Implementing the Global Cellular Automata on CUDA

Benjamin Milde, Niklas Buescher, Michael Goesele

Technische Universität Darmstadt

Abstract: The Global Cellular Automata (GCA) is a generalization of the Cellular Automata. As a massively parallel model, it is used to describe complex systems and algorithms in a coherent way. In this work, we evaluated how the generic GCA model transfers to NVIDIA's CUDA architecture on GPUs using two exemplary GCA algorithms. We compared our CUDA implementations with parallel CPU implementations and a fast and optimized 32-pipeline FPGA implementation. We obtained more than one order of magnitude in performance gain compared to standard CPUs, while our system also compared favourably to the FPGA implementations, showing that the GCA model fits well to current graphics cards using CUDA.

1 Introduction

The Cellular Automata (CA) model dates back to John von Neumann [VBO66] and Konrad Zuse [Zus69]. It models cells arranged in a regular grid, that only interact with particular local neighbor cells. For instance, in the 2D-grid with von Neumann neighborhood the neighbors are: North (N), South (S), East (E) and West (W). The new state C_{new} of a cell C is then computed using a local rule f:

$$C_{new} = f(C, N, S, E, W)$$

The rule can be applied in parallel to all n cells in the grid; if n gets large the model can be called massively parallel. After having computed all new states, the cell states are updated by setting C to C_{new}. A synchronous way of updating is used, that is, in each iteration all C_{new} are computed first. Afterwards all C are set to the new state.
The CA model can be efficiently used for various problems which can be described as a local interaction between cells. Typical applications are biological growth, self organization, artificial worlds, physical fields, movement and collision of particles, and so on. CA simulations have been shown to run well with GPUs, some experiments could show speedups in the range of two orders of magnitude compared to a similar CPU implementation [GBM, vSv].

Applications requiring global communication (access to remote cells) can only be simulated step by step through local communication, at a significant cost in time. For such applications, Hoffmann et al. proposed the Global Cellular Automata (GCA) model [HVW00, HVWH01, HVH01], which allows direct global read access from a cell to an arbitrary cell. The model remains massively parallel because, as in the CA, each cell modifies only its own state. The advantage of the GCA model lies in its generality; many parallel

problems can be mapped directly onto this model [HVW00, HVWH01].

We implemented two different GCA algorithms on the CUDA platform [NBGS08] for NVIDIA GPUs and evaluated the performance compared to a CPU-version. GPUs have a parallel throughput architecture which emphasizes executing many concurrent threads rather slowly, whereas CPUs are optimized for executing a single or a few thread very quickly. The GCA model fits these requirements, as it is by design possible to parallelize the computation of each cell. Thus, given a model with enough cells, the GCA model should fit well to the paradigms of current graphics cards. Only an unfavourable memory access pattern, depending on the individual GCA algorithm might be a bottleneck. As a proof of concept we evaluate two example algorithms, that have been proposed earlier for the GCA model [HHH04, JEH09]. We show that speedups in the range of one to two orders of magnitude are possible for the examined GCA-algorithms.

The remainder of the paper is organized as follows: In Section 2 the basic idea of the implementation of GCA on CUDA is described. In Section 3 we sketch the first algorithm *Bitonic Merge* and show our results. Section 4 describes the second algorithm *Diffusion* in a similar fashion. Section 5 concludes our work and gives a brief description of possible further work.

2 General Idea

To explain our basic idea, we first give a short introduction to the CUDA hardware design. A typical current CUDA-enabled GPU consists of several microprocessors that run in parallel, each managing a set of threads. Threads are organized into blocks, that are managed by the individual microprocessors. Blocks are then organized in a one or two dimensional grid. Threads between different blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores. Threads within a block can cooperate by accessing data through shared memory and by synchronizing their execution to coordinate memory accesses [NBGS08, Cor10].
Access to shared memory is fast (as fast as register accesses if no conflicts occur) compared to the access of global memory. Global memory on the other hand is larger and accessible for all threads in all blocks. There is no predefined order of execution of blocks and so the access to the global memory between two blocks is not synchronized or predictable.
The program that runs on the CUDA device is called kernel. CUDA program execution is invoked on the *host* (the CPU) with an initialization of the device and if necessary some data transfer to device's global memory. The *device* has no access to host memory. After finishing the execution of the kernel on the device, the *host* has to copy the final results back into the host memory.
To gain maximum performance and speedup, a GCA implementation on CUDA should be able to use all microprocessors and should use the maximum available memory to simulate large automata. We decided to use the global memory as the storage for the states s_i for every cell and the pointers $p_{i,j}$ (further algorithms in this paper just use one pointer p_i per

