Proceedings of 2017 4th International Conference on
Control, Decision and Information Technologies (CoDIT'17) / April 5-7, 2017, Barcelona, Spain

A Fast Monte Carlo GPU Based Algorithm for Particle Breakage

Jherna Devi [a,b], F.Einar Kruis [a]
[a] Institute of Technology for Nanostructures (NST) and Center for Nanointegration Duisburg –Essen (CENIDE), University Duisburg-Essen, Duisburg, D-47057, Germany
[b] Department of Information Technology, Quaid-e-Awam University of Engineering Science& Technology, Nawabshah, 67480 Sindh, Pakistan
jherna.devi@uni.due.de
einar.kruis@uni-due.de

***Abstract*—An algorithm for simulating the particle population balance in case of breakage is designed to function on a Graphic Processing Unit (GPU) in a Compute Unified Device Architecture (CUDA). The GPU lowers the computational cost of the particle breakage simulation, which is generally complex and demanding. We simulate particle breakage by a Population Balance-Monte Carlo (PB-MC) simulation method. Data analysis has confirmed that use of the GPU accelerates the execution of the MC program. The computational time of the algorithm linearly increases with the number of simulation entries (SEs). Here, an all-inclusive framework that accelerates the PB-MC simulation of particle breakage dynamics is introduced. The computational efficiency of the simulation method was significantly improved by the parallel computing approach enabled by the GPU. A fast acceptance–rejection (AR) and inverse scheme based algorithm that speeds up the performance of the PB-MC method has been implemented and validated. The complexity of the proposed algorithm is O(N).**

Keywords—GPU, CUDA; Parallel Processing; Particle Breakage; Population Balances

I. Introduction

The dynamics of particle breakage is relevant for different fields such as chemical engineering and particle technology. Particle breakage (sometimes also called 'comminution') refers to the process when a solid particle breaks into smaller-sized units. The Population Balance Equation (PBE) allows to solve the Particle Size Distribution (PSD) with time:

$$\frac{df(v_D,t)}{dt} = -S(v_D)f(v_D,t) + \int_{V_D}^{\infty} S(v_M)b(v_D,v_M)f(v_M,t)dv_M \quad (1)$$

Where $f(v_D,t)$ is the PSD function at time t and $f(v_M,t)dv_M$ is the fraction of particles which size ranges at time t between v and $v+dv$ per unit volume. For a particle v_M of volume v, the breakage rate is given by the expression $S(v)$ is calculated using equation 2. Whereas the equation 3 describes the breakage function $b(v_D,v_M)$ as the probability of formation of 'daughter' particles with volume V_D from a 'mother' particle of volume v_M .

$$S(v) = v_i \quad (2)$$

$$b(v_D,v_M) = \frac{2}{v_M} \quad (3)$$

The death part of the equation 1(first term at RHS) represents the removal of particle v_D due to their breakage into smaller fragments [1]. The birth part of the equation 1 (second term at RHS) defines the addition of particles with volume v_D due to breakage of particles with volume V_M. The particle size distribution can be obtained by modelling or through experiments.

The PBE for breakage is a challenging equation for modelling. In recent years, utilization of population balance methods such as MC modeling is gaining popularity. A good reason behind this could be the very nature of the MC model that is highly discrete in nature.

Several researchers have investigated MC methods for solving the PBE for particle breakage using the MC method [2]. The concepts of constant-number- [3][4][5] and constant-volume simulations [4][5] as time-driven MC methods on central processing units (CPU) have been introduced. The statistical weight of the Simulation Entries (SE) is used to simulate the PBE for breakage in a CPU implementation [5]. The constant-number and constant-volume based schemes are simulated in single cell [4] and multiple cells implementations [5] on the CPU. The CPU-based serial algorithm for particle breakage not only slows down the PB-MC simulation but it also becomes prohibitive to simulate large number of SEs.

