The Computational Efficiency of Monte Carlo Breakage of Articles using Serial and Parallel Processing: A Comparison

Jherna Devi[1]
Institute of Technology for Nanostructures (NST) and Center for Nano Integration Duisburg-Essen (CENIDE)
University Duisburg-Essen, Duisburg, D-47057, Germany
Department of Information Technology, Quaid-e-Awam University of Engineering Science & Technology (QUEST) Nawabshah, 67480, Sindh, Pakistan

Jagdesh Kumar[2]
School of Technology and Innovations
University of Vaasa, Finland
Department of Electrical Engineering, Quaid-e-Awam University of Engineering Science & Technology (QUEST) Nawabshah, 67480, Sindh, Pakistan

***Abstract*—This paper presents a GPU-based parallelized and a CPU-based serial Monte-Carlo method for breakage of a particle. We compare the efficiency of the graphic card's graphics processing unit (GPU) and the general-purpose central processing unit (CPU), in a simulation using Monte Carlo (MC) methods for processing the particle breakage. Three applications are used to compare the computational performance times, clock cycles and speedup factors, to find which platform is faster under which conditions. The architecture of the GPU is becoming increasingly programmable; it represents a potential speedup for many applications compared to the modern CPU. The objective of the paper is to compare the performance of the GPU and Intel Core i7-4790 multicore CPU. The implementation for the CPU was written in the C programming language, and the GPU implemented the kernel using Nvidia's CUDA (Compute Unified Device Architecture). This paper compares the computational times, clock cycles and the speedup factor for a GPU and a CPU, with various simulation settings such as the number of simulation entries (SEs), for a better understanding of the GPU and CPU computational efficiency. It has been found that the number of SEs directly affects the speedup factor.**

Keywords*—Breakage of particles; Central Processing Unit (CPU); Graphics Processing Unit (GPU); CUDA; computational time; clock cycle; speedup factor

I. INTRODUCTION

The breakage of particles is of interest in various fields of engineering and scientific research, including chemical engineering, aerosols, agriculture and medicine [1–3]. The population balance equation (PBE) provides a platform to develop the distributed phase. The PBE includes all the processes, such as nucleation, coagulation and breakage, that produce fragments and break simulation entries or parent particles from the population. Breakage is of major importance for understanding the behavior of, and dealing with, particle systems.

The particle breakage-population balance equation (BP-PBE), which characterizes the breakage dynamics in terms of the time evolution of the particle size distribution (PSD), is shown in Equation :

$$\frac{dn(v)}{dt} = -S(v) \cdot n(v) + \int_v^\infty n(v')S(v')b(v,v')dv' \quad (1)$$

where $n(v)dv$ is the concentration of particle sizes and $n(v)$ is the particle size distribution. The size range between v and $v + dv$ per unit volume, in time t, is defined by the BP-PBE Equation1.

where$S(v)$ is the rate at which a particle of size v breaks and the breakage function $b(v, v')$ describes the number fragment particles of size v resulting from the breakage of one parent particle of sizev'. The death term (first term on the RHS) of Equation (1) represents the deletion of particles of size v due to breakage into smaller fragments [4]. The birth term (second term on the RHS) of Equation (1) defines the addition of fragment particles with volume v due to breakage of particles with volume v', where $v < v'$. The breakage rate $S(v)$ and the breakage kernel $b(v, v')$ can be obtained by modelling and simulation or via experiments [1, 2].

A variety of methods can provide a solution for the PBE [5], such as the sectional method, the method of moments and Monte Carlo methods [1–3]. In this paper, the time-driven Monte Carlo (MC) approach has been implemented for solving the PBE for breakage on the CPU [3, 6] which uses serial processing, and on the GPU [1, 3], which facilitates parallel processing.

The modelling of breakage [6] of the particle is the process of making a model, which is a depiction of the structure and working of the particle breakage process [7]. A model is simple, but it represents similar to the real process. The main objective of the model of a process, for an analyst, is to estimate the outcome of variations in the process. Simulations of processes help to meet the specifications of a particular system, to reduce the chance of failure, to remove unexpected bottlenecks, to avoid overconsumption or underconsumption of resources and to optimize the performance of the process [7]. This paper focuses on: (1) Modelling and simulation of the breakage process serially and parallelly. (2) The differences between the GPU and CPU computational times and the speedup factor for a given breakage rate and breakage function.

The importance of this reported investigation is to provide awareness about parallel processing can save time and cost.

The remainder of this paper is organised as follows. In Section II we have presented the literature review of the Modelling and simulation of the Particle breakage using Monte Carlo methods, the serial and parallel processing differences are briefly summarized, according to different the hardware, software, architecture and the compilation of the algorithm. The discussion and the results are covered in Section III. The paper is concluded in Section IV and the future work highlighted in Section V.

II. Literature Review

A. Monte Carlo Simulation for Particle Breakage

MC methods (MC experiments) consist of a variety of computationally efficient algorithms that rely on recurring random sampling to obtain a numerical output. The MC method is widely used for modelling population balances (MC-PBs) [7]. The MC method has a discrete and stochastic nature [8, 9] and is suited to particle dynamics.

