2009 First International Conference on Advances in System Simulation

Experiments With Single Core, Multi Core, and GPU-based Computation of Cellular Automata

Stefan Rybacki, Jan Himmelspach
and Adelinde M. Uhrmacher
Institute of Computer Science
University of Rostock
18051 Rostock, Germany
Email: {stefan.rybacki, jan.himmelspach, lin}@uni-rostock.de

***Abstract*—Cellular automata are a well-known modeling formalism exploited in a wide range of application areas. In many of those, the complexity of models hampers a thorough analysis of the system under study. Therefore, efficient simulation algorithms are required. We present here a comparison of seven different simulation algorithms for cellular automata: the classical "full" simulator, the classical "discrete event" simulator, a threaded (multi core) variant of each of these, an adaptable threaded variant, and a GPU based algorithm with and without readback of calculated states. The comparison is done based on the M&S framework JAMES II by using a set of well-known models.**

Keywords - Cellular automata, GPU, Multi Core, Simulation algorithm

I. INTRODUCTION

Cellular automata (CA) are used as modeling formalism in many application areas where spatial dynamics play a role, e.g., cell biology [1]. As these models tend to be rather complex referring to number of entities and the range they cover at a temporal and spatial scale, the computation of simulations can be a time consuming task.

To address the problem of efficiency, a variety of simulation algorithms have been developed. However, it is common knowledge that a silver bullet for efficient simulation does not exist. So the question is to identify the most promising algorithm in a given situation, or more general: which algorithm to use in which situation? To find answers to this type of question is the goal of "experimental algorithmics" [2] where algorithm properties are identified in an experimental manner. Using a constant framework is essential to evaluate the algorithm under study in an unbiased manner. The M&S framework JAMES II provides such a framework for any formalism and simulation algorithm [3]. It will form our basis to explore the space of single, multi-core, and GPU based simulation algorithms for cellular automata.

As GPU based simulations are gaining steadily ground the paper starts with an introduction to GPU-based simulation in Section II, before Section III describes the algorithms used in the experiments. Section IV is divided into Section IV-A covering the benchmark models applied, Section IV-B which explains the setup utilized for the experiments done as well as Section IV-C which present the performance results. Afterward Section V concludes and discusses the results retrieved and lessons learned from the experiments.

II. GPU-BASED SIMULATION

GPUs are receiving increasingly attention in simulation, e.g., [4], [5], [6], [7]. As GPUs are trimmed toward a very efficient execution of a set of specific problems, the usage of GPUs instead of CPUs can result in a major speedup if the problem fits the computing architecture of a GPU [5], [8], [9]. In addition, GPUs might also be used in combination with CPUs, e.g., for those parts of the model which are significantly accelerated only, or as any other computation resource in fine-grained and coarse-grained parallel and distributed simulation.

As cellular automata can easily be mapped to a texture based computation schema, a GPU-based realization constitutes a promising alternative to efficiently simulate cellular automata, as has been shown in [10], [11], [12], [13], [14], [15].

There are the following options to incorportate the GPU for simulation:

1) Shader based computation using GLSL[16], HLSL[17], [18] or CG[19]
2) NVIDIA Compute Unified Device Architecture (CUDA) [4], [20]
3) AMD/ATI Stream [21]
4) OpenCL [22]

Besides the shader based approach and the fairly new OpenCL initiative there is currently no general way to leverage the GPU on a wide range of different graphics hardware of different vendors. There are even limitations from vendor to vendor when using shader based solutions for general purpose computing. An advantage of this approach is the multi platform availability when using GLSL based implementation since it is available on all platforms where OpenGL[23] 2.0 and higher is supported. Additionally there are API bindings for a variety of programming languages. For Java this could be for instance JOGL [24] or LWJGL [25].

On the other hand CUDA is a very mature architecture and API which simplifies the process of developing general purpose applications for graphics hardware but is only available on CUDA enabled NVIDIA graphics cards. API bindings for other programming languages like Java are also still missing or incomplete [26], [27].

