Università della Svizzera italiana | Institute of Computing CI

High-Performance Computing Lab **2022**

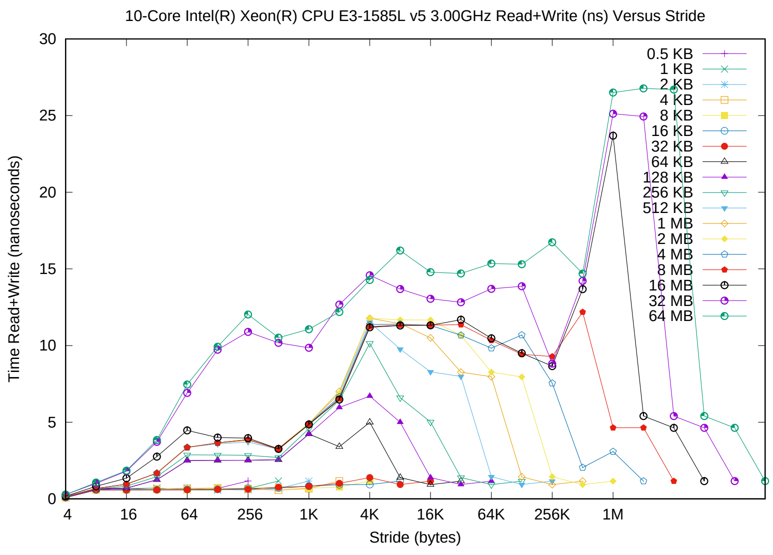
Student: SIMONE TARENZI Discussed with: E. BARDELLI, L. ZANIOL, V. NAIK

Solution for Project 1

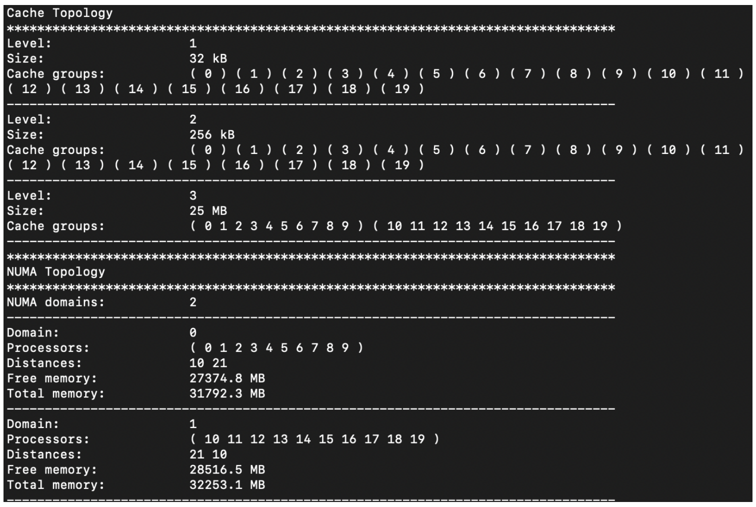
Due date: 12.10.2022 (midnight)

1. Explaining Memory Hierarchies *(25 Points)*

1. From the graph obtained by running the program on the compute node of the cluster, there seems to be pretty clear "steps" where the time to read and write data quickly rises.



Looking at the various lines, it seems that the 64 KB one is the first which starts to deviate from the previous ones, so my guess is that 32 KB is the size of the L1 cache of the compute node processor.
After that, results are a bit harder to interpret: from the 64 KB to the 4 MB line, they all follow pretty much the same trajectory. My guess would have been 128 KB for the L2 cache size and 8 MB for L3 cache, but that wasn't actually the case.
Using the command "likwid-topology" we can take a look at the actual sizes:



