Simulating Anomalous Diffusion on Graphics Processing Units

Karl Heinz Hoffmann*, Michael Hofmann‡1, Jens Lang‡1, Gudula Rünger‡ and Steffen Seeger*1
* *Department of Physics, Chemnitz University of Technology, Germany*
Email: {s.seeger,hoffmann}@physik.tu-chemnitz.de
‡ *Department of Computer Science, Chemnitz University of Technology, Germany*
Email: {mhofma,lajen,ruenger}@cs.tu-chemnitz.de

***Abstract*—The computational power of modern graphics processing units (GPUs) has become an interesting alternative in high performance computing. The specialized hardware of GPUs delivers a high degree of parallelism and performance. Various applications in scientific computing have been implemented such that computationally intensive parts are executed on GPUs. In this article, we present a GPU implementation of an application for the simulation of diffusion processes using random fractal structures. It is shown how the irregular computational structure that is inherent to the application can be implemented efficiently in the regular computing environment of a GPU. Performance results are shown to demonstrate the benefits of the chosen implementation approaches.**

***Keywords*-GPU programming, CUDA, irregular algorithms, random walk simulation, fractal structures.**

I. Introduction

The use of graphics processing units (GPUs) to speed up computations in scientific computing has received an increasing popularity in recent months and years [1]. Different algorithms and applications from various scientific areas have been implemented to run on general purpose GPUs (GPGPUs). GPGPUs extend the design and functionality of traditional GPU hardware and make the high performance graphics capabilities available for non-graphics (general purpose) computations. The architecture of a GPU aims at very regular algorithms. The efficient implementation of irregular algorithms on GPUs is an ongoing research effort.

The special feature of GPUs in comparison with CPUs is their massive parallelism resulting from a large number of processor cores. While recent multicore CPUs have only a few tens of cores, modern GPUs usually have hundreds of cores. Exploiting the high computing capabilities of GPUs requires a parallelisation of algorithms and applications such that all execution units of the GPU will be utilized equally. Major disadvantages of GPUs are the limited memory on video cards and the SIMD architecture of its parallel execution units. Since the SIMD architecture is designed for regular computations, it can be hard to achieve a comparably good performance for irregular computations.

First attempts to GPU computing have used available graphics APIs like OpenGL [2] or DirectX [3] to utilize accelerating graphics hardware. Nowadays, special programming frameworks are used, like CUDA [4] for NVIDIA GPUs or ATI Stream [5] for AMD/ATI GPUs. Another recent development is OpenCL [6], which leads to a platform- and architecture-independent approach to GPU and CPU programming.

This article presents a GPU implementation for the simulation of diffusion processes in porous materials. The application performs a random walk simulation on fractal structures, so-called random Sierpiński carpets, using a master equation approach [7]. The random structure of the simulation surface leads to irregular computations. We present three different variants to implement the simulation application on a GPU using the CUDA framework. The first variant is a regular implementation that fits into GPU programming by neglecting the irregular structure of the application. The other two variants represent improved implementations adapting to the irregular computations of the application while maintaining a computational structure that is appropriate for the specialized GPU hardware. The performance of the different variants is demonstrated with performance results using an NVIDIA GeForce 8800 GT video card and different sets of input data.

The rest of this article is organized as follows. Section II presents related work. Section III introduces the algorithm of the simulation application. Section IV gives a short overview of the features of the GPU hardware and programming. In Sect. V, the three different implementation variants for GPUs are described. Performance results are presented in Sect. VI and Sect.VII concludes the article.

II. Related work

An overview of GPU programming and especially the CUDA framework is given in [8] and [9]. Irregular problems in computational physics have been implemented for GPUs and their results were compared to CPU programs. In [10], Gumerov and Duraiswami presented an implementation of the Fast Multipole Method, an algorithm for the calculation of forces between interacting particles, on graphics processors. The runtime was decreased by a factor of 30–70 compared to a sequential CPU version. Stock and Gharakhani showed in [11] that for another algorithm for the solution of the n-body problem the performance can be improved up to

[1]Supported by Bundesministerium für Bildung und Forschung (BMBF) and Deutsche Forschungsgemeinschaft (DFG).

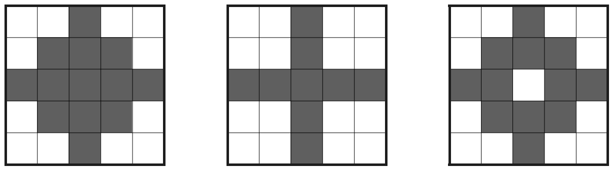

Figure 1. Set of three different generators of size 5×5 consisting of accessible (black) and inaccessible (white) sites.

