Computer Science • 13 (1) 2012 http://dx.doi.org/10.7494/csci.2012.13.1.35

Marcin Worecki
Rafał Wcisło

GPU ENHANCED SIMULATION OF ANGIOGENESIS

Abstract *In the paper we present the use of graphic processor units to accelerate the most time-consuming stages of a simulation of angiogenesis and tumor growth. By the use of advanced CUDA mechanisms such as shared memory, textures and atomic operations, we managed to speed up the CUDA kernels by a factor of 57x. However, in our simulation we used the GPU as a co-processor and data from CPU was copied back and forth in each phase. It decreased the speedup of rewritten stages by 40%. We showed that the performance of the entire simulation can be improved by a factor of 10 up to 20.*

Keywords gpu, angiogenesis, tumor

1. Introduction

Angiogenesis is a biological process of new blood vessel growth. It is an important natural process accompanying wound healing and tissue reproduction. It is the fundamental process in many diseases including cancer, skin diseases, age-related blindness, diabetic ulcers, and cardiovascular diseases. A good understanding of angiogenesis can help develop new medical treatments [4]. There are many models and simulations of angiogenesis. For example, M. Aubert *et al.* [2] have presented angiogenesis modeling based on a set of nonlinear partial differential equations (PDEs). Abbas Shirinifard *et al.* [7] have used multi-cell Glazier-Graner-Hogeweg model (GGH, also known as the Cellular Potts Model) to simulate 3D solid tumor growth and angiogenesis. In our simulations, we used complex automata approach (CxA), which combines the cellular automata modeling (CA) with off-grid particle dynamics coupled by continuum reaction-diffusion equations. However, to simulate spatio-temporal scales in order to enable the observation of tumor development in all growth phases (avascular, angiogenic and metastasis), more computational power is required.

The computational power of GPUs has been increased dramatically and currently it exceeds CPUs by orders of magnitude. The development of GPU computational environments, like CUDA, enables this power to be harnessed in scientific applications.

Particle simulations have been successfully implemented on GPUs. In [5], the authors presented a GPU implementation of all-pairs n-body simulation of gravitational forces. Their simulation runs more than 50 times faster than a tuned serial implementation and 250 times faster than a portable C implementation. J. A. van Meel *et al.* [8] and Joshua A. Anderson *et al.* [1] have implemented molecular dynamics simulations on GPU and obtained a speedup greater than 40. These works motivated us for speeding up our CxA modeling.

The rest of the paper presents algorithms used in our implementation and a comparison of different approaches.

2. CPU implementation

We started with a working and optimized CPU implementation. The simulation combines particle dynamics with cellular automata. There are two kinds of objects:

Spherical cells — represented as coordinates and radius.

Tubes — represented as coordinates of both ends and radii in both ends. Chains of connected tubes form blood vessels.

Both have plenty of properties i.e. standard particle attributes such as position, velocity and force as well as parameters related to the tissue they are made of. There are nearly twenty parameters defining tissue including density, growth speed, and oxygen consumption.

Cell life-cycle is represented by the following states:

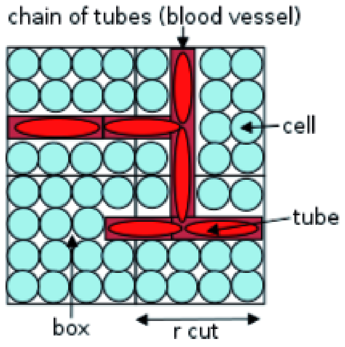
New — initial state.

Mature — a `New` cell becomes `Mature` when it reaches a certain age. A cell in this state can divide itself into two daughter cells (in a `New` state).

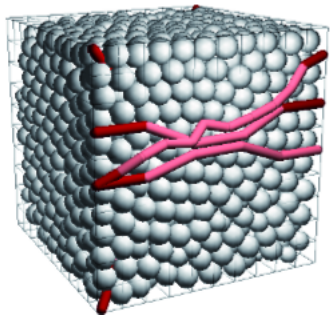
Hypoxia — `New` and `Mature` cell move to this state when oxygen shortage occurs. Cells in `Hypoxia` produce TAF (Tumor Angiogenesis Factor) which stimulates the growth of new blood vessels.

