Optimizing Particle Systems through CUDA-Assisted Multithreading

FADI N. SIBAI
Computer Engineering Dept.
Prince Mohammad Bin Fahd University
Al-Khobar
SAUDI ARABIA

ANDREW POTVIN, STEVEN NGO
Electrical Engineering & Computer Science Dept.
Wichita State University
Wichita, Kansas
U.S.A.

Abstract: - Particle systems present challenges that have warranted and attracted large amount of attention in both usage and optimization. The use of particle systems has driven complexity of simulation to greater needs of data size and accuracy. Optimization, thus, has become a moving target for researchers to reach. Studies show that multithreading has potential to make the simulation efficient while optimizing complex and data-intensive particle systems. The CUDA (Compute Unified Device Architecture) works with programming languages such as C/C++ and Python to make multithreaded parallel programming easier. This work serves to analyze particle systems using CUDA and provide an understanding about how various parameters such as the particle count and grid size influence the simulation performance. We improve the CUDA particles demo by Nvidia using our Python scripts and study the impact of particles and grids on execution time and throughput. Experimental results indicate that a required level of performance can be achieved by varying the number of particles, the size grids, and the orientation of grids as needed.

Key-Words: - Parallel computing methodologies, Parallel programming languages, Application programming interfaces

Received: September 15, 2020. Revised: October 17, 2020. Accepted: November 6, 2020.
Published: December 7, 2020.

1 Introduction

Particle systems provide simulation of real world events at both macro and micro scales. Particle systems have been used as a technique to simulate various effects, models, and game physics [1]. Particle systems were first introduced in a 1982 Start Trek movie. Particle systems represent collections of multitude of small particles, which move then die over time. Particle systems are most commonly used in video games but are also used in animations and arts [2]. Animators such as Pixar use it as an effective tool, which artists utilize to create realistic physical effects such as water, smoke and fire effects. In their movies, Pixar used millions of particles, where the more particles used, the closer it gets to real physics. These particle calculations, and keeping track of each individual particle out of the multitude of particles in the particle system, have formed a workload that may create issues with some computer hardware/software. Unity, a Danish-American video game software development company, uses particle systems when desiring to create some special physical effects [3]. Dynamic objects such as water are difficult to create through sprites or meshes; sprites and meshes are better for solid objects. Optimization of particle systems help creators save time by processing their work quicker without sacrificing accuracy.

Simulation assists individuals in sciences, entertainment, businesses, and research and creates a more accurate and reliable approximation of their respective needs [4]. The study of optimizing a system provides an insight of how the system works and the progression of advancements in the related fields. Optimization techniques make use of calculation reductions, data savings, and better approximations to result in a better and faster processing time. Particle Swarm Optimization [5] is an optimization method, based on particle systems, which iteratively attempts to improve a candidate solution with respect to a quality measure. In this work, we study one such optimization technique and examine its improvements, drawbacks, and implication for future advancements.

The CUDA (Compute Unified Device Architecture), a parallel computing platform and an Application Programming Interface (API), works with programming languages such as C/C++ and Python to make graphics processing unit (GPU)-assisted multithreading easier. OpenCL, another API, is designed to be an open platform agnostic standard. CUDA is a proprietary Nvidia property that performs multithreaded parallel programming on Nvidia GPU cards. Therefore, CUDA is expected to perform better because CUDA/GPU is a complete in-house

solution. In [14], Nvidia reports a top simulation performance of 64K colliding particles in 460 frames per second on an Nvidia Fermi GPU.

In this paper, we optimize a particle system on a CUDA platform using Python. The goal of this optimization effort is to further performance of particle systems and identify system properties such as particle count and grid size on the performance (processing time, and throughput) of particle systems.

This paper is organized as follows. Section 2 reviews related published articles. Section 3 describes the algorithm employed and the experiments conducted. The experimental results are presented and discussed in Section 4. Finally, Section 5 concludes the paper.