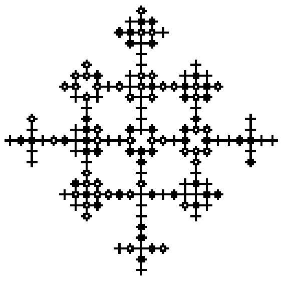

Figure 2. Example iterator of level 3, randomly constructed using the three generators shown in Fig. 1.

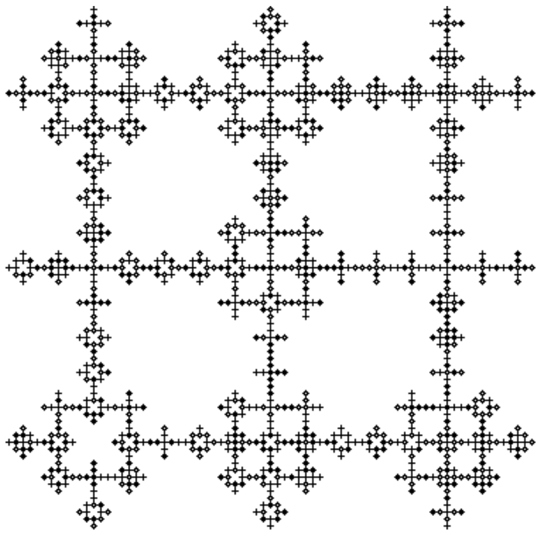

Figure 3. Example of a random Sierpiński carpet consisting of 3×3 random iterators of level 3 that can be used as simulation surface for the random walk.

factor 17 in comparison to a CPU version. In [12] a Monte Carlo simulation of the Ising model was presented that is used to examine the phase transition in ferromagnetic and antiferromagnetic materials. The GPU accelerated version was implemented to work on two- or three-dimensional lattice grids and achieved an increase in performance of up to a factor of 60.

III. Random Walks on Sierpiński Carpets

The subject of the simulation being presented in this article is the anomalous diffusion of particles in porous materials, e. g. the diffusion of pollutants in wetted porous rock or sediment. The porous material is represented by a random Sierpiński carpet [13]. Random walks on fractal structures like random Sierpiński carpets can be used to simulate anomalous diffusion [14].

A random walk on a two-dimensional Cartesian grid proceeds as follows. The random walker starts at some starting point (e. g. the site at the co-ordinate origin). 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. Of special interest is the time dependence of the mean square displacement $\langle r^2(t)\rangle$, where $r(t)$ is the distance of the walker from the starting point after time t. For anomalous diffusion $\langle r^2(t)\rangle$ is not linear but satisfies a power law [15], i. e. $\langle r^2(t)\rangle \sim t^\gamma$ with $0 < \gamma \leq 1$ being the diffusion exponent.

To create a random Sierpiński carpet, a set of *generators* is needed. Figure 1 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* which represent the basic elements of our random Sierpiński carpet. The recursive construction of an iterator starts with one generator selected at random in the first step. In the following construction steps each accessible site is replaced by a new randomly chosen generator. The number of construction steps used to create an iterator denotes the level of an iterator. Figure 2 shows an example for a random iterator of level 3 with a total number of $5^3 \times 5^3$ sites.

The random Sierpiński carpet used for the simulation consists of multiple random iterators (of constant level) that are put together to a large surface. Figure 3 shows an example surface created from 3×3 random iterators. A random Sierpiński carpet created in that way can be used to imitate the macroscopic homogeneity and the microscopic heterogeneity of porous materials like rock, aerogels or cements [13]. It consists of accessible and inaccessible sites and is used as simulation surface for the random walk. The random construction of the iterators results in irregular structures that need special care for the programming.

The random walk simulation uses the master equation approach [16] to calculate the probability distribution for the location of the walker on the surface. The master equation

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

can be used to calculate the probability $p_{x,y}^{(s)}$ for a walker being located at site (x, y) after a walk of s steps with $n_{x,y}$ being the number of accessible neighbour sites.

The random walker steps in each of the four directions with a probability of $\frac{1}{4}$. Therefore, the probability of one site at step s is calculated from the probabilities of its 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 $n_{x,y}$ of this site.

Before the first step, the walker is located at the origin with a probability of one, all other probabilities are zero, i. e.

