U.P.B. Sci. Bull., Series C, Vol. 78, Iss. 2, 2016 ISSN 2286-3540

EFFICIENT IMAGE PROCESSING USING REACTION-DIFFUSION CNN IMPLEMENTED IN CUDA TECHNOLOGY

George Valentin STOICA[1], Radu DOGARU[2], Elena Cristina STOICA[3]

This paper proposes an implementation model for reaction-diffusion Cellular nonlinear networks (RD-CNN) on CPU and GPU platforms. Efficient implementations are proposed in order to speed-up the computational model of the RD-CNN using nVidia's CUDA platform, highlighting the GPU advantages over the CPU.

Keywords: reaction-diffusion CNN, CUDA-enabled GPU, nonlinear image processing

1. Introduction

This paper explores an implementation model for speeding-up the execution time for the highly computational model of the reaction-diffusion CNN (RD-CNN) described in [1]. RD-CNNs as well as standard CNNs are computing intensive, and this is a limiting factor to explore its full potential especially for image processing tasks. Hardware implementations using VLSI or FPGA architectures can provide the required computing power but at a higher acquisition, development and implementation costs. Recent developments of General-Purpose computation on Graphics Processing Units (GPGPU) technology provide scalable, powerful and cost effective solutions [2].

GPU represents an important computing resource due to its widespread cost effective powerful and massively parallel architecture. There are various fields in which GPU accompanies classic CPU architectures or even replace them in order to solve intensive computing tasks: imaging, computer vision, finance, scientific visualizations, cryptography, etc [3]. Investigating the capabilities of GPU could produce affordable and easy to implement models for implementing intensive computation RD-CNN image processor.

Increased interest is focused on adapting and implementing existing algorithms on GPU. One step is to identify the applications or parts of the

[1] Faculty of Electronics, Telecommunications, and Information Technology, University POLITEHNICA of Bucharest, Romania, e-mail: vstoica@yahoo.com
[2] Faculty of Electronics, Telecommunications, and Information Technology, University POLITEHNICA of Bucharest, Romania
[3] Faculty of Electronics, Telecommunications, and Information Technology, University POLITEHNICA of Bucharest, Romania

applications that exploit the multi-threaded capabilities of the multi-core GPU architecture, another step is to identify and efficiently use the GPU specific resources, i.e. multi-core computing power, fast memory resources (registry, shared memory), high bandwidth long latency global memory, specific architecture of threads, blocks, warps, grids. Image or video processing involves large data and requires a significant level of data based parallel processing power, migrating from CPU to GPU can generate 10x-200x typical speed-up [4].

From early experiments GPU architecture becomes a viable solution for CNN implementations, for example the CNN edge detector on GPU execution time is comparable with CPU OpenCV's Laplace transform [5]. A more recent Fermi GPU architecture is included into a comparison along with the Intel i5 CPU, CELL and Xilinx Virtex-5 FPGA, all running the CNN image processor on a 512x512 size image [6]. We introduce the T$_{cit}$, the execution time per cell and iteration, in order to unify the computing performance measurements.

Table 1

CPU, GPU, CELL, FPGA comparison [6]

	Computing resources			
	Intel i5 660	***GTX560 GPU***	***CELL***	***XC5VSX240T FPGA***
T$_{cit}$(ns)	***2.519***	***0.227***	***0.276***	***0.016***
Acceleration	1	11.07	9.13	162.10

2. RD-CNN image processor

Exploring the huge parameter space the reaction-diffusion model can implement many useful image processing tasks. One important advantage of this model is the inherent parallelism and a simple coupling between cells. This makes the RD-CNN suitable for discrete implementation using parallel systems like FPGA, GPU and CPU as well. Choosing a two layers configuration, *u* and *v*, the mathematical continuous-time model of the RD-CNN is:

$$\frac{du_{i,j}}{dt} = f_1(u_{i,j}, v_{i,j}, G) + D_1(u_{i,j+1} + u_{i,j-1} + u_{i-1,j} + u_{i+1,j} - 4u_{i,j})$$
$$\frac{dv_{i,j}}{dt} = f_2(u_{i,j}, v_{i,j}, G) + D_2(v_{i,j+1} + v_{i,j-1} + v_{i-1,j} + v_{i+1,j} - 4v_{i,j}) \qquad (1)$$

where f_1 and f_2 are nonlinear functions given by (2):

$$f_1(u,v) = cu - \frac{1}{3}u^3 - v$$
$$f_2(u,v) = -e(u - bv + a) \qquad (2)$$

The *c*, *a*, *b*, *e* set of cell parameters are called genes, G=[*c*, *a*, *b*, *e*], and D_1 and D_2 are the diffusion coefficients defining the coupling.

