Construction and Implementation of a Simple Agent-Based System on GPU-Architectures

Günter Schneckenreither[1,2 *], Stefan Hepp[2], Daniel Prokesch[2]

[1] dwh Simulation Services, Neustiftgasse 57-59, 1070, Vienna, Austria; *guenther.schneckenreither@dwh.at

[2] Mathematical Modelling and Simulation Group, Inst. f. Analysis and Scientific Computing, Vienna University of Technology, Vienna, Austria40 Vienna, Austria

Abstract. Agent-based modelling and simulation is still an upcoming approach for microsimulation. But a large number of agents with advanced dynamics and interactions requires sophisticated algorithms and lots of computational effort. We try to implement a rather simple but special agent-based model model on GPU-architectures (graphics processing unit). This contribution presents the GPU implementations and and investigates its practicability. Results show that implementation of agent-based systems on a GPU-architecture can deliver enormous speed-ups. Depending on the detailed structure of the system highly advanced algorithms deliver another great performance-boost. The combination of both approaches can deliver up to 1.000 times faster execution of the same simulation.

Introduction

Agent-based models are usually applied when global behaviour of a system is best described by the micro- or macroscopic dynamics of a large set of similar but individual objects. These unique objects are called agents since their properties change individually and depend on the agents specific field of vision of the global system. A typical area of application is for example socio-economics. Every individual of a population can be represented by an agent with distinct features based on statistical data.

A large number of agents with advanced dynamics and interactions requires sophisticated algorithms and lots of computational effort. We try to implement a rather simple but special model on GPU-architectures (graphics processing unit), which are optimized for highly data-parallel operations. Our goal is to investigate the practicability and finally use such advanced programming techniques in order to increase the capacity of agent-based models.

1 Definition of the Test Model

The test model is a rather simple spatial system of moving agents. On a two-dimensional domain two sets of agents interact with each other depending on their distance. The first set are so-called 'signposts', whose positions are arranged at initialization and do not change over the course of simulation. Besides a fixed position the signposts hold a second static vector, which collectively defines a flow field on the domain. The second type of agents are referred to as moving agents since they hold two vectors which define their position and velocity.

In a simple scenario the evolution of the system is purely explicit and consist of two operations which are applied on the moving agents. The first operation changes the position of the agents according to their velocity:

$$\mathbf{x}(t+1) = \mathbf{x}(t) + \mathbf{v}(t) \cdot dt$$

The second operation recalculates the velocity depending on the current velocity and the flow-field-vectors of 'neighbouring' signposts. The term 'neighbourhood' originates from cellular automata and defines a weighted set of interacting signposts for each moving agent. The neighbourhood-function can be for example a two-dimensional bell-shaped curve (normal distribution) or any other function, which describes a decaying likelihood of interaction.

Due to performance issues the neighbourhood-function may have a maximal radius for non-zero weights (compare search algorithms below). Once all weight factors ω_{ij}, $j \in \{1, \ldots, k\}$ for a moving agent a_i are calculated, the velocity changes according to an update function:

$$\mathbf{v}_i(t+1) = \Phi(\mathbf{v}_i(t), \omega_{ij_1} \cdot \mathbf{v}_{j_1}(t), \ldots, \omega_{ij_k} \cdot \mathbf{v}_{j_k}(t))$$

SNE Simulation Notes Europe – Print ISSN 2305-9974 | Online ISSN 2306-0271
SNE 22(3-4), 2012, 169-176 | doi: 10.11128/sne.22.tn.10153

For agent-based systems the dynamics of different agents are usually mutually dependent. In this case the signposts would additionally change their properties according to a set of neighbouring moving agents. This type of coupled relation would for example occur if the motion of agents should obey or at least approximate physical flow of particles or a fluid etc., but is not focus of this investigations. To highlight the lack of such advanced properties we arrange a circular flow-field (flow-field vectors are tangential to the circular flow) and show that the moving agents drift from the centre to the boundary of the domain.

2 GPU-Architecture and Hardware

In the past decade a variety of new parallel computing architectures have been developed. Besides parallelisation on multiple CPUs (central processing unit), CPU cores or even physical machines it is possible to use graphics devices for performing calculations in parallel. This strategy is known as GPGPU (general-purpose computing on graphics processing units). Some of the most common architectures are CUDA (compute unified device architecture) for NVIDIA graphics devices [1], AMD FireStream for ATI devices [2] and OpenCL (open computing language), an abstraction layer for accessing the calculating capacity of multiple hardware devices [3]. The advantage of GPUs is the high throughput on data-parallel operations. On the other hand implementation is more complicated since the physical structure of the device is crucial for constructing the algorithm.