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

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.

After the construction of the random Sierpiński carpet, the simulation of the random walk is performed as follows. First, an array P_{old} is created that is used to store the probabilities of all sites. The initial values of all sites are set to zero and the site at the starting point of the random walk receives a probability of one. Then, the simulation steps are performed. In each simulation step, the path length s is increased by one. For each accessible site, the probability of this site and its neighbours is read from the array P_{old} and used in Eq. (1) to compute the probability for the current step. The new probabilities are stored in an auxiliary array P_{new}. After the probabilities of all sites are calculated, the two arrays switch their roles and the next simulation step proceeds. When the simulation has finished after a certain number of steps, the result (e. g. the mean square displacement of the random walker) can be extracted from the final probability distribution. This results in the following pseudo code:

```
1:  $S$ = maximum number of steps
2:  create the random Sierpiński carpet
3:  create the initial distribution in $P_{old}$
4:  for $s = 1$ to $S$ do
5:      for each site $(x,y)$ do
6:          if $(x,y)$ is accessible then
7:              update $P_{new}[x,y]$ according to Eq. (1)
8:          end if
9:      end for
10:     switch roles of $P_{new}$ and $P_{old}$
11: end for
12: retrieve results from $P_{old}$
```

IV. GPU Programming Specifics

The architecture of GPUs as well as the GPU programming framework can require special adaptations of the algorithms and applications. In the following, we describe the main properties of GPU programming as they arise with the CUDA framework.

A modern GPU (see Fig. 4) consists of several unified shader units, called *multiprocessors* in CUDA terminology. Each multiprocessor consists of a number of stream processors. All multiprocessors have read/write access to the video memory and read access to the texture and constant memory. The shared memory can be accessed only by the

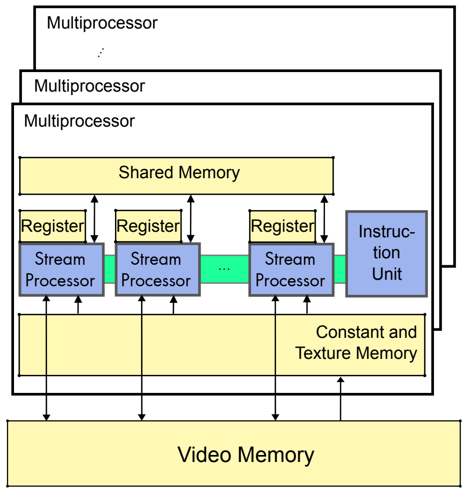

Figure 4. Illustration of the hardware architecture of a GPU device (adapted from [4]).

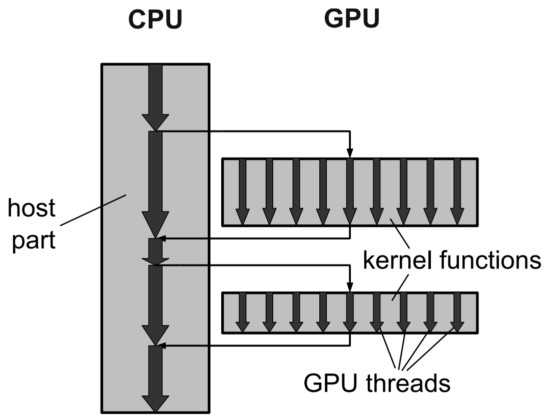

Figure 5. Program flow for a GPU program: The main part is executed on the CPU (host), the computational part is executed on the GPU (device).

stream processors of one multiprocessor. The registers are dedicated to the single stream processors.

In GPU programming with CUDA, the application program is split into a host part, which is executed on the CPU, and a device part, which is executed on the GPU. A typical program flow is shown in Fig. 5. The device part consists of *kernel functions* that are called in the host part of the program, but are executed on the GPU by multiple threads in parallel. All data that is needed by kernel functions has to be transferred from the main memory to the video memory beforehand. A call to a kernel function returns immediately so that the host part can continue while the kernel function is executed by the GPU. A barrier function is available to wait for the completion of the GPU computations. Kernel functions write the results of their computations to the video memory from where they can be transferred back to the main memory afterwards.

When calling a kernel function from the host part, it is necessary to specify the required number of *thread blocks* and the number of threads per block. Each thread executes the kernel function. All threads of one block are executed by one multiprocessor in parallel according to the SIMD principle. The shared memory and a barrier operation can be used for communication and synchronisation between threads within one block. The processing of the blocks is scheduled by the graphics processor.

