Simulation et modélisation de diffusions physiques
Rapport de TER

AHMED Ahmed
Master 1 Informatique
Université de Bretagne Occidentale
Encadrement B.Pottier, LabSTICC

August 8, 2013

Contents

Acknowledgements

During the ten months I have been at the university of Brest and during the TER, I have been influenced by many people around me, who have shaped the way I think and taught me a lot about how to do research. Among those, I want to first thank Prof. Bernard Pottier for providing me with guidance and support, and always the encouragement to try something new and different. Special thanks to Mr.Pierre Yves Lucas for his help in diplaying the map and integeration with the simulation. Thanks for all the teaching stuff members in the IT departement for the their help and support specially in the language.Lastly, I would like to thank Master M1 students with whom I have a good chance to work and exchange experience.

Introduction

The environment around us has many phenomena and has different behaviors according to different parameters, biological, chemical, physical, etc. To represent a simple and abstract reality of this environment we use a concept called environmental modeling. The environmental modeling deals with many environmental problems such as air pollution, diffusion of disease, animal behavior and so on. However there are some difficulties in modeling the environment due to several reasons such as the variation and incompatibility.

There are many techniques to simulate a model such as differential equations, cellular automata and multi-agent approach. We will focus on our work on the cellular automata, that is dynamic model which is spatially and temporally spate. The cellular automate update their state with time, where each cell has a new state based on its state and the state of its neighbors. The update is based also on some rules defined previously. When simulating any model, the most important problem that faces us is the execution time.

In many environmental problems we will work on a model that may cover a big geographical area and thus we require a large numbers of cells to represent that system. For these cells to update their state we will need a long execution time which may exceed days in some models. According to Florent Arrignon, a specialist in environmental modeling, in some models of one million cells, it requires about one week to simulate it. But fortunately due to the increased performance in the computers and more precisely on the GPU, we can simulate a cellular automata environmental model in parallel which will result in enhanced performance result.

Besides, there are different approaches to deal with large blocks of data such as OpenCL and CUDA. In our case study we are addressing a general problem of Forest Fire Spread using CUDA language and discuss different approaches to enhance the performance.

Road Map

Chapter 1 provides a brief introduction on environmental modeling, its features and techniques. Moreover, it we will inroduce the cellular automata, the different approaches to deal with the cell neighbors and examples of some application of cellular automata. Chapter 3 will discuss the different programing approaches on processing large block of data ion parallel and will explain in details the CUDA language. In chapter 4, we will begin a case study environmental model of Forest Fire Spread with the implementation a parallel model and discussed it in terms of performance. Finally chapter 5 a more enhanced pipleine parallel version will be implemented. After that we have the conclusion, future work, bibliography and lastly but not least the appendix which include all the code.

Chapter 1

Environment modeling and Cellular Automata

1.1 Environment modeling

Modeling represents a simple and abstract reality. It allows us to understand a particular feature of the world as we build textual, graphical and other models of reality. With the increased performance of digital computers we can handle complexity in our models by developing well elaborated software. Modeling has great importance as it allows us to understand many aspects around by using data and mathematical representation of any thing around us, any object, person, air pollution, transportation and so on. Modeling can be viewed from many perspectives but we will focus on environmental modeling.

Environmental modeling is used to model environmental phenomena such as biological, physical and chemical phenomena by use of computers and mathematics. There exist many examples, such as plants pollution, air pollutions, fire diffusions, renewable resources management (water, tree, wood). Furthermore environment modeling is used to examine broad range of environmental issues such as the global warming caused by the industrial, agricultural practices and certain waste management. Many discipline mostly chemistry, physics, biology, biochemistry, geology and others contributes in the understanding these models.

According to [1], there exist different types of modeling approach which requires an understanding of the purpose of modeling different environmental systems. The types of modeling approach used are affected by a number of features of environmental systems.

1.1.1 Environmental Systems Features

There are many physical, biological and chemical environmental systems that share common features between them. As stated by [1] we can divide and describe these features to:

1.1.1.1 Complex Non Linear Interactions

The environmental systems consist of complex nonlinear interaction between many processes, physical, biological, chemical, social and economic. For example, water quality in

a stream relies upon land use and management in a catchment. However, decisions on land use will depend on land suitability, which is a function of rainfall, temperature, topography, and soils, as well as on economic, social and cultural features of the individual catchment. The interaction between these different processes is complex and often poorly understood. This complexity can mean that it may be difficult to capture many of the underlying causes and effects of environmental phenomena. Thus when modeling simplifying assumptions about the way in which these factors interact must generally be made.

1.1.1.2 Heterogeneity of System Features

Many scales are there to measure the characteristics of the environmental system. These scales differ spatially and temporally. The spatial scales may be very small in nanometers to very large scales in multiples of kilometers as shown in figure 1.1

Figure 1.1: Different Spatial Scales

Furthermore, the temporal scales may vary from part of milliseconds to hours, days and years as shown in figure 1.2

Figure 1.2: Different Temporal Scales

These features are difficult to characterize and leads to many errors and uncertainty, mostly when the observed data are dynamic and the frequency at which data is sampled is inadequate.

1.1.1.3 Incompatible Scales

It is difficult to create generic environmental models due to the variation of the temporal and spatial characteristics.“For example, the system response to a rainfall event in a surface water system such as a river or stream may occur over hours or clays, while the response of the groundwater system to this recharge event may occur over a number of years, even though these two systems are linked. The characteristic temporal scales of these two systems are very different, so that a model linking these two systems will have to find a compromise between these two scales” [1]. Thus scales are very important feature in the modeling of the system.

1.1.1.4 Inaccessible or Unobservable System Processes

Some system scales are too small or too large to be measured or observed, such as some features of oceanic systems which cover thousands of kilometers. Thus, due to the difficulty to acquire exact measurements and observation of the process, the understanding of the model is approximated and hence affect the accuracy of the model.

However, nowadays due to the revolution of information technology, telecommunication and electronics, there are a variety of tools - wireless sensors, water proof sensors, remote sensors, and climate sensors, high detection sound sensors - that help modelers to achieve their goals with approximately accurate results.

1.1.2 Modeling Techniques

There are known techniques to model systems. We have the Differential equation approach which is widely used, but is suitable only for systems with limited numbers of parameters and small scales. Furthermore, there exists two other approaches Cellular Automata and Agent-based models, used for more number of parameters and to describe spatially heterogeneous systems. In this project we will focus on cellular automata which are introduced in next section.

1.2 Cellular Automata

Cellular Automata mostly abbreviated "CA"is a discrete dynamic model which is temporally and spatially discrete. Spatially discrete, it is composed of a number of cells placed on a regular grid where each cell is described by a state chosen from a finite set. Temporally discrete, at each time the cell changes its state based on a transition function which takes into account the state of the local neighbors of the cell. CA are able to solve many algorithmic problems, so they can be considered as computational systems [2].

1.2.1 Neighborhood

There exist many shapes for the CA, the simplest forms being the one dimensional and two dimensional grids. In the one dimensional elementary CA, the neighborhoods are the left and right cells. Whereas in the two dimensional model, we have several types for neighborhood, some of them are *Moore neighborhood* and *Von-Neumann neighborhood*.

In *Moore neighborhood*, the neighborhoods of a cell are all the cells surrounding it. It includes the north **N**, south **S**, west **W**, east **E**, north-west **NW**, north-east **NE**, south-west **SW** and south-east **SE** cells. Figure 1.3 below represents the Moore neighbors of the black cell.

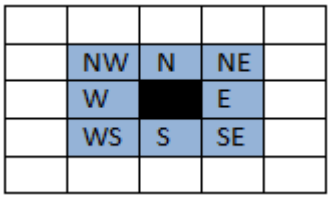

Figure 1.3: Moore neighborhood

Since we have eight neighbor cells and one center cell, then we have nine cells in total which produce 2^9 pattern based on the rules of the transition function.

Instead, the *Von-Neumann neighborhoods* are the north **N**, south **S**, east **E** and west **W** adjacent cells. Thus, we have 2^5 patterns for the states. Figure 1.4 represents the Von-Neumann neighborhood.

There are some models using a specific neighborhood structure, for example in fluvial dynamics models presented by [3], the structure is of 3 to 11 or more cells in one direction.

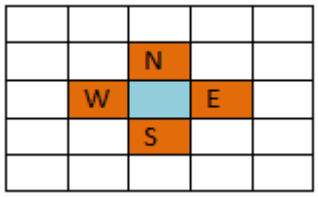

Figure 1.4: Von-Neumann neighborhoods

One of the obvious problems is the cells in the edge. There are different approaches to deal with it, but we will consider one approach that we will use in our project. We will consider a circular approach which can be more clarified by an example. In figure 1.5, the neighbors of the left edge red cells are shown for both the Moore and Von-Neumann types.

Figure 1.5: Circular neighborhoods example 1, left figure:Moore neighborhood - right figure: Von-Neumann neighborhoods

Another example to fix the idea is shown in figure 1.6

Figure 1.6: Circular neighborhoods example 2, left figure:Moore neighborhood - right figure: Von-Neumann neighborhoods

After introducing the cellular automata we can summarize its characteristics as follows

1. States —Selected from finite set
2. Transition rule —how the cell state will change based on the state of its neighbors.
3. Neighbors —connection between cells

1.2.2 Application of Cellular Automata

The Cellular Automata are used widely in many applications for sake of describing many phenomena and real problems. A sample set of applications is given below.

- Electric Power Simulation[5], which simulate the flow of power on a power grid.
- Cryptography and Random Number Generation [6], The CA are used to generate random numbers used in some cryptographic algorithms such as Block cipher.

- Implementing Parallel Computers, some cellular automata have the property of universal computation, which means that in principle they can perform arbitrary computations. According to Wolfram [7], this property may be of more than theoretical interest, and might allow cellular automata to form an architectural model for building practical parallel-processing computers [6]
- Modeling and Simulation, Cellular automata are more efficient in modeling many physical, chemical and biological systems due to the similar mechanism between these systems and CA. Thus, cellular automata are especially suitable for modeling any system that is composed of simple components, where the global behavior of the system is dependent upon the behavior and local interactions of the individual components [6]. Some examples of cellular Automata are:
  - Forest Fire Spread
  - Invasion and diffusion of the potato tuber moths [8]
  - Crystallization [6]
  - Urban development [6]
  - Fractal growth of biological organisms [9]
  - Fluvial changes model [3]

We will work on a case study of Forest Fire Spread which will be explained in details in chapter 3, but before that we will introduce in chapter 2 the essentials in CUDA programming.

Chapter 2

CUDA Programming

2.1 Introduction to GPU and GPU Programming

Graphic Processing Units (GPUs) are *massively multithreaded - many core* chips composed of hundreds of cores and thousands of threads. With this large numbers of cores GPU provide the capability to process large blocks of data in the parallel, thus GPUs are widely used to in parallel processing. They are used to implement many complex and challenging problems in modeling and simulation such as climate modeling, diffusion modeling, in finding medical cures for some disease and so on.

There are different companies that produce GPUs, such as NVIDIA, ATI/AMD, INTEL and others, the common standard for general-purpose parallel programming of heterogeneous systems between these GPUs is OpenCL (Open Computing Language) [4]. It offers the software developers the capability to write efficient, portable code for high-performance servers, desktop computer systems and handheld devices using a diverse mix of multi-core CPUs, GPUs, Cell-type architectures and other parallel processors such as DSPs. However there exists another programming language for NVIDIA GPUs. NVIDIA invented CUDA (Compute Unified Device Architecture) for a parallel computing platform and programming model. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU) [10], thus allowing developers and researchers to use this computing power to solve many problems.

CUDA provides a set of extension to C/C++ language which allows programmers to develop parallel algorithms. Both the CPU and the GPU are used for computations, thus CUDA support heterogeneous computation. The sequential code is executed on the CPU whereas the parallel code is executed on the GPU each with its separate memory resources. There are some differences between these languages, which can be summarized according to [12] as summarize in figure 2.1.

In our case study we will be using CUDA, and we will explain its essentials in the next part.

2.2 Programming in CUDA

We will study the essentials of CUDA using an example and explain all its parts. The code is written in .cu file and compiled using nvcc compiler, which can be downloaded from NVIDIA website. The nvcc generates both instructions for host and GPU as well as

CUDA	OPENCL
Advantages:	
Marketed better.	Support many types of processor architectures.
Developer-support in one package.	Completely open standard.
More built-in functions and features.	
Disadvantages:	
Only works on GPUs of NVIDIA.	Supplied by many vendors, thus not provided as one packet or centrally orchestrated.

Figure 2.1: CUDA vs OpenCL

instructions to send data back and forwards between them. We need also to distinguish between these words,

1. **HOST**, it indicates the CPU.
2. **DEVICE**, it indicates the GPU.
3. **KERNELS**, they are C functions, when called are executed N times in parallel by N threads which allow to perform a single task on multiple data which leads to the Single Instruction Multiple Data **(SIMD)** architecture.

A simplified motherboard architecture with many missing details is shown in figure 2.2. The figure is divided in two parts, the left part for the Host and the top right part for the Device. Each of the host and the device has his own separate memory, the device cannot access the host main memory and the host cannot access the device memory. Thus we need to explicitly transfer the data between the two memories through the PCI bus (or PCIExpress variants).

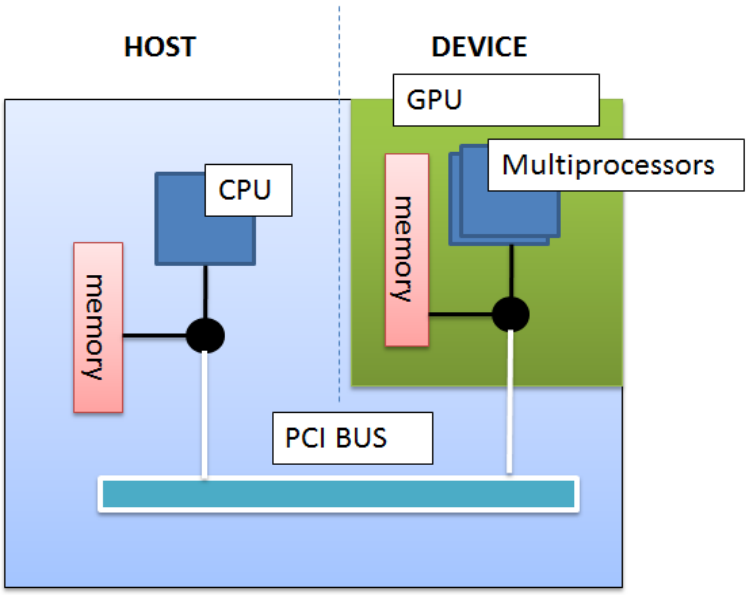

Figure 2.2: Simple motherbard block diagram

In listing 2.1 a simple CUDA program is given to illustrate how CUDA works (this is extracted from NVIDIA CUDA programming guide). The main functionality of this code

is to increment each element of a float array by 1 on the GPU.

Listing 2.1: Increment Array

```
// incrementArray.cu                                                    1
#include <stdio.h>                                                      2
#include <assert.h>                                                     3
#include <cuda.h>                                                       4
void incrementArrayOnHost(float *a, int N)                              5
{                                                                       6
  int i;                                                                7
  for (i=0; i < N; i++) a[i] = a[i]+1.f;                                8
}                                                                       9
                                                                        10
__global__ void incrementArrayOnDevice(float *a, int N)                 11
{                                                                       12
  int idx = blockIdx.x*blockDim.x + threadIdx.x;                        13
  if (idx<N) a[idx] = a[idx]+1.f;                                       14
}                                                                       15
                                                                        16
int main(void)                                                          17
{                                                                       18
  float *a_h, *b_h;           // pointers to host memory                19
  float *a_d;                 // pointer to device memory               20
  int i, N = 10;                                                        21
  size_t size = N*sizeof(float);                                        22
  // allocate arrays on host                                            23
  a_h = (float *)malloc(size);                                          24
  b_h = (float *)malloc(size);                                          25
  // allocate array on device                                           26
  cudaMalloc((void **) &a_d, size);                                     27
  // initialization of host data                                        28
  for (i=0; i<N; i++) a_h[i] = (float)i;                                29
  // copy data from host to device                                      30
  cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);        31
  // do calculation on host                                             32
  incrementArrayOnHost(a_h, N);                                         33
  // do calculation on device:                                          34
  // Part 1 of 2. Compute execution configuration                       35
  int blockSize = 4;                                                    36
  int nBlocks = N/blockSize + (N%blockSize == 0?0:1);                   37
  // Part 2 of 2. Call incrementArrayOnDevice kernel                    38
  incrementArrayOnDevice <<< nBlocks, blockSize >>> (a_d, N);           39
  // Retrieve result from device and store in b_h                       40
  cudaMemcpy(b_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);        41
  // check results                                                      42
  printf("checking␣\n");                                                43
  for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);                         44
  for (i=0; i<N; i++) printf("%f␣%f\n", a_h[i] , b_h[i]);                45
  // cleanup                                                            46
  free(a_h); free(b_h); cudaFree(a_d);                                  47
}                                                                       48
```

Firstly, in order to write and compile CUDA code we need to add its header as shown in line 4. The code is composed of a part that is executed sequentially and a part that executes in parallel invoked from the sequential part.

The lines 5 to 9 define a sequential incremental function which takes as input a pointer to the array to be incremented and the size of the array. This function will be executed on the host. The same functionality of this sequential function can be achieved by the kernel given in lines 11 to 15. So, how we define a kernel? A kernel can be defined by annotating the function name to be executed on parallel by the keyword __**global**__. It indicates that the kernel is to be invoked from the host. The kernel needs an argument which indicates the number of threads to be executed and this is achieved by the parameter N in the parameters list.

Inside the kernel we need to specify the thread Id to be used. There are different hierarchy for threads, for convenience, threadIdx a built in variable is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

There is a limit on the number of threads per block which is currently **1024** for the recent GPUs. However the kernel can be executed by multiple thread blocks which lead to a number of threads equal to number of blocks multiplied by the number of threads per block. Also the blocks are organized into one-dimensional, two-dimensional, or three-dimensional grid of threads. Blocks are accessed through the built-in variable **blockIdx**.

The above example uses a 1D thread hierarchy. To understand how to find the thread id, we need to refer to some variables in the main function. The size of the array is 10 specified by the value of N and the number of threads per blocks is given by the integer blockSize, thus the number of blocks can be found by the following line.

```
int nBlocks = N/blockSize + (N%blockSize == 0?0:1);
```

The part after the plus sign is used to assure that the number of blocks is sufficient for the number of elements, for example, if N and blockSize are 10 and 4 respectively, the number of block will be 2 +1=3. This is shown in figure 2.3

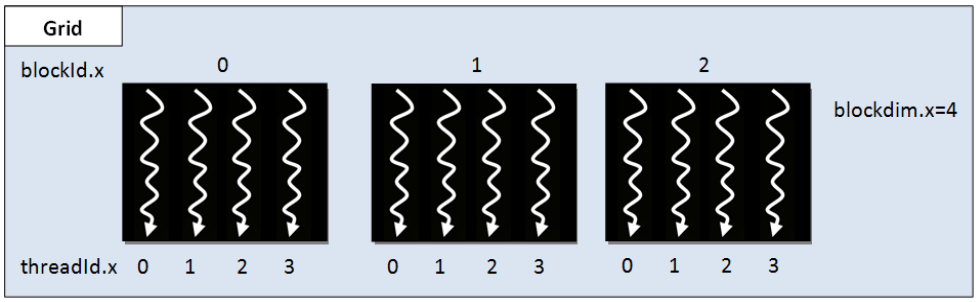

Figure 2.3: 1D Thread hierarchy

Now back to the kernel, to find the thread id, we make us of line 13, thus the values of the thread id (idx) begin from 0. However, we need to assure that not more than N threads are to be executed, this can be achieved using if statement in line 14. The table shown in figure 2.4 below shows the idx (thread id) calculated the input arrays values and the final result in the array.

(idx)	0	1	2	3	4	5	6	7	8	9	11	12
a[idx] input	1	2	3	4	5	6	7	8	9	10	NOT USED	
a[idx] Output	2	3	4	5	6	7	8	9	10	11		

Figure 2.4: Increment array result

After we had explained the kernel we will begin the code of the main function. The line 19 defines pointers to the host memory and allocates them in lines 24 and 25. The pointer **a_h** is used to hold the input to the kernel and **b_h** will hold the result back from the kernel.