We use C++ and CUDA on a machine with the following specifications: 'AMD Athlon Phenom II 920' processor at 2.8Ghz, 3 GB of working memory and the consumer graphics device 'NVIDA GeForce GTX 260' with 896 MB GDDR3 on-board memory using a 448-bit memory interface at 2.2Ghz, and a GT200b core with 24 streaming multiprocessors (SM), running at a core clock of 633Mhz with a shader clock of 1.4Ghz, providing Compute Capability 1.3 with CUDA Toolkit 2.3 and NVIDIA driver version 196.21.

A single SM executes one or more threadblocks consisting of maximally 512 threads [4, Sect. A.1.1]. Threads in a thread block are executed in groups of 32 parallel threads (thread warps) and execute the same instruction path. Executing different control flows within a warp (by predicated execution) reduces performance.

Major performance gain can be achieved through reducing and optimizing memory operations [5]. Several memory spaces are available within the CUDA programming architecture:

- Host memory: This is the RAM (random-access memory) used by the CPU only. Transfer of data to the GPU memory is rather expensive and happens through a PCIe bus with about 5 GB/s.
- Global device memory: This type of memory is attached to the GPU and provides a bandwidth of 50 to 80 GB/s. Copying from and to the (pinned) host memory can happen simultaneously to execution of code. Access to global memory is not cached and has a high latency, but this can be be hidden efficiently by using a large number of threads.
- Constants and texture memory: This type of memory is cached by the SM and provides fast read access. We use textures for the signpost data and the acceleration structures since they are modified infrequently. The maximum size of a texture is limited to 2^{16} by 2^{15} texels for 2D textures, however this is enough to store up to 2^{31} signposts.
- Shared memory: Every thread can use up to 16 KB of shared memory, which has a very low latency and allows communication between threads in a thread block but not different kernels.
- Registers: Every multiprocessor provides 8192 or 16384 32-bit registers in total which are shared between all threads of all thread blocks.

The number of available registers as well as the absolute maximum numbers for active warps, threads and blocks per SM depend on the 'Compute Capability' of the GPU.

The number of threads per thread block is further limited by the number of registers required for a thread block, which is determined by the number of registers used by the kernel multiplied by the number of threads per block and must not exceed the number of registers available on a SM.

The number of active blocks per SM is also limited by the sum of register and shared memory requirements of the blocks. [4, Sect. 4, 5.2, App. A].

3 Implementation on the GPU-Architecture

Additionally to the statements in Section 1 on coupled relations, exchange between the host system and the GPU are neglected. A simulation might require to store intermediate results on the hard-drive or to exchange data with another sub-model, which is not executed on the GPU. Such structures depend on the model and require balancing of computing time and accuracy [6]. The simulation is executed and visualized exclusively on the graphics device since we are interested in the basic performance of a GPU simulation.

There are two basic strategies for implementing the neighbourhood approach:

1. For each signpost, find all agents within the neighbourhood.
2. For each agent, find all signposts within the neighbourhood.

In the first case it is possible that multiple signposts try to update the velocity vector of the same moving agent, which would require performance decreasing synchronization mechanisms. Additionally, the second approach makes it more comfortable to implement optimized data structures for finding neighbouring signposts because their position does not change during simulation. Such data structures can be pre-calculated before the actual simulation and are discussed later on.

The following initialization steps are required:

1. Load or generate initial positions and velocities.
2. Generate the acceleration data structures for the neighbourhood search depending on the algorithms described below.
3. Copy all signpost and agent data to the GPU and initialize any constants and textures used by the kernels.

During simulating the kernel applies two operations per agent and time-step:

1. Find all signposts $b_1,\ldots,b_k$ which influence the agent a_i at position $\mathbf{x}_i(t)$.
2. Calculate the new velocity vector $\mathbf{v}_i(t+1)$ and the new position $\mathbf{x}_i(t+1)$ for agent a_i.

Performance of the second operation is determined by the number of memory reads required to collect the velocity vectors of all neighbouring signposts. This number ($k \cdot C$) is not constant but bounded by the total number of signposts.

The time-critical operation is finding all signposts that influence a given moving agent. For this task several algorithms were implemented and compared.

