized report.

5 Bonus Task

5.1 Performance Plot

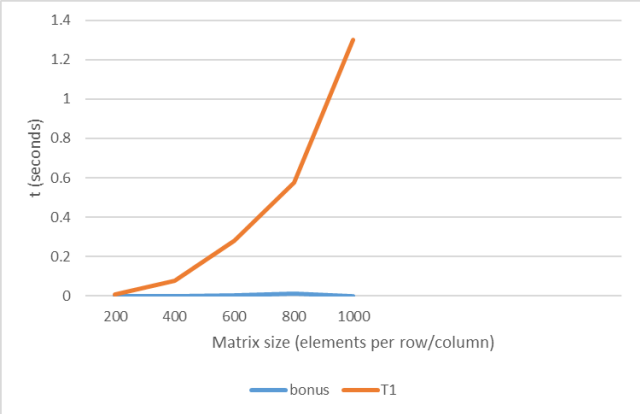

Figure 4: Comparison of runtime for matrix multiplication with all optimizations vs none