Dead — Cells die because of aging or oxygen shortage.

The exchange of oxygen and TAF is modeled by reaction-diffusion equations.



(a) 2D



(b) 3D

Figure 1. Types of objects used in our simulation

Computation of short-range interactions like forces and concentration exchange employs cell list algorithm. Simulation space is divided into cells with an edge length greater than a cut-off radius. In order not to mislead the part of the tissue and the part of the simulation space, in the rest of the document we will refer to the latter as "box". Figure 1a shows types and relationships of objects used in the simulation, detailed description of the simulation method is presented in [9].

2.1. Simulation phases

Our simulation consists of the following phases:

- Object interactions (computation of pressure, concentrations and forces):
  - Cell-cell
  - Tube-cell
  - Tube-tube
- Cells growth and life cycle
- Tubes growth
- Blood flow
- Rearranging cells
- Cell-barrier forces

Profiling of the existing CPU implementation has shown that 95% of the total execution time is spent in only two simulation phases: cell-cell and tube-cell interactions. Those two stages have been rewritten to utilize the potential of GPUs. The following sections provide details on their CPU versions.

2.1.1. Cell-cell interactions

In this stage, we calculate interactions between cells including: force, pressure and diffusion of oxygen and TAF.

Algorithm 1: Cell-cell interactions CPU version

```
 1  foreach box1 in boxes
 2      foreach cells[i] and cells[j], i < j in box1
 3          if cells[i] and cells[j] interact
 4              force = calc_force;
 5              cells[i].force += force
 6              cells[j].force -= force
 7              calculate concentrations for both cells
 8              calculate pressure for both cells
 9          end
10      end
11      foreach box2 in 13 adjacent boxes of box1
12          foreach cells[i] in box1
13              foreach cells[j] in box2
14                  if cells[i] and cells[j] interact
15                      force = calc_force;
16                      cells[i].force += force
17                      cells[j].force -= force
18                      calculate concentrations for both cells
19                      calculate pressure for both cells
20                  end
21              end
22          end
23      end
24  end
```

On line #1 we iterate over all boxes. For each box, we calculate interactions between all particles within the given box. On line #3 we check whether cells interact i.e. if a distance between them is less than a given cut-off radius. Then on lines #11-23 we take care of neighboring boxes. In 3D there are 26 adjacent boxes. As our CPU implementation employs Newton's third law, we need to handle only 13 of them.

For the sake of conciseness, implementations of force, concentrations and pressure calculations were omitted.

2.1.2. Cell-tube interactions

In this phase, we deal with two kinds of objects: cells and tubes. The CPU version of this routine calculates cell-tube and tube-cell interactions in a single run.

Algorithm 2: Cell-tube interactions CPU version

```
 1  foreach tubeChain in tubeChains
 2    tube = first tube in tubeChain
 3    while tube
 4      foreach adjacent box of tube
 5        foreach cell in box
 6          if cell and tube interact
 7            force = calc_force;
 8            cell.force += force
 9            tube.force -= force
10            calculate concentrations cell-tube and tube-cell
11            calculate pressure cell-tube and tube-cell
12          end
13        end
14      end
15      tube = tube.next
16    end
17  end
```

tubeChain on line #1 is a linked-list of tubes. On line #4 we iterate through all boxes adjacent to the tube. As we mentioned earlier, tubes are represented as coordinates of both ends. Tube length can exceed the cut-off radius significantly. In order to find all cells the tube can interact with, we need to check all boxes in between and next to both ends. Note that on lines #8-11 both tube and cell are updated.

3. GPU implementation

Our GPU implementation was written using NVIDIA CUDA technology. As the existing CPU code was written in C++, it was fairly simple to integrate it with the C-like CUDA kernels. In the next sections, we describe the most important details of the GPU implementation.

