Hands On Lab: Memory

Juanjo Costa <jcosta@ac.upc.edu>

Mon, 16 Apr 2023

Contents

Goals

- Evaluate the performance of a code (execution time, cycles)
- Detect the memory access pattern of a code
- Understand the impact of the cache in a code
- Basic use of functions to manage dynamic memory
- Use of the GPU

Directory structure

In the supplied code the following directories are found each with the source code required for the following sections:

- 1_matrix (Performance Evaluation)
- 2_access_pattern (Access pattern)
- 3_dynamic_memory (Dynamic Memory)
- tools (Utilities to measure time and analysis)

Performance Evaluation

It is important to know how to evaluate the performance of our codes. Know how much time our code takes. In our context (video games) it is imperative since this defines the performance of our games, the number of frames that I can show per second. The less time it takes to "calculate" a frame, more frames I could generate.

Measuring the time it takes to execute a a piece of code is relatively simple if I have a function *captureTime* that works as a stopwatch:

```
...
t1 = captureTime();

// code to measure

t2 = captureTime();
temps = t2 - t1; //Time it takes to execute the code
...
```

The only problem is that the resolution and cost of this *captureTime* depends of the hardware and the operating system used. In the case of Windows (TM), its documentation gives us a pretty detailed explanation [TIME].

To give you an idea, in the *measure.c* file, you will find the routines *start_counter* and *get_counter* which basically serve as a stopwatch, returning the number of microseconds that have passed between the two calls. These routines work both on Windows and on Linux.

An additional problem is that the time it takes to execute a certain code is not always exactly the same due to the complexity of the factors involved nowadays. The code to be measured does not run directly on the architecture, but rather there is an operating system, and devices that interrupt the usual flow of execution, ... therefore, what we usually do is run the same code several times and do a statistical analysis of the results to get a value that is more or less stable. The *measure_full* routine, found in *measure.c* file, automates this process and return the minimum execution time of a code. In particular, it runs the code several times, measuring their run times and keeping these values until the difference in time between the smaller values is less than 1 percent (1% or 0.01), which is the factor that we have decided to consider a measure of time stable enough. This routine will be used in the following exercises.

Evaluating arrays ...

To begin with we are going to evaluate a very simple code of matrix traversals. We want to see if it takes the same time to traverse a matrix by rows or by columns. We will see that for small matrixes, both traversals are similar, but for bigger matrixes... this is not true any more.

In *matriu.c* file you have a code that traverses a very small array (4x4). The code shows how long it takes in microseconds to traverse the matrix by rows and by columns. The shown time is the smallest after executing up to 10 times the same code, evaluating that the error between the 3 lowest values is smaller than 1% (0.01).

For the evaluation, compile and execute the application, write down the time it takes for each traversal, increase the matrix rows and columns sizes (using the same value for both) using powers of 2 (8, 16, 32, ...), and repeat.

- Make a table with the size of the array and the execution times for both traversals.
- Is the execution time similar when you the matrix is traversed by rows than when it is traversed by columns? (Repeat a couple of executions to be sure)
- After evaluating the results, what can you extract from it? Is there any change in the execution times for both traversals? How much does it occupy 1 row? Why do you think it happens?
- Compare the cache levels of your processor [CPUDB] with the results that you get in the table.

Access patterns and impact of the cache

The structures we use to store information (such as arrays) should be linearized to saved them to memory (as we saw in theory). And therefore the pattern used to access these structures affects performance (as we have seen in the previous section). Let's see exactly how.

In the following exercise we present 3 programs: *mm-ijk.c, mm-jki.c and mm-kij.c*. They correspond to 3 of the 6 forms ijk of the product of matrices. For example, in the *mm-ijk.c* program the next portion of code is measured (ijk means that the more external loop is the loop controlled by the variable 'i' and the more internal one is controlled by 'k'):

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

The 3 programs perform, in a different way, the same operation: the product of matrices: C = A * B.

There is a macro that defines the size of the arrays, by default of 256x256 floats:

```
#ifndef N
#define N 256  /* Default dimension */
#endif
```

If this size is N = 6 (or smaller), it shows the contents of the resulting array C, so you can check that the 3 programs do exactly the same. If it is larger, it only makes the product and returns the runtime in milliseconds.