cell). Because of the unpredictable access to global memory we divided the memory in two parts, one for generation t of cells and the second for generation $t + 1$. Each thread executes at least one cell. This keeps us very close to the cell model and makes it easy to adapt new GCA-algorithms.
All cells in the GCA model change their state at the same time, so we need a global synchronization among all threads. But as described before, the CUDA architecture has no feature to synchronize between different blocks. We solve this problem by relaunching the kernel from the host after every generation. This costs a constant amount of time to reinitialize the kernel, but it is as of today the only way to simulate a reliable barrier synchronization among all threads on the graphics card. On the host side, we just have to swap the pointers of the cell states after every kernel call, so that the pointer for the new data s_{t+1},p_{t+1} becomes s_t,p_t for the next generation. Algorithm 1 demonstrates this procedure.

Algorithm 1: generic GCA on CUDA host side

Input: InitialStates, InitialPointers
1 $s \leftarrow$ InitialStates
2 $s_{new} \leftarrow$ allocateMem()
3 $p \leftarrow$ InitialPointers
4 $p_{new} \leftarrow$ allocateMem()
5 **for** $g = 1$ *to generations* **do**
6 runKernelOnDevice(s,p,s_{new},p_{new})
7 swap(s,s_{new})
8 swap(p,p_{new})
9 **end**
10 result $\leftarrow s$
Output: result

3 Bitonic Merge

Our first GCA algorithm is the second part of a *Bitonic Sort* algorithm, called *Bitonic Merge*. *Bitonic Sort* is a parallel sorting algorithm and it performs in $O(n \cdot log(n)^2)$ [CLRS09]. *Bitonic Merge* takes a Bitonic sequence as input and returns an ascending ordered list of this sequence. A *Bitonic* sequence is a sequence with $x_0 \leqq \ldots \leqq x_k \geqq \ldots \geqq x_{n-1}$ for some $k, 0 \leqq k < n$, or a circular shift of such a sequence [CLRS09]. In short, a *Bitonic Merger* sorts a bitonic sequence.

We took the GCA algorithm mentioned in [HVH01] and ported the cell rules from the GCA code to a CUDA kernel code. The algorithm works as shown in Figure 1. Each cell compares itself with the cell it points to and then inherits, if necessary, the value of the cell pointed to. The length of the pointer halves from generation to generation until it points on its own cell.

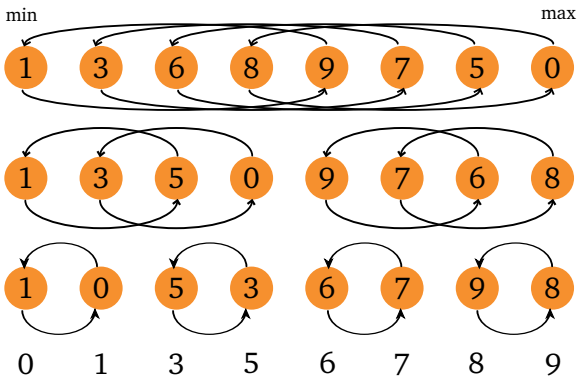

Figure 1: Bitonic Merge sorting an example bitonic sequence, consisting of 8 elements.

