Università della Svizzera italiana | Institute of Computing CI

High-Performance Computing Lab **2022**

Student: Claudio Maggioni Discussed with: M. Cattaneo, G. De Vita, V. Karpenko

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.

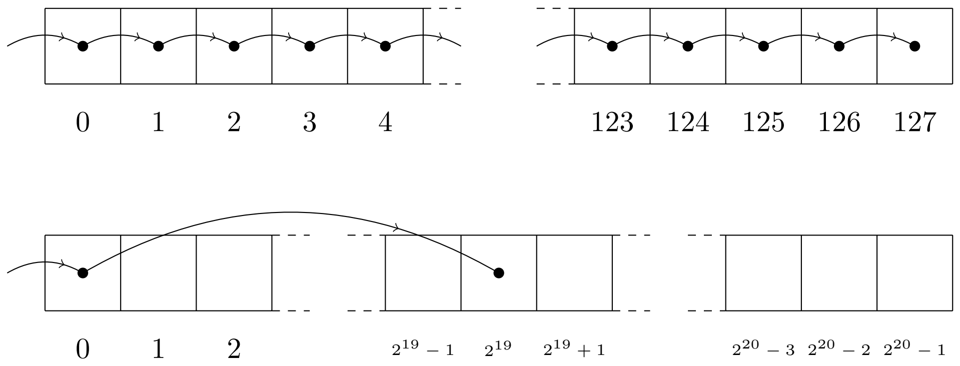


Figure 2: Memory access patterns of `membench.c` for `csize = 128` and `stride = 1` (above) and for `csize =` 2^{20} and `stride =` 2^{19} (below)

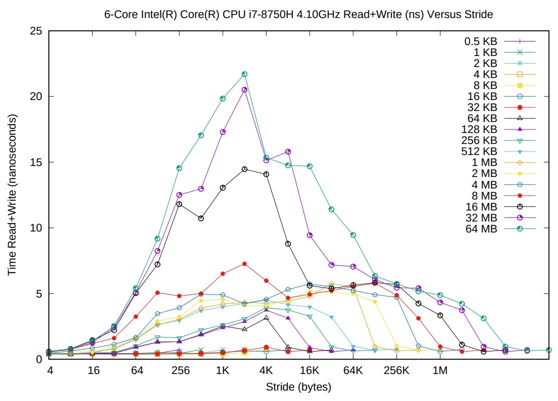
1.2. Memory Access Pattern of `membench.c`

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. The benchmark stops when the index to access is strictly greater than `csize - stride`.

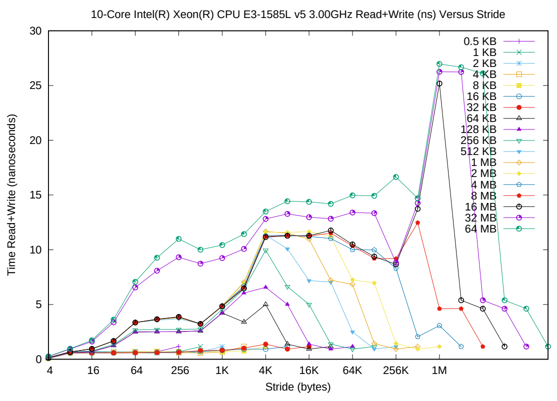
Therefore, for `csize = 128` and `stride = 1` the array will access all indexes between 0 and 127 sequentially, and for `csize =` 2^{20} and `stride =` 2^{19} the benchmark will access index 0, then index $2^{19}-1$. The access patterns for these two configurations are shown visually in Figure 2.

By running the `membench.c` both on my personal laptop and on the cluster, the results shown in Figure 3 are obtained. *csize* values are shown as different data series and labeled by byte size and *stride* values are mapped on the x axis by the byte-equivalent value as well[1]. For `csize = 128` $=$ 512 bytes and `stride = 1` $=$ 4 bytes the mean access time is 0.124 nanoseconds, while for `csize =` 2^{20} $=$ 4MB and for `stride =` 2^{19} $=$ 2MB the mean access time is 1.156 nanoseconds. The first set of parameters performs well thanks to the low *stride* value, thus achiev-

[1]Byte values are a factor of 4 greater than the values used in the code and in Figure 3. This is due to the fact that the array elements used in the benchmark are 32-bit signed integers, which take up 4 bytes each.