The line 20 will declare a pointer to the device memory. The notation _h indicate the host and _d indicate the device. Also we need to allocate memory on the device this is achieved using **cudaMalloc** as given in line 27. Its interface is as follows

```
cudaError_t cudaMalloc(void ** devPtr, size_t size);
```

It allocates size bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory [11]. The memory is now allocated to store the data, so in line 29 we initialize the data on the allocated memory for a_h. After that we copy the data from the main memory to the device memory using cudaMemcpy (line 31), the interface for the function is as follows

```
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum
    cudaMemcpyKind kind);
```

It copies **count** bytes from the memory area pointed to by **src** to the memory area pointed to by **dst**, where kind is one of **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**, or **cudaMemcpyDeviceToDevice**, which specifies the direction of the copy.

The task of sending the data from host to device is illustrated in figure 2.5.

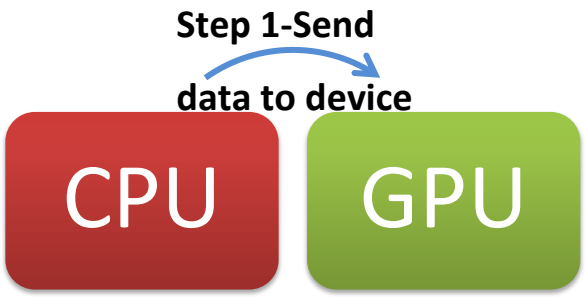

Figure 2.5: Send data to device

Now as the data is in the GPU, we need to execute the kernel (line 39) as shown in figure 3-5, To invoke the method we write the kernel name followed by <<<nb , nt >>>which takes two values, the number of blocks and the number of threads per block. After that it is followed by the arguments of the function.

Figure 2.6: Execute the Kernel

After the execution of the kernel we need to copy the result back from the device to the host as shown in figure 3-6, this can be achieved also by cudaMemcpy utilized with the appropriate direction (line 41).

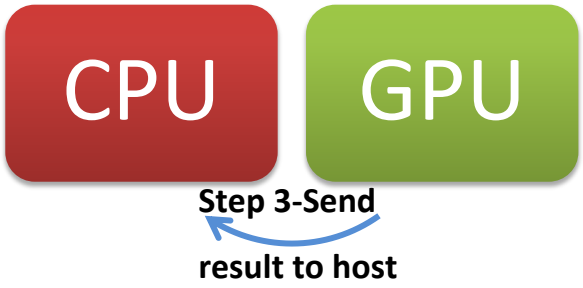

Figure 2.7: Send result from device to host

Lastly, we need to free the memory allocated on the device and host (line 47).

Thus we can summarize the main steps after the allocation of memory required to execute the kernel and before cleaning the memory as follows,

1. Transfer data from host to device
2. Invoke the kernel with the appropriate parameters
3. Transfer the result back from device to Host.

In the next chapter we will begin a case study of Forest Fire Model using CUDA language and we will use all the steps explained above.

Chapter 3

Case Study - Forest Fire Spread Model

3.1 Background

This model is used to simulate the spread of fire in a forest. In reality there are many factors that affect the spread of fire such as humidity, trees density, wind ...etc. In our project and implementation we will consider a simple model of Forest Fire with simple transition rules. The model is represented using 2D cellular automata with Von-Neumann neighborhood. The possible states for the cells are empty, tree, fire and ash. The transition takes place if one cell is fired, and the fire begins to spread. The transition rule is reproduced from a sample Fire Automata in CORMAS[8]:

If a cell is tree at time t, it will become fire at time t+1, if and only if one of its four neighbors is on fire at time t. If the cell is fire at time t, it will become ash at time t+1. If the cell is ash at time t, it will become empty at time t+1.

Figure 3.1 shows a simple example of Fire Automata [8], where the colors white, green, red, and gray represents the states empty, tree, fire, and ash respectively. The figure represents the forest at some type t, where a fire begins in a tree, we will name it *state S0.*

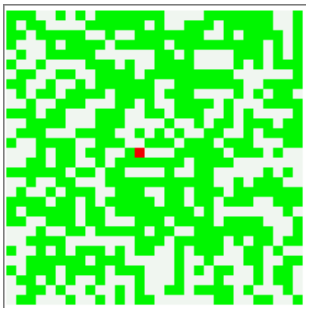

Figure 3.1: Forest Fire at time t (S0)

At time t+1, state S1, the Forest Fire will be as shown as figure 3.2, where we observe that the two trees catch fire and the fire is changed to ash.

Figure 3.3 shows the evolution of the forest from state S2 to S7 and how the fire spread in the forest.

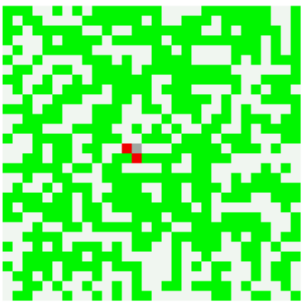

Figure 3.2: Forest Fire at time t+1 (S1)

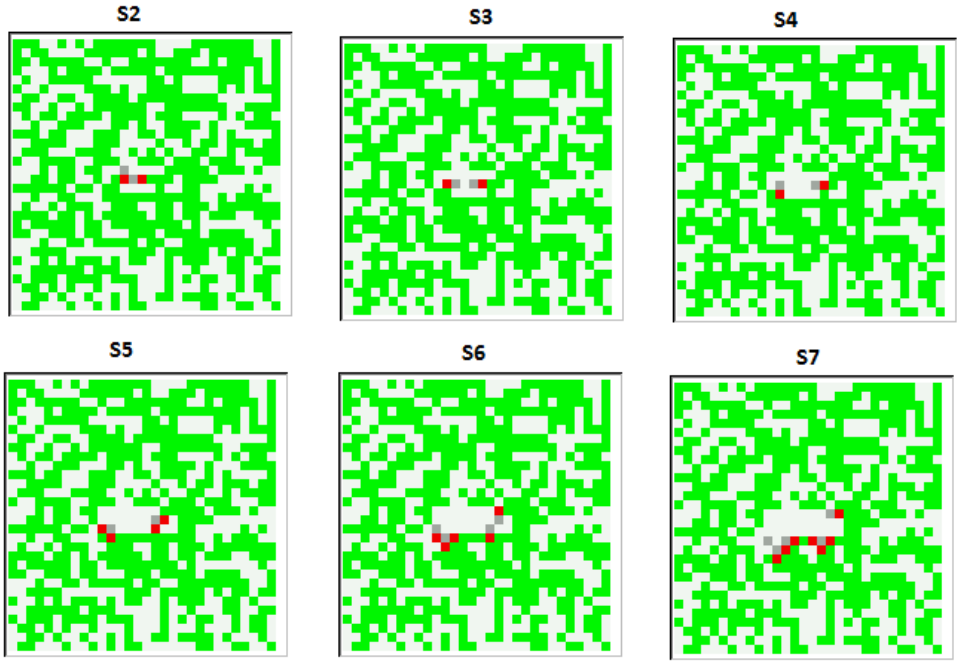

Figure 3.3: Fire Spread Evolution

After some time we recognize that the fire spreads and the empty area increases as shown in figure 3.4. The fire still spreads as there are trees neighbors to fire.

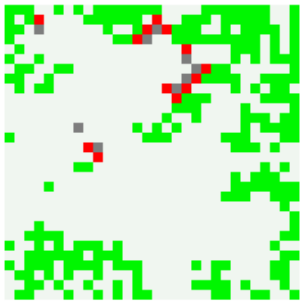

Figure 3.4: Forest Fire after several turns

In this project, the case study we are going to implement this Forest fire model on the GPU and discuss it in terms of performance and optimization. It will be implemented using two different approaches. The first approach is normal massive parallel model, and the second approach is its extension to a pipelined parallel model.

3.2 Hardware and Software

The forest Fire model will be implemented on a Linux machine with NVidia graphics card GeForce GTX 680. The machine is equipped with Intel(R) Xeon(R) CPU E3-1240 V2 @ 3.40GHz processor which is 64 bit CPU, but the kernel used is 32 bit (i686). Besides, the machine is installed with CUDA SDK to develop CUDA applications. The details of this card are given by executing a sample CUDA program name **"deviceQuery' '**, the main details are given below and the rest are given in appendix A.

- Global Memory: 4GB (4294639616 bytes)
- CUDA Cores : 1536
- Maximum number of threads per block: 1024

As we have 1536 CUDA cores, than our kernel is capable to execute 1536 instructions at the same time, which in turn produce considerable speedup of the problem processing. We need to know the number of threads per block as it will be used in the code, and thus the max value that can be chosen is 1024. The global memory available in the GPU is 4GB, which allow storing 1073741824 integer value, which is exactly equal to 32768 * 32768 integer value. Finally to compile CUDA programs we will use **nvcc** compilers as explained in chapter 2 which support C/C++ extension.

3.3 Parallel Forest Fire Model

In this model we will simulate the forest fire spread. The forest states are empty, tree, fire, ash represented by values 0, 1, 2, and 4 respectively. The program has two input methods, the first way asks the user to specify a map BMP image, after that the program reads this image, analyzes its information, size and data and generate initial forest state based on the green color detected in the image, whereas the second way randomly generate a forest. We will explain the details of these input methods later. After we have the forest states, a fire is generated randomly, and the forest state is sent to the GPU. Then the transition kernel is

invoked to apply the transition rules explained previously and the result is sent back to the CPU, in order to be displayed. The transition function will be executed a number of times specified by the user and the evolution of the forest will be displayed on the console.

In the following part we will explain the important parts of the code

3.3.1 Data Structure

The forest is represented by a **struct image** composed of two integer variables, width, height to hold the width and height respectively of the forest, and an array to store the forest state named gridCellState. The array is presented in row − major order [11], i.e. the two dimensional array are represented in a linear 1D array. To clarify the row—major order here is a simple example, suppose we have a forest of size 4 * 4, with the values stored in a two dimensional array as shown in figure 3.5.

0	1	1	1
1	1	0	1
1	1	1	1
1	1	2	0

Figure 3.5: 4*4 Forest grid

Normally the array can be accessed using two indexes, one for the row and one for the column. But fortunately the values are stored contiguously in the memory as shown in figure 3.6, thus we can define a one dimensional array to hold the same data of two dimensional arrays and distinguish the elements using an appropriate indexing formula.

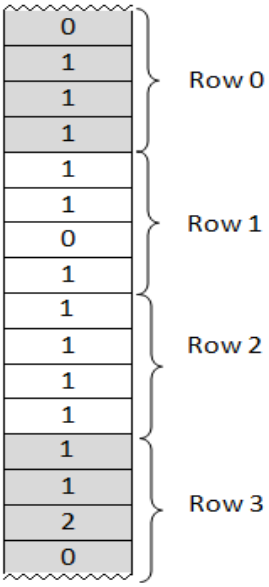

Figure 3.6: Two Dimensional Arrays memory allocation

The one dimensional array can be accessed in two dimensional way using the following formula

```
Index = NumOfColumn * row_index + column_index.
```

Back to the struct definition, the listing below displays the record that we will use in our case study.

Listing 3.1: struct image

```
struct image
{
        int width;   //to hold the number of rows of the grid
        int height;  //to hold the number of columns of the grid
        int *gridCellState; // to hold the forest state
};
```

We will use this struct to declare four pointers of struct image.

- Img, img_d: represent the forest in time "t" on the host and device respectively.
- imgTemp, imgTemp_d: represent the forest in time "t+1" on the host and device respectively.

These pointers are allocated on the host. After that we need to allocate the required memory to hold the gridCellState. For img–>gridCellState and imgTemp–>gridCellState, it will be allocated on the host whereas for img_d–>gridCellState and imgTemp_d–>gridCellState it will be allocated on the device.

3.3.2 Input Methods

The first input way asks the user to specify a map BMP image, then if the image file is read successfully, the width and the height information are extracted from the image header, these information help us to get the size of the forest and stores them in the img pointer. Furthermore, the width and the height are used to allocate the required memory to store the forest states. Then the program analyzes and extracts the RGB color information and compares it to a certain reference color. The reference color in our case is dark green, similar to Google map forest color. The values for the reference color are given in listing 3.2

Listing 3.2: Reference colors

```
//The tree reference color
#define RED_REF 65
#define GREEN_REF 75
#define BLUE_REF 65
//The threshold used in detection
#define THRESHOLD 30
```

The algorithm to detect the color uses Euclidean distance to compare the color of the image pixel to the reference color. If the value detected is within a threshold, then we found a tree, thus we put in state 1. In the other case we put zero for the state. An important point to consider while reading BMP files that the colors are stores in this order, blue, green and then red. A portion of the code for is shown in listing 3.3.

Listing 3.3: Part of ReadBMP function

```
        //store the width
        img->width= width;
        img->height=height;
```

```
//allocation on the host
//allocate memory for the forest states at time t;
img->gridCellState = (int *) malloc ((img->width * img->height) *
    sizeof   (int));

//3 bytes per pixel
int size = 3 * img->width * img->height;
unsigned char* data = (unsigned char*)malloc(size); // allocate 3
    bytes

    per pixel
fread(data, sizeof(unsigned char), size, file); // read the rest of
     the

    data at once
fclose(file);

//detect color based on ecidian distance;

for(int i =0 ; i<size; i+=3){//height
        //Red indeed at i+2. Green indeed at i+1, Blue indeed at i,
             the

            order in the file is B, G , Red
        distance = sqrt(pow((data[i+2] - RED_REF),2) + pow((data[i
            +1] -

            GREEN_REF),2) + pow((data [i] - BLUE_REF),2));
        state=0;
        if(distance < THRESHOLD)
        {
                state=1;
        }
        else state=0;
        //i is 3 times more
        img->gridCellState[(i/3)]=state;
}
```

The second input method asks the user to specify the width and the height of the image and randomly generate a forest state. The number of trees generated with respect to empty ground can be achieved using a probability constant **PROB**. We have implemented different methods for this initialization, one that executes on the CPU and the other can initialize a forest on the GPU. The details of this function can be found in appendix B.

3.3.3 Transition Rule

The kernel TransitionFunc() is the main function to be executed on the GPU. It takes as input the state of the forest in time t and produces the new state to be used in time t+1. The output produced is based on the transition rule explained at the beginning of the chapter and Von-Neumann neighborhood explained in chapter 2. Listing 3.4 shows the transition function code and a simple example of forest state shown in figure 3.7 will be used in the discussion of this transition function.

Listing 3.4: Transition function

```
//Function for transiton rule
__global__ void transitionFunc(int *grid, int *bufferGrid, int width, int
    height){
        //check the cell's 4 neighbors
        //Northj, south, east, west
        int cellState;
        int north, east, south, west;
        int i;
```

```
        i= blockIdx.x * blockDim.x + threadIdx.x;                                    8
        if(i<height*width){                                                          9
           //The north neighbours of the first row are the one in the last        10
               row, else the row above
           north= ((i-width)<0) ?  (height*width) - width  + i : i-width;          11
           //south neighbour of the last row are the first row, esle the          12
               row below
  south= (i+width) >= (height*width) ? ((i+1)%(height*width))  : i+width;          13
           //here we need to check if the east neighbour doesnot exceed the       14
                right border, if so its neighbour
           //is the leftmost element in he same row                               15
           //to get in which row we are, we use (i+width)/width                   16
           east = ((i+1)>=(((i/width)*width)+width)) ? ((i)/width)*width:i       17
               +1;
           //west                                                                 18
           west = (i-1)<((i/width)*width) ?((i/width)*width)+width-1 : i-1;      19
                                                                                 20
           //Calculating the new states                                         21
                                                                                 22
                if(grid[i] ==0) cellState=0;                                     23
                else if (grid[i] ==1){                                           24
                                                                                 25
                        if(grid[north] == 2 || grid[south] ==2                   26
                                        || grid[east] ==2 || grid[west] ==2       27
                                            ){
                                cellState=2;                                     28
                        }                                                        29
                        else{                                                    30
                                cellState=1;                                     31
                        }                                                        32
                }                                                                33
                else if (grid[i] ==2) cellState= 3;                              34
                else cellState= 0;                                               35
                                                                                 36
                //Store the new state of time t+1 in bufferGrid                  37
                bufferGrid[i] = cellState;                                       38
        }                                                                        39
}                                                                                40
```

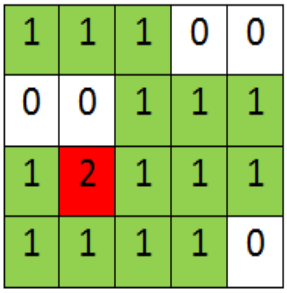

Figure 3.7: Forest Fire State (5*5)

Line 8 is used to find the thread id, as explained in chapter 3. Then we have an *if* statement that ensure that no more than the required threads is executed. At line 11 to 19, we begin to find the index of neighbors for the currents cell. The neighbors index of the current cell are stored in the integer variables north, south, west and east.

As we explained previously the forest state is stored in row-major order, so the north neighbor of the current cell is the current cell index minus the width. For example, the north neighbor of the fired state at index 11, is the cell at index 11-5=6. This is illustrated figure 4-6. In the same way, the south neighbor of the current cell is the current cell index plus the width, remain with the same example, the south neighbor index is 11+5=16. The east and

west are easily achieved by simply adding and subtracting 1 respectively from the current cell index. Figure 3.8 shows a cell with its neighors.

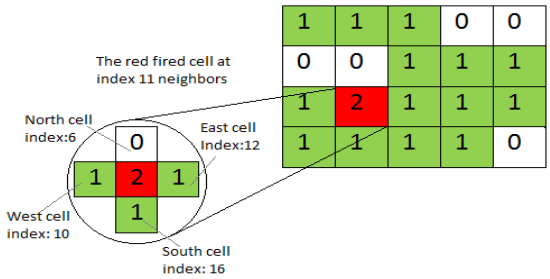

Figure 3.8: Normal Neighbor indexing

Another problem is how to compute indexes of the neighbors for the cells on the edges. As we explained in chapter 4, we will use a circular approach. For example, if the current cell is the cell indexed 0, and then its north neighbor is the cell in the last row of the same column. This can be achieved by having the image size (width*height) minus the width plus the index of the current cell, thus for the first element the north neighbor is (5*4)-5+0=15. It is illustrated in figure 4-7. Furthermore for the same cell the west neighbor is rightmost cell in the same row, refer to figure 4-7. Thus we need first to identify the row: this can be achieved by finding the floor of dividing the current cell index by the width. The next step is to find the starting index of the left-most element in the row, and this can be achieved by multiplying the row index by the width, finally we add the width to this starting index of the current row and subtract one. The expression can be written as:

```
(Floor(i/width) * width ) + width -1
```

Thus if the current cell is 0, the west is (floor(0/5)*5) + 5 -1 = 4. However there is no problem for the west and south neighbors for this cell. A simple example in figure 3.9 illustrate the idea.

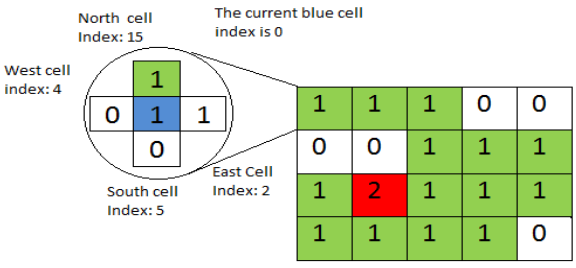

Figure 3.9: North and west neighbor indexing for edge cell

It remains the problem of south neighbors for the cells in the last row as well as the east neighbors for the cells in the right edges. For example, if we are at cell indexed 19 then the south neighbor would be the cell on the first row on the same column indexed 4. This can be obtained by the following expression,

```
(i+width)mod(width*height)
```

Where i is the current cell index and mod is the rest of division. Moreover the east neighbor is the cell on the left-most on the same row. It is illustrated in figure 3.10 and can be found by

```
Floor (i/width) * width
```

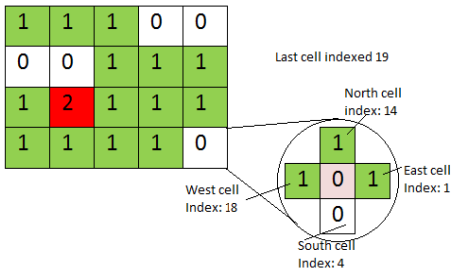

Figure 3.10: South and east neighbor indexing for edge cell

So as we know how to calculate the neighbors, it is now a necessity for the code to check if the cell is in an edge and thus to use the appropriate edge expression. Otherwise the normal neighbor expression is applied. After the neighbors index has been found, we straight forward check the state of the cell according to the transition rules specified above and hold the new state of the cells in a new array called "*bufferGrid*" for the next step in time t+1.