2 Literature Survey

Particle systems are used to model fuzzy objects to represent changes of form, motion, and dynamics [1, 3]; the particles of such a system may affect movie animations [2, 21] and 3D games [22].

Particle swarm optimization (PSO) is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995, inspired by social behavior of bird flocking or fish schooling [5]. Various applications of POS in fluid simulation, virtual reality, and computer animation show promises [6-8]. POS has gained distinction in the recent years due to its ease of application in unsupervised, complex problems that cannot be solved using traditional deterministic algorithms [9-10].

In the realm of deep learning, two popular deep learning APIs PyTorch and TensorFlow both only support CUDA for GPU acceleration [11]. Part of the reason for PyTorch only supporting CUDA in the mainline instead of OpenCL seems to be Advanced Micro Devices' (AMD) push for another computing language API: ROCm (Radeon Open Compute), which PyTorch AMD (an official branch of PyTorch) runs on. The classic XKCD comic regarding standards comes to mind. There was no official statement from Google on why TensorFlow only runs on CUDA for the GPU branch. A prior study utilizes the Adiabatic QUantum Algorithm (AQUA), which is "a Monte Carlo simulation of a quantum spin system written in C++" [12]. The Monte Carlo simulation itself is another naturally parallel problem as it is the averaging of many random guesses, so it is a kind of analog to the Mandelbrot set generator. It is found that CUDA performs better when transferring data to and from the GPU and that CUDA's kernel execution is also consistently faster than OpenCL, despite the two implementations running nearly identical code [12, 13].

Historically, various applications have been studied on shared memory multiprocessors, GPUs, and message passing systems, and their performance evaluated on these systems [17, 18, 19, 20, 25, 26, 27]. Uberflow [23] is a GPU-based particle engine featuring particle advection, sorting, and rendering. Drone [24] studied real-time particle systems on the GPU including storage requirements, integrating the motion equations with Euler integration and Runge-Kutta methods, saving the particle states including position and velocity using double buffering, and changing particle behaviors as a result of changing the velocity or position of a particle. Particle systems were also implemented on the GPU to simulate hundreds of flocking spaceships, featuring collision avoidance, separation, cohesion, and alignment. Cohesion drives the spaceships to the common (position) center, while alignment drives the spaceships to the common velocity, where "common" is calculated by averaging positions or velocities.

3 Algorithm and Experimentation

CUDA, developed by Nvidia, allows developers to access GPU cards for extensive parallelization of their code. Nvidia provides code examples making use of the CUDA interface. One such example is about particles in a controlled environment with various parameters [14]. In this work, we explore such a particle system using CUDA. Due to the nature of particles being separate entities operating on well-defined physical properties, parallelization is a prime candidate for optimization of any system. The algorithm being discussed will make use of this advantage for great performance increases in the simulation while making full use of the GPU. Parallelization will be handled through the organization of particles into grid spaces. These grid spaces will hold particles that will likely interact with each other, but likely not with others outside.

3.1 Algorithm Considered

The discussed and tested algorithm, using the aforementioned CUDA example code by Simon Green [14], provides a framework and environment for testing various configurations and needs with their resulting performance metrics. The system being discussed, while processing particles individually for final updates, particles are approximated together in grids in order to limit the needed number of comparisons and thus reduce time

to update. Only particles sharing the same grid space are checked against each other for collision [14, 15]. This collision check happens with a simple check comparing the distance between two particles against the sum of their radii, as given P1 and P2 in Fig. 1, where P1 and P2 are three-component vectors representing points in three dimensional space, with radii R1 and R2, respectively. In 3D space as commonly used in Engineering and Science disciplines, collision check between P1 and P2 is conducted using Equation (1), as follows.

$$\sqrt[2]{(P1x - P2x)^2 + (P1y - P2y)^2 + (P1z - P2z)^2} < (R1 + R2) \quad (1)$$