AMD/ATIs stream technology is younger than the NVIDIA counter part and also only available on special graphics

978-0-7695-3773-3/09 $26.00 © 2009 IEEE
DOI 10.1109/SIMUL.2009.36

hardware by ATI/AMD and also has currently no bindings for other programming languages than C or C++.

Hopefully OpenCL will eventually fill the glitch between the different APIs existing today once it is widely available on different hardware platforms and supported by a large range of vendors.

III. The algorithms

The following set of algorithms for simulating cellular automata shall be compared.

The first algorithm is the brute force CA algorithm, which computes the new state of each single cell at each step. Thus, $width \times height$ cells per step will be computed, and the complete grid has to be copied first. This algorithm can be used for all types of CA, including CAs where cells change randomly and spontaneously.

```
oldGrid = grid.clone();
for all grid.cells do
  grid.cell = oldGrid.computeNewState (cell)
```

The second algorithm is a discrete event CA algorithm. Assuming that a change can only occur if a cell's own state or the states of its neighbors have changed in the last step, the algorithm maintains a list of candidates for updates. In the worst case, this list will contain all cells of the grid, but usually it will contain only a small subset. Thus, the number of cells being updated will be reduced significantly compared to the first algorithm. The drawback is the additional effort required for maintaining this list, which might lead in certain cases to an even worse performance compared to algorithm 1. Also, the algorithm cannot be used for models where cells might change randomly and spontaneously.

```
for each candidate in candidates do
  grid.cell = grid.computeNewState (candidate)
  if (grid.cell.state != candidate.state)
     add candidate and neighbours to
         newCandidates
candiates = newCandidates
```

The third algorithm is a multi threaded variant of the first algorithm. We create as many threads as physical resources ($CPUs \times cores$) exist, and then compute in parallel, but brute force the complete grid. Again a copy of the complete grid is necessary. If the grid is sufficiently large, the overhead of thread creation should become negligible compared to the speed-up achieved by the parallel computation of the new states.

The fourth algorithm is a multi threaded variant of the second algorithm. We create as many threads as physical resources ($CPUs \times cores$) exist, and then compute in parallel, the cells which might change their state. Thus, in comparison to the third algorithm the computational load per thread should be lower. However, the candidate sets computed by the different threads into one set have to be merged and partitioned in the next step again which requires additional effort.

The fifth algorithm is a multi threaded variant of the second algorithm, and thus comparable to the previous one. But this algorithm can be parametrized: we can increase / decrease the number of threads to be used (thus we can use more or fewer threads than we have physical resources), and we can set a minimal number of cells per thread, and thus we automatically decrease the number of threads if only very few cells have to be updated. For our first set of experiments we set the number of threads to the number of available CPUs (u) times the number of cores (c) ($t = u \times c$) - and thus the same number of cores will be basically used as in the 4th algorithm, but we are using a threshold value of 10 as minimal number of cells to be computed per thread - and thus if the number of cells (n) divided by 10 ($\frac{n}{10}$) is less than t fewer threads will be used. This setting should be slightly better if less work is to be done - however, it might turn out the overhead due to the new calculations might even increase the overall computation time. Also having too many threads compared to computational resources might induce a decrease in performance.

The sixth algorithm makes use of the GPU to calculate the next state of each cell of a cellular automaton. It basically works like the first algorithm where for each cell a new state is calculated according to its neighbors and current state. The cells of the cellular automaton are represented as texture that is rendered into a target texture using specified rules which is then again used as source for the calculation of the next texture. In pseudo code this looks like this:

```
for each pixel in source texture do
  neighborhood = getNeighborhood(pixel)
  target pixel = nextState(pixel, neighborhood)
```

Due to the highly parallel architecture of today's GPUs with up to more than 100 processing units this algorithm can outperform other parallel or multi threaded algorithms.