3.3.4 RunSimulation function

As we described the data structure and transitonFunc which is the kernel, we implemented this function that is responsible for each iteration in manipulating the forest on the GPU. Its main tasks are as follows:

- Copy the forest state from the host to the device.
- Execute the transition function, i.e. the kernel, and produce the new state in imgTemp_d–>gridStates.
- Copy the new state back from device to host
- Make the new state available as input for the next iteration by invoking the swapGrid method

The code in listing 3.5 translates the previously described task.

Listing 3.5: runSimulation function

```
void runSimulation()
{
  int nBlocks = (img->width*img->height)/BLSIZE + ((img->width*img->height)
      %BLSIZE==0?0:1);
        //copy data from host to device
        cudaMemcpy(img_d->gridCellState,img->gridCellState,size_image1,
            cudaMemcpyHostToDevice);
  transitionFunc<<<nBlocks,BLSIZE>>>(img_d->gridCellState, imgTemp_d->
      gridCellState, img->width, img->height);
        //copy data back from device to host
        cudaMemcpy(imgTemp->gridCellState,imgTemp_d->gridCellState,
            size_image1,cudaMemcpyDeviceToHost);
        //store the new state in grid
  swapGrids(img->gridCellState, imgTemp->gridCellState);
}
```

3.3.5 Main function

This main function firstly allocates the required memory both on the host and the device. After that the forest is generated randomly or by reading a map as explained previously. Then A fire is generated randomly using **generateFire()** function to a number of trees specified in the directive **FIREDTREES** after the first step. The next step is to invokes the runSimulation function. There are two approaches to iterate the simulation, we can either use a "*for loop*" for a number of step, or as we will be using the OpenGL (Open Graphics Library), we have a function called "**glutMainLoop**()" which call the "**display**()" function. Then the display function calls the "**runSimulation**()" function and these successions of function calls continues without stop until the user take an action by closing the window. Finally we need to clean the memory. The code is shown in appendix B. However the details of the openGL code are not explained in this report (it is borrowed from CUDA sample programs).

3.3.6 Output

The figures 3.11 to 3.13 below shows the output for the randomly generated forest. In the first figure we found that 3 places has fire marked by circle. Then the second small figure shows 3 steps of the middle fired region. And the third figure after the spread of the fire in the whole forest.

Figure 3.14 to 3.15 shows the output of an image map of Brest city france when fire is generated in this green area.

3.3.7 Execution Time

To evaluate the parallel version execution time, we have compared it with the sequential version execution time. The figure 3.17 represent this execution time where the x *axis* has size values of the forest and the y *axis* the time in seconds. For each forest we execute it a number of time equal to 2*log(ForestSize) , for example a forest of size 1024*1024, is executed 2*(log(1024*1024)) which equal to 2048. The log is to the base 2. We recognize that for large size problems the execution time is very small compared to the sequential one.

Furthermore, for more exact comparison, we had the following table in figure 3.18 which gives the exact execution time in seconds, and the speed up. We observe that the time consummation for small sized forest is more in parallel version than large sized problem, and this is normally true, due to the transfer of this small data between device and host. However our interest is the large sized forest and we are very excited to found that in these forests we can get a very big speed up. Thus, very small execution time compared to sequential verison.

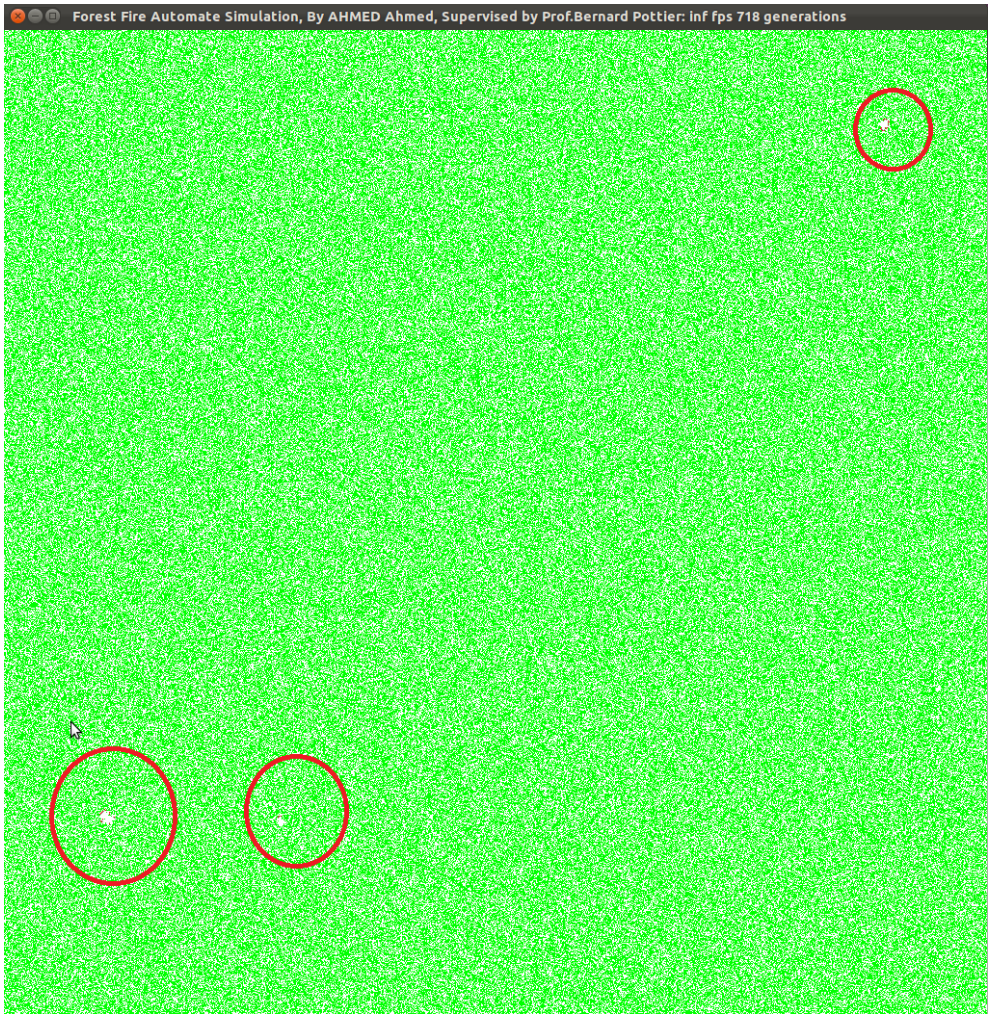

Figure 3.11: Randomly generated fire

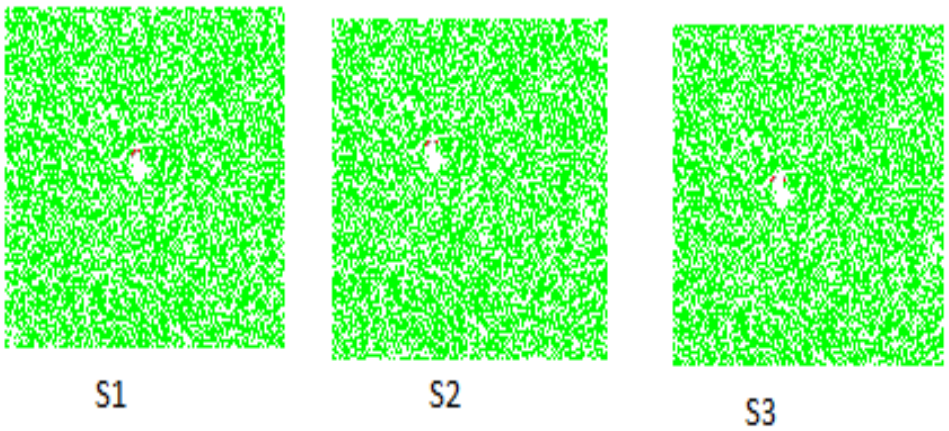

Figure 3.12: 3 steps of transiton for middle region

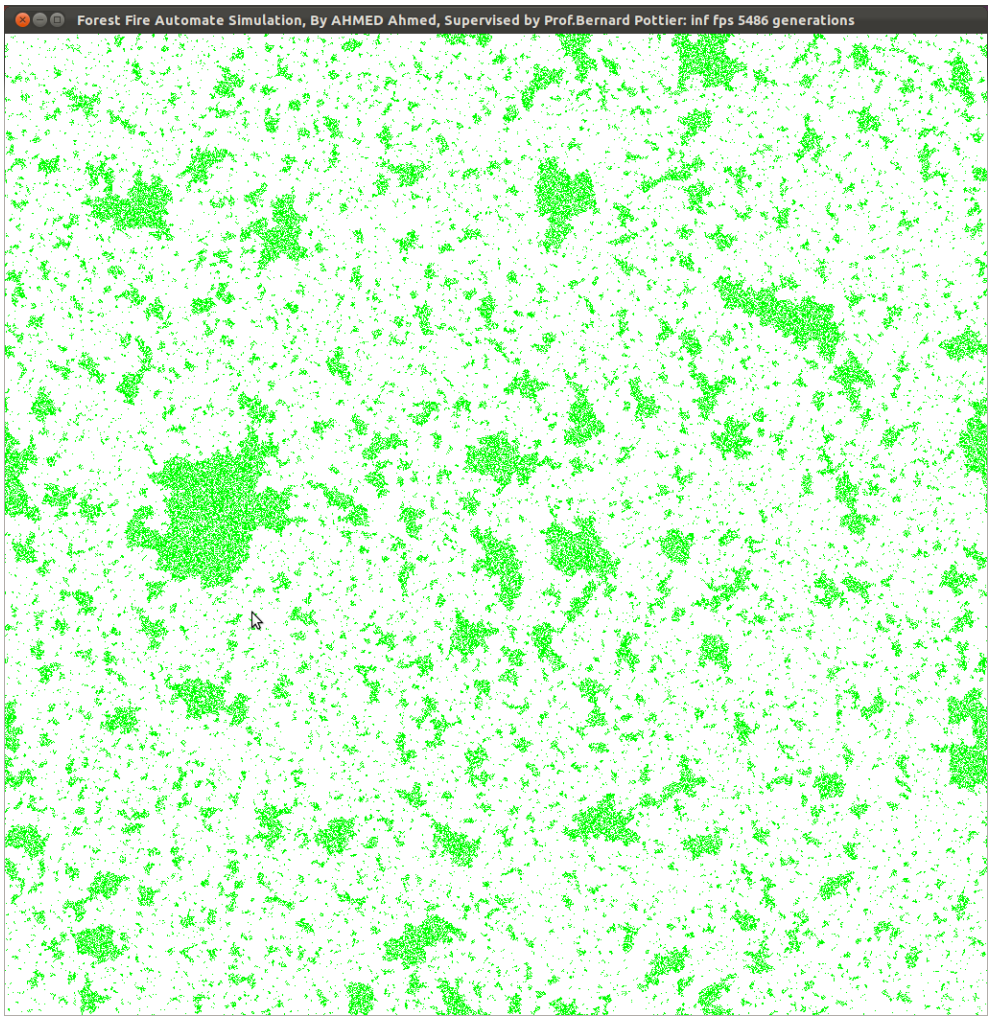

Figure 3.13: After fire spread

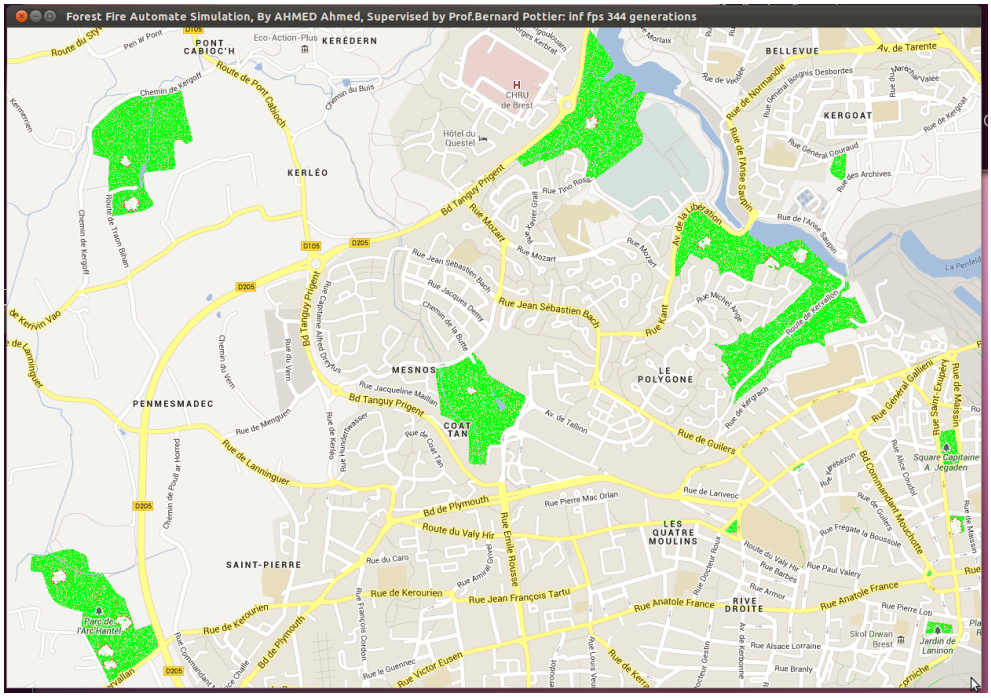

Figure 3.14: Fire spread in Brest city trees areas

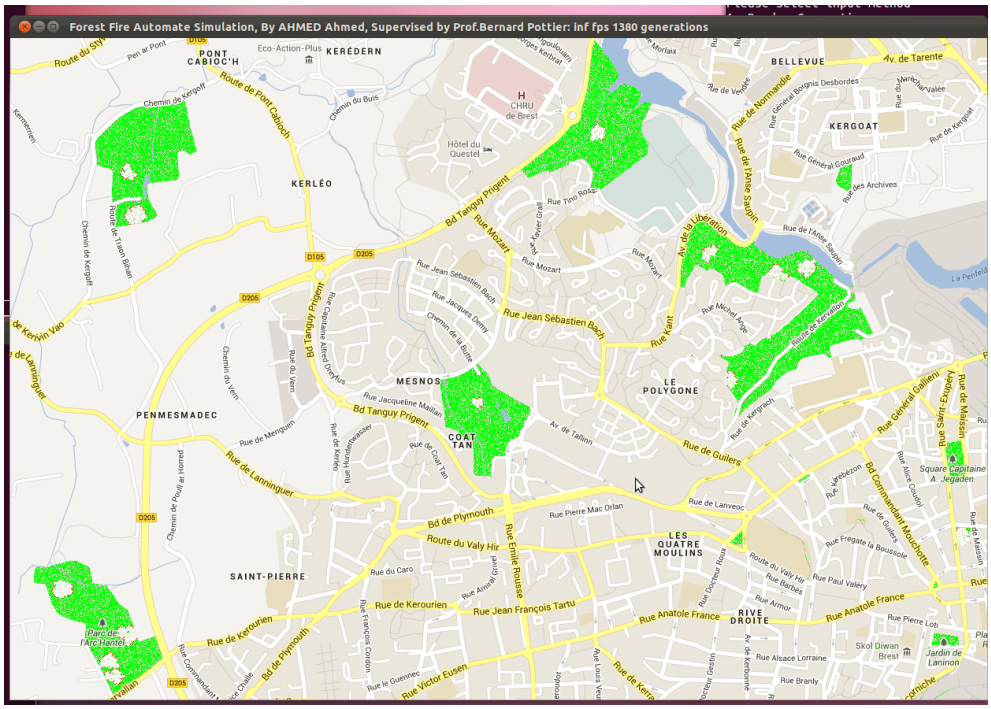

Figure 3.15: Fire spread after some steps

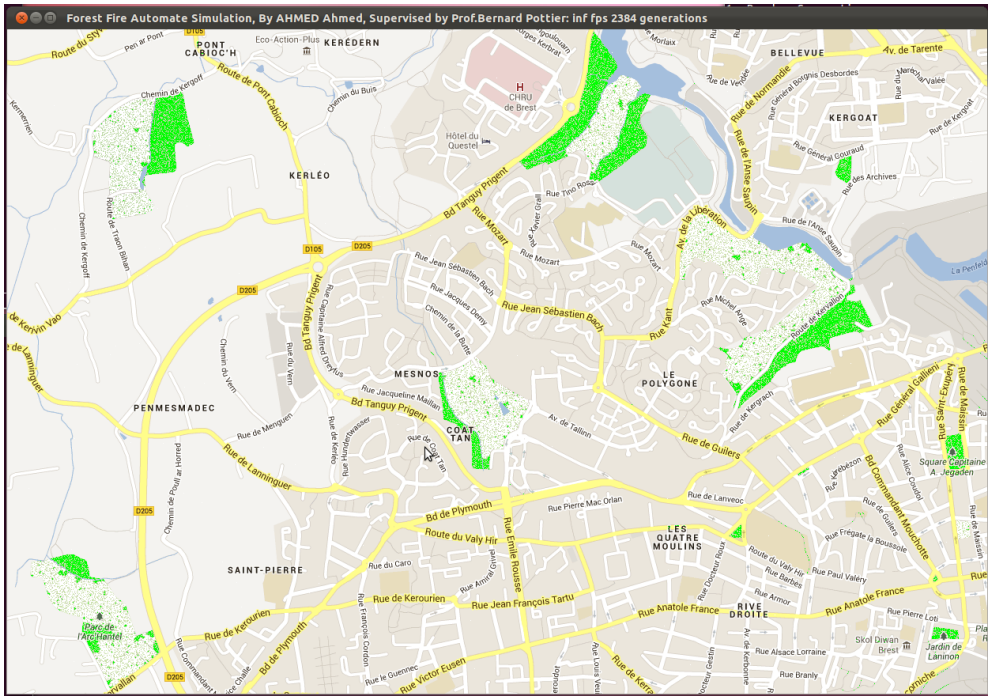

Figure 3.16: After fire stops

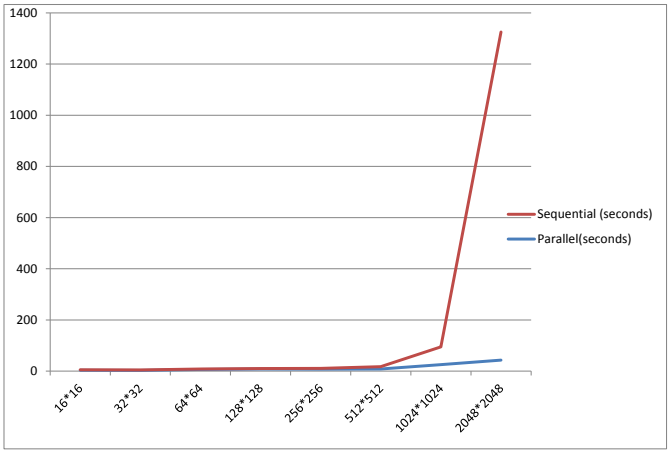

Figure 3.17: Sequential vs Parallel forest Fire

	Parallel(seconds)	Sequential (seconds)	Speed Up
16*16	3.729389	1.67264	0.44850242
32*32	3.037199	1.816259	0.59800461
64*64	6.456442	1.934578	0.29963531
128*128	8.185488	2.301446	0.28116173
256*256	7.797166	2.929686	0.37573729
512*512	8.527426	9.262503	1.08620151
1024*1024	25.578263	69.278088	2.70847508
2048*2048	43.257623	1281.87583	29.6335244

Figure 3.18: Parallel Model Speedup

Chapter 4

Forest Fire Pipeline Version

The latency time in the parallel version is the sum of the time of execution of sending data to GPU, exciting the kernel and returning the result back. So to enhance this time we can use a pipeline approach that will be explained later. But to achieve this approach we need firstly to send the forest part by part to be processed. So the first part of this chapter will be concerned with slicing the image and the second part will apply the pipeline to these slices.

