# Accelerating 3D Cellular automata computation with GP-GPU in the context of integrative biology

Jonathan Caux, David R.C. Hill, Pridi Siregar

# Accelerating 3D Cellular automata computation with GP-GPU in the context of integrative biology

**Jonathan Caux**[1] **David Hill**[2] **Pridi Siregar**[3]

**Research Report LIMOS/RR-10-10**

29 avril 2010

[1] LIMOS-ISIMA, Campus des Cézeaux, 63177 Aubière, jonathan.caux@isima.fr
[2] LIMOS-ISIMA, Campus des Cézeaux, 63177 Aubière, drch@isima.fr
[3] Integrative Biocomputing, Nouvelles Structures – Place du Granier 35135 CHANTEPIE, pridi.siregar@ibiocomputing.com

# Abstract

In this paper we explore the possibility of using GP GPU technology (General Purpose Graphical Processing Unit) in the context of integrative biology. For more than a decade, 3D cellular automata represent a promising approach to handling multi-scale modeling of organs. However, the computing time of such huge automata has limited the experiments. Current GP GPUs now allow the execution of hundreds of threads with a regular PC hosting a device card. This capability can be exploited in the case of cellular automata where each cell has to compute the same algorithm. We have implemented two algorithms to compare different memory usage. The performances show very significant speedup even when compared to the latest CPU processors. The interconnection of GP GPU boards and servers will be considered to build a local grid of hybrid machines.



**Keywords**: GP GPU, Parallel computing, 3D cellular automata, Integrative Biology.



# Résumé

Cet article étudie l'opportunité d'utiliser la technologie GP GPU (General Purpose Graphical Processing Unit) dans le contexte de la biologie intégrative. Depuis plus d'une décennie, les automates cellulaires 3D représentent une approche prometteuse pour manipuler les modèles d'organes multi-échelle. Cependant, les temps de calcul nécessaire pour de tels automates ont limités les expérimentations. Les nouveaux GP GPU permettent maintenant l'exécution de centaines de threads à partir d'un PC standard équipé d'une carte spécifique. Cette capacité peut être exploitée pour les automates cellulaires pour lesquelles il est nécessaire de réaliser le même algorithme pour chaque cellule. Nous avons implémenté deux algorithmes pour comparer les différentes utilisations de la mémoire. Les performances sont significativement améliorées même lorsque l'on compare aux derniers processeurs CPU. L'interconnexion de carte GP GPU et de serveurs sera considérée pour la création d'une grille locale de machine hybride.



**Mots-clés** : GP GPU, calcul parallèle, automate cellulaire 3D, biologie intégrative.

## 1 Introduction

Integrative biology has a large number of solutions available to simulate multi-scale models: Reaction-Diffusion Equations [1], Dissipative particle dynamics [2] or Finite element method [3]. But cellular automata can also be used [4], this has been shown for instance to simulate a part of an organ behavior [5] or the depolarization and repolarization of the heart muscle cells which can be seen as an excitable medium [6]. In this paper, we consider 3D cellular automata, consisting in a cube of cells where each cell can be in a specific state. A finite number of states are considered and a cell state evolves over time according to its current state, but also according to the states of the other cells in a defined neighborhood. When we simulate the evolution of an automaton, the future state of each cell is computed simultaneously according to the state of the cube cells at the current simulation time. With basic rules, it is possible to create a simplified model for natural behavior. The most common example is the Game of Life [7]. This model, created in 1970 by John Conway, allows simulating bacteria reproduction on a 2D toric space. In this model, a cell can only have two states: dead or alive, and evolves according to the number of cells alive in the contacting neighborhood (named Moore's neighborhood). But cellular automata can't be reduced to Conway's Game of Life; the interested reader can consult [8]. For instance, electronic circuits can be simulated with the WireWorld cellular automata [9], and in biology, excitable medium, among others models, can also be simulated with cellular automata [10].

Even if cellular automata are mostly very simple, large 3D models with an important number of iterations require a very long computing time. Moreover, in integrative biology, computation of cellular automata is often only a part of a more complex problem. Working on larger cubes can therefore improve the result sensitivity, and performing this result faster can speed up the global process.

Our idea was to test cellular automaton implementation on GP-GPU (General Purpose Graphic Processing Unit)[4] [11]. This kind of processing unit was first designed to process graphics on a computer, but while the classical CPU computation performances evolution recently began to slow down, the GP-GPU continued to provide very significant speedup. Ten years ago, developers of high performance computing applications started to port scientific software from CPU to GP-GPU to make the most of it [12]. After the initial success, GPU manufacturers started to work on friendlier API for general purpose computation and we are now able to develop directly in languages which are close variants of the C language[5][6]. In the same way, other hardware accelerators like FPGA have been considered to speed up cellular automata computations [13].

The main difficulty lies in the memory manipulation since GP-GPU have various levels of memory with different performances. The GP-GPU global memory which can be accessed by any thread at any time has very important access latency. The shared memory available for each streaming multiprocessor inside a GP-GPU does not have such latency, but this memory is only shared by threads running on the same multiprocessor. In addition, the number of concurrent threads in a streaming multiprocessor is limited. Moreover, memory transfer between the host computer and the GP-GPU device can severely damage the global speedup if the computation time is not significant enough in comparison with the data transfer time.

In this paper, we propose two cellular automata implementations on GP-GPU with different memory usage (Part 3). To understand the benefits of GP-GPU, we compare the results obtained with an Nvidia Tesla C1060 board to a sequential implementation on 2 kinds of CPU (Xeon Core 2 and Nehalem) (Part 4).