The seventh algorithm is equal to the algorithm before but additionally to the usage of the GPU for state transitions it also reads back the calculated states from the graphics hardware and therefore makes them available for further operations like visualization or persistent storage. This algorithm is included because it decreases the speed of GPU computation due to the GPU to CPU data transfer.

IV. Experimental analysis

The usage of a stable environment, a set of different hardware infrastructures, the usage of different "good" problem instances, and to use sufficiently coded (esp. in regards to efficiency) algorithms are among the pre-conditions of a credible experimental algorithmics [2]. But what are good problem instances or benchmark models? This might heavily depend on what you want to show - and thus, it is of equal importance that all models used for one algorithm are used for another one as well. Below we shortly describe a set of selected models, this list is not exhaustive - neither in regards to CA models in general nor in regards to "good benchmark models".

	Machine 1	Machine 2	Machine 3	Machine 4
Type	Laptop	Workstation	Mobile Workstation	Laptop
CPU	Core Duo T7500	Intel Xeon E5420	Core 2 Extreme Q9300	Core Duo T9500
CPUs × Cores	2 × 1	2 × 4	2 × 2	2 × 1
CPU Speed	2.2 GHz	2.5 GHz	2.5 GHz	2.5 GHz
RAM	2 GB	8 GB	8 GB	3 GB
GPU	Intel GM965	NVIDIA Quadro NVS 290	NVIDIA QuadroFX 3700M	NVIDIA GeForce 9500M GS
GPU RAM	384 MB shared	256 MB dedicated	1 GB dedicated	1.7 GB (512 MB dedicated)
GPU Generation	OpenGL 2.0 SM4	OpenGL 2.0 SM4	OpenGL 2.1 SM4	OpenGL 2.1 SM4
Operating System	WinXP Pro SP3 32Bit	WinXP Pro SP2 64Bit	WinXP Pro SP2 64Bit	Vista Home Premium 32Bit

TABLE I
MACHINE CONFIGURATIONS

	Machine 1	Machine 2	Machine 3	Machine 4
SciMark	421.2	488.0	536.3	491.9
FFT	322.8	379.6	419.6	381.0
Jacobi	649.0	751.8	816.8	758.6
Monte Carlo	82.2	104.7	116.8	106.4
Sparse	423.6	487.6	541.1	491.4
LU	628.6	716.3	787.1	722.3

TABLE II
SCIMARK FOR EACH USED MACHINE

	Machine 1	Machine 2	Machine 3	Machine 4
3DMark Score	348 3DMarks	1555 3DMarks	11744 3DMarks	3501 3DMarks
SM 2.0 Score	113	590	4865	1329
SM 3.0 Score	128	484	4705	1311
CPU Score	1313	5884	4257	2135

TABLE III
GPU BENCHMARK FOR EACH USED MACHINE

A simple time measuring based on the computer's clock was used for all experiments. To reduce the variance of run times, we repeated each configuration 11 times. All experiments have been executed with each of the algorithms introduced above.

We executed the experiments on the machines given in Table I. We used the Java SciMark 2.0a[1] benchmark to get a number of comparable performance measurements for the CPUs on the machines (see Table II). We used the Futurmark 3DMark06[2] benchmark to get comparable values for the different GPUs used (see Table III). These values can be used to "normalize" our performance results, and thus to compare this with future results and / or results got on different machines.

Sure the performance comparison between the GPU based algorithm and the rest is "unfair" because it heavily depends on the system setup which algorithm performs better. Here it is "unfair" because the computing power of the machine is currently still relatively "high end" for desktop computers whereby the graphics card is not among the fastest ones. Consequently the results of this comparison can only be taken as a hint what GPU based computation can mean here, and they can be used as a first base to relate further GPU based results to the remaining results achieved in these experiments.

The Java code has been compiled using the JDK 1.6 update 13 (32bit) compiler, we used the Java 1.6 VM update 13 (32bit), Version using "-Xmx1024M" as additional execution parameter as runtime environment. JAMES II has been used in version 0.81. The CA plug-in had the version number 1.02.