4.1 Sliced Forest Fire Model

In the previous parallel forest fire application, a forest state is sent fully to the GPU, apply the transition rule on the kernel and send the result back to host. These steps are performed for a number of times specified by the variable steps. However to enhance the performance and deal with very large problems, we need to send the forest state to the GPU part by part or slice by slice. For example a forest of size 1024 * 1024 (more than one million) cell can be sent to the GPU in eight parts of size 1024 * 128 or we can even send it in 1024 part, where each part o 1024*1. Figure 4.1 show a forest of size 32*32 and it is divided into 4 slices, where each slice is sent to the GPU to be processed separately. We need to remark that the width is always the same and the partitioning depends on the height.

4.1.1 Difficulties in the sliced approach

- **Slice offset**
  The full forest state is stored on the host, and to send the forest part by part we need to hold the starting offset of each part from the original forest state. This can be achieved by extending the image struct used in the previous forest model as shown in listing 4.1

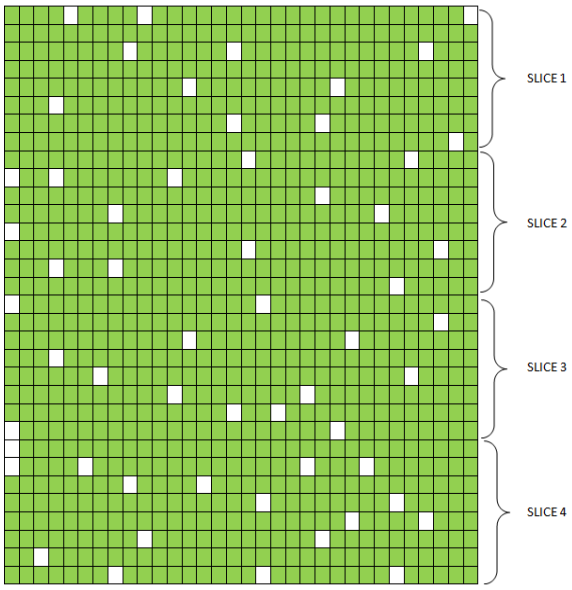

Figure 4.1: Forest 32*32 divided into slices

Listing 4.1: struct image

```
struct image                                                    1
{                                                               2
        int width;   //to hold the number of rows of the grid  3
        int height;  //to hold the number of columns of the grid 4
        int *gridCellState;                                     5
        int offset; //the offest is used while sending an image part 6
            by part
};                                                              7
```

- **Identification of the number of slices**
  The number of slices is specified previously in the program, but we need to adapt it according to the forest size. We can clarify this point by the following example. Suppose the number of slices is 4 and the height is 32, then the four slices can represent the whole forest state regardless of its width. However there are some forest sizes where the height cannot be divided by the slice numbers, suppose that the number of slices is 10 and the height is 38, then dividing 38 by 10 will give us 3 and the remainder will not be used, thus we will have eight rows of no use. And the appropriate value for the slice number is 13. In the code our starting number of slice is stored in a define directive named SLICE, this value is used to get each part height by dividing the height of image by it. The size of a slice is stored in a variable named part which is equal to the width of the image multiplied the slice height found by dividing the height of image by the SLICE. We need to note that, in the division we only take the quotient. For the rest of elements that are not included by the number of slices, a local variable named slice_num is adapted to the appropriate number of slices required as shown in the pseudo code below

```
part = forest_width * foor(forest_height / slice)
        slice_num=SLICE;
while (slice_num * part) < (forest_height*forest_width) do
        slice_num = slice_num + 1;
end while
```

- **Neighbors**
  As we know each cell updates its state depending on the state of its neighbors, therefore we need to send the north and south neighbors, i.e. one more row from top of slice and one more row from bottom of slice. Thus the size of a slice will be more by two widths. However we have several scenarios that will be explained in the function **toGPU**. Figure 4-B shows how the second slice of the example shown previously in figure 4.2 is sent to the GPU with north and south neighbors shown in blue

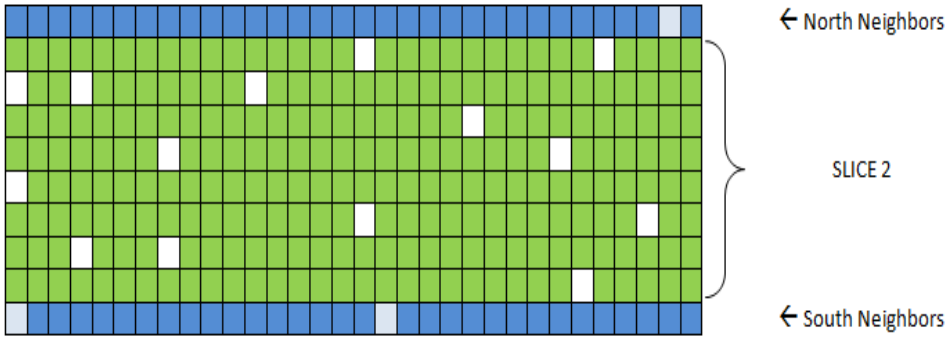

Figure 4.2: Slice of forest

4.1.2 Coding

As explained previously to process data on GPU, we need to send data to GPU, execute the kernel and finally send the result back to host. Thus we defined three functions to perform this tree tasks.

- toGPU
- processOnGPU
- fromGPU

We are going to illustrate each function on details.

4.1.2.1 toGPU()

This function is used to send a slice of the forest to the device. It has two parameters where both are pointers of type struct image. The first pointer **tin** is a pointer to the whole forest on the host whereas the second pointer **tout** is pointer to a part of the forest on the GPU. We have to note carefully that the memory on the GPU has been already allocated previously to hold one slice with two neighbor rows. The code starts by declaring some variables and then calculating the size of a part as explained previously, and then it calculates the size of a part in addition to the neighbors. After that it adapts the slice number as explained previously. A part of the code is shown in listing 4.2.

Listing 4.2: Slice size

```
part = tin->width * (tin->height/SLICE);

//part  with neighbors hold a part with north and south neighbors
//add two rows, one on top for north neighbors
//and one on bottom for south neighbors
```

```
part_with_neighbors=part + (2*tin->width);                                  6
                                                                            7
//Check if the image can be fully represented by the number of slice        8
    seleced,
slice_num =SLICE;                                                           9
while ((slice_num * part )  < (tin->height * tin->width))                   10
        slice_num++;                                                        11
```

Now to copy the data from the host to GPU, we need to use cudaMemcopy, but we have different scenario.

- Copying the first slice
- Copying the second slice to before the last slice
- Copying the last slice

Copying the first slice

Firstly we need to copy the north neighbors of the first slice which is the last row in the image as shown in listing 4.3. This last row starting index can be found as shown in line 2. Then in line 4 we calculate the size of bytes for this last row. After that we begin to copy this last row to the memory pointed by the second pointer for a number of bytes found by size_north. Figure 4.3 illustrate the transfer of this north neighbor.

Listing 4.3: First slice transfer to GPU

```
//North row indexing begin at the leftmost elements of last row             1
int north_row = (tin->width*(tin->height-1));                               2
//size of north neighbors                                                   3
size_t size_north = tin->width * sizeof(int);                               4
//copy to gpu, firstly the north neighbor                                   5
cudaMemcpy(tout->gridCellState,tin->gridCellState+north_row,size_north,     6
    cudaMemcpyHostToDevice);
//copy the the first part from begining to size of one part plus size of a  7
    width
size_t remaing_part = (part + (tin->width))*sizeof(int);                    8
cudaMemcpy(tout->gridCellState+(tin->width),tin->gridCellState,remaing_part  9
    ,cudaMemcpyHostToDevice);
```

After that we begin to transfer the first slice information, the copy (line 9) starts from the first position of the forest until the size of one part added by one row for the south neighbor as shown in figure 4.4. We have to notice that the information is stored contiguously after the north row copied previously.

Copying the second slice to before the last slice

To copy the slices from the second one to the slice before the last is straight forward. The reason for that is the existence of the north and south neighbor directly before and after the slice respectively. Thus we begin the copy from the position of one row before the slice and the copy stops when we arrive to the size of a slice and two width size, hence it includes the south neighbor too. The code is shown in listing 4.4 below and figure 4.5 to illustrate the idea.

Listing 4.4: Copy slices between first and last slice

```
//copy a part of the image including one row before and one row after for   1
    NORTH and SOUTH neighbour respectively
size_t part_size = part_with_neighbors * sizeof(int);                       2
cudaMemcpy(tout->gridCellState,tin->gridCellState+(tin->offset - tin->width  3
    ), part_size ,cudaMemcpyHostToDevice);
```

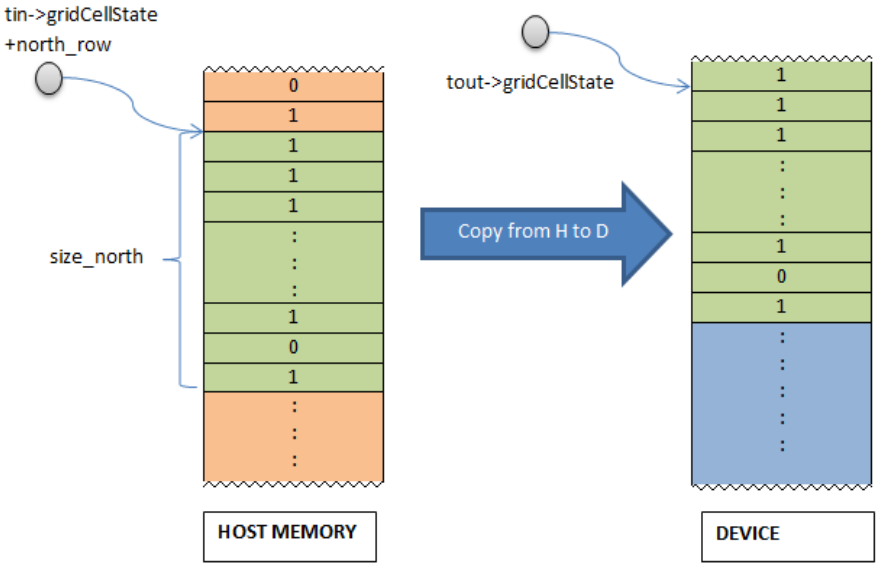

Figure 4.3: Copy north neighbors of first slice

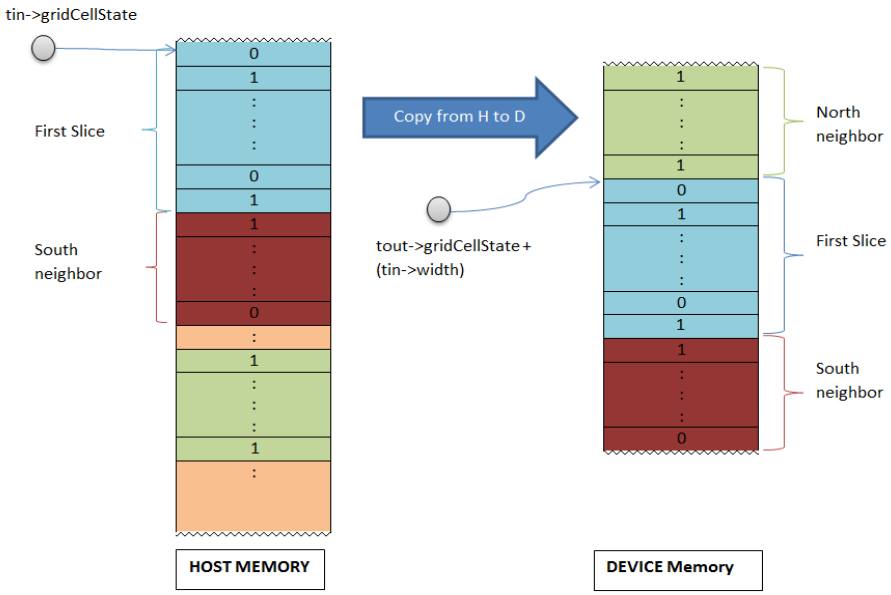

Figure 4.4: Copy first slice with south neighbor

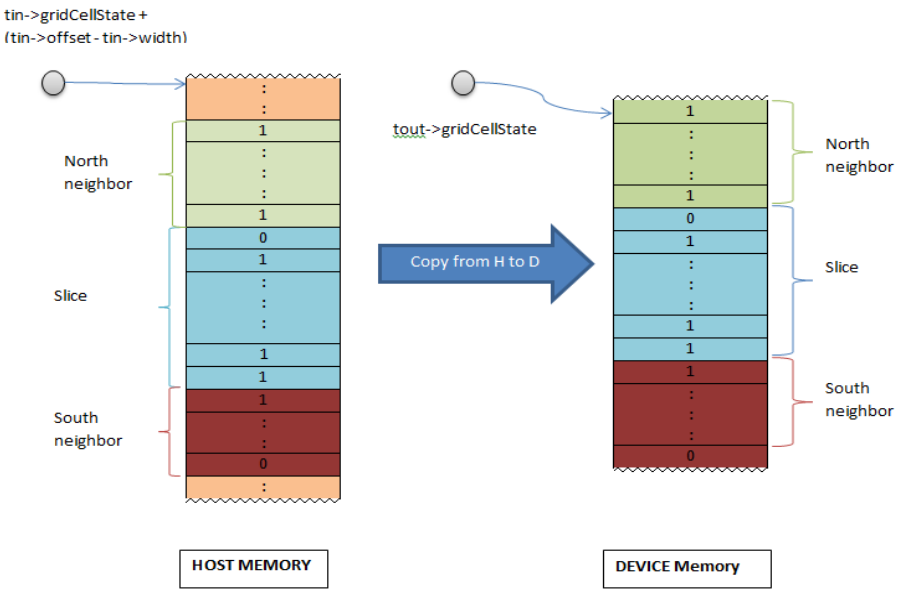

Figure 4.5: Copy slices between first and last slice

Copying the last slice

In copying the last slice, we have to to take care of the number of rows in this slice. This is obtained in last_part integer by subtracting the size of the image from the multiplication of the part size by the number of slices. Furthermore, we need to add one row for the north neighbor. For the south neighbor, it will be the first row of the forest. One of the important points here, if the size of the last slice with its neighbors does not fulfill the size of a part to be send to the GPU. This will produce false results, thus to process this problem, we used the variable part_remaining, which obtains the number of elements to be copied from the beginning of the image to fill the memory. Only the top row will be used while processing whereas the rest data will be discarded. The code below shows how we first copy the last slice information with its north neighbor and then copying the south neighbor with or without more rows to fill all the memory.

Listing 4.5: Copying the last slice

```
last_part = (tin->width *tin->height) - (part*(slice_num-1)) + tin->width;
//The remaining part to be used from top
part_remaining = part_with_neighbors-last_part;

//Copy the last part , with one row before that represet the north neighbor
size_t last_part_size = last_part * sizeof(int);
cudaMemcpy(tout->gridCellState,tin->gridCellState+(tin->offset - tin->width
    ),last_part_size,cudaMemcpyHostToDevice);
//copy the remaing part
size_t part_remaining_size = part_remaining * sizeof(int);
cudaMemcpy(tout->gridCellState+ last_part,tin->gridCellState,
    part_remaining_size,cudaMemcpyHostToDevice);
```

4.1.2.2 processOnGPU()

This function is used to invoke the kernel and is similar to the kernel explained previously with a small difference. In the previous version the number of blocks is used to occupy a full forest but in this version the number of blocks is used to occupy a part or a slice of the forest as given in listing 4.6

Listing 4.6: processOnGPU

```
//process the data on the GPU                                                    1
void processOnGPU(struct image *tin,struct image *tout)                          2
{                                                                                3
        //The number of blocks within a gird                                     4
        int nBlocks = (tin->width*(tin->height/SLICE))/BLSIZE+ ((tin->width      5
            *(tin->height/SLICE))%BLSIZE==0?0:1);
        transitionFunc<<<nBlocks,BLSIZE>>>(tin->gridCellState, tout->            6
            gridCellState, tin->width, tin->height/SLICE);
}                                                                                7
```

This function is used to call the kernel. The kernel here also differs from the previous version. The difference is that in the previous version we start processing from the first element in the memory, but in this version we need to begin the process from the second row. The number of threads executed is equal to size of a part without the neighbor. Thus to make the first thread works on the first element of the slice shown in figure 4.6, we need to add to the thread id the width of a row in order to access the appropriate position of the element. Therefore we can generalize it and add the width to all thread index.

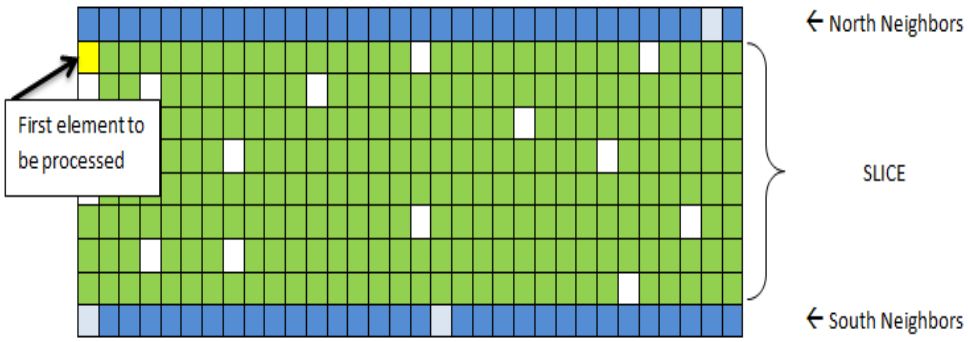

Figure 4.6: First cell in the slice

As shown in the figure above the position of the first element is (i+width) and to obtain the north neighbor we need to subtract the width from i+width which in turns is equal to "I" the thread id. To get the south neighbor we add to (i+width) a width, thus the south = (i+width+width). It is shown in the listing **??** below

Listing 4.7: struct image

```
i= blockIdx.x * blockDim.x + threadIdx.x;                                        1
. . .    . . .     . . .                                                         2
//and the Norht neighbour will be i+width-width = i                              3
           north= i;                                                             4
           //south neighbour                                                     5
           south= i+width+width;                                                 6
. . .    . . .     . . .                                                         7
```

The rest of the code is similar to the previous one but we replace the thread id with thread id + width. The whole kernel code is available in appendix C.

4.1.2.3 fromGPU()

When we send the result back from the device to the host, we need only to send the slice information without its neighbor. After calculating the size of the part and adapting the number of slices, we begin the transfer. We have two cases, the first case for all the slices except the last one and the other case for the last slice. For the first case we copy the whole memory allocated on the GPU for that part towards the host as shown in the code below, but we need to specify to which part of the original forest this information belongs. This can be achieved by using the offset, which holds the position of the part in the forest.

Listing 4.8: Copy result for slices before the last

```
size_t size_image = tin->width*(tin->height/SLICE) *sizeof(int);
cudaMemcpy(tout->gridCellState+tout->offset,tin->gridCellState,size_image,
    cudaMemcpyDeviceToHost);
```

To copy the last part, we first calculate the size of last part and then copy just that part towards the host, discarding the rest as shown in the following code.

Listing 4.9: Copy result of last slice

```
last_part = (tin->width *tin->height) - (part*(slice_num-1));
size_t size_image = last_part *sizeof(int);
cudaMemcpy(tout->gridCellState+tout->offset,tin->gridCellState,size_image,
    cudaMemcpyDeviceToHost);
```

4.1.2.4 All together

In order to process a full forest at time t, we need to send the data towards the GPU, execute the kernel and receive the result back for each slice, thus we need a number of iteration equal to the number of slices to process one image. After that we use this result for time t+1 and this process continues until the number of steps. In this version we recognize that for each part, the time required for one part cycle is

```
Time = Time_toGPU + Time_ kernel + Time_fromGPU
```

But we can achieve better time consummation by using the pipeline approach, which can produce a result of time equal to

```
Time = Max(Time_toGPU, Time_ kernel , Time_fromGPU)
```

This is less time compared to the normal version. We will make use of the thread libraries available in c language to achieve this objective.

