10th International Conference on Hydroinformatics
HIC 2012, Hamburg, GERMANY

CADDIES: A NEW FRAMEWORK FOR RAPID DEVELOPMENT OF PARALLEL CELLULAR AUTOMATA ALGORITHMS FOR FLOOD SIMULATION

MICHELE GUIDOLIN, ANDREW DUNCAN, BIDUR GHIMIRE, MIKE GIBSON, EDWARD KEEDWELL, ALBERT S. CHEN, SLOBODAN DJORDJEVIĆ, DRAGAN SAVIĆ
Centre for Water Systems, University of Exeter, North Park Road, Exeter, EX44QF, UK

A recent trend in the development of flood simulation algorithms shows the move toward fast simplified models instead of slow full hydrodynamic models. CADDIES is a research project that aims to develop a real/near-real time pluvial urban flood simulation model using the computational speed of cellular automata (CA) algorithms. This paper presents a component of the software framework that is part of the CADDIES project. Its objective is to simplify the development of CA algorithms and their acceleration using modern high performance hardware and techniques. The performance results obtained on a simple case study clearly demonstrate the effectiveness and efficiency of the CADDIES framework.

INTRODUCTION

CADDIES (Cellular Automata Dual-DraInagE Simulation) is a research project that aims to develop cellular automata (CA) algorithms [1] for fast pluvial flood modelling. The objective is to improve the speed and efficiency of flood modelling that involves both urban surface and drainage system (2D urban surface flow and 1D sewer flow) for real/near-real time applications.

Typical flood simulations use full hydrodynamic models, which are computationally expensive at finer resolution, and may easily take many hours to complete even if run on modern hardware. CA algorithms do not solve the full hydraulic equations but employ simple operations of the CA rules; thus execution speeds could be orders of magnitude faster. Furthermore, CA algorithms are well suited for parallel execution on modern high performance hardware, since the computation of a cell's state depends only on the previous state of the local neighbouring cells and its previous state.

However, defining a complex physical model using simple CA rules and developing algorithms that take full advantage of the parallel computation of modern hardware is not a trivial process. Therefore, there is a need for a software tool that allows developers to prototype parallel CA algorithms rapidly. This should happen automatically so that users can easily profit from modern hardware architecture.

The CADDIES project includes a new software framework that provides developers with various tools that simplify development of parallel CA algorithms for pluvial flood simulation. The framework is composed of a graphical application and a CA application programming interface (API). The former is used to manage and visualise input and output data as well interfacing with GIS [2]; thus the need for the developers to invest valuable time

and resources on the creation of input and output code, GIS interface and visualisation tools is reduced. The CA API defines a standard set of methods, data structures and variables that can be used to develop parallel CA algorithms. The CA API can handle different implementations depending on the type of hardware and CA algorithm used. Thus it allows a developer to focus mainly on writing the rules of the CA algorithm, while the data management, parallelisation and execution of the algorithm is handled by the specific implementation of the CA API.

In this paper, the design and objectives of the CADDIES CA API are presented. The usage of the CA API is then illustrated using as example a CA algorithm for flood modelling developed in the CADDIES project [3]. Then, two implementations of the CA API available in the CADDIES framework are introduced: a simple serial version that uses a single CPU for the computation and a highly parallel implementation that uses the graphics processing unit (GPU) for the computation. The effectiveness and efficiency of the CA API is tested by analysing the performance of this CA algorithm using these two implementations of the CA API and by comparing their performance with the one of the physical based non-inertial urban inundation model (UIM) developed by Chen et al.[4].

CELLULAR AUTOMATA ALGORITHM DEVELOPMENT

Cellular automata offer a versatile method for deriving reduced computational load for models of complex physical systems [5]. A CA is a discrete model which is typically composed of a regular grid of cells defined by a discrete location and a set of states. The evolution of each cell's state is governed by local transition rules that use the previous state of the cell and the previous states of the neighbouring cells of the CA. Since the computation of the transition rules in each cell uses the data of the surrounding cells of the previous steps (it does not depend on the data at the same time step), CA algorithms are well suited to parallel computation.