## 2 Accelerate cellular automata computation

### 2.1 3D Cellular automata

A simple heart model can be created using a cellular automaton. After selecting an appropriate neighborhood (a Moore cubic-shaped neighborhood with a range of one for example), we can simulate the propagation of an electric stimulation with 3 states:

- A cell is in an "activated" state for an amount of predefined time. In this state, this cell stimulates its neighborhood cells which are in an "inactive" state. This state corresponds to cell depolarization.
- A cell is in a "refractory" state for a certain predefined time. It can't be stimulated by an "activated" cell. This state corresponds to cell repolarization.
- The rest of the time, a cell is in an "inactive" state.

[4] Four-Dimensional Cellular Automata Acceleration Using GPGPU: http://eric_rollins.home.mindspring.com/gpgpu/index.html. Accession date: 20/01/2010.

[5] What is CUDA ? - Official CUDA web site: http://www.nvidia.co.uk/object/cuda_what_is_uk.html. Accession date: 20/01/2010.

[6] OpenCL Overview - Official OpenCL web site from Khronos : http://www.khronos.org/opencl/. Accession date: 20/01/2010.

But this heart model is far from perfect. A critical point is that the number of cells used will directly affect the model accuracy. If the whole heart is represented by a $10^3$ cells cube, the accuracy will be low compared to the use of a cube with $100^3$ cells. To compare both possibilities, a ventricle will be represented by a cube of $3^3$ cells in the first case whereas in the second case we can use $30^3$ cells (27 cells compared to 27000). In the second case the shape of the ventricle 3D model will be close to a real one. Getting a better accuracy cannot be reached only by a growth in cube size. Each time each dimension grows by a factor of 10, the total number of cells grows by a factor of 1000 (1000 cells compared to 1.000.000 cells in the second case with a $100^3$ cube). Having more cells implies more computations and this significantly slows down the simulation performances. In addition, even if we assume that a $100^3$ cells cube is a good dimension to represent a heart and that we have enough computing, it could be interesting to have a more representative model of the system. These improvements generally require a computation increase. For example, the state corresponding to the cell repolarization ("refractory") can be subdivided into two parts: a first one called the "absolute refractory state", where the cell can't be reactivated by an incoming impulse and a second, called the "relative refractory state" where it can. This small modification will only slightly increase computing power but more complex modifications can be considered with a wider impact on computing time.
One way or another, if the modeling technique retained is based on cellular automata, increasing the computation speed could be a good way to improve the model representation without damaging the global simulation speed and therefore achieve better accuracy.

### 2.2 GP-GPU utilization

In order to accelerate the computing time of 3D cellular automata simulation, we have tried a parallelization using a GP-GPU (General Purpose Graphical Processing Unit). GP-GPU is particularly powerful when used to process the same instruction set on an important number of independent data. This approach can be useful for cellular automata evolution if we consider the splitting of the cellular space in sub-cubes. However, neighborhood interactions are going to be a weighty problem for the program. To compute an iteration on a sub-cube, we need a fast access to cells' neighborhood, otherwise the new state of border cells can't be computed. In the next figure, the dark grey sub-cube will need the current state of every black cells present in other sub-cubes to compute the next state (a non-toric cube representation with a Moore cubic-shaped neighborhood with a range of one is used in this figure).

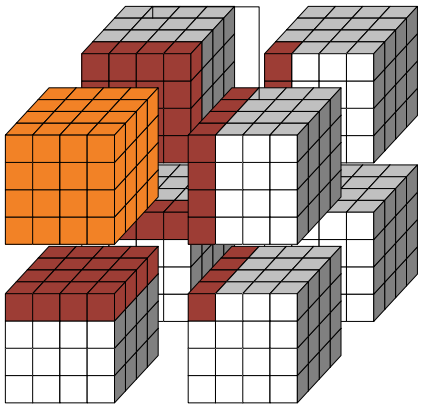

**Figure 1:** Example of Moore cubic-shaped neighborhood in red with a range of one.

### 2.3 Problems induced by the current GP-GPU architectures

To correctly understand the consequences while programming with GP-GPU, it is important to realize that GP-GPU design is quite different from current CPU architectures. While a CPU possesses few cores, each of them allowing the execution of one thread at a time, a GP-GPU possesses a small number of streaming multiprocessors, each of them allowing the parallel execution of numerous threads, supporting vector computing in a SIMD approach (Single Instruction Multiple Data). For instance an Nvidia Tesla 10 (see **Figure 2**), will have 240 vector cores split in 30 streaming multiprocessors (SM) with 8 thread processors (SP Thread

Processors) each[7]. Each thread processor can run four SIMD warps of 32 threads with the same control flow for different data leading to potentially 1024 threads on each of the thirty streaming multiprocessor.