3.1 Linear Search

The simplest method for finding all signposts within a certain radius of an agent is to walk through the list of all signposts and compare the distance. Velocity vectors of appropriate signposts are added to the velocity vector of the current agent and then divided by $k+1$, where k is the number of neighbouring signposts (linear non-weighted influence).

The advantage of this approach is that all threads perform equal memory reads from the signpost array and thus can be cached very efficiently using texture memory. Furthermore, only a few instructions are needed to perform the lookup and there is almost no divergent control flow.

The only difference in control flow between threads arises when the influence of a signpost which is not within the neighbourhood region of a moving agent is ignored. On the other hand every agent needs to read and process each signpost in every step. The performance is therefore $\Theta(n \cdot m)$ for n agents and m signposts. Even for small m this method is slower than using a uniform grid.

3.2 Uniform Grid Search

In order to speed things up the domain is divided into a regular grid. The cell index of an agent or a signpost can quickly be determined without any memory lookups (except for the constants specifying the grids structure). Prior to the simulation the list of all signposts is sorted by cell index so that signposts of the same cell are stored consecutively. An additional index array stores the index of the first and the last signpost of each grid cell.

If the cell size is greater or equal to the neighbourhood radius only signposts in the same cell as the agent and in at most three adjacent cells can affect the motion of the agent. The adjacent cells which need to be checked can be determined by comparing the distance of the agent to the borders of its cell with the neighbourhood radius (no lookups into global memory except for cached constants).

To calculate the new velocity of an agent (from its cell index) the index of the first and the last signpost in each of the four 'neighbouring' cells is taken from the index array.

The signposts which actually influence the agent are then searched in the four resulting lists of signposts using a linear search as described above.

If there are at most k signposts in each cell, at most $4 \cdot k$ signposts are checked for every agent in every step, therefore the performance of this algorithm is of order $O(4n \cdot k)$for n agents.

This method works best if all signposts feature the same neighbourhood structure (as it is the case in this simulation) and if they are distributed uniformly without areas of strong aggregation. Generally a dense distribution leads to a larger value for k, whereas a sparse distribution can lead to empty cells and memory consumption in the index array.

If the grid diameter is smaller than the neighbourhood radius, more cells have to be checked for neighbouring signposts but fewer signposts must be neglected. However, since every agent still needs to find and process all signposts within its radius, a very small cell size cannot reduce the number of memory reads below the number of signposts affecting the agent but introduces a larger overhead for index array reads.

Assuming that the influence of the signposts does not depend on the distance between the agent and the signpost, a smaller grid size could still improve the performance for dense distributions. For every cell the velocity vectors of all signposts which completely overlap the cell can be accumulated and stored in advance, thus reducing the number of individual signposts needed to be checked during simulation.

The algorithm to sort the signposts and create the index array can also be implemented efficiently with CUDA using the following algorithm (compare [7]):

1. Allocate a memory array for the index array.
2. For every signpost, calculate its grid cell index based on its position. Clearly this can be done by starting a kernel for every signpost.
3. Sort the signposts regarding to their cell index. This can be done on the GPU e.g. by using the radix-sort implementation of the CUDA Thrust library.
4. For every grid cell, find the index of the first and the last signpost in the sorted signpost array. This can be done by starting a thread for every signpost.

Since the index calculation can be done entirely on the GPU, this algorithm can also be used to handle moving signposts or to let the agents interact with each other.

3.3 Balanced Tree Search

To tackle the memory overhead for non-uniform signpost distributions, a third search method was implemented using cells with variable size.

Instead of a uniform grid a kD-tree is constructed so that each leaf has the same depth and each subdivision of an area is done so that the two new areas contain roughly the same number of signposts while keeping the cell size larger than the signpost diameter.

The algorithm is similar to the uniform grid search method, except that finding an agents cell requires $\log_2(m)$ memory read operations to descent the kD-tree, where m is the overall number of cells. Again adjacent cells need to be checked if the distance of an agent to the cell border is smaller than the neighbourhood radius. This search is more complex but can be pre-calculated for every cell.

Early tests have shown that even if the signposts in adjacent cells are not checked, the cell search algorithm leads to a decreased performance compared to the uniform grid approach. The only benefit could be that in a non-uniform or sparse distribution fewer cells are needed to get the same maximum number of signposts per cell. But even a 2048 by 2048 cell uniform grid, which is more than enough for our purposes, requires only about 33Mb of additional memory for the index array. Therefore the implementation of this search method was not completed in favour of the uniform grid search.