Developing the discrete-time image processor, [1] proposes the following formulae:

$$
\begin{aligned}
& u_{i,j} = \lambda x_{i,j} \\
& v_{i,j} = \lambda x_{i,j} \\
& for \quad t = 1, \ldots T, for \quad allcells(i,j) \\
& \left| \begin{array}{l}
u^{+}_{i,j} = u_{i,j} + \Delta t\left[f(u_{i,j}, v_{i,j}) + D_1 (u_{i+1,j} + u_{i-1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}) \right] \\
v^{+}_{i,j} = v_{i,j} + \Delta t\left[f(u_{i,j}, v_{i,j}) + D_2 (v_{i+1,j} + v_{i-1,j} + v_{i,j-1} + v_{i,j+1} - 4v_{i,j}) \right] \\
u_{i,j} = u^{+}_{i,j} \\
v_{i,j} = v^{+}_{i,j}
\end{array} \right.
\end{aligned}
\tag{3}
$$

There are two cell layers, corresponding to $u_{i,j}$ and $v_{i,j}$ cells. Each layer is initialized with the input image to be processed, i.e. $x_{i,j}$, assuming that each $x_{i,j} \in [-1,1]$ and λ is the gain parameter that may influence the dynamics and the output image. If not specified otherwise, $\lambda=0.5$. The number of iterations, T, corresponds to a period of time of the continuous time model. Δt is another parameter that can tune the discrete time model, from various experiments $\Delta t_{crit} \approx 0.12$ may lead to an unstable system, thus if not specified otherwise we will consider $\Delta t=0.1$. By selecting the genes G and the number of iterations T, there can be implemented a wide range of image filters. Fig. 1 shows the evolution of the two layers using specific parameters to implement the *edge detection* filter.

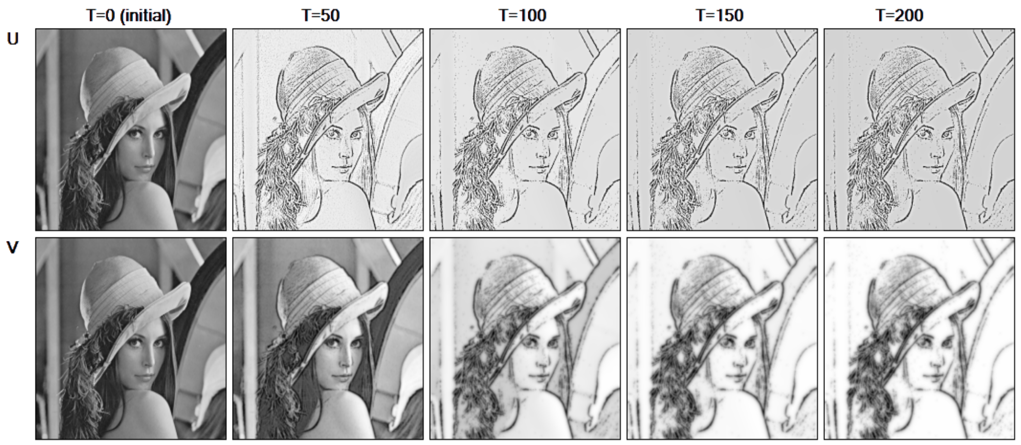

Fig. 1. The evolution of the RD-CNN image processor for a=-0.5, b=1.2, c=1, e=-0.1, D1=0, D2=2.2 and T=50,100,150,200 implementing the *edge detection* filter

The image processor presented in (3) describes an iterative intensively computational process suited for implementation in various platforms like multi

CPU, GPU, FPGA. Among all, one particular platform is analyzed in this paper: GPU and more specifically the nVidia’s CUDA C platform.

