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 x n)	Byte size of 3 matrices (A, B, and C)	GFLOPs
2048	16777216B = 16MB	50331648B = 48MB	0.352
1024	4194304B = 4MB	12582912B = 12MB	0.68
512	1048576B = 1MB	3145728B = 3MB	1.837
256	262144B = 256KB	786432B = 768KB	3.284
128	65536B = 64KB	196608B = 192KB	3.606
64	16384B = 16KB	49152B = 48KB	5.14

Table 1: GFLOPs attending to `n`

Answer the following questions:

⇒ Which are L1D, L2 and L3 sizes? Use `lscpu -C` to answer the question.
L1D: 48KB L2: 1.3MB L3: 24MB

⇒ 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 my case, n = 512 (~3MB) already exceeds the size L2, and n = 2048 (~50MB) exceeds the size of L3.

⇒ Considering the basic matrix multiplication algorithm, do you think the innner loop is good in terms of locality? why?
No, because if matrices are very long, cache blocks loaded at the beginning of the rows are unloaded before finishing the row, and therefore we are only taking advantage of one of the values of every block loaded in cache.

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

- We see a big decrease in performance from n = 64 to n = 128, as n = 128 exceeds the size of my L1D cache.
- Another big drop occurs from n = 256 to n = 512, as n = 512 exceeds the size of my L2 cache.
- The biggest drop occurs from n = 512 to n = 1024. Both of them exceed the size of L2 but not of L3, so the drop is likely related to the locality problem with our loop that accesses B elements with stride n. Probably, cache blocks loaded to access just one element are being replaced before accessing more elements, and then they have to be reallocated again several times.
- There is another drop from n = 1024 to n = 2048, as n = 2048 exceeds the size of L3.

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?
ikj, at 17.089 Gflops, closely followed by kij, at 16.105 Gflops.

Which is (or What are) the worst one(s)?
All the rest did much worse, but the 2 worst ones are kji, at 0.340 Gflops, and jki, at 0.325 Gflops.

⇒ 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?
In both cases, j is the inner loop. Therefore, elements in C and B are accessed sequentially (one after the other), and in A, we keep accessing the same element in every iteration of the loop j.

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?
Because L1D, L2, and L3 still fill up at some point.

Matrix transponse

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 transponse:

```
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?
   4 (2 for the blocks, and 2 for calculating the transpose of the block)

   ⇒ Which is the goal of the two outer loops?
   To select the block to compute next. They divide the matrix in blocks that fit in the cache.

   ⇒ Which is the range of elements covered by each loop?
   The outer 2 just "select" the block to compute, while the inner 2, while the inner 2 cover every element in the block.

   ⇒ How much should loop indexes be incremented in each iteration?
   The loops that select the block should increment it by BLOCKSIZE, while the others just by one, to move to the next element.

   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`.
For large numbers of n (such as 4096), there is a very big increase in performance (almost 2x).