[1]http://math.nist.gov/scimark2/download_java.html, 16.06.2009
[2]http://www.futuremark.com/products/3dmark06, 16.06.2009

A. Benchmark models

We restrict ourselves here to models which do not make use of random numbers. This has two major reasons: it is pretty unlikely to find characteristic behavior of the algorithms under test by using random values here - in principal we should be able to observe the behavior with our benchmark models as well. The second reason is that not all simulation algorithms can handle random numbers: the discrete event simulation algorithms can only handle cells which have a probability to change after itself or one of its neighbors has been changed in the step before, the current implementation of the GPU based algorithms do not support random transitions at all.

a) Game of life: We used the game of life model as described in [28]. This model uses the "Moore" neighborhood. A cell stays alive if there are two or three living cells in its neighborhood, otherwise it dies. If a dead cell has exactly two living neighbors it gets alive.

```
1 @caversion 1; dimensions 2;
2 neighborhood moore;
3 state DEAD, ALIVE;
4 rule{ALIVE}:!ALIVE{2,3}->DEAD;
5 rule{DEAD}:ALIVE{3}->ALIVE;
```

b) Parity: We used the parity model as described in [28]. This model uses the "von Neumann" neighborhood. A cell gets alive if the number of living cells in its neighborhood is odd, otherwise it dies.

```
1 @caversion 1; dimensions 2;
2 state DEAD, ALIVE;
3 rule{ALIVE} : ALIVE{1} | ALIVE{3} -> DEAD;
4 rule{DEAD}  : ALIVE{1} | ALIVE{3} -> ALIVE;
```

c) Majority: We used the simple majority model as described in [28]. This model uses the "Moore" neighborhood. A new cell gets the state the majority of its neighbors has. It remains in its state if no majority exits.

```
1 @caversion 1; dimensions 2;
2 neighborhood moore;
3 state STATE1, STATE2;
4 rule : STATE1{5,} -> STATE1;
5 rule : STATE2{5,} -> STATE2;
```

d) Wireworld: The wireworld idea can be used to model the movement of electrons, and thus it can be used to model computers. The rules are pretty simple (see code), and thus the behavior of the model is tightly coupled to the initial grid state - here wires which shall be used to transport electrons

have to be modeled. For our benchmark runs we are using an initial setting which describes a prime number generator[3].

```
1  @caversion 1; dimensions 2;
2  neighborhood moore;
3  state : BLANK, COPPER, HEAD, TAIL;
4  rule{BLANK}  : -> BLANK;
5  rule{COPPER} : HEAD{1,2} -> HEAD;
6  rule{HEAD}   : -> TAIL ;
7  rule{TAIL}   : -> COPPER;
```

e) Benchmark model: Benchmark models can try to cover a broad range of model behaviors, but they always should be created in a manner that their usage can reveal properties of the simulation algorithms evaluated by using these. Here we describe a simple benchmark model which allows to determine the run time behavior of simulation algorithms if a constant number of cells is about to change for a given grid. The changing cells are always switching between two states, the remainder remain stable. By using different grid sizes and different percentages of toggling cells we can find out whether there is a tradeoff point for the simulation algorithms. For our simulation algorithms it makes a difference which cells are toggling, thus we cannot always use the first n percent of the grid cells. If the discrete event based algorithms work on a dense block of changing cells the list of cells to be updated contains many duplicates (which are automatically ignored), if they are equally distributed over the grid the list of cells to be updated can be equal to the overall list of cells. But this has no impact on the brute force algorithms. However, these tests cannot reveal the real performance on real models, e.g., the Game of life model described above the number of cells decreases over the time and finally ends up with a constant (and most often small) number of cells to be changed.

```
1  @caversion 1; dimensions 2;
2  state BLANK, ON, OFF;
3  rule{BLANK}:->BLANK;
4  rule{ON}:->OFF;
5  rule{OFF}:->ON;
```

B. Experimental setup

We executed 119 different combinations of models, initial states, and simulation algorithms altogether on four different machines. For all models we used grids of differing sizes (see graphs for more details). Model (e) is initialized using blocks of initialized cells, so that large blank blocks exist as well. As percentage of toggling cells we used 25, 50 and 75 percent. All experiments have been executed on the machines described above.