3. Implementation model

Implementing the discrete-time image processor described in (3) is straightforward, as presented in the next *C* like code:

```
void rd_cnn(float* u1, float * v1, float * u2, float * v2, int N){
  for(i=1; i<N-1; i++){
    for(j=i*N+1; j<(i+1)*N-1; j++){
      u2[j]=u1[j]+dt*(f1(u1[j],v1[j])
            +D1*(u1[j+1]+u1[j-1]+u1[j-w]+u1[j+w]-4.0f*u1[j]));
      v2[j]=v1[j]+dt*(f2(u1[j],v1[j])
            +D2*(v1[j+1]+v1[j-1]+v1[j-w]+v1[j+w]-4.0f*v1[j]));}}}
void main(){
  for(t = 0; t<T; t++)  {
    rd_cnn(u1, v1, u2, v2, N); u2↔u1; v2↔v1;}}
```

Note that the initial 2D matrixes were transformed into 1D arrays using row-major convention: all elements of the same row are placed into consecutive memory locations. This is due the *flat* memory space used in both CPU and GPU platforms, in which the main memory/global memory elements are accessed in a linear mode. CUDA C uses row-major memory layout.

The above code lacks some steps as the initialization of the `u1` and `v1` matrixes from the input image. The input image contain *NxN* pixels and in the case of grayscale images each pixel is an *[0, 255]* range integer value. Each value must be transformed into *[-0.5, 0.5]* range floating point values as presented in (3). Additional processing must be performed at the end of the *T* steps: transforming the image back to grayscale image. In this case extra care must be taken: values of the `u1` and `v1` elements could exceed initial range and proper scale must be performed. Frontier elements must be processed separately from the internal elements since there are no left/right/up/down cells (depending on the specific cell position), without this approach errors will be propagated to the inner cells.

In order to have a relevant base for measuring the efficiency of GPU implementations, there will be some simple optimizations to the initial code. For example instead of accessing each memory location multiple times it is more efficient to create a local variable that can be placed into the registry, initialize it one and use it many times:

```
float utemp, vtemp;
for(i=1; i<N-1; i++){
  for(j=i*N+1; j<(i+1)*N-1; j++){
    utemp=u1[j]; vtemp=v1[j];
    u2[y]=utemp+dt*(f1(utemp,v1[j])
            +D1*(u1[j+1]+u1[j-1]+u1[j-N]+u1[j+N]-4.0f*utemp));}}
```

Accessing an element can be made using the notation `u1[i]` or `*u1`, this transforming the iteration through the entire array into the following:

```
float utemp, vtemp;
float* u1temp, v1temp;
u1temp = u1; v1temp = v1;
for(i=1; i<N-1; i++){
  for(j=i*N+1; j<(i+1)*N-1; j++){
    utemp=u1temp++; vtemp=v1temp++;}}
```

Using such optimizations there will be significant improvements and the optimized execution code is twice as fast compared with the non-optimized version, as presented in the Table 2.

Table 2

Execution time for optimized vs. non-optimized CPU code for various image size

Execution time (s)	Image size			
	512x512	1024x1024	2048x2048	4096x4096
CPU non-optimized	2.56	10.47	43.26	166.11
CPU optimized	1.33	5.34	21.08	88.54

4. GPU implementation using CUDA

RD-CNN exhibits a large degree of parallelism and a very simple coupling between neighborhood cells. As resulted from (3), for a specific iteration and cell, there is independence between cells, i.e. each $u_{i,j}$ cell state is computed from its corresponding value and neighborhood cells values set in the previous iteration. Synchronization between iterations is required.

The massive degree of parallelism of the current GPUs can be exploited to speed-up the execution time of RD-CNN. As opposite to GPU computing, CPU computing using multithreading has some disadvantages: uses less power for computing and focuses on complex data cache or flow control, uses complex and slow structures for thread management, uses limited number of cores and threads, has slower memory bandwidth. CPU is generally optimized for sequential code execution, while the GPU is focused on massive floating point calculations inherited from the initial purpose: 2D and 3D graphics. GPU trades large cache and complex control chip circuitry to higher execution throughput.

Algorithms that focus on large number of arithmetic operations and lower number of memory read/write operations will benefit more from the GPU architectures. Such an algorithm is the implementation model of RD-CNN: initial data is loaded intro GPU memory, a large number of computations intensively iterations are performed, and at the end the final image is transferred back to CPU. Efficient GPU programming patterns are based on dividing the problem into a large number of threads using fast memory resources (registry, shared memory, and cache) and minimizing long latency main memory accesses. Rather than dividing the problem in few large blocks as accustomed in multithreading CPU implementations, GPU allows (and benefits) from computing each cell in a separate thread thus obtaining hundreds, thousands and even tens of thousand of threads that will be efficiently managed by the GPU control unit.