3.1. New data model for GPU algorithms

Because structures used by the CPU algorithm are tightly coupled with a visualization code, GPU code uses its own data model. In order to make memory reads coalesce, we changed an array of objects to a separate array for each property and flattered nested structures. Streaming multiprocessor (SM) schedules threads in groups called warps. Coalesced memory access occurs when all threads in a half warp access continuous memory locations. In that case, SM can issue one memory transaction handling multiple threads. A separate GPU model also allowed us to use build-in CUDA types (int4, float4) supported by texture memory. Before each phase the CPU model is converted to the new GPU representation and copied to GPU global memory.

3.2. Cell-cell interactions

In the CPU implementation, this phase takes around 54% of the total execution time, so it was the first candidate to be rewritten.

Algorithm 3: Cell-cell interactions on GPU (first approach)

```
 1  index = blockId*BLOCK_SIZE+threadId
 2  if index < num_of_cells
 3    box1 = computeBoxId(pos[index])
 4    foreach box2 in 27 adjacent boxes of box1  // 26 + box1
 5      foreach cell2 in box2
 6        if cells[index] and cell2 interact
 7          force = calc_force;
 8          cells[index].force += force
 9          calculate concentrations
10          calculate pressure
11        end
12      end
13    end
14  end
```

In the GPU implementation, we used a fine-grained approach. Algorithm 3 presents the code executed by a single thread. Each thread is responsible for calculation of interactions of a single cell. On line #2 we check whether the thread index is not greater than the number of cells (it can happen when the num-of-cells is not a multiple of BLOCK-SIZE).

On line #4 we assume that every box is adjacent to itself. As the reader can see on line #8, we resigned from Newton's third law. Thus the interaction between a pair of cells is computed twice (when we calculate interactions between cell A and B, only A is updated).

The first reason for this is that we wanted to avoid synchronization. Atomic floating-point operations are available only on devices of compute capability 2.x, which we did not utilize. The second reason is that employing Newton's third law would introduce scattered memory accesses that perform poorly on GPU.

The first approach shown in Alg. 3 resulted in a speedup of about 6x, which has shown that it does need further improvements.

3.3. Neighbor list generation

Performance analysis of the cell-cell interactions kernel revealed that the main reason for poor performance is the low occupancy. Our algorithm is memory bound. One approach to coping with memory latency is to run more threads per multiprocessor. GTX 295 is able to schedule up to 1024 threads per single streaming multiprocessor (SM). Because our kernel requires 38 registers per thread, SM can schedule only 384 threads to fit the number of registers it contains.

In order to decrease the register usage, our huge cell-cell interactions kernel has been broken down into two separate stages:

- Neighbor list generation
- Calculations of interactions between cells

3.4. Use of shared memory

Neighbor list generation can be improved using shared memory. Shared memory is visible to all threads of a block. It is almost as fast as registers (when some conditions are met). It can be used as a cache of global memory. We used the method presented in [1].

Algorithm 4: NeighborList with shared memory

```
 1  index = blockId*BLOCK_SIZE+threadId
 2  if index < num_of_cells
 3    box1 = computeBoxId(pos[index])
 4  else
 5    box1 = computeBoxId(pos[blockId*BLOCK_SIZE]) // first cell in the b
 6  foreach box2 in 27 adjacent boxes of box1
 7    synchronize()
 8    index2 = box2*BLOCK_SIZE + threadId;
 9    shared[threadId] = attributes[index2]
10    synchronize()
11    if index < num_of_cells
12      foreach cell2 in box2
13        if cells[index] and cell2 interact
14          Neighbors[index] += cell2
15        end
16      end
17    end
18  end
```

Algorithm 4 requires the block size to be equal to the maximum number of cells in a box. On lines #2-5 we assign the id of the current box to the variable box1. In order to calculate the box Id, we use either the position of the cell at index or position of the first cell in the block. On line #6, we iterate over all adjacent boxes. First, we make sure that all threads reached line #7 by invoking synchronize(). Each thread loads the position and radius of one of the neighboring cells and stores them in shared memory. On line #10, we wait until all of the shared memory locations are populated. When we reach line #12, each thread can iterate through properties of all cells in a neighboring box using values loaded into shared memory. Figure 2 shows a single thread block handling one of the neighboring boxes.