This GCA is based on a one dimensional line of cells, so we decided to put all pairs of state and pointer (s_i, p_i) sequentially into the global memory. We are able to support up to

$$\begin{aligned} size_{cell} &= sizeof(state) + sizeof(pointer) \\ n &= mem_{global}/(2 \cdot size_{cell}) \end{aligned}$$

n cells, which is also the maximum of the supported length of the sequence. Algorithm 2 offers a detailed CUDA kernel pseudocode based on the GCA implementation. If we wanted to support more cells than the allowed number of threads, the algorithm can be easily changed to support multiple cells by calculating more than one cell in each thread.

We compared our CUDA implementation with a similar CPU implementation and a multi-core OpenMP implementation. We measured the time needed to sort a sequence of 2^n cells of a 32-bit floating point type. For the CPU version we used the same cell rules (and C code) as in the CUDA implementation, the parallel execution of all cells is simply wrapped by a for loop. The results are shown in Figure 2. All timings are measured as the complete time it takes to sort the sequence, overhead for initializing the device, copying buffers and relaunching the CUDA kernel on the GPU is also included. For the experiments two consumer graphics cards were used, a NVIDIA GFX470 and a NVIDIA GFX260. An Intel Q9550 running at 3GHz was used to estimate single core performance and also an OpenMP version was created to fully use the Q9550's 4 cores.
Observing timings for small numbers of cells ($< 2^{19}$), there is no performance gap between the CPU and the GPU versions. This is because it takes a constant time (around 100ms) to initialize the hardware and copy over the buffers and also because the available parallelism on the graphics card may not be fully exploited. But with a simulation with 2^{26} cells, we observed a speedup of 232 times to a single core of the Intel Q9550 processor. The OpenMP implementation running on the same quad-core improved the CPU performance and resulted in a GPU speedup of 147 times compared to a quad-core.

Algorithm 2: CUDA Kernel pseudo-code for the Bitonic GCA-merger. The used & is a bit-wise AND, $\gg$ is a bit-wise right shift.

```
   Input: s, p, s_new, p_new
1  own_pos ← thread_id
2  other_pos ← 0
3  if (own_pos & p[own_pos]) = 0 then
4      other_pos ← own_pos + p[own_pos]
5      if s[other_pos] < s[own_pos] then
6          s_new[own_pos] ← s[other_pos]
7      end
8      else
9          s_new[own_pos] ← s[own_pos]
10     end
11 end
12 else
13     other_pos = own_pos - p[own_pos]
14     if s[other_pos] > s[own_pos] then
15         s_new[own_pos] ← s[other_pos]
16     end
17     else
18         s_new[own_pos] ← s[own_pos]
19     end
20 end
21 p_new[own_pos] ← p[own_pos] ≫ 1
```

To put the results into perspective, we further compared the runtime of optimized full sort algorithms for both architecture types with the GCA implementation. Here we used the STL std::sort implementation on the CPU and the sorting algorithm from the thrust library [1] on the GPU. Both algorithms sort a random array of float integers, whereas the GCA algorithm only performs a sort on a bitonic sorted list of float integers. Transfer times of the data to the GPU are included in the timings.

To account for this and to get a rough estimate, we assume that Bitonic Merge requires $log(n)$ times the time to sort a random sequence than sorting a bitonic sequence, since Bitonic Sort requires in total $O(log^2(n) \cdot n)$ single comparisons and Bitonic Merge does $O(log(n) \cdot n)$ comparisons. Under this rough estimate, the GCA algorithm is slower by at least a factor of 10 compared to an optimized code. Furthermore, the STL implementation on a single core is for large numbers of n at least 20 times faster than a the OpenMP implementation of the GCA algorithm.