4.2 Pipeline Version

In the pipeline structure, we will use the simultaneous multithreading feature of inter i7. For each of the tasks toGPU(), processOnGPU() and fromGPU() we will use a separate thread. Thus on total we need 3 threads that will be executed on parallel. In this pipeline architecture the output from the first thread will be input for the second thread and the output from the second thread is the input for the third thread and the output from the third thread is the final result. The data between all the threads are of the same size so we will define a struct that will represent the input and output data for each thread and declare an array of it of size three, where each element of the array is used by a thread. The code is shown in listing 4.10

Listing 4.10: Data for threads

```
//data for each thread
struct data_thread {
  int threadId;
  struct image  *dataIn, *dataOut;
};

struct data_thread  data[THREADS];
```

The first thread T1 will be responsible for sending data to the GPU, thus its pointer to dataIn− >greatergridCellState is allocated on the host whereas the dataOut− > gridCellState on the GPU. As explained previously the image will be sent part by part towards the GPU so the allocated space for dataIn–>gridCellState is for the whole forest but the dataOut–>gridCellState allocated memory is of a size of a part with its neighbors. The initialization of all threads will be done through *init_data_thread()* function, but the input for thread 1 is previously allocated during the random generation or reading a BMP image. So what is remaining is to allocate memory for the gridCellState on the GPU as shown in listing 4.11.

Listing 4.11: Allocation for thread 1 on device

```
err = cudaMalloc(&dt->dataOut->gridCellState,size_image_part);
if(err)return true;
        break;
```

The second thread T2 will be responsible for the function processOnGPU() which in turns call the kernel. Thus the allocation will be on the GPU memory for the gridCellState. The dataIn is the data arrived from the GPU and this is the result that will be sent to the next thread. The function init_data_thread() will also allocate the memory for the dataIn–>gridCellState with a size of a part on the GPU and also will allocate the memory for dataOut–>gridCellState with a size of a slice without neighbors. Then this allocated memory will be initialized to zero as shown in the code below. The purpose of this initialization will be explained later.

Listing 4.12: Allocation and initialization for thread 2

```
err = cudaMalloc (&dt->dataIn->gridCellState,size_image_part);
        err |= cudaMalloc(&dt->dataOut->gridCellState,  part * sizeof(int))
            ;
        if(!err){
        //Initialization
        SetZero<<<nBlocks1,BLSIZE>>>(dt->dataIn->gridCellState,dt->dataIn->
            width, (dt->dataIn->height/SLICE + 2));
        SetZero<<<nBlocks,BLSIZE>>>(dt->dataOut->gridCellState,dt->dataIn->
            width, dt->dataIn->height/SLICE);
        }
```

The third thread T3 is used to transfer the data form GPU to host. The gridCellState is allocated on the GPU for the input and allocated on the host for the output. The dataIn is the data received from thread 2 and the dataOut is the final result. As usual the function init_data_thread() will allocate the memory for the dataIn–>gridCellState with a size of a slice without neighbors and also allocate the memory to hold the last gridCell result on the host with size of a full forest. After that this allocated memory will also be initialized to zero. The full code is available in apprendix C.

4.2.1 Functionality of the threads

After the threads have been declared and initialized, we need to create and execute these threads as shown in listing 4.13. The functionality of the threads is assigned in the function called threadStep(). The data related to each thread is passed as the last argument in the function pthread_create().

Listing 4.13: Creation and execution of threads

```
//To create and execute threads
for (i=0; i<THREADS;i++){
        pthread_create(&threads[i],NULL,threadStep,(void *)&(data[i]));
}
```

The function threadStep() launches the appropriate function depending on a number, the thread id. The code for this function is shown listing 4.14.

Listing 4.14: threadStep function

```
void * threadStep(void * args)
{
        int id;
        struct data_thread *sdata;
        sdata = (struct data_thread *) args;
        id  =  sdata->threadId;
        switch (id) {
                case T1:
                        //Thread 1 is responsible to send the data from
                            Host to Device GPU
                        toGPU(sdata->dataIn,sdata->dataOut);
                        break;
                case T2:
                        //Thread 2 execute the kernel
                        processOnGPU(sdata->dataIn, sdata->dataOut);
                        break;

                case T3:
                        //Thread 3 send the data from the Device to Host
                        fromGPU(sdata->dataIn, sdata->dataOut);
                        break;
        }
        return 0;
}
```

You can refer to the appendix to other parts of thread related code. So these threads works concurrently, with periodic barrier synchronization. In the first turn, all the threads will be working on the same time and the only useful result is the result of toGPU(). The result of processOnGPU() and fromGPU() will be of no use in the first turn. An important question arises, how to transfer and share scurely the data between the threads?

It can be done by using swapping function called dataMove() as shown in listing 4.15. And this is the reason for why we initialized the data to zero in the thread initialization phase as we will need to execute some threads without useful results.

Listing 4.15: Exchange data between threads

```
void dataMove()
{
        int *tmp;
        //Exchange the data between the threads.
        //The input of the T2 is the output from T1
        tmp = data[0].dataOut->gridCellState;
        data[0].dataOut->gridCellState = data[1].dataIn->gridCellState;
```

```
        data[1].dataIn->gridCellState = tmp;                                    8
                                                                                9
        //The input of the T3 is the output from T2                             10
        tmp = data[1].dataOut->gridCellState;                                   11
        data[1].dataOut->gridCellState = data[2].dataIn->gridCellState;         12
        data[2].dataIn->gridCellState = tmp;                                    13
}                                                                               14
```

The next observation is that again the second turn the result of from kernel will be of no use. After that from the third turn all the results are useful because the third will be having input of first thread before two turns and the second thread will have input of the first thread before on turn. Thus in general we can conclude, to obtain an output for a particular input in thread 1, we need two more cycles to propagate the data from first thread until the last thread. Also this is applied at the end of the pipeline where we have to have two more turns to have the final result. This is shown in figure 4.7 where the threads marked in * are executed without any useful need. The figure represents five slices of an image to be processed on the GPU and the result sends back to host.

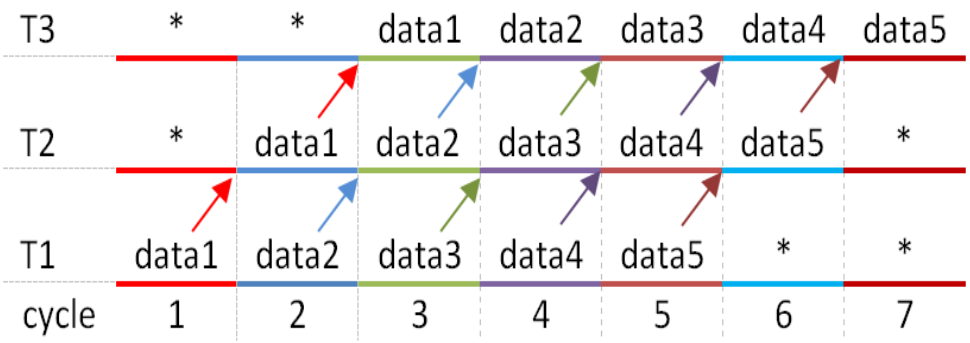

Figure 4.7: Pipelined thread

After we receive the final result of the last slice we use the full image result as input in the next iteration for the forest and the operation is repeated as explained previously a number of times equal to the value specified in the variable steps. The code is available in the appendix with an output similar to the simple parallel version.

4.2.2 Execution Time

Figure 4.8 illustrates the execution time for the parallel version compared with the pipelined version, where the number of sliced is set to 30. The comparison with the parallel version, shows that pipelining consumes more time. This is normally true, due to some reasons. Firstly we test the pipeline approach with small data compared to the GPU memory of 4GB. Also the synchronization time for the multithreading has an excessive overhead cost when used with these small data. Conceptually, we can expect to have a smaller execution time and better result if we deal with targeted larger data space.

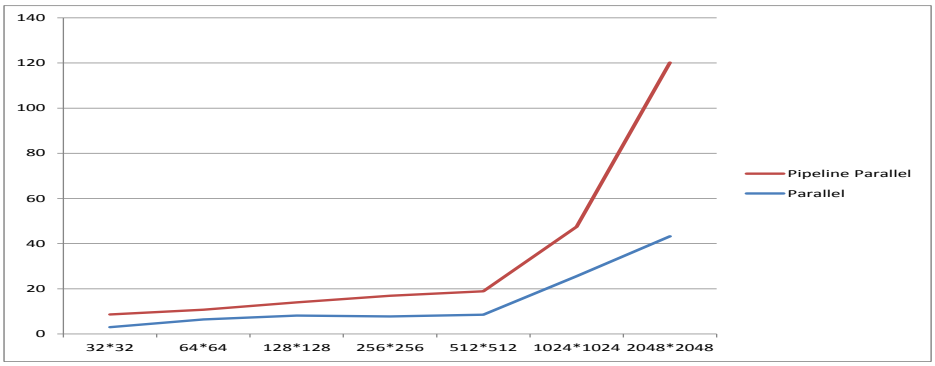

Figure 4.8: Parallel vs Pipeline Parallel

Chapter 5

Conclusion

With environmental modeling we are able to understand many real life features and phenomena. The phenomenon that is under study is brought into a representation that can be simulated. One of these modeling techniques is the Cellular automata. Due to its spatial and temporal structure and its change of state we can implement many diffusion problems, e.g. the forest fire spread. One of the important problems that face the modelers is the execution time, and our main objective was to minimize it. Fortunately, due to the GPUs available today we reached a much smaller execution time compared with the sequential version. Moreover, dividing the environment into slices and sending it to the GPU, is a better approach when used with pipeline structure and using large complex data that occupy the whole capacity of the GPU, but this case is the focus of ongoing studies.

Future work

- The Forest Fire cellular automata can be extended to many real life problems such as the fluvial dynamic in a landscape in terms of sediment [3], pest modeling for South East Asia, locusts in Madagascar, Senegal, etc...

  Moreover, deeper study on potential extension application field for celullar automata can be performed with a study of the problems at limits of space and clock resolution.

- The cellular fire automata model can be improved to simulate fire in real environment taking into account all the fire diffusion factors (wind, humidity, etc..). We can also integrate it with some other synchronous simulations.

  For example, we have the NetGen tool developed in the LabSTICC lab of University of Brest[12, 13, 14]. This tool allows to simulate distributed algorithms used in wireless sensors algorithms. It is also connected with geographic map tool developed by Pierre Yves Lucas. On the map we have a number of wireless sensors that are distributed over a wide area. There is the possibility to combine the network modeling with the fire spread simulation in two one complex simulation scheduling algorithms, where if a fire is generated in an area, then it is diffused using the CA simulation and the work of the sensors to simulate the propagation of an alert concerning the detection of the fire.

  In this type of problems we have to take into account two types of time, time to diffuse the fire the time to propagate the news of fire. We need to arrive to know whether the massive parallelism has a positive impact in quick diffusion and broadcasting the

alert. Thus, among the points of interest to be studied, is the synchronization of time between the two simulation models.

Another points is the spatial synchronization, how and where the sensors that detect a fire in an area diffuse the news in priority to the area affected. Also, what are the stability and the accuracy of the network system? Can it be able to withstand the fire? To more clarify the idea, An image of a map of Brest city in France is shown below, the first figure shows a fire simulation model where fire is diffused in the forest area and the second figure shows a network sensors models where the sensors are distributed in the same forest areas.

Thus the possibility of scheduling and combining the physical and observation systems is achievable.

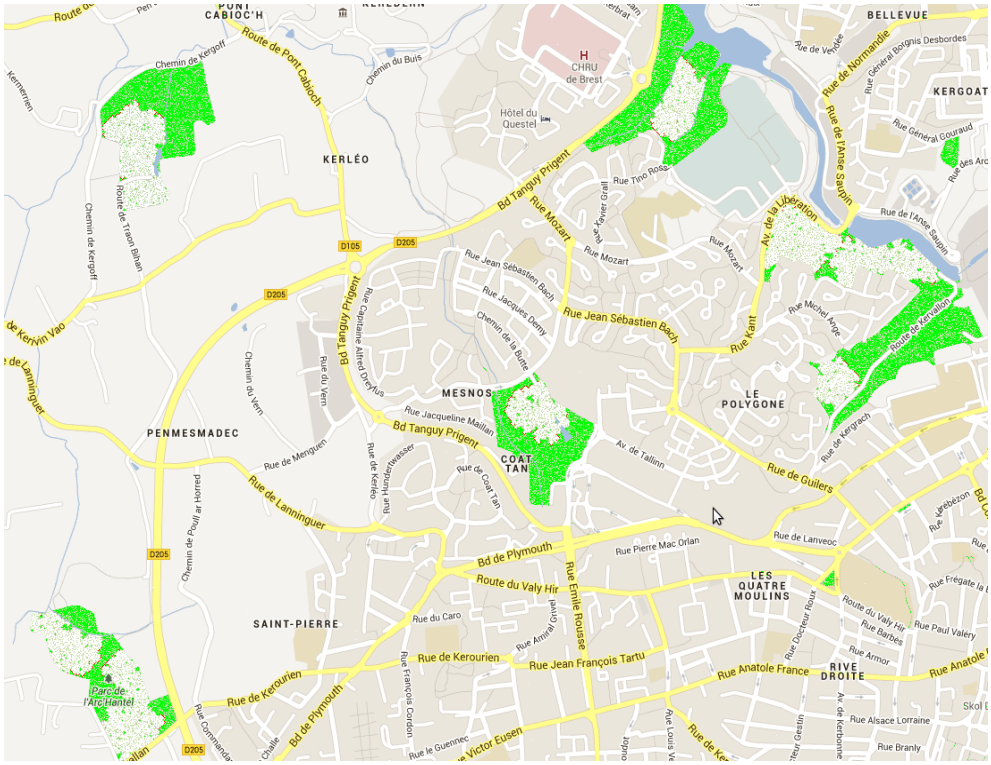

Figure 5.1: Fire Simulation Brest City

Technically, during the internship I have acquired much knowledge about the subject of environmental modeling, cellular automata and their application. I improved my programming skill in C using CUDA. Also I gained some skills in using new libraries such as OpenGL. Personally, I learned how to search for information, organize my ideas, write well organized code, face problems and never surrender.

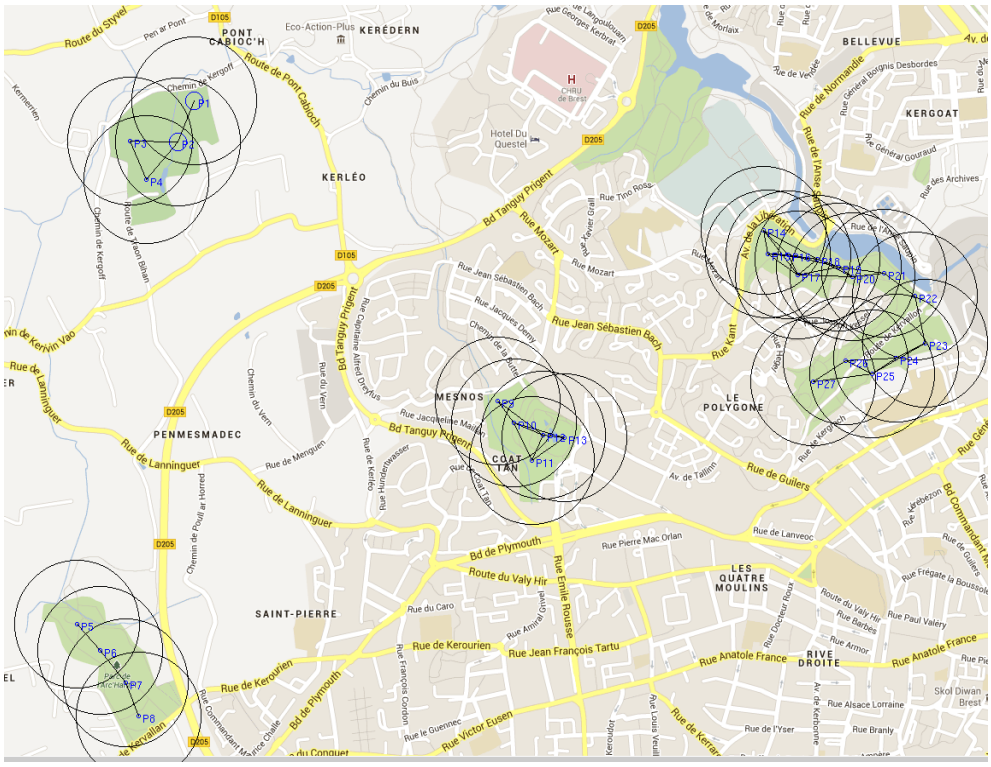

Figure 5.2: NetGen Simulation Brest City

List of Figures

Bibliography

[1] R.A. Letcher and A.J. Jakema. *Types of Environmental Mode*, Center for Resource and Environmental Studies, The Australian National University, Australia.

[2] Francesco Berto and Jacopo Tagliabue. *Cellular Automata*, 26 Mars 2012, http://plato.stanford.edu/entries/cellular-automata/.

[3] Florent Arrignon. *Intégration des donnés environnementales de l,observation à l'outil de modélisation*, 26 octobre 2012, Presentation by MAD-Environnement.

[4] OpenCl Website. http://www.khronos.org/opencl/

[5] Cellular Automata for Electric Power simulation project. 2004, http://www.cs.sjsu.edu/faculty/rucker/capow/intro.html

[6] Michael J Young. *Typical Uses of Cellular Automata*, 12 November 2006 http://www.mjyonline.com/CellularAutomataUses.html

[7] Wolfram, S. *Cellular Automata*, Los Alamos Science, Volume 9, Fall 1983, 2-21.

[8] COmmon-pool Resources and Multi-Agent Simulations (CORMAS) CIRAD research center, http://cormas.cirad.fr/.

[9] Wiki books website http://en.wikibooks.org/wiki/Cellular_Automata/Applications_of_Cellular_Automata.

[10] CUDA official website. *NVIDIA CUDA Getting Started Guide for Linux* , http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/index.html.

[11] Wikipedia, http://en.wikipedia.org/wiki/Row-major_order.

[12] Hritam Dutta, Thibault Failler, Nicolas Melot, Bernard Pottier and Serge Stinckwich. *An execution flow for dynamic concurrent systems: simulation of WSN on a Smalltalk/CUDA environment.*, *DYROS, in SIMPAR10, Darmstadt, Novembre 2010. ISBN 978-3-00-032-863*

[13] Adnan Iqbal et Bernard Pottier, *Meta-Simulation of Large WSN on Multi-core Computers*, *DEVS 2010, SimSpring 2010, Apr 2010, Orlando (ACM/SCS)*,

[14] Bernard Pottier et Pierre-Yves Lucas, *Concevoir, simuler, exécuter. Une chaîne de développement pour réseau de capteurs*, *UBIMOB 2012, CEPADUES, UPPA*.