The blocks are organized in a three-dimensional grid and the threads of one block are organized in a two-dimensional grid. A kernel function can access the block ID and the thread ID of the thread that executes the function. These IDs can be used to distinguish the different instances of the kernel function, e.g., to assign them different parts of the data that is processed. With these IDs, it is also possible to calculate a global thread ID that is unique over all threads.

Due to the SIMD characteristics of the GPU hardware, branches should be avoided inside kernel functions. Even if a branch is taken only by a few threads of one block, the instructions of both paths of the branch have to be executed by all threads of the block. The instructions of a path that is not taken according to a previous branch are executed, but their results are discarded (i. e. these instructions have no effect for the particular thread).

V. GPU Implementation of the Random Walk Simulation

In the following, we present three different implementation variants for performing the random walk simulation on random Sierpiński carpets on a GPU. Variant A represents a first straightforward implementation neglecting the irregular structure of the computations. Variants B and C represent gradual improvements that take into account the structure of the irregular application and the specifics of GPU programming.

A. Implementation Variant A

The first implementation variant ignores the irregular structure of the simulation surface and treats them as a fully occupied regular data structure. The host part and the kernel function of Variant A are shown in Fig. 6. The construction step of the random Sierpiński carpet as described in Sect. III is performed at the beginning in the host part of the program. Inaccessible sites are marked with a negative probability value in the initial distribution. Two arrays for storing the probabilities are allocated as two-dimensional arrays in the video memory and the initial probability distribution is copied from the main memory to the video memory.

In each simulation step, a kernel function is called by the host program so that a separate thread is started to update each site of the random Sierpiński carpet. The block and thread IDs are used to determine the x and y co-ordinates of the site that the thread is assigned to. These co-ordinates are used to access the probability values in the two-dimensional arrays directly. A negative value for the probability of a site shows that this site is inaccessible and leads to an immediate termination of the corresponding thread. Otherwise the new probability for the site is calculated from Eq. (1) and stored in the auxiliary array P_{new}.

Figure 6. Pseudo code for implementation Variant A.

```
 1: // host part
 2: $S$ = maximum number of steps
 3: create random Sierpiński carpet
 4: allocate $P_0[-S \dots S, -S \dots S]$
 5: allocate $P_{old}[-S \dots S, -S \dots S]$ (in video memory)
 6: allocate $P_{new}[-S \dots S, -S \dots S]$ (in video memory)
 7: create initial distribution in $P_0$
 8: copy $P_0$ to $P_{old}$
 9: for $s = 1$ to $S$ do
10:    // kernel function call
11:    for each $i = 1$ to $(2S + 1)^2$ do
12:       call $\mathrm{A}(P_{old}, P_{new})$
13:    end for
14:    wait for completion of the kernel functions
15:    swap $P_{old}$ and $P_{new}$
16: end for
17: copy $P_{old}$ to $P_0$
18: retrieve results from $P_0$
19:
20: // device part
21: kernel function $\mathrm{A}(P_{old}, P_{new})$
22:    determine $(x, y)$ from the block and thread IDs
23:    if $P_{old}[x, y] \geq 0$ then
24:       $p = 0, n = 0$
25:       for $(\hat{x}, \hat{y})$ = $(-1, 0), (1, 0), (0, -1), (0, 1)$ do
26:          if $P_{old}[x + \hat{x}, y + \hat{y}] \geq 0$ then
27:             $p$ += $P_{old}[x + \hat{x}, y + \hat{y}]$
28:             $n$++
29:          end if
30:       end for
31:       $P_{new}[x, y] = (p + (4 - n) \cdot P_{old}[x, y])/4$
32:    end if
33: end kernel function
```

The host program waits for the completion of all threads executing the kernel function. After that, the array with the old probabilities and the auxiliary array with the new probabilities are swapped and the next simulation step proceeds.

As previously mentioned in Sect. IV, branches inside the kernel function (e.g., as in line 26) should be avoided due to the SIMD architecture of the GPU. However, an alternative version of the kernel function which avoids this branch using a computation step that is executed by all threads has led to a slight decrease in performance. In this case, the branch avoidance introduced additional memory accesses that

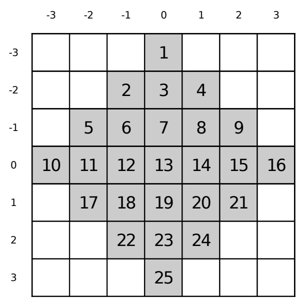

Figure 7. Numbering of sites for implementation Variant B for the assignment of threads to sites in the third simulation step ($s = 3$).

overcompensated the performance benefits from the omitted branches.