To sum up these results, the first GCA algorithm suits perfectly onto the CUDA architecture and outperforms the CPU by two orders of magnitudes, as long as the number of cells is big enough to fully utilize the graphics card. The *slow* CPU performance is probably due to the large number of uncached memory accesses. But it is important to mention, that these

[1] http://code.google.com/p/thrust/

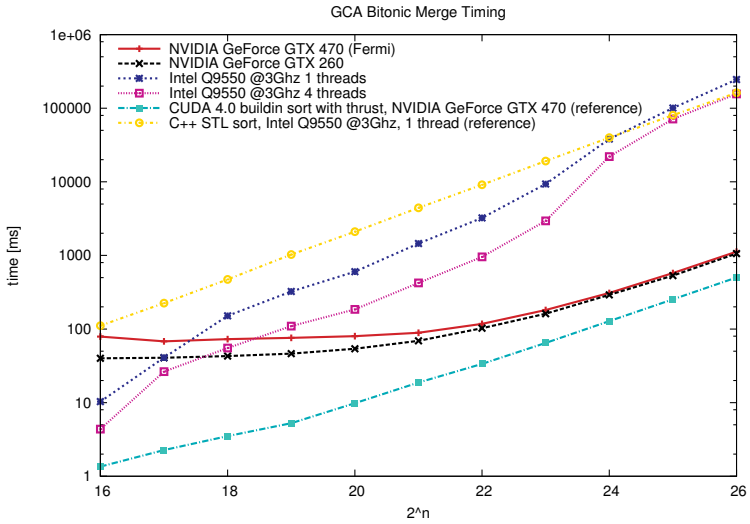

Figure 2: Timing diagram of the GCA Bitonic Merge algorithm. Comparing the runtime of the NVIDIA GFX470 and GFX260 graphics cards with an Intel Q9550, running at 3GHz with one and four threads. To put the results into perspective, timings of a full sorting algorithm on the CPU (STL std::sort) and on the GPU (CUDA Thrust library sort) on the same input sizes are included.

results do not reflect a comparison of the sorting algorithm *Bitonic Merge* between CPU and GPU. Our results only reflect the performance of *Bitonic Merge*-GCA algorithm on both hardware types. Moreover we observed that the optimized sort algorithms used for comparison are by at least one order of magnitude faster than the implemented GCA sort algorithm no matter which architecture is used.

4 Diffusion

The second implemented algorithm is a simulation of the diffusion of particles [JEH09]. The algorithmic model itself is just a proof-of-concept diffusion simulation, without aiming at chemical or physical correctness. The behavior of this model is shown in Figure 3. This GCA is based on a two dimensional grid of cells. Each cell has a one bit state, representing either an existing particle or an empty cell. The pointer (pseudo-)randomly changes from generation to generation between the 12 possibilities, described in the shown access pattern (Figure 3). If at any given generation two cells point to each other they will swap their state. Hoffmann et al. [JEH09] implemented a fast 32-pipelined FPGA version of this algorithm. This offers us the possibility to compare our results with a different architecture than the typical single-/multicore CPU versions. Algorithm 3 gives a CUDA-Kernel pseudocode for the implemented rules.

Algorithm 3: CUDA Kernel pseudo-code for the Diffusion GCA algorithm. The used % is meant as the arithmetic modulo operator.

Input: s, p, s_{new}, p_{new}, generation (the number of the generation), pattern, patternsize

```
1  xpos ← thread_x
2  ypos ← thread_y
3  p_pos ← compute local address according to x- and ypos
4  p_value ← p[p_pos]
5  p_neighbor_pos ← compute address according to xpos, ypos, p_value and the used
   pattern
6  // assuming that pointer and inverse are on opposite positions in pattern array
7  swap ← (((patternsize-1) - p[p_neighbor_pos]) = p_value)
8  // swap data if two cells point to each other
9  if swap then
10     s_new[p_pos] ← s[p_neighbor_pos]
11     // random reassignment of global pointer
12     p_new[p_pos] ← (p_value + (p_pos % patternsize) + (generation % patternsize) + 2 +
       (xpos % 2) + (ypos % 2)) % patternsize
13 end
14 else
15     s_new[p_pos] ← s[p_pos]
16     // random reassignment of global pointer
17     p_new[p_pos] ← (p_value + (p_pos % patternsize) + (generation % patternsize) + 1 +
       (xpos % 2) + (ypos % 2)) % patternsize;
18 end
```