A fast acceptance–rejection (AR) scheme [6] and an inverse scheme [7] that increase the performance of MC particle breakage algorithm have been designed and validated. The algorithm was mainly designed to evaluate the size class after breakage. The correctness of the proposed scheme was validated in comparison with analytical solutions [8]. The evaluation criterion was the computational efficiency of the particle breakage [9].

The stagnation of CPU clock speeds have driven researchers toward parallel architectures that provide massive computational power through separate processing units. To fully exploit the strengths of the multiprocessing architectures, parallel execution programs are required [9].

GPUs are extremely fast parallel microprocessors. A multiprocessor comprises many stream processors (the smallest computational units in the multiprocessor), which communicate with each other through shared memory [10]. The GPU can accelerate computations and applications by loading the code parts with high compute-loading [11].

This paper describes an algorithm based on MC methods implementing AR and inverse sampling on a GPU. A new parallel algorithm is introduced in which low-volume and

978-1-5090-6465-6/17/$31.00 ©2017 European Union

combined schemes are implemented in a time-driven PB-MC method.

II. Computational Procedure

Our test bed makes use of a NVDIA GeForce GTX 780 GPU programmed with CUDA. Three algorithms were compared: the number-based (NB) algorithm keeps the SEs constant, the volume-based (VB) one conserves the total volume, and the low-volume based (LV) one preferentially creates small-sized particles. These schemes were tested using 100 simulations with different random numbers, each simulation having 10^3 or even more or less SEs. For accuracy and efficiency, the time step of the simulation is shortened. In order to shorten the time step, we applied a time step tuning factor (αT) with 40 synchronization points (characteristics time). The algorithms were verified by comparisons with two analytical cases [8].

TABLE I. INITIAL SETTINGS FOR SIMULATIONS

Initial Settings for Rendering	**Value**
Simulation Entries (SEs)	10^5
Time Step Tuning Factor (α_T)	0.1
Initial (v_0)	$5.236e^{-7}$ m^3
Statistical Weight W_0	$1.0e^{14}$/SEs
Number of Synchronization points	40
Number of Simulations	100
Combined Scheme	VB 90% - LVB 10%

A monodisperse initial population of volume v_0 has been assumed. These mother particles are broken into daughter particles occupying lower size classes (fig.1). The simulation tracks the dimensionless particle mass, v/v_0.

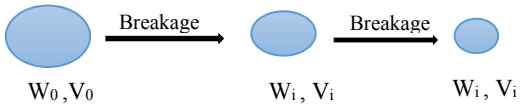

Fig. 1. *Breakage of a SE*

As shown in fig.2, all the necessary simulations are executed in parallel and the fundamental steps in the simulation algorithm are the MC-based AR and inverse sampling implementation.

In general, the MC method is a complex computational task because of its' computing cost and numerical complexity [10]. A comprehensive description of the implementation of the MC method on the GPU mediated by CUDA kernels is presented here. The GPU implementation provides a number of parallel threads which speed up the processing of the algorithm. Every thread performs a process.

A. The PB-MC simulation procedure is as follows:

- Initialization of the SEs properties on the CPU.
- Allocation of the memory on the GPU for the SEs initialized on CPU.
- Calculation of the breakage rate of each SE on the GPU.
- Calculation of the maximum breakage rate per simulation for calculating the time step Δt followed by the calculation of the simulation time on the GPU.
- Calculation of the breakage probability of the SE and generation of the random number r. This will involve breakage of the particle according to the settings defined for the scheme. This is followed by updating of SE properties such as volume and statistical weight.
- Update the time (characteristic time), and transfer of data to the CPU which checks whether the simulation has reached the end or not. If not, then the steps 4, 5 and 6 are repeated.
- Transfer of the results from the GPU back to CPU.

