Journal of Computational Physics 249 (2013) 67–79

Contents lists available at SciVerse ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

GPU-accelerated Monte Carlo simulation of particle coagulation based on the inverse method

J. Wei, F.E. Kruis [*]

Institute for Nanostructures and Technology, Faculty of Engineering Science, and CENIDE (Center for Nanointegration Duisburg-Essen), University of Duisburg-Essen, 47057 Duisburg, Germany

ARTICLE INFO

Article history:
Received 19 September 2012
Received in revised form 11 March 2013
Accepted 22 April 2013
Available online 3 May 2013

Keywords:
Monte Carlo
Particle
Coagulation
Inverse method
GPU
CUDA

ABSTRACT

Simulating particle coagulation using Monte Carlo methods is in general a challenging computational task due to its numerical complexity and the computing cost. Currently, the lowest computing costs are obtained when applying a graphic processing unit (GPU) originally developed for speeding up graphic processing in the consumer market. In this article we present an implementation of accelerating a Monte Carlo method based on the Inverse scheme for simulating particle coagulation on the GPU. The abundant data parallelism embedded within the Monte Carlo method is explained as it will allow an efficient parallelization of the MC code on the GPU. Furthermore, the computation accuracy of the MC on GPU was validated with a benchmark, a CPU-based discrete-sectional method. To evaluate the performance gains by using the GPU, the computing time on the GPU against its sequential counterpart on the CPU were compared. The measured speedups show that the GPU can accelerate the execution of the MC code by a factor 10–100, depending on the chosen particle number of simulation particles. The algorithm shows a linear dependence of computing time with the number of simulation particles, which is a remarkable result in view of the n^2 dependence of the coagulation.

1. Introduction

The dynamics of dispersed particle systems is of interest in several areas of science and engineering including combustion, aerosol reactor engineering, atmospheric aerosols and chemical engineering [1,2]. The evolution of a dispersed phase is governed by the population balance equation, which can account for all the processes that generate, modify and remove particles from the population, such as coagulation, breakage, nucleation, condensation/evaporation. Among those mechanisms, coagulation is of primary importance with regard to the understanding of the behavior, handling and treatment of particle systems. Furthermore, it is mathematically one of the most demanding processes as it requires to take into account binary particle interactions leading to a n^2 dependence. The particle coagulation process is described by the general dynamic equation (GDE [1])

$$\frac{\partial n(v,t)}{\partial t} = \frac{1}{2}\int_0^v \beta(v-u,u)n(u,t)n(v-u,t)du - n(v,t)\int_0^\infty \beta(v,u)n(u,t)du \tag{1}$$

[*] Corresponding author. Address: Universität Duisburg-Essen, Bismarckstr. 81, 47057 Duisburg, Germany. Tel.: +49 203 379 2899; fax: +49 203 379 3268.
E-mail address: einar.kruis@uni-due.de (F.E. Kruis).

0021-9991/$ - see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jcp.2013.04.030

where $n(v, t)$ is the particle size distribution (PSD) at time t and $\beta(u, v)$ is the coagulation rate for two particles with volume u and v. The first term on the right-hand of above equation is accounting for the formation of a particle with volume v due to the coagulation event between a particle of volume u and a particle of volume $(v - u)$; the coefficient 1/2 is introduced since collisions are counted twice in the integral. The second term indicates the disappearance of a particle with volume$v$ as a result of collisions with particles of other sizes.

Different methods are available at this point to solve this integro-differential equation, including sectional methods [3–6], methods of moments [7–10] and Monte Carlo (MC) methods [11–19]. All these methods have different precision, algorithm complexity and computational efficiency. To be more specific, sectional methods take moderate computational time, yet their sectional representations often result in rather complicated algorithms. Methods of moments are computationally efficient, however usually require knowledge about the shape of the PSD which can however greatly change during the simulation, especially when another mechanism occurs simultaneously such as nucleation. In contrast to these deterministic schemes stand the MC methods which have a stochastic nature. By describing the particle population with a limited number of simulation particles, MC methods can describe directly the dynamic evolution of a particle population in dispersed systems and solve Eq. (1) in an indirect way. The advantages of the MC methods can be summarized as follows:

- The discrete character associated with MC methods makes them the natural choice for processes such as coagulation that are inherently discrete.
- MC methods allow to provide information about particle history.
- They are capable of describing multivariate particle properties, i.e., they can provide the joint property distribution functions.
- The numerical algorithms are in general straightforward, thus greatly minimizing the programming efforts.

The major shortcoming associated with the MC methods lie in their limited precision and high computing cost. This is because MC methods are essential stochastic simulation methods in which the computational precision is inversely proportional to the square root of the total number of the samples (e.g., simulation particles) [13], therefore the number of simulation particles should be sufficiently large in order to maintain the computation accuracy. Currently, MC methods for particle dynamics are limited to some 10^5 simulation particles in a single simulation due to the computational burden, and usually not more than 10^4 particles are applied. Under this circumstance, high performance computing (HPC) clusters with high-end microprocessors can be resorted to in order to alleviate the heavy computational burden but this is very costly. On the other hand, another promising solution that has appeared recently is the use of many-core processors, which focuses more on the parallel execution of relatively simple core applications. In the last years, graphics processing units (GPUs) such as the NVIDIA® GeForce® series have seen a tremendous technological development. In the past decade, the performance improvement of general-purpose microprocessors has slowed significantly, whereas GPUs have continued to improve relentlessly and far outpace that of the former in terms of the computational capacity. For example, as far as the hardware in the present work concerned, the peak performance of a CPU with four cores running on 2.66GHz is 42.6 GFLOPS (a GFLOP depicts 10^9 float point operation per second), whereas the peak performance of a NVIDIA GTX 285 GPU containing 240 cores can attain 1062 GFLOPS. In the past, GPUs were designed to speed up the performance of graphic cards for gaming applications in the consumer market and were not accessible by a higher level programming language, but since several years, the NVIDIA GPUs can easily be programmed by means of CUDA™ (Compute Unified Device Architecture, [20]). In addition to the large number of FLOPs, the GPU solutions are also quite cost-effective as the market price is only 1-2 EUR per core. Finally, they have low electricity consumption in comparison to conventional HPC cluster, minimal requirements for workspace and no needs for high-powered air conditioning [21].