C. Performance Results

Here we present the results retrieved using the setups described above. Each Figure contains the results of one benchmark model. The y-axis shows the throughput of simulation steps per second, the x-axis groups the simulation algorithms per model configuration used, the z-axis shows the

[3]http://www.quinapalus.com/wi-index.html, 16.06.2009

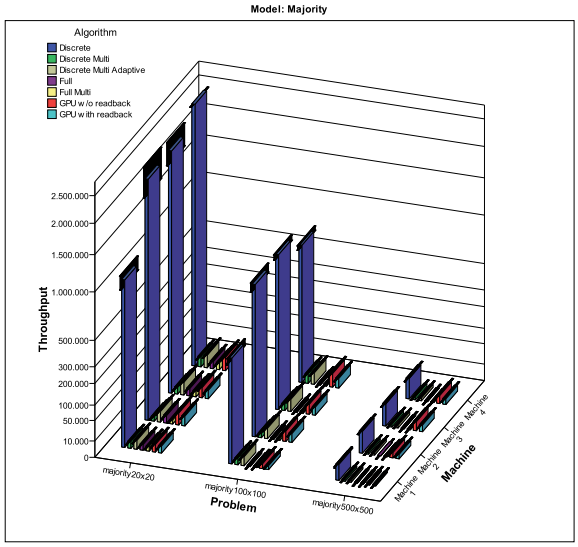

Fig. 1. Results by using the "majority" model. Per model setup (groups on x-axis) we have used all simulation algorithms (bars per row). All experiments have been executed on all machines (z-axis).

x/y axis combinations per machine. The black bars on top of the colored bars represent the variance which is pretty small in regards to the throughput and which can thus be ignored in the discussions. Additionally to the figures presented Table IV shows selected results of Machine 3.

In Figure 1 the results for the "majority" model are shown, and in Figure 2 the results for the "game of life" model are shown. Both models reach a "stable" state after a few simulation steps - from then on the number of cells to be updated is in the interval smaller or equal a fixed n. Both models have been initialized with well defined, but random initial settings. The very high throughput of the simple discrete event simulation algorithm is due to the fact that it has to do almost nothing after a couple of simulation steps - and that its overhead is pretty low in comparison the other discrete event simulation algorithms. The full and GPU based algorithms do not "recognize" the low number of cells to be updated - and thus their throughput remains stable. In the game of life model the discrete event simulator has a very low throughput from model sizes 500 x 500 on - this is related to the number of cells to be updated.

In Figure 3 the results for the "wireworld" model are shown and in Figure 4 the results for the "parity" model are shown. Both models imply a higher rate of cells to be updated during the run time. Due to the number of cells to be updated the CPU based algorithms have a relatively small throughput (besides the small parity model). The threaded CPU based algorithms are outperformed by those, that are not threaded.

In Figure 5, the results for the "benchmark" model are shown. The chart reveals that the discrete event based algorithm is only better, if less than 50 percent of the cells on a pretty small grid have to be updated. Otherwise it is outperformed by the the full simulator. In the end the GPU based algorithms take over again.