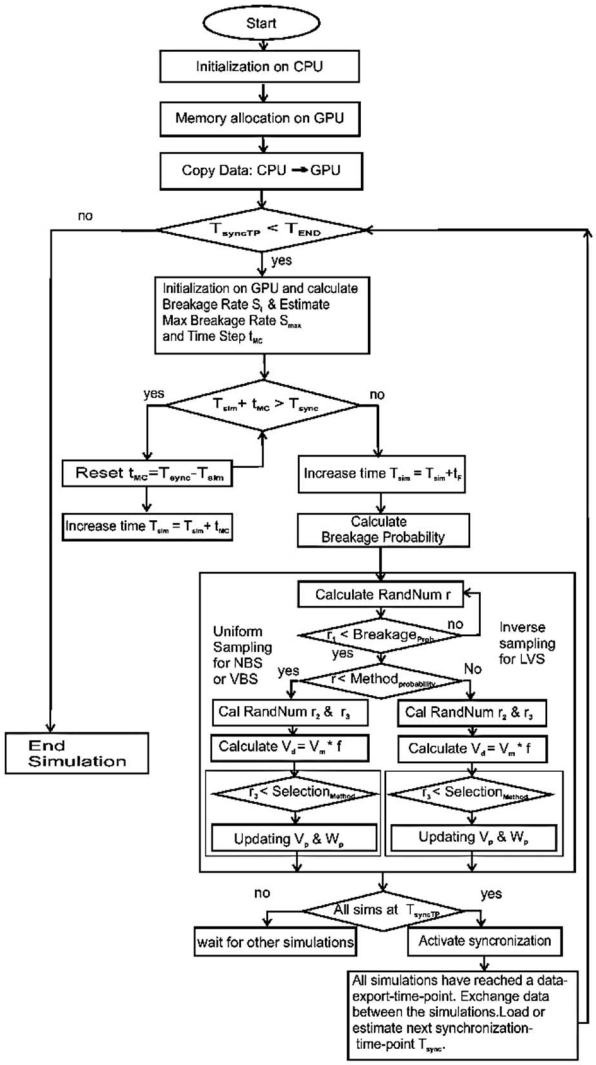

Fig. 2. *Flowchart of the parallel algorithm on GPU*

Out of the seven steps mentioned above, the steps 4 ,5 and 6 truly represents the inherent dynamics of the MC method for breakage and also gives a clear indication on whether the MC method is able to deliver the expected efficacy on the GPU. Every thread computed a breakage rate (S_i) , and the maximum breakage rate (S_{max}) was estimated by a parallel algorithm shown in the Appendix. The time step Δt of the simulation was calculated as given in equation 4 and tuned to maximize the accuracy of the algorithm;

$$\Delta t = \frac{1}{S_{max}} * \alpha_T \qquad (4)$$

For every time step with time step tuning factor α_T , a series of breakage events will be realized. The probability p_i of event i occurring in Δt is given by a Poisson distribution [12].

$$p_i = 1 - e^{(-v_i * \Delta t)} \qquad (5)$$

This CUDA implementation runs the computations on the GPU. At each of the 40 synchronization points, the algorithm checks whether all simulations have reached the synchronization point. When this occurs, the algorithm writes the raw data (raw results) to the CPU memory, and all simulations continue their processing until the synchronization endpoint is reached. The breakage algorithm begins and ends on the CPU, and the parallel processing is performed by the GPU.

B. Breaking and updating of SEs

Most of the processing time is spent at simulating the breakage of a SE and subsequently updating of values after breakage. This process is time-intensive because it uses two types of MC sampling (AR and Inverse) with both samplings using random numbers.

The algorithm initializes and sets the conditions for the process and as it is shown in the table 1 above, the final results of algorithm are based on a mixture of schemes found most suitable for accuracy of the results. The combination of 90% of the VB scheme and 10% of the LV scheme is implemented and validated with help of analytical solutions. During the invocation of the kernel, the routine controls the SEs to be constant and checks which simulations are active. With random numbers distributed between 0 and 1, a random number generated r_1 assists in determining whether a SE will break or not. If the breakage probability of the SE is greater than the random number generated, the SE will be selected for the breakage. When a SE is selected for breakage, the algorithm chooses the sampling for the breaking process either uniform sampling (AR sampling) or inverse sampling. A random number r ($0 < r < 1$) selects the breakage scheme (general, number based, volume based or low volume). The breakage function, which determines the particle size distribution, differs for each scheme.

