Computer Science • 14 (3) 2013 http://dx.doi.org/10.7494/csci.2013.14.3.385

Paweł Topa
Paweł Młocek

# USING SHARED MEMORY AS A CACHE IN CELLULAR AUTOMATA WATER FLOW SIMULATIONS ON GPUs

**Abstract** *Graphics processors (GPU – Graphic Processor Units) recently have gained a lot of interest as an efficient platform for general-purpose computation. Cellular Automata approach which is inherently parallel gives the opportunity to implement high performance simulations. This paper presents how shared memory in GPU can be used to improve performance for Cellular Automata models. In our previous works, we proposed algorithms for Cellular Automata model that use only a GPU global memory. Using a profiling tool, we found bottlenecks in our approach. With this paper, we will introduce modifications that takes an advantage of fast shared memory. The modified algorithm is presented in details, and the results of profiling and performance test are demonstrated. Our unique achievement is comparing the efficiency of the same algorithm working with a global and shared memory.*

**Keywords** Cellular Automata, GPU computation, modelling physical phenomena

## 1. Introduction

Cellular Automata is a modeling paradigm for simulating a variety of natural phenomena and physical processes [1, 2]. It models a physical system as a lattice of cells that are characterized by a specific set of states that change according to certain rules of local interactions. The most important advantage of this approach is the ability to construct very fast and conceptually simple algorithms. Moreover, Cellular Automata can be easily parallelized just by simple static domain decomposition. Additionally, the amount of data that has to be exchanged between computational nodes is relatively small, since a neighbourhood is limited to the nearest cells. All these features provide the opportunity for effective implementation of the model on the GPU platform.

In recent years, the computing power of GPUs increased rapidly surpassing, in some applications, the performance of general purpose processors. This has led to the conclusion that GPU can be also utilized for general purpose computing (GPGPU *General-Purpose Computing on Graphics Processing Units*) [3, 4]. When the problem fits into GPU computing architecture, the benefits of its use can greatly exceed the capabilities of CPU. In addition, GPUs can be also used in combination with traditional CPU platform (see examples in [5, 6]). In this case, the part of the model that can be efficiently processed by GPU have to be extracted from a whole. In most cases, as the GPU has a very specific architecture. the algorithms that solve the given problem have to be redesigned or implemented "from scratch".

Initially, the GPU computations were implemented using languages that were originally designed to write code for vertex and pixel shaders units: HLSL (High Level Shading Language), GLSL (OpenGL Shading Language) and Cg (C for graphics). The main drawback of these languages is the fact that their instructions and data structures are designed for graphics processing. In case of general purpose computing the necessary types of data have to be emulated. In order to meet the needs of developers who wanted to use the computing power of GPUs, NVIDIA announced in 2007, the CUDA (*Compute Unified Device Architecture*) [7] architecture with a dedicated "C for CUDA" programming language. For the same purpose, the Khronos Group in 2008 presented an open programming environment OpenCL (*Open Computing Language*) [8]. The goal of this project is to provide a unified programming environment for creating programs that use various computational resources available in modern computers: CPU units as well as GPU units. Both tools allow easy programming of the GPU for any type of computational problems.

The CUDA and OpenCL provide methods to access various types of memory. The most important differences between them are throughput and latency. Global memory is accessible by all threads without any limits and programmer uses this memory in a very simple way. This type of memory is implemented outside the chip what makes that its efficiency is lower than shared memory (or local memory in OpenCL nomenclature). Shared memory stores data that are local for kernels and are accessible only by threads that belong to the same workgroup. Physically, this memory is usually located on-chip what makes that its throughput and latency are comparable to registers. High efficiency makes that, it is worth to modify the algorithms of the model to use local memory as a cache.

In [20] we presented a Cellular Automata model of water flow running in GPU environment. Our model was implemented with OpenCL framework. We have focused on adapting the algorithm to the specifics of the graphics processor. As the tests showed the performance of our GPU implementation significantly (up to several hundreds times) exceed basic sequential CPU version. At that stage we did not evaluate how different types of memory: global and shared can influence the performance. In the current paper, we present a new version of the algorithm that exploits shared(local) memory and evaluates its performance. Our goal was to find an algorithm that efficiently uses the properties of the CUDA platform and OpenCL framework. We used the profiling tools to identify the bottlenecks in our first algorithm. Based of these analyzes, we propose the modifications that eliminate the weak points.

This paper is organized as follows. The next section presents several related works. Section 3 describes briefly the original algorithm. In subsection 3.1 we present the changes made to the algorithm to run on the GPU using global memory. The results of profiling are presented and discussed at the end of this subsection. Subsection 3.2 shows the modifications that are introduced to the algorithms in order to use fast shared memory as a cache. The performance of both algorithms (with global and shared memory) are presented and discussed. At the end, we summarize our achievements.

## 2. Related works

Graphics Processor Units (GPU) are specialized for processing in parallel a large amount of data with regular structure, i.e. set of vertices that constitute a graphic scene. Cellular Automata has similar features, e.g. large amount of cells are organized in a regular grid with the same rules applied to each cell. The use of graphics processors for the simulation with the use of Cellular Automata is presented and discussed in many papers.