The GPU is a massively multi-threaded architecture containing hundreds of processing cores. The cores of the GPU execute in a Single Instruction, Multiple Data (SIMD) mode at the lowest level. Eight cores are grouped in SIMD fashion into streaming multiprocessors (SMs), hence all cores in a SM execute the same instruction. In particular, the NVIDIA GTX 285 used in the present study has 30 of these SMs, with each SM consisting of 8 stream processors (SPs), thus making for a total of 240 processing cores [22]. The GTX 285 also has different types of memory at various levels. For instance, each SM has one 16KB of fast on-chip shared memory and clocked at 1.47GHz. A set of 32-bit registers is evenly assigned to the threads in each SM. In addition, GTX 285 GPU comes with 1GB of device DRAM memory which can be accessed by all the threads in the grid. The memory is connected via a PCI-Express bus with a bandwidth 4 GB/s to a North Bridge chip, which also connects the GPU to the CPU and main memory on CPU. It is important to realize that the GPU contains both on-chip shared memory and off-chip global memory. In order to gain maximum performance, the GPU allows that the different levels of the memory system can be chosen by the programmer. The global memory has a large memory size but slow access while the shared memory has fast access but a much smaller memory size and it is visible only by other threads in the same block. In addition, for the MC simulation in the present work, the global memory, together with the total particle properties jointly determine the maximum number of particles that can be accommodated on the GPU. As an example, each simulation particle in this work has three properties (size, weight, total coagulation rate) as a single precision floating number and requires 12 bytes. Therefore, 8×10^7 simulation particles require 9.6×10^8 bytes, which can be easily stored within the global memory of 1.0 GB of the GTX285.

To the best of our knowledge, no efforts have been made to develop MC algorithms for population balance modeling which are suitable for the new computing architecture of a GPU. This article investigates the possibility of speeding up the execution of a MC method for describing particle coagulation by designing a new GPU-friendly numerical algorithm for the inverse method [12,15] with the help of CUDA. It particularly compares the computational accuracy and measures the performance improvement achieved on the GPU. The paper is organized as follows: Section 2 describes the inverse MC method for particle coagulation. An implementation of the MC method for the GPU is detailed in Section 3. Section 4 shows the validation of the new method and the performance improvement achieved on the GPU. Section 5 concludes the paper.

2. Monte Carlo algorithm for particle coagulation based on the Inverse Method

A simplified differentially weighted Monte Carlo method [15] based on the Inverse Method with smart bookkeeping [12] was chosen as basis for the present work. This method assigns a weight value to each simulation particle which allows to increase the precision when combined with a "shift action" which evenly distributes the number of simulation particles over the chosen size intervals [15]. This shift action was however not included in the algorithm. The weighting of the simulation particles furthermore has the advantage that it allows to deal with coagulation events between simulation particles coming from different cells in the combined CFD-MC simulation [18,23]. Mathematically, the method consists of three main parts, each of them will be detailed in the following subsections.

2.1. Initialization of the total coagulation rate C'_i

The initialization step requires the calculation of the coagulation kernels for all of the possible particle pairs i, j. For each simulation particle i the total coagulation rate C'_i has to be determined:

$$C'_i = \frac{1}{V^2} \sum_{j=1, j \neq i}^{n_s} \beta'_{i,j} \tag{2}$$

where V is the volume of the simulation system, n_s is the total number of simulation particles, $\beta'_{i,j}$ is the modified collision kernel between simulation particle i and j and is calculated as

$$\beta'_{i,j} = \beta_{i,j} \frac{2w_i \max(w_i, w_j)}{w_i + w_j} \tag{3}$$

with w_i and w_j being the particle weights [15]. Note that the corresponding particle concentration of the simulation particle i, which represents a group of particles, is w_i/V. Note also that the coagulation kernel $\beta_{i,j}$ is identical to $\beta(u, v)$ in Eq. (1) and that $\beta_{i,j}$ is symmetric, i.e., $\beta_{i,j} = \beta_{j,i}$, but $\beta'_{i,j}$ is no more symmetric because w_i is usually not equal to w_j.

2.2. Selection of a particle pair for coagulation

This step is responsible for selecting a pair of particles. Fig. 1 shows the basic procedure of this step. As can be seen, in order to fulfill the task of determining the coagulation pair it requires the knowledge of the cumulative probabilities of each particle and the average time between two successive coagulation events, Δt, calculated as

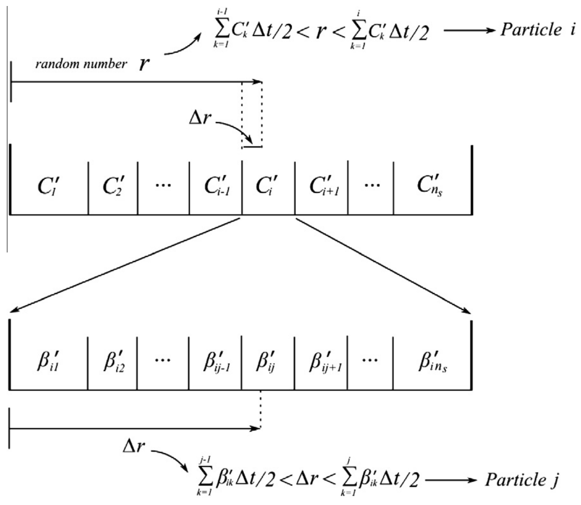
$\sum_{k=1}^{i-1} C'_k \Delta t/2 < r < \sum_{k=1}^{i} C'_k \Delta t/2$
Particle i
random number r
Δr
C'_1 C'_2 ··· C'_{i-1} C'_i C'_{i+1} ··· C'_{n_s}
β'_{i1} β'_{i2} ··· β'_{ij-1} β'_{ij} β'_{ij+1} ··· β'_{in_s}
$\sum_{k=1}^{j-1} \beta'_{ik} \Delta t/2 < \Delta r < \sum_{k=1}^{j} \beta'_{ik} \Delta t/2$
Particle j

Fig. 1. Schematics of the inverse method for choosing a pair of coagulating particles.

$$\Delta t = \frac{V}{\sum_{k=1}^{n_s} C'_k} \tag{4}$$

Then a desired particle pair (i, j) can, in turn, be accepted when the following conditions are satisfied

$$\sum_{k=1}^{i-1} C'_k \Delta t/2 < r < \sum_{k=1}^{i} C'_k \Delta t/2, \quad i \in [1, n_s], \tag{5a}$$

and

$$\sum_{k=1}^{j-1} \beta'_{i,k} \Delta t/2 < \Delta r < \sum_{k=1}^{j} \beta'_{i,k} \Delta t/2, \quad j \in [1, n_s], \tag{5b}$$

where the random number r lies in the interval (0, 1) and yields the first particle, $\Delta r = r - \sum_{k=1}^{i-1} C'_k \Delta t/2$ yields the second particle after the first particle has been chosen.