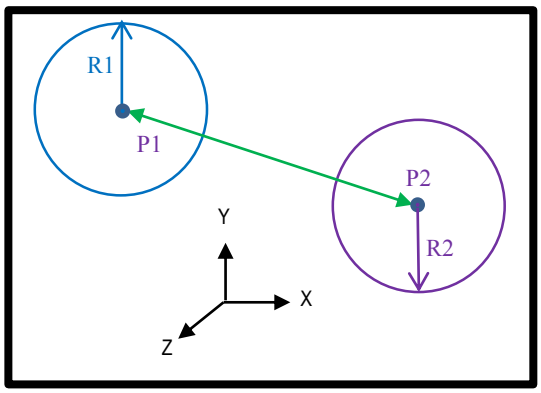

Fig. 1: Collision check between two particles

Equation (1) is derived from the Pythagorean theorem and the Euclidean distance formula [16]. A true state of Equation (1) indicates intersection (i.e., collision) of particles, while a false state indicates no collision. It is important to understand that an optimization to the formula in Equation (1) by removing the square root, as shown in Equation (2), is needed for better performance.

$$(P1x - P2x)^2 + (P1y - P2y)^2 + (P1z - P2z)^2 < (R1 + R2)^2 \quad (2)$$

Equation (2) provides a less computationally expensive early exit before performing more expensive collision resolution, especially when multithreaded parallel computing is used.

The simulation environment constructs a 2 x 2 x 2 cube area centered at the origin (0, 0, 0) and proceeds to fill it with the given number of particles in the specified arrangement. Arrangements can be in either an organized grid spaced evenly among the particles, or a random location for each particle within the cube. The cube is then divided in memory into grid cells where each grid cell can hold a small number of particles (the cell size should be ideally around the size of a few particles). Since each particle is in a three-dimensional space with a three-dimensional volume, a particle at minimum will be in one grid cell, but can be at maximum in eight grid cells (by being placed in their intersection points). Each grid cell can then keep a reference to each cell in their region and only process updates on these particles.

Updates are able to run in parallel and brought together at the end of the total batch set. Resolution of resulting collisions is handled in this step using digital elevation model (DEM) data. This allows particles to transfer "energy" between each other until the collisions in the local system have dissipated, and participants come to rest.

3.2 Experimental Details

For this study, the CUDA particles demo [14] by Nvidia is modified and managed to create a benchmark. The data measuring performance changes because of the multithreaded algorithm properties. The properties of the particle system configuration are modified to measure results such as processing time and throughput, and information about the algorithm.

The existing code in the particles demo provides a large amount of configuration and a platform for performing tests. However, we modify the code to allow for additional testing on the particles simulation. The main improvement is to allow for an extra command line argument for benchmarking that would set the starting organization of the particles. In the original demo, benchmark mode would default particle positions to an organized grid that chooses an equal distance from neighboring particles. The demo code is also modified to allow for a random positioning of particles. Experiment data is collected using an Nvidia GeForce 940MX GPU card with 384 CUDA cores.

3.2.1 Experiment Automation Script

In order to facilitate the increased needs and provide easy collection of data, a Python script is developed to wrap calls to the benchmark demo and to pass the correct configuration values. The Python script allows for:

- A variable number of particles to simulate in the system at one time.
- The grid size to divide into each axis. For example, a grid of size 24 would be 24 x 24 x 24 = 13824 total grid cubes.
- A type of simulation (GRID or RANDOM) to run. The type determines the starting orientation of particles in the system as discussed earlier.

- A number of iterations to run the simulation for the benchmark.
- A number of test runs with the same configuration values. The average values of each run are considered and calculated.