Rybacki et al. [9] investigated seven different simulation algorithms (two of them were GPU-run algorithms) for cellular automata and compared them for a few well-known CA rules (e.g. Game of Life, Parity). He concludes their article with statement that the usefulness of GPU-based Cellular Automata algorithms strongly depends on the models that we want to simulate.

Gobron et al. ([10, 11]) applied cellular automata in computer graphics for visualization and creating various surface effects like automatic texturing and creating surface imperfections. Their models were implemented for GPUs with GLSL language and gained performance up to 200 times better than CPU implementations. In [12], the same author demonstrated that the Cellular Automata model of a retina implemented for GPUs was at least 20 times faster than one with a classical CPU.

Bilotta et al. [13] demonstrated how the Cellular Automata model of lava flow (MAGFLOW model [14] ) can be ported to CUDA environment. The MAGFLOW model is relatively complex and include several factors as ground elevation, lava thickness, heat quantity, temperature, and amount of solidified lava. Processing is divided into several-stages pipeline, each of them implemented as a separated kernel. The authors have devoted much attention to the optimal use of different types of memory, but does not discuss other scenarios of their use. The next paper from this team [15] focused on using multiple-GPUs systems.

Ferrando et al. [16] used a GPU hardware to implement the Cellular Automata model of evolving surfaces during wet etching. This model use global memory as well as shared memory but the authors does not discuss how it influences the performance.

Quesada-Barriuso et al. [17] presented two versions of the Cellular Automata based watershed algorithm. This algorithm is widely used for non-supervised image segmentation. The authors compare the traditional approach based on synchronous updating with a new method in which the automata is updated asynchronously. The new algorithm uses shared memory to store pixels during the calculation. Unfortunately, the authors consider only this one variant of using the shared memory.

Miao et al. [19] demonstrated how GPU can be applied for modelling pedestrians evacuation. Their Cellular Automata model exploits various types of memory provided by CUDA platform but the authors do not discuss how it influences the efficiency.

Caux et al. [18] compared the performance of global memory and shared memory using simple Cellular Automata heart model. They significantly simplified the implementation that uses shared memory. The data are copied from global memory to shared memory before computing new states but the threads handling cells at the border are not used to compute the next state.

Although the advantages of shared memory are emphasized in most publications relatively little space is devoted to the analysis of the potential benefits of its use.

## 3. Model of water flow

Cellular Automata algorithm presented in this paper, simulates flow of water through the terrain. It is based on a model of lava flow developed by Di Gregorio et al [21]. The algorithm was adapted to model water flow [22] and later it was applied to model anastomosing river phenomena [23, 24, 25].

The Cellular Automata for modeling water flow is defined as follows:

$$CA_{FLOW} = \langle Z^2, A_i, A_o, X, S, \delta \rangle \tag{1}$$

where:

- $Z^2$ — set of cells located on regular mesh and indexed by $(i, j)$;
- $A_i \subset Z^2$ — set of inflows;
- $A_o \subset Z^2$ — set of sinks;

- $X$ — Moore neighbourhood;
- $S = \{(g_{i,j}, w_{i,j}\}$ — set of parameters;

  $g$ — terrain altitude,

  $w$ — amount of water,
- $\delta : (g^t_{i,j}, w^t_{i,j}) \rightarrow (g^{t+1}_{i,j}, w^{t+1}_{i,j})$ — set of rules of local interactions.

A regular grid of cellular automata represents a terrain through which water flows. Arbitrary defined cells act as sources that supply water to the system. At each step of simulation, the amount water in these cells is supplemented to a fixed value. Similarly, another group of cells acts as drains and removes water from the system.

The flow of water is driven by the difference of water level in cells. The water level in a particular cell is calculated as the sum of terrain altitude $g$ and amount of water $w$. The rule that we applied to each cell is trying to equalize the level of water in this cell and its neighbours.

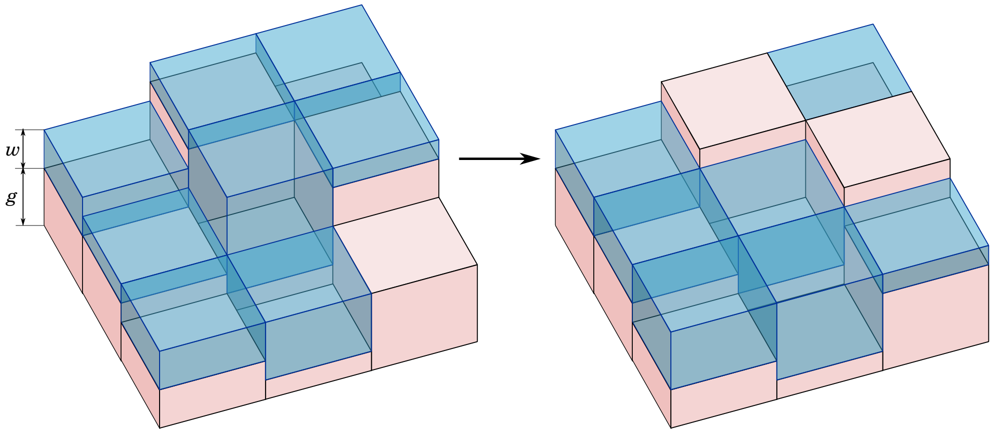

**Figure 1.** Distribution of water between surrounding cells.

The key part of the model is the algorithm that calculates an exchange of water between neighbouring cells (called *level minimization algorithm*). The algorithm calculates an expected average level of water in central cell and its neighbours. The pseudocode of this algorithm is presented in Algorithm 1.

