Lab 7: MEMORY LOCALITY AND CACHE BLOCKING

Computer Architecture and Engineering (3rd year)
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Goals:

- Understand how performance is affected by program's memory access patterns.
- Modify memory access patterns to improve performance.
- Implement the *cache blocking* technique.

Matrix multiplication

Background

Matrixes are two-dimensional data structures where each element is accessed through two indexes. To multiply two matrixes, 3 simple nested loops can be used. Assuming that the matrixes A, B, and C are all of size $n \times n$ and are stored linearly in memory, a possible implementation of the multiplication algorithm ($C = A \times B$) would be the following one:

```
void multMat_ijk( int n, float *A, float *B, float *C ) {
    for ( int i = 0; i < n; i++ )
        for ( int j = 0; j < n; j++ )
            for ( int k = 0; k < n; k++ )
                C [i*n + j] + = A [k + i*n] * B [j + k*n];
}
```

Matrix multiplication is used in many linear algebra algorithms and its efficiency is essential for many applications, such as rendering in game graphics or training and inference in neural networks.

Considering the the inner loop iteration of the previous algorithm (the one carried out using index `k`), it is observed that:

- Consecutive components (stride 1) of matrix A are accessed.
- The components of matrix B are accessed with stride n.
- The same component of the matrix C (stride 0) is accessed.

It is worth mentioning that the order of the loops can be exchanged without altering the computation finally carried out. However, the order of the loops does matter for performance, since it alters the access patterns to components of matrixes A, B and C. Cache memories work better (with more hits and less misses) when memory accesses exhibit a higher spatial and temporal locality (strides are reduced as much as possible). Optimizing the memory access pattern of a program is essential for high performance and this is the proble this lab addresses.

Exercises

1. Study how the basic matrix multiplication algorithm previously presented scales with the size of the matrix. The file `matrix1.c` contains the implementation of the matrix multiplication algorithm under consideration. This implementation can be compiled with the command:

```
$ gcc -O3 -o matrix1 matrix1.c
```

⇒ Compile `matrix1.c` and run `matrix1`. Use the results of such execution to fill the following table considering the range of `n`'s value from its maximum to its minimum. Use the necessary table rows.

n	Byte size of 1 matrix ($n \times n$)	Byte size of 3 matrixes (A,B and C)	GFLOPs
2048	2048*2048*4 = 16'78 MB	16'77*3 = 50'31MB	0.658
1024	1024*1024*4 = 4'1943MB	12.58MB	1.862
512	512*512*4 = 1'04858MB	3.14574MB	1.757
256	256*256*4 = 0'262144MB	0.786432MB	2.599
128	"" = 0'065536MB	0.196608MB	3.107
64	"" = 0'015625MB	0.044687MB	3.745

We are multiplying by 4 because in the code each "cell" of the matrix has space for 1 float (=4B). Therefore, we multiply the number of cells by its capacity

Table 1: GFLOPs attending to `n`

Answer the following questions:

ONE-SIZE = size per core; ALL-SIZE= size taking into consideration all cores

⇒ Which are L1D, L2 and L3 sizes? Use `lscpu -C` to answer the question.

L1D: 32KB, L2: 32KB, L3: 16MB L3 is the one shared by all cores. The others are independent

⇒ Considering the size of the 3 matrixes, which are the values of `n` exceeding the size of L2?, and in the case of L3?

In the case of L2, the matrices 2048, 1024, 512 will exceed the size of L2. Only the matrix 2048 will surpass L3's capacity (considering all caches are empty at the beginning)

⇒ Considering the basic matrix multiplication algorithm, do you think the innner loop is good in terms of locality? why?

No, as we are accessing different blocks of memory in each iteration of k due to the way we are accessing B

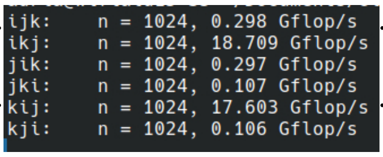
⇒ Why do you think that performance drastically, and not smoothly, decreases for high values of `n`?

