Università della Svizzera italiana

Institute of Computational Science ICS

High Performance Computing **2019**

Student: Gabriel Fernandes de Oliveira Discussed with: N/A

Solution for Assignment 2

Due date: 20 October 2019, 13:30

In this exercise you will practice in data access optimization and performance-oriented programming for cluster environments.

1. Explaning memory hierarchies *(30 Points)*

1. The following data was found using the likwid module and the meminfo file, as suggested by the exercise.

Main memory	65.69 GB
L3 cache	20 MB
L2 cache	256 kB
L1 cache	32 kB

2. Both graphs were created and added to this assignment:

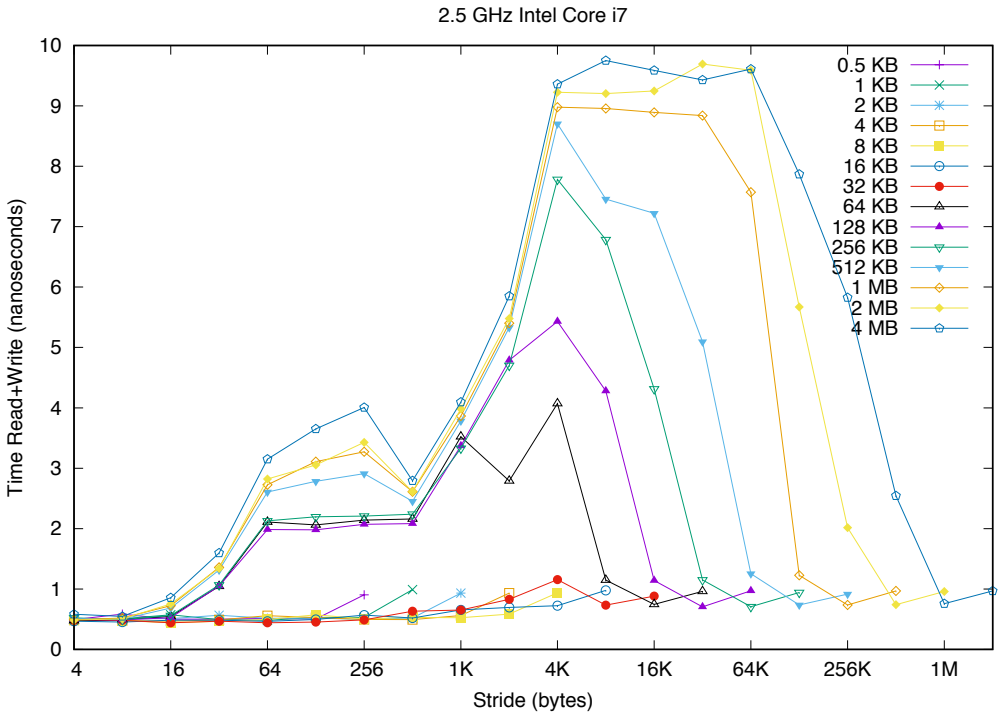

Figure 1. This is the graph generated for my Macbook Pro running on macOS Sierra 10.12.6