The water level in central cell is denoted as the sum of the water amount $w$ (which can be distributed to the neighbours) and the terrain altitude $q[0]$ (which cannot be changed). The water levels in four neighbouring cells are denoted as $q[1]...q[4]$. The algorithm calculates an average level for all these cells (lines from 8 to 13). Next, the cells with water level higher than the newly calculated average level are eliminated from further calculation (lines 15 to 20). These operations are repeated as long as a cell can be eliminated (variable *newControl* in line 21). Finally, for all not eliminated cells the new average level is calculated and the amount of water that has to be moved from the central cell is calculated and stored in $f$ table (lines from 22 to 28).

**Algorithm 1** Original algorithm calculating flow on cellular automata.

```
Require: $w, q[0 \dots m]$
Ensure: $average, f[0 \dots m]$
for $i = 0 \to m$ do
    $eliminated[i] \leftarrow false$
end for
$newControl \leftarrow false$ {Calculate new average level}
repeat
    $qSum \leftarrow w$
    $count \leftarrow 0$
    for $i = 0 \to m$ do
        if $\neg eliminated[i]$ then
            $qSum \leftarrow qSum + q[i]$
            $count \leftarrow count + 1$
        end if
    end for
    $average \leftarrow qSum/count$
    for $i = 0 \to m$ do
        if $q[i] > average \wedge \neg eliminated[i]$ then
            $newControl \leftarrow true$
            $eliminated[i] \leftarrow true$
        end if
    end for
until $\neg newControl$
for $i = 0 \to m$ do
    if $eliminated[i]$ then
        $f[i] \leftarrow 0$
    else
        $f[i] \leftarrow average - q[i]$
    end if
end for
```

The algorithm in this version is not optimized for execution on the GPU architecture due to the conditional statements that may serialize the execution of the code.

## 3.1. Implementation for GPU with global memory

The model with algorithms that uses global memory is in details presented in [20]. Here we recall only the main characteristics of this algorithm. The regular lattice of cellular automata is simple mapped onto the 2-dimensional grid of threads. The lattice is divided into regions of size 16 × 16 cells. The same size has workgroups of threads. Each thread processes and updates one cell but uses data from four neighbouring cells. Threads that are located at the edges of workgroup area have to access to cells that are assigned to other workgroups. When we use global memory, it is not an issue — such the transactions are transparent to the programmer.

For implementing the model on GPU the original algorithm has been redesigned [20]. The "reversed" algorithm (see Algorithm 2) takes the following data:

- $w$ — contains amount of water in central cell,
- $q[0]$ — contains only terrain altitude for central cell,
- $q[1] \dots q[m]$ — stores water level (ground and water) for all neighbouring cells.

The value of $m$ may vary from 5 (von Neumann neighbourhood) to 9 (Moore neighbourhood) — in our implementation we use von Neumann neighbourhood. The cells are initially sorted according to their water level $q[i]$. Next, in the loop $i\colon 0 \to m$ the average level for the first $i$ elements is calculated. The loop is repeated as long as the condition is met: $i * q[i] \leq \sum_{j=0}^{i} q[j]$. The result of calculations is stored in table $f$, which contains the amount of water that has to be moved from the central cell to the selected neighbours.

**Algorithm 2** "Reversed" algorithm for calculating new average level of water.

```
Require: $w, q[0 \dots m]$
Ensure: $average, f[0 \dots m]$
 1: $qSorted \leftarrow \mathrm{sortAscending}(q, m)$
 2: $i \leftarrow 0$
 3: $qSum \leftarrow 0$
 4: while $(i < m) \wedge (w \geq i \cdot qSorted[n] - qSum)$ do
 5:     $qSum \leftarrow qSum + qSorted[i]$
 6:     $i \leftarrow i + 1$
 7: end while
 8: $average \leftarrow (qSum + p)/i$
 9: for $j = 0 \to m - 1$ do
10:     $f[j] \leftarrow \max(0, average - q[j])$
11: end for
```

The flow values ($f$ table) are accumulated in a global table. At the end of a single step of simulation that table contains the amount of water that has to be added or deducted from the $w$ parameter (the water amount) of each cell.

The algorithm presented above has been implemented as two OpenCL kernels executed sequentially for each cell:

1. `gpu_best_levels`, (see Listing 1), calculates expected average level in central cell and its neighbours. The results are stored in `best_levels[]` table,
2. `gpu_distribute`, (see Listing 2), basing on the content of the `best_levels[]` table update water amounts in all cells.

