GPU-BASED EXTENDED CELLULAR MODEL IMPLEMENTATION

A. A. Emelyanov, R. M. Dmitrienko
Special Computing Technologies LLC
N.Novgorod, Russian Federation
office@hopcomp.net

A Cellular Automaton (Cellular Model) is a convenient and efficient way to solve a broad class of problems that belong to different areas of science, including (but not limited to) physics, math, chemistry, biology. A lot of natural phenomena and common tasks that are described via differential equations are easily modeled this way. But in spite of obvious advantages, CA-based algorithm implementations for common CPUs are highly parallel and hence extremely resource-intensive. Thus, the idea of SIMD GPU-based CA implementation looks feasible. In this paper, we will discuss the problems we faced while developing GPU-based cellular model implementation, and the ways we solved them.

CA – Cellular Automaton

SIMD – Single Instruction Multiple Data

GPU – Graphics Processing Unit

I. INTRODUCTION

A Cellular Model (a Cellular Automaton) is a set of cells and their corresponding states. These cells form a grid; a certain set of rules determines each cell's new state depending on the nearby cells' current states at any given moment of time. The most common are the automata whose cells' new states are determined only by their own current state and those of corresponding adjacent cells. Cubic grids are the most popular, but the irregular grids are possible as well. Different kinds of cellular automata are studied: synchronous and asynchronous, deterministic and probabilistic, with regular and irregular cell disposition. An Extended Cellular Model is a generic mathematical model of a Cellular Automaton that encapsulates all of the aforementioned kinds of Cellular Automata.

One of the most important properties of an Automaton is the “Locality of the rules”. It means that the cell's new state is only determined by the adjacent cells and probably by itself. This fact allows us to suppose that it is possible to implement a Cellular Automaton using a SIMD processing unit, such as a GPU. First we'll discuss the limitations of the SIMD architecture (as implemented by a GPU) that are related to the problem of efficient Cellular Automata implementation.

II. PROBLEM DEFINITION AND ANALYSIS

When it comes to implementation of highly parallel GPU-based algorithms, the most important limitation that arises is caused by the memory access collisions. A memory access collision is a situation when more than one of the active threads attempt to access memory either for reading, or for writing. At the hardware level, this problem is usually partially solved by the means of memory organization: either a collision-free memory paragraph access method, or a collision-free separated memory bank access, where each bank is spread across the whole address space. Thus, the problem of efficient algorithm implementation is solved by choosing an optimal layout of input and output data in the memory and an optimal sequence of memory access attempts by the threads.

From the software implementation point of view, a Cellular Automaton is a closely coupled net of finite state machines (the cells), where the connections represent the input and output data of these FSMs. From the programming point of view, the only difference between a Synchronous and an Asynchronous Cellular Automata is the global cell states' buffer that is required by a Synchronous CA. Considering the fact that the GPU threads access memory at different moments of time, it is possible to implement both Synchronous and Asynchronous Cellular Automata equally efficiently (when all of the other conditions are the same). A Movable Cellular Automaton may be reduced to an ordinary Cellular Automaton by adding the cell's coordinate to the cell's state variables list, while establishing connections between each and every cell. Thus, instead of recalculating cell adjacency before new cell states' calculation, it is safe to assume that every other cell is a neighbor of the current cell, and hence it becomes possible to use these “neighbor”'s coordinates for state recalculation. This kind of algorithm transformation does not increase the processing load, as the quantity of operations remains the same, only their order is changed. Implementing determined and probabilistic Cellular Automata is even simpler. Probabilistic state recalculation algorithms are slightly more complex, but since the random numbers (which are essential in this case) are calculated independently for each cell, and it is possible to use GPU registers to store such small amounts of data, memory access collisions are avoided. Furthermore, it is possible to precalculate the random numbers' sequence and

store it optimally in the memory before launching the computing algorithm.

So, the only problem of the GPU-based Cellular Model implementation that remains is the irregularity of the grid. Let's suppose that the cells' connections are predetermined and bear no regular structure. This is highly likely to cause collisions when threads will attempt to access memory areas that store the parameters of the cells, since the order of access attempts is not known beforehand (it depends only on the Cellular Automaton's cells' interconnections). It is also important to note that this problem is irrelevant for movable Cellular Automata, as the cells' interconnection structure is unknown beforehand at the beginning of every calculation step (and thus must be recalculated); so, it is possible to connect each and every cell, while maintaining the same efficiency. However, this is not the case for Automata with irregular grids: there can be much less actual connections than the squared number of cells, so that the SIMD architecture-based implementation's efficiency will drop significantly.

