Quentin Pinçon  **Game Of Life using CUDA**

In this project, the goal was to implement the Conway’s game of life using CUDA, a developing technology invented by NVIDIA for their GPUs. We were given some functions creating a table of 0 and 1, corresponding to a cell, and reading it. The goal is to determine the next state of each cell, using only its present state and the ones of its neighbors. This project seems perfect to be implemented using a GPU. First, we will see the implementation with one cell per thread, then with multiple cells per threads, and to finish with multiple cells per threads, using shared memory.

**I] One cell per thread**

For this part I decided to launch as much as threads than the number of elements of the array, that is N². So each thread deals with a single element, by looking at its neighbors. In my implementation, I use size of block of 25x25 threads. That is arbitrary, and can be changed. Before launching the computation, we need first to set the grid of blocks. I went for a square grid, of size N/25 because in my implementation, I use size of block of 25x25 threads. That is arbitrary, and can be changed.

We can easily deduct from the variable blockIdx and threadIdx where the thread is in the grid. From these variables, we can find the corresponding element in the initial array. The tricky point was to make the initial array circular. To do that, I didn’t change the initial array (maybe I should have, we will see it later), but I used modulo operation.

I made my kernel function really simple by defining a function that counts the neighbors of a given cell. This is in this function that the modulo operations are. Maybe they are a bit complicated but I didn’t find anything better than this, and they work just fine.

After that function being computed, which returns the number of alive neighbors, I just check their number according to the rules of the game and change the output array accordingly.

For multiple iterations, I didn’t want to copy any array so that I don’t lose time for nothing, that is why every iteration I change the order of the input and output arrays. I didn’t want to manipulate pointers of device memory since I didn’t know for sure their behavior.

At the end of the function the new table is stored into a new file : table XXXxXXX-Yit.bin, Y being the number of iterations.

To try if it worked, I had of course to have a reference to compare with, that is why I wrote a serial function also. I figured that I could use the function that I only defined for the device to do it, that is why countAlive() is a device and host function. This function is just doing with 2 for loops what the kernel function is doing with threads. We will see later how the results are drastically different.

To compare, I also created a function print to check visually the results, see if they are possible. I did almost all my testing with a 50x50 table, to check easily if the results are the same from an implementation to another. I could have also create a checking function that scan 2 arrays element by element to see if they are the same, but I didn’t find it necessary.

**II] Multiple cells per thread**

For this part, I used a lot of the previous implementation, since the principle in the same. Since there is more cells handled by one thread, we need to make the grid smaller. I decided to go not for a square grid as before, but for a rectangular one. Only the abscissa is changed, not the ordinate, meaning that a thread will take care of a number of cells contiguous is memory, like is showing the following image :

Four cells per thread

One cell per thread

Block(1,0)

Block(1,1)

Block(1,0)

Block(0,0)

Block(0,1)

Block(0,0)

Thread

(0,0)

ArraySize

ArraySize

0 1 2 3

The proportions are not correct, but the principle is here.

To do this, to the previous code is added a for loop of a number of cells. This number of thread can be chosen in my program. The index of the array access just need to be changed so that the result would be correct.

**III] Multiple threads with shared memory**

Now, we will talk about the implementation using the block shared memories. In theory, the access of shared memory is about 100 times faster than the access to global memory, making it really fast, almost as fast as register access.

For instance, if we take a block of 25x25 threads, in the basic implementation without shared memory, we will have 25x25x8 (8 per cell, the neighbors) = 5000.

Now let’s say we store every cell needed into shared memory to compute those 25x25 cells, we need 27x27 cells into the shared memory, because we want all cells to be computed plus all cells on the sides of the block, which won’t be computed but will be needed for the others. So with this configuration we make 27x27 calls to global memory to fill the shared memory, which is only 729 calls, more than 5 times less than the first configuration. We can omit the accesses to shared memory after because they are almost instant.

This fact is even more noticeable with multiple cells per blocks. My configuration, which we saw before, doesn’t change the size of the block but only the size of the grid.x. Let’s take for instance 4 blocks per threads. Now, a 25x25 block makes 25x25x4x8 calls to global memory, which is 20000 calls.

The shared memory has the dimensions of (25+2)x(25\*4 + 2), because only the size in the x direction changes. It’s a size of 2754 elements, as many accesses to global memory to do. This time it’s almost 10 times less than the regular code without shared memory !

This is, of course, only theory. It assumes than the filling of shared memory can be done without any practical problem, which is not the case. For example, in my case, I didn’t manage to make it better than the previous versions ; it is even longer than them.

I first tried to make the threads on the sides of the block fill the shared memory, doing for each thread the bottom left and the top left for the left column for example, like this :

Threads

v

Shared memory

But after some testing, I realized that the time was not better. I figured it was because there were some bank conflicts accessing the global memory, and that I should correct it to have better wall clock times. That was led to the actual implementation there is my code.

To avoid the bank conflicts, I made conditions for all possible cases : is the thread is at top right corner, if it’s on the right side, on the lower right corner, one the down side… so that every part of the global memory is only accessed once within one block.

This implementation, which sounds optimized on paper, is not faster than the previous versions neither. The reason is that there is way too much if statements in the code to check the thread disposition in the block. Ifs depending on the position of the thread in CUDA are not good, because when a thread passes a condition, the others that don’t just go idle instead of doing other things.