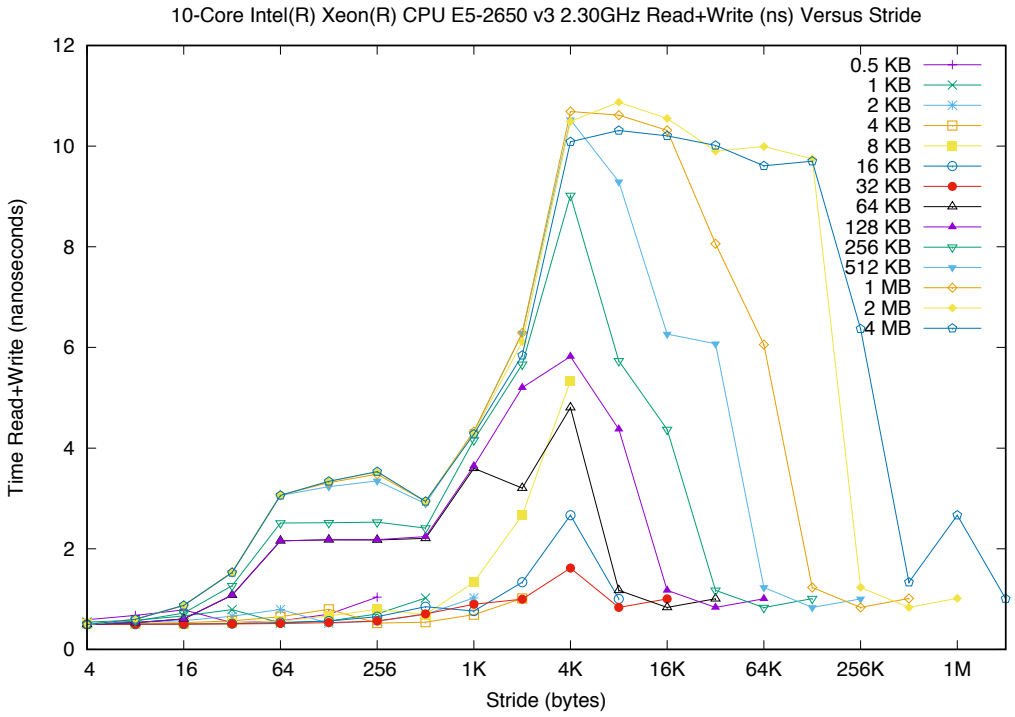

Figure 2. Graph generated on icsmaster

The ps files were also added to this assignment folder, they are called respectively generic_mac.ps and generic_icsmaster.ps.

On both graphs the difference between cache response times is very visible. The arrays with size $\leq 32kB$ have a very good performance (operations taking around 0.5ns) for every value of stride, since they can be completely stored in L1 cache. The arrays with size $\leq 256kB$ show a regular plateau between stride of 64 and 512 bytes where read and write takes about 2ns, this is mainly due to the fact that these arrays can be stored in L2 cache, providing a access to the array elements that is slower than that of L1, but is still efficient. Finally, another plateau can be noticed for array sizes between $1MB$ and $4MB$, for stride between $4kB$ and $128kB$, this is due to the cache L3, which has a retrieval time worse than that of caches L1 and L2, but still way faster than that of the main memory.

3.
   - $csize = 128 = 0.5kB$ and $stride = 1 = 4B$ This example has a very good performance, around 0.5ns for both icsmaster and my computer, that is mainly due to the good spatial

locality associated with a small value of stride. Because the stride is 1, every element loaded in a cache line will be used, hence the good spatial locality and performance of this case.

- $csize = 2^{20} = 4MB$ and $stride = csize/2 = 2^{19} = 2MB$ This is another extreme case with a good performance, the read+write takes about 1ns for both computers, that is due to the temporal locality associated with traversing the same elements already loaded on cache, reusing their cache copies.

  Since the stride is half the size of the array only two elements of the array are being accessed every time the array is traversed. On the first array traversal these two elements are loaded into the cache, and on all following traversals they are retrieved directly from cache, hence the good temporal locality and performance in this case.

4. The points with best temporal locality in the graph are the ones closes to the end of every line. For example, for the array of size $4MB$, the stride values $2MB$ and $1MB$ offer a very good temporal locality. This happens because with a large stride value, few elements of the array are accessed. These few elements accessed can all fit into the cache, and are reused many times since the array is being traversed multiple times, hence the good temporal locality and the high performance on these conditions.

   One can notice similar high performances on other array sizes, when the stride value is close to half or a quarter of the array size.

   Another example of good temporal locality happens for small array sizes ($\leq 32KB$), since these arrays can be fully stored on the L1 cache. The first time that the array is traversed every element is loaded into the cache, then for every following traversal the whole array is on cache, so the cache elements are simply reused.