3.5. Use of texture memory

In order to compute cell interactions each thread needs to load properties of all neighboring cells. Those reads are not coalesced and perform poorly on the GPU. Different threads read the same cells multiple times. Fortunately, those accesses are read-only and can be enhanced by cached texture memory. The improvement achieved by the use of texture memory is presented in Figure 3.

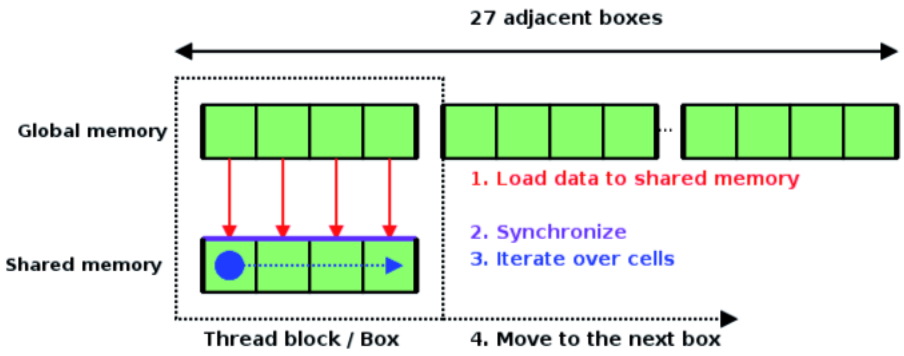


Figure 2. Graphical representation of Algorithm 4 for the block size of length 4. Dashed square represents a thread block handling one neighboring box

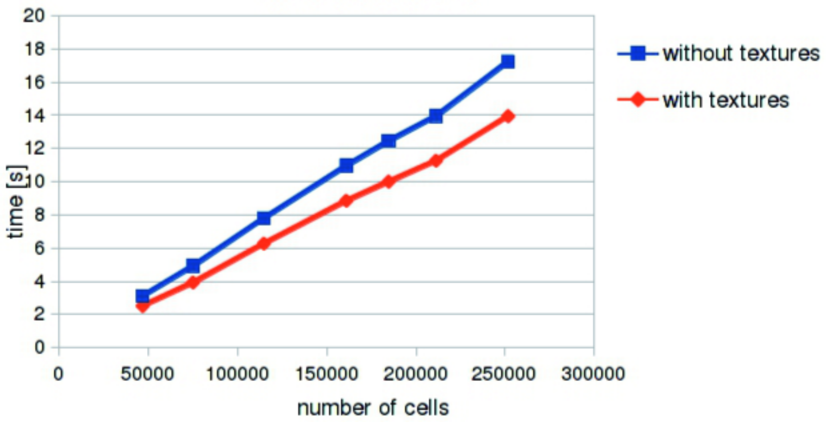


Figure 3. Improvement achieved by the use of texture memory

3.6. Cell-tube interactions

In the serial CPU implementation it was easy to take advantage of Newton's third law and compute cell-tube and tube-cell interactions at once.

However, a parallel implementation would require extensive synchronization to do so. In order to avoid synchronization, this phase has been broken down into two steps:

- tube-cell interactions,
- cell-tube interactions.

The most time-consuming part is searching for objects that interact. Fortunately, devices of compute capability greater than 1.1 support atomic integer operations. By use of an atomicInc function (atomic incrementation), we can create a list of neighboring tubes for each cell when we calculate tube-cell interactions. Having a ready to use list of neighboring tubes, we can calculate cell-tube interactions in a negligible time.

As we mentioned before, GPU kernels use their own dedicated model different from the CPU one. In both tube-cell and cell-tube stages we do not need to know what tubes chain a particular tube belongs to. Because of that, we use just an array of tubes.

Both algorithms 5 and 6 present the code executed by a single thread.

Algorithm 5: GPU version of tube-cell interactions

```
i = blockId*BLOCK_SIZE+threadId
if i < num_of_tubes
  tube = tubes[i]
  foreach adjacent box of tube
    foreach cell[j] in box
      if cells[j] and tube interact
        count = atomicInc(neighborCount[j])
        neighbors[j, count] += i
        force = calc_force;
        tube.force -= force
        calculate_concentrations(tubes[i], cells[j])
        calculate_pressure(tubes[i], cells[j])
        // update only tubes[i]
      end
    end
  end
end
```

Each thread is responsible for the calculation of interactions between a single tube and all cells in its neighborhood. The variable 'neighborCount' stores the number of neighbors of a particular tube. It is initialized with zeros before this routine is called. The variable 'neighbors' is a 2D array containing a list of tubes interacting with a cell at the index 'j'.