(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).

ing very good space locality and maximizing cache hits. However, the second set of parameters achieves good performance as well thanks to the few values accessed with each pass, thus improving the temporal locality of each address accessed. This observation applies for the few last data points in each data series of Figure 3, i.e. for *stride* values close to *csize*.

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 largest values of `stride` for each size (like `csize / 2` and `csize / 4`) achieve better mean times for the aformentioned effect of having *stride* values close to *csize*.

The pattern that can be read from the graphs, especially the one for the cluster, shows that the *stride* axis is divided in regions showing memory access time of similar magnitude. The boundary between the first and the second region is a *stride* value of rougly 2KB, while a *stride* of 512KB roughly separates the second and the third region. The difference in performance between regions and the similarity of performance within regions suggest the threshold stride values are related to changes in the use of the cache hierarchy. In particular, the first region may characterize regions where the L1 cache, the fastest non-register memory available, is predominantly used. Then the second region might overlap with a more intense use of the L2 cache and likewise between the third region and the L3 cache.

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. A pseudocode listing of the implementation is provided in Figure 4.

In order to achieve a correct and fast execution, 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

```
INPUT: A (n by n), B (n by n), n
OUTPUT: C (n by n)

s := 26 # block dimension

A_row := <matrix A converted in row major form>
C_temp := <empty s by s matrix>

for i := 0 to n by s:
    i_next := min(i + s, n)

    for j := 0 to n by s:
        j_next := min(j + s, n)

        <set all cells in C_temp to 0>

        for k := 0 to n by s:
            k_next := min(k + s, n)

            # Perform naive matrix multiplication, incrementing cells of C_temp
            # with each multiplication result
            naivemm(A_row[i, k][i_next, k_next], B[k, j][k_next, j_next],
                    C_temp[0, 0][i_next - i, j_next - j])
        end for

        C[i, j][i_next, j_next] = C_temp[0, 0][i_next - i, j_next - j]
    end for
end for
```

Figure 4: Pseudocode listing of my blocked matrix multiplication implementation. Matrix indices start from 0 (i.e. row 0 and column 0 denotes the top-left-most cell in a matrix). `M[a, b][c, d]` denotes a rectangular region of the matrix M whose top-left-most cell is the cell in M at row a and column b and whose bottom-right-most cell is the cell in M at row $c-1$ and column $d-1$.

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 5;

- Converts matrix A into row major format. As shown in Figure 6, by having A in row major format and B in column major format, iterations across matrix block in the inner most loop of the algorithm (the one calling *naivemm*) cache hits are maximised by achieving space locality between the blocks used. This achieved approximately an increase of performance of two percentage points in terms of CPU utilization (i.e. from a baseline of 4% to 6%),

- 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. This optimization achieves an extra half of a percentage point in terms of CPU utilization (i.e. from the 6% discussed above to roughly 6.5%).

- Exploits some compiler optimizations, namely using the compiler optimizer at the `-O3` setting and using the `-ffast-math` and `-march=haswell` to respectively apply some floating point

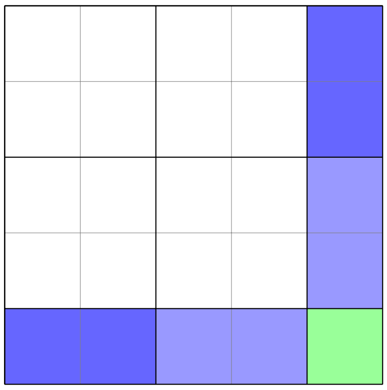


Figure 5: Result of the block division process of a square matrix of size 5 using a block size of 2. The 2-by-1 and 1-by-2 rectangular remainders are shown in blue and the square matrix of remainder size (i.e. 1) is shown in green.

arithmetic optimizations and to set the compiler target to the exact ISA of the cluster's processor. Note that these flags are applied to all implementations used in the benchmark as the flags were added in the `Makefile`. In addition, the `naivemm` algoritms was inlined in the actual `dgemm-blocked.c` source code instead of being separated into a function to be called to achieve better performance and compiler optimizations. All these changes increased the CPU utilization from 6.5% to an average of 13.45%.