B. Implementation Variant B

Variant A starts a separate thread for each site in every simulation step. With the initial probability distribution (see Eq. (2)) the walker is located at the origin and all other sites have a probability of zero. Since the walker is only allowed to move to a neighbouring site in one simulation step, all sites that are far away from the origin keep their probability of zero in the beginning. Only sites that are within the range of the longest currently considered path of the walker can have a non-zero probability and need to be computed.

The area of sites with non-zero probability is diamond-shaped and centred at the origin. The area is growing by one site in each direction in each simulation step. To consider all paths of the length s, only the sites (x, y) with

$$s \leq |x| + |y| \tag{3}$$

have to be taken into account. In implementation Variant B threads are started only for sites that fulfil this inequation.

We use a row-wise numbering of the sites from the top to the bottom in the diamond-shaped area to assign each thread with its global thread ID to one site. Figure 7 shows an example for the assignment of threads to sites in the third simulation step. The x and y co-ordinates of a site are calculated from the global thread ID i as follows:

$$j = \begin{cases} i & \text{if } i \leq (s+1)^2 \ , \\ 2(s^2+s+1) - i & \text{if } i > (s+1)^2 \ , \end{cases} \tag{4}$$

$$k = \begin{cases} -1 & \text{if } i \leq (s+1)^2 \ , \\ 1 & \text{if } i > (s+1)^2 \ , \end{cases} \tag{5}$$

$$x = k\left(\lceil\sqrt{j}\rceil + (\lceil\sqrt{j}\rceil - 1)^2 - j\right) \ , \tag{6}$$

$$y = -k(\lceil\sqrt{j}\rceil - s - 1) \ . \tag{7}$$

The calculation differs for the upper and the lower half of the surface. This results in the case differentiation in the definition of the auxilary variables j and k.

Figure 8. Pseudo code for implementation Variant B.

```
// host part
$S$ = maximum number of steps
create random Sierpiński carpet
allocate $P_0[-S \dots S, -S \dots S]$
allocate $P_{old}[-S \dots S, -S \dots S]$ (in video memory)
allocate $P_{new}[-S \dots S, -S \dots S]$ (in video memory)
create initial distribution in $P_0$
copy $P_0$ to $P_{old}$
for $s = 1$ to $S$ do
  // kernel function call
  for each $i = 1$ to $s^2 + (s+1)^2$ do
    call $\mathrm{B}(P_{old}, P_{new})$
  end for
  wait for completion of the kernel functions
  swap $P_{old}$ and $P_{new}$
end for
copy $P_{old}$ to $P_0$
retrieve results from $P_0$

// device part
kernel function $\mathrm{B}(P_{old}, P_{new})$
  $i$ =global thread ID
  determine $(x, y)$ according to Eq. (4)–(7)
  if $P_{old}[x, y] \geq 0$ then
    $p = 0, n = 0$
    for $(\hat{x}, \hat{y})$ = $(-1, 0), (1, 0), (0, -1), (0, 1)$ do
      if $P_{old}[x + \hat{x}, y + \hat{y}] \geq 0$ then
        $p$ += $P_{old}[x + \hat{x}, y + \hat{y}]$
        $n$++
      end if
    end for
    $P_{new}[x, y] = (p + (4 - n) \cdot P_{old}[x, y])/4$
  end if
end kernel function
```

The total number of threads required in simulation step s corresponds to the number of sites inside the diamond-shaped area and is equal to $s^2 + (s+1)^2$. The execution time of one thread increases by the time required for the calculation of the site co-ordinates according to the equations (4)–(7). Figure 8 shows the modified version of the program.

C. Implementation Variant C

In the implementation Variants A and B, inaccessible sites are recognized by their negative probability values. This causes an immediate termination of the kernel function. Because of the SIMD principle of the GPU, threads of inaccessible sites require the same amount of time as threads of accessible sites. Since the probabilities of inaccessible sites are zero during the whole simulation, the costs for calculating these sites can be avoided.

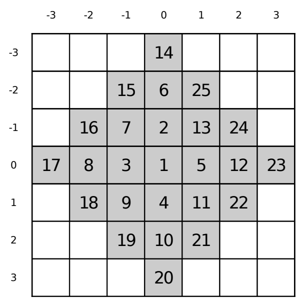

Figure 9. Numbering of sites for implementation Variant C for the assignment of threads to sites in the third simulation step.

