Università
della
Svizzera
italiana

**Institute of
Computing
CI**

High-Performance Computing Lab **2022**

Student: Claudio Maggioni Discussed with: –

Solution for Project 1

Due date: 12.10.2022 (midnight)

Contents

1. Explaining Memory Hierarchies *(25 Points)*

1.1. Memory Hierarchy Parameters of the Cluster

By invoking `likwid-topology` for the cache topology and `free -g` for the amount of primary memory, the following memory hierarchy parameters are found:

Main memory	62 GB
L3 cache	25 MB per socket
L2 cache	256 kB per core
L1 cache	32 kB per core

All values are reported using base 2 IEC byte units. The cluster has 2 sockets and a total of 20 cores (10 per socket). The cache topology diagram reported by `likwid-topology -g` is shown in Figure 1.

Socket 0:

0	1	2	3	4	5	6	7	8	9
32 kB	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB
256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB
25 MB									

Socket 1:

10	11	12	13	14	15	16	17	18	19
32 kB	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB
256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB
25 MB									

Figure 1: Cache topology diagram as outputted by `likwid-topology -g`. Byte sizes all in IEC units.

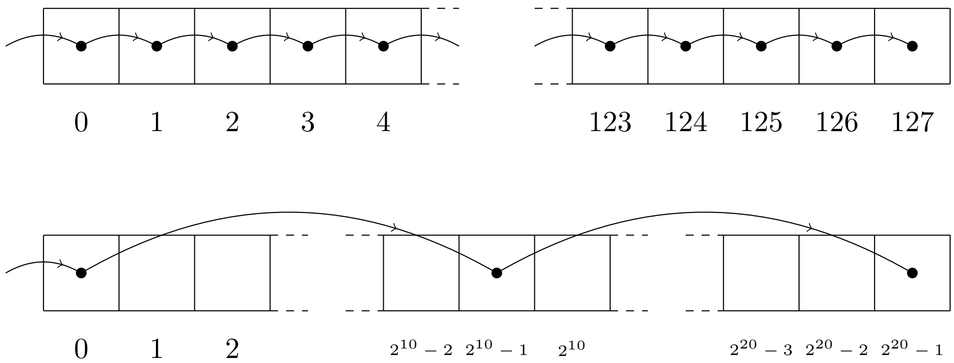


Figure 2: Memory access patterns of `membench.c` for `csize = 128` and `stride = 1` (above) and for `csize =` 2^{20} and `stride =` 2^{10} (below)

1.2. Memory Access Pattern of `membench.c`

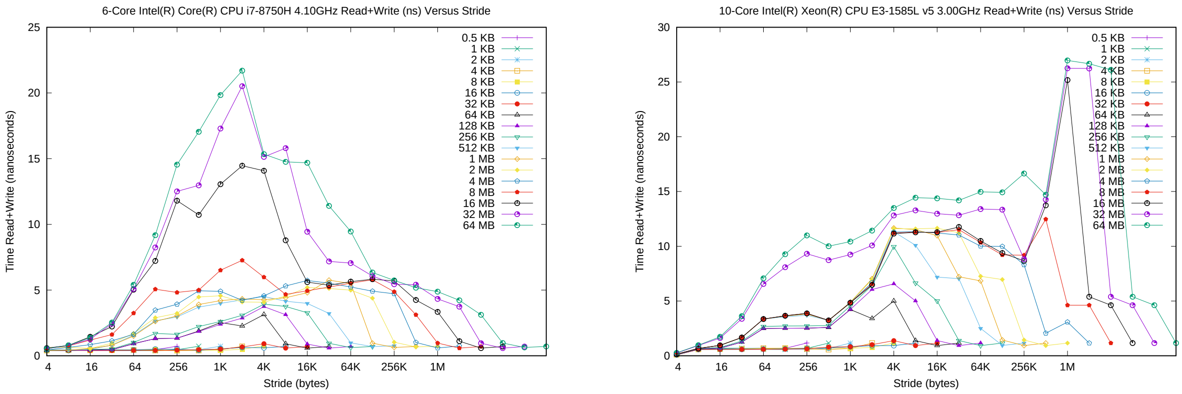
The benchmark `membench.c` measures the average time of repeated read and write overations across a set of indices of a stack-allocated array of 32-bit signed integers. The indices vary according to the access pattern used, which in turn is defined by two variables, `csize` and `stride`. `csize` is an upper bound on the index value, i.e. (one more of) the highest index used to access the array in the pattern. `stride` determines the difference between array indexes over access iterations, i.e. a `stride` of 1 will access every array index, a `stride` of 2 will skip every other index, a `stride` of 4 will access one index then skip 3 and so on and so forth.

Therefore, for `csize = 128` and `stride = 1` the array will access all indexes between 0 and 127 sequentially, and for `csize =` 2^{20} and `stride =` 2^{10} the benchmark will access index 0, then index $2^{10}-1$, and finally index $2^{20}-1$. The access patterns for these two configurations are shown visually in Figure 2.