3.4 Caching and Other Search Algorithms

A cache could be used to reduce the number of signpost-checks. The only memory space where a persistent cache between two kernel executions can be implemented is global memory. The cache must have a fixed size because resizing allocated memory is not possible during kernel execution and allocating memory is a costly operation.

Consequently implementing a cache is beneficial only if accumulated influences can be calculated for multiple time steps and the resulting information does not require too much memory space. Since signpost influences can vary frequently it is difficult to calculate and store useful cache data without much overhead. In sparse distributions only few signposts need to be checked. Thus even a cache hit can result in more memory operations compared to an implementation without cache.

Other alternatives to search for signposts are a k-Nearest-Neighbour (k-NN) search or a range search combined with storing the signposts as nodes of a kD-tree. However this has similar advantages and drawbacks as the balanced tree search; additional index arrays are not required but an additional $\log_2(m)$ overhead in memory reads and a stack for the tree lookups is required.

This could be implemented using bit-arrays which can be stored in registers or shared memory to avoid costly global memory access if the traversed tree is a binary tree and the maximum tree depth is fixed at compile-time.

The control flow and the memory access pattern can diverge greatly between consecutive threads, which has a negative performance impact for CUDA kernels. Therefore the benefit of such more complex algorithms compared to the uniform grid search is a lower memory requirement for large and sparse datasets and the ability to search for arbitrary ranges at the cost of more complex algorithms which require more control flow, arithmetic and memory operations. Arbitrarily deep trees cannot be handled due to the lack of dynamic data structures in the CUDA architecture.

As already mentioned before, both signposts and agents can be sorted by a grid cell index on the GPU. If this is done for every simulation step, this can be used for several things. If the signposts are sorted, they can be moved around while keeping the acceleration data structures up-to-date. If the agent array is sorted, agents can interact with each other efficiently.

An additional benefit of sorting the agents is that consecutive agents will usually fall into the same grid cell and therefore threads calculating the next step of consecutive agents will have a similar control flow and perform similar memory reads which the hardware is able to coalesce into a single read or at least increase the likelihood for a cache hit for texture memory reads.

As an additional optimization data required by all threads for agents in the same cell like the list of signposts in this cell can be cached in shared memory, thus speeding up random reads of the same data by multiple threads. However the performance impact of this optimization has not been tested.

4 Further Technical Details and Visualization

For debugging and comparison purposes, two simulation kernels have been implemented. The first implementation uses the CPU exclusively, the second kernel transfers all data to the GPU memory at initialization and performs the simulation including visualization entirely on the GPU. If the CUDA simulation results should be written to disk, the agent array needs to be written back to the host memory for every frame which should be stored.

During development and for analysing results of a simulation – especially in the case of a spatial model – it can be very useful to have a decent front-end for visualizing the dynamics of the system. Such a front-end was build using OpenGL. Several features of the system can be displayed or masked during simulation. The signposts and optionally the grid are pre-rendered once into a texture using an OpenGL frame-buffer. For the agents, three different modi have been implemented.

Moving agents are either rendered as soft points with three pixel radius using OpenGL and supplying the coordinates and colour of the agents as a vertex array, as simple triangles, again using OpenGL and vertex-arrays, or by drawing the agents as single pixels into a texture buffer and displaying the texture. In all three modi, the data is written either by the CPU to a memory-mapped OpenGL buffer, or – if CUDA is used – directly from graphics card memory to an OpenGL buffer in the graphics card memory using a CUDA kernel and the CUDA OpenGL interoperability methods.

In the render-to-texture mode a simple scatter algorithm is used, i.e. for every agent a kernel is started which writes the pixel at the agents position. This can result in multiple writes to the same pixel, where the order of the writes is random for the CUDA implementation, which can lead to rapid colour changes between frames (on the CPU, all agents are processed in serial, so there are no conflicts due to concurrency).

A better approach for the CUDA implementation would be a gather algorithm, similar to the algorithm used to create the index array of the uniform grid search. However, since the performance of the OpenGL implementation using vertex-buffers is as fast as the current render-to-texture implementation, the more complicated gather algorithm was not implemented.