Our results are shown in Figure 4. We observe the same behavior as in the first algorithm. With a lower amount of cells, the CPU version is faster than the GPU version. This is based on the initialization of the GPU, which requires constant setup time.
With a larger number of cells ($n = 55050240$), the GPU outperforms single and quadcore CPU versions. The achieved speedups of this algorithm are lower than the previous one, but the GPU is still approximately 40 times faster than the single threaded CPU version and 13 times faster than the CPU version with 4 threads. The same CPU - a Intel Q9550 running at 3GHz - and the Geforce GFX 470 was used to measure the timings. The mentioned 32-piplined FPGA implementation [JEH09] cannot deal with such a large amount of cells. To compare our results with the FPGA results, we take a look at the number of cells evaluated in one second. Table 1 shows the results. Our CUDA implementation has approximately 1.5 more throughput than a highly optimized FPGA implementation of the same algorithm in 2008. It is stated to be 50 times faster than a single CPU version on a Intel Pentium 4 (2.6GHz). In this case, the CUDA implementation compares favourably to the highly optimized FPGA version. However, this can only be achieved by using large amounts of cells on the CUDA graphics card.

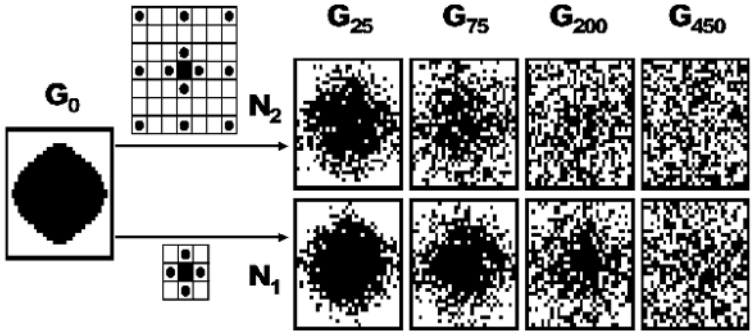

Figure 3: Visualization of the GCA Diffusion algorithm and its link pattern. The evolution of the starting population is shown after 25, 75, 200 and 450 generations G. The lower row illustrates a Cellular Automata, in contrast to the upper row illustrating a Global Cellular Automata. If at any given generation two cells point to each other they will swap their state. Furthermore the links are randomly changed according to the link pattern N_2 or N_1.

Table 1: GCA diffusion throughput comparison FPGA vs. CUDA

	FPGA	CUDA
cells per second	676.800.000 cells/s	917.504.000 cells/s

5 Conclusion and Future Work

We took two exemplary but different GCA algorithms and just ported the cell rules one by one from [HVH01, JEH09] without making special optimizations to CUDA kernel code. We wrapped these rules with our generic host side code and observed a speedup from 10 to 270 times. These two GCA algorithms suit perfectly the GPU hardware design without any very specific optimizations. Many similar GCA-algorithms can be ported without any adjustments of the generic model and therefore we think the GCA model fits well onto CUDA.

Lee et al. [LHS+10] noted, that current GPUs and CPUs are much closer in performance (2.5X), than previously reported differences of more than one order of magnitude, like our CUDA implementation of *GCA Bitonic Merge*. We agree, that highly optimized CPU-versions of GCA algorithms will probably decrease the performance difference between GPU and CPU. However, our approach opposes the high-end optimization of cache access and memory alignment for GCA algorithms, because our goal is a generic implementation for the CUDA device. The key intention behind the generic model is, that most GCA algorithms and kernels can be ported with ease onto the CUDA device, resulting in good performance without further optimizations.

Future work can be the evaluation of algorithms with a large rule set, for example a traffic simulation or rules which require time consuming arithmetic, for example Fourier transformations. Another open field is the implementation of the GCA-W (write) model