If the value of r is less than a proposed value for combination of schemes, then the breakage is carried out according to the uniform sampling which describes the part of volume based scheme. In such conditions the SE breakage occurs through uniform sampling and if the generated random number r is greater than proposed value for combination of schemes then the breakage function performs the breakage of the SE by using inverse sampling.

After the selection of the sampling method, two random numbers are generated which are uniformly distributed random numbers r_2 and r_3 in the range [0, 1]. Here, r_2 calculates the daughter particle volume, and r_3 identifies the particle for breakage from the breakage function.

The general breakage scheme computes the probability distribution function for selecting the new volume, and the statistical weight of the simulation entry, which depends on the mother volume.

The last step of the algorithm updates the population by calculating the new statistical weight of the SE and storing it where the mother article was stored. The statistical weight is recalculated by the selected scheme.

III. Results

It was found that the GPU based MC algorithm for breakage is computationally efficient and faster than reported run times for CPU-based serial implementations of the constant-number and constant-volume MC algorithm [**5**]. The computational times are calculated for various time step tuning factors (α_T) [0.001, 0.003, 0.01, 0.03, 0.1,1 and 3]. A time-step tuning factor of 0.1 was found to lead to accurate results. In this test, 100 simulations were executed in parallel, and each simulation runs a mixture of 90% of the volume-based and 10% of low-volume schemes, which showed accurate results. These results are validated with analytical solution [8].

Fig.3 shows the error in the dimensionless simulated geometric mean of the diameter as function of dimensionless time. It can be seen that in most simulations the ratio increases with time, however the simulation with VB and 10^5 SEs it decreases slightly.

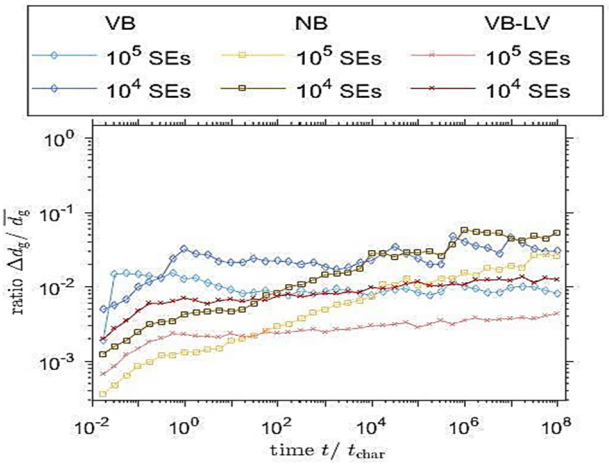

Fig. 3. *The statistical errors of the simulation for a tuning factor of 0.1.*

The simulations with VB and NB applying 10^4 SEs shows larger fluctuations. The ratio lies between 0.1 and 0.0001 with VB-LV and 10^5 SEs, showing smaller variations in time. The VB-LV simulations presented small fluctuations, while the VB-LV combination with 10^5 SEs presented the best results over time. In addition, all simulations presented smaller values of the ratio for larger values of the number of simulation entries.

The ratio that is shown in fig.4 represents the statistical error contribution in the geometric standard deviation of the PSD. It can be observed that in majority of the simulations the error increases with time, however in case of simulation using VB and 10^5 SEs, the error decreases slightly. The ratio lies between 0.1 and 0.001 with VB-LV and 10^5 SEs presenting the smaller variation, while VB with 10^4 and 10^5 SE presented the larger variation. Moreover, the VB, NB and VB-LV simulations presented smaller variations for larger values of the number of SEs. The NB simulations presented variations that increase with time and VB simulations presented the larger fluctuations. The VB-LV simulations presented low fluctuations and low variations, while the VB-LV 10^5 SEs presented the best results through time.