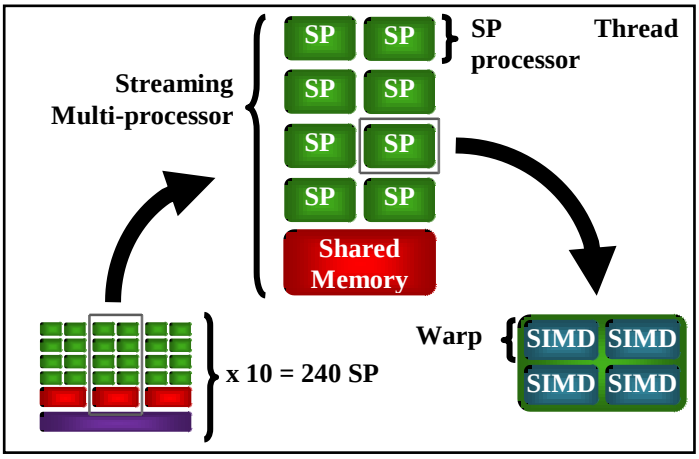


**Figure 2:** Simplified architecture of an Nvidia Tesla 10.

With CUDA, all the threads needed to execute a kernel must be grouped in blocks and all these blocks must have the same, limited number of threads. All the threads of a block are executed simultaneously on the same multiprocessors and therefore can make use of its shared memory. To get the best results from GP-GPUs, we have to place data in shared memory. This memory is fast and managed by a streaming multiprocessor. The latency for accessing global memory is very high and we have to limit its access. Thus, dependant threads needing fast communications have to use the shared memory within the same streaming multiprocessor (SM).
When mapping cellular automata to a GP-GPU architecture we will not be able to place all the threads on a single SM, and consequently we will have to communicate using more global memory accesses, implying a significant loss in performance. The difficulty lies in successfully managing memory copy from global to shared memory. Another important consideration is the memory transfer between the host and the GP-GPU which can be very significant if the following ratio: computation duration divided by the transfer duration, is too small.

## 3 GP-GPU Implementation

### 3.1 First implementation

The first implementation we tested doesn't use shared memory. Its goal is to use the GP-GPU computation capability to see if we can already obtain a speed-up of the global computation with a basic approach.

To make use of this GP-GPU computation capacity, an elementary thread is used to compute the new state of each cell. The algorithm is then kept simple and an excerpt of the code is presented below.

```
On CPU:
  // Initialize the cell grid.
  int    * matrice, * matriceGPU;

  initGrid( &matrice, size);
  cudaMalloc( ( void ** ) &matriceGPU,
              size * sizeof( int ) );

  // Copy the cell grid from the CPU to the GPU.
  cudaMemcpy(
              matriceGPU,
              matrice,
              size * sizeof( int ),
              cudaMemcpyHostToDevice
            );

  // For each iteration needed:
  for ( i = 0; i < nbIte; ++i )
```

[7] CUDA Programming Guide v2.3 - NVIDIA_CUDA_Programming_Guide_2.3.pdf: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf. Accession date: 20/01/2010.

```
      // Run the GPU kernel to compute iteration with
      // one thread for each cell.
      nextGeneration<<< nbBlock, nbThread >>>( … );

    //Copy back the cell grid.
    cudaMemcpy(
                matrice,
                matriceGPU,
                size * sizeof( int ),
                cudaMemcpyHostToDevice
              );

  On GPU kernel:
    // Compute the global thread index.
    threadIdx = threadIdx.x + blockIdx.x * blockDim.x;

    // Compute the new state of the cell corresponding
    // at this index in function of its neighborhood states.
    […]
```

**Code 1**: Partial code of the basic algorithm.

In this version, we first copy the cell cube on the GP-GPU. Then, we run the kernel using a thread for each cube cell. Each thread can compute the new state of the cell it handles depending on the neighborhood state. Consequently, at the end of the last thread execution, all the cells are updated for an iteration. When all the iterations have been computed, we transfer the resulting cube back onto the CPU.

This very simple algorithm has two major limiting operations. The first one is logically the copy of the initial cube from the CPU to the GP-GPU and the result cube from GPU to CPU. The second limit of this basic algorithm is almost masked in the previous code: it deals with memory access to the cells and their neighborhood.

For the first point, copying time costs will be very difficult to improve. A solution could be to take advantage of asynchronous memory transfer by dividing computation in multiple parts for example. But this solution would significantly affect the algorithm simplicity and reusability, and moreover it will not allow a significant global speed-up in most cases, particularly when we have a large number of iterations to compute. The more the iteration number will be important, the less the copy time costs will be significant in comparison with computing cost for all iterations.

The second point is related to global memory access. When we perform the Game of Life in three dimensions, it is necessary to know the neighborhood of each cell. This leads to a maximum of 27 cell states if we consider the current cell state. In this first version, all memory access is done in global memory with a very high access latency (between 400 and 600 clock cycles on a Tesla T10 for instance).

If we consider the handling of processing 2 cells at a time, we can notice that 2 neighboring cells share 18 cells in their common neighborhood. Thus we can obtain all their neighborhood with only 36 global memory access instead of 54. And results are even better if we increase the block size to small cubes: a $3^3$ cube (27 cells) will need only 125 global memory access (the block including neighboring cells is a $5^3$ cube) instead of accessing 729 cells if we consider the handling on a single cell basis (27 x 27). The following implementation tries to take advantage of this possibility.

### 3.2 Second implementation

To limit at minimum the global memory access, the code hereafter makes use of shared memory between threads of a same CUDA software block. Since the size of a software block is limited, it is impossible to share all the cell information between all the threads needed to handle a large cube. Each software block works on a part of the cube and each thread inside this block works on the generation of the new state of a single cell. Before computing an iteration, data required for a CUDA software b(lock must be loaded from global memory into shared memory. This is done in every software block. When the uploading is done, a synchronization of all the threads of a same block is necessary to ensure that all the data is correctly uploaded in shared memory. After this synchronization, the computing of the new state can be done in function of neighborhood states read in shared memory instead of global memory. The new state obtained for the local cube is written in global memory otherwise it would be lost after the end of the block computation.

```
  On CPU:
     […]
     int  neighborSize = […];
     int  sharedSize   = ( size / nbThread + neighborSize ) *
                         sizeof ( int );
     […]
     // Run the GPU kernel to compute iteration with
```

```
    // one thread for each cell.
    nextGeneration<<< nbBlock, nbThread, sharedSize >>>(…);
    […]

On GPU kernel:
    // Upload from global memory to shared memory
    int   sharedMatrice[];
    sharedMatrice[ threadIdx.x ]   =   matriceGPU[ … ];

    __syncthreads();

    // Compute the new state of the cell corresponding at
    // this index in function of its neighborhood states
    // present in shared memory.
    […]
```