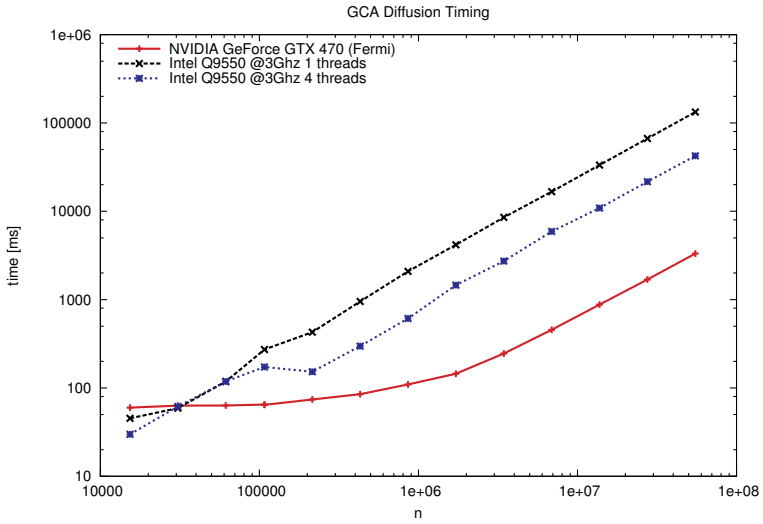

Figure 4: Timing diagram of the GCA Diffusion algorithm. Comparing the runtime of the NVIDIA GFX470 graphics cards with an Intel Q9550, running at 3GHz with one and four threads.

[Hof09] on CUDA to see whether the speedup increases or decreases by using write accesses. After exploring the GCA variants, a full automatic GCA-to-CUDA compiler could be build, as it exists for C and FPGA targets [JEH07].

Acknowledgments

The authors would like to thank Rolf Hoffmann for originally proposing to convert the CGA model to CUDA and for his advice and support during the course of the project.

References

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms, third edition*. MIT Press, 2009. 3

[Cor10] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*. 2010. 2

[GBM] Stéphane Gobron, Hervé Bonafos, and Daniel Mestre. GPU Accelerated Computation and Visualization of Hexagonal Cellular Automata. (c):512–521. 1

[HHH04] R. Hoffmann, W. Heenes, and M. Halbach. Implementation of the Massively Parallel Model GCA. In *International Conference on Parallel Computing in Electrical Engineering, 2004. PARELEC 2004*, pages 135–139, 2004. 1

[Hof09] R. Hoffmann. The GCA-w Massively Parallel Model. *Parallel Computing Technologies*, pages 194–206, 2009. 5

[HVH01] R. Hoffmann, K.P. Völkmann, and W. Heenes. Globaler Zellularautomat (GCA): Ein neues massivparalleles Berechnungsmodell. In *17th PARS Workshop*, 2001. 1, 3, 5

[HVW00] R. Hoffmann, K.P. Völkmann, and S. Waldschmidt. Global cellular automata GCA: An universal extension of the CA model. In *ACRI 2000 Conference*. Springer, 2000. 1

[HVWH01] R. Hoffmann, K.P. Völkmann, S. Waldschmidt, and W. Heenes. GCA: Global Cellular Automata. A Flexible Parallel Model. *Parallel Computing Technologies*, pages 66–73, 2001. 1

[JEH07] J. Jendrsczok, P. Ediger, and R. Hoffmann. The global cellular automata experimental language GCA-L, 2007. 5

[JEH09] J. Jendrsczok, P. Ediger, and R. Hoffmann. A scalable configurable architecture for the massively parallel GCA model. *International Journal of Parallel, Emergent and Distributed Systems*, 24(4):275–291, 2009. 1, 4, 18, 5

[LHS+10] Victor W. Lee, Per Hammarlund, Ronak Singhal, Pradeep Dubey, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, and Srinivas Chennupaty. *Debunking the 100X GPU vs. CPU myth*. ISCA '10. ACM Press, New York, New York, USA, 2010. 5

[NBGS08] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008. 1, 2

[VBO66] J. Von Neumann, A.W. Burks, and Others. Theory of self-reproducing automata. 1966. 1

[vSv] Luděk Žaloudek, Lukáš Sekanina, and Václav Šimek. Accelerating Cellular Automata Evolution on Graphics Processing Units. *Development*. 1

[Zus69] K. Zuse. *Rechnender Raum*. Vieweg Braunschweig, Germany, 1969. 1