DATA SCIENCE &
SCIENTIFIC COMPUTING

Assignment 3 on Memory Hierarchy

Foundation of High Performance Computing

Baurice Nafack

Lecturers: **Prof. Stefano Cozzini and Prof. Luca Tornatore**

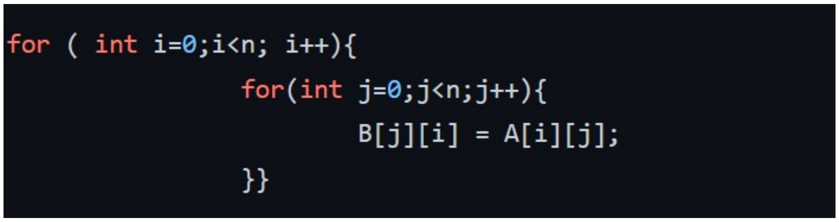
Date Last Edited: June 12, 2022

1 Analyze the performance of dense matrix transposition: Row-major and column-major access in C++

In this study, we will investigate the performance and efficient of the transpose matrix using row-majors and column-majors access.

1.1 Matrix Transpose performance

The following code helps to compute the matrix transpose of $n \times n$ matrix $B = A^T$.



```
for ( int i=0;i<n; i++){
                for(int j=0;j<n;j++){
                        B[j][i] = A[i][j];
                }}
```

I interchanged the loops over i and j (loop over i become inner loop and loop over j become the outer loop:column-wise manner) to measure the computation time for the different n values. I them display the computation time as a function of n (figure 1 and 2). To speed up the runtime of the code, we perform different levels of optimization during compilation from none $(-O0)$ through to $(-O3)$. Graph 1 and 2 below shows our results.

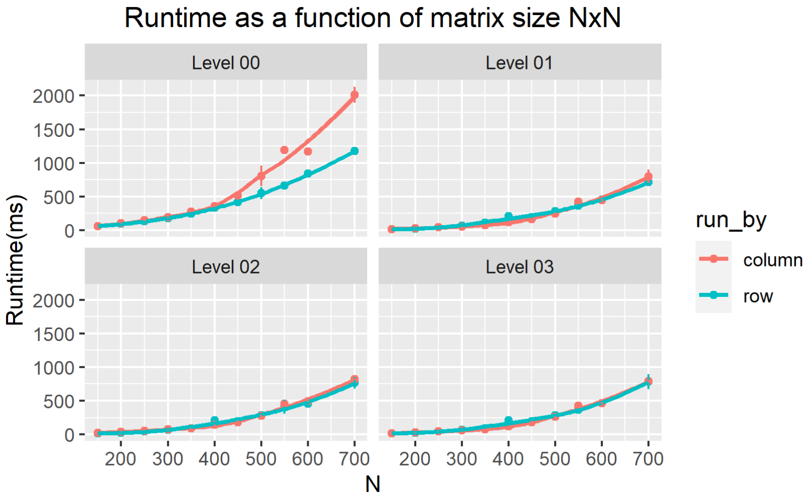


Figure 1: Matrix transposition runtime for a different level of optimization based on row-major access and column-major access.

Discusion

Graph 1 and 2 show that for level $-O0$, the most efficient code to compute the transpose of multiple matrix dimensional arrays is when using row-major access even for the higher level of optimization. We find that the row-major access for large two-dimensional arrays is approximately 2 times faster than the column-major access. This is because row-major access explores the memory non-sequentially (let's explore this in detail in the next section).

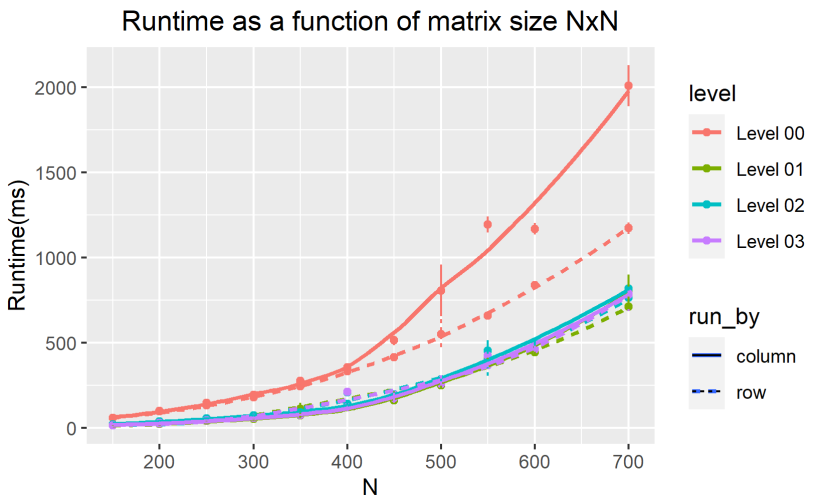


Figure 2: An overview of matrix transposition runtime based on row-major and column-major access.

