Original published: K. H. Hoffmann, M. Hofmann, J. Lang, G. Rünger, and S. Seeger. Accelerating physical simulations using graphics processing units. *it - Information Technology*, 53(2):49–59, 2011. Online available at http://dx.doi.org/10.1524/itit.2011.0625.

Accelerating physical simulations with graphics processing units

Karl Heinz Hoffmann, Michael Hofmann, Jens Lang,
Gudula Rünger, Steffen Seeger

1st March 2011

Abstract

Graphics processors receive gaining popularity in all fields of application in which there is a need for high computational power. Also, in scientific computing, using graphics processing units to speed up simulations is an active field of research. In this article, a graphics processor implementation of a simulation of anomalous diffusion is presented.

Furthermore, in this article the different frameworks for graphics programming such as CAL, Brook+, CUDA and OpenCL with their specific properties, are compared. Additionally, an overview of different physical applications that have been implemented on graphics processors successfully is given.

Zusammenfassung

Grafikprozessoren erfreuen sich steigender Beliebtheit in allen Anwendungsbereichen, in denen es auf hohe Rechenleistung ankommt. Auch im Wissenschaftlichen Rechnen sind sie Gegenstand aktueller Forschung. In diesem Artikel wird eine Grafikprozessor-Implementierung zur Simulation anomaler Diffusion in porösen Materialien vorgestellt.

Weiterhin werden in dem Artikel die verschiedenen Framework zur Grafikprozessorprogrammierung wie CAL, Brook+, CUDA und OpenCL mit ihren spezifischen Eigenheiten kurz vorgestellt. Außerdem gibt der Artikel einen Überblick über verschiedene physikalische Anwendungen, die bereits erfolgreich auf Grafikprozessoren implementiert wurden.

1 Introduction

The fast increasing computational power of graphics processing units (GPU) offers new possibilities for all kinds of parallel applications. Not only graphics-related computations can be implemented on recent GPUs, but also general-purpose computation. Especially in simulations of physical phenomena, which have a huge demand for computational power, GPUs can be employed to speed up computations. Currently, the peak performance of

GPUs is approximately 5 to 10 times higher than of comparable CPUs. Furthermore, the performance of GPUs is growing faster than of CPUs. Consequently, GPU algorithms are becoming increasingly important in high-performance computing in the near future.

In the early times of general-purpose graphics computing, GPUs were programmed on a low level via an assembler language [23], or the capabilities of the OpenGL graphics library have been used [6]. In 2008, Nvidia delivered its CUDA framework [17] for high-level programming of Nvidia graphics processors. As CUDA made GPU programming easy to handle, general-purpose GPU computing became popular very quickly. CUDA and other comparable frameworks enable also non-experienced programmers to implement their algorithms on GPUs.

In this article, we give a short introduction to GPU programming and present an overview on several popular physical simulations that were implemented for GPUs. Furthermore, we present a GPU implementation of a specific application for the simulation of anomalous diffusion in a GPU implementation [10].

The rest of this article is organised as follows. The specifics of GPU programming including an overview on programming frameworks is presented in Sect. 2. Section 3 gives an overview of a variety physical applications implemented on GPUs. In Sect. 4, an implementation of the anomalous diffusion simulation is described. Section 5 concludes the article.

2 GPU Programming

Programs that use general purpose GPU computing consist of two parts: a host part that is executed on the CPU and a device part that is executed on the GPU. As the GPU is unable to perform I/O operations, the input data needed has to be provided by the CPU.

A GPU programming framework is a set of software libraries and programming tools which aid the programmer to develop GPU programs. For the host part of the program, the libraries provide functions to find out how many graphics devices are present on the machine and what properties they have. Furthermore, they allow the copying of data to and from the device memory and to call functions that are executed on the GPU. These functions are called *kernel functions*. For the device program, a compiler is provided. Additionally, in some cases, the framework includes a debugger or a profiler. The runtime environment for device programs is usually provided by the graphics driver, which is aware of the general purpose compute capability of the graphics device.

For the device part of the program, a special programming language is used. Often, this programming language follows the C99 standard while some constructs such as static variables or recursive function calls are forbidden.

GPUs usually have a large number of compute cores (several hundreds). They can execute a large number (up to some tens of thousands) of data-parallel threads simultaneously. As these threads are very light-weight, tasks can be distributed in a very fine-grained way. Often, a thread carries out only a very small task before being terminated and a new thread being started. The calls of kernel function return immediately.