Implementation Variant C stores the carpet site-wise in a one-dimensional array omitting all inaccessible sites. The inaccessible sites are randomly located on the surface. If only accessible sites are stored in the array, it is not possible any more to determine the positions of neighbouring sites from the array positions. Therefore, the adjacency relationship between the sites has to be stored separately in an array containing the indices of the sites' neighbours. Inaccessible neighbours are marked by a special index, e. g. NOT_PRESENT. This array is set up during the creation of the random Sierpiński carpet by the CPU and transferred to the video memory at the beginning of the simulation. As the carpet does not change its structure during the simulation, the content of this array is constant.

Similarly to Variant B, it is necessary to choose an appropriate numbering of the (accessible) sites inside the diamond-shaped area to assign them to the threads. The row-wise numbering from the top to the bottom is disadvantageous, because it leads to a renumbering of the sites in every simulation step. This would require an expensive rebuild of the neighbours array that stores the adjacency relationship between the sites. Instead, we use a numbering of the sites that starts in the origin and increases while moving outwards. Figure 9 shows an example for the chosen numbering. With this numbering, all sites have constant indices during the whole simulation. Figure 10 shows the modified version of the program.

The sites are assigned directly to the threads using their thread IDs as indices. A translation between the indices and the co-ordinates of the sites is only necessary during the creation of the carpet. The number of threads required in one simulation step now depends on the number of accessible sites. Due to the irregular structure, it cannot be determined when calling the kernel function. Instead, it is calculated in advance during the creation of the carpet.

Omitting inaccessible sites requires less memory for storing the probabilities values. However, storing the adjacency relationship requires additional memory. The total memory consumption of implementation Variant C depends on the amount of accessible sites. In general, we expect a reduction in memory consumption which makes it possible to store larger carpets in the limited video memory.

Figure 10. Pseudo code for implementation Variant C.

```
// host part
$S$ = maximum number of steps
create random Sierpiński carpet
determine adjacency relationship
$t[1 \dots S]$ = number of threads required in each step
$M$ = total number of accessible sites
allocate $P_0[M]$
allocate $P_{old}[M]$, $P_{new}[M]$ (in video memory)
allocate $neighbours[M, 4]$ (in video memory)
create initial distribution in $P_0$
copy $P_0$ to $P_{old}$
copy adjacency relationship to $neighbours$
for s = 1 to $S$ do
    // kernel function call
    for each $i = 1$ to $t[s]$ do
        call C($P_{old}, P_{new}, neighbours$)
    end for
    wait for completion of the kernel functions
    swap $P_{old}$ and $P_{new}$
end for
copy $P_{old}$ to $P_0$
retrieve results from $P_0$

// device part
kernel function C($P_{old}, P_{new}, neighbours$)
    $i$ = global thread ID
    $p = 0, n = 0$
    for $j = 1$ to 4 do
        if $neighbours[i, j] \neq$ NOT_PRESENT then
            p += $P_{old}[neighbours[i, j]]$
            $n$++
        end if
    end for
    $P_{new}[i] = (p + (4 - n) \cdot P_{old}[i])/4$
end kernel function
```

VI. Performance Results

The generator set shown in Fig. 1 was used to compare the performance of the different implementation variants. The *mean occupancy rate* of a random Sierpiński carpet specifies the amount of accessible sites in relation to the total number of sites. For the three sample generators, the number of accessible sites (out of 25) is 13, 9 and 12, respectively. The iterators were constructed with 3 recursion steps. The resulting mean occupancy rate with these settings is $\left(\frac{13+9+12}{3 \cdot 25}\right)^3 \approx 0.093 \approx 0.1$. This means that approximately one out of ten sites is accessible. Since the number of accessible sites can have an influence on the

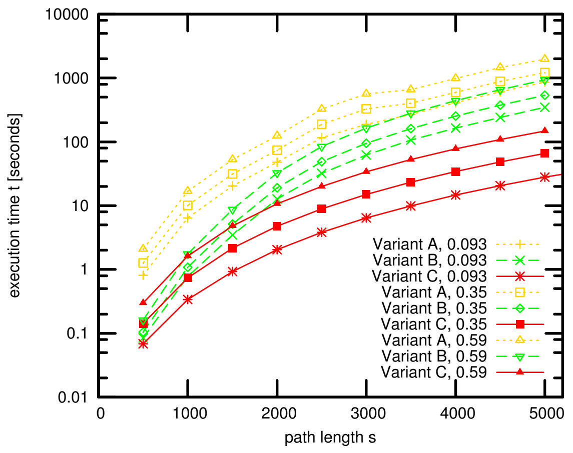

Figure 11. Execution times for implementation Variants A, B and C using three generator sets with different occupancy rates.