The MC simulation comprises of more than 1,000 iterations [10] or recalculations. During an MC simulation, the input probability distribution values are tested at random [11]. Every set of tests is called a recalculation or iteration, and the result of that test set is stored. Usually, Monte Carlo methods are classified by the time discretization system into event-driven Monte Carlo methods and time-driven Monte Carlo methods.

Event-driven MC [12] first estimates awaiting for time or time interval Δt among two consecutive events and then select the event randomly that occurs during time interval Δt.

Time-driven Monte Carlo [12] calculates a time step Δt, also known as a pre-specified time step, and considers all possible events that may happen within that time step.

The two most widely used sampling strategies for simulating breakage in Monte Carlo methods are the acceptance-rejection (AR) strategy [2, 10] and the inverse strategy [1, 12].

These sampling strategies have different ways of choosing the particle or SE to break randomly as shown in Fig. 1

Acceptance-rejection sampling is a more straightforward way of selecting the required particle: it randomly chooses a particle, calculates the breakage probability and checks to see whether or not to accept it. The Monte Carlo method is used to handle the particle breakage. Many attempts are often needed to find a suitable particle for breakage.

On the other hand, inverse sampling chooses a desirable particle by generating a random number between 0 and 1 and comparing it to a normalized value (R in [2, 3]) for the considered part of the PSD. Using the inverse scheme, one can always find a particle after a finite number of attempts (up to the total number of simulation particles (SEs)).

The MC method using AR sampling features inherent parallelism [2][13]. This is because the choices of a random particle (to be broken) via AR sampling are uncorrelated, and hence can be made in parallel.

This inherent feature of Monte Carlo AR sampling is easy to implement on a parallel architecture, such as the GPU [1, 2, 14]. GPUs can execute millions of lightweight threads in parallel and simultaneously.

B. Serial and Parallel Processing

Population balance equation processes, such as coagulation, breakage and nucleation, have been modelled on CPUs [8, 11] and GPUs [1, 2, 13]. It is worth emphasizing that an ideal grouping with high efficiency is important for PB-MC, because the increase in the number of simulation entries leads to an increase in the accuracy of the implemented process or algorithm, while the computing efficiency decreases. Computational power has increased over the last 10 years, but it is also important to increase computational efficiency for estimating particle dynamics. The MC simulation can be accelerated in two ways [15] as follows.

1) CPU parallel processing, via OpenMP (open multi-processing) and MPI (message passing interface) [15].

2) GPU parallel processing: processing via the GPU and CUDA [16].

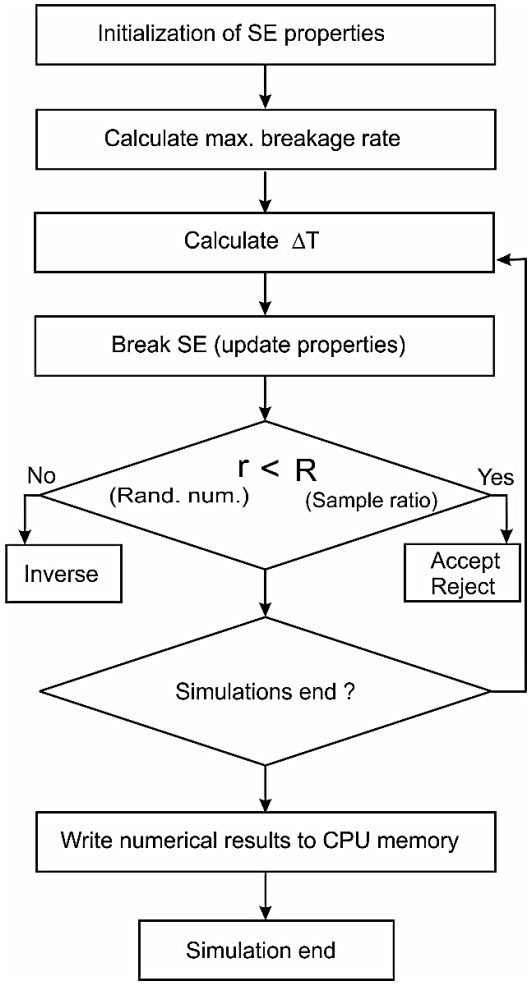

Fig. 1. Simulation Model.

In fact, the parallel computing of MC simulations utilizes more computer resources concurrently to decrease computational time. The CPU time increases linearly with computational complexity O(N) [1]. Monte Carlo acceptance-rejection sampling [16] has a significant effect on computational performance.

Recently, fast inverse and AR MC sampling on the GPU have been proposed [1, 2], to enhance the performance of MC methods for particle breakage.

Several nanotechnology applications have high inherent parallelism. The GPU is capable of high performance, as it supports a large number of cores, [16], and in many applications, GPU gets high performance. The clock speed of the serial processing [17] has driven the attention of the researchers to the parallel architecture,[16, 18–22] which is capable of providing tremendous computational efficiency [1, 2]. Usually, MC simulations require looping during the simulation of particle breakage events on CPU at the end of the simulation the computational cost increases [10]. The GPU and its advanced capabilities used for rendering of 3D games basically timeframe [23]. Now the GPU competencies are being coupled to accelerate computational workload by modelling of the complex and computationally expensive processes in different fields of scientific research.