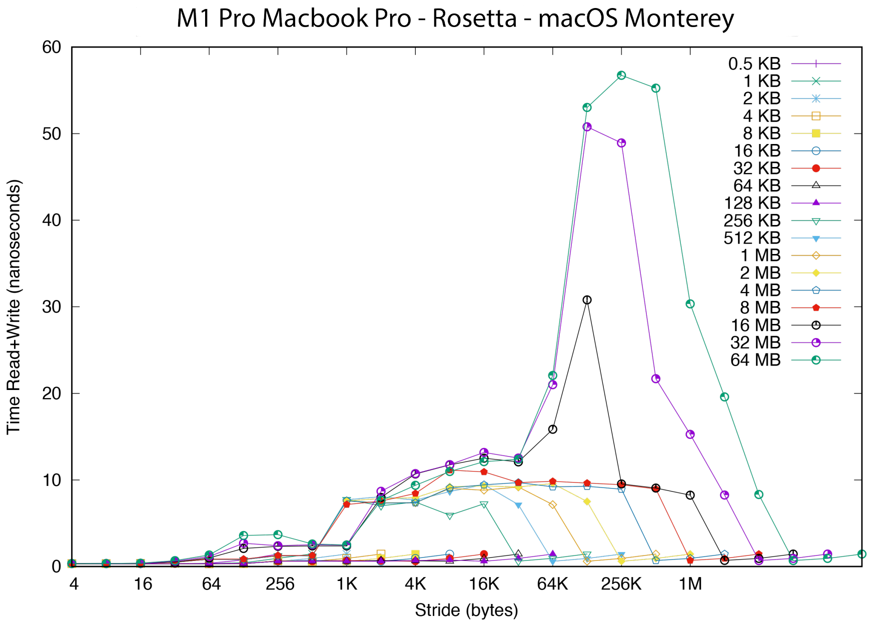
```
Cache Topology
*******************************************************************************
Level:                  1
Size:                   32 kB
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 4 ) ( 5 ) ( 6 ) ( 7 ) ( 8 ) ( 9 ) ( 10 ) ( 11 )
( 12 ) ( 13 ) ( 14 ) ( 15 ) ( 16 ) ( 17 ) ( 18 ) ( 19 )
-------------------------------------------------------------------------------
Level:                  2
Size:                   256 kB
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 4 ) ( 5 ) ( 6 ) ( 7 ) ( 8 ) ( 9 ) ( 10 ) ( 11 )
( 12 ) ( 13 ) ( 14 ) ( 15 ) ( 16 ) ( 17 ) ( 18 ) ( 19 )
-------------------------------------------------------------------------------
Level:                  3
Size:                   25 MB
Cache groups:           ( 0 1 2 3 4 5 6 7 8 9 ) ( 10 11 12 13 14 15 16 17 18 19 )
-------------------------------------------------------------------------------
*******************************************************************************
NUMA Topology
*******************************************************************************
NUMA domains:           2
-------------------------------------------------------------------------------
Domain:                 0
Processors:             ( 0 1 2 3 4 5 6 7 8 9 )
Distances:              10 21
Free memory:            27374.8 MB
Total memory:           31792.3 MB
-------------------------------------------------------------------------------
Domain:                 1
Processors:             ( 10 11 12 13 14 15 16 17 18 19 )
Distances:              21 10
Free memory:            28516.5 MB
Total memory:           32253.1 MB
-------------------------------------------------------------------------------
```

L1 cache: 32 KB
L2 cache: 256 KB
L3 cache: 25 MB
Main Memory: 64 GB

3. In both cases at csize = 128 KB with stride = 8 KB and at csize = 1 MB with stride = 512 KB we are accessing the L1 cache in linear time.

4. We will have good temporal locality when we are accessing data which is stored in the same cache layer: for example all array sizes up until 32 KB share the same temporal locality, since they can be stored in L1 cache. Array sizes such as 32 and 64 MB instead only show ok temporal locality from 4 to 256 KB stride, because they are so big that they will have trouble getting accessed in the same cache layer.
Array sizes from 512 KB to 8 MB show instead two clear plateaus where I'm guessing they switch from L2 to L3 cache layer.

Finally let's take a look at the performance graph of an M1 Pro Macbook Pro.



Having 2.9 MB of L1, 28 MB of L2 and 24 MB of L3 cache helps getting good temporal locality at array sizes up to 16 MB at all stride values.

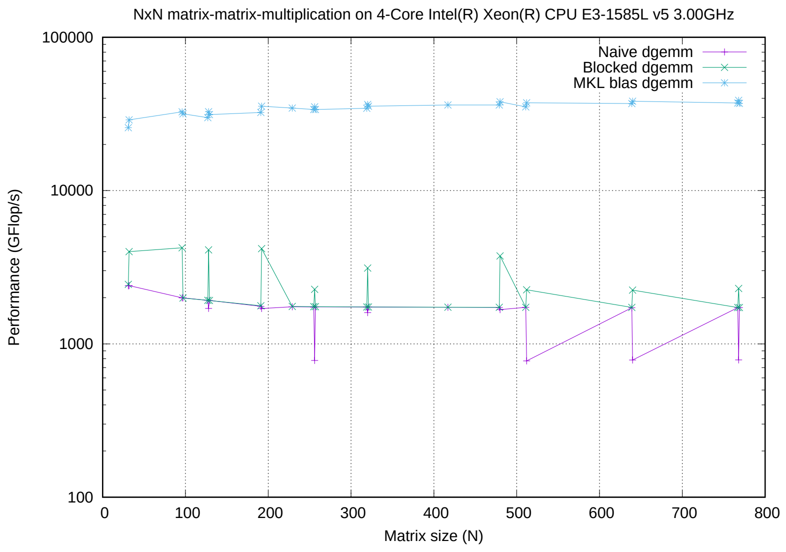
2. Optimize Square Matrix-Matrix Multiplication *(60 Points)*