**Code 2:** Pseudo-code of the shared algorithm.

In this solution, the aim is the reduction of the global memory access but nothing has been done for the memory transfer between CPU and GP-GPU.
With this approach we still have global memory access to load the shared memory and to update the global memory after an iteration. Even if there is a diminution in terms of global memory access, this approach results in an increase in the total number of memory access (global and shared) but most of them are shared memory with a final increase in performance.
To maximize the number of cells shared in a same block, the largest block possible has been used. Each block contains 512 threads, allowing the processing of an $8^3$ size cube. To limit each thread access to the global memory, the state of each cube cell is loaded in shared memory. This choice implies that all threads handling the cells at the border of the $8^3$ cube are not used to compute the next state. To obtain the next iteration state, only the threads processing the inner cube of $6^3$ cells are considered.

## 4 Results

The two algorithms previously presented have been tested on personal computers, the main one using two Intel Nehalem CPUs (at 2.53 Ghz) with an NVidia Tesla C1060 board. To compare the benefits of hybrid computing, the CPU only algorithm computes for each cell the new state with a single core (a Core 2 Xeon and a Nehalem Xeon have been considered).

### 4.1 Memory transfer costs

Figure 3 confirms what has been said previously when explaining the basic algorithm: transfer costs between CPU and GPU are not significant anymore when we compute many iterations. In fact, for a constant cube size, the global simulation duration should be linear with the number of iterations; to this global computing time we naturally add two memory transfer costs for copying the initial cube from the CPU memory to the GPU global memory and for copying back the final result. In Figure 3, with a small $10^3$ cube, the impact of this transfer time is important if we have a small number of iterations and becomes insignificant after 400 iterations.

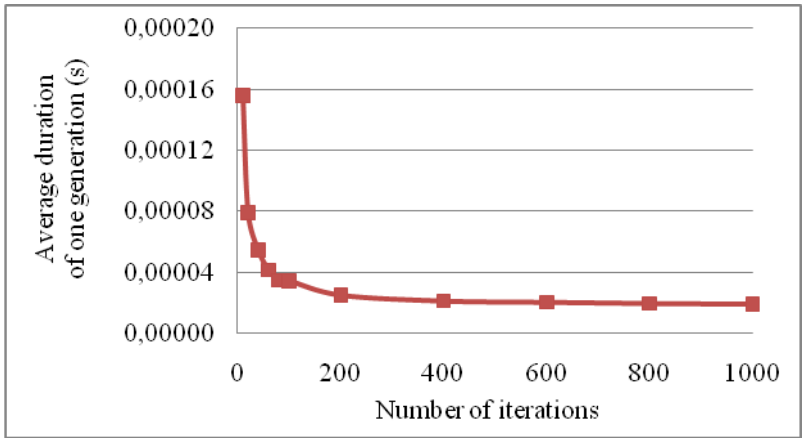


**Figure 3**: Average duration (in seconds) of one generation for a cube with $10^3$ cells.

The next figure shows the obvious importance of the cube size. For big cubes the transfer time measured became rapidly insignificant even with a small number of iterations. Now that the memory transfer problem has been analyzed, it is possible to start comparing CPU and GPU results.


**Figure 4**: Average time for one iteration (100 replicates) for the following 3 tasks: allocation of 2 cubes (cube size), copy of a cube from CPU to GPU and copy of the final resulting cube (from GPU to CPU)

### 4.2 Basic GPU implementation results

The first results deal with the basic implementation which uses only global memory.

Figure 5 shows a comparison between the pure CPU algorithm (Core2 Q9300 at 2.5GHz) and its hybrid implementation with a GP-GPU for a $20^3$ cube (Tesla T10 – C1060). We see that the best results are obtained with a significant number of iterations. The performance improvement stops after reaching a threshold.
In Figure 5, we see that the sequential implementation using the CPU is always slower for a $20^3$ cube when compared to the GP-GPU implementation.

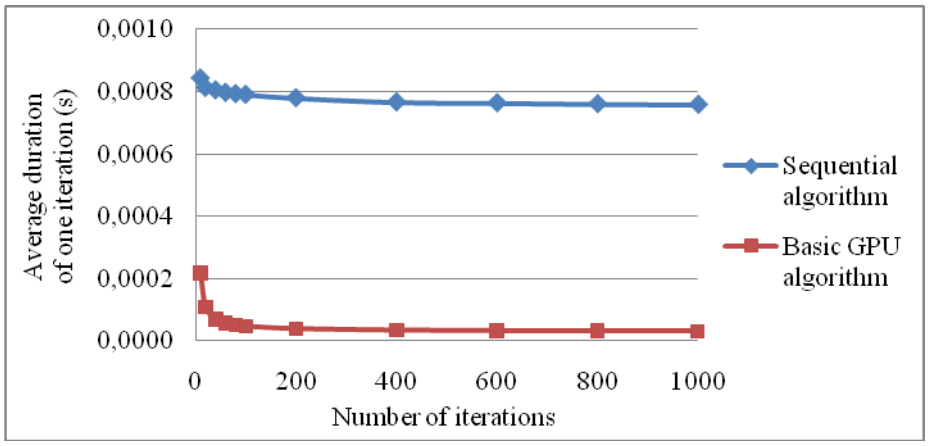


**Figure 5**: Average duration of an iteration depending on the number of iterations for a $20^3$ cube.

In Figure 6, we notice that when using a GP-GPU, the computing time of an iteration grows slowly with the cube size, whereas the computing time for an iteration according to the cube size shows a polynomial growth if we use the sequential CPU algorithm.