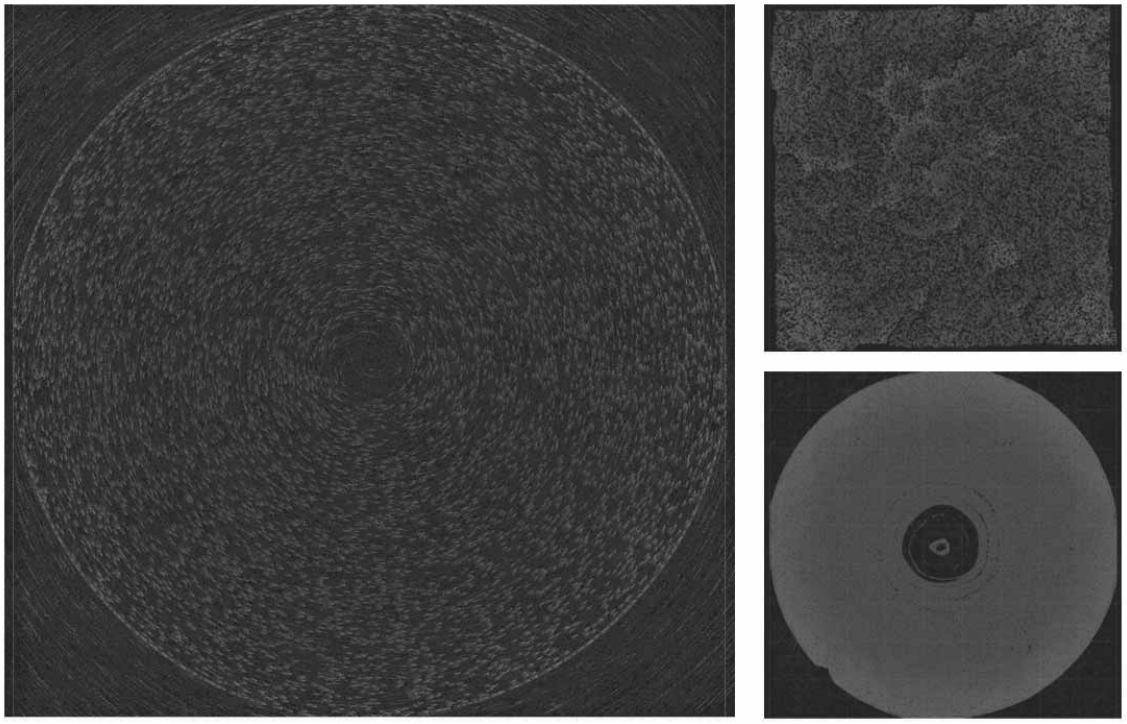

Figure 1. Few agents represented as triangles (left) and 1,000,000 agents drawn as pixels (right).

The speed of the agents is visualized using the colour of the points or triangles, ranging from red for slow agents to green for fast agents (Figure 1).

	CPU			GPU		
Agents	Points	Triangles	Texture	Points	Triangles	Texture
10,000	1.5	2.0	3.5	2.1	2.2	2.3
1,000,000	90.0	220.0	70.0	8.0	18.0	7.0

Table 1: Additional overhead needed for rendering.

5 Results

To measure the performance of the implementation, the average computation time per step was measured. For all measurements, the same setup was used: The signposts are placed randomly in a rectangular field directing the agents into circular motion, the agents are placed randomly on the domain and boundary conditions are reflective. Visualization has been disabled for these benchmarks.

Using a uniform grid data structure with 100 cells increases the performance even for a relatively low number of signposts (Figure 2). Between the CPU and the GPU implementation, speed-ups between 70 times to 100 times can be observed for the linear search approach. If a uniform grid is used, the GPU is about 50 to 70 times faster than the CPU implementation. As expected the simulation time is linear in the number of agents. For CUDA, a small constant overhead can be observed for less than 100,000 agents.

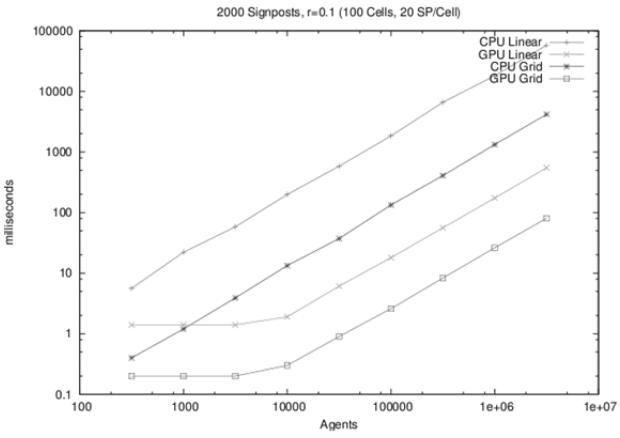

Figure 2. Calculation time depending on the number of agents.