The use of global memory made the code of both kernels very simple. The calculations are performed without branch instructions — if necessary selection was "emulated" with built-in functions `step` (`step`*(x, y)* return 0.0 if $x < y$, else 1.0) and `max`. Also a short loop `while` is replaced by the sequence of instructions. The sorting function was also realized as a static sorting network for a five elements table. The Cellular Automata data (terrain altitude $g$, amount of water $w$) were stored in global memory. The kernels simple copy necessary data from global memory to private variables (`float q[5]` table) or directly index global tables (e.g. `level`, `best_level`). Each kernel reads from its own central cell and from four nearest neighbours and write to its own cell. Due to regular data and local neighbourhood the memory coalescing was achieved almost automatically. We do not have to introduce any special optimization to effectively utilize the bandwidth.

```
__kernel void gpu_best_levels(__global float level[],
                              __global float terrain[],
                              __global float best_levels[]) {
    int x = get_global_id(0), y = get_global_id(1);
    int w = get_global_size(0), h = get_global_size(1);

    float q[5];
    q[0] = terrain[x + y*w];
    q[1] = level[x + (y - 1)*w];
    q[2] = level[(x - 1) + y*w];
    q[3] = level[(x + 1) + y*w];
    q[4] = level[x + (y + 1)*w];
    best_levels[x + y*w] = best_level(level[x + y*w], q);
}

float best_level(float w, float q[]) {
    float qSum = w - q[0];
    sort5(q);
    qSum += q[0];
    int i=1;
    t = step(q[1], qSum);      qSum += t*q[1];     i += t;
    t = step(q[2] * 2, qSum); qSum += t*q[2];     i += t;
    t = step(q[3] * 3, qSum); qSum += t*q[3];     i += t;
    t = step(q[4] * 4, qSum); qSum += t*q[4];     i += t;

    return qSum/i;
}

void sort5(float a[]) {
#define SWAP(i, j) {float t=min(a[i], a[j]);
                    a[j]=max(a[i], a[j]); a[i]=t;}
    SWAP(0, 1); SWAP(3, 4); SWAP(2, 4);
    SWAP(2, 3); SWAP(0, 3); SWAP(1, 4);
    SWAP(0, 2); SWAP(1, 3); SWAP(1, 2);
#undef SWAP
}
```

Listing 1: Source code of the **gpu__best_levels(...)** kernel functions

```
__kernel void gpu_distribute(__global float level[],
                             __global float terrain[],
                             __global float sources[],
                             __global float best_levels[]) {
```

```
    int x = get_global_id(0), y = get_global_id(1);
    int w = get_global_size(0), h = get_global_size(1);
    float a_level = level[x + y*w];
    float a_terrain = terrain[x + y*w];
    float change = sources[x + y*w];
    change += max(0, best_levels[x + (y - 1)*w] - a_level);
    change += max(0, best_levels[(x - 1) + y*w] - a_level);
    change += max(0, best_levels[(x + 1) + y*w] - a_level);
    change += max(0, best_levels[x + (y + 1)*w] - a_level);
    change += max(a_terrain, best_levels[x + y*w) - a_level;

    level[x + y*w] = max(change + a_level, a_terrain);
}
```

Listing 2: Source code of **gpu_distribute(...)** kernel function

Nvidia CUDA Toolkit provides profiling tools that give us insight into what way the memory transactions are performed. We used them to analyse how efficiently our algorithm uses the global memory. We tested GPU with Compute Capability 1.2 and grid with 512 by 512 cells (see Table 1). The store operations were performed optimally as 64-byte transactions. It comes from fact that each thread saves calculated data only once, in cell that is directly assigned to this cell (see Listing 1, line 13 and Listing 2, line 16).

**Table 1**

The results of profiling session for device with CC 1.2 and grid size 512 × 512.

| | |
|---|---|
| Compute Capability: 1.2. | Workgroup size: 16 × 16 threads. |
| Grid size: 512 × 512 cells. | No. of workgroups: 1024. |

| | | Global no. of transactions | | | Transactions/workgroup | | |
|---|---|---|---|---|---|---|---|
| Store | | 32B | 64B | 128B | 32B | 64B | 128B |
| | `gpu_best_levels` | 0 | 16384 | 0 | 0 | 16 | 0 |
| | `gpu_distribute` | 0 | 16384 | 0 | 0 | 16 | 0 |
| | Comment: Both kernels perform one write operation. Optimal transaction has size 64B = 16 × 4B float number. Optimal amount of transactions can be calculated as follows: $\frac{512\times512}{16}$ of 64B transactions. Profiling confirms that store operations are realized in the optimal way. | | | | | | |

| | | Global no. of transactions | | | Transactions/workgroup | | |
|---|---|---|---|---|---|---|---|
| Load | | 32B | 64B | 128B | 32B | 64B | 128B |
| | `gpu_best_levels` | 16384 | 81920 | 16384 | 16 | 80 | 16 |
| | `gpu_distribute` | 16384 | 114688 | 16385 | 16 | 112 | 16 |
| | Comment: Kernel `gpu_best_levels` read data from two tables (`level` and `terrain`; see Listing 1) and kernel `gpu_distribute` read data from three tables (`level`, `terrain` and `sources`; see Listing 2). Theoretical, non-redundant number of 64B reads should be: 2 × 16 × 1024 = 32 768 (32 per workgroup) for `gpu_best_levels` and 3 × 16 × 1024 = 49 152 (48 per workgroup) for `gpu_distribute`. It is respectively 29% and 33% of total transactions calculated during profiling. | | | | | | |

There is more complicated situation with “load” operations. Each thread read data from its own cell and from four neighbouring cells. Figure 2 presents how these transactions are realized. Data read from central cell and its neighbours (operations 1, 2, and 3) above and below are coalesced as 64B transaction. Transactions of type 4 are not aligned to the boundary of 64B, and do not fall into 128B block. Thus, they are realized as one 32B transactions and one 64B transactions. Transactions of type 5 also are not aligned to the boundary of 64B, but they can be realized by 128B transactions. As it is shown in Figure 2, the transactions of type 4, an 5 are used only partially for transferring the useful data. We have also a lot of redundant transactions from global memory — each cell is transferred four times. Using the (shared) local memory as a cache would significantly improve the performance.