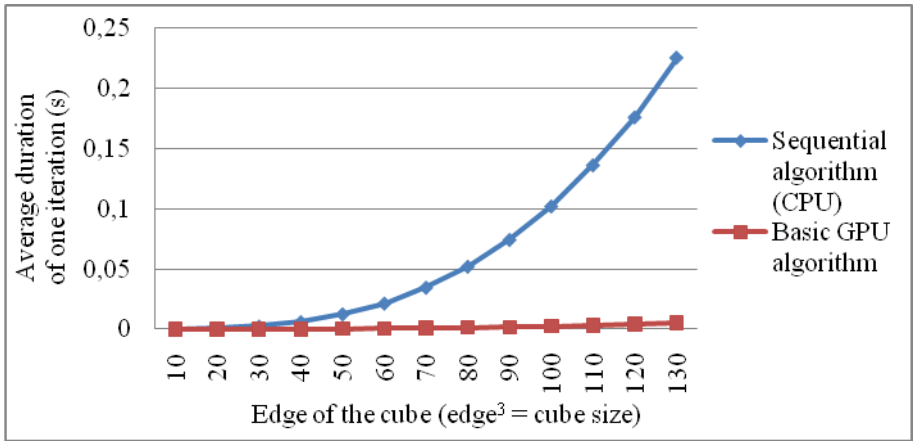


**Figure 6**: Average duration of one generation for 1000 iterations depending on the cube size.

Figure 7 presents the speed-up for different cube size depending on the number of iterations. For each cube size we have an upper limit; we also note that the speed-up first increases with cube size and stays approximately at the same level when the cube size reaches $80^3$ and over.
This last point can be seen more clearly on Figure 8 which represents the speed-up for 1000 iterations depending on the cube size.

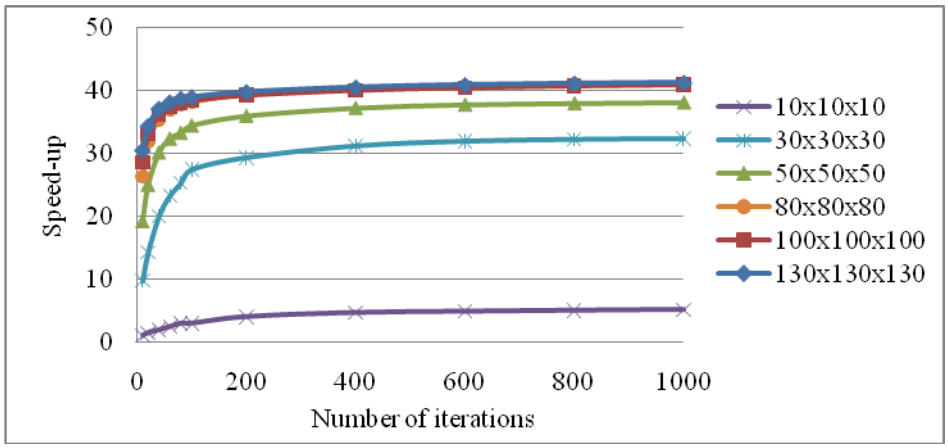


**Figure 7**: Basic GPU implementation speed-up compared to a sequential CPU implementation for different cube sizes depending on the number of iteration.

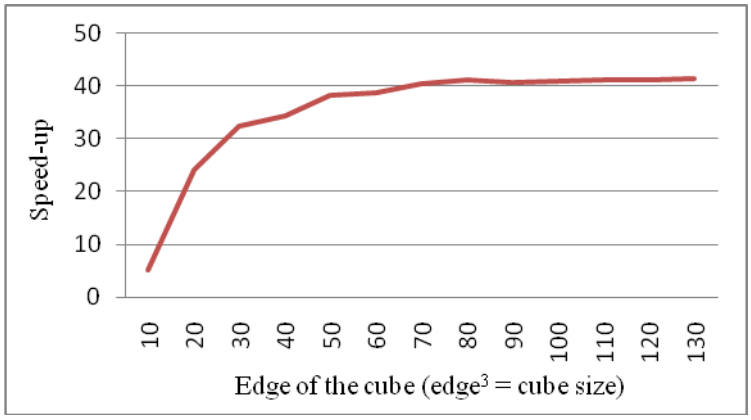


**Figure 8**: Speed-up evolution for 1000 iterations depending on the cube sizes (50 measures for each point).

Now that the speed-up trend has been identified, we can discuss speed-up results. If we ignore cube sizes smaller than $80^3$, the average speed-up for 1000 iterations is 41x when compared to a single Q9300.

### 4.3 GPU implementation with shared memory results

The next implementation takes advantage of shared memory between threads of the same streaming multiprocessor to minimize global memory access.

Figure 9 shows the performance increase compared to the first implementation (using a $100^3$ cube). It confirms that the two trends are really similar and also that the shared memory algorithm is better than the previous one.