The chosen matrix block size for running the benchmark on the cluster is:

$$s = 32$$

as shown in the pseudocode. This value has been obtained by running an empirical binary search on the value using the benchmark as a metric, i.e. by running `./run_matrixmult.sh` several times with different values. For square blocks (i.e. the worst case) the total size for the matrix A and B sub-block and the `C_temp` temporary matrix block for C is:

$$\text{Bytes} = \text{cellSize} * s^2 * 3 = 8 * 32^2 * 3 = 24576$$

given that a double-precision floating point number, the data type used for matrix cells in the scope of this project, is 8 bytes long. The obtained total bytes size is fairly close to the L1 cache size of the processor used in the cluster (32Kb = 32768 bytes), which is expected given that the algorithm needs to exploit fast memory as much as possible. The reason the empirically best value results in a theoretical cache allocation that is only half of the complete L1 cache size is due to some real-life factors. For example, cache misses tipically result in aligned page loads which may load unnecessary data.

A potential way to exploit the different cache levels is to apply the blocked matrix algorithm iteratively multiple times. For example, OpenBLAS implements DGEMM by having two levels of matrix blocks to better exploit the L2 and L3 caches found on most processors.

The results of the matrix multiplication benchmark for the naive, blocked, and BLAS implementations are shown in Figure 7 as a graph of GFlop/s over matrix size or in Figure 8 as a table. The blocked implementation achieves up to 200% more FLOPS than the naive implementation for the largest matrix dimensions. However, the blocked implementation achives roughly an eighth of the FLOPS the Intel MKL BLAS based implementation achieves.

I was unable to run this benchmark suite on my personal machine due to Intel MKL installation issues that prevented the code to compile.

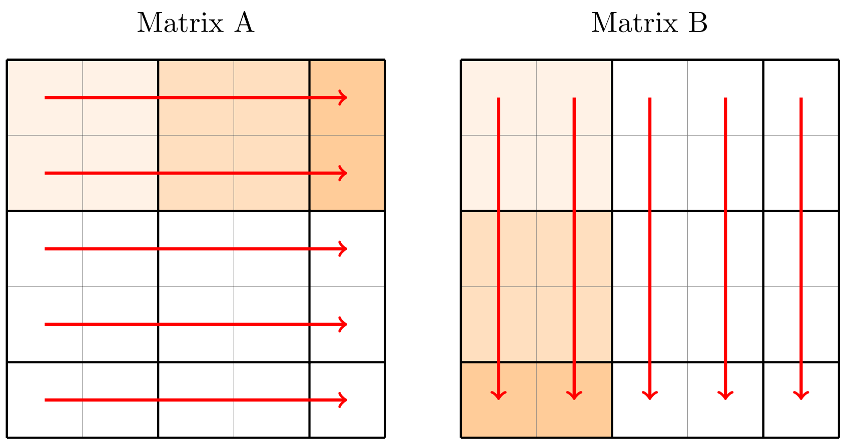


Figure 6: Inner most loop iteration of the blocked GEMM algorithm across matrices A and B. The red lines represent the "majorness" of each matrix (A is converted by the algorithm in row-major form, while B is given and used in column-major form). The shades of orange represent the blocks used in each iteration.