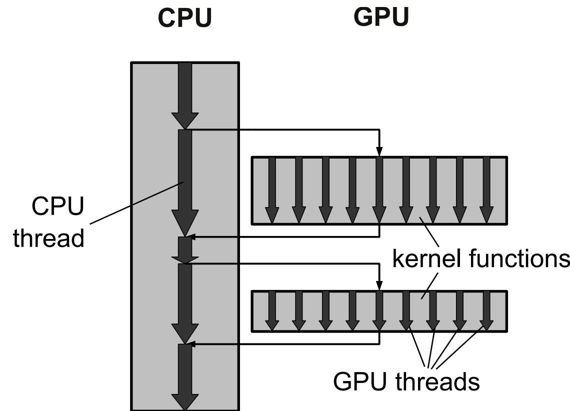

Figure 1: Program flow of a GPU program: the main part is executed on the CPU as a CPU thread (host); the computational part is executed on the GPU by kernel functions (device).

Hence, the CPU and the GPU can work concurrently.

2.1 Hardware

A modern GPU as shown in Fig. 2 consists of several *multiprocessors*. Each multiprocessor consists of a number of *stream processors*. These stream processors are the actual compute cores. All stream processors within one multiprocessor execute the same code according to the SIMD execution model.

All stream processors of a multiprocessor share a large number of registers (in the order of magnitude of several thousands). Additionally, each of them is assigned a fast private memory. For all stream processors of one multiprocessor, there is a single local memory which can be used for fast data exchange. Furthermore, there is a large—but slow—global memory on the device. On Nvidia graphics devices, there is no implicit cache for the global memory. In contrast, on AMD graphics devices, memory accesses to the global memory are cached.

Global memory read accesses and write accesses should be performed in *coalesced* order. This means that consecutive threads should access consecutive memory addresses instead of reading or writing randomly. Coalesced memory accesses bring a performance gain as data can be retrieved from or stored on the memory in larger chunks. In contrary, random accesses have to be processed independently and result in a stall of the execution unit.

Due to their main purpose of graphics processing, GPUs had the shortcoming of being able to process only floating point operations with single precision for a long time. Since mid-2008, GPUs are able to execute double precision calculations. But these double precision computations still do not achieve the same performance as single precision computation. Hence, in some cases, it can be favourable to perform "mixed precision" calculations as proposed in [13] which means that some less critical calculations are

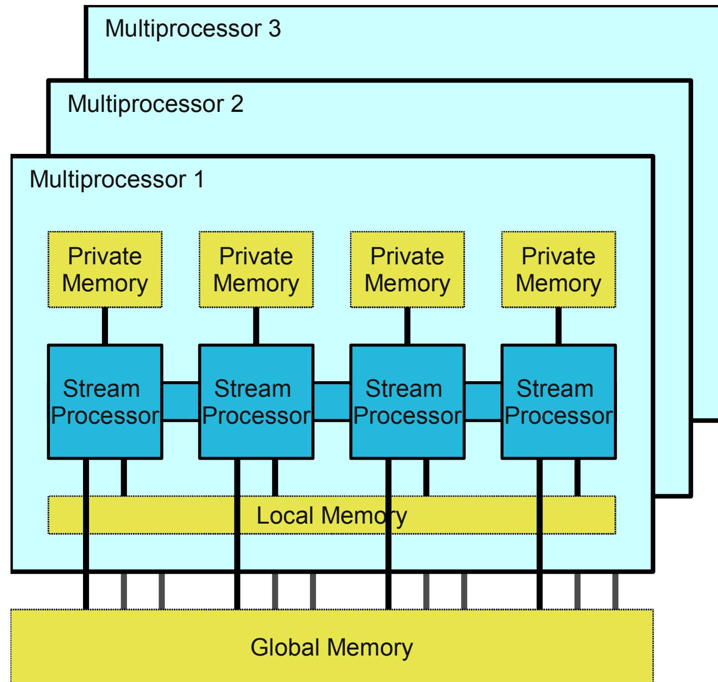

Figure 2: Simplified layout of the hardware architecture of a GPU device.

executed with single precision while for the result, double precision accuracy is achieved.

2.2 CAL

CAL is a low-level API for AMD's GPUs. It allows programming the GPUs in the so-called *Intermediate Language*. The Intermediate Language is an assembler-like, hardware-independent language. In this context, "hardware-independent" means that it is an abstraction from the real hardware of the specific AMD GPU, however it cannot be used for other manufacturer's GPUs.