Various CA algorithms can be defined not only by different transition rules but also by the underlying CA grid used. Changing the attributes of the CA grid used by the algorithm can change how accurately a physical system is modelled. However, it is difficult to know in advance which type of CA grid is more effective.

A CA grid can vary on the following attributes: the type of cell used (rectangular, hexagonal, triangular, etc.); the structure of the CA grid implemented (regular, rectilinear, unstructured, etc.); the number of neighbourhood cell used; the type of action to execute on the border cells (fixed value, function, wrap-around values, etc.) and the number and type (e.g. discrete, continuous) of cell states. Thus, to model a physical system accurately using a CA algorithm, a developer needs not only to write the CA rules that best define the system, but also to find the best attributes of the underlying CA grid for these rules to obtain the most accurate results.

CADDIES CELLULAR AUTOMATA API

The two main objectives of the CADDIES CA API design are: 1) to simplify the development of CA algorithms for flood modelling allowing the highest flexibility possible; and 2) to simplify the accelerated execution of CA algorithms using modern high performance hardware and techniques.

A CA algorithm written using the CADDIES CA API is composed of three parts:

- The CA execution, which contains the loading, initialisation and management of the data as well as the data flow control, i.e., the main computational loop.
- The CA transition functions, which define the transition rules of the CA algorithm, i.e., the computation that is executed in each cell of the CA grid.
- The CA options, which define the attributes of the CA grid such as cell type, grid type, border cell type, etc.

The main idea of the CADDIES CA API is that a developer needs to write the code of the CA execution and the various CA transition functions only once. After that, the CA API can be used to produce the same CA algorithm for any type of CA grid. Furthermore, once a developer has selected the rules and CA grid attributes and tested the CA algorithm using serial computation, he/she should be able to execute the same CA algorithm using high performance acceleration techniques without changing the code or with minimum effort.

Since a code written using the CA API can produce CA algorithms that use grids of different attributes and run on different hardware platforms, there are going to be different implementations of the CA API. This will depend on the CA options and hardware targeted. However, run-time flexibility can result in slow code execution. To avoid that situation, the CA API implementation desired is chosen at compile time. This produces the fastest possible execution speed.

The methods, data structures and variables of the CA API are divided in two groups, which are each designed to perform different tasks. The entities of the first group, based on C++ language (classes, objects and methods), are used in the code of the CA execution. While entities of the second group, based on the C language (constant, methods and macros), are used in the code of the CA transition functions. This difference in the language used is necessary, since a CA transition function can be executed on specific hardware, such as a GPU, where a C++ compiler might not be supported or might have a limited implementation.

In the following part of this section, the use of the CA API in defining a cellular automaton (the CA transition function) and in the execution of the CA itself(the main loop) is introduced by using as example the initial two-dimensional CA algorithm developed for the CADDIES project [3]. In this algorithm, the movement of water is modelled by calculating the outflows from each cell based on ranking the water surface elevation of the cells within the neighbourhood.

CA Transition Function

A CA transition function represents the computation made in each cell, i.e., the transition rules. The order of computation of the cells can be arbitrary during the computation of the

CA transition function and can be executed in parallel, i.e., multiple cells can be computed simultaneously. The CA grid to which the CA transition function is applied can be of any type, see Figure 1, and it is chosen at compile-time.

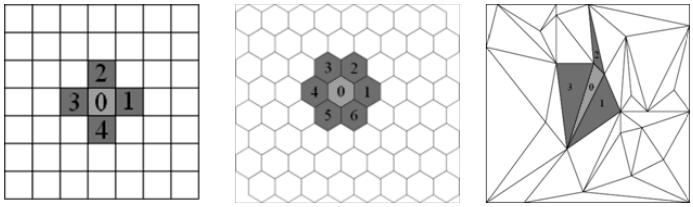

Figure 1. Example of three CA grids with different attributes

Multiple CA transition functions can be called during the execution of the CA algorithm. Each CA transition function can take a different number of buffers as input and output parameters. The values in the buffers represent the data used in the computation of the CA transition function. The buffers can be of different types: a cell buffer which contains a value for each cell of the CA grid, an edge buffer which contains a value for each edge of the grid, and a vertex-buffer which contains a value for each vertex of the grid. Every type of buffer can contain discrete (state) or continuous (real) values. It is also possible to retrieve the information about the attributes of the cell and the underlying grid in a CA transition function. For example, these could be the number of edges, edge length, the number of neighbours, area of cell, etc.