2.3. Updating the total coagulation rates

The C'_i values for all simulation particles have to be modified after the occurrence of a coagulation event, because the size and weight of the two selected simulation particles have changed as a result of that event. Evidently, the overhead of the computation would be expensive if one is still using Eq. (2) to reach this goal, as one would have $n_s(n_s - 1)$ different particle pairs. Hence, a smart bookkeeping technique [12], aiming at reducing the computing cost, is employed here. From the implementation point of view, this technique is straightforward. Assuming the selected coagulation pair to be (i, j), the smart bookkeeping technique corrects C'_k of arbitrary particle k due to the collision event between particle pair (i, j), with

$$C'^2_k = C'^1_k - \beta'^1_{k,i} - \beta'^1_{k,j} + \beta'^2_{k,i} + \beta'^2_{k,j} \quad \forall k \neq i, j, \tag{6}$$

where C'^1_k and C'^2_k represent the total coagulation rate of particle k with other particles *before* and *after* the current coagulation event; $\beta'^1_{k,i}$ and $\beta'^2_{k,i}$ stand for the modified coagulation rate between particle k and i before and after the coagulation event. A careful examination of Eq. (6) shows that it updates C'_k by merely computing the coagulation rate between the particle k and the coagulation pair (i, j) (i.e., ignoring the other particles). In this fashion, the burden of the calculation of C'^2_k has been greatly alleviated. On the other hand, the computation of C'_i and C'_j of the coagulation pair i and j still requires the use of Eq. (2). Overall, instead of calculating $n_s(n_s - 1)$ coagulation kernels and performing an equal number of additions, only $4(n_s - 2) + 2(n_s - 1) = 6n_s - 10$ kernels as well as summations have to be calculated. For 10^4 simulation particles, this means only 1.2×10^5 function evaluations and summations instead of 2×10^8.

3. Implementation of the inverse method with smart book-keeping on the GPU

3.1. Overview

As pointed out in the previous sections, the computing cost of the Inverse MC for particle coagulation might still be high even with the smart bookkeeping technique, especially for systems in which also a spatial inhomogeneity has to be taken into account (multiple cells). As the method described in Section 2 mainly consists of simple numerical calculations in loops and double loops, it is to be expected that the MC code can be accelerated by using a GPU, which provides massively parallel processors. To this end, the following parallel algorithm for the Inverse method on the GPU has been designed, which includes the following steps:

1. Perform the initialization of the data involved, mainly of the particle properties. This step is carried out on the CPU.
2. Allocate global memories on the GPU for receiving the initial particle properties and kinetic data from the CPU.
3. Compute the initial C'_i of each particle on the GPU.
4. Calculate on the GPU the sums over all C'_i values for calculating Δt, generate a random number r, and select a particle pair.
5. Calculate on the GPU the new values C'_i of all particles by means of smart bookkeeping.
6. Update the current time t, transfer it to the CPU which judges whether the simulation comes to an end. If not, return to step 4 and perform steps 4 to 6.
7. Transfer the results from the GPU back to the CPU.

Steps 3, 4 and 5 correspond to the three key parts of MC described in Sections 2.1, 2.2 and 2.3 and determine the computational efficiency of the MC code on the GPU. Their implementation on the GPU by means of CUDA kernels will be discussed in more detail in the Sections 3.2 through 3.4.

3.2. Kernel for initialization

The calculation of C_i' for all particles is straightforward. It consists of two loop levels, with one loop iterating over all particles and other loop stepping through the concrete terms for a specified particle. Computing C_i' for each particle based on Eq. (2) is expensive as it involves $n_s - 1$ terms, thus totaling $n_s(n_s - 1)$ terms for all particles. However, a careful examination of Eq. (2) shows that the computation of C_i' for each particle is independent of that of other particles, and the calculation of each coagulation kernel is also independent of that of others. Therefore it is ideally suited for parallelization using a GPU where each thread is used to compute one coagulation kernel. In order to illustrate the procedure, it is assumed that the particle number involved in the simulation is 2000 and that each thread block contains 512 threads, which is currently the maximum thread number per block allowed by CUDA. The data handling is illustrated in Fig. 2.

Due to the limitations in the maximum number of threads (*threadNum*) and blocks allowed by CUDA (*maxBlockNumCUDA*), the number of blocks required to deal with the chosen number of simulation particles (*partNum*) is:

$$blockPerPart = \frac{partNum + threadNum - 1}{threadNum}. \tag{7}$$

With *partNum* = 2000, the required number of blocks is 4. This leads to a certain number of unused threads in one block which is unavoidable for a general algorithm but can be avoided by choosing *partNum* as multiple of 512. For large values of *partNum*, it can occur that *partNum* × *blockPerPart* becomes larger than the maximum block number allowed by CUDA. Therefore the maximum block number is chosen as

$$maxBlockNum = int\left(\frac{maxBlockNumCUDA}{blockPerPart}\right) \cdot partNum. \tag{8}$$

When the block index *bid* surpasses this value, it is increased by *maxBlockNum* and the initializing calculations are repeated. In Fig. 3 it can be seen how the two particles indices *pid1* and *pid2* are determined as function of *bid* and the thread index *tid*. After the thread calculated the collision kernel, it is stored in the shared memory, allowing a rapid parallel summation, e.g., requiring only 9 cycles for 2^9=512 threads. Note that $\beta'_{0,0}$ (as well as $\beta'_{1,1}$, $\beta'_{2,2}$, etc.) means the coagulation of particle with itself and it will be ignored in the real computing by putting $\beta'_{i,i} = 0$.

The partial sums for each block, $C_{ij}PartialSum[bid]$, are stored in global memory. In Appendix B the pseudo-code of the initializing kernel is shown in algorithm 1. It consists of two CUDA kernels. The first one performs the coagulation kernel