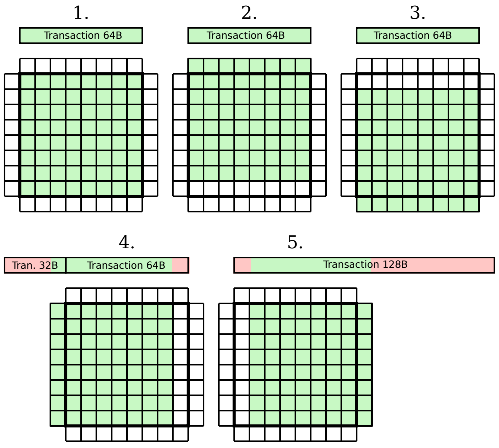

**Figure 2.** Read transactions performed by the algorithm.

## 3.2. Implementation with local memory

In the similar way as in the implementation with global memory, we defined two kernels that are sequentially executed for each cell:

- `wfca_best_levels` — calculates the expected average level of water in central and neighbouring cells,
- `wfca_distribute` — calculates the exchange of water between cells according to the previously calculated average level.

In order to use local memory, we introduced the following modifications in the code of the kernels:

- each thread in a workgroup reads its own cell and stores it in local memory where it is accessible for other threads in this workgroup,
- during calculation the data referring to the neighbouring cell are read from local memory,

- the selected threads have to additionally read neighbouring cells that lie outside the area assigned to this workgroup.

The general form of the algorithm remains the same. Modifications are connected with loading data thus we present only changes that have to be applied to both kernels.

The following declaration allocate buffer in local memory (shared memory in CUDA terms):

```
const int BUFFER_H = 18;
const int BUFFER_W = 19;
__local float buffer[BUFFER_W * BUFFER_H];
```

Width of buffer (`BUFFER_W`) is chosen to prevent bank conflicts when half-warp accesses local memory. Cells in one row can be accessed in parallel as they are located in different banks.

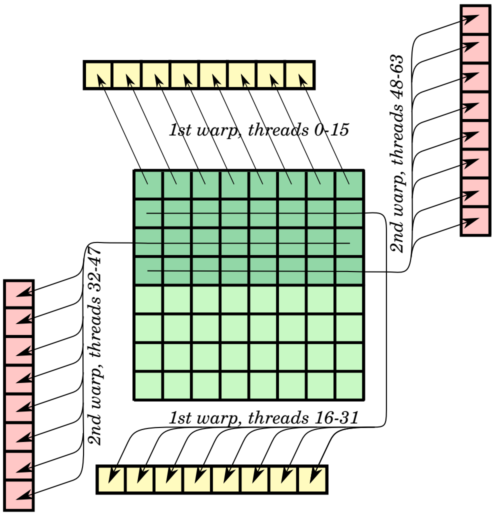

**Figure 3.** Threads located at the edge of the workgroup have to read additional data from area assigned to other workgroups (for the clarity workgroups has size limited to $8 \times 8$).

The code beneath implements loading data for cells that are assigned to this block of threads — green area in Figure 3:

```
int lx = get_local_id(0), ly = get_local_id(1);
int sx = get_local_size(0), sy = get_local_size(1);
buffer[(lx + 1) + (ly + 1)*BUFFER_W] = level[x + y*w];
```

Some threads have to execute a different branch of code and read cells that assigned to other workgroups. Normally, it causes that thread will not be executed in parallel (if they are in the same warp). We organized processing in our code in this way that such the threads are grouped into two warps. We use the threads from the first four rows of the workgroup to read additional cells that lie outside the block (see Fig. 3). We expect that it minimizes the chance of serialized execution. Warp is formed by threads with sequential indices, i.e. threads 0–31 form the first warp, threads 32-63 form the next and so on. Within the half-warp (16 threads), the threads should execute the same path of execution, i.e. read data for 16 cells in row or column. The code beneath implement this method:

```
if ( ly == 0 || ly == 1 ) { //One warp: 2*16 thread
     int gy = y - ly;
     int tb = (sy+1)*ly;
     //row below and beneath (yellow region)
     buffer[(lx+1) + (tb)*BUFFER_W] = level[x + (gy-1+tb)*w];
} else if ( ly == 2 || ly == 3) { //Second warp: 2*16 thread
     int gy = y - ly, gx = x - lx;
     int lr = (sx+1)*(ly & 1);
     //left and right column (red region)
     buffer[(lr) + (lx+1)*BUFFER_W] = level[(gx-1+lr) + (gy+lx)*w];
}
```

We must ensure that no part of the working group did not try to read the data that has not yet been stored in local memory. For this purpose, we use the local barriers that synchronize all threads:

```
barrier(CLK_LOCAL_MEM_FENCE);
```

In theory, this version should use the following number of transaction in global memory:

- $16 \times 64$B transactions for cells assigned to workgroup (green area),
- $2 \times 64$B transactions for below and beneath the workgroup area (yellow area),
- $32 \times 32$B transactions for left and right edge of the workgroup area (red area),
- $16 \times 64$B transactions that read `terrain` table.