During the computation of a cell, to read/write all the values of a buffer, i.e., to access every cell of the grid, is not permitted. It is only possible to read/write the values of a limited set of visible cells. The visible cells (neighbourhood) are composed of the visited cell on which the transition rules are computed (called main cell) and its neighbour cells. The number of neighbour cells depends on the attributes of the CA grid chosen.

In a CA transition function, two types of indices, a global index and a local index, can be used to access the values of a buffer and the cell information. The former is used to index the main cell in the buffers, while the local index is used to identify a cell between the visible cells. The local index of the main cell is 0, see Figure 1.

Figure 2 shows an example of the CA API usage in the code of a CA transition function. This function computes the outflow fluxes for each edge of the main cell given the ranking of the elevation and water level within the neighbourhood. The function and its parameters are defined in the first 4 lines. The parameters are: the CA grid, an input/output real values edge buffer (which will contain the newly computed outflow), and two input real values cell buffers (one which contains the elevation, and one which contains the water level). In line 5, the CA grid parameter is initialised; this parameter must be passed as the first argument of each method of the CA API used in the CA transition function.

In line 6 and 7, two arrays of real values are created; the sizes of these arrays are the number of visible neighbours plus one and the number of edges in a cell plus one, respectively. The additional ones are used to store also the main cell (index 0). The values of these two numbers change at compile time depending on the attributes of the CA grid

chosen, i.e., both are 4 for the first grid of Figure 1, both are 6 for the second grid and both are 3 for the third grid. In line 8, the global index in the buffer of the main cell, local index 0, is retrieved. In line 10 and 11, the values of the neighbour and the main cells from the elevation buffer, and the values of the edges of the main cell from the outflow buffer are loaded in the two local arrays, respectively. These arrays are then used as normal C-language arrays to loop through the values/and attributes of the visible cells and edges.

```
 1 CA_FUNCTION outflow(CA_GRID grid,
 2                     CA_EDGEBUFF_REAL_IO OUTF,
 3                     CA_CELLBUFF_REAL_I  ELV,
 4                     CA_EDGEBUFF_REAL_I  WL) {
 5   CA_GRID_INIT(grid);
 6   CA_ARRAY_CREATE(grid, CA_REAL, aelv,  caNeighbours+1);
 7   CA_ARRAY_CREATE(grid, CA_REAL, aoutf, caEdges+1);
 8   CA_INDEX index = caIndex(grid,0);
 9   ...
10   caReadCellBuffRealCellArray(grid, ELV, aelv);
11   caReadEdgeBuffRealEdgeArray(grid, ELV, index, aoutf);
```

Figure 2. An example of CA transition function code that uses the CADDIES CA API

CA Execution

The CA execution contains the code that loads, initialises and manages the data using the main CA API. Thus, the CA grid is created in the code of the main loop. While the CA API is designed to have different type of CA grid during the computation of the CA transition functions, the CA grid in the CA execution is always considered to be a regular square grid to simplify the management of the data.

```
 1 #include CA_2D_INCLUDE(outflow)
 2
 3 int main(...){
 4  ...
 5  CA::Grid         GRID(ncols,nrows,cellsize);
 6  CA::CellBuffReal ELV(GRID);           // Elevation
 7  CA::CellBuffReal WL(GRID);            // Water Level
 8  CA::EdgeBuffReal OUTF(GRID);          // Outflow
 9  CA::Box          area(GRID.box());    // Computational Area
10  ...
11  while(loop) {
12   // Execution of the "outflow" CA transition function
13   CA::Execute::function(area,outflow,GRID,OUTF,ELV,WL);
14   ...
15  }}
```