1) Hardware: With regard to the hardware used for simulation, the CPU by Intel with four cores maximum performance, while the Nvidia GPU with 2,304 cores [24] can deliver 4,156 Gigaflops. The use of GPUs in high-performance computing also affects the computational cost. The GPU has two main characteristics [25] as follows: Table I. Allowing contact with the GPU's cores using CUDA programming. CUDA by Nvidia [23] is embedded in the standard C language. The function of a GPU is known as a kernel distribution function and is executed on the GPU cores [25]. 2. Achieving the distribution of the task via a kernel grid, where the kernel grid is split into blocks, with each block subdivided into several threads. The GPU has off-chip global memory and on-chip shared memory [26, 27]. This enables the programmer to select the type of memory used according to the requirement of the kernel [28].

2) Architecture: GPUs and CPUs were developed using different theories. CPUs can provide a prompt response time for individual tasks [29], whereas GPUs are built specifically for graphical applications and for rendering [27]. As stated above, the CPU used in modelling the breakage of a particle consists of four cores and a great deal of cache memory as shown in Fig. 2, which can deal with some application threads simultaneously.

The CPU memory is also known as host memory. The GPU, on the other hand, consists of hundreds of cores [14] and is capable of handling thousands of threads in parallel. The Nvidia graphics processing units are composed of streaming multiprocessors (SMs) [29].

Every SM contains a couple of cores. The GPUs have three types of memory: very fast and high-latency global memory, the on-chip low-latency shared the memory of each SM and the local private memory of each thread [29, 30]. The computational efficiency increases with more appropriate use of GPU memory. This parallel processing of the GPU can accelerate the algorithm by 60-70 times, compared with the serial processing of the CPU. The GPU is also more cost-efficient than an ordinary CPU. The general-purpose central processing unit is able to run several applications.

The graphics card cannot work alone, without CPU support. The GPU processes the task in parallel and the CPU controls it. The CPU invokes the task on the GPU, and the execution of the kernel is handled by the CUDA library, which runs on the central processing unit, while the GPU executes the kernel (functions). Both the CPU and the GPU work in parallel [29]. Here, the Peripheral Component Interconnect Express (PCIe) enables CPU-GPU data communication. The PCIe speed also limits the CPU-GPU data transfer time.

TABLE. I. HARDWARE USED FOR IMPLEMENTATION AND COMPRESSION OF THE BREAKAGE ALGORITHM WITH CUDA 8.0

Simulation Hardware		
Simulation Tool	CPU	GPU
Generation	4th	Kepler
Model	Intel Core i7-4790	GeForce GTX 780
Core	4	2,304
Clock Rate	3.60 GHz	1.006 GHz
Flops	44.24 Giga	4,156Giga

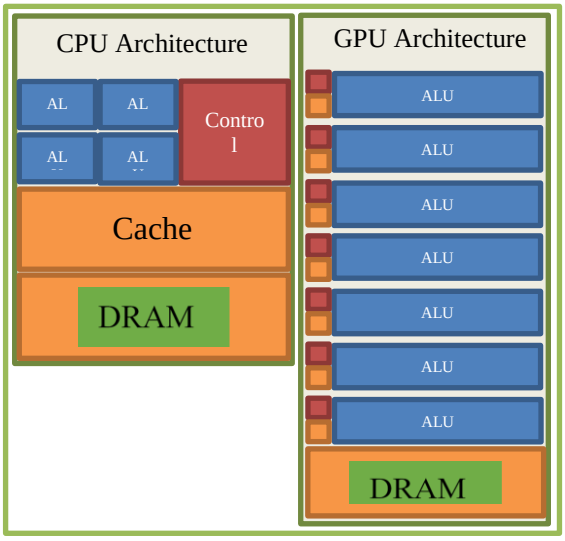

Fig. 2. The Architecture of the CPU and GPU.

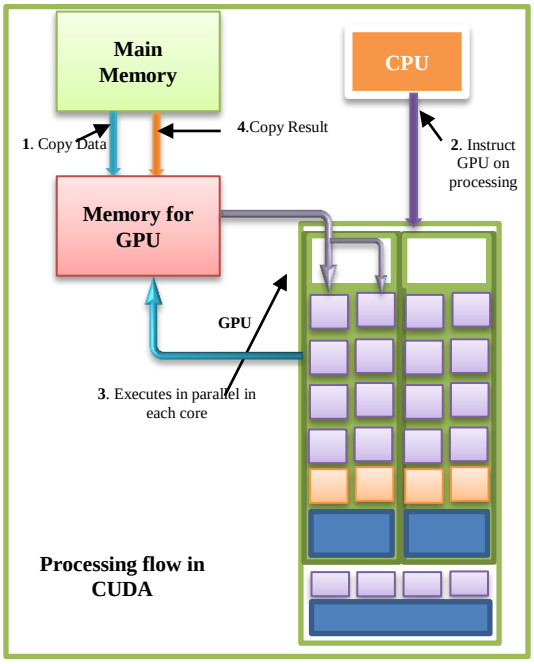

Fig. 3. The CUDA Processing Flow Copies Data from the main Memory to the GPU Memory and the CPU Instructs the GPU on the Process. The GPU Executes it with Each Core in Parallel, then Copies the Result from the GPU Memory to the main Memory.