Based on the aforementioned suggestions, the problem that we need to solve is actually a problem of designing of an efficient implementation of a Cellular Automaton with irregular connections' structure. Implementing a Cellular Automaton with regular connections' structure is just a special case of this problem.

III. The proposed solution

Each cell contains its state variables and a list of neighboring cells. In-memory data layout is done this way:

a_i - cell i

U^i - a set of neighbors of cell i

B^i - a set of cells, for which the cell I is a neighbor

$a_j \in B^i \Leftrightarrow a_i \in U^j$

$M(A)$ - cardinality of the set A

Step 1. The cells are ordered by the number of cells that a cell is adjacent to, beginning from the greatest number.

$$M(B^i) \geq M(B^{i+1})$$

Step 2. Cells' state variables are laid out this way: the first variables of cells' states are stored beginning from a memory paragraph's boundary, then the second variables are stored beginning from the next paragraph's boundary, and so on.

p_j^i - state variable j for then cell i

$S(p_j)$ - size of the state variable j

PH - size of a memory paragraph

N - quantity of cells

In-memory layout of parameters:

$$p_1^1\ p_1^2\ p_1^3\ p_1^4 \ldots p_1^N\ SPACE_1$$
$$p_2^1\ p_2^2\ p_2^3\ p_2^4 \ldots p_2^N\ SPACE_2$$
$$\ldots$$
$$p_K^1\ p_K^2\ p_K^3\ p_K^4 \ldots p_K^N\ SPACE_K$$

where K is the number of state variables.

SPACE – memory boundary alignment space

Size of

$$SPACE_j = \left(PH - \left(\left(S(p_j) * N \right) mod PH \right) \right) mod PH$$

where a mod b is a reminder of a/b.

It does not matter whether all of the state variables fit into the same memory paragraph or not. Nevertheless, it is important to keep the number of used paragraphs low while avoiding memory access collisions.

Step 3. The list of cell's neighbors is a sequence of cells' numbers (according to the one we've got at the Step 1); the fields that correspond to the non-neighboring cells are filled with 0 (-1 may be used if 0 is taken by the first cell).

A separate thread is launched per each cell. A single thread may correspond to a number of cells in case the total amount of cells exceeds the thread count limit. Each thread reads its corresponding cell's neighbors' states and recalculates the cell's state variables accordingly. If the neighboring cell is marked as 0 (i.e. there's no such neighbor), the thread enters waiting state until a valid cell is found.

Memory access collision avoidance and processing efficiency are achieved by the fact that the waiting threads attempt no memory access, and the number of such threads may grow significantly while the neighbor list is being processed. It is worth mentioning that SIMD threads are always synchronous and hence are put into waiting state only when they miss a conditional branch of the code which is entered by some other threads (they remain in the waiting state until others exit the branch). Thus, this algorithm may become especially efficient when applied to Cellular Models with irregular cells' disposition (where the efficiency depends significantly on the structure of connections). A major performance gain is also possible for Asymmetric Cellular Automata, where the density of cellular interconnections is distributed unevenly and has certain distinct peaks and troughs.

IV. Conclusion

We have described our approach to the GPU-based Cellular Models implementation. This approach is frequently employed for our development which is done in conjunction with subject matter experts from different areas of science, such as physics, chemistry, meteorology and so on. We have implemented it using NVidia GPUs (as a part of the CUBLIC(TM) project), and the estimated performance gain (as compared to the transformation of the source Automaton into an Automaton where each and every cell is connected to every other cell) ranges from 2 to 5 times.

References

[1] Kalgin Konstantin Victorovich KalginKV@gmail.com Supercomputer Software Department (SSD, ssd.sscc.ru), “Cellular Aitomata on GPU” *http://ssdonline.sscc.ru/kalgin/cuda/ca.gpu.pdf*

[2] Psakhie, S.G.; Horie, Y.; Korostelev, S.Yu.; Smolin, A.Yu.; Dmitriev, A.I.; Shilko, E.V.; Alekseev, S.V. (1995). «Method of movable cellular automata as a tool for simulation within the framework of mesomechanics». Russian Physics Journal 38 (11)

[3] Psakhie, S.G.; Horie, Y.; Ostermeyer, G.P.; Korostelev, S.Yu.; Smolin, A.Yu.; Shilko, E.V.; Dmitriev, A.I.; Blatnik, S.; Spegel, M.; Zavsek, S. (December 2001). «Movable cellular automata method for simulating materials with mesostructure». Theoretical and Applied Fracture Mechanics 37 (1-3)