Parameters used in Test Run 1:
- Particle Amount: varied from 8192 to 65536
- Grid Size: 64
- Arrangement: GRID
- Iterations: 300
- Number of trials: 100

Parameters used in Test Run 2:
- Particle Amount: 32768
- Grid Size: varied from 32 to 256
- Arrangement: GRID
- Iterations: 300
- Number of trials: 100

Parameters used in Test Run 3:
- Particle Amount: varied from 8192 to 65536
- Grid Size: 64
- Arrangement: RANDOM
- Iterations: 300
- Number of trials: 100

Parameters used in Test Run 4:
- Particle Amount: 32768
- Grid Size: varied from 32 to 256
- Arrangement: RANDOM
- Iterations: 300
- Number of trials: 100

Test run 3 is an exact copy of Test run 1 except that the particle arrangement is RANDOM instead of GRID. Similarly, Test run 4 is an exact copy of Test run 2 except that the particle arrangement is RANDOM instead of GRID. Test runs 1 and 3 vary the particle count while keeping the grid size fixed at 64. Test runs 2 and 4 vary the grid size while keeping the particle count fixed at 32768.

3.2.2 Experiment Questions

The experiments conducted in this work are to find answers to four questions, each one is important for its own implication.

The first question: How the number of particles being simulated in the benchmark would affect the processing time? What implication would it have? This question has the possibility to display the improvements of multithreading in particles simulation.

The following hypothesis seek to give answers to the question: If the number of particles is increased, then the time to process all computations scales linearly. This is because, while multithreaded, all cores are used and threads are processed in all cores simultaneously.

The second question: What difference would the grid size make on the performance of the system? This question is very interesting because this is the main optimization that the demo focuses on. The grid size determines the number of checks against the neighboring particles. The grid size also determines how the particles are organized in memory. Thus, results could be used to comment on the data access, data manipulation, and overall performance.
The following hypothesis seeks to give answers to the question: If the grid size is increased, the time to process all computations decreases. This is because, in case of an increased grid size, there are fewer collisions to be removed from the system resulting in a breakdown of the optimization algorithm. In case of a decreased grid size, the time to process increases because there are more particle comparisons to be made each update.

The third question seeks to answer what impact would it have on the throughput of the system while changing the number of particles and the size of grid. The throughput measures the thousands of particles that must be processed per second in the simulation. A higher throughput should be able to give a better indication of the efficient processing of the system and efficient use of the hardware.
The following hypotheses seek to give answers to the question: If the particle count or grid size is varied, then the throughput will change linearly with the change of each variation, because the throughput is directly proportionate to the amount of processed particles (a larger grid size should have a higher particle count) at that time.

The fourth question is about the grid arrangement, whether the results would differ when conducted with a GRID or RANDOM orientation at the starting point. If the results differ, how? The GRID orientation should provide a consistent and overall stable result, while a RANDOM orientation of particles could result in more complex and unpredicted ways (when compared with the results from the GRID orientation).
The following hypothesis seek to give answers to the question: If the processing times due to a random start and a grid start are compared, then a random start should require more time to process, on average,

because a random start should create compact groups of particles that requires more iterations of the DEM solving than a grid start.

4 Experimental Results & Discussion

This section serves to present the experimental results and then attempts to reason why they occurred, and what could possibly be learned from the information. Numbers reported are averages of 100 runs or trials.

4.1 Number of Particles

The processing time obtained by changing the number of particles used in the simulation is as expected on the surface, i.e., the time increases on higher particle counts and decreases on lower ones, as shown in Fig. 2, for both GRID and RANDOM orientations. The grid size is fixed at 64. The GRID arrangement slightly outperforms the RANDOM arrangement with a particle count of 32768, or greater.