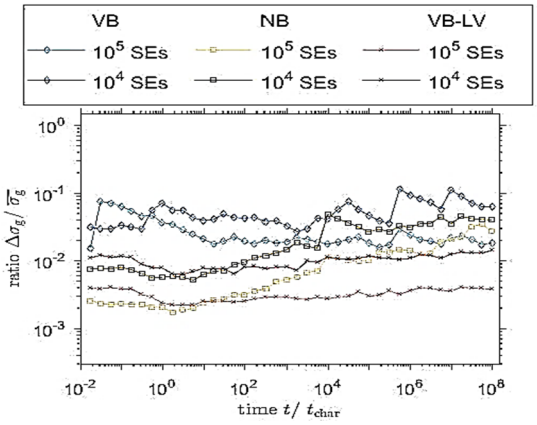

Fig. 4. *The statistical error for tuning factor 0.1 .*

Fig.5 shows the number of steps needed to perform the simulation in a time frame of $t\_{end} = 10^{13}$ for different time step tuning facotrs.

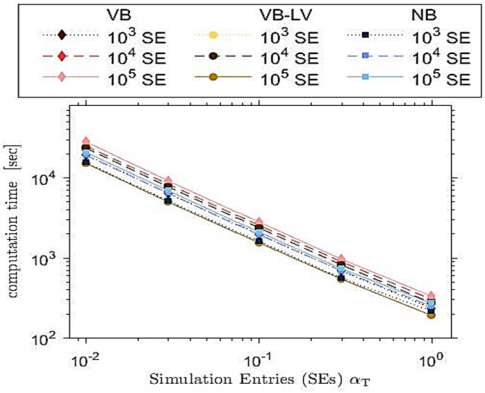

Fig. 5. *Time Steps for different tuning factors*

We measured the computation time for VB, NB, and VB-LV simulations with different number of SEs, using different time step tuning factors α_T. Fig.6 shows the computation time, the VB, NB, and VB-LV simulations presented very similar results. As expected, the larger number of entries greatly increases the computation time. Moreover, smaller tuning factors presented larger computation time, inversely proportional.

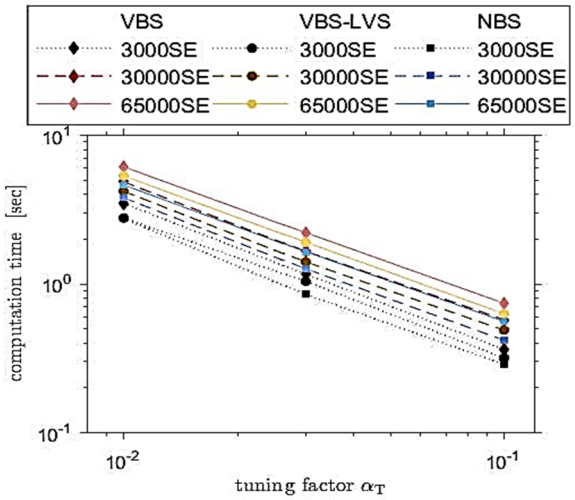

Fig. 6. *CPU time required to simulate a* $t\_{end} = 10^{13}$

In the proposed algorithm, different tuning factors have been tested and the results indicated that the tuning factor 0.1 is perfectly suitable for sufficient accuracy. This algorithm has multiple routines for performing tasks such as:

- Initialization of the SEs that needs few seconds
- Initialization of the simulation time before starting the rendering.
- Calculation of the initial breakage rate

The algorithm also tracks the simulation time and store it so that it can track and control the flow of simulations within the synchronization point. When the simulation is finished and is ready to move to the next synchronization point, the algorithm updates the number of active simulation. If simulations are not ready yet, then algorithm will wait until all the simulations have reached the synchronization point.