Figure 3. An example of CA execution code that uses the CADDIES CA API

Figure 3 shows an example of the main CA API usage in the CA execution C++ code. In the first line, the code includes the CA transition function called "outflow". In the main, the code creates a CA grid object using the class CA::Grid (line 5). The given parameters define the columns, rows and cell size of the CA grid. The following objects represent respectively two buffers (line 6 and 7) that contain a real value for each cell of the CA grid, which are used to store the elevation (ELV) and water level (WL) of the desired area, and a buffer (line 8) that contains a real value for each edge of the CA grid (CA::EdgeBuffReal),

which is used to store the outflow fluxes (OUTF). Inside the while loop (line 13), the CA transition function "outflow" is called using the following arguments: the area of the grid where to perform the computation (the full CA grid in this example), the name of the function, the CA grid, the outflow fluxes buffer, the elevation and the water level buffers.

CA API IMPLEMENTATIONS

As previously stated, the CADDIES CA API can have multiple implementations. These can vary depending on the attributes chosen for the CA grid and/or on the type of hardware targeted. In this work, two implementations are presented. In both, the CA grid used in the algorithm is a regular one with square cell and one level of Von Neumann neighbourhood, as the first grid of Figure 1. However, they differ in the final hardware targeted. The first implementation creates the CA grid and the buffers using simple memory arrays, executing the CA transition functions on the CPU; thus the computation of the generated CA algorithms is set to be completely serial even if it is executed on a multi-core CPU. The second implementation uses the OpenCL library [6] to create the CA grid and the buffers. This library adds specific extension to the C language to execute code on the GPU of graphics cards which is highly specialised for parallel computations [7]. Thus the computation of the generated CA algorithms is highly parallel and run on the GPU.

OpenCL uses the idea of an external kernel to represent a computation on a device. A kernel is a function that is executed by a larger number of concurrent threads, i.e., independent similar computations. The OpenCL library creates and manages the synchronisation of the kernel's threads as well the data used by the kernel. The OpenCL implementation of the CA API transforms the CA transition function in a kernel; thus the CA transition function is executed by a very large number of concurrent threads, of the order of thousands, which uses the graphics card memory for the cell/edge buffers.

Figure 4 shows in simple terms the difference between the CPU implementation of the CA API and the GPU OpenCL implementation performing the computation of a CA transition function. In the case of the simple serial implementation, the CA transition function is executed by a single thread which performs the computation one cell a time. In the case GPU, the CA transition function is executed in each cell of the CA grid by an individual thread independently.

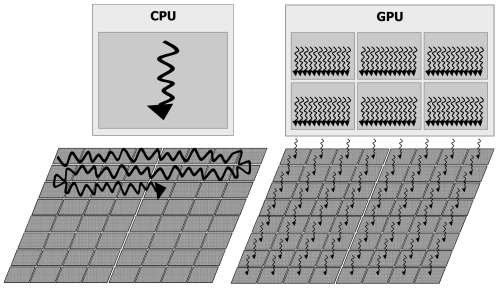

Figure 4. Execution of CA transition function on a single CPU, multi-core CPU and GPU

EXPERIMENTAL RESULTS

In order to show the effectiveness and efficiency of the CA API in creating CA algorithms that can be easily executed in parallel on modern hardware, the performance and the numerical results of two implementations of the CA API are compared using the physical based UIM model as reference. The CA algorithm used in these tests is the one that computes the outflow of a cell by ranking the water surface elevation of the cells within the neighbourhood [3].

The case study used in these tests is the Stockbridge area in Keighley, UK. The size of the area considered is around 0.4 km^2. The elevation buffer is loaded using a digital elevation model (DEM) of 2m resolution obtained from LiDAR. The CA grid is composed of 377x269 cells with open type boundary set, i.e., water is free to move out if the topography allows. An effective rainfall of 42.3 mm/hr intensity, which corresponds to 100 year return period with 1-hour storm duration, is applied uniformly over the terrain under consideration [8]. The total simulation time in these tests is three hours.

The CA API code used to generate the CA algorithm is the same for both implementations. This experimental results show how is possible to easily run a CA algorithm with higher performance by simply changing the underlying implementation of the CA API used, without changing the code of the algorithm.