The optimization had the objective of implementing a way to multiply matrices using submatrices of NxN cells at a time: this enables saving more of those numbers in localized, faster to access memory, and then, after the calculations are done, saving them back into the resulting matrix. This would result in less accesses to the slow memory where the matrices are stored and, therefore, improve performance.
The idea was to implement a way to do that for all matrices, no matter their dimensions: all calculations would be done in blocks of 2x2 and then, once close to the boundaries of the matrix, go back to the previous naive implementation. Sadly I wasn't able to do that, so instead I've implemented a way to check if the matrix dimensions are divisible by 2 or 3, and then use blocks of 2 or 3, respectively.
This, however, gives no speed benefit to calculations done on matrices whose dimensions aren't divisible by 2 or 3, like all prime numbers for example, but since the program is multiplying matrices of various sizes, there will still be an overall increase in speed.

```
void square_dgemm (int n, double* A, double* B, double* C){
   int i,j,k;
   if (the matrix is divisible in blocks of 3x3){
      for (i=0; i<n3; i+=3){
         for (j=0; j<n3; j+=3){
            //save each cell of this block in local variables
            double c0 = C(i,j);
            double c1 = C(i,j+1);
            double c2 = C(i,j+2);
            double c3 = C(i+1,j);
            //etc until c8
            for (k = 0; k < n; k++){
               //save the results in the local variables
               c0 += A(i,k) * B(k,j);
               c1 += A(i,k) * B(k,j+1);
               c2 += A(i,k) * B(k,j+2);
               c3 += A(i+1,k) * B(k,j);
               //etc
            }
            //save the final calculations back in to the matrix
            C(i,j) = c0;
            C(i,j+1) = c1;
            C(i,j+2) = c2;
            C(i+1,j) = c3;
            //etc
         }
      }
   }
   else if (the matrix is divisible in blocks of 2x2){
      //same thing but there will be 4 local variables
   }
   else if (the matrix is not divisible by 3 or 2){
      //use the naive implementation
   }
}
```

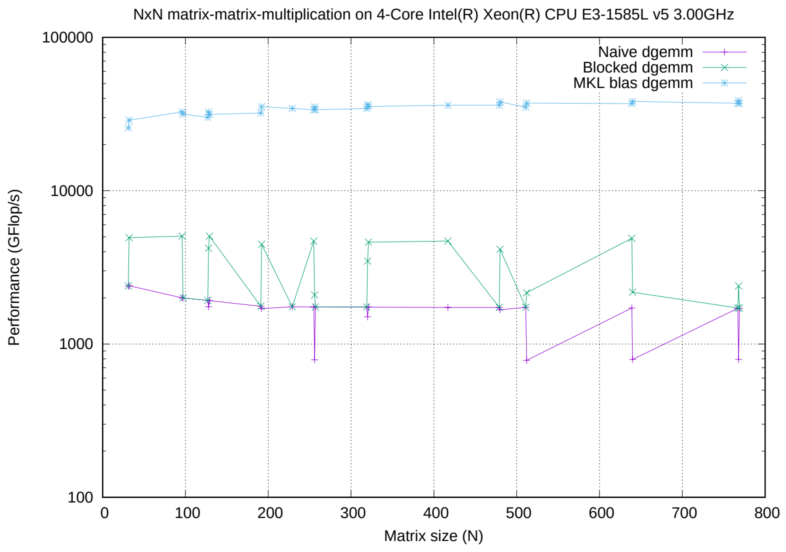
Some matrices will obviously be divisible by both 2 and 3, this is why the code checks with 3 first, so that in those cases it can always use the faster implementation.

First of all, let's take a look at the performance of the implementation of 2x2 blocks alone.