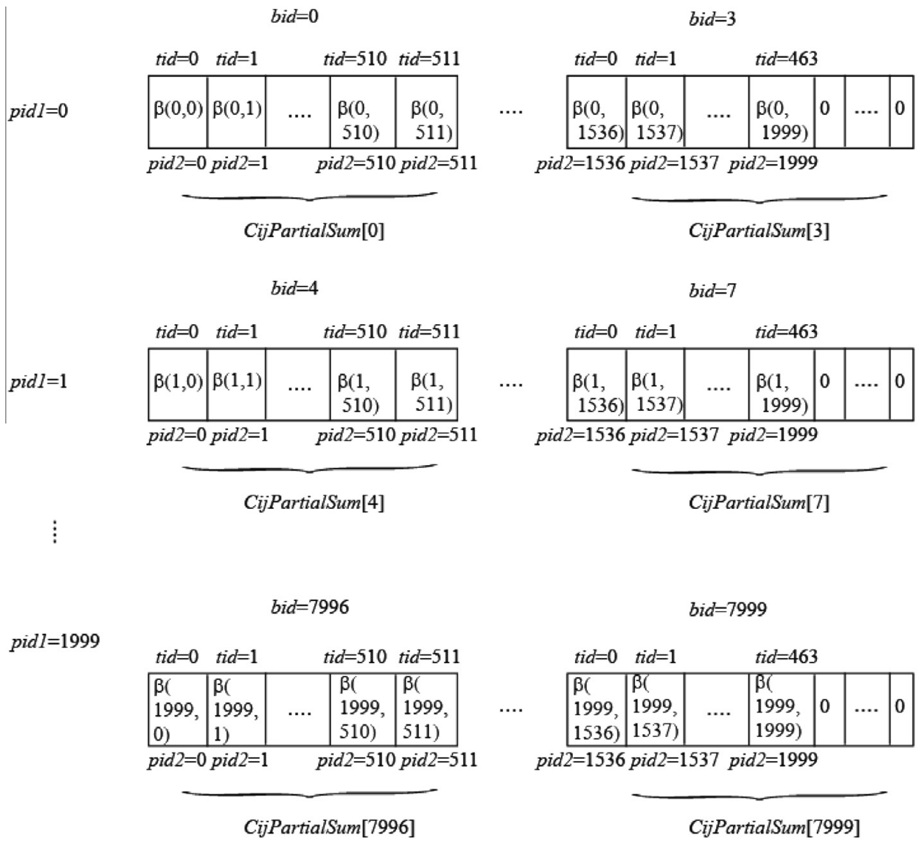
bid=0
bid=3
tid=0 tid=1 tid=510 tid=511
tid=0 tid=1 tid=463
pid1=0
β(0,0) β(0,1) .... β(0, 510) β(0, 511)
β(0, 1536) β(0, 1537) .... β(0, 1999) 0 0
pid2=0 pid2=1 pid2=510 pid2=511
pid2=1536 pid2=1537 pid2=1999
CijPartialSum[0]
CijPartialSum[3]
bid=4
bid=7
pid1=1
β(1,0) β(1,1) .... β(1, 510) β(1, 511)
β(1, 1536) β(1, 1537) .... β(1, 1999) 0 0
CijPartialSum[4]
CijPartialSum[7]
bid=7996
bid=7999
pid1=1999
β(1999, 0) β(1999, 1) .... β(1999, 510) β(1999, 511)
β(1999, 1536) β(1999, 1537) .... β(1999, 1999) 0 0
CijPartialSum[7996]
CijPartialSum[7999]

Fig. 2. Distribution of the initialization computations over the threads and blocks of the GPU.

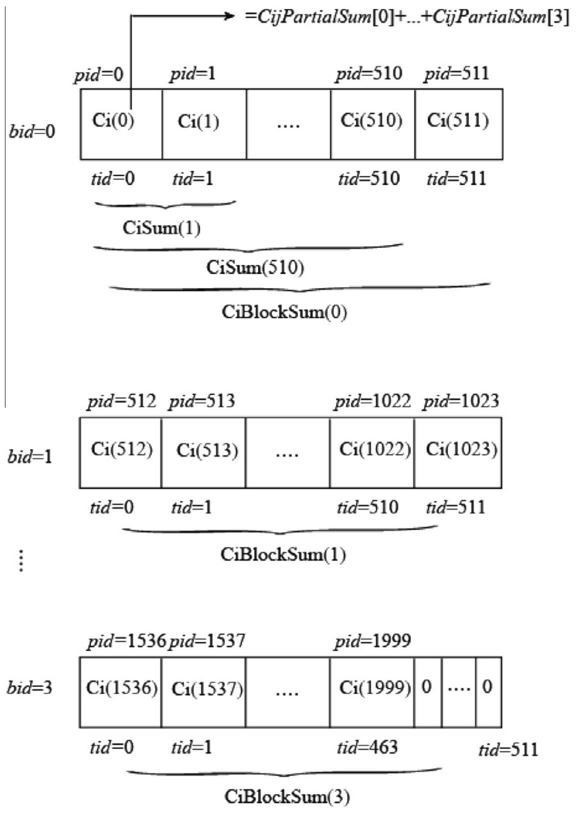

Fig. 3. Schematics illustrating the data handling necessary for the summation procedure.

calculation, the parallel summation and the storage of the partial sums $C_{ij}PartialSum[bid]$ in global GPU memory. The second sub-kernel then calculates C'_i for all simulation particles (termed as $Ci[pid]$ in pseudo-code) and stores it in global memory.

3.3. Kernel for selecting a particle pair for coagulation

Selecting a particle pair for coagulation is accomplished on the basis of Eqs. (5a) and (5b). By and large, the selection procedure cannot be fully parallelized and effectively accelerated. This is because the selection procedure is an intrinsic sequential operation. However, the procedure also exhibits some data parallelism. To be more specific, computing the leftmost term in Eqs. (5a) and (5b) is exactly an all-prefix-sums operation (see Appendix A) that can be effectively and easily handled by CUDA with a work-efficient parallel scan algorithm [24].

The CUDA kernel used for particle pair selection is composed of four sub-kernels, the pseudo-code is given in Appendix B in algorithm 2. All sub-kernels require *blockNum* blocks so that they can deal with *partNum* simulation particles using *threadNum* threads:

$$blockNum = \frac{partNum + threadNum - 1}{threadNum}. \tag{9}$$

The first sub-kernel determines the sum over the C'_i in each block by means of the parallel all-prefix summation method earlier described, and stores the individual sums $CiSum[pid]$ as well as the sums over the blocks as $CiBlockSum[bid]$ in global memory. In the second sub-kernel, first the individual $CiSum[pid]$ are corrected with the help of the block sums. A schematic diagram of the data distribution over the blocks and threads is shown in Fig. 3. In one thread the time step is calculated based on Eq. (4) as well as a random number r based on a seed value stored in global memory (and subsequently stored in global memory as future seed value), allowing now by means of a rapid interval search to determine the first selected particle *pid1* according to Eq. (5a) as well as Δr and place them in global memory. The random number generator used, *ranqd1*, is a very fast linear congruential random number generator with a period of 2^{32} [25]

$$r_{i+1} \equiv a \cdot r_i + b(\mathrm{mod}\, p), \quad i \geqslant 0, \tag{10}$$

where $a = 1664525$, $b = 1013904223$ and $p = 2147483648$. It generates a new random number r_{i+1} starting from a previous random number r_i or an initial seed value r_0. r_{i+1} is distributed in the interval $[-2^{31}, 2^{31} - 1]$, so that is casted into a random number in the interval [0, 1] by a suitable transformation.

The third and fourth sub-kernel operate in a similar way for selecting the second particle *pid2*, with the difference that Eq. (5b) based on Δr is applied now and that the values of C'_{ij} are newly calculated and not taken from global memory.