2.3 Brook+

Brook+ is a the high-level language for AMD GPUs based on the Stanford University project *BrookGPU* which ported the stream programming language *Brook* to graphics hardware [3,5]. Brook implements the stream programming concept. Brook+ is built on top of CAL.

The basic elements of stream programming are *streams* and *kernels*. Streams can be seen as infinite vectors of integer or floating point numbers. They are manipulated or combined with other streams by kernel functions. This programming model makes it easy for the programmer to formulate an algorithm in a way that can be parallelised by the compiler. Kernel functions are executed by the GPU in parallel. Each stream processor of the GPU is working on one vector entry.

Currently, Brook+ supports only AMD's GPUs. Its predecessor BrookGPU additionally supported OpenGL, DirectX and CPU backends.

2.4 CUDA

CUDA is Nvidia's GPU programming framework. In Version 3.0, it comprises a compiler, a debugger, and a profiler for GPU programs.

The model of CUDA follows the structure given by the hardware. The program is divided into a host part and a multi-threaded device part. Different address spaces for the different kinds of memory on the device are provided. A fixed number of threads forms a thread block. For each kernel function that is called from the host the number of threads to be executed and the size of the thread blocks has to be given.

All threads of one block are executed concurrently on one multiprocessor. The execution order of the thread blocks is defined by the framework. As communication is only possible between stream processors within one multiprocessor, no functionality for communication between different blocks is provided. Likewise, synchronisation is only possible between threads of one thread block. Not affected by this constraint is the use of the global memory, for instance for reduction variables. For the synchronisation of memory accesses, atomic functions such as *atomicAdd()* or *atomicExch()* are provided.

2.5 OpenCL

OpenCL is a manufacturer-independent standard for cross-platform parallel programming. Although developed primarily for the programming of graphics processors, it is also suitable for executing parallel programs on CPUs.

The main characteristics of OpenCL are similar to CUDA. An important difference is that the device functions are only compiled at run time. In this way, the OpenCL driver can fine-tune the compilation to the target platform while preserving platform independence. Nevertheless, it is still possible to load precompiled device functions.

The generality of OpenCL that comes along with its platform independence might lead to a performance penalty. Danalis et al. have developed a GPU benchmark [7] and investigated this issue. It turned out that "artificial" kernels that measure low-level hardware characteristics (such as device memory bandwidth or peak performance) have an almost identical runtime in OpenCL and CUDA implementations. In contrast, non-trivial kernels (such as FMM, sgemm matrix multiplication or sorting) are less efficient when written in OpenCL. This indicates that the OpenCL compilers need to be further improved.

3 Physical Simulations on GPUs

The specified peak performance of up to a few TFLOPS for a current GPU hardware is clearly in favour of GPUs as general computing devices, especially when compared to the peak performance of a few tens of GFLOPS for current desktop or server CPU hardware. It is these two orders of magnitude in computing power, combined with a three to four

times larger memory bandwidth that make it worthwhile for scientists to consider GPUs as a viable alternative to existing computing solutions.

The following subsections will discuss some of the many kinds of physical simulations that can benefit from GPUs particularly well. But, not all types of applications may benefit equally well as the ratio of computing power compared to random access memory bandwidth (for large data sets) is worse by at least an order of magnitude. For that reason, applications that are bound by memory bandwidth require even more careful implementation on general purpose GPUs than on high performance desktop and server CPUs.

The methods used in computational science can be classified according to many attributes, but here we will mainly discuss stochastic vs. deterministic and continuous vs. discrete methods.

3.1 Stochastic Methods

Stochastic methods calculate statistic properties of either a large random ensemble of items or samples along some kind of trajectory of a system to calculate approximations to quantities of interest. The laws of statistics imply that, given a proper random sampling of the states of the system of interest, precision of the approximated quantities improves with the number of random samples that make up the ensemble or trajectory. That is, the uncertainty about the exact value of the quantities of interest decreases with larger number of samples considered.

Thus, for the ensemble-based methods, a useful implementation strategy is to let each processing element of a GPU handle evolution and calculation of the properties of a single item of the ensemble and then use appropriate reduction methods to calculate ensemble averages of interest.

For methods that need to follow a trajectory, such as Monte Carlo methods, efficient use of the parallel architecture of GPUs is not a simple problem, as the next point on the trajectory is usually calculated from the current state. This introduces a dependence between the successive states which is best handled by a single processing element.