3) CUDA and C programming: CUDA is a piece of software that deals with the GPU device, which can execute the computation in parallel. A GPU is a known device and it can execute lightweight threads in parallel through the CUDA programming as can be seen in Fig. 3. The GPU kernel, written in C, represents an extended programming language, compiling the device code and working on large quantities of data in parallel.

4) Compilation of the GPU and CPU code UDA: There are a few steps to compile the GPU CUDA program. Frist CUDA front end cudafe (cudafe.exe) splits the CUDA program into two parts: host and device code. The host code consists of a C/C++ program, and the device code uses GPU CUDA kernels. The host code part is compiled by the standard C compiler GCC and the device code is compiled using the CUDA compiler NVCC. The CUDA compiler NVCC converts intermediate code as in the type of assembly programming known as parallel thread execution (PTX). PTX is a low-level programming language used for GPUs by Nvidia as in Fig. 4. Furthermore, PTX code is interpreted into the binary code of the graphics processing unit cubin, which uses the ptxas compiler [30]. The compilation time also depends on the number of instructions and programs per kernel.

The increase in compilation and speed evaluation is directly proportional to the number of instructions [31] and programs per kernel.

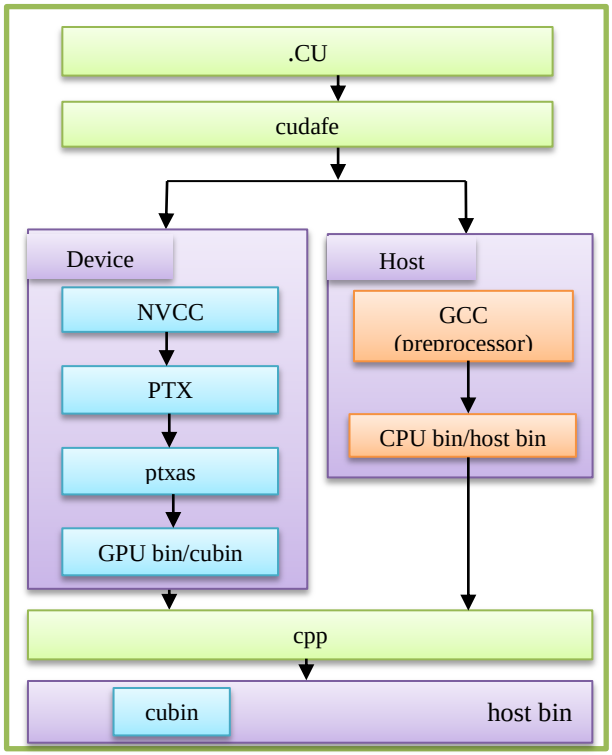

Fig. 4. CUDA Program Compilation.

III. Simulation Results and Discussion

A. Discussion

The process of breakage of particles has been simulated. Considering the performance on the GPU GeForce GTX 780, the most obvious and substantial outcome is the speedup factor of the breakage algorithm, compared to the performance on the CPU Intel Core i7-4790. The efficiency difference is greater than the difference between floating-point rates for the CPU and the GPU.

In this study, the particles are represented as simulation entries (SEs). For comparison, the results are obtained from calculations using the CPU and the GPU. A well-parallelized algorithm produced the same results as the sequential version.

The simulations were performed for various SEs on a 3.60-GHz i7 CPU and a GeForce GTX-780 GPU.

This detailed comparison of the computational efficiencies of the GPU and CPU can be used as guidance for choosing a hardware tool for the MC simulations. The results show the following.

The computational performance of GPU rendering is superior to that of CPU rendering for a single-cell scenario.

The acceptance-rejection sampling procedure is simple and easy to program. The GPU shows improved computational efficiency over the CPU.

The speedup ratio for the CPU and GPU demonstrates the difference between them. The speedup factor increases as the number of simulation entries increases. The GPU accelerates the rendering of complex processes, but the GPU acceleration power is limited by the quantity of input data allowed by the memory of the graphics card. However, this could be more than millions of SEs. For ideal performance, at least 100,000 particles are needed, with 500 simulations (iterations). The GPU requires full utilization in order to take advantage of the hardware's latency. It is not optimal to use the GPU for a low number of simulation entries, because most of the GPU time is spent on initializations for the calculations.

The computations are much faster on a GPU GeForce GTX 780 compared to a CPU. The GPU-based algorithm runs more than 65 times faster than our portable C implementation. As a result, there is an average speedup ratio of 65 compared to the scalar C++ code.

B. Results

1) Comparison of computational efficiency on CPU (serail) and GPU (Parallel): The computational efficiency of CPU- and GPU-based code has been compared. A difference in computing time can clearly be seen between the GPU and CPU, for the same settings. The GPU accelerates the rendering, as shown in Fig. 5. The computational time efficiency of the simulations depends on the number of SE settings.