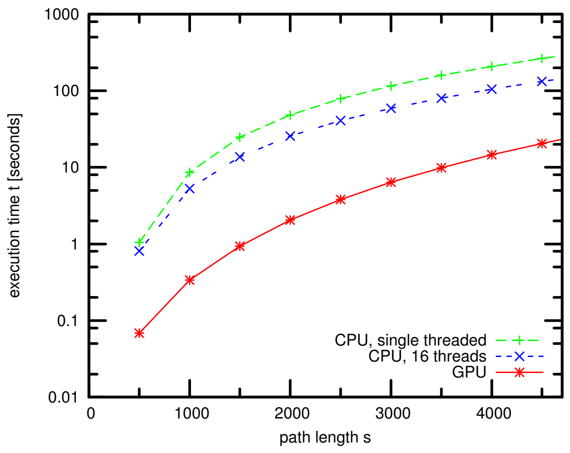

Figure 12. Execution times for a parallel CPU version (1 and 16 threads) in comparison to the GPU version (Variant C) using a generator set with occupancy rate of 0.093.

performance, two additional generator sets with occupancy rates 0.35 and 0.59 were used.

The simulation program was executed on a quad-core Opteron system with an NVIDIA GeForce 8800 GT video card. In Fig. 11, the execution times for the three implementations with the given generator sets are shown as a function of the path length s. Only the execution times of the main computational loop are shown. The additional time required for initialization, copying the data structures to and from the video memory and for the analysis of the data was less than 2 % of the total execution time for larger path lengths s. The initialization in Variant C was about 5–10 times faster than the initializations in Variants A and B. This can be attributed to the higher amount of data that has to be transferred to the video memory in Variants A and B.

The results in Fig. 11 show that the execution time of all variants increases for an increasing path length. Variant B is faster than Variant A for all occupancy rates because the threads for sites that are further away from the origin than the longest currently considered path are not started any more. This saves a great amount of execution time. Variant C was even faster than Variant B. Here, the execution times for threads of inaccessible sites were saved. The occupancy rate of the generator sets also has an influence on the execution time. Figure 11 shows that the execution time decreased significantly with a lower occupancy rate.

For a simulation with s random walker steps, a maximum area of $(2s+1)^2$ sites is needed. Thus, for s steps, no more than $s(2s+1)^2 \in \mathcal{O}(s^3)$ updates of the probabilities of the sites are needed. The upper bound for the execution time is determined by this number of updates. To get a more accurate view of the influence of the occupancy rate on the execution time, the results of the experiments were approximated with the function

$$t(s) = as^3 \qquad (8)$$

Table I

VALUES FOR $a \cdot 10^9$ IN EQ. (8) FROM EXPERIMENTAL RESULTS AND LEAST-SQUARES FIT.

Occupancy Rate	0.093	0.35	0.59
Variant A	6.742	9.71	15.94
Variant B	2.684	4.14	7.24
Variant C	0.21671	0.5310	1.1898

using the least-squares method. The resulting values for a are shown in Table I. The value of a determines the asymptotic behaviour of the execution time. Higher values for a represent a larger increase of the execution time.

The results show that for low occupancy rates the execution time of Variant B was about one third of the execution time of Variant A. Variant C is about an order of magnitude faster than Variant B. The advantage of Variant C in comparison to Variant B decreases from factor 12 to factor 6 when using generators with a higher occupancy rate. The reason for this is that the number of threads that are saved on the transition from Variant B to Variant C decreases with an increasing occupancy rate.

The total size of the video memory limits the maximum size of the simulation surface. The largest number of simulation steps was about 5000 for Variants A and B and about 9500 for Variant C.

In [17], an implementation of this application for SMP clusters is presented that handles the irregular carpet similarly to Variant C. This CPU version uses a coarse-grained approach that operates on entire iterators. The calculation of one simulation step on one iterator represents one task. The tasks are distributed to the threads via a task pool. Figure 12 shows results for the execution time of Variant C on the GPU and for the parallel CPU version with 1 and 16 threads. The CPU version was executed on a multicore system with 16 cores (4 × AMD Opteron 8347 processors, 1.9 GHz). The

GPU version turned out to be much faster. The times indicate a performance increase of about an order of magnitude. This compares to the results in [10] and [12].