3.4. Kernel for updating C'$_i$

The C'_i values of the selected particles need to be recalculated based on Eq. (2) after each coagulation event whereas for all other particles the C'_i values can be adjusted with the help of Eq. (6). Both operations can be efficiently parallelized applying *blockNum* blocks and *threadNum*threads. Its CUDA implementation is straightforward and similar to the procedures already explained in Section 3.2. Appendix B, algorithm 3 shows the corresponding pseudo-code.

4. Results and discussions

4.1. Validation procedure

The GPU-based algorithm is validated by comparison with a benchmark numerical solution, which is here an implementation of discrete sectional model [26] using 200 sections and a sectional spacing of 1.12. As coagulation kernel the Brownian coagulation kernel for the free-molecule regime [27] has been used in the present work:

$$\beta_{i,j} = K_F(d_i + d_j)^2\left(\frac{1}{d_i^3} + \frac{1}{d_j^3}\right)^{1/2}, \tag{11}$$

where d_i and d_j stand for the diameter of particle i and j, respectively; $K_F = (3\kappa_B T/\rho_p)^{1/2}$ with T being the temperature, κ_B the Boltzmann constant and ρ_p the particle density.

The significant parameters used for the validation are tabulated in Table 1, where d_0 is the initial particle diameter, $\sigma_{g,0}$ is the initial geometric standard deviation of the PSD and N_0 is the initial particle number concentration. The evolution of average particle size, number concentration as well as the geometric standard deviation is simulated for a time t up to $10^3\tau_{char}$ where τ_{char} is the characteristic coagulation time, defined as the time needed to decrease the initial particle concentration by a factor of 2 [27]:

$$\tau_{char} = \frac{1}{\left(\frac{3V_t}{4\pi}\right)^{1/6} K_F N_0^{5/6}}, \tag{12}$$

where V_t is the aerosol volume fraction, $V_t = N_0\pi d_0^3/6$.

The simulations in this work were performed on a desktop system equipped with an Intel Core ™ 2 Quad 2.66 GHz Processor, 8 GB RAM, and a video card NVIDIA Geforce GTX 285 GPU. The GTX 285 has 240 cores with 16 KB of shared memory and 1 GB of global memory. It was operated in single precision under Ubuntu 9.04, with the NVIDIA Graphics Driver version 185.18, and CUDA SDK/Toolkit version 2.2.

4.2. Results of the validation of the GPU-based Inverse Method

The computing accuracy of the MC code on the GPU was first verified by comparison with the benchmark discrete-sectional method. The mean values obtained from 100 simulations with different random numbers and using 2000 simulation particles in comparison to the benchmark solution are seen in Figs. 4 and 5. The curves show the increase in geometric particle diameter d_g, which is characteristic for coagulation. The GPU-based Inverse Method shows a good agreement with the benchmark solution coming from the sectional model. In addition, the development of the geometric standard deviation from 1.0 initially to a self-preserving size distribution (SPSD) with $\sigma_g = 1.455$ shows that also the width of the size distribution is correctly simulated.

Although systematic errors can effectively be detected by taking the value over a large number of simulations as is done in Fig. 4 and 5, it is also useful to get information about the statistical errors as function of the number of runs with different random numbers (n_r) and number of simulation particles (n_s). Fig. 6 shows the mean error in d_g evaluated over $n_t = 16$ different time points between $10^{-3}\tau_{char}$ and $10^3\tau_{char}$, when performing the simulation $n_r = 100$ times:

$$\varepsilon_{mean} = \frac{\sum_{i=1}^{n_t}\left(\left|\left(\sum_{j=1}^{n_r} d_g(i,j)/n_r\right) - d_g^{sec}(i)\right|/d_g^{sec}(i)\right)}{n_t}. \tag{13}$$

Table 1
Parameters employed in the simulation.

d_0 (nm)	$\sigma_{g,0}$	N_0 (1/m^3)	T (K)	ρ_p (kg/m^3)
3.0	1.0	10^{17}	300	1000

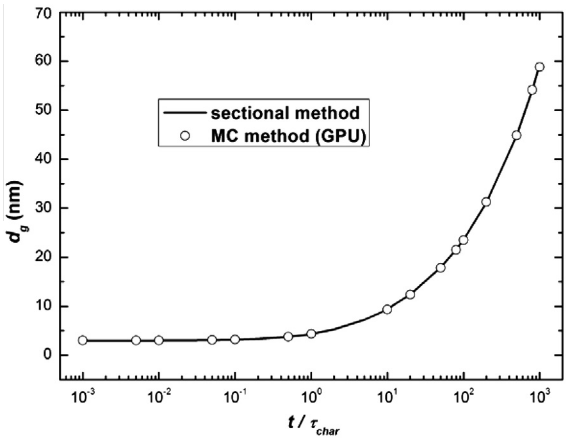

Fig. 4. Validation of the evolution of geometric particle diameter d_g with dimensionless simulation time. The number of simulation particles $n_s = 2000$, whereas the number of runs $n_r = 100$.

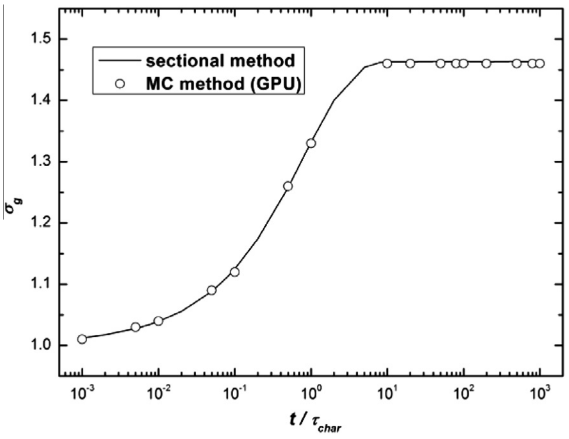

Fig. 5. Evolution of geometric standard deviation σ_g with dimensionless simulation time. Conditions as in Fig. 4.

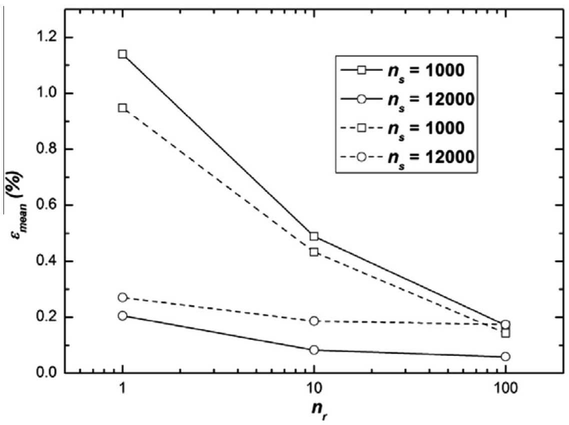

Fig. 6. Mean error ε_{mean} in the geometric particle diameter d_g (solid line) and σ_g (dashed line) calculated with the GPU-based inverse method by means of comparison to the benchmark solution as function of the number of MC runs n_r. The results are shown for two different numbers of simulation particles ($n_s = 1000$ or 12,000).