Algorithm 6: GPU version of cell-tube interactions

```
i = blockId*BLOCK_SIZE+threadId
if i < num_of_cells
  for j=0 to neighborCount[i]
    tubeIdx = neighbors[j]
    force = calc_force(tubes[tubeIdx], cells[i]);
    cells[i].force += force
    calculate_concentrations(cells[i], tubes[tubeIdx]);
    calculate_pressure(cells[i], tubes[tubeIdx]);
    // update only cells[i]
  end
end
```

Algorithm 6 uses variables 'neighborCount' and 'neighbors' prepared in Alg. 5. Each thread calculates interactions between a single cell and tubes stored in the variable 'neighbors'.

3.7. Use of Page-Locked Host Memory

In each time step, in both phases data need to be converted to the new model and copied back and forth between the GPU and RAM. Memory handling takes up to 40% of cell-cell interactions stage.

The overhead introduced by the memory copying can be mitigated by use of page-locked host memory(pinned memory). CUDA runtime handles pinned memory more efficiently and allows concurrent copying and kernel execution [6]. Use of page-locked host memory resulted in 5% improvement of the execution time of both phases.

4. Results

All tests were conducted on the same machine. We used Intel Xeon E5540 at 2.53GHz CPU and Nvidia GeForce GTX 295 (containing 240 CUDA cores). Machine runs Red Hat Enterprise Linux Server release 6.0. CUDA version was 3.2.16.

For the comparison of neighborsList kernels, texture memory, and speedups of cell-cell interactions we used randomly generated inputs with no tubes. In order to prevent cells from hypoxia and death, oxygen consumption was set to zero.

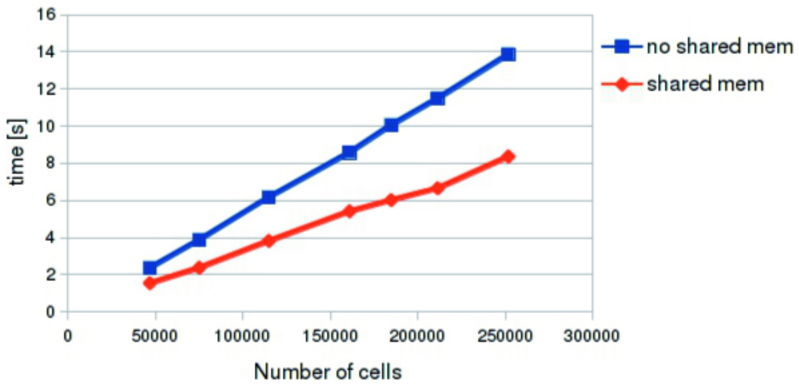


Figure 4. Comparison of neighbors-list kernels execution time (the less the better)

In Figure 5 we can see that “neighbor list” approach outperforms the all-in-one-kernel algorithm. Figure 4 shows that the use of shared memory has brought significant improvement of neighbor list generation.

Speedups of the cell-cell interactions phase is presented in Figure 6. Our CUDA kernel is around 57 times faster than its CPU counterpart. The memory copying and data conversion decrease speedup of the whole phase to 30x. Speedup of the entire simulation fluctuates around 20x.

The overhead of the memory handling is substantial. It accounts for 40% of the execution time of both phases. The share of the model conversion and data-transfer in the execution time of cell-cell interactions stage is shown in Figure 7.