However, Monte Carlo methods have found widespread use to approximate values of integrals or to provide samples of a system that represent a series of typical states of such a system in contact with some heat bath of a given temperature. For such applications, the main computing power has to be invested in evaluation of some kind of energy function defined on the current 'state' of the system.

An example from physics is the Ising model of magnetic materials. The model uses a 2D or 3D lattice of "spins" which may have two orientations (up or down) indicating the orientation of the magnetic moment of atoms in a material. The Ising model specifies a particular model (the Hamiltonian or energy function) that describes the interaction of (neighbouring or adajacent) spins. In each Monte Carlo step, the orientation of some spins is altered according to some rule (the move-class) and the energy function has to be evaluated, which can be the most time-consuming part of the simulation if spins interact non-locally. Then, the new state is accepted as the next current state with a specified acceptance probability, depending on the change in energy between the current and new

state considered.

Tomov et al. implemented the Ising model on graphics processors [24] using OpenGL. Preis et al. presented a CUDA implementation for the 2D and the 3D Ising model [18]. In order to synchronize the updates, both used a checkerboard algorithm to partition the system into non-interacting domains. During the first half-step, all "white" sites of the checkerboard update their spins, during the second half-step all "black" sites do the update.

Martinsen et al. presented a method for simulating photon transport including scattering and absorption in turbid media [15]. Their main issue was to prevent race conditions that arise when different absorption events occur simultaneously. With using separate memory places in the global memory, their GPU implementation was about 70 times faster than the CPU implementation. Molnár Jr. et al. presented a simulation which models the flow of pollution particles in the air by advection and turbulent diffusion [12].

Meredith et al. [16] evaluated implementing different functions of a quantum Monte Carlo application on GPUs. They also studied whether using single precision floating point arithmetic has an influence on the accuracy of the result. They found that for a number of real science problems, single precision arithmetic is sufficient.

3.2 Deterministic Methods

In contrast to stochastic methods, deterministic methods try to calculate the evolution of a system (e.g. a system of N particles) from deterministic equations that constitute the laws of motion for that system.

Depending on the particular problem, some form of either time stepping or determination of a stationary state is performed. Given a current state, explicit time-stepping methods need to calculate forces that will change this state and then perform an integration step to determine the state at a certain time shortly after the current time. Implicit methods attempt to solve a system of equations derived by discretisation of the equations of motion. The state at some later time is calculated as the solution to a set of 'constraint' equations imposed by current state and forces. Though usually more difficult, this may give advantages in numerical stability and overall time to solution, as e.g. considerably larger and thereby less total time steps may be used.

The main computational load is typically due to a large number of particles considered (e.g. molecular dynamics simulations) and/or processing of a large dataset that describes the current state of the system. Therefore, efficient use of GPUs for these kind of simulations (e.g. fluid dynamics simulations or many-particle problems) poses a non-trivial problem, too.

As an example, we discuss the issues that arise with molecular dynamics simulations. Applications of these methods range from life sciences (e.g. protein folding) to material sciences (e.g. self-assembly of nano particles). Molecular dynamics simulations calculate the time evolution of a large number N of particles. To do so, one iteratively determines the interaction between the particles and then integrates the equations of motion for each particle. These are simple first order differential equations where each processing element may perform integration independently, provided the force terms are known.

Calculation of the forces is, however, the crucial step. This is mainly due to the large number of interactions that need to be considered. In the most exact (quantum mechanical) modelling, the interaction between two particles may depend on the state of all other particles. A considerable reduction in complexity can often be achieved if only pairwise interaction need to be considered. For *direct methods*, interactions between all pairs of particles are evaluated, resulting in an $\mathcal{O}(N^2)$ effort for evaluation of the forces in each time step.

Fast methods apply several strategies to reduce that effort. The first step is to split the interaction into a short-range and a long-range part where the long-range part is either neglected (cut-off) or handled semi-analytically (e.g. by Ewald summation). The second step is, for each particle, to evaluate only the interaction with particles in the immediate proximity. If the particles close to a particular particle considered can be determined with constant effort (e.g. due to a sorted tree structure), the overall time complexity will be considerably lower than $\mathcal{O}(N^2)$.

Hamada and Iitaka proposed a method for the direct calculation of interactions on GPUs [9] that inspired many other authors: the *Camomile Scheme*. The interactions between the particles are calculated block-wise with adapting the block size to the size of the shared memory of the GPU (cf. Fig. 2). The block-wise calculation also brings the possibility to calculate interactions between a larger amount of particles than the amount that can be held in the GPU memory. Belleman et al. [4] and Schive et al. [19] presented further applications and improvements of the Camomile Scheme.