2. Optimize Square Matrix-Matrix Multiplication *(70 Points)*

1. My implementation of the block matrix multiplication (figure 4) was based on the pseudo-code provided in the motivation file (figure 3). Analysing the code snippets one can notice the similarities:

```
Blocked matrix multiplication C = C + A · B

method BlockedMM(A, B, C)

for i=1 to n/s
   for j=1 to n/s
      Load C_{i,j} into fast memory
      for k=1 to n/s
         Load A_{i,k} into fast memory
         Load B_{k,j} into fast memory
         NaiveMM (A_{i,k}, B_{k,j},  C_{i,j}) using only fast memory
      end for
      Store C_{i,j} into slow memory
   end for
end for
```

Figure 3. Pseudo-code provided on the motivation file

```
void square_dgemm (const double  *A, const double  *B,  double  *C, const unsigned  M){
    double Ab[BLOCK_SIZE * BLOCK_SIZE], Bb[BLOCK_SIZE * BLOCK_SIZE], Cb[BLOCK_SIZE * BLOCK_SIZE];
    for (int i=0;i<M; i+=BLOCK_SIZE) {
        for (int j=0;j<M; j+=BLOCK_SIZE) {
            slow_read(Cb, C, i, j, M);
            for(int k=0;k<M;k+=BLOCK_SIZE){
                slow_read(Ab, A, i, k, M);
                slow_read(Bb, B, k, j, M);
                naive_mm(Cb, Ab, Bb);
            }
            slow_write(C, Cb, i, j, M);
        }
    }
}
```

Figure 4. Function *square_dgemm* written

The *slow_read* and *slow_write* functions can be seen in more detail in figures 5 and 6.

The *slow_read* function copies a submatrix of $BLOCK\_SIZE$ lines and columns from a matrix located in slow memory, arrays A, B, C, to the local arrays created, Ab, Bb and Cb.

In case the submatrix being copied is not entirely contained by the original matrix, the copied matrix will be padded with zeroes, until it reaches the size $BLOCK\_SIZE$.

This is useful because it guarantees that the copied matrix has always the same dimensions: $BLOCK\_SIZE \times BLOCK\_SIZE$.

```
void slow_read(double* to, const double *from, int i, int j, const unsigned M){
    int cnt = 0;
    for(int ii = i;ii < i + BLOCK_SIZE;++ii){
        for(int jj = j;jj < j + BLOCK_SIZE;++jj){
            if(ii >= M || jj >= M)
                to[cnt++] = 0;
            else
                to[cnt++] = from[ii * M + jj];
        }
    }
}
```

Figure 5. Load function

```
void slow_write(double *C, double *Cb, int i, int j, const unsigned M){
    for(int ii=0;ii < BLOCK_SIZE && i + ii < M;++ii){
        for(int jj=0;jj < BLOCK_SIZE && j + jj < M;++jj){
            C[(i+ii) * M + j + jj] += Cb[ii*BLOCK_SIZE + jj];
        }
    }
}
```

Figure 6. Store function

The naive matrix multiplication only receives references to matrices on fast memory with dimensions $BLOCK\_SIZE \times BLOCK\_SIZE$ and does the classical algorithm for matrix multiplication.

```
void naive_mm(double *C, double *A, double *B){
    for(int i=0;i<BLOCK_SIZE;++i){
        for(int j=0;j<BLOCK_SIZE;++j){
            double ans = 0;
            for(int k=0;k<BLOCK_SIZE;++k)
                ans += A[i*BLOCK_SIZE + k] * B[k*BLOCK_SIZE + j];
            C[i * BLOCK_SIZE + j] += ans;
        }
    }
}
```

Figure 7. Naive matrix multiplication

The best theoretical value for the block size is $\sqrt{M/3}$, being M the size of the cache.

Using only the cache L1 $M = 32kB$ (for icsmaster), that means that the optimal block size is $\sqrt{32kB/3} = 103B$ since every *double* in C has $8B$, $103B \approx 37$ elements. Using the cache L2, with $M = 256kB$, the theoretical optimal block size is around 103 elements.