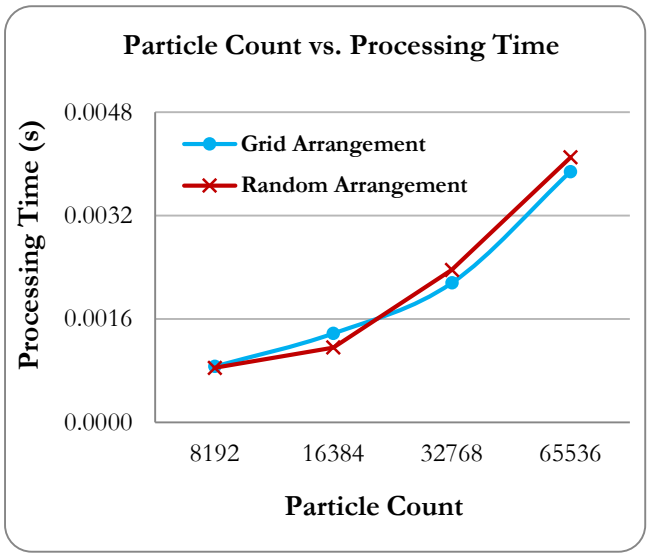

Fig. 2: Processing time due to particle count.

However, a deeper analysis shows that there must be some more information to extract from the results. As the particle size increases, the time to process does not linearly increase with the particle size. Therefore, in the case of ~16K particles, the time to process was 0.00137 sec. each update, but with ~32K particles, the time to process was only 0.00216 sec each update. A doubling of the particles would possibly indicate a doubling of the processing time (minus overhead of each update setup), but this was not the case, instead the result is about ~58% increase in processing time. This proportional gap between processing times for doubled particle size seems to get smaller as particle size increases. The interesting trend of this data seems to indicate that multiplicative growth in particle count causes only linear (and not multiplicative) growth in processing time.

Another interesting trend is the change in throughput at low particle count in a GRID or RANDOM arrangement. The grid size is fixed at 64. The throughput increases from a low point when the particle number is increased as shown in Fig. 3 (grid size =64). This may be due to more difficulty in parallelizing the lower particle amounts, or possibly a limitation of the algorithm used.

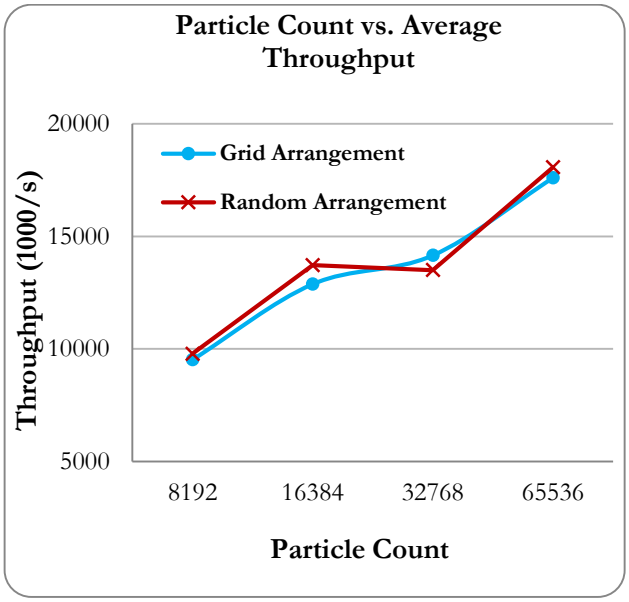

Fig. 3: Throughput due to particle count.

4.2 Size of Grid

The grid size, starting at 32 and moving up to 256, shows a decrease in processing time for both orientations, as shown in Fig. 4, with the particle count fixed at 32764. Before size 128, the processing time decreases sharply as the grid size increases. Past grid size 128, the processing time remains nearly the same, to a degree. Note that as the grid size increase from 32 to 256, the processing times drops by about 4% for the GRID arrangement case, and by approximately 2% for the RANDOM arrangement case.

The behavior of the processing time with the grid size 128 or higher could be due to the higher grid size spreading the particles away from each other and thereby reducing the number of collisions computed, as only particles sharing the same grid space are checked against each other for collision. When the particles are positioned far from each other, they are also checked for collision. However, given that the

particles are far from each other, computations related to collisions, such as updating the velocities and direction of colliding particles, are skipped, thereby reducing the execution time.