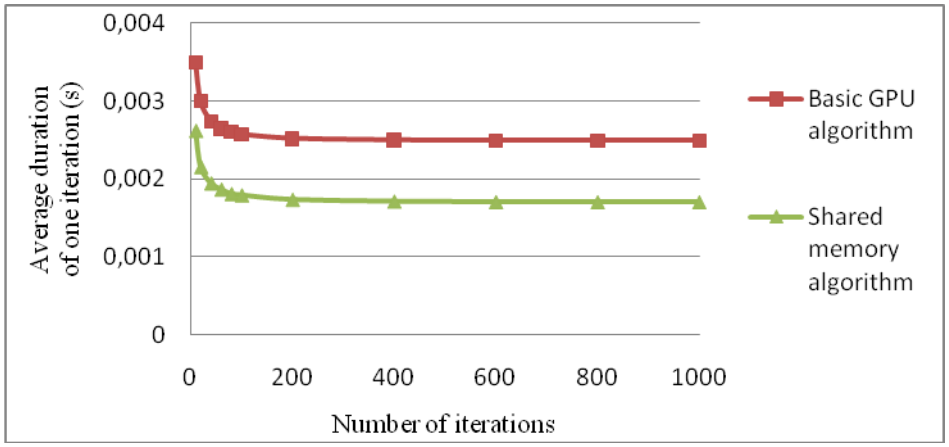


**Figure 9**: Average duration of an iteration depending on the number of iterations for a $100^3$ cube (50 replicates).

Figure 10 compares the speed-up for 1000 iterations with both implementations. Speed-ups are very similar for small cubes but as expected the shared memory algorithm has better speed-up for larger cubes (reaching 60x compared to 40x). With this algorithm the upper limit is reached later than with the previous algorithm (see Figure 11).

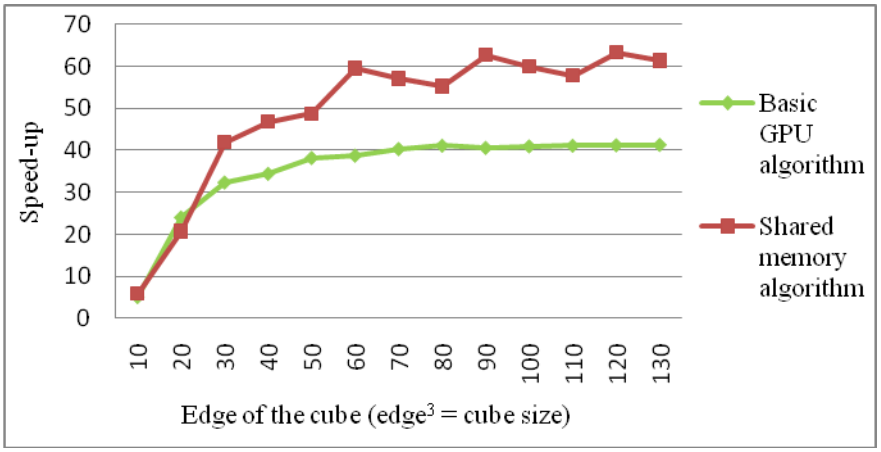


**Figure 10**: Comparison of speed-up evolution for 1000 iterations depending on the cube sizes (50 measures for each point).

These results can be generalized for larger cubes. Figure 12 shows speed-up evolution for 1000 iterations comparing the CPU algorithm for two different processors: the previous Core2 Q9300 but also a Nehalem 5500. The first point we can notice is that the speed-up did not slow-down with the increase in cube size (the $800^3$ cube size limit will be discussed in conclusion). The second point is that speed-up obtained with the GPU is far better when we compare it with a Q9300 Xeon processor (average speed-up of 62.8x) than in comparison with a Nehalem 5500 processor (average speed-up of 20.5x). This difference shows an impressive advance in CPU performances, at almost the same frequency, particularly if we consider that the Xeon Q9300 was still considered as a very good Intel processor at the beginning of year 2009.