Appendix A

Machine Specification

This appendix decribe the machine we are working on with all the specification of the GPU.

```
CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce␣GTX␣680"
  CUDA Driver Version / Runtime Version          5.0 / 5.0
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 4096 MBytes (4294639616
      bytes)
  ( 8) Multiprocessors x (192) CUDA Cores/MP:    1536 CUDA Cores
  GPU Clock rate:                                1137 MHz (1.14 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D
      =(65536,65536), 3D=(4096,4096,4096)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D
      =(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
         simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 5.0, CUDA Runtime
    Version = 5.0, NumDevs = 1, Device0 = GeForce GTX 680
```

Appendix B

Forest Fire Model Version 1

This appendix contain all the code of the Forest Fire model that sends a full forest to the GPU. The header file code .h is as follows

```
//FireSimulation
//Two inputs ways, by reading an image or by randomly iniitialize an input
#include "FireSimulationV1.h"


struct image *img,*img_d, *imgTemp,*imgTemp_d;

size_t size_image1;

int fpsCount = 0;        // FPS count for averaging
int fpsLimit = 2;        // FPS limit for sampling
int g_Index = 0;
unsigned int frameCount = 0;
StopWatchInterface *timer = NULL;

bool g_pause = false;
bool g_singleStep = false;
bool g_gpu = true;

unsigned char* data ; // pixel Array of image

int main ( int argc, char **argv ){


            size_t size_struct;
            img=(image *)malloc(sizeof(struct image));
            bool success;
            int inputMethod;
            success=false;


            printf("Fire␣Diffusion␣Simulator\n\n");
                        printf("\nPlease␣select␣input␣method");
                        printf("\n1.␣Random␣Generation");
                        printf("\n2.␣Read␣a␣forest␣image\n");
                        scanf("%d",&inputMethod);

                        if(inputMethod == 1)
                                    success = randomGeneration(img);
                        else if(inputMethod == 2)
                                    success = readBMP(img);
                        else
                                return 0;
```

```
                    if(success){

      printf("We have four states, empty (0 - White), tree (1 - Green),
          fire (2 - Red), ash (3 - Blue)\n");



                    //size of an struct
      size_struct = sizeof(struct image);

                    //allocate memory for the image struct variables
                    img_d=    (image *)malloc(size_struct);
                    imgTemp = (image *)malloc(size_struct);
                    imgTemp_d=(image *)malloc(size_struct);

                    printf("w: %d, h:%d\n",img->width,img->height);
      size_image1 = img->width *img->height  * sizeof(int);

            //allocate memory for the new forest state at time t+1
                    imgTemp->gridCellState=(int *)malloc(size_image1);

          //allocation on the device
                //allocate memory for the forest states at time t;
                    cudaMalloc(&img_d->gridCellState,size_image1);
                    //allocate memory for the new forest state at time
                        t+1
                    cudaMalloc(&imgTemp_d->gridCellState,size_image1);


                    sdkStartTimer(&timer);

            glutInit(&argc, argv);
            glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_ALPHA |
                GLUT_DEPTH);
            glutInitWindowSize(img->width, img->height);
            glutCreateWindow("Fire automata");
            glutDisplayFunc(display);
            glutKeyboardFunc(keyboard);
            glClearColor(0, 0, 0, 1.0);
            generateFire(img,FIREDTREES);
                    glutMainLoop();

            sdkDeleteTimer(&timer);

                    //for(int i=0; i<10;i++)
                      //  run();

      //printf("\nAll steps finished\n");
                    cudaFree(img_d->gridCellState);
                    cudaFree(imgTemp_d->gridCellState);
                    free(imgTemp->gridCellState);
                    free(img_d);
                    free(imgTemp);
                    free(imgTemp_d);
                    }
                    free(img->gridCellState);
                    free(img);
        return 0;
}


//Function to execute the simulation, called by display
void runSimulation()
{
  int nBlocks = (img->width*img->height)/BLSIZE + ((img->width*img->height)
      %BLSIZE==0?0:1);
        //copy data from host to device
```

```
        cudaMemcpy(img_d->gridCellState,img->gridCellState,size_image1,
            cudaMemcpyHostToDevice);
    transitionFunc<<<nBlocks,BLSIZE>>>(img_d->gridCellState, imgTemp_d->
        gridCellState, img->width, img->height);
        //copy data back from device to host
        cudaMemcpy(imgTemp->gridCellState,imgTemp_d->gridCellState,
            size_image1,cudaMemcpyDeviceToHost);
        //store the new state in grid                                          112
    swapGrids(img->gridCellState, imgTemp->gridCellState);
}


                                                                             117
void display()
{

    sdkStartTimer(&timer);
        glMatrixMode(GL_PROJECTION);    /* specifies the current matrix */   122
        glLoadIdentity();                       /* Sets the currant matrix
            to identity */
        gluOrtho2D(0,img->width,0,img->height); /* Sets the clipping
            rectangle extends */

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
                                                                             127
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glEnable(GL_BLEND); //enable the blending
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
                                                                             132
        glColor3f(1,1,1);
        glPointSize (1);

        glBegin(GL_POINTS);
                                                                             137
        for(int i=0;i<img->height;i++){
                for(int j=0;j<img->width;j++){
                  int cell = img->gridCellState[i * img->width + j];
                  int idx = (((i*img->width)+(j)) * 3 ) ;
                        if(cell == 0){                                       142
                          //if the cell  is 0 than ground color white
        //glColor3f(1.0f,1.0f,1.0f); // White
        glColor3f(((float) data[idx + 2])/255.0, ((float) data[idx + 1]) /
            255.0 , ((float) data[idx]) / 255.0);
        //glColor3f(0.0f, 0.0f,0.0f);
        glVertex3f(j,i,0);                                                   147
      }
                        else if(cell == 1) {
                          //if the cell  is 1 than tree color tree

        glColor3f(0.0f,1.0f,0.0f); // Green                                  152
        glVertex3f(j,i,0);
                }
      else if(cell == 2){
        //if the cell is 2 than fire color red
        glColor3f(1.0f,0.0f,0.0f); // Red                                   157
                                glVertex3f(j,i,0);
                        }
      else if(cell == 3) {
        //if cell is 3 than qsh colored gray
        //glColor3f(0.6f,0.6f,0.6f); // Gray                               162
        //glColor3f(1.0f,1.0f,1.0f);
        glColor3f(1.0f,1.0f,1.0f); // White
                          glVertex3f(j,i,0);
                  }
                }                                                            167
        }
```

```
        glEnd();
        glutSwapBuffers();
        glutPostRedisplay();
        glFlush ();
        if (!g_pause || g_singleStep)
        {
                if (g_gpu)
                        runSimulation();

                g_singleStep = false;
        }
        sdkStopTimer(&timer);
        computeFPS();
}


void generateFire(struct image *img, int firedTreeCount)
{

        //After the first step, a  random selected cell is fired
    srand(time(NULL));int random;
    for(int i=0; i<firedTreeCount; i++){
                random = rand()%((img->height * img->width) -1);
                img->gridCellState[random]=2;
   }
}

//Function for transiton rule
__global__ void transitionFunc(int *grid, int *bufferGrid, int width, int
     height){
        //check the cell's 4 neighbors
        //Northj, south, east, west
        int cellState;
        int north, east, south, west;
        int i;
        i= blockIdx.x * blockDim.x + threadIdx.x;
        if(i<height*width){
           //Calculating the new states
                if (grid[i] == 0)
                  cellState = 0;
                else if (grid[i] == 1)
                {

           //The north neighbours of the first row are the one in the last
               row, else the row above
           north= ((i-width)<0) ?  (height*width) - width  + i : i-width;
           //south neighbour of the last row are the first row, esle the
               row below
           south= (i+width) >= (height*width) ? ((i+width)%(height*width))
                : i+width;
           //here we need to check if the east neighbour doesnot exceed the
                right border, if so its neighbour
           //is the leftmost element in he same row
           //to get in which row we are, we use (i+width)/width
           east = ((i+1)>=(((i/width)*width)+width)) ? ((i)/width)*width:i
               +1;
           //west
           west = (i-1)<((i/width)*width) ?((i/width)*width)+width-1 : i-1;


                        if (   grid[north] == 2 || grid[south] == 2
                                        || grid[east]  == 2 || grid[west]
                                             == 2 )
                        {
                                cellState = 2 ;
                        }
                        else
```

```
                        {
                                cellState = 1 ;
                        }
                }
                else if (grid[i] == 2) cellState = 3 ;
                else cellState = 3;

                //Store the new state of time t+1 in bufferGrid
                bufferGrid[i] = cellState;
        }
}

//swap two grids
void swapGrids(int *&grid, int *&bufferGrid){
        int  *temp;
        temp = grid;
        grid = bufferGrid;
        bufferGrid= temp;
}
/*******************************************Initialization on the GPU
    **************************/
bool initializationOnGPU(struct image *img)
{
        printf("Initializing...\n");
        int size_image = (img->height * img->width)    * sizeof(int);
        int size_devstate = (img->height * img->width) * sizeof(
            curandState );
        curandState* devStates;
        int * temp;
        int err_devState = cudaMalloc ( &devStates,size_devstate);
        int err_temp = cudaMalloc(&temp,size_image);
    printf("err_devState:␣%dÂ \n",err_devState);
        if(!err_devState && !err_temp){
                        //the number of threads within a block
                int nBlocks = ((img->width*img->height)/BLSIZE)+ (((img->
                    width*img->height)%BLSIZE)==0?0:1);
                //generate the seeds
                setup_kernel<<<nBlocks,BLSIZE>>>(devStates, time(NULL),img
                    ->width, img->height);
                //Inialize the image grid
                initializeArray<<<nBlocks,BLSIZE>>>(devStates, temp,PROB,
                    img->width, img->height);
                cudaMemcpy(img->gridCellState,temp,size_image,
                    cudaMemcpyDeviceToHost);
                cudaFree(temp);
                cudaFree(devStates);
                printf("Initialization␣Successfully....\n");
                return true;
        }
        else{
                printf("Error␣memory␣Allocation");
                return false;
        }

}
__global__ void setup_kernel ( curandState * state, unsigned long seed,int
    width, int height )
{
     int j = blockIdx.x * blockDim.x + threadIdx.x;
    if(j< width* height)
        curand_init ( seed, j, 0, &state[j] );
}

__global__ void initializeArray( curandState* globalState,int *grid, float
    prob, int width, int height )
{
    int j = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    int val;
    if(j<height * width){
                curandState localState = globalState[j];
                float RANDOM = curand_uniform( &localState );
                globalState[j] = localState;                                    292
                if(RANDOM <= prob)   //Probatility to have tree is 0.58
                                val = 1;
                else
                                val = 0;
                grid[j]=val;                                                   297
    }
}
/*********************************************Initialization on the CPU
    ************************/

void initializeArray(struct image *img){                                       302
       //fill the array with random values, 0 inicate empty and 1 indiate
           tree
        int random=0;
        srand(time(NULL));
        for(int i=0;i<img->height;i++){
                for(int j=0;j<img->width;j++){                                 307
                        // The initial state of each cell of the spatial
                            grid is either set to #tree with
                        //a probability p or to #empty with a probability
                            1-p.
                        float val = (float)rand() / RAND_MAX;
                        if(val <= PROB)   //Probatility to have tree is
                            0.58
                                random = 1;                                    312
                        else
                                random = 0;
                        img->gridCellState[i*img->width + j]=random;

                }                                                              317
        }
}

/*************************************************Input Methods
    ***************************************/
//Function that ask the user to enter the path of image and initialize it     322
    on
//based on a reference number and threshold value,
//the reference number is used to detect the green color in the image, i.e
    to detect the trees
//it then stores the state of the image in img
bool readBMP(struct image *img)
{                                                                              327
        char fileName[100];
        printf("\nPlease␣enter␣file␣name␣.bmp:\n");
        scanf("%s",fileName);

        FILE* file = fopen(fileName, "rb");                                    332
        if(file==0)
        {
          printf("\nCould␣not␣open␣the␣file");
          return false;
        }                                                                      337
        else
        {
        unsigned char info[54];
        int distance;
        int state=0;                                                           342
        int width, height;
        fread(info, sizeof(unsigned char), 54, file); // read the 54-byte
            header
        // extract image height and width from header
```

```
        width = *(int*)&info[18];
        height = *(int*)&info[22];

        printf("\nwidth:%d",width);
        printf("\nheight:%d",height);

        //store the width
        img->width= width;
        img->height=height;
        //allocation on the host
        //allocate memory for the forest states at time t;
        img->gridCellState = (int *) malloc ((img->width * img->height) *
            sizeof(int));

        //3 bytes per pixel
        int size = 3 * img->width * img->height;
        //unsigned char* data = (unsigned char*)malloc(size); // allocate 3
             bytes per pixel
        data = (unsigned char*)malloc(size);
        fread(data, sizeof(unsigned char), size, file); // read the rest of
             the data at once
        fclose(file);

        //detect color based on ecidian distance;

        for(int i =0 ; i<size; i+=3){//height
                //Red indeed at i+2. Green indeed at i+1, Blue indeed at i,
                     the order in the file is B, G , Red
                distance = sqrt(pow((data[i+2] - RED_REF),2) + pow((data[i
                    +1] - GREEN_REF),2) + pow((data [i] - BLUE_REF),2));
                state=0;
                if(distance < THRESHOLD)
                {
                                if (((double)rand() / (double)RAND_MAX) >
                                    0.3)
                        state=1;
                }
                else state=0;
                //i is 3 times more
                img->gridCellState[(i/3)]=state;
        }
return true;
}
}

//Random generation of a forest
bool randomGeneration(struct image *img)
{
        int width, height;
        bool succ;
        printf("\nEnter␣width:␣");
        scanf("%d",&width);
        printf("\nEnter␣height:␣");
        scanf("%d",&height);
        img->width= width;
        img->height=height;
        //allocation on the host
        //allocate memory for the forest states at time t;
        img->gridCellState = (int *) malloc ((img->width * img->height) *
            sizeof(int));
        succ =true;
        initializeArray(img);
        //succ = initializationOnGPU(img);
        if(succ)
                return true;
        else
                return false;
```

```
}
                                                                                407

/*********************************************************************OpenGL
     Display Helper *********************/
void computeFPS()
{
    frameCount++;                                                               412
    fpsCount++;

        if (fpsCount == fpsLimit) {
        char fps[256];
                                                                                417

        float ifps = 1.f / (sdkGetAverageTimerValue(&timer) / 1000.f);
        sprintf(fps, "Forest␣Fire␣Automate␣Simulation,␣By␣AHMED␣Ahmed,␣
            Supervised␣by␣Prof.Bernard␣Pottier:␣%3.1f␣fps␣%d␣generations",
            ifps, frameCount);

        glutSetWindowTitle(fps);                                                422
        fpsCount = 0;
        //if (g_CheckRender && !g_CheckRender->IsQAReadback()) fpsLimit = (
            int)MAX(ifps, 1.f);

        //checkCudaErrors(cutResetTimer(timer));
        sdkResetTimer(&timer);                                                  427
        //AutoQATest();
    }
 }
                                                                                432

void keyboard(unsigned char key, int x, int y)
{

        if (key=='␣')                                                           437
        {
                g_pause = !g_pause;
        }

        if (key=='s')                                                           442
        {
                g_pause = true;
                g_singleStep = true;
        }
        display();                                                              447
}


/*************************************************************Printing
    helpers ********************************/
//print the data of gridCellState in image                                      452
void dumpArray(struct image *img){
        printf("\n**************************************\n");
        //print contents of the array
        for(int i=0;i<img->height;i++){
                for(int j=0;j<img->width;j++){                                  457
                        printf("%d␣",img->gridCellState[i * img->width + j
                            ]);
                }
                printf("\n");
        }
}                                                                               462

////print the data of gridCellState in image, with qa color identifying
    each state
void dumpArrayColor(struct image *img){
```

```
        printf("\n****************************************\n");
        //print contents of the array                                   467
        int *grid = img->gridCellState;
        int width = img->width;
        for(int i=0;i<img->height;i++){
                for(int j=0;j<img->width;j++){
                                                                        472
                        if(grid[i * width + j] == 0) printf("\e[1;0m%d␣",
                            grid[i * width + j]);
                        else if(grid[i * width + j] == 1) printf("\e[1;32m%
                            d␣",grid[i * width + j]);
                        else if(grid[i * width + j] == 2) printf("\e[1;31m%
                            d␣",grid[i * width + j]);
                        else if(grid[i * width + j] == 3) printf("\e[1;34m%
                            d␣",grid[i * width + j]);
                }                                                       477
                printf("\e[1;0m\n");
        }
}

/*********************************************************Delay helper   482
    *********************************/
//Delay between the outputs
void wait ( int seconds ){
        clock_t endwait;
        endwait = clock () + seconds * CLOCKS_PER_SEC ;
        while (clock() < endwait) {}                                    487
}
```