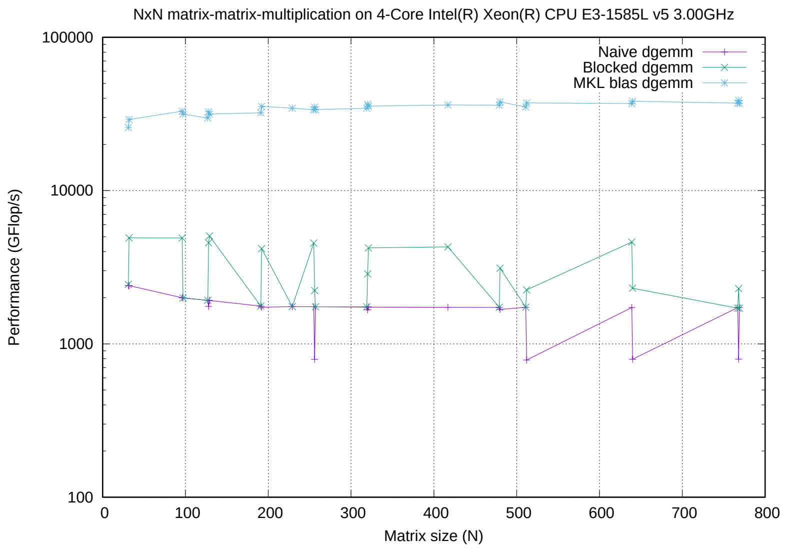
Only even matrices get an improvement in performance, but those that do are 2 to 2.5 times as fast as the ones that use the naive implementation. For example, for matrix size 128, the peak speed went from 1713.7 Mflop/s to 4218.25 Mflop/s.

Now let's look at the final implementation with both 3x3 and 2x2 blocks.

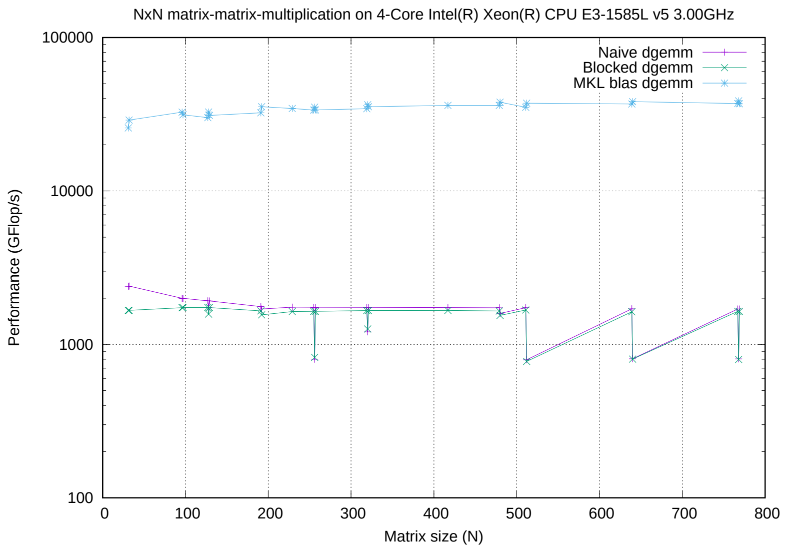


More matrices now get an improvement in performance, and the ones divisible by blocks of 3x3 now get 2.5 times the performance of the naive implementation as a basis: matrix size 129 went from 1919.15 Mflop/s to 5056.54 Mflop/s.
In the end the performance of the whole program went from a 4.5 peak percentage to an 8 percent peak.

I was actually thinking of adding an implementation that used 4x4 blocks too but, surprisingly, it performed slightly worse than the implementation with both 2x2 and 3x3 blocks.

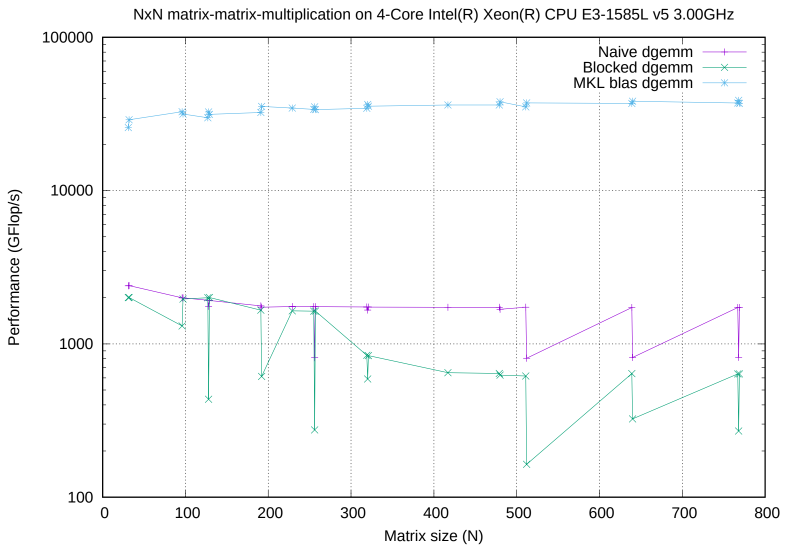


To see the actual impact that local variables can have, I tried to see what would happen after removing the one in the naive implementation.



It's not much when it's just a single one, but we've already seen how much of a difference they can make when there's multiple accesses to the same ones over and over again.

And, finally, one last test where the order of the j and k indeces gets flipped.



3. Quality of the Report *(15 Points)*