The random number is generated by a linear congruential generator (LCG) algorithm in two separate parts. In the first part, the initial seeds are calculated and set followed by the second part that involves calculating the random number on invocation. In order to save memory, the initial seed is stored on global memory and the random number generation calculation is done using a device function. When there is a need for a random number, the device function invokes and generates it. The algorithm does not store all random numbers on the memory. By using the device function, it is possible to save memory in the algorithm. In that event, the algorithm will utilize more time to calculate breakage of the SEs and update the new values following breakage. This is because the algorithm uses two types of sampling called the AR and

Inverse. The kernel (breakage of 50000 SEs and updating of the properties) is invoked 2118 times in 136.7ms with a time-step tuning factor of 0.1.

To estimate the maximal breakage rate (S_{max}) the algorithm needed to invoke two kernels. The 1st kernel estimates the max breakage rate per block and 2nd kernel estimates the max breakage rate per simulation. The GPU based algorithm can simulate a large number of SEs in a short time.

TABLE II. ALGORITHM KERNELS AND TIME NEEDED TO COMPLETE THE TASK FOR 50000 SEs, CASE1, TUNING FACTOR αT 0.1 AND VB-LV(90%-10%)

Computational Time of Algorithm Routines	
Breakage Algorithm Routines	***Time***
Initialization of Properties	41.984 µs
LCG Random Number Generation	3.438 ms
Calculation of Breakage Rate of SE	3.616 µs
Estimation of Max Breakage Rate	54.364 ms
Breakage of SEs and Update of the Properties	136.719 ms
Update ActiveSim Indices	3.258 ms

The NVIDIA visual profiler used to measure the performance and analysis of the algorithm. Table II shows the computational time of routines in the algorithm.

Conclusion

This paper presented an in-depth comparative evaluation of different types of general breakage schemes available today. The processing power of the CPUs is increasing rapidly, it has now become possible to carry out highly resource-intensive data processing using a parallel computing architecture. In this study, a robust framework is presented with the objective of enhancing and accelerating the highly intensive Population Balance Monte Carlo simulations of particle breakage. This objective is conceived upon the understanding that parallel computing can revolutionize the efficiency of central processing units. This study described the successful implementation and validation of a highly efficient acceptance–rejection (AR) and inverse scheme based algorithm leading to a substantial speed-up in the performance of particle breakage based on GPU. The results obtained from this study led to the observation that the GPU is capable of improving the performance of breakage algorithms to a large extent and allows simulation of a large number of SE. A new LVS scheme has been introduced in order to render also the smaller particle size classes with sufficient precision. The VBS-LVS scheme produces the most accurate results even for longer simulations time and for higher number of SEs. All schemes produce results acceptable for the initial stage but for longer times the NBS and VBS schemes are not producing accurate results, as they are not able to render small particles even for large number of SEs. The combined scheme can render the full spectrum of the particle size distribution.

As a scope for future research, the acceptance–rejection (AR) and inverse scheme based algorithm that is presented in this paper can be further augmented to accommodate processes such as particle breakage in multiple compartments which is essentially a single particle event in nature.

Acknowledgment

The support of the Deutsche Forschungsgemeinschaft DFG in the scope of the priority program SPP 1679 is greatly acknowledged. J. Devi has been supported by the Quaid-E-Awam University, Nawabshah Sindh (Pakistan) by a scholarship under the Faculty Development Program.