Because the higher the size of each matrix, the more block changes we have to make, which is costly

2. Provide several implementations to the matrix multiplication algorithm by modifying the order of existing loops. Implement in file `matrix2.c` the 6 possible matrix multiplication algorithm variants (ijk, ikj, jik, jki, kij, kji). Compile the program and execute `matrix2`.

⇒ Which of the 6 variants is providing the best performance?
The best is ikj

Which is (or What are) the worst one(s)?
The worst is kji



```
ijk:    n = 1024, 0.298 Gflop/s
ikj:    n = 1024, 18.709 Gflop/s
jik:    n = 1024, 0.297 Gflop/s
jki:    n = 1024, 0.107 Gflop/s
kij:    n = 1024, 17.603 Gflop/s
kji:    n = 1024, 0.106 Gflop/s
```

⇒ Study the indexes used within the inner loop to access matrixes, do you find any rational to justify previous results and provide hints for the selection of a loop order to improve performance?

The best combination is ikj because the number of block changes performed by the matrixes is less frequent. As i is directly related with the block change of C and A, by putting i as the outer loop we utilize better the block than with other combinations, as these other combinations tend to change more blocks

3. Update the basic matrix multiplication algorithms with the best loop order. Modify file `matrix1.c` to reflect your loop order selection attending to the . Compile and execute the new version of `matrix1`.

   ⇒ Despite the improvement of performance, performance strongly degrades for big sizes of `n`, why is this happening?

   Although the performance has exponentially improve, as the 2048 matrix does not fit in, there is change in block and performance decreases

Matrix transpose

Background

Sometimes we want to swap rows and columns of a matrix. This operation is called *transpose*. An efficient implementation of this operation can be very useful when performing complex linear algebra operations. The transpose of the matrix A is denoted A^T. The following figure illustrates an example of transposing the elements of a 5x5 matrix:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

⇒

1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24
5	10	15	20	25

The following code, is the most basic code to perform matrix transpose:

```
for( i = 0; i < n; i++ )
    for( j = 0; j < n; j++ )
        dst[i*n +j] = src[i + j*n];
```

As it happens with matrix multiplication, this code leads to multiple cache misses due to the low reuse of cache data. The *cache blocking* technique can promote such data reuse.

The cache blocking technique reduces the cache miss rate by improving the temporal and spatial locality of memory accesses. When the technique is applied to matrix transpose, the matrix is divided into sub-matrixes A_{ij}, and each sub-matrix is transposed separately to its final location in the transpose matrix, as shown in the following figure:

$$\begin{array}{|c|c|c|}\hline A_{11} & A_{12} & A_{13} \\ \hline A_{21} & A_{22} & A_{23} \\ \hline A_{31} & A_{32} & A_{33} \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|c|}\hline A_{11}^T & A_{21}^T & A_{31}^T \\ \hline A_{12}^T & A_{22}^T & A_{32}^T \\ \hline A_{13}^T & A_{23}^T & A_{33}^T \\ \hline \end{array}$$

This technique significantly reduces the size of the the algorithm data set, which results in a cache miss rate reduction and consequently, a performance improvement.

Exercices

1. Implement matrix transponse using the cache blocking technique.

   The file `transpose.c` contains two functions that perform matrix transponse. The first function implements the basic algorithm seen above. The second function must be implemented by the student.

   Think about the following questions before carrying out the implemention:

   ⇒ How many loops should contain the algorithm using cache blocking?
   We will use 4 loops: the bi, bj, i, j distribution

   ⇒ Which is the goal of the two outer loops?
   Traverse the different blocks

   ⇒ Which is the range of elements covered by each loop?
   BLOCKSIZE elements.

   ⇒ How much should loop indexes be incremented in each iteration?
   In each iteration loop, the outer loops must be incremented by BLOCKSIZE amount, while the 2 inner loops increment by 1 until finishing the loop

   Once all questions have been answered, go on with the implementation and compile and execute the file `transpose.c` to check its correct operation and quantify the resulting performance improvement.

Improve the performance of matrix multiplication using cache blocking

The cache blocking technique can also be applied to matrix multiplication. Propose an implementation based on the best loop order you have previously identified. Apply the cache blocking technique to the **two most outer loops** of the algorithm and store the implementation in file `matrix3.c`. Next, check the performance improvement with respect to `matrix1`, specially for big values of `n`.