The principle code .cu is given is the following part.

```
//FireSimulation                                                        1
//Two inputs ways, by reading an image or by randomly iniitialize an input
#include "FireSimulationV1.h"


struct image *img,*img_d, *imgTemp,*imgTemp_d;                          6

size_t size_image1;

int fpsCount = 0;        // FPS count for averaging
int fpsLimit = 2;        // FPS limit for sampling                      11
int g_Index = 0;
unsigned int frameCount = 0;
StopWatchInterface *timer = NULL;

bool g_pause = false;                                                   16
bool g_singleStep = false;
bool g_gpu = true;

unsigned char* data ; // pixel Array of image
                                                                        21
int main ( int argc, char **argv ){


            size_t size_struct;
            img=(image *)malloc(sizeof(struct image));                  26
            bool success;
            int inputMethod;
            success=false;

                                                                        31
            printf("Fire␣Diffusion␣Simulator\n\n");
                        printf("\nPlease␣select␣input␣method");
                        printf("\n1.␣Random␣Generation");
                        printf("\n2.␣Read␣a␣forest␣image\n");
                        scanf("%d",&inputMethod);                       36
```

```
                    if(inputMethod == 1)
                            success = randomGeneration(img);
                    else if(inputMethod == 2)
                            success = readBMP(img);
                    else
                           return 0;

                    if(success){

printf("We have four states, empty (0 - White), tree (1 - Green),
    fire (2 - Red), ash (3 - Blue)\n");



                    //size of an struct
size_struct = sizeof(struct image);

                    //allocate memory for the image struct variables
                    img_d=    (image *)malloc(size_struct);
                    imgTemp = (image *)malloc(size_struct);
                    imgTemp_d=(image *)malloc(size_struct);

                    printf("w: %d, h:%d\n",img->width,img->height);
size_image1 = img->width *img->height  * sizeof(int);

      //allocate memory for the new forest state at time t+1
                    imgTemp->gridCellState=(int *)malloc(size_image1);

    //allocation on the device
          //allocate memory for the forest states at time t;
                    cudaMalloc(&img_d->gridCellState,size_image1);
                    //allocate memory for the new forest state at time
                        t+1
                    cudaMalloc(&imgTemp_d->gridCellState,size_image1);


                    sdkStartTimer(&timer);

      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_ALPHA |
          GLUT_DEPTH);
      glutInitWindowSize(img->width, img->height);
      glutCreateWindow("Fire automata");
      glutDisplayFunc(display);
      glutKeyboardFunc(keyboard);
      glClearColor(0, 0, 0, 1.0);
      generateFire(img,FIREDTREES);
                    glutMainLoop();

      sdkDeleteTimer(&timer);

                    //for(int i=0; i<10;i++)
                      //  run();

//printf("\nAll steps finished\n");
                    cudaFree(img_d->gridCellState);
                    cudaFree(imgTemp_d->gridCellState);
                    free(imgTemp->gridCellState);
                    free(img_d);
                    free(imgTemp);
                    free(imgTemp_d);
                    }
                    free(img->gridCellState);
                    free(img);
  return 0;
}
```

```
//Function to execute the simulation, called by display
void runSimulation()
{
  int nBlocks = (img->width*img->height)/BLSIZE + ((img->width*img->height)
      %BLSIZE==0?0:1);
        //copy data from host to device
        cudaMemcpy(img_d->gridCellState,img->gridCellState,size_image1,
            cudaMemcpyHostToDevice);
  transitionFunc<<<nBlocks,BLSIZE>>>(img_d->gridCellState, imgTemp_d->
      gridCellState, img->width, img->height);
        //copy data back from device to host
        cudaMemcpy(imgTemp->gridCellState,imgTemp_d->gridCellState,
            size_image1,cudaMemcpyDeviceToHost);
        //store the new state in grid
  swapGrids(img->gridCellState, imgTemp->gridCellState);
}


void display()
{

  sdkStartTimer(&timer);
        glMatrixMode(GL_PROJECTION);    /* specifies the current matrix */
        glLoadIdentity();                       /* Sets the currant matrix
            to identity */
        gluOrtho2D(0,img->width,0,img->height); /* Sets the clipping
            rectangle extends */

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glEnable(GL_BLEND); //enable the blending
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

        glColor3f(1,1,1);
        glPointSize (1);

        glBegin(GL_POINTS);

        for(int i=0;i<img->height;i++){
                for(int j=0;j<img->width;j++){
                  int cell = img->gridCellState[i * img->width + j];
                  int idx = (((i*img->width)+(j)) * 3 ) ;
                        if(cell == 0){
                          //if the cell  is 0 than ground color white
        //glColor3f(1.0f,1.0f,1.0f); // White
        glColor3f(((float) data[idx + 2])/255.0, ((float) data[idx + 1]) /
            255.0 , ((float) data[idx]) / 255.0);
        //glColor3f(0.0f, 0.0f,0.0f);
        glVertex3f(j,i,0);
      }
                        else if(cell == 1) {
                          //if the cell  is 1 than tree color tree

        glColor3f(0.0f,1.0f,0.0f); // Green
        glVertex3f(j,i,0);
                }
      else if(cell == 2){
        //if the cell is 2 than fire color red
        glColor3f(1.0f,0.0f,0.0f); // Red
                                glVertex3f(j,i,0);
                        }
      else if(cell == 3) {
        //if cell is 3 than qsh colored gray
```

```
        //glColor3f(0.6f,0.6f,0.6f); // Gray
        //glColor3f(1.0f,1.0f,1.0f);
        glColor3f(1.0f,1.0f,1.0f); // White
                        glVertex3f(j,i,0);
                  }                                                          166
                }
        }
        glEnd();
        glutSwapBuffers();
        glutPostRedisplay();                                                 171
        glFlush ();
        if (!g_pause || g_singleStep)
        {
                if (g_gpu)
                        runSimulation();                                     176

                g_singleStep = false;
        }
        sdkStopTimer(&timer);
        computeFPS();                                                        181
}


void generateFire(struct image *img, int firedTreeCount)
{                                                                            186

        //After the first step, a  random selected cell is fired
    srand(time(NULL));int random;
    for(int i=0; i<firedTreeCount; i++){
                 random = rand()%((img->height * img->width) -1);            191
                 img->gridCellState[random]=2;
   }
}

//Function for transiton rule                                                196
__global__ void transitionFunc(int *grid, int *bufferGrid, int width, int
    height){
        //check the cell's 4 neighbors
        //Northj, south, east, west
        int cellState;
        int north, east, south, west;                                        201
        int i;
        i= blockIdx.x * blockDim.x + threadIdx.x;
        if(i<height*width){
           //Calculating the new states
                if (grid[i] == 0)                                            206
                  cellState = 0;
                else if (grid[i] == 1)
                {

           //The north neighbours of the first row are the one in the last   211
               row, else the row above
           north= ((i-width)<0) ?  (height*width) - width  + i : i-width;
           //south neighbour of the last row are the first row, esle the
               row below
           south= (i+width) >= (height*width) ? ((i+width)%(height*width))
                : i+width;
           //here we need to check if the east neighbour doesnot exceed the
                right border, if so its neighbour
           //is the leftmost element in he same row                          216
           //to get in which row we are, we use (i+width)/width
           east = ((i+1)>=(((i/width)*width)+width)) ? ((i)/width)*width:i
               +1;
           //west
           west = (i-1)<((i/width)*width) ?((i/width)*width)+width-1 : i-1;
                                                                             221
```

```
                        if (   grid[north] == 2 || grid[south] == 2
                                        || grid[east]  == 2 || grid[west]
                                            == 2 )
                        {
                                cellState = 2 ;                                 226
                        }
                        else
                        {
                                cellState = 1 ;
                        }                                                       231
                }
                else if (grid[i] == 2) cellState = 3 ;
                else cellState = 3;

                //Store the new state of time t+1 in bufferGrid                 236
                bufferGrid[i] = cellState;
        }
}

//swap two grids                                                                241
void swapGrids(int *&grid, int *&bufferGrid){
        int  *temp;
        temp = grid;
        grid = bufferGrid;
        bufferGrid= temp;                                                       246
}
/*******************************************Initialization on the GPU
    *************************/
bool initializationOnGPU(struct image *img)
{
        printf("Initializing...\n");                                            251
        int size_image = (img->height * img->width)    * sizeof(int);
        int size_devstate = (img->height * img->width) * sizeof(
            curandState );
        curandState* devStates;
        int * temp;
        int err_devState = cudaMalloc ( &devStates,size_devstate);              256
        int err_temp = cudaMalloc(&temp,size_image);
    printf("err_devState:␣%dÂ \n",err_devState);
        if(!err_devState && !err_temp){
                        //the number of threads within a block
                int nBlocks = ((img->width*img->height)/BLSIZE)+ (((img->       261
                    width*img->height)%BLSIZE)==0?0:1);
                //generate the seeds
                setup_kernel<<<nBlocks,BLSIZE>>>(devStates, time(NULL),img
                    ->width, img->height);
                //Inialize the image grid
                initializeArray<<<nBlocks,BLSIZE>>>(devStates, temp,PROB,
                    img->width, img->height);
                cudaMemcpy(img->gridCellState,temp,size_image,                  266
                    cudaMemcpyDeviceToHost);
                cudaFree(temp);
                cudaFree(devStates);
                printf("Initialization␣Successfully....\n");
                return true;
        }                                                                       271
        else{
                printf("Error␣memory␣Allocation");
                return false;
        }
                                                                                276
}
__global__ void setup_kernel ( curandState * state, unsigned long seed,int
    width, int height )
{
     int j = blockIdx.x * blockDim.x + threadIdx.x;
    if(j< width* height)                                                        281
```

```
        curand_init ( seed, j, 0, &state[j] );
}

__global__ void initializeArray( curandState* globalState,int *grid, float
    prob, int width, int height )
{
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int val;
    if(j<height * width){
                curandState localState = globalState[j];
                float RANDOM = curand_uniform( &localState );
                globalState[j] = localState;
                if(RANDOM <= prob)   //Probatility to have tree is 0.58
                                val = 1;
                else
                                val = 0;
                grid[j]=val;
    }
}
/*******************************************Initialization on the CPU
    *************************/

void initializeArray(struct image *img){
       //fill the array with random values, 0 inicate empty and 1 indiate
           tree
        int random=0;
        srand(time(NULL));
        for(int i=0;i<img->height;i++){
                for(int j=0;j<img->width;j++){
                        // The initial state of each cell of the spatial
                            grid is either set to #tree with
                        //a probability p or to #empty with a probability
                            1-p.
                        float val = (float)rand() / RAND_MAX;
                        if(val <= PROB)   //Probatility to have tree is
                            0.58
                                random = 1;
                        else
                                random = 0;
                        img->gridCellState[i*img->width + j]=random;

                }
        }
}

/************************************************Input Methods
    ************************************/
//Function that ask the user to enter the path of image and initialize it
    on
//based on a reference number and threshold value,
//the reference number is used to detect the green color in the image, i.e
    to detect the trees
//it then stores the state of the image in img
bool readBMP(struct image *img)
{
        char fileName[100];
        printf("\nPlease␣enter␣file␣name␣.bmp:\n");
        scanf("%s",fileName);

        FILE* file = fopen(fileName, "rb");
        if(file==0)
        {
          printf("\nCould␣not␣open␣the␣file");
          return false;
        }
        else
        {
```

```
        unsigned char info[54];
        int distance;
        int state=0;
        int width, height;
        fread(info, sizeof(unsigned char), 54, file); // read the 54-byte
            header
        // extract image height and width from header
        width = *(int*)&info[18];
        height = *(int*)&info[22];

        printf("\nwidth:%d",width);
        printf("\nheight:%d",height);

        //store the width
        img->width= width;
        img->height=height;
        //allocation on the host
        //allocate memory for the forest states at time t;
        img->gridCellState = (int *) malloc ((img->width * img->height) *
            sizeof(int));

        //3 bytes per pixel
        int size = 3 * img->width * img->height;
        //unsigned char* data = (unsigned char*)malloc(size); // allocate 3
             bytes per pixel
        data = (unsigned char*)malloc(size);
        fread(data, sizeof(unsigned char), size, file); // read the rest of
             the data at once
        fclose(file);

        //detect color based on ecidian distance;

        for(int i =0 ; i<size; i+=3){//height
                //Red indeed at i+2. Green indeed at i+1, Blue indeed at i,
                     the order in the file is B, G , Red
                distance = sqrt(pow((data[i+2] - RED_REF),2) + pow((data[i
                    +1] - GREEN_REF),2) + pow((data [i] - BLUE_REF),2));
                state=0;
                if(distance < THRESHOLD)
                {
                                if (((double)rand() / (double)RAND_MAX) >
                                    0.3)
                        state=1;
                }
                else state=0;
                //i is 3 times more
                img->gridCellState[(i/3)]=state;
        }
return true;
}
}

//Random generation of a forest
bool randomGeneration(struct image *img)
{
        int width, height;
        bool succ;
        printf("\nEnter␣width:␣");
        scanf("%d",&width);
        printf("\nEnter␣height:␣");
        scanf("%d",&height);
        img->width= width;
        img->height=height;
        //allocation on the host
        //allocate memory for the forest states at time t;
        img->gridCellState = (int *) malloc ((img->width * img->height) *
            sizeof(int));
```

```
        succ =true;
        initializeArray(img);
        //succ = initializationOnGPU(img);                                  401
        if(succ)
                return true;
        else
                return false;
}                                                                           406


/*****************************************************************OpenGL
     Display Helper *********************/
void computeFPS()
{                                                                           411
    frameCount++;
    fpsCount++;

        if (fpsCount == fpsLimit) {
        char fps[256];                                                      416


        float ifps = 1.f / (sdkGetAverageTimerValue(&timer) / 1000.f);
        sprintf(fps, "Forest␣Fire␣Automate␣Simulation,␣By␣AHMED␣Ahmed,␣
            Supervised␣by␣Prof.Bernard␣Pottier:␣%3.1f␣fps␣%d␣generations",
            ifps, frameCount);
                                                                            421
        glutSetWindowTitle(fps);
        fpsCount = 0;
        //if (g_CheckRender && !g_CheckRender->IsQAReadback()) fpsLimit = (
            int)MAX(ifps, 1.f);

        //checkCudaErrors(cutResetTimer(timer));                            426
        sdkResetTimer(&timer);
        //AutoQATest();
    }
 }
                                                                            431


void keyboard(unsigned char key, int x, int y)
{
                                                                            436
        if (key=='␣')
        {
                g_pause = !g_pause;
        }
                                                                            441
        if (key=='s')
        {
                g_pause = true;
                g_singleStep = true;
        }                                                                   446
        display();
}


/*************************************************************Printing      451
    helpers *******************************/
//print the data of gridCellState in image
void dumpArray(struct image *img){
        printf("\n**************************************\n");
        //print contents of the array
        for(int i=0;i<img->height;i++){                                     456
                for(int j=0;j<img->width;j++){
                        printf("%d␣",img->gridCellState[i * img->width + j
                            ]);
                }
```

```
                printf("\n");
        }
}

////print the data of gridCellState in image, with qa color identifying
    each state
void dumpArrayColor(struct image *img){
        printf("\n**************************************\n");
        //print contents of the array
        int *grid = img->gridCellState;
        int width = img->width;
        for(int i=0;i<img->height;i++){
                for(int j=0;j<img->width;j++){

                        if(grid[i * width + j] == 0) printf("\e[1;0m%d␣",
                            grid[i * width + j]);
                        else if(grid[i * width + j] == 1) printf("\e[1;32m%
                            d␣",grid[i * width + j]);
                        else if(grid[i * width + j] == 2) printf("\e[1;31m%
                            d␣",grid[i * width + j]);
                        else if(grid[i * width + j] == 3) printf("\e[1;34m%
                            d␣",grid[i * width + j]);
                }
                printf("\e[1;0m\n");
        }
}

/************************************************************Delay helper
    **********************************/
//Delay between the outputs
void wait ( int seconds ){
        clock_t endwait;
        endwait = clock () + seconds * CLOCKS_PER_SEC ;
        while (clock() < endwait) {}
}
```

Appendix C

Forest Fire Model Version 2

This appendix contain the second version of forest model, which sends the forest slice by slice to the device and use pipeline approch. The header file code .h is as follows

```
// includes, system
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#ifdef _WIN32
#  define WINDOWS_LEAN_AND_MEAN
#  define NOMINMAX
#  include <windows.h>
#endif

// OpenGL Graphics includes
#include <GL/glew.h>
#if defined (__APPLE__) || defined(MACOSX)
#include <GLUT/glut.h>
#else
#include <GL/freeglut.h>
#endif

// includes, cuda
#include <cuda_runtime.h>
#include <cuda_gl_interop.h>

// Utilities and timing functions
#include <helper_functions.h>    // includes cuda.h and cuda_runtime_api.h
#include <timer.h>               // timing functions

// CUDA helper functions
#include <helper_cuda.h>         // helper functions for CUDA error check
#include <helper_cuda_gl.h>      // helper functions for CUDA/GL interop


#include <time.h>
#include <cuda.h>
#include <curand_kernel.h>


#include <sys/unistd.h>
#include <pthread.h>

#define FIREDTREES 4    //Numbers of trees to be fired randomly
#define PROB 0.58       //Probabilty for the nubmer of trees in the forest
    generation
```

```
//The initial number of slice,  i.e then umber of parts to which an image
    height will be divded
//it will be adaped according to the size of the image
#define SLICE 1024

//used for threads
#define THREADS 3
#define T1      0
#define T2      1
#define T3      2

//The number of thread per block
#define BLSIZE 1024

//Used for image color detection
//The tree reference color
#define RED_REF 65
#define GREEN_REF 75
#define BLUE_REF 65
//The threshold used in detection
#define THRESHOLD 30

struct image
{
        int width;   //to hold the number of rows of the grid
        int height;  //to hold the number of columns of the grid
        int *gridCellState;
  int offset; //the offest is used while sending an image part by part
};


//data for each thread
struct data_thread {
  int threadId;
  struct image  *dataIn, *dataOut;
};

struct data_thread  data[THREADS];


//Input methods
//random generation and read an image
bool randomGeneration(struct image *img);
bool readBMP(struct image *img);


//OpenGL funtion that calls the simulation function
void display();

//Initialize a fires,called by randomGeneration
void initializeArray(struct image *img);

//allocate and intialize the thread data
bool init_data_thread(struct data_thread *dt, int id);
//free the allocated memory space of data of the thread
void free_data(struct data_thread dt);
//thread calls
void * threadStep(void * args);
//Exchange of data between threads
void dataMove();

//Send data from Host to Device
void toGPU(struct image *tin,struct image *tout);
//Execute the kernel
void processOnGPU(struct image *tin,struct image *tout);
//send the data from Device to Host
```

```
void fromGPU(struct image *tin,struct image *tout);
//Random generaion of fire                                                  111
void generateFire(struct image *img, int firedTreeCount);

//Initialze to zero
__global__ void SetZero(int *tabi,int w, int h);
                                                                            116
//the transition function to be executed on the GPU, the new state is hold
//in bufferGrid depending on the vaues of the current state
//in grid
__global__ void transitionFunc(int *grid, int *bufferGrid,int width,int
    height);
                                                                            121
//copy the new state grid "bufferGrid" to the main grid "grid"
void swapGrids(int *&grid, int *&bufferGrid);

//print the grid without color
void dumpArray(struct image *grid);                                         126
//print te grid colored
void dumpArrayColor(struct image  *grid);
void dumpArrayColor(int *grid, int width,int height);

//Delay between the outputs                                                 131
void wait ( int seconds );

//functions used by openGl for the output
void computeFPS();
void keyboard(unsigned char key, int x, int y);                             136
```