A GPU implementation of the fast multipole method was presented by Gumerov et al. [8]. It runs completely on the GPU and achieves a speedup factor of 72 compared to a serial CPU implementation. Yotoka et al. presented an algorithm that is also fully implemented on the GPU [26], and can also be executed on a cluster of GPUs. Further, they show that, on GPUs, the pseudo-particle method [1, 14] performs similarly to the fast multipole method, being an alternative for fast particle simulations.

Stone et al. [20] extended the popular, highly-tuned parallel molecular dynamics program NAMD by GPU-based non-bonded force calculation. Hardware interpolation of textures is used to evaluate distance-dependend functions in this implementation. All atoms are sorted into bins so that only atoms in neighbouring bins have to be loaded to the GPU. The atoms in all other bins are outside the cut-off radius and interactions with them can be neglected. They achieved a speedup factor of about 5.

Van Meel et al. [25] and Anderson et al. [2] simultaneously published a method for implementing molecular dynamics entirely on GPUs. Both use cell lists to determine which cells are inside the cut-off radius of a certain particle and hence do interact with it and which cells are beyond the cut-off radius. They differentiate in how the cell lists are organized. Van Meel et al. use an “array of place holders” where a virtual particle is placed at each position on which no real particle is situated. If a real particle moves to a virtual particle’s position, the latter is removed. Anderson et al., however, use space-filling curves in order to preserve the space locality of the particles in memory.

Sunarso et al. [21] presented a method for the study of macroscopic flows of liquid crystal molecules. To their knowledge, this was the first work on molecular dynamics simulation to deal with non-spherical particle systems. They use the cell-list method

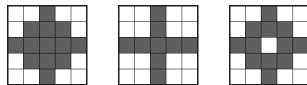

Figure 3: Example set of three generators of size 5×5 consisting of accessible (black) or inaccessible (white) sites.

to ensure that interactions with particles outside the cut-off radius are not calculated. As the cells move due to the macroscopic flow, a special method for the generation of cell lists had to be found. Furthermore, they discussed the best trade-off between use of memory and use of stream processors.

An important point of all GPU implementations of molecular dynamics is to find an efficient method for handling the irregular computations of short-range interactions, e.g. due to the cut-off radius. A second point is to maximize GPU usage. Both, the number of computing cores and the amount of memory can be a bottleneck. So, a good trade-off has to be found. Furthermore, it has to be decided how much of the calculation is to be performed by the GPU. While in the early days, it was common to only perform the computationally intensive force calculations on the GPU, nowadays often the whole simulation is carried out on the graphics device.

4 Simulating Anomalous Diffusion

In this section, a simulation of anomalous diffusion of liquids in porous materials and its implementation on a GPU is presented. Compared to normal diffusion, anomalous diffusion moves slower as the liquid is being obstructed by different structures on all length scales. In the real world, there are numerous processes where anomalous diffusion can be observed. Examples are the diffusion of oil in porous materials or of water in sediments.

4.1 Random Walks on Fractal Structures

As shown in [22], *Random Sierpiński Carpets* model the structure of porous media with the demand of obstacles on several length-scales quite well. Random walks on fractal structures like Random Sierpiński carpets can be used to simulate anomalous diffusion. For performing the simulation, the Random Sierpiński carpet is represented by a two-dimensional Cartesian grid that consists of accessible and inaccessible sites. A random walk starts with a random walker that is located at an initial starting site. To perform one step, the walker selects one of the four directions at random and moves to the adjacent site in the chosen direction. By performing contiguous steps, the walker moves across the grid and performs the random walk.

To create a Random Sierpiński carpet, a set of *generators* is needed. Figure 3 shows an example set with three different generators of size 5×5. Each generator consists of sites which can be accessible (black) or inaccessible (white). The generators are used to construct so-called random *iterators*. The recursive construction of an iterator starts with

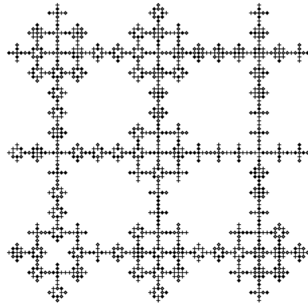

Figure 4: Final Random Sierpiński Carpet of 3×3 iterators of level 3.

one generator selected at random and then replacing the accessible sites by randomly chosen generators in the following construction steps. The Random Sierpiński carpet used for the simulation consists of multiple random iterators (of equal size) that are put together to a large surface. Figure 4 shows an example surface created from 3×3 random iterators.