2 Issues with matrix representation

We need to understand how data is stored and accessed in the memory to understand our results. Matrices in C/C++ are stored in a row-major layout (see figure 9) in the RAM, but in column-major order in Fortran. The cache memory mapping is give by the figure 3.

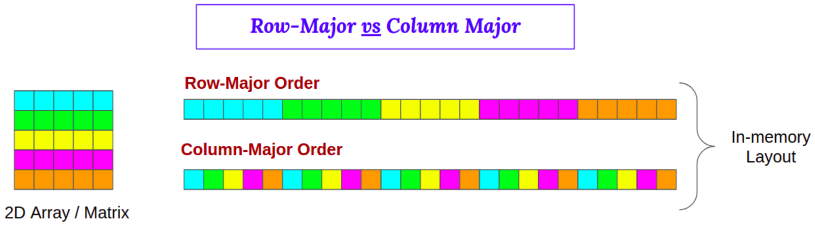


Figure 3: Cache mapping

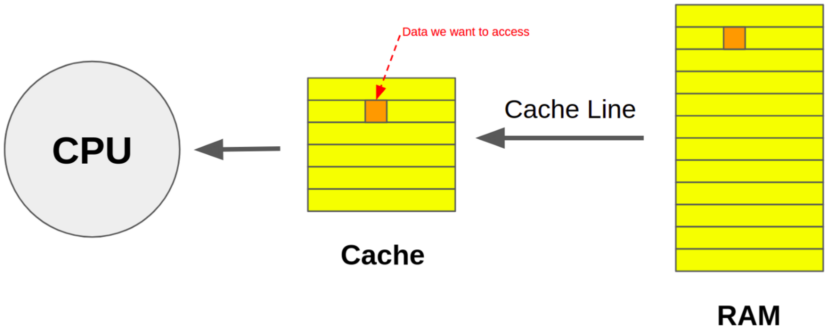


Figure 4: Matrix Transpose representation

When the CPU needs to access data, the system first checks to see if it is in the cache. This is called cache hit If the the data was discovered in the cache. As a result, data can be accessed more quickly (which is trivial since it takes longer to load data from RAM). When data cannot be loaded from the cache and must be loaded from memory, this is called a cache miss. Our findings can now be easily analyzed.

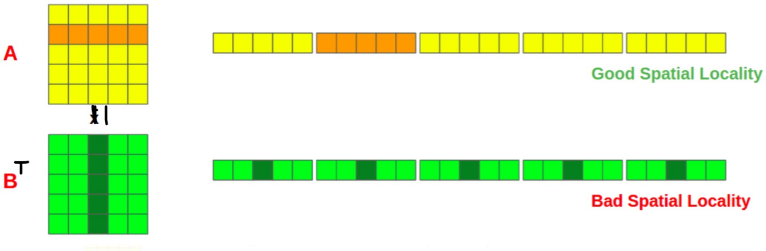
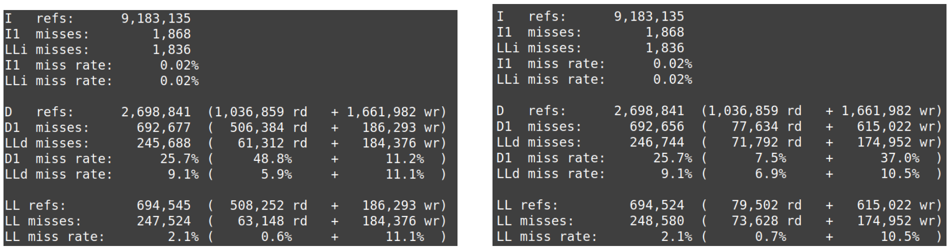


Figure 5: Matrix storing

We require access to the entire second column of matrix B as well as the entire second row of matrix A. Looking at the in-memory layout in the diagram, you can see that accessing the second row of matrix A provides good spatial locality because all of the data we require are in a single cache line. In matrix B, however, we must access different cache lines for each column value, resulting in a higher cache miss rate. So iterating the matrix column-wise isn't optimal and should be avoided if possible. Additionally, transposing the matrix should reduce L1 cache misses when using row-major access.