The tests are executed on two hardware configurations. The first machine contains an Intel Core 2 Quad Q8300 (four core CPU– 2.5 GHz) and NVIDIA GeForce GTX 285 GPU (240 CUDA cores – 648 MHz) which runs an Ubuntu 10.04 64bit Linux system. The second machine contains Intel Core I5-2500K (four core CPU – 3.2 GHz) and NVIDIA GeForce GT 550 TI (192 CUDA cores – 900 MHz) which runs Windows 7 64bit system.

Table 1. A performance and accuracy comparison of the two different CA API implementations applied to the same CA algorithm

	UIM	Simple Serial			OpenCL			
M	Time (s)	Time (s)	S_p^{UIM}	RMSE (m)	Time (s)	S_p^{Ser}	S_p^{UIM}	RMSE (m)
1	10371.1	280.9	36.9	0.027	50.8	5.5	204.2	0.027
2	5885.7	161.6	36.4	0.029	44.5	3.6	132.3	0.029

Table 1 shows the runt time of UIM and the average run time of ten executions of the CA Algorithm tested on the two machines using the two different implementations. Furthermore it shows the speed-up of the two implementations over UIM (S_p^{UIM}) and the speed-up of the OpenCL implementation over the serial one (S_p^{Ser}). The CA algorithm that uses the simple serial implementation is over thirty times faster than the UIM model. Furthermore the CA algorithm that uses the GPU for the main computation, i.e., the OpenCL implementation, runs between three to six times faster and thus over 100 times faster than UIM. The difference in speed-ups between the two machines is due to the difference in performance between the CPU and GPU installed. Table 1 also shows the root mean square error (RMSE) value of the maximum flood inundation of each test using the depth given by

the UIM model. The CA algorithm produces numerical comparable results, irrespective of the CA API implementation used.

CONCLUSION

This paper introduces the cellular automata application programming interface (CA API) that is part of the software framework of the CADDIES project. Thanks to the methods, variables and class of this API, it is possible to simply develop CA algorithms for flood simulation that use different types of CA grid. Thus a developer can mainly focus on finding the CA rules and the CA grid that best model the desired physical system. The tests performed show that CA based algorithms for pluvial flood modelling can be more than an order of magnitude faster than physical based model such as UIM. Furthermore, by using the CA API it is possible to accelerate even more any CA algorithm produced effortlessly once the code is written.

ACKNOWLEDGMENT

The authors would like to acknowledge the funding provided by the U.K. Engineering and Physical Sciences Research Council, grant GR/J09796 (Cellular Automata Dual Drainage Simulation, CADDIES). The authors would like to thank the U.K. Environment Agency for supplying the LiDAR datasets.

REFERENCES

[1] Ilachinski, *Cellular automata: a discrete universe*. World Scientific Pub Co Inc, 2001.

[2] M. Guidolin, A. Duncan, E. Keedwell, A. S. Chen, S. Djordjevic, and D. Savic, “Design of a graphical framework for simple prototyping of pluvial flooding cellular automata algorithms,” in *Urban Water Management: Challenges and Opportunities*, Exeter, 2011, vol. 1, pp. 205-210.

[3] B. Ghimire, A. S. Chen, S. Djordjevic, and D. Savic, “Application of cellular automata approach for fast flood simulation,” in *Urban Water Management: Challenges and Opportunities*, Exeter, 2011, vol. 1, pp. 265-270.

[4] S. Wolfram, “Cellular automata as models of complexity,” *Nature*, vol. 311, pp. 419-424, Oct. 1984.

[5] A. Munshi and others, “The OpenCL specification version 1.1,” *Khronos OpenCL Working Group*, 2011.

[6] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, “GPU Computing,” *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879-899, May 2008.

[7] B. Ghimire, A. S. Chen, M. Guidolin, S. Djordjevic, and D. A. Savic, “A New Two-Dimensional Cellular Automata Approach For Fast Urban Flood Inundation Modelling,” in *Understanding Changing Cimate and Environment and Finding Solutions*, Hamburg, Germany, 2012.

[8] A. Chen, S. Djordjevic, J. Leandro, and D. Savic, “The urban inundation model with bidirectional flow interaction between 2D overland surface and 1D sewer networks,” *NOVATECH 2007*, 2007