The random walk simulation uses the following master equation to calculate the probability distribution for the location of the walker on the surface:

$$p_{x,y}^{(s)} = \frac{1}{4}\left(p_{x-1,y}^{(s-1)} + p_{x+1,y}^{(s-1)} + p_{x,y-1}^{(s-1)} + p_{x,y+1}^{(s-1)} + (4-n_{x,y})\, p_{x,y}^{(s-1)}\right) \quad . \tag{1}$$

The random walker steps in each of the four directions with a probability of $\frac{1}{4}$. The probability $p_{x,y}^{(s)}$ for the walker being located at site (x, y) after step s is calculated from the probabilities of the four adjacent sites in the preceding step $s-1$. Due to the irregular structure of the surface, not all four adjacent sites need to be accessible. This results in a certain probability for the walker to rest on the current site, depending on the number of accessible neighbours $n_{x,y}$. The simulation starts with the walker being located at the origin with a probability of one, all other probabilities are zero:

$$p_{0,0}^{(0)} = 1 \ \ , \ \ p_{x,y}^{(0)} = 0 \ \text{ for } \ (x,y) \neq (0,0) \ \ . \tag{2}$$

After s steps, the probability distribution describes the probabilities of all paths of length s. The probability of inaccessible sites is defined as zero for all steps s.

4.2 GPU Implementation

In [10] we have presented different GPU implementations for the simulation of random walks on fractal structures via the master equation approach. Due to the irregular structure of the Random Sierpiński Carpet, the random walk cannot be implemented straightforward on GPUs as they are designed for very regular computations. However, neglecting the irregular fractal structures and treating them as simplified regular ones introduces a large amount of computational overhead. In the following, we describe an improved implementation for GPUs that exploits the irregular structure of the Random

```
__global__ void calculate(float* pold, float* pnew,
                          int4* neighbours)
{
  int i = getThreadId();

  float pold_i = pold[i];
  float pnew_i = 4 * pold_i;

  if (neighbours[i].w != INACCESSIBLE)
    pnew_i += pold[neighbours[i].w] - pold_i;
  if (neighbours[i].x != INACCESSIBLE)
    pnew_i += pold[neighbours[i].x] - pold_i;
  if (neighbours[i].y != INACCESSIBLE)
    pnew_i += pold[neighbours[i].y] - pold_i;
  if (neighbours[i].z != INACCESSIBLE)
    pnew_i += pold[neighbours[i].z] - pold_i;

  pnew[i] = pnew_i * 0.25;
}
```

Figure 5: CUDA kernel function for updating the probability of a single site according to the master equation.

Sierpiński Carpets to minimize the total amount of computations that are necessary for the random walk simulation.

The probability distribution of the previous and the current simulation step are stored in two one-dimensional arrays. The probabilities of all inaccessible sites are omitted, because they are zero in every simulation step. The structure of the fractal surface is stored in a *neighbour* array, that contains for each site the indices of its four adjacent sites. A special index value marks inaccessible neighbours. Creating the Random Sierpiński Carpet as well as preparing the neighbour array and the initial probability distribution is performed by the CPU in the host part of the program. Before the start of the simulation, these arrays are copied to the global GPU memory so that they can be accessed by the device part of the program that is executed on the GPU.

In each simulation step, the probabilities of all accessible sites that are currently in range of the walker are updated. The new probabilities of the current simulation step are calculated according to the master equation (Eq. (1)) using the old probabilities from the previous simulation step. This update is performed by the GPU using a kernel function that is executed by a separate thread for every site. The kernel function uses pointers to the neighbour array and to the arrays with the old and new probabilities as input parameters. When the update of the current simulation step is finished, the next simulation step is performed using the same kernel function, but switching the arrays with the old and new probabilities.