In summary, we have $32 + 34 = 66$ transactions what is better than version with global memory, which has to do $16 + 5 \times 16 + 16 = 112$ transactions. Profiling results acknowledge these predictions (see Table 2).

**Table 2**
Number of transactions performed by version using shared memory. Grid has 512 × 512 cells and workgroup has 16 × 16 threads (1024 workgroups).

| Kernel | Loads 32B | | Loads 64B | | Store 64B | |
|---|---|---|---|---|---|---|
| | #32B | per w-g | #64B | per w-g | #64B | per w-g |
| `wfca_best_levels` | 32768 | 32 | 34816 | 34 | 16384 | 16 |
| `wfca_distribute` | 32768 | 32 | 67584 | 66 | 16384 | 16 |

## 4. Results

Our implementations were developed and tested on several Nvidia graphic cards. This producer provides good support for programmers on Linux platform, thus performance evaluation and profiling analysis was made for these processors. The features of tested processors are enumerated in Table 3.

**Table 3**
Some technical parameters of graphics cards used in tests.

| GPU | No of cores | Compute Capability | Memory Bandwidth |
|---|---|---|---|
| Nvidia Geforce 9800 GT | 112 | 1.0 | 57.6 GB/s |
| Nvidia Quadro NVS 140M | 16 | 1.1 | 11.2 GB/s |
| Nvidia Quadro FX 4800 | 192 | 1.3 | 76.8 GB/s |
| Nvidia Geforce GTX 295 | 2×240 | 1.3 | 223 GB/s |

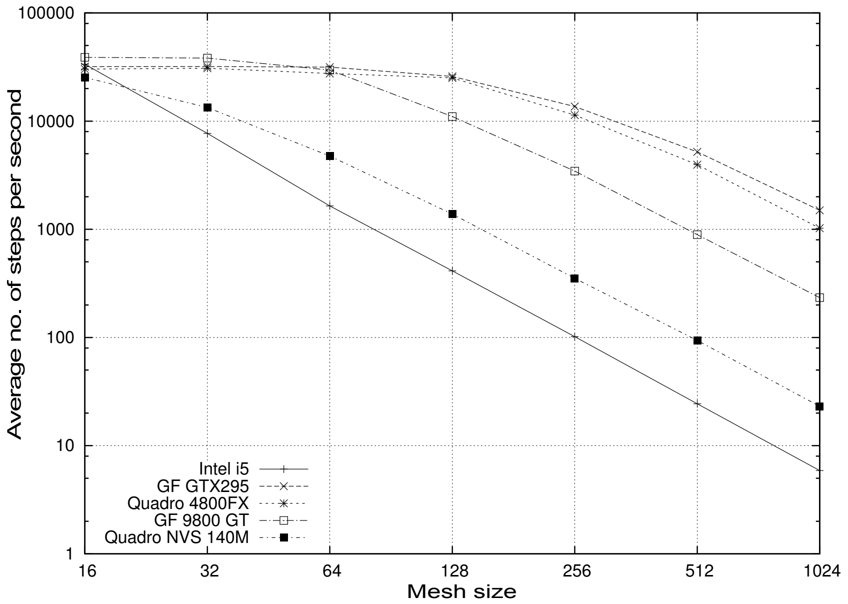

**Figure 4.** Performance achieved by algorithm with shared memory on several tested GPU compared to single core Intel i5.

Figure 4 presents best performance achieved for algorithm using shared memory. Program was tested on selected GPU and compared with their version for traditional CPU (Intel i5 2.8 GHz). The weakest GPU in our tests, Nvidia Quadro NVS 140M, outperforms CPU more 10 times. The most powerful GPU we tested, Nvidia Geforce GTX295, is about 250 times better than CPU. All GPUs present relatively lower performance for small size of lattice. It comes from the fact that such a small Cellular Automata is not able to fill all the processing units that GPUs provide.