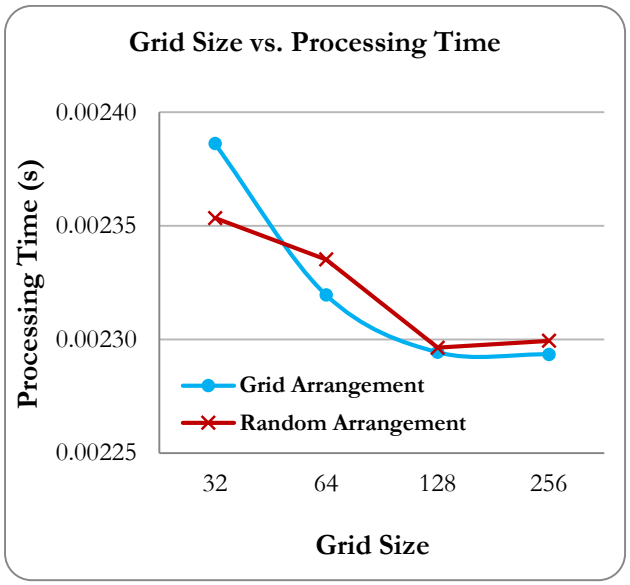

Fig. 4: Processing time due to grid size.

The grid size to throughput results are demonstrated in Fig. 5. The plots of Fig. 5 were obtained with the particle count fixed at 32764. In the GRID starting orientation, the throughput flattens to a consistent value for the higher grid sizes (128 and above). In case of the RANDOM orientation, the throughput increases steadily with the grid size as would have been more expected. These results can be explained by the predictable nature of the grid resulting in even distribution of particles, while random would result in not so even distribution of particles in the grids, thus a less optimal distribution of particles.

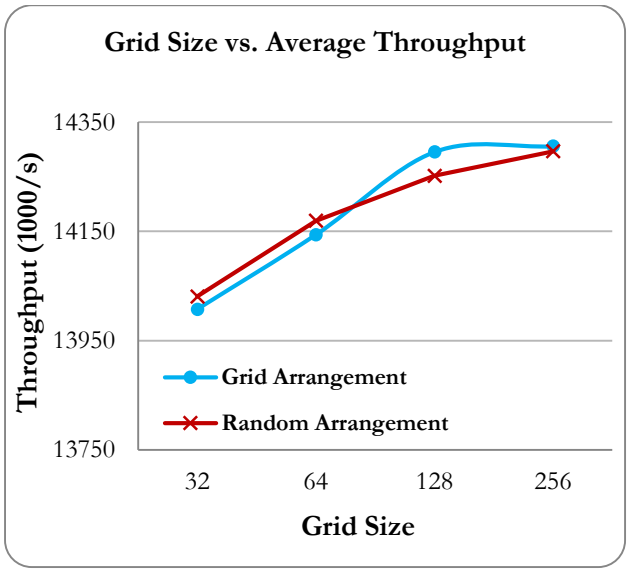

Fig. 5: Throughput due to grid size.

Moreover, there seems to be higher parallelism taking place in the GPU with larger grid sizes resulting in lower processing time and higher throughput, until the grid size reaches 128-256 at which point the number of particles processed per second starts to saturate.

4.3 Orientation of Particles

In the experiments, the particles' starting orientation is varied between GRID and RANDOM arrangements. It is observed, in Fig. 2, that the processing time consistently increases when the particle count increases from ~8K to ~64K for both orientations. However, for both orientations, the processing time decreases sharply when the grid size increases from 32 to 128 and remains about the same for higher grid sizes as shown in Fig. 3.

From Figs. 3 and 5, the average throughput increases when either the particle count or the grid size increases, as expected. However, for the GRID arrangement, the average throughput seems to even out at grid size 128 and higher as illustrated in Fig. 5. Although more parallelism takes place at higher grid sizes, the fixed particle count dilutes the number of particles processed in parallel resulting in the flattening of the throughput curve starting with a grid size of 128.