A will be scanned n times, so if the cache cannot hold a whole row of elements, there will be $n \times n/L$ caches misses. When n is large enough, B is scanned with stride of n for each iteration, leading to $n \times n$ cache misses. In row-major, the total number of cache misses is $n \times n + n \times n/L$. The access latency increases with each cache miss, which slows down transposition. Using the following command, we can calculate the cache miss rate for $N = 700$;
valgrind –tool=cachegrind ./cache



```
I   refs:      9,183,135
I1  misses:        1,868
LLi misses:        1,836
I1  miss rate:      0.02%
LLi miss rate:      0.02%

D   refs:      2,698,841  (1,036,859 rd   + 1,661,982 wr)
D1  misses:      692,677  (  506,384 rd   +   186,293 wr)
LLd misses:      245,688  (   61,312 rd   +   184,376 wr)
D1  miss rate:      25.7% (     48.8%     +      11.2%  )
LLd miss rate:       9.1% (      5.9%     +      11.1%  )

LL refs:         694,545  (  508,252 rd   +   186,293 wr)
LL misses:       247,524  (   63,148 rd   +   184,376 wr)
LL miss rate:        2.1% (      0.6%     +      11.1%  )
```

```
I   refs:      9,183,135
I1  misses:        1,868
LLi misses:        1,836
I1  miss rate:      0.02%
LLi miss rate:      0.02%

D   refs:      2,698,841  (1,036,859 rd   + 1,661,982 wr)
D1  misses:      692,656  (   77,634 rd   +   615,022 wr)
LLd misses:      246,744  (   71,792 rd   +   174,952 wr)
D1  miss rate:      25.7% (       7.5%    +      37.0%  )
LLd miss rate:       9.1% (       6.9%    +      10.5%  )

LL refs:         694,524  (   79,502 rd   +   615,022 wr)
LL misses:       248,580  (   73,628 rd   +   174,952 wr)
LL miss rate:        2.1% (       0.7%    +      10.5%  )
```

a) Column-major access with optimization **b) Row-major access**

Figure 6: Cache miss rate for row-major and column-major access.

My computer's L1 cache is 64 bit (I checked using getconf LEVEL1_DCACHE_LINESIZE). Since the 2D dimension array we declare is of double type (8 bits, so L= 8 elements), the theoretical total number of missing reading caches in a L1D cache line for column-major access is 551250, which is slightly lower than the experimental result. There are more reading cache misses than writing cache misses for the row-major access. We conclude, based on figure 2, that reading cache miss costs more than writing cache miss when transposing matrices.

Cachegrind simulates the first level and last level caches. An L1 miss typically costs around 10 cycles, and an LL cache miss can cost 200 cycles. The static shows us that the D1 miss rate for reading is 48.8% and 11.2% for write miss in column-major access. In column row-major access, the D1 miss rate for reading is 7.5% and 37.0% for write miss.

2.1 Array Allocation and stack

There is a limit to the array allocation when we use the automatic storage declaration to get a matrix placed in contiguous memory. We got a segmentation fault when N exceeded 700. It corresponds to $700^2 \times 8 =$

$3.8KB$ in memory. Because our stack's maximum size is $16KB$, the operating system allocates a maximum memory stack to our process. We declared the matrix as static to test our operation on a very large matrix. This help to have a matrix with with maximum size 10000×10000. We still have the same interpretation as in subsection 1.1.

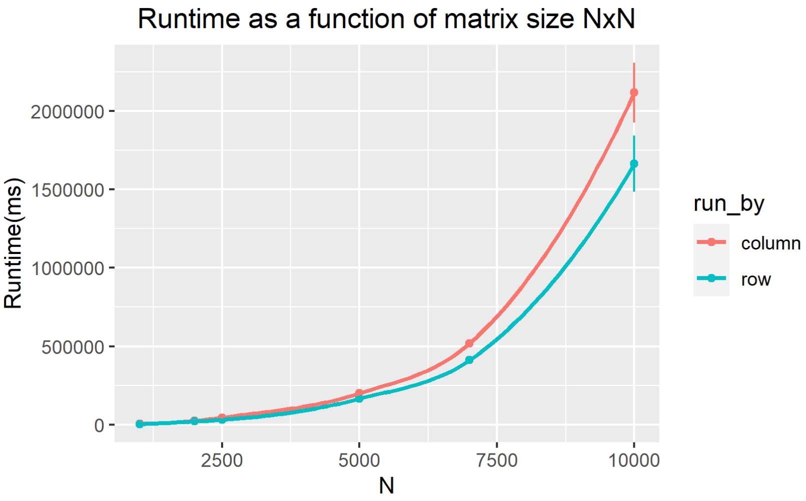
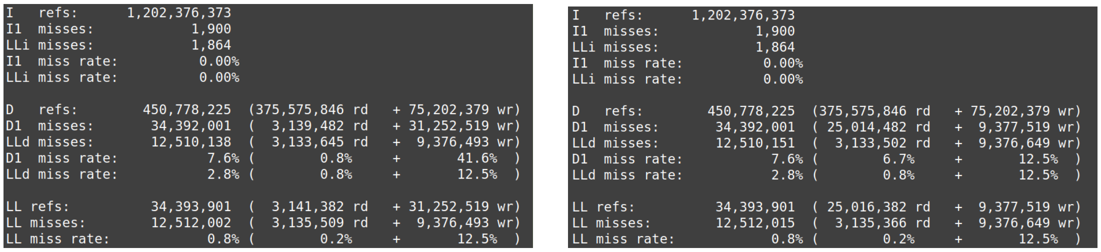