In a similar way, the mean error in the geometric standard deviation is evaluated and shown in Fig. 6. As can be expected, ε_{mean} decreases when n_r or n_s is increased. In this type of MC simulation, the question is often what is the more efficient procedure to obtain a better precision: increasing n_r or n_s. Interestingly, the figure shows that the mean error is clearly smaller with $n_s = 12000$ as compared to a simulation with $n_s = 1000$ and n_r a factor 10 larger which requires approximately the same computer time. This is due to the linear scaling of the computer time with n_s which will be shown in the next paragraph. Therefore, we can conclude that taking a large value of n_s while repeating the simulation only a limited number of times is the better strategy for optimizing computing efficiency.

4.3. Computing time and speedup of the GPU-accelerated inverse method

After having validated the code, we now turn towards the computing time, as the purpose of the GPU implementation is to speed up the simulation. The computing time on the GPU (t_G) is now compared to the one on the CPU (t_C) with the conventional non-parallel implementation of the Inverse Method. The computing time on the GPU does not include the time for data transfer between CPU and GPU because the particle data is fully stored and analyzed on the GPU. In Fig. 7, it can be seen that the GPU needs only 30 s for finishing the simulation with 8000 simulation particles while the CPU needs 1650 s.

An effective speedup factor α can now be defined as the ratio of the computing time on the GPU and on the CPU:

$$\alpha = \frac{t_C}{t_G}. \tag{14}$$

The speedup factor is shown in Fig. 8 for different numbers of simulation particles n_s between 1000 and 12,000. As can be seen, speedup factors between 10 and 100 are obtained. The speedup turns out to be fairly independent of the simulation time, whereas it is clearly increasing with the number of simulation particles. In GPU programming, usually the speedup factor becomes larger when the number of blocks is increasing at a fixed number of threads and this is the case when increasing the number of simulation particles. A speedup of 100 shows the efficiency of the parallel software, as the number of GPU processors is 240 having a clock speed lower than that of the CPU.

The measured speedup is jointly determined by the individual speedup of each sub-kernel, as described in Sections 3.2–3.4. In general, the speedup for the initialization kernel is the largest, whilst for the selection kernel is the smallest. As an

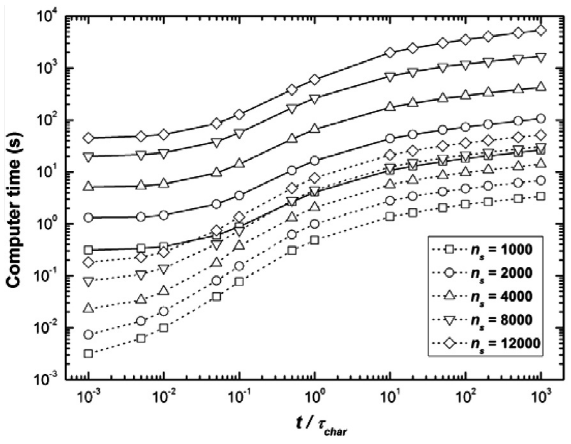

Fig. 7. Computing time used on CPU (solid line) and GPU (dashed line) as a function of dimensionless simulation time, for different numbers of simulation particles n_s between 1000 and 12,000, whereas the number of runs $n_r = 1$.

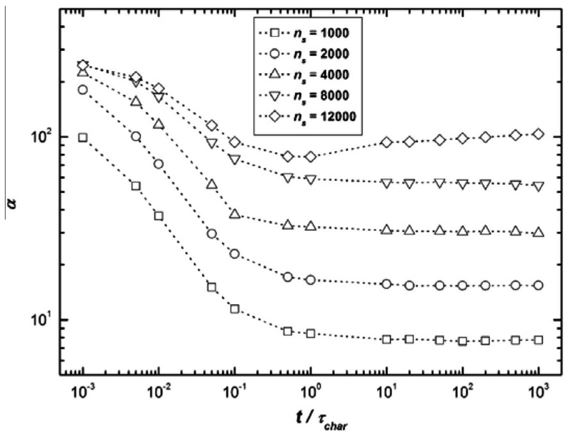

Fig. 8. Measured speedups as a function of simulation time.

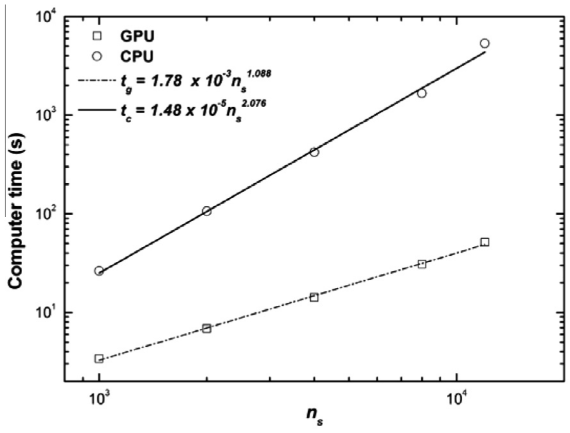

Fig. 9. Computer time *vs.* number of simulation particles.

example, the speedups for these three kernels are 239, 0.82 and 18, respectively, for a single coagulation event using 1000 simulation particles.

Careful study of Fig. 7 shows that the computer time increases more rapidly with number of simulation particles for the CPU-based program than for the GPU-accelerated one. This can be more easily seen in Fig. 9 where the required computer time, here for simulation in interval $[\tau_{char}, 10^3\tau_{char}]$, is plotted as a function of the number of simulation particles. It appears that the CPU software scales with particle number as $(n_s)^{2.08}$, while the computing time of the GPU scales with $(n_s)^{1.09}$. This is an important achievement, as in order to obtain sufficient precision n_s has to be chosen sufficiently large as shown in the preceding paragraph.