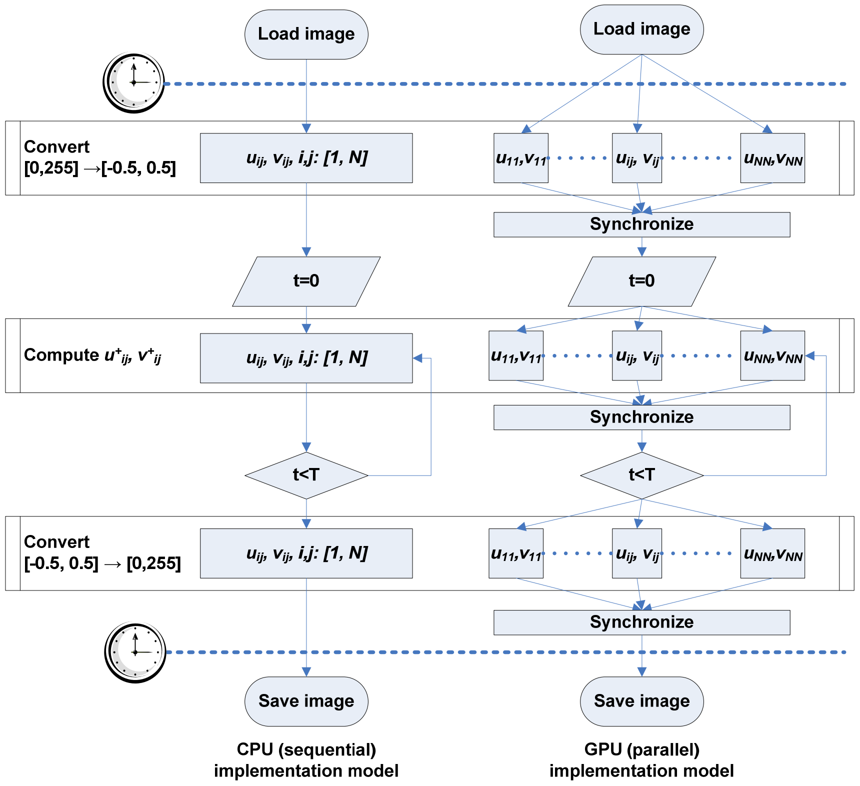

Fig. 2. CPU sequential and GPU parallel implementation model of RD-CNN

Using CUDA C we can easily transform the CPU sequential implementation into a parallel one that will run on GPU. But first we will just implement the sequential code into GPU and run it in a single thread. This will highlight the general recommendation that sequential algorithms achieve better execution time using CPU platform due to their special architecture designed for such problems. As presented in the Table 3, there are few orders of magnitude difference when comparing the single thread versions of CPU and GPU executing the same code, highlighting that GPU shows its power only when all of its resources are intensively used.

Table 3

CPU vs. GPU single thread execution time over various image sizes

Execution time (s)	Image size			
	128x128	256x256	512x512	1024x10246
CPU sequential	0.13	0.27	1.33	5.34
GPU sequential (1 thread)	2.73	11.24	44.91	180.16

The CUDA parallel programming model is based on the decomposition of the problem into independent blocks of threads and assigning each block on the available physical multiprocessors and cores. This enables the automatic scalability: while each block can be independently executed, different hardware with different number of physical multiprocessors can produce the same result with different performance, i.e. the GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors. The numerous threads from each block provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism represented by the blocks [7]. CUDA parallel programming model is based on the following concepts:

- *kernel*: user defined functions that are executed by the threads
- *threads*: the smallest execution entity
- *blocks*: a collection of threads that runs on a physical multiprocessor
- *grid*: a collection of blocks

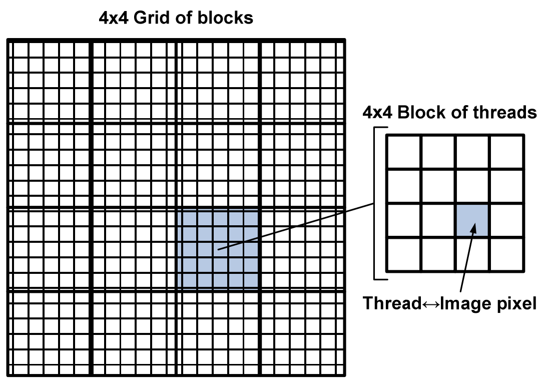

Fig. 3. Image decomposition into bidimensional grids and blocks, and cell (pixel) level threads

In order to assist the developers and match the decomposition of the problem, blocks and grids can have one-dimensional, two-dimensional, or three-dimensional structure: threads in a block and blocks in a grid can be organized using one-, two-, or three-dimensional structure.