References

[1] P. J. Hill and K. M. Ng, "New discretization procedure for the breakage equation," *AIChE J.*, vol. 41, no. 5, pp. 1204–1216, 1995.
[2] B. K. Mishra, "Monte Carlo simulation of particle breakage process during grinding," *Powder Technol.*, vol. 110, no. 3, pp. 246–252, 2000.
[3] K. Lee and T. Matsoukas, "Simultaneous coagulation and break-up using constant-N Monte Carlo," *Powder Technol.*, vol. 110, no. 1–2, pp. 82–89, 2000.
[4] H. Zhao, A. Maisels, T. Matsoukas, and C. Zheng, "Analysis of four Monte Carlo methods for the solution of population balances in dispersed systems," *Powder Technol.*, vol. 173, no. 1, pp. 38–50, 2007.
[5] K. F. Lee, R. I. A. Patterson, W. Wagner, and M. Kraft, "Stochastic weighted particle methods for population balance equations with coagulation, fragmentation and spatial inhomogeneity," *J. Comput. Phys.*, vol. 303, no. 154, 2015.
[6] J. Wei and F. E. Kruis, "A GPU-based parallelized Monte-Carlo method for particle coagulation using an acceptance-rejection strategy," *Chem. Eng. Sci.*, vol. 104, pp. 451–459, 2013.
[7] A.D, Sathyagal, D.Ramkrishna, and G.Narsimhan, "Solution of inverse problems in population balances II: particle breakup," *Comput. Chem. Eng.*, vol. 19, no. 4, pp. 437–451, 1994.
[8] R. M. Ziff and E. D. McGrady, "The kinetics of cluster fragmentation and depolymerisation," *J. Phys. A. Math. Gen.*, vol. 18, no. 15, pp. 3027–3037, 1985.
[9] Z. Xu, H. Zhao, and C. Zheng, "Accelerating population balance-Monte Carlo simulation for coagulation dynamics from the Markov jump model, stochastic algorithm and GPU parallel computing," *J. Comput. Phys.*, vol. 281, pp. 844–863, 2015.
[10] X. Mei and X. Chu, "Dissecting GPU Memory Hierarchy through Microbenchmarking," *Tr*, pp. 1–14, 2015.
[11] B. Szilágyi and Z. K. Nagy, "Graphical Processing Unit (GPU) Acceleration for Numerical Solution of Population Balance Models Using High Resolution Finite Volume Algorithm," *Comput. Chem. Eng.*, vol. 91, pp. 167–181, 2016.
[12] Prasher C.L, *Crushing and Grinding Process Handbook*. Verlag Wiley & Son, Chichester, 1987.

Appendix

Pseudo code of the maximal breakage rate

SEs: Number of particles per simulation chosen for the simulation
ParticleIdxLimit: The number of total SEs time NoSim render by the algorithm
Tid_Num: Number of threads per block used in the algorithm
NoSim: Total number of simulations used in algorithm to render the SEs
Bid_PerSim: Number of blocks per simulation used in algorithm**.**
NoActiveSims: Simulations which are still and never reached on Synchronization point
Bbid_ActualSim: Active block of the active simulation
blockDim: Total thread numbers run in a block
girdDim: Total number of number of blocks in rendering
tid ThreadIdx

bid BlockIdx

Algorithm 1 to calculate the number of blocks per simulation used

1. Read SEs Tid_Num
2. **if** SEs % Tid_Num == 0 {
3. Block_Sim = $\frac{SEs}{Thread\_Num}$
4. **} else {**
5. Block_Sim = $\frac{SEs}{Thread\_Num}$ + 1
6. **}**

Algorithm 2 Pseudo Code for Simulation entry (SE) index Every SE have its own unique id and ParticleIdxLimit to **keep constant number of SEs.**

Calculate total number of blocks used for rendering

1. Bid_PerSim ← girdDim / NoActiveSims
2. Bbid_ActualSim ← bid % Bid_PerSim
3. SimNum ← ActiveSimNum
4. **ParticleIndex** ← (SimNum*SEs) + tid + ((Bbid_ActualSim) *blockDim
5. //Array start from index 0.To render the SEs of simulation number 1 we need to use SimNum+1
6. **ParticleIdxLimit** ← SEs*(SimNum+1)

Algorithm 3 to calculate the MaxBreakagerate in each simulation. MaxBreakagerate kernel consist of two sub kernels.