5. Conclusions

Simulating particle coagulation using Monte Carlo methods is in general a challenging computational task due to its numerical complexity and the computing cost. Currently, the lowest computing costs are obtained when applying graphic processing units originally developed for speeding up graphic processing in the consumer market. To improve the computing performance, a MC method based on the inverse method has been implemented on a GPU, which provides a large number of parallel threads at low costs. The adaption of the inverse method to GPU computing was described in detail, as optimized GPU-based algorithms are rather different from conventional algorithms. A discrete-sectional method was utilized as a benchmark to validate the algorithm. A study was made of the effect of varying the number of simulation particles as well as the number of MC runs on the computational accuracy. It was shown that for the developed parallelized software taking a large value of simulation particles while repeating the simulation only a limited number of times is the better strategy for optimizing computing efficiency. Speedup factors between 10 and 100 are obtained for 1000 and 12,000 simulation particles, respectively. It appears that the GPU-based algorithm leads to a linear scaling of the computing time with number of simulation particles. This is an important result in view of the n^2 dependence of the coagulation problem. It has to be emphasized, however, that the inverse method is especially suited for systems where there is no change in the particle properties between coagulation due to the updating procedure. When the particle properties change due to other mechanism such as transport from other cells in a CFD surrounding or particle growth, other MC methods such as the acceptance-rejection method are more suitable.

Acknowledgement

The research project no. 330 Z of the research association Institut für Energie- und Umwelttechnik e. V. (IUTA), has been funded via AiF within the agenda for the promotion of industrial cooperative research and development (IGF) by the German Federal Ministry of Economics and Technology based on a decision of the German Bundestag.

Appendix A:. The all-prefix-sums operation

Definition: The all-prefix-sums operation takes a binary associative operation$\oplus$, and an array of n elements

$$[a_0, a_1, \ldots, a_{n-1}]$$

and returns

$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})]$$

where $\oplus$ is addition. For example, the all-prefix-sums operation on the following array

$$[1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8]$$

will produce

$$[1 \quad 3 \quad 6 \quad 10 \quad 15 \quad 21 \quad 28 \quad 36]$$

Appendix B:. Pseudo-codes

partNum: number of particles chosen for the simulation
threadNum: number of threads per block, use the maximum allowed by CUDA, i.e., 512
blockPerPart: number of blocks assigned to each particle
blockNum: number of blocks necessary to deal with *partNum* particles using 512 threads
maxBlockNumCUDA: maximum block number allowed in CUDA protocol, e.g., 65535
maxBlockNum: maximum block number which can be used in initializing kernel
coag_computing(*i*, *j*): device function for computing the coagulation rate between particle *i* and *j*
For all kernels:
tid ← *threadIdx.x.*
bid ← *blockIdx*.x

```
Algorithm 1: Initializing kernel, consisting of two subkernels
1: BlockSize: maxBlockNum (1st subkernel), blockNum (2nd subkernel)
   2: //launch first subkernel for calculating the contributions of the individual blocks to Ci, stored in CijPartialSum[bid]:
   3: maxBlockNum ← (maxBlockNumCUDA/blockPerPart)∗blockPerPart
   4: while bid < partNum∗blockPerPart do
   5: pid1 ← bid / blockPerPart //index of the first particle
   6: pid2 ← (bid % blockPerPart)∗threadNum + tid //index of the second particle
   7: //Computing all contributions to Ci and save the results in shared memory
   8: if pid2 < partNum && pid1 != pid2 do
   9: shared_CijPartialSum[tid] ← coag_computing(pid1, pid2)
   10: end if
   11: //within a block:
   12: shared_CijPartialSum[0] ← Σ(CijPartialSum[tid]) //a parallel summation
   13: //Save the intermediate results in global memory CijPartialSum, used for the second subkernel:
   14: if tid == 0 do
   15: CijPartialSum[bid] ← shared_CijPartialSum[tid]
   16: end if
   17: Syncthreads
   18: bid ← bid + maxBlockNum
   19: end while
   20: // launch the second subkernel, used for adding up the partial sums into the total Ci value
   21: pid ← bid∗threadNum + tid
   22: for k = 0, k < blockPerPart, k++ do
   23: Ci[pid] += CijPartialSum[pid∗blockPerPart + k]
   24: end for
```

```
Algorithm 2: Selecting kernel, consisting of four sub-kernels
1: Blocksize = blockNum
   2: //launch the first subkernel for calculating the sums over the Ci's for each block which is stored as CiBlockSum[bid], also
   the partial sums over the Ci's are calculated for each particle blockwise and stored in CiPartialSum
   3: pid ← bid∗threadNum + tid
   4: shared_CiPartialSum[tid] ← Ci[pid] //below a sync
   5: //within a block:
   6: shared_CiPartialSum[0] ← ⊕(shared_CiPartialSum[tid]) //a parallel all-prefix-sums operation
   7: Syncthreads
```

(continued on next page)

```
8: CiSum[pid] ← shared_CiPartialSum[tid]
9: Syncthreads
10: if tid == 0 do
11: CiBlockSum[bid] ← shared_CiPartialSum[threadNum -1]
12: end if
13://launch the second subkernel to calculate the correct CiSum[pid] by adding CiBlockNum[pid] for all particles. Calculate
time step and select the first particle
14: for k = 1, k < blockNum, k++ do
15: CiSum [pid] += CiBlockSum[k-1]
16: end for
17: //within the first thread of the last available block:
18: $\Delta t$ ← 2.0/CiSum[partNum-1] //average coagulation time step
19: lower ← 0, upper ← partNum -1
20: while (upper – lower) > 1 do
21: mid ← (lower +upper)/2
22: if (r < CiSum[mid]/2*$\Delta t$) do //r is a random number between 0 and 1
23: upper ← mid
24: else
25: lower ← mid
26: end if
27: end while
28: pid1 ← upper //find the first particle
29: temp ← CiSum[lower]

30://launch the third subkernel for calculating sums over the Cij's for each block which is stored as CijBlockSum[bid], also
the partial sums are calculated for each particle blockwise and stored in CijPartialSum[pid]
31: pid ← bid*threadNum + tid
32: //Calculate the coagulation rate between particle i and other particles:
33: shared_CijPartialSum[tid] ← coag_computing(pid1, pid)
34: Syncthreads
35: //within a block:
36: shared_CijPartialSum[0] ← $\oplus$(shared_CijPartialSum[tid]) // $\oplus$ is a parallel all-prefix-sums operation
37: Syncthreads
38: CijPartialSum [pid] ← shared_CijPartialSum[tid]
39: Syncthreads
40: if tid == 0 do
41: CijBlockSum[bid] ← shared_CijPartialSum[threadNum - 1]
42: end if
43://launch the fourth subkernel: calculating the correct CijPartialSum[pid] by adding CijBlockSum[bid] for all particles.
Select the second particle
44: for k = 1, k < blockNum, k++ do
45: CijSum[tid] += CijBlockSum[k-1]
46: end for
47: //within the first thread of the last block:
48: lower ← 0, upper ← partNum -1
49: while (upper – lower) > 1 do
50: mid ← (lower +upper)/2
51: if r < (temp + CijSum[mid])/2*$\Delta t$ do
52: upper ← mid
53: else
54: lower ← mid
55: end if
56: end while
57: pid2 ← upper //find the second particle
```