The highly multithreaded GPUs encourage the use of massive, fine-grained data parallelism in CUDA. Efficient threading support in GPUs allows applications to expose a much larger amount of parallelism than available hardware execution resources with little or no penalty [8]. Applying this technique to the image processing problem using RD-CNN we can use one thread to compute the state of one cell. Threads are grouped in a two-dimensional block of threads. Blocks of threads are grouped in a two-dimensional grid of blocks. The grid and block are sized to have one thread per cell.

Implementing the model presented in Fig. 3 requires three kernel functions that will run on the GPU hardware:

```
__global__ void gpu_int_to_float (float* u, float* v, const
unsigned char* image, int N){
  int i = (blockIdx.y * blockDim.y + threadIdx.y) * N
            + blockIdx.x * blockDim.x + threadIdx.x;
  float temp = (float)image[i]/ 256.0f - 0.5f;
  u[i] = temp;  v[i] = temp;}

__global__ void gpu_float_to_int (float* u, const unsigned char*
image, int N){
  int i = (blockIdx.y * blockDim.y + threadIdx.y) * N
            + blockIdx.x * blockDim.x + threadIdx.x;
  image[i] = (unsigned char)( 255.0f * (u[i] + 0.5f));}

__global__ void gpu_rd_cnn(float* u1, float * v1, float * u2,
float * v2, int N){
  i = (blockIdx.y * blockDim.y + threadIdx.y) * N
            + blockIdx.x * blockDim.x + threadIdx.x;
  u2[i]=u1[i]+dt*(f1(u1[i],v1[i])
            +D1*(u1[i+1]+u1[i-1]+u1[i-N]+u1[i+N]-4.0f*u1[i]));
  v2[i]=v1[i]+dt*(f2(u1[i],v1[i])
            +D2*(v1[i+1]+v1[i-1]+v1[i-N]+v1[i+N]-4.0f*v1[i]));}

void main(){
  dim3 dimBlock(blockSizeX, blockSizeY);//e.g.32x32 threads/block
  dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);//e.g.16x16 blocks/grid
  gpu_int_to_float<<<dimGrid,dimBlock>>>(u1, v1, source_image, N);
  cudaDeviceSynchronize();
  for(t = 0; t<T; t++)  {
    gpu_rd_cnn(u1, v1, u2, v2, N);
    cudaDeviceSynchronize();
    u2↔u1; v2↔v1; }
  gpu_float_to_int<<<dimGrid,dimBlock>>>(u1, output_u_image, N);
  gpu_float_to_int<<<dimGrid,dimBlock>>>(v1, output_v_image, N);
  cudaDeviceSynchronize();}
```

Using 32x32 threads per block and N/32xN/32 grid of blocks, the execution time is dramatically reduced compared with the CPU version, as shown in the Table 4.

Table 4

CPU sequential vs. GPU multithread execution time for various image sizes and T=200 iterations

Execution time (s)	Image size			
	512x512	1024x1024	2048x2048	4096x4096
CPU sequential	1.33	5.34	21.08	88.54
GPU multithread	0.05	0.16	0.51	1.96

Efficient implementation using CUDA must consider the hardware resources and limitations. For example, in order to simplify the control unit of the GPU, threads from a block are grouped in *warps* and are managed together: all the threads from the warp executes the same instruction at one moment (but on different data), all the threads from the warp are executed by the hardware cores or all are waiting for a resource if one or may of them are waiting for that resource. The warp size is a property of each device. The tested GPU has 32 threads per warp. If there are blocks having less than 32 threads, then there will be an inefficient usage of the hardware resources since not all the 32 cores allocated to the warp are occupied by a thread. As presented in the Table 5. it is clearly that higher hardware occupancy provides better performance.