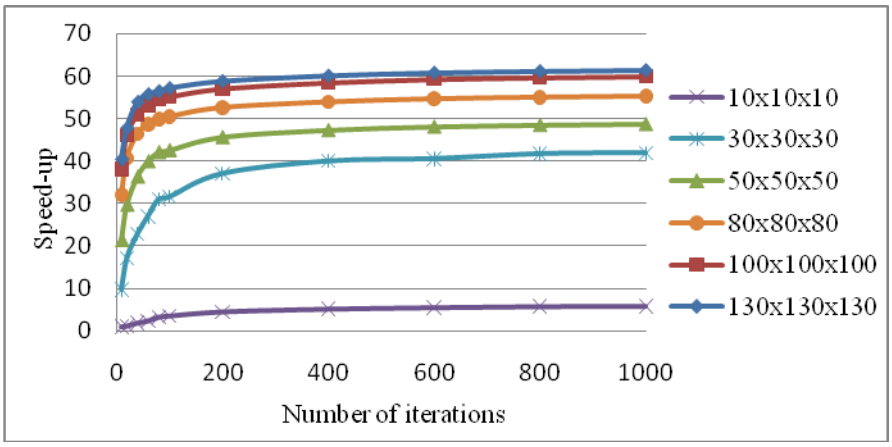


**Figure 11**: Shared memory GPU implementation speed-up for different cube sizes depending on the number of iterations.

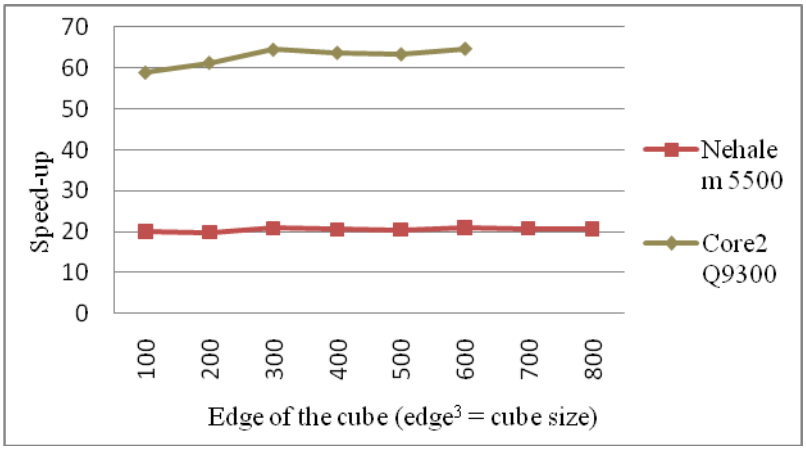


**Figure 12**: Speed-up evolution on large cube for 1000 iterations depending on the cube sizes.

## 5 Future research

In the case of 3D cellular automata, whatever the implementation, basic or more sophisticated, the use of hybrid computing shows interesting speed up. But improving this solution is still possible. Here are some possibilities which it could be interesting to explore.

First, it is possible to have all the threads of a block to work on computing the next status of a cell. Currently, all the threads corresponding to the block border are only used to transfer neighborhood status from global to shared memory. To make them work, it is necessary to make them load multiple cell status from global to shared memory. This solution, which has not been tested yet, does not give any guarantee of speed-up and it probably needs to be tested on different automata configurations.

It would also be interesting to see if using another block size can improve the performance. At the moment a block of 512 threads is used to represent $8^3$ cubes. But it could be interesting to test blocks of 256 threads to represent $6^3$ cubes.

The second implementation was proposed to improve the memory access of the first version (which used only global memory). The fastest implementation uses mainly shared memory but it still needs to transfer data from the global memory to the shared memory and we think that this transfer could be done faster. In my implementation, each block of threads corresponds to a block of the cube cells in the example of Figure 13, the 125 threads of a block access 125 cube cells in global memory (the part framed in orange). When each thread accesses the state of a cell, it's clear that the global memory index will not be continuously accessed and with the CUDA architecture, we know that memory access can be done more efficiently by reading continuous data.

A solution could be to reorganize the cube data to allow continuous global memory access for all the threads in a same block. This better memory access, implies that more computation would have to be done on each thread to compute the correct index in the cube but it would be interesting to see if it improves the performance. However this kind of dedicated implementation becomes quite obfuscated and difficult to maintain.

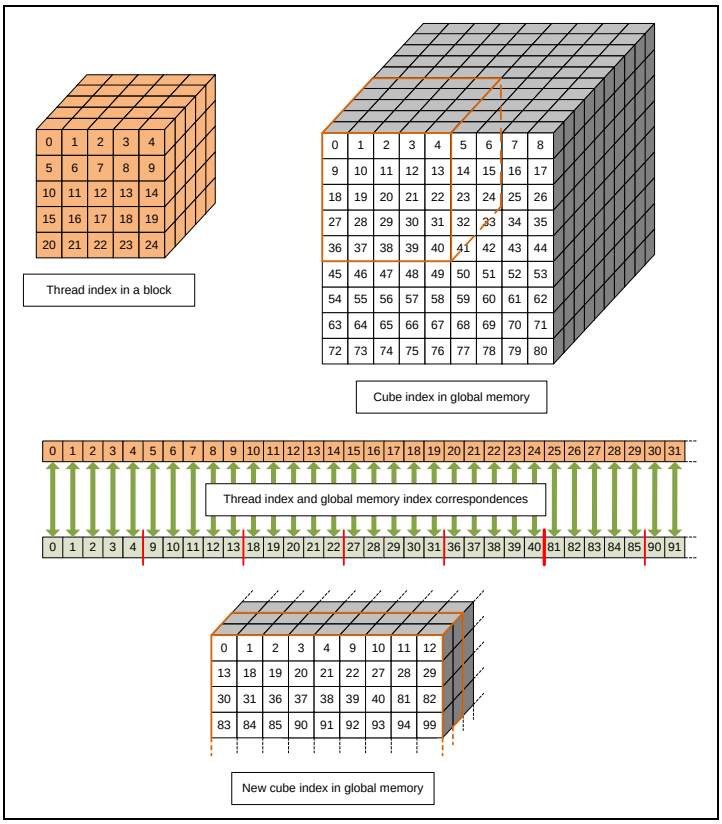


**Figure 13**: Improvement of data storage to increase global memory access.

Another solution to improve the results could be to create specific generation algorithm for each cellular automata algorithm by taking advantage of their particularities. For example, in the case of excitable medium, there are generally some important cube parts where no activities are present. It could be possible to locate these parts and therefore not do any computation because these parts won't evolve. But there is an important problem with this solution. It's logically the development cost. If many cellular automata algorithms are used, developing for each one a specific generation algorithm which takes advantage of its particularities could be an enormous task. And even if the number of cellular automata algorithms is limited, specific processing must be fast enough to not damage the global computation time. In the previous example, if searching for a large empty cube part is longer than computing a new generation on this part, performance increase can't be guaranteed.