The execution time depends on the number of simulations (iterations) for different SE settings and is calculated on the GPU. The GPU program performs the simulation for the breakage of a SE (particle), for various numbers of iterations. As shown in Fig. 5, the time required to simulate the process and compute the results for 500 simulations, for 10^5, 10^4, 10^3 and 10^2 SEs, is around 41, 4.1, 0.6 and 0.2s, respectively. If the number of simulations increases by 10 times, the computational time also increases by a factor of approximately 10, as predicted. The plot shows that for a small number of SEs, very less time is required to render and compute the results on the GPU. The computational time is directly proportional to the number of SEs and the number of simulations. For 150, 250, 350, 450 and 500 simulations, the time taken to render 10^5 SEs was 12.6, 20.8, 29, 37.3 and 41.4 seconds, respectively.

The computational time for the CPU-based algorithm executing the breakage process for various SE settings repeated for different numbers of simulations, is shown in Fig. 5. The time required to simulate and compute the results for 500 simulations, for 10^5, 10^4, 10^3 and 10^2 SEs, was around 2,600, 230, 22 and 2 s respectively. This clearly shows that the computational time increases by a factor of 10 with an increase in SEs by the same factor, as expected. The serial algorithm for a low number of SEs needs much less computational time to render and compute the results on the CPU and the computational time is directly proportional to the number of SEs.

Fig. 7 shows the computational time plotted logarithmically for different synchronization points of the simulations, for 10^5 SEs and 500 simulations (iterations or repetitions of a piece of code for statistical purposes). Each synchronization point (data export time point: after a certain time, the simulation computes the results and stores them in the CPU memory) is used to calculate the statistical results. Simple observation shows that the serial code (CPU) and parallel code (GPU) for the maximum number of SEs (10^5) take 2,600s and 41s, respectively.

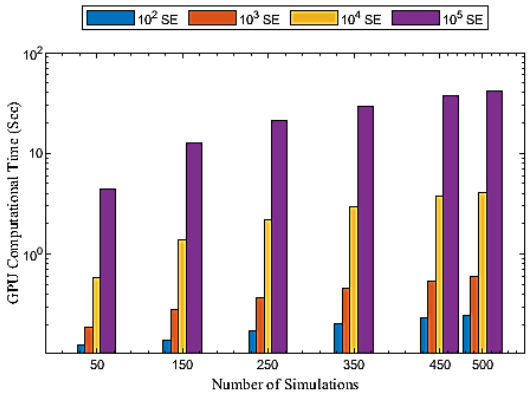

Fig. 5. GPU Program Computational Time for different SE Settings and different Numbers of Simulations.

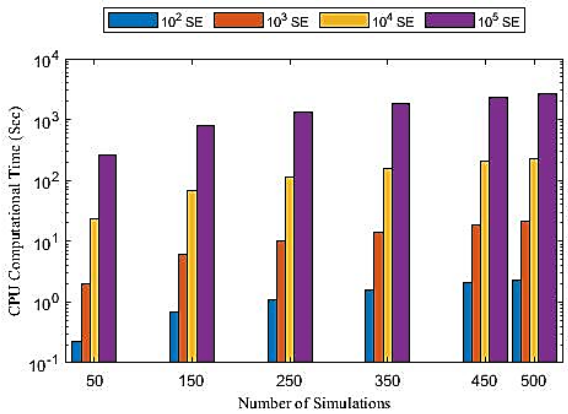

Fig. 6. CPU Program Computational Time for different SE Settings and different Numbers of Simulations.

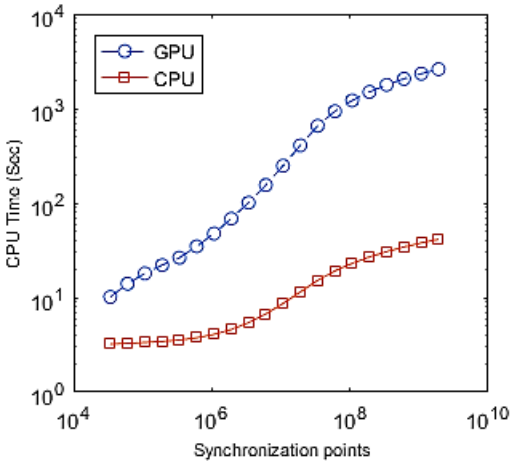

Fig. 7. Computational Time Synchronization Points Defined in the Simulations.

The clock register (clock ()) is used to obtain the time measurements as in equation (2):

$$\text{Computational time} = (\text{end\_clock}() - \text{start\_clock}()) \quad (2)$$

Simple observation shows that the serial code and parallel code for the maximum number of SEs (10^5) take 2,600 s and 41 s respectively. The performance difference between the CPU and GPU computational efficiencies is better illustrated in the speedup plot. To determine the time in seconds (s), the function clock () is divided by the CLOCKS_PER_SEC macro.

$$\text{Computational time} = \frac{(\text{end\_clock}() - \text{start\_clock}())}{\text{CLOCKS\_PER\_SEC}} \quad (3)$$

The macro CLOCKS_PER_SEC represents the number of clock ticks per second calculated as in equation (3).