In practice, though, one can notice that the block size that provides the best performance is 16, as seen in the next question.

2. As previously stated the best performance was achieved by using a block size of 16.

   Follows the graph plotted on my computer (Macbook Pro running macOS Sierra 10.12.6 with 2.5GHz Intel Core i7):

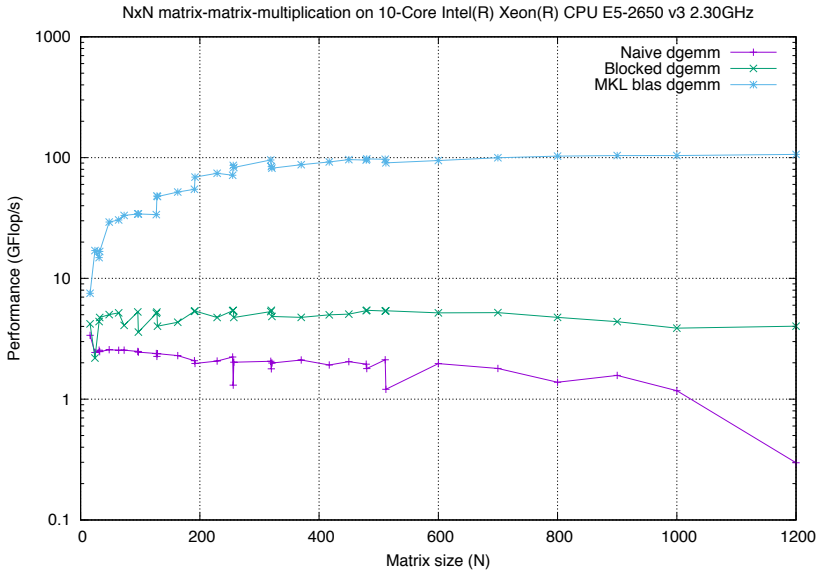

Figure 8. Graph for the experiment on Macbook Pro on macOS Sierra 10.12.6 on 2.5GHz Intel Core i7

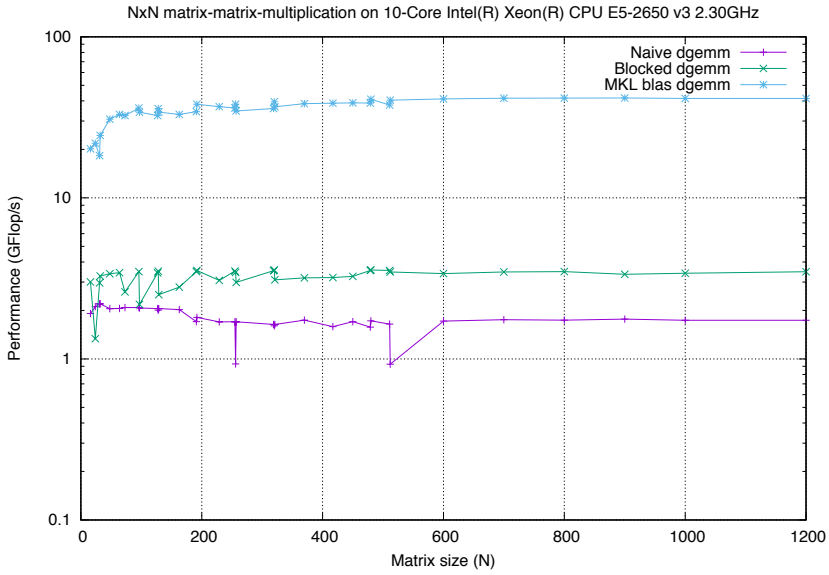

Figure 9. Graph for the experiment on icsmaster

3. Analysing the Naive dgemm line one can notice two unusual drops in performance, these happen in exact powers of two: 256 and 512.

