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: 15 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:

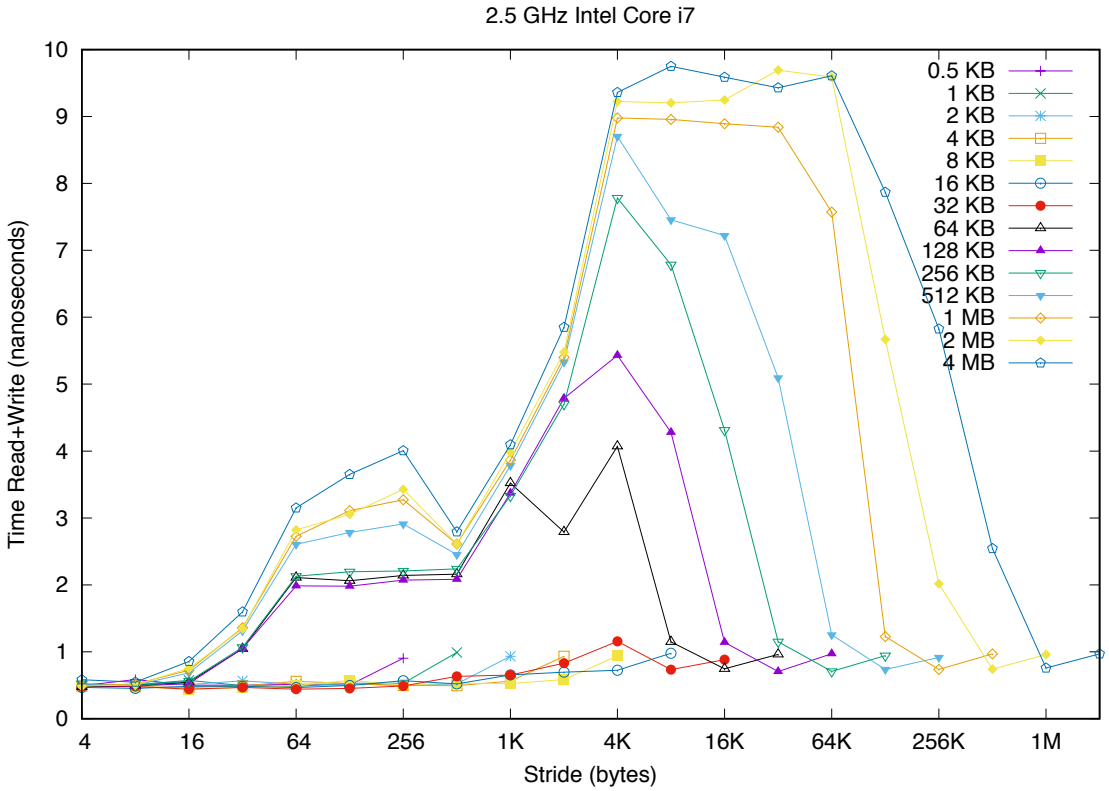

Figure 1. This is the graph generated for my Macbook Pro running on macOS Sierra 10.12.6

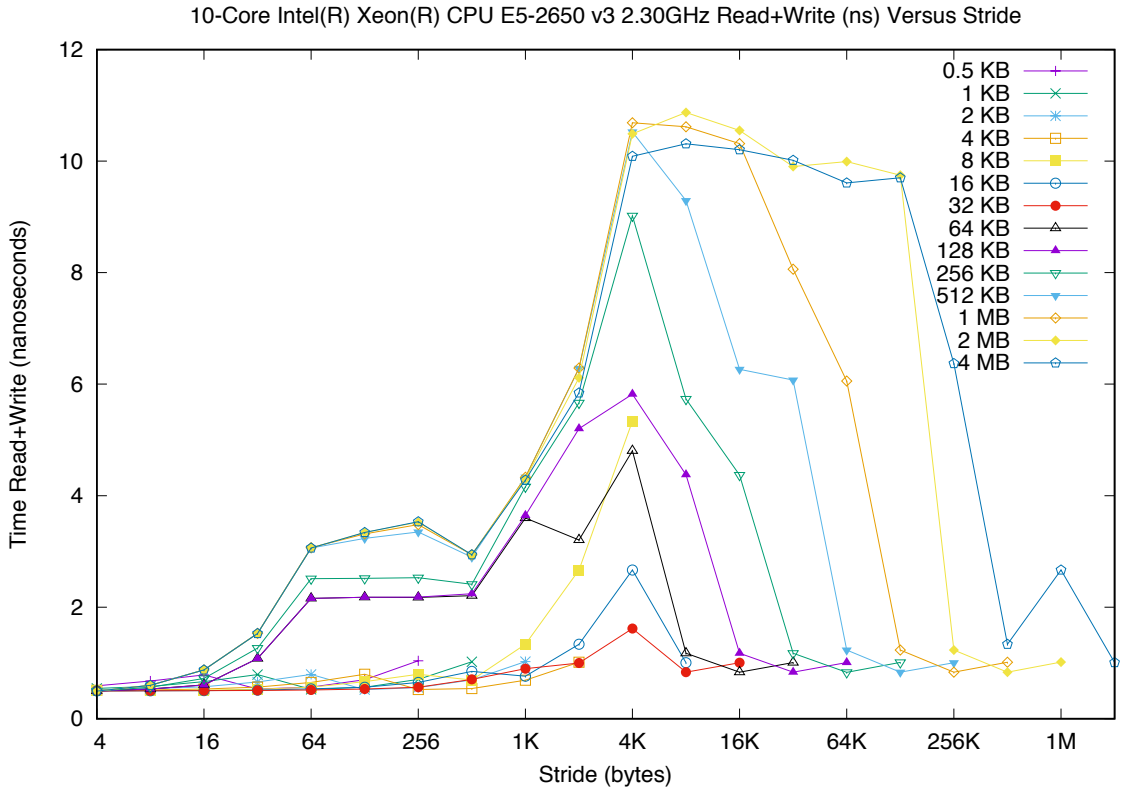

Figure 2. Graph generated on icsmaster

The ps files were also added to this assignment folder, they are called respectivelly 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.

  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 also visible in this case.

4. The points with best temporal locality in the graph are the ones closes to the end of each 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 **??**) follows the one provided in pseudo-code in the motivation file as seen in figure **??**:

   The *slow_ read* and *slow_ write* functions can be seen in more detail in figure **??**.

```
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