Table 5

Execution time as function of the number of threads in a warp for 512x512 pixels image and T=200 iterations

Execution time (s)	Threads per warp					
	1	2	4	8	16	32
GPU multithread	1.12	0.58	0.30	0.17	0.10	0.05

One disadvantage of GPU over CPU is the global memory access latency. Depending on the hardware version there is 200-800 clock cycles latency while accessing a byte from the global memory [7]. For maximum performance, these memory accesses must be coalesced as with accesses to global memory. Global memory resides in the device memory and device memory is accessed via 32-, 64-, or 128-byte memory transactions. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. By convention the image matrix is organized using row-major configuration as described in section 3. Simultaneously accessing consecutive cells from a row ensures that these accesses can be included into a single transaction. Organizing the blocks in horizontal tiles ensures that consecutive threads will access consecutive memory locations as a single transaction. As opposite to this configuration, when the blocks are organized in vertical tiles then each threads will access scattered memory locations with limited to no chance that these access to be grouped into a single transaction, as presented in the Fig. 4.

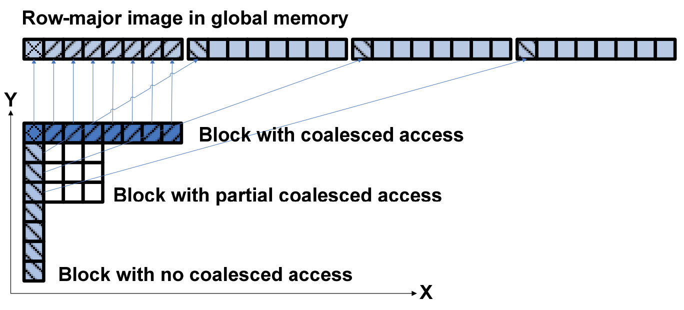
Row-major image in global memory
Y
Block with coalesced access
Block with partial coalesced access
Block with no coalesced access
X

Fig. 4. Horizontal blocks vs. vertical blocks and the memory access to the row-major image matrix

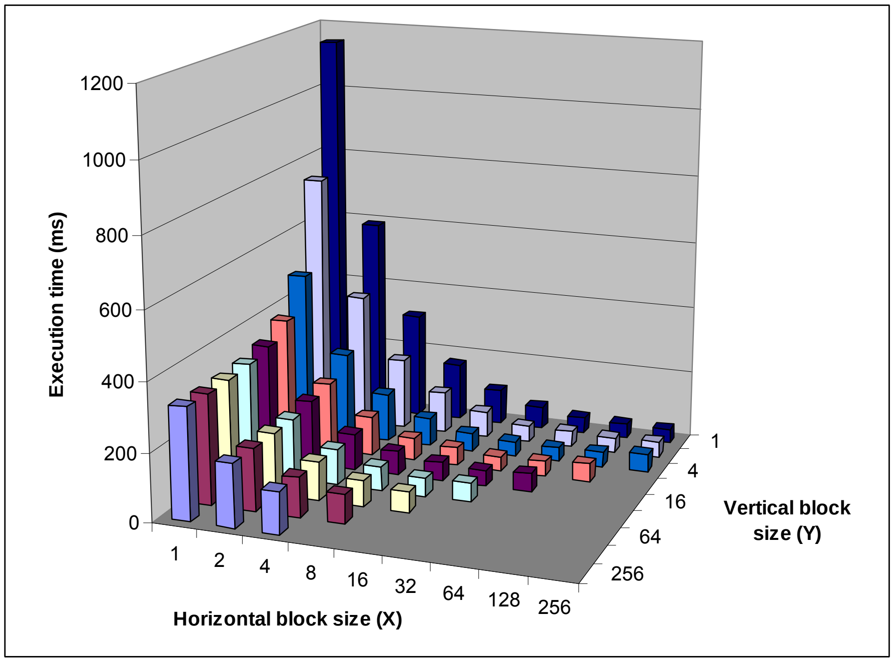

Fig. 5. Execution time for different block dimensions for 512x512 image size and T=200 iterations

The difference between coalesced and not coalesced global memory access is reflected into the execution time, as presented in the Fig. 5. Horizontal blocks (i.e. 256x1, 256x2, 128x1 etc.) performs coalesced memory access and the execution time is at least half compared with the no coalesced memory access specific to vertical blocks (i.e. 1x256, 2x256, 1x128 etc.). Note that there is a hardware limit for maximum 1024 threads per block.