The principle code .cu is given is the following part.

```
#include "FireSimulationV2.h"

int slice_num; void * status;int part; pthread_t * threads;                  3
void runSimulation();


int fpsCount = 0;        // FPS count for averaging
int fpsLimit = 2;        // FPS limit for sampling                           8
int g_Index = 0;
unsigned int frameCount = 0;
StopWatchInterface *timer = NULL;

bool g_pause = false;                                                       13
bool g_singleStep = false;
bool g_gpu = true;


int main (int argc, char **argv){                                           18

        int i,err;

        size_t size_struct;
                                                                            23
        pthread_attr_t attr;
        int inputMethod;


        printf("Fire␣Diffusion␣Simulator\n\n");                             28
        printf("\nPlease␣select␣input␣method");
        printf("\n1.␣Random␣Generation");
        printf("\n2.␣Read␣a␣forest␣image\n");
        scanf("%d",&inputMethod);
                                                                            33
        size_struct = sizeof(struct image);

        //allocation of memory to the image pointer that will hold the
            initial
```

```
//image
data[0].dataIn =(image *)malloc(size_struct);


if(inputMethod == 1)
        randomGeneration( data[0].dataIn);
else if(inputMethod == 2)
        readBMP( data[0].dataIn);
else{
        printf("Exting␣application");
        return 0;
}


printf("We␣have␣four␣states,␣empty␣(0␣-␣White),␣tree␣(1␣-␣Green),␣
    fire␣(2␣-Red),␣ash␣(3␣-␣Blue)\n");


//size of each image slice, the image slice is the width * (height/
    the number of slices possible),
part = (data[0].dataIn->width* (data[0].dataIn->height/SLICE));


for(i=0; i<THREADS;i++){
        //allocate memory for the image struct variables
        //data[0] is already allocated before while reading image
            or random forest generation
        if(i!=0){
                data[i].dataIn =(image *)malloc(size_struct);//img
                    on HOST that hold state at time t
                //store the width ang height values
                data[i].dataIn->width =  data[0].dataIn->width;
                data[i].dataIn->height = data[0].dataIn->height;
        }
        data[i].dataOut =         (image *)malloc(size_struct);
        //store the width ang height values
        data[i].dataOut->width =  data[0].dataIn->width;
        data[i].dataOut->height = data[0].dataIn->height;
}


//allocation for the threads
threads=(pthread_t *)malloc( sizeof(pthread_t)*THREADS);

//initializes the thread attributes
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

//initialize the threads and perfrom the required allocation
err =0;
for (i=0; i<THREADS;i++)
err |= init_data_thread(& data[i], i);
//If there is no errors in memory allocation
if(!err){
        //dumpArrayColor(data[0].dataIn);


        if(SLICE<=data[0].dataIn->height){
                slice_num =SLICE;


                //an image height is divided to SLICE, and send
                    part by part to the GPU.
                //if the height is not fivisible bythe number of
                    slice, then we need to
```

```
                        //add more slices... for example if the slices
                            number is 10 and the input image
                        //width and height are 16 and 13 respectively. Then
                             the number of slices to
                        //represent all the data is 13. Another example, if
                             the width=32 and height=38,
                        //then the number of slices requires is 10 + 3 =13.
                        while ((slice_num * part )  < (data[0].dataIn->
                            height * data[0].dataIn->width))
                                slice_num++;

            sdkStartTimer(&timer);

            glutInit(&argc, argv);
            glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_ALPHA |
                GLUT_DEPTH);
            glutInitWindowSize(data[0].dataIn->width, data[0].dataIn->
                height);
            glutCreateWindow("Fire␣automata");
            glutDisplayFunc(display);
            glutKeyboardFunc(keyboard);
            glClearColor(0, 0, 0, 1.0);

            generateFire(data[0].dataIn,FIREDTREES);

                        glutMainLoop();

            sdkDeleteTimer(&timer);

            //runSimulation();


                }
                else{
                        printf("SLICE␣size␣is␣smaller␣than␣height\nExiting␣
                            application...\n");
                }

                //Deallocation of memory
                for (i=0; i<THREADS;i++)
                        free_data(data[i]);

                for(int i=0; i<THREADS;i++){
                        //allocate memory for the image struct variables
                        free(data[i].dataIn) ;
                        free(data[i].dataOut) ;
                }
                free(threads);

                 //Explicitly destroys and cleans up all resources
                     associated with the current device.
                 //It will reset the device immediately.
                cudaDeviceReset();

        }
        return 0;
}


//Function to execute the simulation, called by display
void runSimulation()
{
 //As we have three threads, we need Two more steps to get the result
                            //of last slice, so +2
                            int i,j;
                            for ( j=0; j<slice_num+2; j++){
```

```
                                        //to not exceed the size of the
                                            array
                                        //in the last two steps the
                                            transfer to GPU
                                        //will be of no use                     158
                                        if(j<(slice_num+2))
                                                data[0].dataIn->offset =
                                                    part*j;
                                        //if we arrived the third thread,
                                            then it contains the offset of
                                            the first thread,
                                        if(j>=2){
                                                data[2].dataOut->offset =       163
                                                    part*(j-2);
                                        }

                                        //To create and execute threads
                                        for (i=0; i<THREADS;i++){
                                                pthread_create(&threads[i],     168
                                                    NULL,threadStep,(void
                                                    *)&(data[i]));
                                        }
                                        //if the threads doesnot finish, it
                                             suspend the execution until
                                        //the concerned thread finish
                                        for (i=0; i<THREADS; i++){
                                                pthread_join(threads[i],&       173
                                                    status);
                                                if(status);
                                                        //error
                                        }

                                        //transfer the output of the first    178
                                            thread to the input of second
                                            thread,
                                        //and transfer the output of the
                                            second thread to the input of
                                            first thread
                                        dataMove();
                                }
                                //swap the states of the image, i.e. put
                                    the input for the second loop
                                //the new state generated as result by the    183
                                    trransition function
                                swapGrids(data[0].dataIn->gridCellState,
                                    data[2].dataOut->gridCellState);
}

void display()
{                                                                               188

  sdkStartTimer(&timer);
        glMatrixMode(GL_PROJECTION);    /* specifies the current matrix */
        glLoadIdentity();                       /* Sets the currant matrix
            to identity */
        gluOrtho2D(0,data[0].dataIn->width,0,data[0].dataIn->height);   /*    193
            Sets the clipping rectangle extends */

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();                                                       198
        glEnable(GL_BLEND); //enable the blending
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

        glColor3f(1,1,1);
        glPointSize (1);                                                        203
```

```
        glBegin(GL_POINTS);

        for(int i=0;i<data[0].dataIn->height;i++){
                for(int j=0;j<data[0].dataIn->width;j++){
                  int cell = data[0].dataIn->gridCellState[i * data[0].
                      dataIn->width + j];
                    if(cell == 0){
                      //if the cell  is 0 than ground color white
        glColor3f(1.0f,1.0f,1.0f); // White
        glVertex3f(j,i,0);
      }
                    else if(cell == 1) {
                      //if the cell  is 1 than tree color tree
        glColor3f(0.0f,1.0f,0.0f); // Green
        glVertex3f(j,i,0);
                 }
      else if(cell == 2){
        //if the cell is 2 than fire color red
        glColor3f(1.0f,0.0f,0.0f); // Red
                                glVertex3f(j,i,0);
                    }
      else if(cell == 3) {
        //if cell is 3 than qsh colored gray
        glColor3f(0.6f,0.6f,0.6f); // Gray
                      glVertex3f(j,i,0);
                }
              }
        }
        glEnd();
        glutSwapBuffers();
        glutPostRedisplay();
        glFlush ();
        if (!g_pause || g_singleStep)
        {
                if (g_gpu)
                        runSimulation();

                g_singleStep = false;
        }
        sdkStopTimer(&timer);
        computeFPS();
}


//The functions that defines the functionality of each thread
void * threadStep(void * args)
{
        int id;
        struct data_thread *sdata;
        sdata = (struct data_thread *) args;
        id  =  sdata->threadId;
        switch (id) {
                case T1:
                        //Thread 1 is responsible to send the data from
                            Host to Device GPU
                        toGPU(sdata->dataIn,sdata->dataOut);
                        break;
                case T2:
                        //Thread 2 execute the kernel
                        processOnGPU(sdata->dataIn, sdata->dataOut);
                        break;

                case T3:
                        //Thread 3 send the data from the Device to Host
                        fromGPU(sdata->dataIn, sdata->dataOut);
                        break;
```

```
        }
        return 0;
}

//function to free the allocated memory
void free_data(struct data_thread dt)
{
        int id;
        id = dt . threadId;
        switch (id){
                case T1 :
                        cudaFree(dt.dataOut->gridCellState);
                        free(dt.dataIn->gridCellState);
                        break;
                case T2 :
                        cudaFree(dt.dataIn->gridCellState);
                        cudaFree(dt.dataOut->gridCellState);
                        break;
                case T3 :
                        cudaFree(dt.dataIn->gridCellState);
                        free(dt.dataOut->gridCellState);
                        break;
                }
}


void dataMove()
{
        int *tmp;
        //Exchange the data between the threads.
        //The input of the T2 is the output from T1
        tmp = data[0].dataOut->gridCellState;
        data[0].dataOut->gridCellState = data[1].dataIn->gridCellState;
        data[1].dataIn->gridCellState = tmp;

        //The input of the T3 is the output from T2
        tmp = data[1].dataOut->gridCellState;
        data[1].dataOut->gridCellState = data[2].dataIn->gridCellState;
        data[2].dataIn->gridCellState = tmp;
}

//Initilaize the thread data
//return true in case of allocation failure
bool init_data_thread(struct data_thread *dt, int id)
{
        int size_image, part,partWithNeighbours,size_image_part;
        int i,err;
        //size of a full image, used on host
        size_image = (dt->dataIn->width*dt->dataIn->height)*sizeof(int);
        //part is the dimension size of a part
        part = (dt->dataIn->width*(dt->dataIn->height/SLICE));
        //Each slice has two more rows the directly upper and directly
            before the slice
        partWithNeighbours = part + (2 * dt->dataIn->width);
        //size of memory required for a part
        size_image_part = partWithNeighbours * sizeof(int);

        err= 0;
        dt->threadId = id;
        dt->dataIn->offset =0;
        dt->dataOut->offset =0;

        //number of blocks required for a part i.e. a slice without
            neightbors
        int nBlocks = (dt->dataIn->width*(dt->dataIn->height/SLICE))/BLSIZE
            + ((dt->dataIn->width*(dt->dataIn->height/SLICE))%BLSIZE
```

```
            ==0?0:1);
        //number of blocks reuqured for a part i.e. a slice with two rows
            for north and south neighbors
int nBlocks1 = (dt->dataIn->width*((dt->dataIn->height/SLICE)+2))/BLSIZE+
    ((dt->dataIn->width*((dt->dataIn->height/SLICE)+2))%BLSIZE==0?0:1);

        switch (id){
                case T1:
                        //Allocate the memory required by thread 1 on the
                            Device,
                        //i.e. the memory on the GPU to which the data will
                             be send
                        err = cudaMalloc(&dt->dataOut->gridCellState,
                            size_image_part);
                        if(err)return true;
                        break;
                case T2 :
                        //allocate the memory required by thread 2 to
                            process the the data on  the GPU
                        err = cudaMalloc (&dt->dataIn->gridCellState,
                            size_image_part);
                        err |= cudaMalloc(&dt->dataOut->gridCellState,
                            part * sizeof(int));
                        if(!err){
                                //Initialization
                                SetZero<<<nBlocks1,BLSIZE>>>(dt->dataIn->
                                    gridCellState,dt->dataIn->width, (dt->
                                    dataIn->height/SLICE + 2));
                                SetZero<<<nBlocks,BLSIZE>>>(dt->dataOut->
                                    gridCellState,dt->dataIn->width, dt->
                                    dataIn->height/SLICE);
                        }
                        else
                                return true;
                                break;
                case T3 :
                        err= cudaMalloc(&dt -> dataIn->gridCellState,part *
                             sizeof(int));
                        if(!err){
                                SetZero<<<nBlocks,BLSIZE>>>(dt->dataIn->
                                    gridCellState,dt->dataIn->width, dt->
                                    dataIn->height/SLICE);
                                dt->dataOut->gridCellState= (int *)malloc(
                                    size_image);
                                for(i=0;i<dt->dataOut->width*dt->dataOut->
                                    height;i++)
                                dt->dataOut->gridCellState[i]=0;
                        }
                        else
                                return true;
                        break;
        }
        return false;
}


//************************************Initializing code
    ************************************************

void initializeArray(struct image *img){
        //fill the array with random values, 0 inicate empty and 1 indiate
            tree
        int random=0;
        srand(time(NULL));
```

```
        for(int i=0;i<img->height;i++){
                for(int j=0;j<img->width;j++){
                        // The initial state of each cell of the spatial
                            grid is either set to #tree with
                        //a probability p or to #empty with a probability
                            1-p.
                        double val = (double)rand() / RAND_MAX;
                        if(val <= 0.58)   //Probatility to have tree is
                            0.58
                                random = 1;
                        else
                                random = 0;
                        img->gridCellState[i*img->width + j]=random;
                }
        }
}


__global__ void SetZero(int *tabi,int w, int h)
{
    //int i = blockIdx.y * blockDim.y + threadIdx.y;;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if(j<h*w)   tabi[j] = 0;
}


//Generate Fire code
//It take as input the image and the number of trees to be fired randomly
void generateFire(struct image *img, int firedTreeCount)
{
        int i;
        //After the first step, a  random selected cell is fired
    srand(time(NULL));int random;
    for(i=0; i<firedTreeCount; i++){
                 random = rand()%((img->height * img->width) -1);
                 img->gridCellState[random]=2;
   }
}

//************************************************Transition Code
    ***************************************
//process the data on the GPU
void processOnGPU(struct image *tin,struct image *tout)
{
        //The number of blocks within a gird
        int nBlocks = (tin->width*(tin->height/SLICE))/BLSIZE+ ((tin->width
            *(tin->height/SLICE))%BLSIZE==0?0:1);
        transitionFunc<<<nBlocks,BLSIZE>>>(tin->gridCellState, tout->
            gridCellState, tin->width, tin->height/SLICE);
}


//send data from HOST to DEVICE
void toGPU(struct image *tin,struct image *tout)
{
        int part,part_with_neighbors;
        int slice_num;
        int last_part, part_remaining;
        //part: hold the number of elments for a part,
        //for example if image 10*9, and SLICE=2, then
        //the part is 10*(floor(9/2)) =40;
        part = tin->width * (tin->height/SLICE);

        //part  with neighbors hold a part with north and south neighbors
        //add two rows, one on top for north neighbors
        //and one on bottom for south neighbors
```

```
part_with_neighbors=part + (2*tin->width);

//Check if the image can be fully represented ny the number of
    slice seleced,
//explained in details in the main function
slice_num =SLICE;
while ((slice_num * part )  < (tin->height * tin->width))
  slice_num++;

if(tin->offset/part==0){
        //if we are in the begining and since we need the last row
            state of the original image, which are
        //considered as NORTH neighbours, firstly, we copy this
            last row of the original image and then
        //the first part of the image containing  inaddition to one
             more row for the SOUTH neighbours.

        //North row indexing begin at the leftmost elements of last
             row
        int north_row = (tin->width*(tin->height-1));
        //size of north neighbors
        size_t size_north = tin->width * sizeof(int);
        //copy to gpu, firstly the north neighbor
        cudaMemcpy(tout->gridCellState,tin->gridCellState+north_row
            ,size_north,cudaMemcpyHostToDevice);
        //copy the the first part from begining to size of one part
             plus size of a width
        size_t remaing_part = (part + (tin->width))*sizeof(int);
        cudaMemcpy(tout->gridCellState+(tin->width),tin->
            gridCellState,remaing_part,cudaMemcpyHostToDevice);
}
//if we are in between the first and last slice
else if((tin->offset/part>0) &&((tin->offset/part<slice_num-1))){
        //copy a part of the image including one row before and one
             row after for NORTH and SOUTH neighbour respectively
        size_t part_size = part_with_neighbors * sizeof(int);
        cudaMemcpy(tout->gridCellState,tin->gridCellState+(tin->
            offset - tin->width), part_size ,cudaMemcpyHostToDevice
            );
}
else if (tin->offset/part==slice_num-1){
//if we are in the last part we copy the last part including one
    row before the last part for NORTH neighbours
// and also we copy the first row of original image which represent
     the south neigbhors of the last part
//if the part is not fully filled we copy the rest of elements from
     the top until we fill the whole part
        last_part = (tin->width *tin->height) - (part*(slice_num-1)
            ) + tin->width;

        //The remaining part to be used from top
        part_remaining = part_with_neighbors-last_part;

        //Copy the last part , with one row before that represet
            the north neighbor
        size_t last_part_size = last_part * sizeof(int);

        cudaMemcpy(tout->gridCellState,tin->gridCellState+(tin->
            offset - tin->width),last_part_size,
            cudaMemcpyHostToDevice);
        //copy the remaing part
        size_t part_remaining_size = part_remaining * sizeof(int);
        cudaMemcpy(tout->gridCellState+ last_part,tin->
            gridCellState,part_remaining_size,
            cudaMemcpyHostToDevice);
}
}
```

```
                                                                        488
//send data bacck -result- from DEVICE to HOST
void fromGPU(struct image *tin,struct image *tout)
{
        int part,last_part;
        //part: hold the number of elments for a part,                  493
        //for example if image 10*9, and SLICE=2, then
        //the part is 10*(floor(9/2)) =40;
        part = (tin->width * (tin->height/SLICE));
        int slice_num =SLICE;
        while ((slice_num * part )  < (tin->height * tin->width))       498
                slice_num++;

        //copy data from host to device
        if ((tout->offset/part) != (slice_num-1)){
                size_t size_image = tin->width*(tin->height/SLICE) *sizeof(  503
                    int);
                cudaMemcpy(tout->gridCellState+tout->offset,tin->
                    gridCellState,size_image,cudaMemcpyDeviceToHost);
                //dumpArrayColor(tout->gridCellState+tout->offset, tin->
                    width, tin->height/SLICE);
        }
        else{
                last_part = (tin->width *tin->height) - (part*(slice_num-1)  508
                    );
                size_t size_image = last_part *sizeof(int);
                cudaMemcpy(tout->gridCellState+tout->offset,tin->
                    gridCellState,size_image,cudaMemcpyDeviceToHost);
        }
}
                                                                        513
__global__ void transitionFunc(int *grid, int *bufferGrid, int width, int
    height){
        //check the cell's 4 neighbors
    //Northj, south, east, west
    int cellState;
        int north, east, south, west;                                   518
        int i;
        i= blockIdx.x * blockDim.x + threadIdx.x;
        if(i<height*width){
                //The input grid is two rows more for the north row
                    neighbour and south
           //The first row is only used for neihbour state, so the first  523
               element to check is i+width,
           //and the Norht neighbour will be i+width-width = i
           north= i;
           //south neighbour
           south= i+width+width;
           //here we need to check if the east neighbour doesnot exceed the  528
                right border, if so its neighbour
           //is the leftmost element in he same row
           //to get in which row we are, we use (i+width)/width
           east = ((i+width+1)>=((((i+width)/width)*width)+width)) ? ((i+
               width)/width)*width:i+width+1;
           //west
           west = (i+width-1)<(((i+width)/width)*width) ?(((i+width)/width)  533
               *width)+width-1 : i+width-1;

           //Calculating the new states

                if(grid[(i+width)] ==0) cellState=0;
                else if (grid[i+width] ==1){                            538

                        if(grid[north] == 2 || grid[south] ==2
                                        || grid[east] ==2 || grid[west] ==2
                                            ){
                                cellState=2;
```

```
                        }
                        else{
                                cellState=1;
                        }
                }
                else if (grid[i+width] ==2) cellState= 3;
                else cellState= 0;

                //Store the new state of time t+1 in bufferGrid
                bufferGrid[i] = cellState;
        }
}
//swap two grids
void swapGrids(int *&grid, int *&bufferGrid){
        int  *temp;
        temp = grid;
        grid = bufferGrid;
        bufferGrid= temp;
}


/***********************************************Input Methods
    ************************************/
//Function that ask the user to enter the path of image and initialize it
    on
//based on a reference number and threshold value,
//the reference number is used to detect the green color in the image, i.e
    to detect the trees
//it then stores the state of the image in img
bool readBMP(struct image *img)
{
        char fileName[100];
        printf("\nPlease␣enter␣file␣name␣.bmp:\n");
        scanf("%s",fileName);

        FILE* file = fopen(fileName, "rb");
        if(file==0)
        {
          printf("\nCould␣not␣open␣the␣file");
          return false;
        }
        else
        {
        unsigned char info[54];
        int distance;
        int state=0;
        int width, height;
        fread(info, sizeof(unsigned char), 54, file); // read the 54-byte
            header
        // extract image height and width from header
        width = *(int*)&info[18];
        height = *(int*)&info[22];

        printf("\nwidth:%d",width);
        printf("\nheight:%d",height);

        //store the width
        img->width= width;
        img->height=height;
        //allocation on the host
        //allocate memory for the forest states at time t;
        img->gridCellState = (int *) malloc ((img->width * img->height) *
            sizeof(int));

        //3 bytes per pixel
        int size = 3 * img->width * img->height;
```

```
        unsigned char* data = (unsigned char*)malloc(size); // allocate 3
            bytes per pixel
        fread(data, sizeof(unsigned char), size, file); // read the rest of
             the data at once
        fclose(file);

        //detect color based on ecidian distance;                          608

        for(int i =0 ; i<size; i+=3){//height
                //Red indeed at i+2. Green indeed at i+1, Blue indeed at i,
                     the order in the file is B, G , Red
                distance = sqrt(pow((data[i+2] - RED_REF),2) + pow((data[i
                    +1] - GREEN_REF),2) + pow((data [i] - BLUE_REF),2));
                state=0;                                                   613
                if(distance < THRESHOLD)
                {
                        state=1;
                }
                else state=0;                                              618
                //i is 3 times more
                img->gridCellState[(i/3)]=state;
        }
return true;
}                                                                          623
}

//Random generation of a forest
bool randomGeneration(struct image *img)
{                                                                          628
        printf("Random␣gereration");
        int width, height;
        bool succ;
        printf("\nEnter␣width:␣");
        scanf("%d",&width);                                                633
        printf("\nEnter␣height:␣");
        scanf("%d",&height);
        img->width= width;
        img->height=height;
        //allocation on the host                                           638
        //allocate memory for the forest states at time t;
        img->gridCellState = (int *) malloc ((img->width * img->height) *
            sizeof(int));
        succ =true;
        //initializeArray(img);
        printf("allocated");                                               643
        initializeArray(img);
        //initializationOnGPU(img);
        if(succ)
                return true;
        else                                                               648
                return false;
}

/*********************************************************************OpenGL
     Display Helper **********************/
void computeFPS()                                                          653
{
    frameCount++;
    fpsCount++;

        if (fpsCount == fpsLimit) {                                        658
        char fps[256];


        float ifps = 1.f / (sdkGetAverageTimerValue(&timer) / 1000.f);
        sprintf(fps, "Forest␣Fire␣Automate␣Simulation,␣By␣AHMED␣Ahmed,␣    663
            Supervised␣by␣Prof.Bernard␣Pottier:␣%3.1f␣fps␣%d␣generations",
```

```
            ifps, frameCount);

        glutSetWindowTitle(fps);
        fpsCount = 0;
        //if (g_CheckRender && !g_CheckRender->IsQAReadback()) fpsLimit = (
            int)MAX(ifps, 1.f);

        //checkCudaErrors(cutResetTimer(timer));
        sdkResetTimer(&timer);
        //AutoQATest();
    }
 }


void keyboard(unsigned char key, int x, int y)
{

        if (key==' ')
        {
                g_pause = !g_pause;
        }

        if (key=='s')
        {
                g_pause = true;
                g_singleStep = true;
        }
        display();
}


//*************************************************************Helpers
    **********************/
//Print functions
void dumpArray(struct image *img){
        int i,j;
        printf("\n**************************************\n");
        //print contents of the array
        for(i=0;i<img->height;i++){
                for(j=0;j<img->width;j++){
                        printf("%d ",img->gridCellState[i * img->width + j
                            ]);
                }
                printf("\n");
        }
}
void dumpArrayColor(struct image *img){
        printf("\n**************************************\n");
        //print contents of the array
        int i,j;
        int *grid,width;
        grid = img->gridCellState;
        width = img->width;
        for(i=0;i<img->height;i++){
                for(j=0;j<img->width;j++){
                        if(grid[i * width + j] == 0) printf("\e[1;0m%d ",
                            grid[i * width + j]);
                        else if(grid[i * width + j] == 1) printf("\e[1;32m%
                            d ",grid[i * width + j]);
                        else if(grid[i * width + j] == 2) printf("\e[1;31m%
                            d ",grid[i * width + j]);
            else if(grid[i * width + j] == 3) printf("\e[1;34m%d ",grid[i *
                 width + j]);
                }
                printf("\e[1;0m\n");
        }
```

```
}                                                                                723
void dumpArrayColor(int *grid, int width,int height){
        int i,j;
        printf("\n**************************************\n");
        //print contents of the array
        for( i=0;i<height;i++){                                                  728
                for(j=0;j<width;j++){
            if(grid[i * width + j] == 0) printf("\e[1;0m%d␣",grid[i * width
                 + j]);
                        else if(grid[i * width + j] == 1) printf("\e[1;32m%
                            d␣",grid[i * width + j]);
                        else if(grid[i * width + j] == 2) printf("\e[1;31m%
                            d␣",grid[i * width + j]);
            else if(grid[i * width + j] == 3) printf("\e[1;34m%d␣",grid[i *   733
                 width + j]);
                }
                printf("\e[1;0m\n");
        }
}
//Delay between the outputs                                                      738
void wait ( int seconds ){
        clock_t endwait;
        endwait = clock () + seconds * CLOCKS_PER_SEC ;
        while (clock() < endwait) {}
}                                                                                743
```