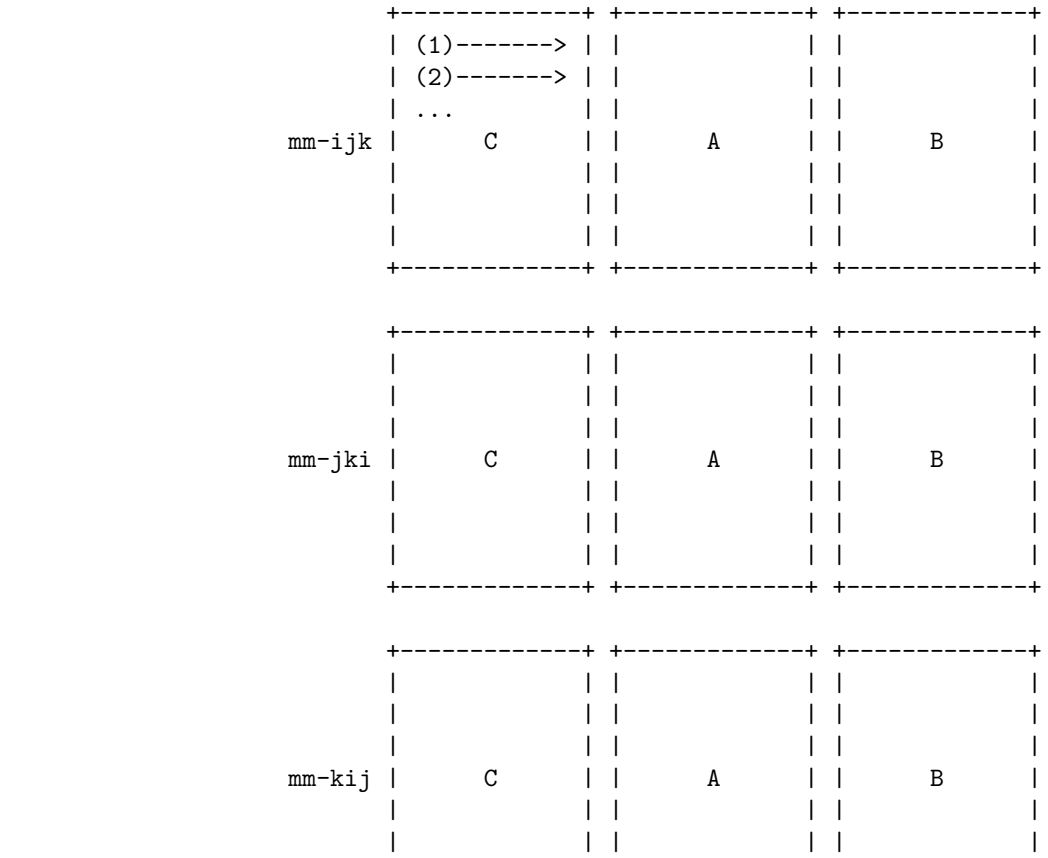
For the following exercises, choose one of the cases.

- It is important to know the access pattern to memory of a code. For example in the previous code, matrix C is accessed through C[i][j] and due to the order of the 'i' and 'j' loops, the following elements are accessed in order:

```
C[0][0], C[0][1], C[0][2], ... C[0][N],
C[1][0], C[1][1], C[1][2], ... C[1][N],
...
```

Therefore, all the elements from the first row are accessed, then all from the second and so on.

Draw, for the chosen case, the order in which the matrices A, B and C are traversed (the order in which the first matrix is traversed is included in the drawing for reference):



```
        |             | |             | |             |
        +-------------+ +-------------+ +-------------+
```

- Write down the execution time of the chosen version:

```
       | Temps execució (en ms)
     N | mm-ijk | mm-jki | mm-kij
-------+--------+--------+--------
   256 |        |        |
   512 |        |        |
  1024 |        |        |
```

- Assuming that each element in the array occupies 4 bytes, and that the size of our system's page is 4Kbytes, calculate for the chosen case the number of virtual memory pages that are used when completely executing the most internal loop 1 time. For example, in the mm-ijk case, at each execution of the 'k' loop 'C' matrix access a single element (4 bytes), while the 'A' matrix accesses a whole row (consecutive elements) and 'B' matrix accesses a whole column (non-consecutive elements).

```
       |             mm-ijk             |
     N | matriu A | matriu B | matriu C |
-------+----------+----------+----------+
   256 |          |          |    1     |
   512 |          |          |          |
  1024 |          |          |          |
-------+----------+----------+----------+

       |             mm-jki             |
     N | matriu A | matriu B | matriu C |
-------+----------+----------+----------+
   256 |          |          |          |
   512 |          |          |          |
  1024 |          |          |          |
-------+----------+----------+----------+

       |             mm-kij             |
     N | matriu A | matriu B | matriu C |
-------+----------+----------+----------+
   256 |          |          |          |
   512 |          |          |          |
  1024 |          |          |          |
-------+----------+----------+----------+
```

Dynamic memory

The *malloc* and *free* routines from the C library allow a user program to request dynamic memory to the operating system. The operating system assigns new space to the process, but because the granularity of the call (that requests bytes) and the granularity of the OS (reserves and maps pages of 4096 bytes) do not match, it may happen that the user can execute non-legal code .

The *mem.c* file contains a code that reserves an array of 10 characters and initializes its positions. Then try to access positions outside of this array.

- Run the code, observe the result and try to deduce what is happening.

The *mem2.c* file shows the pernicious effects of using memory that we have not explicitly reserved.

- Run the code, observe the result and try to deduce what is happening.

GPU fun

Nowadays, although the cache is important, in the context of video games, the workload is carried out by the GPU and therefore it is even more important. At [GPU] Keith O'Conor (coder and CTO of Romero Games) gives a very good explanation of the importance of GPUs (and enough references to be entertained for a looong time).

References

..[TIME]: https://docs.microsoft.com/es-es/windows/desktop/SysInfo/acquiring-high-resolution-time-stamps "Acquiring high-resolution time stamps"

..[CSAPP]: http://csapp.cs.cmu.edu/2e/home.html "Computer Systems: A Programmer's Perspective. Randal E. Bryant and David R. O'Hallaron, Carnegie Mellon University, 3rEdition"

..[MOUNT]: http://csapp.cs.cmu.edu/3e/mountain.tar "Memory Mountain source code"

..[CPUDB]: https://www.techpowerup.com/cpudb/ "CPU Database"

..[GPU]: http://www.fragmentbuffer.com/gpu-performance-for-game-artists/ "GPU performance for game artists. Keith O'Conor"

..[Romero Games]: https://www.romerogames.ie/