- **Find MaxBreakagerate per block in a simulation**
- **MaxBrekagerate from all blocks in a simulation**

Invoke 1st kernel MaxBreakagerate per block in a simulation

1. I**f ParticleIndex** < ParticleIdxLimit **{**
2. // calculate MaxBreakagerate in each block
3. **//** shared memory for the block
4. SharedMax_Rate_bid **[**Tid_Num**]**
5. //BreakageRate transfer from global in to shared memory.
6. Max_Rate_bid**[**Tid_Num**]** ← BreakageRate[Tid_Num]
7. // Do parallel compression in shared memory
8. for (I = 0; I <=8; i++) **{**
9. TidDivbleby ← 1<< i // 2^i // thread indices, which are divisible by TidDivbleby
10. distance ← 1<< (i-1) // 2^(i-1)//distance between two SEs
11. if (tid % TidDivbleby == 0) **{**
12. //evaluate Max_Rate_bid[tid] and Max_Rate_bid [tid+ distance] store higher value in Max_Rate_bid[tid]
13. Max_Rate_bid[tid] ← Max_Rate_bid[tid+ distance] or
14. Max_Rate_bid[tid]
15. Syncthreads
16. } else
17. //last block of the simulation
18. for (I = 0; I <=8; i++) **{**
19. TidDivbleby ← 1<< i // 2^i // thread indices, which are divisible by TidDivbleby
20. distance ← 1<< (i-1) //2^(i-1)//distance between two SEs
21. //check out of bounds and this out of bound check only true for last block
22. if (tid % TidDivbleby ==0) and (distance+ particleIndex < ParticleIdxLimit) **{**
23. //evaluate Max_Rate_bid[tid] and Max_Rate_bid [tid+ distance] store higher value in Max_Rate_bid[tid]
24. Max_Rate_bid[tid] ← Max_Rate_bid [tid+ distance] **or** Max_Rate_bid[tid]
25. Syncthreads
26. }
27. //the results are being stored from shared memory to Global
28. If tid ← 0 **{**
29. BreakRate [SimNum*Bid_PerSim + blockNoForActualSim] ← Max_Rate_bid [0]
30. } }}

Invoke 2nd kernel MaxBrekagerate from all blocks in a simulation

1. **If ParticleIndex** < ParticleIdxLimit {
2. **//** Calculate MaxBreakagerate in each block
3. Shared MaxRate_bid [Tid_Num**]** // shared memory for the block
4. Max_Rate_bid**[**Tid_Num**]** ← BreakRate[Tid_Num]
5. // BreakageRate from global memory transfer in to shared memory. do parallel compression in shared memory
6. for (I = 0; I <=8; i++) **{**
7. TidDivbleby ← 1<< i // 2^i // thread indices, which
8. are divisible by TidDivbleby
9. distance ← 1<< (i-1)//2^(i-1)//distance between two SEs
10. if (tid % TidDivbleby ==0 && ← distance + CmpareElementIdx < bid_ActualSim) **{**
11. //evaluate Max_Rate_bid[tid] and Max_Rate_bid [tid+ distance] store higher value in Max_Rate_bid[tid]
12. Max_Rate_bid[tid] ← Max_Rate_bid [tid+ distance] or
13. Max_Rate_bid[tid]
14. Syncthreads
15. }
16. else {
17. //last block of the simulation
18. for (I = 0; I <=8; i++) **{**
19. TidDivbleby ← 1<< i // 2^i // thread indices, which are divisible by TidDivbleby
20. distance ← 1<< (i-1)//2^(i-1) // distance between two SEs
21. // Check out of bound and this out of bound check only true for last block
22. if (tid % TidDivbleby ==0) **{**
23. //evaluate Max_Rate_bid[tid] and Max_Rate_bid [tid+ distance] store higher value in Max_Rate_bid[tid]
24. Max_Rate_bid[tid] ← Max_Rate_bid[tid+distance] or Max_Rate_bid[tid]
25. Syncthreads
26. }}
27. //the results are being stored from shared memory to global
28. If tid ← 0 **{**
29. MaxBreakRate[SimNum*Bid_PerSim+bid_ActualSim] ← Max_Rate_bid [0]
30. }}}