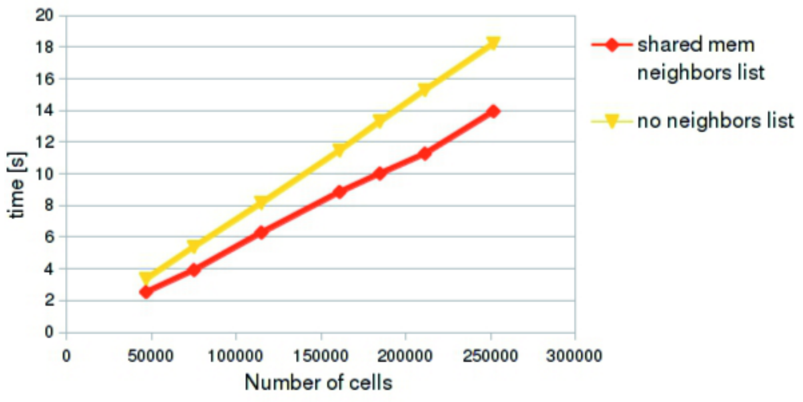


Figure 5. Comparison of approach with neighbors-list and no-neighbor-list

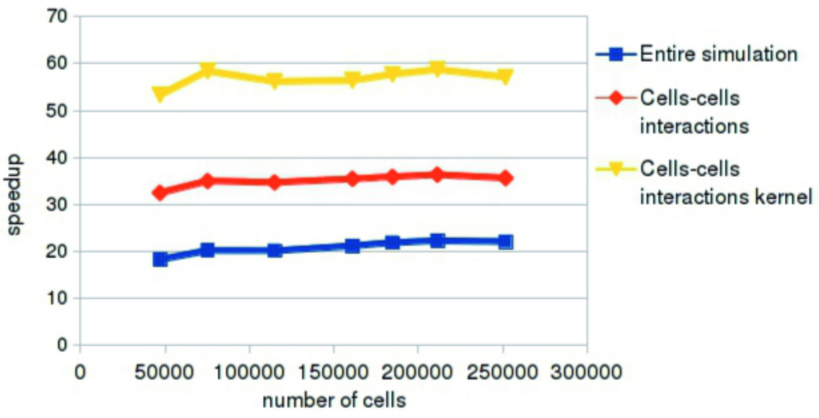


Figure 6. Speedups of cell-cell interactions

Performance tests of tube-cell interactions have been performed on randomly generated input with an extremely large number of tubes. Figure 8 shows speedups of the tube-cell interactions phase. The overall speedup decreases with an increasing number of tubes. There are a few reasons for that. First, the more tubes, the longer it takes to load the input data. The second reason is tube-tube interactions phase. Normally, blood vessels are sparse and hardly ever interact with each other. In our extreme case, tubes are so dense that their interactions considerably affect overall performance. When the number of tubes reaches 9996, tube-tube interactions take 29% of the total execution time.

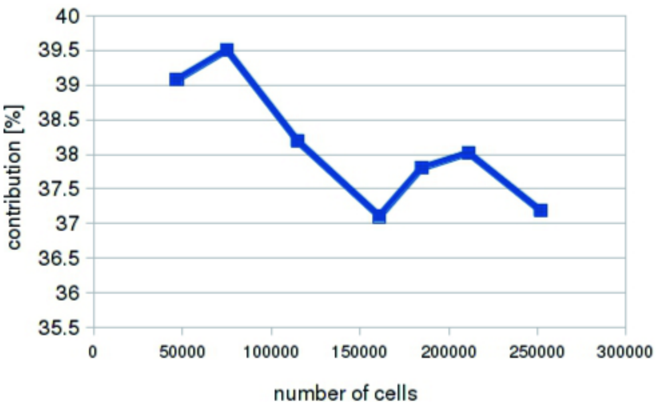


Figure 7. The contribution of the model conversion and data-transfer to the execution time of cell-cell interactions stage

