Question

Below are two programs that are almost identical except that I switched the i and j variables around. They both run in different amounts of time. Could someone explain why this happens?

Version 1

```
#include <stdio.h>
#include <stdlib.h>

main () {
  int i,j;
  static int x[4000][4000];
  for (i = 0; i < 4000; i++) {
    for (j = 0; j < 4000; j++) {
      x[j][i] = i + j; }
  }
}
```

Version 2

```
#include <stdio.h>
#include <stdlib.h>

main () {
  int i,j;
  static int x[4000][4000];
  for (j = 0; j < 4000; j++) {
     for (i = 0; i < 4000; i++) {
       x[j][i] = i + j; }
   }
}
```

Answer

As others have said, the issue is the store to the memory location in the array: `x[i][j]`. Here's a bit of insight why:

You have a 2-dimensional array, but memory in the computer is inherently 1-dimensional. So while you imagine your array like this:

```
0,0 | 0,1 | 0,2 | 0,3
----+-----+-----+----
1,0 | 1,1 | 1,2 | 1,3
----+-----+-----+----
2,0 | 2,1 | 2,2 | 2,3
```

Your computer stores it in memory as a single line:

`0,0 | 0,1 | 0,2 | 0,3 | 1,0 | 1,1 | 1,2 | 1,3 | 2,0 | 2,1 | 2,2 | 2,3`

In the 2nd example, you access the array by looping over the 2nd number first, i.e.:

```
x[0][0] 
        x[0][1]
                x[0][2]
                        x[0][3]
                                x[1][0] etc...
```

Meaning that you're hitting them all in order. Now look at the 1st version. You're doing:

```
x[0][0]
                                x[1][0]
                                                                x[2][0]
        x[0][1]
                                        x[1][1] etc...
```

Because of the way C laid out the 2-d array in memory, you're asking it to jump all over the place. But now for the kicker: Why does this matter? All memory accesses are the same, right?

No: because of caches. Data from your memory gets brought over to the CPU in little chunks (called 'cache lines'), typically 64 bytes. If you have 4-byte integers, that means you're geting 16 consecutive integers in a neat little bundle. It's actually fairly slow to fetch these chunks of memory; your CPU can do a lot of work in the time it takes for a single cache line to load.

Now look back at the order of accesses:

The second example is: 1. grabbing a chunk of 16 ints 2. modifying all of them 3. repeat 4000*4000/16 times.

That's nice and fast, and the CPU always has something to work on.

The first example is: 1. grab a chunk of 16 ints 2. modify only one of them 3. repeat 4000*4000 times

That's going to require 16 times the number of "fetches" from memory. Your CPU will actually have to spend time sitting around waiting for that memory to show up, and while it's sitting around you're wasting valuable time.

Important Note:

Now that you have the answer, here's an interesting note: there's no inherent reason that your second example has to be the fast one. For instance, in Fortran, the first example would be fast and the second one slow. That's because instead of expanding things out into conceptual "rows" like C does, Fortran expands into "columns", i.e.:

`0,0 | 1,0 | 2,0 | 0,1 | 1,1 | 2,1 | 0,2 | 1,2 | 2,2 | 0,3 | 1,3 | 2,3`

The layout of C is called 'row-major' and Fortran's is called 'column-major'. As you can see, it's very important to know whether your programming language is row-major or column-major! Here's a link for more info: http://en.wikipedia.org/wiki/Row-major_order