1.3. Analyzing Benchmark Results

The `membench.c` benchmark results for my personal laptop (Macbook Pro 2018 with a Core i7-8750H CPU) and the cluster are shown in figure 3.

The memory access graph for the cluster's benchmark results shows that temporal locality is best for small array sizes and for small `stride` values. In particular, for array memory sizes of 16MB or lower (`csize` of $4 \cdot 2^{20}$ or lower) and `stride` values of 2048 or lower the mean read+write time is less than 10 nanoseconds. Temporal locality is worst for large sizes and strides, although the



(a) Personal laptop

(b) Cluster

Figure 3: Results of the `membench.c` benchmark for both my personal laptop (in Figure 3a) and the cluster (in Figure 3b).

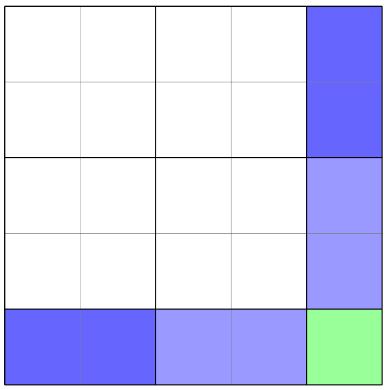


Figure 4: Result of the block division process of a square matrix of size 5 using a block size of 2. The 2-by-1 and 1-by-2 rectangular remainders are shown in blue and the square matrix of remainder size (i.e. 1) is shown in green.

largest values of `stride` for each size (like `csize / 2` and `csize / 4`) achieve better mean times due to the few elements accessed in the pattern (this observation is also valid for the largest strides of each size series shown in the graph).

2. Optimize Square Matrix-Matrix Multiplication *(60 Points)*

The file `matmult/dgemm-blocked.c` contains a C implementation of the blocked matrix multiplication algorithm presented in the project. Other than implementing the pseudocode, my implementation:

- Handles the edge cases related to the "remainders" in the matrix block division, i.e. when the division between the size of the matrix and the block size yields a remainder. Assuming only squared matrices are multiplied through the algorithm (as in the test suite provided) the block division could yield rectangular matrix blocks located in the last rows and columns of each matrix, and the bottom-right corner of the matrix will be contained in a square matrix block of the size of the remainder. The result of this process is shown in Figure 4;
- Converts matrix A into row major format. As shown in Figure 5, by having A in row major format and B in column major format, iterations across matrix block in the inner most loop of

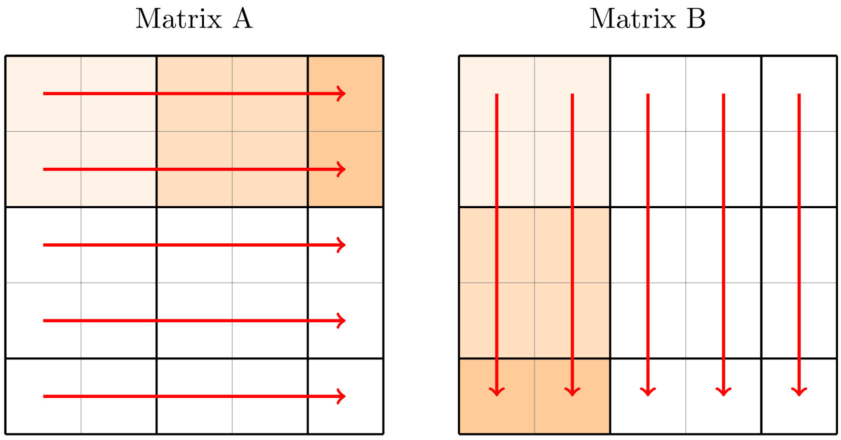


Figure 5: Inner most loop iteration of the blocked GEMM algorithm across matrices A and B. The red lines represent the "majorness" of each matrix (A is converted by the algorithm in row-major form, while B is given and used in column-major form). The shades of orange represent the blocks used in each iteration.

the algorithm (the one calling *naivemm*) cache hits are maximised by achieving space locality between the blocks used;

- Caches the result of each innermost iteration into a temporary matrix of block size before storing it into matrix C. This achieves better space locality when *naivemm* needs to store values in matrix C. The block size temporary matrix has virtually no stride and thus cache hits are maximised. The copy operation is implemented with bulk copy `memcpy` calls.

The results of the matrix multiplication benchmark for the naive, blocked, and BLAS implementations are shown in Figure 6 as a graph of GFlop/s over matrix size or in Figure 7 as a table. The blocked implementation achieves on average 50% more FLOPS than the naive implementation thanks to the optimisations in space and temporal cache locality described. However, the blocked implementation achives less than a tenth of FLOPS compared to Intel MKL BLAS based one due to the microarchitecture optimization the latter one is able to exploit.