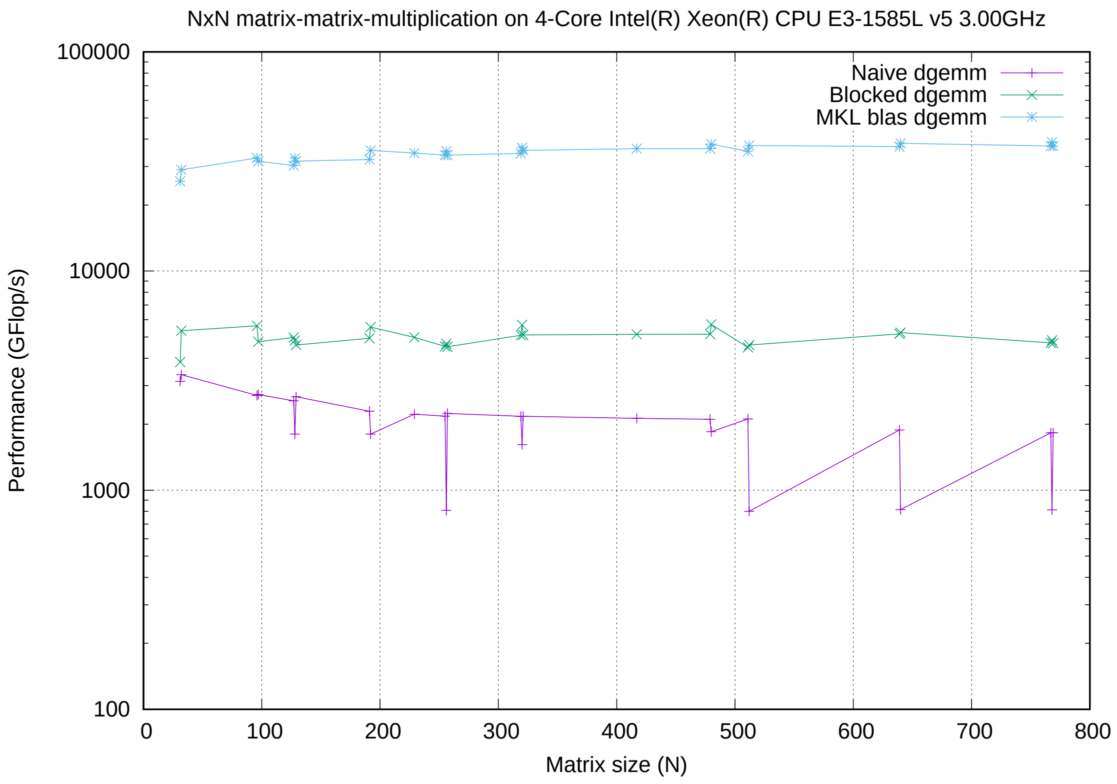


Figure 7: GFlop/s per matrix size of the matrix multiplication benchmark for the naive, blocked, and BLAS implementations. The Y-axis is log-scaled.

	Naive		Blocked		BLAS	
Size	MFLOPS	% CPU	MFLOPS	% CPU	MFLOPS	% CPU
31	3140.45	8.53	3844.56	10.45	25677.4	69.78
32	3364.78	9.14	5342.55	14.52	28952.1	78.67
96	2703.08	7.35	5620.08	15.27	32816.4	89.18
97	2729.68	7.42	4754.1	12.92	31699.2	86.14
127	2556.58	6.95	4977.82	13.53	30274.5	82.27
128	1803.41	4.90	4817.8	13.09	32721.7	88.92
129	2669.26	7.25	4594.25	12.48	31746.4	86.27
191	2290.09	6.22	4931.27	13.40	32263.1	87.67
192	1801.66	4.90	5549.67	15.08	35491.2	96.44
229	2218.61	6.03	4982.59	13.54	34557.2	93.91
255	2178.15	5.92	4528.43	12.31	33771.3	91.77
256	808.413	2.20	4652.68	12.64	35221.1	95.71
257	2238.93	6.08	4512.33	12.26	33807.9	91.87
319	2174.45	5.91	5093.38	13.84	34415.8	93.52
320	1612.13	4.38	5674.61	15.42	36500.2	99.19
321	2173.64	5.91	5111.09	13.89	35508.1	96.49
417	2125.36	5.78	5143.98	13.98	36157.6	98.25
479	2107.13	5.73	5152.51	14.00	36186.4	98.33
480	1848.43	5.02	5703	15.50	37971.3	103.18
511	2112.99	5.74	4479.96	12.17	35144	95.50
512	801.127	2.18	4596.26	12.49	37362.5	101.53
639	1881.94	5.11	5168.59	14.05	36989.1	100.51
640	815.847	2.22	5232.97	14.22	38267.8	103.99
767	1825.75	4.96	4701.09	12.77	37220.8	101.14
768	812.933	2.21	4826.12	13.11	38744	105.28
769	1825.38	4.96	4686.21	12.73	37076.1	100.75

Figure 8: MFlop/s and CPU utlisation per matrix size of the matrix multiplication benchmark for the naive, blocked, and BLAS implementations.