The overall results show that for solving problems with irregular, sparse data structures on GPUs, it is necessary to preprocess the input data in a way that allows a regular computation. Avoiding branches inside the GPU program to comply with the SIMD architecture of GPUs does not automatically lead to a performance increase. The application presented in this article is a memory bandwidth bounded code with only a few instructions in the branches of the kernel function. For this kind of application, it can be favourable to retain the branches if they help to prevent unnecessary memory accesses. The programming model of the GPU allows a very fine-grained parallelisation of irregular applications. The light weight threads of the GPU eliminate the need for partitioning the data in a way that there exists a sufficient computational load per thread. While the GPU version of our application assigns every single site to a separate thread, this would cause too much overhead on a CPU. An efficient and balanced partitioning of the data, which can be a major challenge, especially for irregular problems, can be achieved very easily in this case. Additionally, memory bandwidth bounded codes, like the application used in this article, with only few computations per data element, can benefit from the superior memory bandwidth of GPUs in comparison to CPUs.

VII. Summary

In this article, we have presented three implementation variants of an algorithm for random walk simulations via the master equation approach on random Sierpiński carpets. We have shown that such irregular algorithms can be implemented efficiently on GPUs. The original algorithm was adapted with respect to the SIMD architecture of the GPU to achieve a uniform utilization of the parallel execution units. We have exploited the irregular structure of the random Sierpiński carpet to reduce the total amount of computations that are necessary to perform the random walk. The resulting irregular computations were implemented such that they can be executed appropriately on a GPU. The presented performance results demonstrate the benefits of the chosen approaches. The GPU variant of the simulation program achieved an increase in performance of about one order of magnitude in comparison to a single threaded CPU version.

References

[1] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[2] "www.opengl.org."

[3] "www.microsoft.com/windows/directx."

[4] *NVIDIA CUDA – Programming Guide*, version 2.3.1.

[5] *ATI Stream Computing – User Guide*, april 2009.

[6] Khronos OpenCL Working Group, *The OpenCL Specification*, version 1.0.

[7] A. Franz, C. Schulzky, N. A. Do Hoang, S. Seeger, J. Balg, and K. H. Hoffmann, "Random Walks on Fractals," in *Parallel Algorithms and Cluster Computing*. Springer, 2006, vol. 52, pp. 303–313. [Online]. Available: http://www.springerlink.com/content/v3654631p5k268u7

[8] O. Schenk, M. Christen, and H. Burkhart, "Algorithmic performance studies on graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1360–1369, 2008.

[9] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, 2008.

[10] N. A. Gumerov and R. Duraiswami, "Fast multipole methods on graphics processors," *Journal of Computational Physics*, vol. 227, no. 18, pp. 8290–8313, 2008.

[11] M. J. Stock and A. Gharakhani, "Toward efficient GPU-accelerated n-body simulations," in *46th AIAA Aerospace Sciences Meeting and Exhibit*. AIAA, January 2008.

[12] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, "GPU accelerated Monte Carlo simulation of the 2D and 3D ising model," *Journal of Computational Physics*, vol. 228, no. 12, pp. 4468–4477, 2009.

[13] S. Tarafdar, A. Franz, C. Schulzky, and K. H. Hoffmann, "Modelling porous structures by repeated Sierpinski carpets," *Physica A: Statistical Mechanics and its Applications*, vol. 292, no. 1-4, pp. 1–8, 2001.

[14] R. Metzler and J. Klafter, "The random walk's guide to anomalous diffusion: a fractional dynamics approach," *Physics Reports*, vol. 339, no. 1, pp. 1–77, 2000.

[15] K. H. Hoffmann and J. Prehl, "Anomalous Transport on Disordered Fractals," in *Anomalous Transport*, 1st ed., G. Radons, R. Klages, and I. M. Sokolov, Eds. Berlin: Wiley-VCH, July 2008, pp. 397–428.

[16] H. Gould, J. Tobochnik, and W. Christian, *An Introduction to Computer Simulation Methods*, 3rd ed. Addison-Wesley, March 2006.

[17] 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 International Euro-Par Conference*, ser. LNCS, vol. 4128. Springer, 2006, pp. 1043–1052.