Figure 7: Matrix transposition runtime for large matrix size



```
I   refs:      1,202,376,373
I1  misses:            1,900
LLi misses:            1,864
I1  miss rate:          0.00%
LLi miss rate:          0.00%

D   refs:        450,778,225  (375,575,846 rd   + 75,202,379 wr)
D1  misses:       34,392,001  (  3,139,482 rd   + 31,252,519 wr)
LLd misses:       12,510,138  (  3,133,645 rd   +  9,376,493 wr)
D1  miss rate:           7.6% (        0.8%     +       41.6%  )
LLd miss rate:           2.8% (        0.8%     +       12.5%  )

LL refs:          34,393,901  (  3,141,382 rd   + 31,252,519 wr)
LL misses:        12,512,002  (  3,135,509 rd   +  9,376,493 wr)
LL miss rate:            0.8% (        0.2%     +       12.5%  )
```

```
I   refs:      1,202,376,373
I1  misses:            1,900
LLi misses:            1,864
I1  miss rate:          0.00%
LLi miss rate:          0.00%

D   refs:        450,778,225  (375,575,846 rd   + 75,202,379 wr)
D1  misses:       34,392,001  ( 25,014,482 rd   +  9,377,519 wr)
LLd misses:       12,510,151  (  3,133,502 rd   +  9,376,649 wr)
D1  miss rate:           7.6% (        6.7%     +       12.5%  )
LLd miss rate:           2.8% (        0.8%     +       12.5%  )

LL refs:          34,393,901  ( 25,016,382 rd   +  9,377,519 wr)
LL misses:        12,512,015  (  3,135,366 rd   +  9,376,649 wr)
LL miss rate:            0.8% (        0.2%     +       12.5%  )
```

a) Row-major access **b) Column-major access**

Figure 8: Cache miss rate for row-major and column-major access with $N = 5000$.

According to the Cachegrind simulation figure 8, row-major access has the lowest reading cache miss rate (0.8%) and writing cache miss rate (41.6%), however column-major access has the highest cache miss rate and lowest write miss rate.

3 Stream Benchmark

As CPU speeds increase by about 80% per year, memory device speeds have grown by only about 7% per year, which means a growing part of computer workloads will be determined by memory access and transfer times rather than computing time. We will propose a simple benchmark test in this part to measure the optimal memory bandwidth (MB/s) and the corresponding computation performance of four long operations. The copy operation measures the transfer rate in the absence of an arithmetic operation, the scale operation adds a simple arithmetic operation, the sum operation allows multiple loud/store operations on vector machines, and the triad operation allows chained/overlapped/fused multiplication/add operations. In a triad, the array size is defined so that each array is larger than the cache of the machine to be tested, and the code is structured so that data is not reused.

The maximum bandwidth obtained from Stream Triad results is around $54995MB/s$ and it is obtained at 12 and 24 threads. According to the graph, it doesn't matter where the memory was allocated. The array

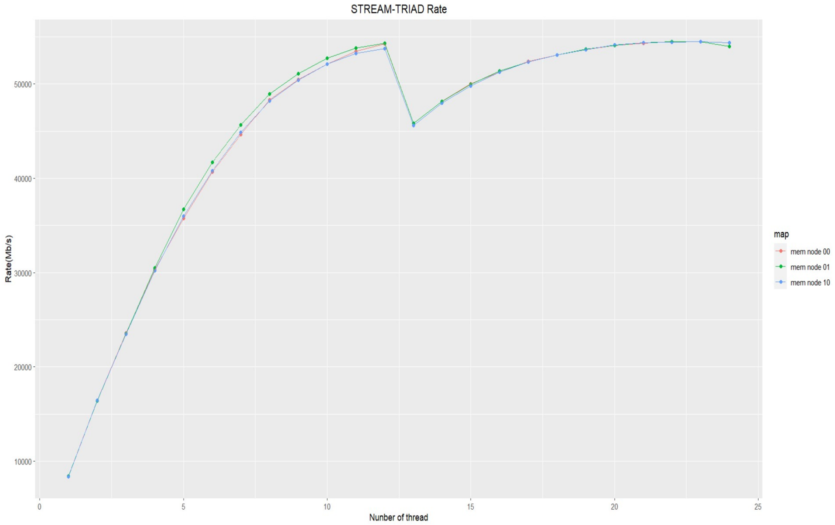


Figure 9: Triad benchmark rate

size used is 32000000.