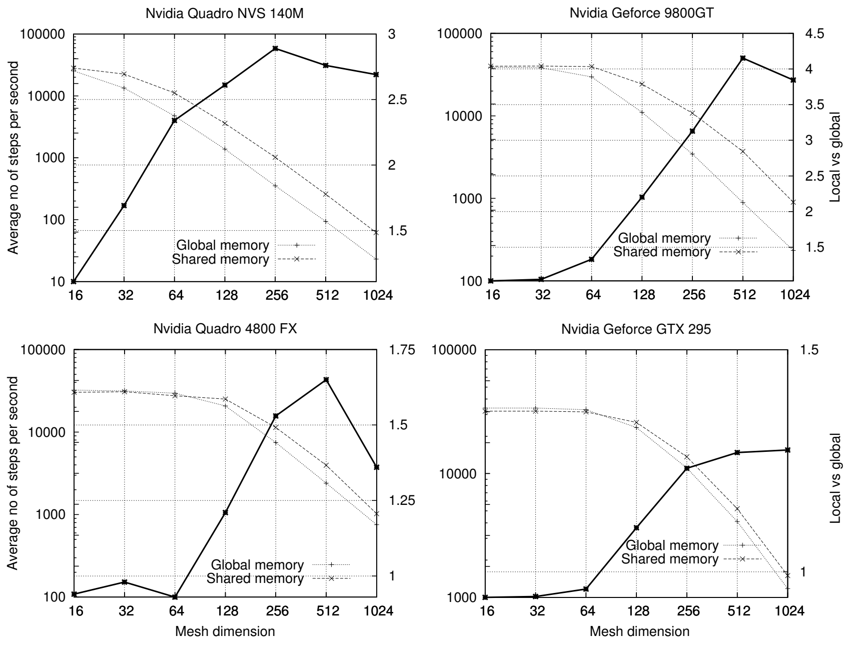

**Figure 5.** The comparison of the algorithm performance for different GPUs. Each chart demonstrates performance for algorithms with global and local memory. Performance ratio is also included (thick line).

Figure 5 demonstrates the benefits of using the algorithm with shared memory. The Nvidia Quadro NVS 140M chip is optimized for laptops and for low power consumption. Additionally, graphic card uses part of system memory as their own (Nvidia TurboCache technology). Version with shared memory is more than twice better than version with global memory. This GPU has only 16 stream processors, and this is the main reason of its low performance — the improvement resulting from the use of shared memory cannot be overstated.

The biggest gain from the use of shared memory we received for the Geforce 9800 GT (Nvidia G86 processor) card. The algorithm with shared memory is more than four times better then version with global memory. The benefit comes from bigger difference in bandwidth between global and shared memory. Nvidia Quadro FX 4800 (GT200GL chip) has a higher memory bandwidth, and the algorithm with shared memory is only up to 1.7 times better. The same trend is observed in the case of Nvidia Geforce GTX 295 (which is, in fact, a dual GT200 processor). This graphic card has 223 GB/s of theoretical bandwidth, and local memory version is only up to 1.2 times better than the version with the global memory.

## 5. Conclusions

In this paper, we have shown that GPUs can be used to accelerate the modeling of a physical phenomena with Cellular Automata paradigm. In our previous work [20] we present simple but very efficient algorithm that uses GPU and global memory. In this work we concentrate on efficient use of shared memory. We proposed modifications that allow to use the power of shared memory. Using a profiling tool, we found the bottleneck of our previous algorithm, which was redundant access to global memory. We decided to introduce the modifications that use fast shared memory as a cache. Efficient use of shared memory requires a thorough knowledge of its structure and operation. Our modifications take into account the special structure of this memory and do not introduce excessive overhead of operations. As the result, this implementation appeared to be up to four times better than previous one.

The unique result of our works is the comparison how the same algorithm behaves when only a global memory is used and when the shared memory is exploited. Our tests showed that efficient global memory can significantly reduce the profits from the application of shared memory, used even in most optimized manner.

Processors with Compute Capabilty 2.0 and higher (e.g. Fermi architecture) have cache for all transactions in global memory. We plan to carefully check whether these capabilities can effectively compete with manually managed shared memory.

Our future works will also focus on searching for other ways of optimizations. We plan to investigate other mapping schema, e.g. one work-item is processing many cells or the work-group has shape other than square.

The algorithms can be used in the future as a basis for other models of flow phenomena like the flow of water, lava or mud. Moreover, the shared memory management schema we use in our algorithm can be also efficiently used in another cellular automata model implemented on GPU.

### Acknowledgements

*The authors would like to thank the anonymous reviewers for their inspiring comments and suggestions. This research is partially financed by the Polish Ministry of Higher Education and Science, project NN519443039 and partially by AGH-UST grant No. 11.11.120.865 The authors also acknowledge BioGeoLab, Institute of Geological Sciences, Polish Academy of Sciences.*

## References

[1] Wolfram S.: *A New Kind of Science.* Wolfram Media, Inc. 2002.

[2] Chopard B., Droz, M.: *Cellular Automata Modeling of Physical Systems.* Cambridge University Press, 1998.

[3] Owens J. D., Houston M., Luebke D., Green S., Stone J. E., Phillips J. C.: *GPU computing.* In *Proceedings of the IEEE*, vol. 96(5), 2008, pp. 879–899.

[4] *GPU Computing Gems Jade Edition.* Applications of GPU Computing Series, editor-in-chief Wen-mei W. Hwu, Elsevier, 2011.

[5] Kurdziel M., Boryczko K.: Dense affinity propagation on clusters of GPUs, In *Parallel Processing and Applied Mathematics, 9th international conference, PPAM 2011*, Toru, Poland, September 11–14, 2011 : revised selected papers, Pt. 1 / eds. Roman Wyrzykowski et al. Lecture Notes in Computer Science, Springer-Verlag, vol. 7203, 2012, pp. 599–608.