2) Speedup factor: The speedup factor accelerating the particle dynamics simulation shows that the speed is higher than for the algorithm implemented in serial programming (CPU). The most common method for assessing the advantage of using the GPU is to calculate the computational times for the GPU and CPU [30, 32]. Speedup (S) is the computational time of the serial code (T_{CPU}) divided by the computational time of the parallel code (T_{GPU}), where both are computing the same result, i.e.,

$$S = \frac{T_{CPU}}{T_{GPU}} \quad (4)$$

To make a comparison, Fig. 5 and Fig. 6 show the CPU time for the CPU and GPU as a function of the simulation time. Simple examination shows that the CPU simulation is much slower than the GPU simulation. Usually, the speedup factor is calculated as in equation (4) and is plotted on the y-axis and the SEs or numbers of iterations (simulations) on the x-axis, as shown in Fig. 8. The plot shows that the GPU processing was 65 times faster than CPU processing. The GPU operated up to its maximum processing limit and achieved a speedup factor of 65. The GPU cannot operate any faster than this, for the considered particle breakage algorithm [2]. It does not matter if the number of simulation entries increases. For lower numbers of SEs, the GPU process is much faster than the CPU computations. To use the maximum power of the GPU, we used the maximum number of SEs (10^5), with 500 simulations. Fig. 8 shows the usage of the maximum computational power of the GPU. It is clear from Fig. 9 that the GPU reaches a certain limit, and the speedup factor is approximately constant after that limit. The number of SEs and the number of simulations no longer affects the efficiency of the GPU.

The speedup factor shows that GPU processing is much faster than for a normal CPU. We do, in fact, achieve a considerable speedup over the CPU case, although we should expect significant speedups (30 to 65 times), according to the GPU acceleration computation.

Fig. 8 shows the real speedup achieved using the graphics processing unit. There is a significant difference in the speedup factor with variations in the number of SEs. It increases to certain limit, depending on the memory of the graphics card. The speedup ratio, compared to the CPU for the GTX card, clearly shows the performance change between the CPU and the GPU.

The speedup plot depends on the SEs. There are no differences in computational efficiencies. If we consider only 1,000 SEs and 10 simulations (iterations) the total number of threads to be launched on the GPU is 10,000 (SEs x iterations). Alternatively, or we take 10000 SEs and 1 simulation. The computational time is calculated and found as the same time consumed by both settings. For the most significant number of SEs and simulations, a speedup factor of up to 65 was obtained.

3) Clock cycles: The clock cycle is the speed of the processor. It is the time taken to complete a full process [33].One can predict this from the speed of the processor [34]. The i7 CPU with a speed of 3.6 GHz is used as a hardware tool in this study. The speed of a processor is measured in Gigahertz (GHz).

This means that this CPU processor can perform 3,600,000,000 clock cycles per second. Fig. 9 shows that for all input SEs, GPUs show their advantages over CPUs, leveraging their parallel computing abilities. Specifically, for 10^5 input SEs and 500 simulations, 4.16×10^{10} cycles are needed by the GPU, as shown in Fig. 9, compared to 9.39×10^{12} cycles for the CPU. The major overhead for GPUs and CPUs arises from executing instructions, which depend on memory access. The serial and parallel algorithm clock cycles are summarized in Table II.

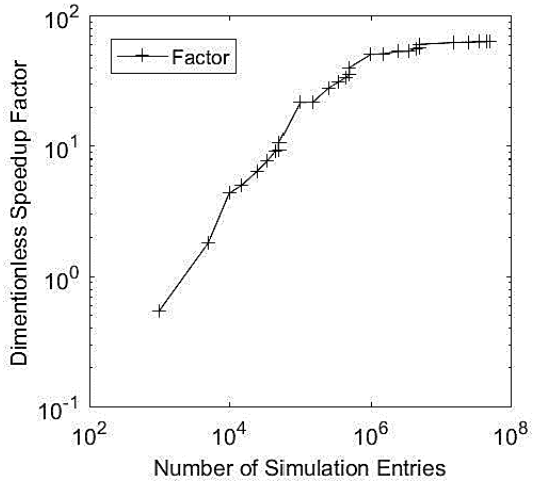

Fig. 8. The Dimensionless Speedup Factor for the Total SEs used for the Simulations. The Total Number of Simulation Entries was Simulated only once. No Iterations were Performed. Dimensionless Speedup Factor for the different SE and Iteration Settings. The Speedup is Plotted on the y-Axis and the x-Axis shows the Number of Iterations for the SEs.