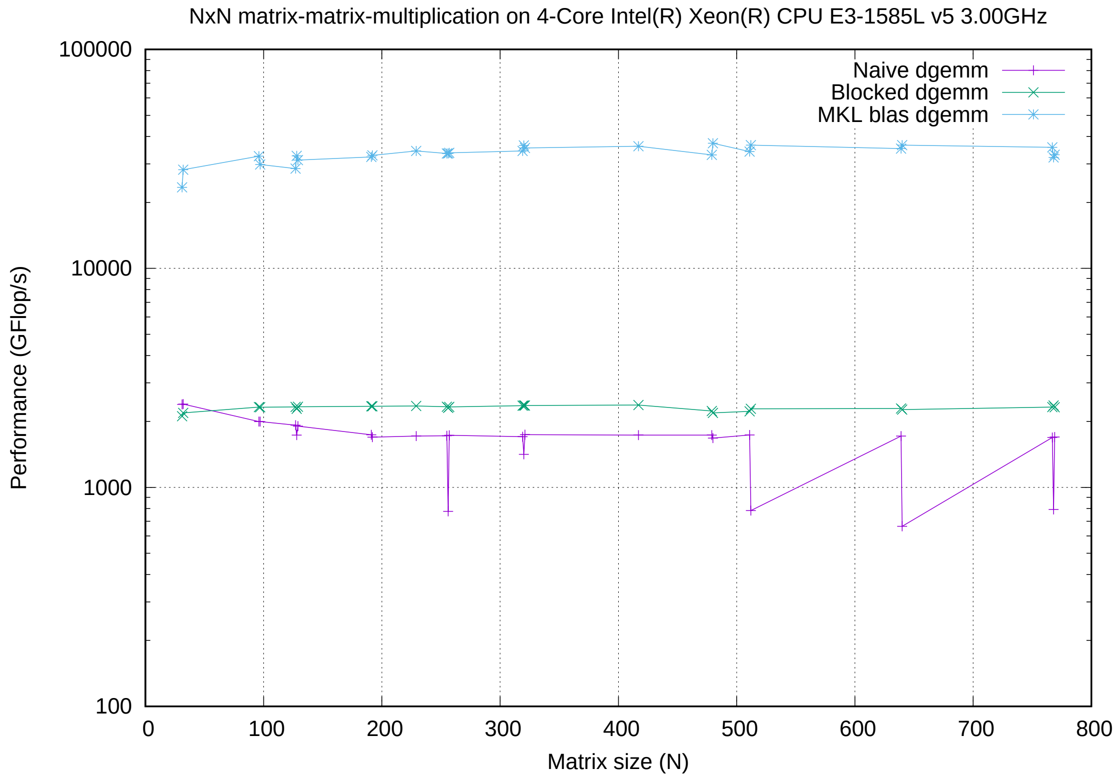


Figure 6: GFlop/s per matrix size of the matrix multiplication benchmark for the naive, blocked, and BLAS implementations. The Y-axis is log-scaled.

	Naive		Blocked		BLAS	
Size	MFLOPS	% CPU	MFLOPS	% CPU	MFLOPS	% CPU
31	2393.33	6.50	2112.63	5.74	23449.20	63.72
32	2400.13	6.52	2187.44	5.94	28198.90	76.63
96	1998.74	5.43	2325.39	6.32	32542.30	88.43
97	1996.01	5.42	2322.81	6.31	29801.30	80.98
127	1923.81	5.23	2330.30	6.33	28557.80	77.60
128	1731.98	4.71	2282.93	6.20	32643.30	88.70
129	1903.31	5.17	2334.25	6.34	31198.20	84.78
191	1736.78	4.72	2345.91	6.37	32247.30	87.63
192	1694.44	4.60	2345.38	6.37	32830.60	89.21
229	1715.10	4.66	2351.01	6.39	34360.90	93.37
255	1720.39	4.67	2335.21	6.35	33477.70	90.97
256	777.65	2.11	2306.48	6.27	33473.90	90.96
257	1729.27	4.70	2330.68	6.33	33686.50	91.54
319	1704.80	4.63	2360.03	6.41	34335.20	93.30
320	1414.84	3.84	2364.53	6.43	36438.10	99.02
321	1741.30	4.73	2366.38	6.43	35433.70	96.29
417	1733.00	4.71	2378.34	6.46	36133.70	98.19
479	1731.17	4.70	2233.05	6.07	32951.40	89.54
480	1678.77	4.56	2187.87	5.95	37260.00	101.25
511	1733.60	4.71	2224.61	6.05	34128.00	92.74
512	782.96	2.13	2284.85	6.21	36526.40	99.26
639	1714.42	4.66	2292.78	6.23	35249.20	95.79
640	663.42	1.80	2264.70	6.15	36538.70	99.29
767	1690.82	4.59	2324.83	6.32	35718.50	97.06
768	792.04	2.15	2363.92	6.42	32116.80	87.27
769	1696.95	4.61	2321.31	6.31	33033.90	89.77

Figure 7: MFlop/s and CPU utlisation per matrix size of the matrix multiplication benchmark for the naive, blocked, and BLAS implementations.