[6] Marks M., Jantura J., Niewiadomska-Szynkiewicz E., Strzelczyk P., Gd K.: Heterogenous GPU&CPU cluster for high performance computing in cryptography, *Computer Science*, vol. 13(2):63–79, 2012.

[7] CUDA Zone `http://www.nvidia.com/object/cuda_home_new.html`

[8] Khronos Group `http://www.khronos.org/opencl/`

[9] Rybacki S., Himmelspach J., Uhrmacher A. M.: Experiments with Single Core, Multi-core, and GPU Based Computation of Cellular Automata, In *Advances in System Simulation, 2009. SIMUL'09. First International Conference on, 2009*, pp. 62–67.

[10] Gobron S., Finck D., Even Ph., Kerautret B.: Merging Cellular Automata for Simulating Surface Effects, In Cellular Automata, Yacoubi S., Chopard B., Bandini S. eds., Lecture Notes in Computer Science, vol. 4173, 2006, pp. 94–103.

[11] Gobron S., Coltekin A., Bonafos H., Thalmann D.: GPGPU Computation and Visualization of Three-dimensional Cellular Automata. *The Visual Computer*, Springer Berlin/Heidelberg, vol. 27(1):67–81, 2011.

[12] Gobron S., Devillard F., Heit, B.: Retina Simulation using Cellular Automata and GPU Programming, *The Machine Vision and Applications Journal*, Springer Berlin/Heidelberg, 18(6):331–34, 2007.

[13] Bilotta G., Rustico E., Hérault A., Vicari A., Russo G., Del Negro C., Gallo G., Porting and optimizing MAGFLOW on CUDA. Annals of Geophysics, 54(5):580–591, 2011.

[14] Vicari A., Hérault A., Del Negro C., Coltelli M., Marsella M., Proietti C. Modeling of the 2001 lava flow at Etna volcano by a Celluar Automata approach *Environ. Modell. Softw.*, 22(10):1465–1471, 2007.

[15] Rustico E., Bilotta G., Hrault A., Del Negro C., Gallo G. Scalable multi-GPU implementation of the MAGFLOW simulator. *Annals Of Geophysics*, 54(5):592–599, 2011.

[16] Ferrando N., Gosalvez M. A., Cerda J., Gadea R., Sato K.: Octree-based, GPU implementation of a continuous cellular automaton for the simulation of complex, evolving surfaces. *Computer Physics Communications*, 182(3):628–640, 2011.

[17] Quesada-Barriuso P., Heras D. B., Arguello, F.: Efficient GPU Asynchronous Implementation of a Watershed Algorithm Based on Cellular Automata: In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, 2012, pp. 79–86.

[18] Caux J., Siregar P., Hill D.: Accelerating 3D Cellular Automata Computation

with GP-GPU in the Context of Integrative Biology. In *Cellular Automata – Innovative Modelling for Science and Engineering*, Alejandro Salcido (ed.), InTech, 2011, ISBN: 978-953-307-172-5.

[19] Miao Q., Lv Yisheng, Zhu, F.: A cellular automata based evacuation model on GPU platform. In *Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference on*, 2012, pp. 764–768.

[20] Topa P., Mocek P.: GPGPU implementation of Cellular Automata model of water flow. In *Parallel Processing and Applied Mathematics : 9th international conference, PPAM 2011*, Toruń, Poland, September 11–14, 2011 : revised selected papers, Pt. 1 / eds. Roman Wyrzykowski et al. Lecture Notes in Computer Science, Springer-Verlag, vol. 7203, 2012, pp. 630–639.

[21] Di Gregorio S., Serra R.: An empirical method for modelling and simulating some complex macroscopic phenomena by cellular automata. *Future Generation Computer Systems*, 16(2–3):259–271, 1999.

[22] Topa P.: River Flows Modelled by Cellular Automata. In *Proceedings of 1st SGI Users Conference*, Cracow, Poland, ACC Cyfronet UMM, 2000, pp. 384–391

[23] Topa P.: A Distributed Cellular Automata Simulation on Cluster of PCs. In *Proceedings of the International Conference on Computational Science-Part I (ICCS '02)*, eds. Peter M. A. Sloot et al. Lecture Notes in Computer Science, vol.2329, Springer Berlin/Heidelberg, 2002, pp. 97–106.

[24] Topa P., Paszkowski M.: Anastomosing Transportation Networks. In *Proceedings of 4th International Conference on Parallel Processing and Applied Mathematics 2001*, eds. Wyrzykowski R. et al. Lecture Notes in Computer Sciences vol. 2328, Springer-Verlag Berlin/Heidelberg, 2002, pp. 904–911.

[25] Topa P., Dzwinel W., Yuen D.: A multiscale cellular automata model for simulating complex transportation systems, *International Journal of Modern Physics C*, 17(10):1–23, 2006.

## Affiliations

**Paweł Topa**
AGH University of Science and Technology, Department of Computer Science, Krakow, Poland, `topa@agh.edu.pl`,
Institute of Geological Sciences, Polish Academy of Sciences, Research Centre in Kraków, Krakow, Poland

**Paweł Młocek**
AGH University of Science and Technology, Department of Computer Science, Krakow, Poland, `pawel.mlocek@gmail.com`

**Received**: 20.01.2013
**Revised**: 11.02.2013
**Accepted**: 28.02.2013