TABLE. II. CLOCK CYCLES OF THE SERIAL AND PARALLEL IMPLEMENTED ALGORITHM

Iterations	CPU Clock Cycles				GPU Clock Cycles			
	10^2 SEs	10^3 SE	10^4 SEs	10^5 SEs	10^2 SEs	10^3 SEs	10^4 SEs	10^5 SEs
	10^{12}				10^{10}			
50	0.0008	0.0071	0.0823	0.9472	0.0126	0.0190	0.0578	0.4415
150	0.0025	0.0217	0.2467	2.8080	0.0139	0.0282	0.1379	1.2716
250	0.0039	0.0362	0.4115	4.6980	0.0173	0.0366	0.2180	2.0995
350	0.0056	0.0508	0.5638	6.5376	0.0206	0.0460	0.2977	2.9244
450	0.0075	0.0653	0.7513	8.3916	0.0231	0.0544	0.3768	3.7594
500	0.0081	0.0757	0.8273	9.3924	0.0243	0.0604	0.4137	4.1628

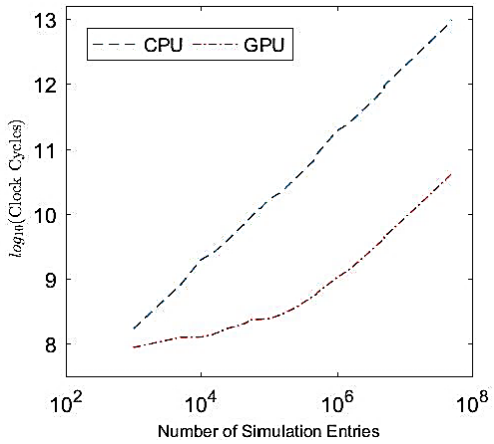

Fig. 9. The Clock Cycle for the Total SEs (SEs * NoSim = Number of Simulation Entries) used to Simulate the Breakage Process. Execution Cycles of the Two Versions of Hardware Tools used. The y-Axis Shows Clock Cycles for the GPU and CPU for Various SEs and the x-Axis shows the Number of Simulations. A Log Scale is used.

IV. CONCLUSIONS

We provided here a new background for modelling particle breakage using a graphics processing unit as a parallel environment and a CPU for serial processing. We performed simulations of the particle breakage process with different input SEs and simulation settings on a 3.60-GHz Intel Core i7 CPU and a GeForce GTX-780 GPU.

It was found that the GPU-based rendering is much faster than the CPU- based rendering. GPU saves time and produces fast results. Significant speedup was achieved for the graphics processor over the central processing unit, for the reported breakage algorithm (BP-PBE).

In the case of a stochastic particle model, the typical speedup is around 60-65 times, depending on the particle numbers (SEs) considered. Comparisons of CPU and GPU simulation results show that there are differences due to the different computational platforms. Significant differences between GPU and CPU results can be observed for the case of a large number of simulation entries. However, the monitored differences in the computational time for the simulation of particles on the GPU and CPU are of the order of the characteristic CPU time. This demonstrates that the GPU is a proficient and efficient tool for rendering parallel processes, such as modelling the breakage of particles.

The objective of this paper is to show the advantages of using the GPU for rendering the breakage of particles. A fast and efficient breakage algorithm is developed, and the performance improvement of the parallel algorithm on the GPU is observed and compared with the efficiency of CPU implementations. This study examined the computational efficiency of single-cell implementations on both the GPU and the CPU.

V. FUTURE WORK

In future, the authors are interested to implement breakage-coagulation algorithms to evaluate the efficiencies of various algorithms. Moreover, multiple-cell implementations could also be optimized to improve the performance of the breakage process. Each cell will have a different breakage rate, decreasing with time. Small particles take more time to break, and it can easily be predicted that if the breakage rate is reduced, then the computational time will increase, and the speedup factor will also be affected.

ACKNOWLEDGMENT

The support of the Quaid-E-Awam University, Nawabshah Sindh (Pakistan) by a scholarship under the Faculty Development Program is greatly acknowledged.

REFERENCES

[1] J. Devi and F. E. Kruis, “A fast Monte Carlo GPU based algorithm for particle breakage,” IEEE-CODIT 17, pp. 784–789, 2017.

[2] G. Kotalczyk, J. Devi, and F. E. Kruis, “A time-driven constant-number Monte Carlo method for the GPU-simulation of particle breakage based on weighted simulation particles,” Powder Technology, vol. 317, pp. 417–429, 2017.

[3] J. Devi, G. Kotalczyk, and F. E. Kruis, “Accuracy control in Monte Carlo simulations of particle breakage,” IJMIC, vol. 31, no. 3, p. 278, 2019.

[4] P. J. Hill and K. M. Ng, “New discretization procedure for the breakage equation,” AIChE J., vol. 41 No. 5, pp. 1204–1216, 1995.

[5] J. Kumar, G. Warnecke, M. Peglow, and S. Heinrich, “Comparison of numerical methods for solving population balance equations incorporating aggregation and breakage,” Powder Technology, vol. 189, no. 2, pp. 218–229, 2009.

[6] 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,” Journal of Computational Physics, vol. 303, pp. 1–18, 2015.

[7] W. K. Chan et al., Eds., Modeling as the practice of representation: Proceedings of the 2017 Winter Simulation Conference, December 3-6, 2017, Las Vegas, NV. Piscataway, NJ, Madison, WI: IEEE; Omnipress, 2017.