I think I looked at the problem the wrong way and therefore, didn’t choose the right solution to implement. Instead of launching the same number of threads as before, maybe I should have launched more threads to fill the shared memory and then make some of them go idle. I think it’s a better way to do it. To do this, the input table should be changed as well, by making it bigger, so that the sides of the matrix are projections of the last opposite side.

Maybe this implementation would have been better, anyway theoretically it is, and unfortunately I didn’t have the time to do it.

Also, a good idea that I saw online is to use more than one cell per integer, since we only use one bit per cell. This way we could have more cells with the same number of integers (maybe 32), and therefore an improving of the results. I thought it was worth mentioning.

Now let’s take a look at the results (all in milliseconds) :

**Table500x500.bin**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Iterations** | **One cell** | **2 cells** | **4 cells** | **10 cells** | **2 cells shared** | **4 cells shared** | **10 cells shared** | **Serial** |
| 10 | 1,2 | 1 | 1,1 | 2,6 | 1,9 | 1,7 | 1,7 |  |
| 100 | 10,9 | 10,9 | 10,9 | 11 | 18,6 | 16,4 | 16,4 |  |
| 1000 | 104,4 | 91,2 | 105,2 | 247,3 | 184,3 | 162,2 | 160,4 | 24460 |

**Table1000x1000.bin**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Iterations** | **One cell** | **2 cells** | **4 cells** | **10 cells** | **2 cells shared** | **4 cells shared** | **10 cells shared** | **Serial** |
| 10 | 4,1 | 3,6 | 4,3 | 10,1 | 7 | 6,1 | 5,9 |  |
| 100 | 40 | 33,9 | 42 | 98,5 | 69,1 | 60,1 | 58,7 |  |
| 1000 |  | 329,1 | 406,6 | 963 | 682,7 | 593,7 | 580,7 | 98750 |

**Table2000x2000.bin**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Iterations** | **One cell** | **2 cells** | **4 cells** | **10 cells** | **2 cells shared** | **4 cells shared** | **10 cells shared** | **Serial** |
| 10 | 15,6 | 12,7 | 14,4 | 37,4 | 27,2 | 23,6 | 23 |  |
| 100 | 149,9 | 123 | 138,5 | 364,9 | 269,6 | 233,9 | 226,1 |  |
| 1000 | 1429 | 1186 | 1333,4 | 3560,2 | 2666,5 | 2308,1 | 2231,3 | 392790 |

**Table 4000x4000.bin**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Iterations** | **One cell** | **2 cells** | **4 cells** | **10 cells** | **2 cells shared** | **4 cells shared** | **10 cells shared** | **Serial** |
| 10 | 60,5 | 48,7 | 53,1 | 150 | 108,1 | 93,6 | 90,8 | 21560 |
| 100 | 587,1 | 478,2 | 514,4 | 1456,5 | 1072,8 | 928,8 | 900,1 | 178590 |
| 1000 | 5593 | 4615 | 4957,8 | 14221,6 | 10609,2 | 9162,9 | 8894,7 | Very long |

The first thing we see by analyzing those arrays is that the computation using CUDA is far faster than the CPU computation, beyond comparison (about 350 times faster). The reason is that this problem is really suitable for GPU usage, because a use of a big amount of ALUs is important. We won’t be talking further about the speedup from serial to GPU, since there is no important point; it’s the technology which makes the change.

During my testing, I realized that the optimal number of cells handled by one thread should be 2 to have better results, and the arrays shows it well. This is as far as the use of shared memory is excluded. The results are globally 20 % faster with 2 cells per thread. With four cells per threads the results are still good than with one cell per thread, but not as much as with 2. In another hand, results are really bad for 10 cells per threads, showing that it’s not good for performance to let a thread to a larger number of calculations.

The reason of this I think is that the number of pending threads is less important with multiple cells per threads, so the computation can occur really fast without any thread waiting. However as the number of cells per thread rises, the number of calculations to do grows also, resulting into an increasing of the time of computation

Now with the use of shared memory, we have a different situation. As we saw in the calculations before, in theory the more there are cells per threads, the more of shared benefits are used, since we do more and more accesses to it with a slow amount of global memory access to fill the shared memory. This is why the time is decreasing with 2, 4 or 10 cells per threads. However with a higher number of cells per threads, the limit of size of shared memory is reached, so this number can’t go too high.

As you can see the results are not satisfying though. Indeed as discussed before, the if statements are really slowing down the program, by doing a really slow filling of the shared memory. When a condition is evaluated depending of the position of a thread, the corresponding thread we do the computation inside the if. However, the other threads will have to go idle and wait for the end of this block to resume to their computation. Since in my code there is a high number of ifs, the computation is far from being optimal for the threads.

Most of the program spends the time by filling the shared memory, but accessing it is really fast.

As we have seen, Conway’s Game Of Life is really suitable for a CUDA implementation. The results of classical CPU and of GPU use are from different worlds. However even with this in mind, there is always room for optimization, by using a thread for more computation, or by using shared memory, which unfortunately I partially failed. There is still other ways to improve the results, for example store more than one cell in an integer.