Agents	Signposts	Radius	CPU (ms)	GPU (ms)	Signposts/Cell	Speed-up
		0.001	33.8	0.5	0.025	68
		0.0032	37.0	0.7	0.250	53
10^5	10^5	0.01	93.6	3.8	2.500	25
		0.032	603.7	30.1	25.000	20
		0.1	5224.0	213.1	250.000	25
10^6	10^6	0.001	652.5	10.3	0.250	63
10^7	10^6	0.001	4966.8	187.0	0.250	26

Table 2. Performance depending on number of agents and neighbourhood radius.

Observing the average time per simulation step over the number of signposts exhibits the same linear behaviour.

For the linear search again a speed-up of about 100 times can be achieved by CUDA. For the grid search the speed-up is again lower between a factor 10 and 20.

For the linear search approach all kernels perform the same memory reads in the same order and therefore can be coalesced and cached more effectively than for the grid search approach which issues memory reads in a more random fashion (this could be tackled by sorting the agents by their grid cell index).

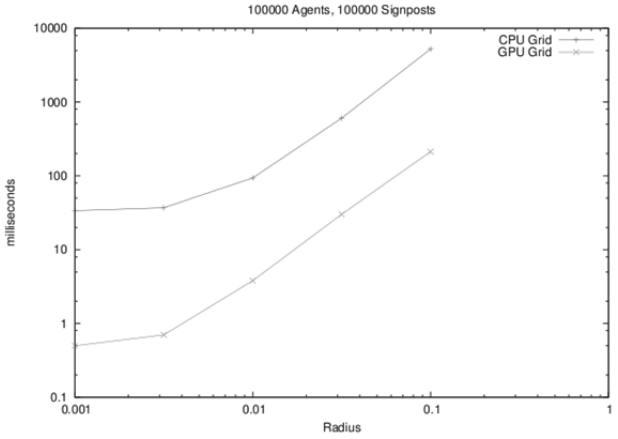

Figure 3. Calculation time depending on the neighbourhood radius for a large number of agents.

When a grid is used with k cells to speed up the signpost search for m signposts, only $\frac{4m}{k}$ signposts will be checked on the average per step by every agent instead of all m signposts.

This leads to a speed-up of the simulation (except for a higher memory access penalty due to more random memory reads).

If the grid cells are made as small as possible, i.e. the diameter of the neighbourhood, the simulation time increases quadratically with the signpost diameter and linearly with the number of signposts, as shown in Table 2 and Figure 3.

Development (familiarizing with CUDA and implementation) took about 6 man-weeks for advanced programmers. The additional CPU-only implementation imposed nearly no additional overhead since the code for the CPU implementation and the CUDA kernels is quite similar, and simplified debugging the algorithms.

6 Conclusions

Results show that implementation of agent-based systems on a GPU-architecture can deliver enormous speed-ups. Depending on the detailed structure of the system highly advanced algorithms as discussed in Section 4 deliver another great performance-boost. The combination of both approaches can deliver up to 1,000 times faster execution of the same simulation.

Of course the applicability of both GPGPU and advanced search algorithms depends on the model itself. In general every model requires distinct implementation techniques, which are often more complex than a straight-forward implementation. However, depending on the structure of a model higher implementation effort can be very profitable.

The test model inhibits a unidirectional structure, which is predestined for applying highly parallel techniques. But also large spatial agent-based models with coupled relations can be improved using a (uniform) grid approach or search trees.

Depending on the interaction of agents static acceleration data structures can be used, but also dynamic look-up tables can be more efficient than a linear search approach.

The crucial performance gain for all approaches can be achieved by optimizing memory access.

References

[1] www.nvidia.com/object/cuda_home.html

[2] ati.amd.com/technology/streamcomputing

[3] www.khronos.org/opencl

[4] 'NVIDIA CUDA Programming Guide 2.3.' NVIDIA Corporation, October 2009.

[5] 'NVIDIA CUDA Best Practices Guide 2.3.' NVIDIA Corporation, July 2009.

[6] Stam, J. (2009). Maximizing GPU Efficiency in Extreme Throughput Applications. Video Presentation presented at the NVIDIA GTC09, October 2009.

[7] Tonge, R. (2009). . Spatial Data Structures for Massively Parallel Computing.Video Presentation presented at the NVIDIA GTC09, October 2009.

Submitted: March, 2011
Revised: July 15, 2012
Accepted: October 10, 2012