   In order to access this effect I computed the following graphs, by creating more testcases with sizes around 256 (figure 10) and 512 (figure 12):

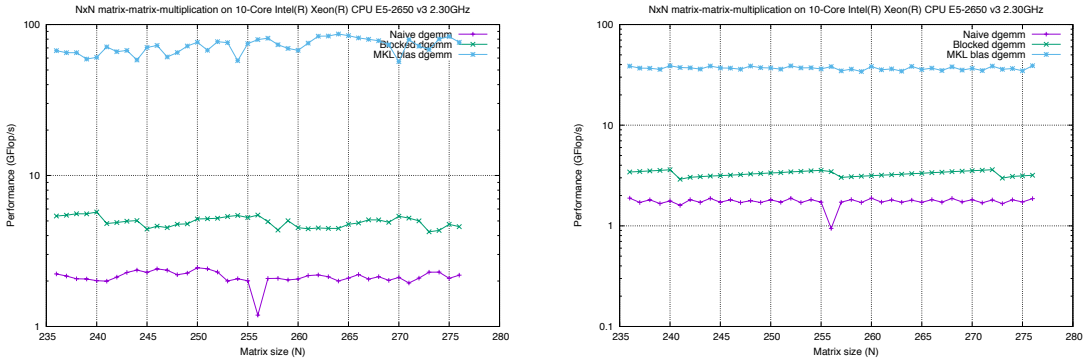

Figure 10. Graphs centered on 256 on Macbook and icsmaster, respectively

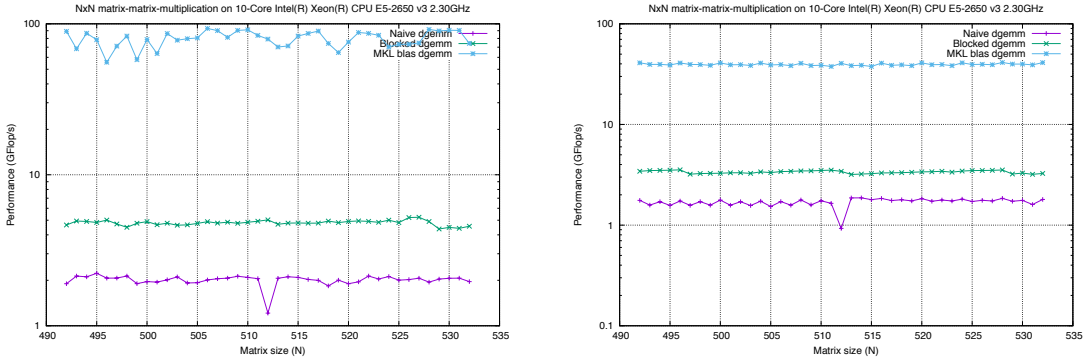

Figure 11. Graphs centered on 512 on Macbook and icsmaster, respectively

The explanation for that is the great number of rewrites happening on cache.

Since both matrix size and cache number of lines are powers of two, many data will be stored in the same cache lines. This ends up causing a lot of rewrites and cache misses, consequently very few elements will be used from the cache, most of the memory accesses will have to be done on slow memory.

We can also notice that this doesn't seem to affect the other algorithms (Blocked and MKL), since both make a better use of the machine's cache, maximizing the reusage of the elements loaded in cache and thus achieving a better performance.

The second thing one can notice in the graph is the poor performance that the blocked algorithm developed has for small matrix sizes:

That is due to the chosen block size being very close to the matrix size.