Algorithm 3: Updating kernel

```
1: Blocksize = blockNum
  2: pid ← bid*threadNum + tid
```

```
3://updating Ci of particles other than pid1 and pid2 using the smart bookkeeping technique:
4: temp $\leftarrow$ 0.0, k $\leftarrow$ 0
5: **if** pid != pid1 && pid != pid2 && pid < partNum **do**
6: oldBeta1 $\leftarrow$ coag_computing(pid, $pid1^{old}$) //using previous value of particle pid1
7: oldBeta2 $\leftarrow$ coag_computing(pid, $pid2^{old}$) //using previous value of particle pid2
8: newBeta1 $\leftarrow$ coag_computing(pid, $pid1^{new}$) //using new value of particle pid1
9: newBeta2 $\leftarrow$ coag_computing(pid, $pid2^{new}$) //using new value of particle pid2
10: Ci[pid] $\leftarrow$ Ci[pid]– oldBeta1 – oldBeta2 + newBeta1 + newBeta2
11: **end if**
12: Syncthreads
13: //updating particle pid1:
14: **if** pid != pid1 **do**
15: CijPartialSum[pid] $\leftarrow$ coag_computing($pid1^{new}$, pid)
16: **end if**
17: Syncthreads
18: CijPartialSum[pid] $\leftarrow$ $\Sigma$(CijPartialSum[tid]) //a parallel summation operation
19: Syncthreads
20: **for** k = 0, k < blockNum, k++ **do**
21: temp += CijPartialSum[k*threadNum]
22: **end for**
23: Ci[pid1] $\leftarrow$ temp
24: Syncthreads
25: //updating particle pid2 in the same way as pid1, line 14–24
```

References

[1] S.K. Friedlander, Smoke, Dust and Haze: Fundamentals of Aerosol Behavior, Wiley, New York, 1997.
[2] D. Ramkrishna, Population Balances: Theory and Applications to Particulate Systems in Engineering, Academic Press, San Diego, 2000.
[3] J. Jeong, M. Choi, A sectional method for the analysis of growth of polydisperse non-spherical particles undergoing coagulation and coalescence, J. Aerosol Sci. 32 (2001) 565–582.
[4] D. Mitrakos, E. Hinis, C. Housiadas, Sectional modeling of aerosol dynamics in multi-dimensional flows, Aerosol Sci. Technol. 41 (2007) 1076–1088.
[5] C.Y. Wu, P. Biswas, Study of numerical diffusion in a discrete-sectional model and its application to aerosol dynamics simulation, Aerosol Sci. Technol. 29 (1998) 359–378.
[6] J.D. Landgrebe, S.E. Pratsinis, A discrete-sectional model for powder production by gas phase chemical reaction and aerosol coagulation in the free-molecular regime, J. Colloid Interface Sci. 139 (1990) 63–86.
[7] S.H. Park, K.W. Lee, M. Shimada, K. Okuyama, Alternative analytical solution to condensational growth of polydisperse aerosols in the continuum regime, J. Aerosol Sci. 32 (2001) 187–197.
[8] M.Z. Yu, J.Z. Lin, T. Chan, A new moment method for solving the coagulation equation for particles in Brownian motion, Aerosol Sci. Technol. 43 (2008) 781–793.
[9] M. Yamamoto, A moment method of an extended log-normal size distribution application to Brownian aerosol coagulation, J. Aerosol Res. 19 (2004) 41–49.
[10] D.A. Terry, R. McGraw, R.H. Rangel, Method of moments for a laminar flow aerosol reactor model, Aerosol Sci. Technol. 34 (2001) 353–362.
[11] A.L. Garcia, A Monte Carlo simulation of coagulation, Physica 143A (1987) 535–546.
[12] F.E. Kruis, A. Maisels, H. Fissan, Direct simulation Monte Carlo method for particle coagulation and aggregation, AIChE 46 (9) (2000) 1735–1742.
[13] K. Liffman, A direct simulation Monte-Carlo method for cluster coagulation, J. Comput. Phys. 100 (1992) 116–127.
[14] M. Smith, T. Matsoukas, Constant-number Monte Carlo simulation of population balances, Chem. Eng. Sci. 53 (1998) 1777–1786.
[15] H.B. Zhao, F.E. Kruis, Reducing statistical noise and extending the size spectrum by applying weighted simulation particles in Monte Carlo simulation of coagulation, Aerosol Sci. Technol. 43 (2008) 781–793.
[16] K. Lee, T. Matsoukas, Simultaneous coagulation and break-up using constant-*N* Monte Carlo, Powder Technol. 110 (2000) 82–89.
[17] Y. Efendiev, M.R. Zachariah, Hybrid Monte Carlo method for simulation of two- component aerosol coagulation and phase segregation, J. Colloid Interface Sci. 249 (2002) 30–43, http://dx.doi.org/10.1006/jcis.2001.8114.
[18] H. Zhao, C. Zheng, A population balance-Monte Carlo method for particle coagulation in spatially inhomogeneous systems, Comput. Fluids 71 (2013) 196–207.
[19] A. Maisels, F.E. Kruis, H. Fissan, Direct Monte Carlo simulation for simultaneous nucleation, coagulation, and surface growth in dispersed systems, Chem. Eng. Sci. 59 (2004) 2231–2239.
[20] NVIDIA Corporation, NVIDIA GeForce GTX 285 Specifications, 2008.
[21] F. Molnar Jr., T. Szakaly, R. Meszaros, I. Lagzi, Air pollution modelling using a Graphics Processing Unit with CUDA, Comput. Phys. Commun. 181 (2010) 105–112, http://dx.doi.org/10.1016/j.cpc. 2009.09.008.
[22] NVIDIA Corporation, Compute Unified Device Architecture Programming Guide, version 2.0, 2007.
[23] F.E. Kruis, J.M. Wei, Till van der Zwaag, Stefan Haep, Computational fluid dynamics based stochastic aerosol modeling: combination of a cell-based weighted random walk method and a constant-number Monte-Carlo method for aerosol dynamics, Chem. Eng. Sci. 70 (2012) 109–120.
[24] M. Harris, M. Garland, Optimizing parallel prefix operations for the Fermi architecture, in: Wen-Mei W. Hwu (Ed.), Gpu Computing Gems Jade Edition, Elsevier, New York, 2011, pp. 29–38 (Chapter 3).
[25] T. Preis, P. Virnau, W. Paul, J.J. Schneider, GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model, J. Comput. Phys. 228 (2009) 4468–4477.
[26] S. Lu, Collision integrals of discrete-sectional model in simulating powder production, AIChE J. 40 (1994) 1761–1764.
[27] T.T. Kodas, M. Hampden-Smith, Aerosol Processing of Materials, Wiley-VCH, 1999.