[8] 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 Technology, vol. 173, no. 1, pp. 38–50, 2007.

[9] F. E. Kruis, J. Wei, T. van der Zwaag, and S. 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," Chemical Engineering Science, vol. 70, pp. 109–120, 2012.

[10] J. Wei and F. E. Kruis, "A GPU-based parallelized Monte-Carlo method for particle coagulation using an acceptance–rejection strategy," Chemical Engineering Science, vol. 104, pp. 451–459, 2013.

[11] Z. Xu, H. Zhao, and C. Zheng, "Fast Monte Carlo simulation for particle coagulation in population balance," Journal of Aerosol Science, vol. 74, pp. 11–25, 2014.

[12] G. Kotalczyk and F. E. Kruis, "Fractional Monte Carlo time steps for the simulation of coagulation for parallelized flowsheet simulations," Chemical Engineering Research and Design, vol. 136, pp. 71–82, 2018.

[13] G. Kotalczyk and F. E. Kruis, "A Monte Carlo method for the simulation of coagulation and nucleation based on weighted particles and the concepts of stochastic resolution and merging," Journal of Computational Physics, vol. 340, pp. 276–296, 2017.

[14] NVIDIA Corporation (2013), "NVIDIA GeForce GTX 780 Specifications," https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780/specifications, 2013.

[15] Kruis, F.E: Zhao, H, J. Wei, and C. & Zheng, "A parallelized population balance-Monte Carlo method for diffusion and coagulation of nanoparticles," pp. 26–29, 2010.

[16] J. Wei and F. E. Kruis, "GPU-accelerated Monte Carlo simulation of particle coagulation based on the inverse method," Journal of Computational Physics, vol. 249, pp. 67–79, 2013.

[17] M. Woźniak, K. Kuźnik, M. Paszyński, V. M. Calo, and D. Pardo, "Computational cost estimates for parallel shared memory isogeometric multi-frontal solvers," Computers & Mathematics with Applications, vol. 67, no. 10, pp. 1864–1883, 2014.

[18] J. Wei, J. Wang, Q. H. Wu, J. Chen, and N. Jia, "Multisegment pulverised coal mill model and online implementation for condition monitoring," International Journal of Modelling, Identification and Control, vol. 1, no. 3, pp. 206–214, 2006.

[19] J. Wei, "A parallel Monte Carlo method for population balance modeling of particulate processes using bookkeeping strategy," Physica A: Statistical Mechanics and its Applications, vol. 402, pp. 186–197, 2014.

[20] 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," Journal of Computational Physics, vol. 281, pp. 844–863, 2015.

[21] S. Hong, T. Oguntebi, and K. Olukotun, Eds., Efficient Parallel Graph Exploration on Multi-Core CPU and GPU: IEEE, Oct. 2011.

[22] S. Ryoo et al., Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. New York, NY: ACM, 2008.

[23] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium ,Austin, TX, USA, pp. 134–144, 2011.

[24] J. P. Harvey, GPU acceleration of object classification algorithms using NVIDIA CUDA, 2009.

[25] V. W. Lee et al., "Debunking the 100X GPU vs. CPU myth: An Evaluation of Throughput Computing on CPU and GPU," SIGARCH Comput. Archit. News, vol. 38, no. 3, p. 451, 2010.

[26] X. Mei and X. Chu, "Dissecting GPU Memory Hierarchy Through Microbenchmarking," IEEE Trans. Parallel Distrib. Syst., vol. 28, no. 1, pp. 72–86, 2017.

[27] J. Nickolls and W. J. Dally, "The GPU Computing Era," IEEE Micro, vol. 30, no. 2, pp. 56–69, 2010.

[28] J. Wei, "A Fast Monte Carlo Method Based on an Acceptance-Rejection Scheme for Particle Coagulation," Aerosol Air Qual. Res., 2013.

[29] F. Molnár, T. Szakály, R. Mészáros, and I. Lagzi, "Air pollution modelling using a Graphics Processing Unit with CUDA," Computer Physics Communications, vol. 181, no. 1, pp. 105–112, 2010.

[30] C. P. Da Silva, D. M. Dias, C. Bentes, M. A. Pacheco, and L. F. Cupertino, "Evolving GPU machine code," Journal of Machine Learning Research, vol. 16, pp. 673–712, 2015.

[31] T. E. Lewis and G. D. Magoulas, Eds., Identifying similarities in TMBL programs with alignment to quicken their compilation for GPUs. New York, NY, USA ©2011: ACM, 2011.

[32] M. S. Friedrichs et al., "Accelerating molecular dynamic simulation on graphics processing units," (eng), Journal of computational chemistry, vol. 30, no. 6, pp. 864–872, 2009.

[33] H. McVeigh, "Factors influencing the utilisation of e-learning in post-registration nursing students," (eng), Nurse education today, vol. 29, no. 1, pp. 91–99, 2009.

[34] S. Abdel-Hafeez and A. Gordon-Ross, "An Efficient O(N) Comparison-Free Sorting Algorithm," IEEE Trans. VLSI Syst., vol. 25, no. 6, pp. 1930–1942, 2017.