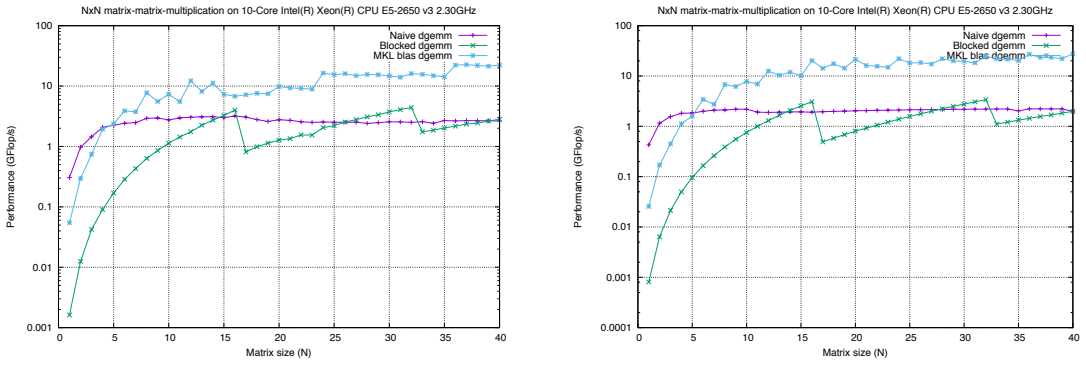

Figure 12. Graphs run with matrix size from 1 to 40 on Macbook and icsmaster, respectively

The huge drop at performance at the beginning of the graph happens because the matrices sizes are smaller than the block size (16). My algorithm simply copies the small matrix to a 16x16 matrix padded with zeroes, making much more operations than needed.

One can also notice that the peaks in the blocked algorithm performance happens in multiples of the block size: 16 and 32. Also, the worst performances are associated with the values right after these multiples: 17 and 33. This can also be explained by the padding of zeroes: For instance, a matrix with size 32 will be divided in four blocks of size 16, whereas a matrix with size 33 will be divided into 9 blocks, 5 of which will be mostly consisted by zeroes.

As the matrix size gets closer to a multiple of 16, the amount os positions padded with zero decreases, consequently increasing the performance of the multiplication.

4. There were two main differences in the performance between my machine and icsmaster.

   The first big difference is on the performance of "MKL blas dgemm". My machine reaches a plateau of about 100 GFlops/s, when the matrix size (N) is greater or equal to 300, whereas icsmaster reaches a plateau of about 30 GFLlops/s, when $N \geq 50$, as can be seen on figures 8 and 9.

   This difference can be explained by analysing the difference in processors that my machine and icsmaster have. Making a comparison in intel's website between my processor (Core i7-4870HQ) and icsmaster's processor (Xeon E5-2650) (the comparison can be seen in ark.intel.com/content/www/us/en/ark/compare.html?productIds=83504,81705), one can see that my processor has better specifications, as shown in the following table:

	Macbook	**icsmaster**
Processor base frequency	2.50 GHz	2.30 GHz
Max Turbo frequency	3.70 GHz	3.00 GHz

   Intel's Math Kernel Library makes good use of this extra power present in my machine, running way more efficiently on my machine than on icsmaster.

   The second big difference can be seen on the performance of the naive algorithm for $N \geq 700$. When $N = 1200$, my machine has a performance of 0.2 GFlops/s while icsmaster operates at about 1.1 GFlops/s, this is visible on figures 8 and 9.

   This is due to the difference of cache L3 sizes between icsmaster and my machine. While icsmaster can hold up to 20MB in L3, my machine can only hold 6.3MB (information retrived by running *sysctl hw.l3cachesize* on the terminal).

   Three matrices of $N = 700$ occupy at least N^2 doubles, which can be translated to $3 \times (700^2 \times 8B) \approx 12$MB. This means that icsmaster can still hold these matrices on cache, but my machine has to fetch them from memory, explaining why my machine has a drop in performance for $N \geq 700$ whereas icsmaster can mantain a constant performance at about 1.1 GFlops/s.

5. Using Intel's MKL can be hard, since many functions that are commonly used in C can be replaced for more optimized functions. For instance, instead of using *malloc* for memory allocation, one can use *mkl_ malloc* that allows one to set the alignment of the buffer.

   Nevertheless, although MKL has a steep learning curve, there are tutorials available on Intel's website and the performance gained through the usage of such library is worth it, as this experiment proves it.