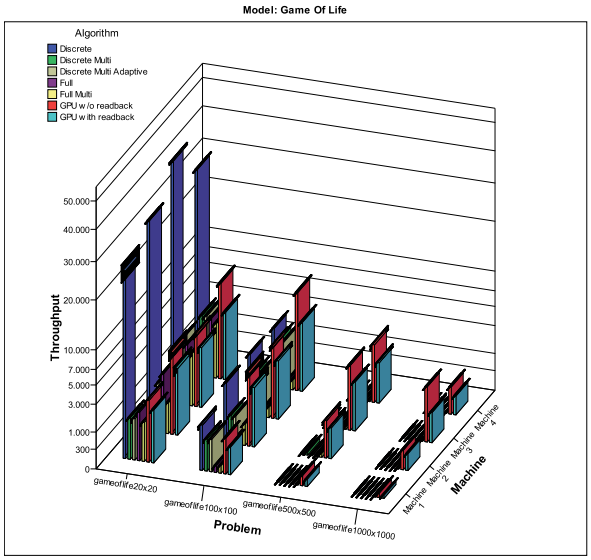

Fig. 2. Results by using the "game of life" model. Per model setup (groups on x-axis) we have used all simulation algorithms (bars per row). All experiments have been executed on all machines (z-axis).

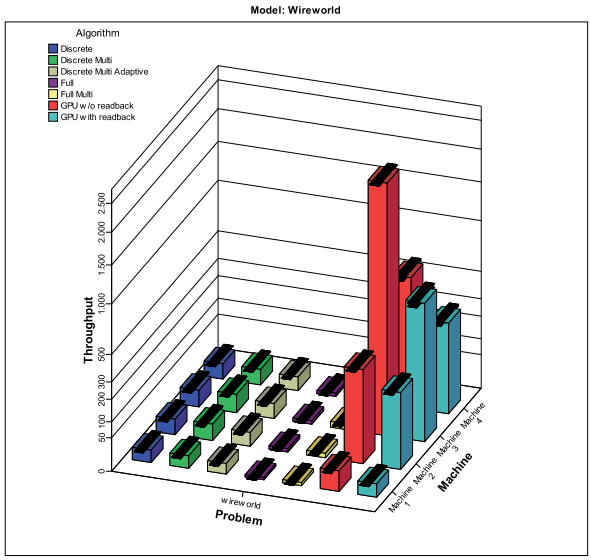

Fig. 3. Results by using the "wireworld" model. We have used all simulation algorithms to compute the model (bars per row). All experiments have been executed on all machines (z-axis).

Model	Problem	Throughput of algorithms in *simulation steps per second*						
		Discrete	Discrete Multi	Discrete Multi Adaptive	Full	Full Multi	GPU	GPU readback
Benchmark	100×100 25%	49.43	41.96	32.84	111.99	127.21	3959.00	2556.29
Benchmark	100×100 75%	46.74	45.48	38.30	110.77	136.05	3817.42	2498.55
Benchmark	20×20 25%	7915.60	2037.62	2135.78	4866.60	2093.47	4134.99	2653.13
Benchmark	20×20 75%	2590.60	1090.25	1106.97	4164.84	1491.92	4044.74	2693.06
Game of Life	1000×1000	0.99	1.79	0.90	0.77	2.00	2192.64	675.99
Game of Life	100×100	2615.85	1232.68	1236.03	65.67	90.30	3714.69	2428.42
Parity	100×100	72.36	82.39	75.42	81.33	121.98	3860.18	2412.50
Parity	500×500	23.72	31.28	19.55	3.56	7.11	3418.49	1749.58
Wireworld	631×958	15.12	17.76	12.72	1.66	3.19	2229.71	689.85

TABLE IV
SELECTED RESULTS OF SEVERAL PROBLEMS FOR MACHINE 3

f) Comparison: Comparing the results for the different models and simulation algorithms reveals that the GPU based algorithms often outperform the CPU based ones on all the machines and cards provided for large and computationally intensive models. But for a number of problems / problem sizes the CPU based algorithm perform better as the GPU based ones - thus we cannot recommend to rely completely on the GPU based algorithm only. The current implementations of the multi threaded algorithms do not increase the throughput significantly in most cases – this only happens sometimes on the 8 core machine (Machine 2), if the computational load is very high. The adaptive discrete event simulation algorithm has only being used with a single parametrization (see above). The threshold used is pretty small and a larger threshold could result in a further speed up.