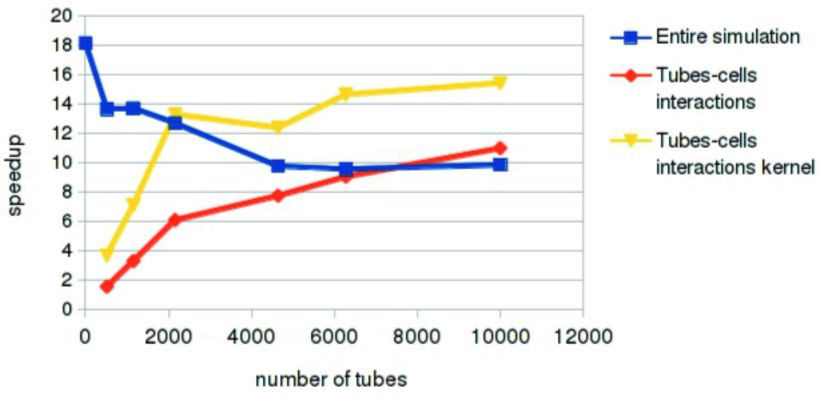


Figure 8. Speedups of tube-cell interactions

5. Conclusions

In this paper, we have shown that GPUs can be used to accelerate the modeling of a complex biological process. As shown in section 3.2, the simple (naive) GPU version of cell-cell interactions runs 6 times faster than its CPU counterpart. By use of advanced CUDA mechanisms such as shared memory, textures, atomic operations, and tailoring algorithms to GPUs, we managed to increase the speedup of our CUDA kernel up to 57x. The entire simulation runs 10-20 times faster than the single threaded CPU version.

The speedup obtained is quite satisfactory, taking into account the complexity of the system modeled. However, there are still implementation problems that can be solved better. The overhead introduced by the memory moving can be totally eliminated by implementing the entire simulation on GPU. Some stages may work slower than on CPU, but the overall performance will be improved significantly. Rewriting everything to CUDA would also enable us to make the visualization more efficient (objects could be rendered directly from the GPU memory [3]). Another possible improvement is to perform calculations iteratively on subsets of cells. In the current solution, the number of objects modeled is limited by the amount of the GPU memory, which is normally less than the amount of installed RAM. We could also take advantage of multiple GPUs. A lot of contemporary GPUs are made in the SLI technology. Unfortunately, CUDA is not able to distribute data across all graphic cards automatically. It needs to be done explicitly by a programmer.

The modeling system can be used in the future as a framework for a virtual laboratory and problem solving environment for educational purposes and finally for *in silico* experiments, which can play the role of angiogenesis assays in the planning of a cancer treatments.

Acknowledgements

This research is financed by the Polish Ministry of Higher Education and Science, project N519 579338 and partially by AGH grant No. 11.11.120.865.

References

[1] Anderson J. A., Lorenz C. D., Travesset A.: *General purpose molecular dynamics simulations fully implemented on graphics processing units*. Journal of Computational Physics February, 2008.

[2] Aubert M., Chaplain M. A., McDougall S. R., Devlin A., Mitchell C. A.: *A Continuum Mathematical Model of the Developing MurineRetinal Vasculature*. Bulletin of Mathematical Biology, 2011.

[3] Green S.: *Particle Simulation using CUDA*. NVIDIA, 2010.

[4] Li W., Hutnik M., Smith R., Li V.: Understanding angiogenesis. [online], 2011. `http://www.angio.org/ua.php`

[5] Nguyen H.: *GPU Gems 3*. Addison-Wesley Professional, 2007. Chapter 31, Fast N-Body Simulation with CUDA.

[6] NVIDIA.: *CUDA C Programming Guide*. NVIDIA, 2010.

[7] Shirinifard A., Gens J. S., Zaitlen B. L., Popawski N. J., Swat M., Glazier J. A.:. *3D Multi-Cell Simulation of Tumor Growth and Angiogenesis*. Public Library of Science, 2009.

[8] van Meel J. A., Arnold A., Frenkel D., Zwart S .F. P., Belleman R. G.: *Harvesting graphics power for MD simulations*. Molecular Simulation, 34: 3, pp. 259–266, 2008.

[9] Wcisło R., Dzwinel W., Yuen D. A., Dudek, A. Z.: *A 3-D model of tumor progression based on complex automata driven by particle dynamics*. Journal of Molecular Modeling, 2009.

Affiliations

Marcin Worecki
AGH University of Science and Technology, Department of Computer Science, Krakow, Poland, `wor18@wp.pl`

Rafał Wcisło
AGH University of Science and Technology, Department of Computer Science, Krakow, Poland, `wcislo@agh.edu.pl`

Received: 9.12.2011
Revised: 9.01.2012
Accepted: 30.01.2012