5. Conclusions

This paper proposes an implementation model for reaction-diffusion CNN as defined in [1]. The model requires a significant amount of computing power necessary to implement image processing tasks and there is a need for efficient implementations. One cost effective solution is to implement the model using the massive parallel power of GPU. Another advantage in favor of this solution is that current development tools does not requires learning GPU new or specific programming languages, for example nVidia's CUDA platform is compatible with C, C++, FORTRAN easing the conversion of existing CPU applications for GPU execution.

Several techniques of GPU resources usage were tested. According to the experiments presented in this paper the acceleration over the CPU implementation can go up to 77x, this using the GPU's global memory, shared memory, local registers and computing power.

Table 6

CPU and GPU execution time and corresponding acceleration for various image size and T=200 iterations

		Image size						
		512x512	768x768	1024x1024	1536x1536	2048x2048	3072x3072	4096x4096
Execution time (s)	CPU sequential	1.329	2.474	5.338	12.065	21.084	48.534	85.442
	GPU multi-threading	0,034	0,052	0,081	0,173	0,287	0,641	1,115
Execution time per cell and iteration T_{cit} (ns)	CPU sequential	25,349	20,972	25,454	25,569	25,134	25,714	25,464
	GPU multi-threading	0,648	0,441	0,386	0,367	0,342	0,340	0,332
Acceleration		**39x**	**48x**	**66x**	**70x**	**73x**	**76x**	**77x**

Measurements were performed on the following hardware/software architecture:

- Windows 7/32 bit operating system, nVidia CUDA C Toolkit v5.5
- CPU Intel Core 2Duo E6320 CPU running at 1.860 GHz [9], 2GB DDR2 DRAM

- GPU nVidia GeForce GTX 650 Ti Boost using Kepler architecture compatibility 3.0, four 980 MHz base clock multiprocessors, each having 196 cores with a total of 768 cores, 1GB GDDR5 DRAM with a bandwidth of 144.2 GB/s [10].

Tests were performed on 8bpp grayscale images with various dimensions. Computations were made using single precision floating point numbers. The measured execution time includes the data transfer between the CPU main memory and GPU global memory and back. Initial memory allocation and final memory freeing were not measured.

Future work will extend the GPU model for using other CUDA specific resources like shared memory, more registries. Instead floating point operations a version using only integer operations can be tested for more efficient execution time while obtaining similar results. Current experiments were conducted only for still images obtaining for example $\cong 0.05$ seconds processing time for 1024x1024 pixels grayscale images, this being equivalent with 20fps. Further enhancements and interleaving CPU-GPU-CPU image copy and GPU image processing can be used for real-time video processing.

R E F E R E N C E S

[1]. *R. Dogaru*, “Applications of Emergent Computation in Reaction-Diffusion CNNs for Image Processing”, International Conference on Control Systems and Computer Science (CSCS), Bucharest, 29-31 May 2013 , pp. 370 – 377

[2]. *K.V. Kalgin*, “Implementation of algorithms with a fine-grained parallelism on GPUs”, Numerical Analysis and Applications, Vol.4, No.1, 2011, pp. 46-55.

[3]. *** GPU Applications by domain, http://www.nvidia.com/object/gpu-applications-domain.html

[4]. *R. Di Salvo and C. Pino*, “Image and Video Processing on CUDA: State of the Art and Future Directions”, MACMESE'11 Proceedings of the 13th WSEAS international conference on Mathematical and computational methods in science and engineering, pp. 60-66, 2011

[5]. *R. Dolan and G. DeSouza*, “GPU-Based Simulation of Cellular Neural Networks for Image Processing”, Proceedings of International Joint Conference on Neural Networks, Atlanta, Georgia, USA, June 14-19, 2009

[6]. *E. Laszlo, P. Szolgay and Z. Nagy*, “Analysis of a GPU based CNN implementation”, 13th International Workshop on Cellular Nanoscale Networks and Their Applications (CNNA), Turin, Aug. 29-31, 2012

[7]. *** CUDA C Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[8]. *D. B. Kirk and W. W. Hwu*, “Programming Massively Parallel Processors”, Second Edition, Morgan Kaufmann, 2013

[9]. *** Intel Core 2Duo E6320 CPU specifications http://www.intel.com/support/processors/sb/CS-032819.htm

[10]. *** nVidia GeForce 650Ti Boost Hardware specifications, http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-650ti-boost/specifications