5 Conclusion

Particle simulations are important for many modern applications including scientific research and exploration, where the data size and complexity is ever growing. As a result, particle simulations require constant improvement through more accurate simulation and higher particle count. The need for improved simulation has led to a constant demand for optimization of both hardware and software. Studies show that particle placement and organization in memory form one such optimization area that can provide large performance improvement. The improvement has shown the potential of parallel computing in the simulation for less time and more throughput.

In this work, we study a particle system using CUDA Python that provides an understanding about how various parameters such as the particle count and grid impact on the simulation performance. According to the experimental results, both the processing time and throughput increase when the particle count increases from ~8K to ~64K. The uses of grids in particle optimization may reduce the processing time by checking and processing particle collisions against the immediately near ones. For a required performance, the grid size and arrangement can be

varied as needed depending on particle properties such as radius and collision response.

We plan to study the impact of particle configuration variance on performance in one of our next endeavors.

References:

[1] William Reeves, Particle Systems — a Technique for Modeling a Class of Fuzzy Objects, *ACM Trans. Graphics,* Vol. 2, No. 2, 1983, pp. 91-108.

[2] Jenny Green, How Particles Impact Movie Animation, 2017, https://obportland.org/how-particles-impact-movie-animation/

[3] Unity Technologies, *Introduction to Particle Systems*, 2020, https://learn.unity.com/tutorial/introduction-to-particle-systems#

[4] Adam Augustyn, Patricia Bauer, Emily Rodriguez, et al, Simulation: Scientific Method, 2016, https://www.britannica.com/science/simulation

[5] James Kennedy, Russell Eberhart, Particle swarm optimization. *Proceedings of the IEEE International Conference on Neural Networks (ICNN'95)*, Perth, Australia, 1995, doi:10.1109/ICNN.1995.488968

[6] Matthias Müller, David Charypar, and Markus Gross, Particle-Based Fluid Simulation for Interactive Applications, *Proceedings of 2003 ACM SIGGRAPH Symposium on Computer Animation*, 2003, pp. 154–159.

[7] Matthias Müller, Bruno Heidelberger, Marcus Hennix, John Ratcliff, Position Based Dynamics, *Workshop in Virtual Reality Interactions and Physical Simulation (VRIPHYS),* 2006, pp. 109-118.

[8] Erin Hastings, Ratan Guha, and Kenneth Stanley, Interactive Evolution of Particle Systems for Computer Graphics and Animation, *IEEE Transactions on Evolutionary Computation*, Vol. 13, No. 2, 2009, pp. 418-432, doi: 10.1109/TEVC.2008.2004261

[9] Yudong Zhang, Shuihua Wang, and Genlin Ji, A Comprehensive Survey on Particle Swarm Optimization Algorithm and Its Applications, *Mathematical Problems in Engineering*, Hindawi, 2015, https://doi.org/10.1155/2015/931256

[10] Saptarshi Sengupta, Sanchita Basak, and Richard Peters II, Particle Swarm Optimization: A survey of historical and recent developments with hybridization perspectives. *J. Machine Learning and Knowledge Extraction*, Vol. 1, No. 1, 2019, pp. 157-191. https://doi.org/10.3390/make1010010

[11] Nikolya Dimolarov, On the state of Deep Learning outside of CUDA's walled garden, 2019, https://towardsdatascience.com/on-the-state-of-deep-learning-outside-of-cudas-walled-garden-d88c8bbb4342

[12] Kamran Karimi, Neil Dickson, and Firas Hamze. A Performance Comparison of CUDA and OpenCL, 2011, https://arxiv.org/vc/arxiv/papers/ 1005/1005.2581v1.pdf

[13] Ayaz Khan, Mayez Al-Mouhamed, Muhammed Al-Mulhem, Adel Ahmed, RT-CUDA: A Software Tool for CUDA Code Restructuring, *Int. J. Parallel Prog*., Vol. 45, 2016, pp. 551–594. https://doi.org/10.1007/s10766-016-0433-6

[14] Simon Green, *Particle Simulation Using CUDA*, Nvidia, 2010, http://developer.download.nvidia.com/assets/cuda/files/particles.pdf