However, we only selected problems computable by all algorithms - thus there is a need for the badly performing "full" CA simulation as well: these are the only algorithms which currently can handle "spontaneous" state changes for now.

V. CONCLUSION

The comparison of the seven different algorithms for cellular automata with five different models with different starting configurations each simulated on four different machines revealed once again the fact that there is no perfect algorithm for everything. In addition not every idea of an improvement may turn out to be a good idea in the end. GPU based algorithms are an alternative, but whether they are useful or not heavily depends on the model to be simulated (not all rule types are yet supported in our implementation). However, they can at least be used in addition to other algorithms, e.g., to compute on the CPU and the GPU. Further work should be done on tuning the multi threaded algorithms, and on using different executable model implementations, as the later might also have an effect on the performance (please note in JAMES II symbolic and executable models are distinguished). On the GPU site, new algorithms, e.g., based on CUDA, and their performance will be of interest. JAMES II, including the CA methods described, is available at www.jamesii.org.

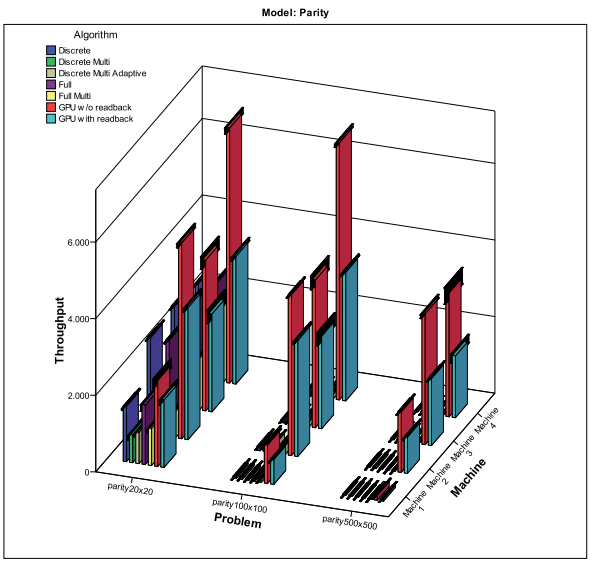

Fig. 4. Results by using the "parity" model. Per model setup (groups on x-axis) we have used all simulation algorithms (bars per row). All experiments have been executed on all machines (z-axis).

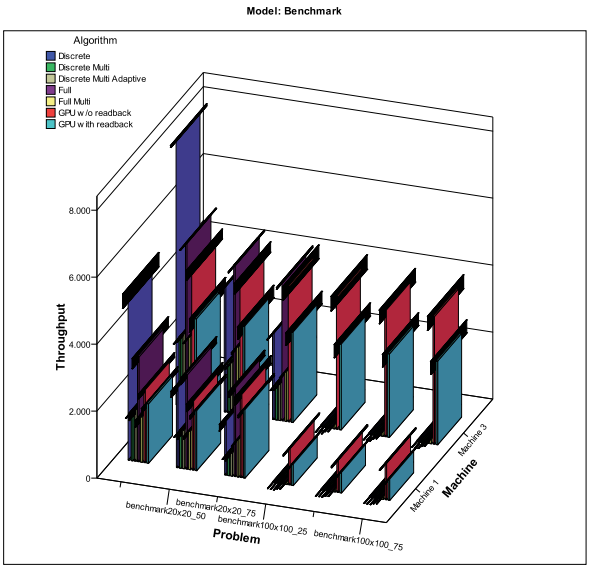

Fig. 5. Results by using the "benchmark" model. Per model setup (groups on x-axis) we have used all simulation algorithms (bars per row). All experiments have been executed on Machine 1 and 3 (z-axis).