The outline of the kernel function is as follows: First, each thread determines its thread ID to select a site that needs to be updated. Then, it is checked whether each

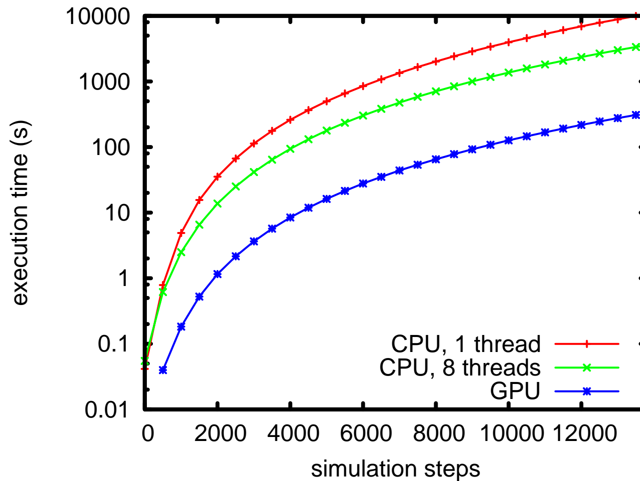

Figure 6: Execution times for a parallel CPU implementation (1 and 8 threads) in comparison to the GPU implementation using the generator set of Fig. 3.

neighbour site is accessible or not. If a neighbour site is accessible, then its old probability value is used to compute the new probability value of the selected site. Finally, the new probability value is stored in the array that is used to store the new probability distribution of the current simulation step. Figure 5 shows the corresponding CUDA kernel function. The neighbour array uses the CUDA specific data structure `int4` to benefit from coalesced memory accesses.

4.3 Results

The GPU implementation was compared to a multi-threaded CPU version of the random walk simulation based on task pools [11]. Updating the sites of the iterators is represented by tasks that stored in a task pool. In each simulation step, a number of threads is used to execute the tasks in parallel. The simulation surface is dynamically extended by adding new iterators if the probabilities at the borders of the surface become greater than zero.

The GPU version was run on a Nvidia GeForce 8800 GT video card with 1 GiB of RAM. Here, only the execution times for updating the probabilities with the GPU program are shown. The CPU version was run on an eight-core (4×dual-core) AMD Opteron 865 (1.8 GHz) machine with 4 GiB of RAM.

Figure 6 shows execution times of the GPU implementation and of the task pool CPU implementation with a single thread and with 8 threads. The GPU implementation achieves a speedup of 32 in comparison to the single-threaded results of the CPU implementation. In comparison to the multi-threaded results with 8 threads, still a performance increases of about a factor of 11 is achieved.

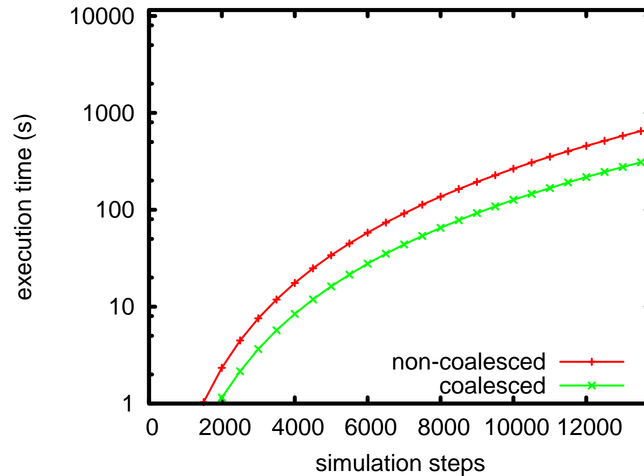

Figure 7: Execution times for GPU implementations with and without using coalesced memory access (using the `int4` and a user-defined record).

A reasonable performance increase was achieved by using the CUDA built-in `int4` data structure for the neighbour array to benefit from coalesced memory accesses. Figure 7 shows the execution times with and without coalescing. The optimized memory accesses lead to a performance increase by a factor of 2. However, extending the usage of coalesced memory accesses to the arrays that store the old and the new probability distribution has not lead to significant further improvements.

5 Summary

GPUs, nowadays, are used for all kinds of applications. Also, in scientific computing and for physical simulations, they are employed and often lead to a significant speedup. Despite their architecture for very regular computations, also irregular problems such as the simulation of diffusion on fractal structures presented in this article or the fast multipole method can be implemented on GPUs. They run efficiently on GPUs and perform better than on single-core or even multi-core CPUs. Special attention has to be given to the memory access. Coalesced memory access should be used wherever possible as this leads to a huge performance increase.

The most popular programming framework seems to be CUDA, probably because of its user-friendliness and its early presence. CAL, however, is not encouraged to be used by non-experienced programmers and, hence, not widely used. Instead, AMD proposes its Brook+ framework. Though, it appears not to receive the attention by the developer community it might deserve. In the near future, OpenCL might be the solution which receives more and more attention. Due to its platform and manufacturer independence it is a good solution for portable general purpose GPU programming.