[15] Mikael Kalms, *High-performance particle simulation using CUDA*, Department of Electrical Engineering, Linkoping University. 2015, https://liu.diva-portal.org/smash/get/diva2:816727/FULLTEXT01.pdf

[16] Eric Weisstein, Distance Equations, Wolfram Resource, 2020, https://mathworld.wolfram.com/ Distance.html

[17] A Asaduzzaman, Fadi N. Sibai, H. ElSayed, Performance and Power Comparisons of MPI vs PTHREAD Implementations on Multicore Systems, *Proc. 9th IEEE Int. Conference on Innovations in Information Technology (IIT'13),* 2013, pp. 1-6.

[18] F. N. Sibai, S. Mohammad, H. Kidwai, B. Qamar, F. Awad, Parallel Implementation and Performance Analysis of a 3D Oil Reservoir Data Visualization Tool on the Cell Broadband Engine and CUDA GPU, *Proc. 14th IEEE Int. Conference on High Performance Computing and Communications (HPCC-12)*, 2012, pp. 970-975.

[19] F. N. Sibai, A. El-Moursy, Performance evaluation and comparison of parallel conjugate gradient on modern multi-core accelerator and massively parallel systems, *International Journal of Parallel, Emergent and Distributed Systems*, Vol. 29, No. 1, 2014, pp. 38-67. https://www.tandfonline.com/doi/pdf/10.1080/17445760.2012.762774

[20] F. N. Sibai, Performance analysis and workload characterization of the 3DMark05 benchmark on modern parallel computer platforms, *ACM*

SIGARCH Computer Architecture News, Vol. 35, No. 3, 2007, pp. 44-52.

[21] B. Reeves, K. Sims, Particle Systems and Artificial Life, *History of Computer Graphics and Animation*, Wayne Carlson (Ed.)., https://ohiostate.pressbooks.pub/graphicshistory/chapter/19-1-particle-systems-and-artificial-life/

[22] Iain Cantlay, High-Speed Off-Screen Particles, *GPU GEMS 3*, nVIDIA, H. Nguyen (Ed.), 2007, https://developer.nvidia.com/gpugems/gpugems3/part-iv-image-effects/chapter-23-high-speed-screen-particles

[23] Peter Kipfer, Mark Segal, Rudiger Westermann, UberFlow: *A GPU-based Particle Engine*, *Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS Symposium on Graphics Hardware*, Grenoble, France, 2004, pp. 29-30, https://pdfs.semanticscholar.org/9129/ef791f8cb29c0cec6a11c7d387207e93fce3.pdf

[24] Shannon Drone, Real-Time Particle Systems on the GPU in Dynamic Environments, *Proceedings of ACM SIGGRAPH'07*, 2007, pp. 80-96, https://doi.org/10.1145/1281500.1281670

[25] Hashir Kidwai, F. N. Sibai, T. Rabie, "Image Magnification and Reduction Using High Order Filtering on the Cell Broadband Engine," *Proc. 5th IEEE International Multi-Conference on Systems, Signals and Devices (SSD'08)*, Amman, Jordan, July 2008, pp. 1-5.

[26] A. Asaduzzaman, F. N. Sibai, H. ElSayed, "'Performance and Power Comparisons of MPI vs PTHREAD Implementations on Multicore Systems," *Proc. 9th IEEE Int. Conference on Innovations in Information Technology (IIT'13),* Al Ain UAE, 2013, 6 pages.

[27] F. N. Sibai, "Evaluating the Performance of Single and Multiple Core Processors with PCMark®05 and Benchmark Analysis," *ACM Performance Evaluation Review*, Vol. 35, No. 4, March 2008, pp. 62-71.

Contribution of individual authors to the creation of a scientific article (ghostwriting policy)

Fadi Sibai was responsible for the general theme and editing.
Andrew Potvin and Steven Ngo were responsible for the programming and obtaining the results.