VI. ACKNOWLEDGMENTS

This work has been funded by the German research foundation (DFG). In addition we have to thank our colleague Roland Ewald for his performance testing environment and his support in executing the experiments.

REFERENCES

[1] K. Takahashi, S. Arjunan, and M. Tomita, "Space in systems biology of signaling pathways–towards intracellular molecular crowding in silico," *FEBS Lett.*, vol. 579, pp. 1783–1788, Mar 2005.

[2] D. Johnson, "A theoretician's guide to the experimental analysis of algorithms," in *Fifth and Sixth DIMACS Implentation Challenges*, 2002.

[3] J. Himmelspach and A. M. Uhrmacher, "Plug'n simulate," in *ANSS '07: Proceedings of the 40th Annual Simulation Symposium*. Washington, DC, USA: IEEE Computer Society, Mar. 2007, pp. 137–143.

[4] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[5] L. Nyland, M. Harris, and J. Prins, "Fast n-body simulation with cuda," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley Professional, August 2007, ch. 31.

[6] K. S. Perumalla, "Efficient execution on gpus of field-based vehicular mobility models," in *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation (PADS) 2008*. Los Alamitos, CA, USA: IEEE Computer Society, 2008.

[7] D. Strippgen and K. Nagel, "Using common graphics hardware for multi-agent traffic simulation with cuda," in *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, pp. 1–8.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1370–1380, 2008.

[9] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra, "Physically-based visual simulation on graphics hardware," in *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002, pp. 109–118.

[10] S. Druon, A. Crosnier, and L. Brigandat, "Efficient cellular automata for 2d/3d free-form modeling," in *In WSCG*, 2003, p. 102108.

[11] S. Gobron, F. Devillard, and B. Heit, "Retina simulation using cellular automata and gpu programming," *Mach. Vision Appl.*, vol. 18, no. 6, pp. 331–342, 2007.

[12] S. Gobron and D. Mestre, "Information visualization of multi-dimensional cellular automata using gpu programming," in *IV '07: Proceedings of the 11th International Conference Information Visualization*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 33–39.

[13] C. Kauffmann and N. Piche, "Cellular automaton for ultra-fast watershed transform on gpu," 2008, pp. 1–4.

[14] J. Tran and D. Jordan, "New challenges for cellular automata simulation on the gpu," 2008.

[15] J. Singler, "Implementation of cellular automata," Poster of GP2-Workshop 2004 in Los Angeles.

[16] R. J. Rost, *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, January 2006.

[17] M. Corporation, *Microsoft DirectX 9 Programmable Graphics Pipeline*. Redmond, WA, USA: Microsoft Press, 2003.

[18] Microsoft, "Writing hlsl shaders in direct3d 9," http://msdn.microsoft.com/en-us/library/bb944006(VS.85).aspx, 16.01.2009.

[19] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: a system for programming graphics hardware in a c-like language," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*. New York, NY, USA: ACM, 2003, pp. 896–907.

[20] NVIDIA, "What is cuda," http://www.nvidia.com/object/cuda_what_is.html, 16.01.2009.

[21] AMD/ATI, "Gpu technology for accelerated computing," http://www.amd.com/stream, 16.01.2009.

[22] K. Group, "Opencl - the open standard for parallel programming of heterogeneous systems," http://www.khronos.org/opencl/, 16.01.2009.

[23] ——, "The industry's foundation for high performance graphics," http://www.opengl.org/, 16.01.2009.

[24] "Jogl - java bindings for opengl," https://jogl.dev.java.net/, 16.01.2009.

[25] "The lightweight java game library (lwjgl)," http://lwjgl.org/, 16.01.2009.

[26] "Jcufft and jcublas," http://javagl.de/index.html, 16.01.2009.

[27] H. A., "Jacuzzi," http://jacuzzi.sourceforge.net/, 16.01.2009.

[28] N. Gilbert and K. Troitzsch, *Simulation for the Social Scientist*, 2nd ed. Open University Press, 2008.