## 6 Conclusion

We have implemented two cellular automata algorithms on a Tesla C1060 board. The basic principle is the same for both algorithms, a GPU thread is used to compute the new state of a cell. But to compute a new cell state, a thread has to read the states of the 27 neighbouring cells (if we consider a classic Moore neighborhood extended to a 3D automaton). The total number of memory accesses can imply a significant access time when relying on the GP-GPU global memory. A second implementation has been created to minimize the number of global memory accesses by using the fast shared memory present on a GP-GPU streaming multiprocessor. The states of all the cells needed by all the threads running on a same multiprocessor are transferred only once from global memory to shared memory. This is sufficient to compute a significant set of cells in parallel with efficient shared memory accesses.

We have tested both approaches and the results show that we have an interesting speed-up for any size of cube when enough iterations are computed. On small cubes, when a little number of iterations is considered, the transfer costs between the host and the GPU make the whole process significantly slower. When large cubes are considered ($100^3$ and above), the average speed-up for the first implementation compared to a sequential algorithm running on a Xeon Core 2 CPU at 2.5 GHz is around 41x (13x when compared to a Nehalem Xeon at 2.53 Ghz). If shared memory is used (second implementation), the speed-up grows to 65x when compared to a Xeon Core 2 CPU (and 20x a Nehalem Xeon).
The results observed show that even with the best CPUs such as Intel Nehalem, we obtain significant speedup. However, we have to consider the following limiting points. First, the CPU algorithm used for comparison with the GP-GPU algorithm is a sequential one. Now, most of the computers have multiple cores and taking the best

of all of them will significantly increase the regular host performance. This point is not so important in integrative biology because cellular automata are only a part of the whole computation and the computation power of host cores will be used for other computations which have lower speed-up while computing on GP-GPU.

The second point is the cube size. We have to take into account the current memory limit for Tesla GP-GPUs (4 GB). To compute a new iteration on a GP-GPU, we use two cubes, one handling the cells current statuses and the other to write the new statuses. A Tesla C1060 has 4GB of global memory, hence the size of the cube can't be too important. Two cubes with $900^3$ cells can't be processed on this kind of GPU because it would need more than 5.8GB if 4 bytes are used to store a cell status ($1000^3$ can be considered if we store cell states on a single byte).

The next Telsa architecture (Fermi T20) will significantly improve memory access performances with ECC memory, a configurable cache and it will also be able to address up to 1TB of memory. With this kind of architecture, a cube with $5000^3$ cells will be handled on a single GP-GPU board, enabling a fine multi-scale modeling for simulated organs. Grids and clusters of GP-GPU will be considered to simulate interactions between organs [14].

## 7 References


[1] Shaoying Lu, Anushka P. Michailova, Jeffrey J. Sauverman, Yuhui Cheng, Zeyun Yu, Timothy H. Kaiser, Wilfred W. Li, Randolph I. Bank, Michael J. Holst, J. Andrew McCammon, Takeharu Hayashi, Masahiko Hoshijima, Peter Arzberger, and Andrew D. McCulloch. Multiscale Modeling in Rodent Ventricular Myocytes. IEEE Engineering in Medicine and Biology Magazine. March-April 2009; Volume 28 - Number 2; pp. 46-57.

[2] Igor V. Pivkin, Peter D. Richardson and George Em Karniadakis. Effect of Red Blood Cells on Platelet Aggregation.IEEE Engineering in Medicine and Biology Magazine. March-April 2009; Volume 28 - Number 2; pp. 32-37.

[3] Edward A. Sander, Triantafylllos Strylianopoulos, Robert T. Tranquillo and Victor H. Barocas. Image-Bases Biomechanics of Collagen-Based Tissus Equivalents.IEEE Engineering in Medicine and Biology Magazine. May-June 2009; Volume 28 - Number 3; pp. 10-18.

[4] John von Neumann, AW Burks, 1966, The Theory of Self-reproducing Automata, Univ. of Illinois Press, Urbana, Illinois.

[5] T. Alarcón, H. M. Byrne, P. K. Maini: Towards whole-organ modelling of tumour growth. Progress in Biophysics and Molecular Biology 2004, 85:451-472.

[6] P. Siregar, J. P. Sinteff, N. Julen and P. Le Beux, An Interactive 3D Anisotropic Cellular Automata Model of the Heart, Computers and Biomedical Research, Volume 31, Issue 5, October 1998, Pages 323-347.

[7] Martin Gardner, (October 1970), "Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life"", Scientific American 223: 120–123.

[8] Wolfram Stephen, A new kind of Science, Wolfram Media, Inc, ISBN 1-57955-008-8, 2002.

[9] A. K. Dewdney. The cellular automata programs that create wireworld, rugworld and other diversions, Computer Recreations, *Scientific American*, Jan: 146-149. 1990.

[10] G. B. Ermentrout and L. Edelstein-Keshet (1993). Cellular automata approaches to biological modelling. Journal of Theoretival Biology 160, 97–133.

[11] Luebke DL, Harris M., Krüger J., Purcell T., Govindaraju N., Buck I., Wooley C., Lefohn A., GPGPU : General purpose computation on graphics hardware, In SIGGRAPH'04, Proceedings of the SIGRAPH 2004 conference, course notes, New York, NY, USA, ACM Press (2004), doi:10.1145/1103900.1103933.

[12] C. Trendall and A.J. Stewart , "General calculations using graphics hardware with application to interactive caustics," In Rendering Techniques '00 (Proc. Eurographics Rendering Workshop), pp. 287-298. Springer, June 2000.

[13] Woundeberg Mike, Using FPGAs to Speed Up Cellular Automata Computations, Master's thesis, University of Amsterdam, 2006.

[14] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for high performance computing. In SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, page 47, Washington, DC, USA, 2004. IEEE Computer Society.