References

[1] C. Anderson. An implementation of the fast multipole method without multipoles. *SIAM J Sci Stat Comput*, 13(4):923–947, 1992.

[2] J. Anderson, C. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J Comput Phys*, 227(10):5342–5359, 2008.

[3] ATI. *ATI Stream Computing. User Guide.*

[4] R. G. Belleman, J. Bédorf, and S. F. Portegies Zwart. High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA. *New Astron*, 13(2):103–112, 2008.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, page 786. ACM, 2004.

[6] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck. CryptoGraphics: Secret key cryptography using graphics cards. In *Proc. of the RSA Conf., Cryptographer's Track (CT-RSA)*, pages 334–350. Springer, 2005.

[7] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, Ph. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU '10: Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[8] N. A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *J Comput Phys*, 227(18):8290–8313, 2008.

[9] T. Hamada and T. Iitaka. The chamomile scheme: An optimized algorithm for n-body simulations on programmable graphics processing units. arXiv ePrint: astro-ph/0703100v1, 2007.

[10] K. H. Hoffmann, M. Hofmann, J. Lang, G. Rünger, and S. Seeger. Simulating Anomalous Diffusion on Graphics Processing Units. In *Proc. of the 11th IEEE Int. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-10)*. IEEE, 2010.

[11] K. H. Hoffmann, M. Hofmann, G. Rünger, and S. Seeger. Task Pool Teams Implementation of the Master Equation Approach for Random Sierpinski Carpets. In *Proc. of the 12th Int. Euro-Par Conf.*, volume 4128 of *LNCS*, pages 1043–1052. Springer, 2006.

[12] F. Molnár Jr., T. Szakály, R. Mészáros, and I. Lagzi. Air pollution modelling using a graphics processing unit with cuda. *Comput Phys Comm*, 181(1):105–112, 2010.

[13] J. Kurzak and J. Dongarra. Implementation of mixed precision in solving systems of linear equations on the cell processor: Research articles. *Concurr. Comput. : Pract. Exper.*, 19:1371–1385, 2007.

[14] J. Makino. Yet another fast multipole method without multipoles–pseudoparticle multipole method. *J Comput Phys*, 151(2):910–920, 1999.

[15] P. Martinsen, J. Blaschke, R. Künnemeyer, and R. Jordan. Accelerating Monte Carlo simulations with an NVIDIA graphics processor. *Comput Phys Comm*, 180(10):1983–1989, 2009.

[16] J. S. Meredith, G. Alvarez, T. A. Maier, T. C. Schulthess, and J. S. Vetter. Accuracy and performance of graphics processors: A quantum monte carlo application case study. *Parallel Comput*, 35(3):151–163, 2009.

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

[18] T. Preis, P. Virnau, W. Paul, and J. J. Schneider. GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. *J Comput Phys*, 228(12):4468–4477, 2009.

[19] H.-Y. Schive, C.-H. Chien, S.-K. Wong, Y.-C. Tsai, and T. Chiueh. Graphic-card cluster for astrophysics (gracca) – performance tests. *New Astron*, 13(6):418–435, 2008.

[20] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *J Comput Chem*, 28(16):2618–2640, 2007.

[21] A. Sunarso, T. Tsuji, and S. Chono. GPU-accelerated molecular dynamics simulation for study of liquid crystalline flows. *J Comput Phys*, 229(15):5486–5497, 2010.

[22] S. Tarafdar, A. Franz, Ch. Schulzky, and K. H. Hoffmann. Modelling porous structures by repeated Sierpinski carpets. *Physica A*, 292(1-4):1–8, 2001.

[23] C. J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proc. of the 35th Annual ACM/IEEE Int. Symp. on Microarchitecture*, 2002.

[24] S. Tomov, M. McGuigan, R. Bennett, G. Smith, and J. Spiletic. Benchmarking and implementation of probability-based simulations on programmable graphics cards. *Comput Graph*, 29(1):71–80, 2005.

[25] J. A. van Meel, A. Arnold, D. Frenkel, S. F. Portegies Zwart, and R. G. Belleman. Harvesting graphics power for md simulations. *Mol Simulat*, 34(3):259–266, 2008.

[26] R. Yokota, T. Narumi, R. Sakamaki, S. Kameoka, S. Obi, and K. Yasuoka. Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence. *Comput Phys Comm*, 180(11):2066–2078, 2009.