# Enhancing Performance of Simulations using GPGPU

Shadi Alawneh and Dennis Peters
*Electrical and Computer Engineering*
*Faculty of Engineering and Applied Science*
*Memorial University*
{*shadi.alawneh, dpeters*}*@mun.ca*

***Abstract*— General Purpose GPU computing, or GPGPU, is the use of a GPU (graphics processing unit) to do general purpose scientific and engineering computing. The model for GPU computing is to use a CPU and GPU together in a heterogeneous co-processing computing platform. The sequential part of the application runs on the CPU and the computationally-intensive part is accelerated by the GPU. From the users perspective, the application just runs faster because it is using the high-performance of the GPU to boost performance. We have applied this technique to some sub-problems that form part of an ice-floe simulation problem and conducted an experiment to measure the performance of the GPU with respect to the CPU. The experiment consists of implementing a serial and parallel algorithm to detect and locate the intersection between polygons. We run the serial and parallel algorithms on several different sets of polygons and compare the performance.**

***Index Terms*— GPGPU, CUDA.**

## I. Introduction

The Sustainable Technology for Polar Ships and Structures (referred to as STePSS or STePS$^2$)[1] project supports sustainable development of polar regions by developing direct design tools for polar ships and offshore structures. Direct design improves on traditional design methods by calculating loads and responses against defined performance criteria. The project goal is to increase the understanding of interactions between ice and steel structures such as ships and oil rigs. The project began in July 2009 and has a duration of five years. It takes place at the St. John's campus of Memorial University of Newfoundland and is funded by government and private sector partners. The deliverables of the project include a numerical model which accurately handles collision scenarios between ice and steel structures. We are using General Purpose GPU computing, or GPGPU,[1], [2] to implement some of the numerical models in this project.

"Commodity computer graphics chips, known generically as Graphics Processing Units or GPUs, are probably today's most powerful computational hardware for the dollar. Researchers and developers have become interested in harnessing this power for general-purpose computing, an effort known collectively as GPGPU (for 'General-Purpose computing on the GPU')."[2] GPUs are particularly attractive for many numerical problems, not only because they provide tremendous computational power at a very low cost, but also because this power/cost ratio is increasing much faster than for traditional CPUs. A reason for this is a fundamental architectural difference: CPUs are optimized for high performance on sequential code, with many transistors dedicated to extracting instruction-level parallelism with techniques such as branch prediction and out-of-order execution. On the other hand, the highly data-parallel nature of graphics computations enables GPUs to use additional transistors more directly for computation, achieving higher arithmetic intensity with the same transistor count.[2] Many other computations found in modelling and simulation problems are also highly data-parallel and therefore can take advantage of this specialized processing power.

Hence, in this research we are trying to use the benefit of the high performance of the GPU to implement fast algorithms that can simulate ice-ice and ice-structure interactions in a very short time. In this paper, we present some initial results of measuring the performance of the GPU with respect to the CPU through an experiment consisting of implementing both serial and parallel algorithms to detect and locate the intersection between polygons and running both algorithms on several different sets of polygons to compare the performance.

### A. Ice Floe Simulation

The particular problem that we are investigating is to simulate the behaviour of floating ice floes (pack ice, see Fig. 1) as they move under the influence of currents and wind and interact with land and other structures, possibly breaking up in the process. In a two-dimensional model, we model the floes as convex polygons and perform a discrete time simulation of the behaviour of these objects. One of the steps in the simulation, then, is to compute the points of contact between floes in the ice field.

Fig. 1. Ice Floe[3]

[1] http://www.engr.mun.ca/steps2/index.php

Another deliverable of this project is the Ice Simulation Viewer, which is being developed to display the data produced by the simulation. This program displays frames of ice field data sequentially to provide its user with a video of a simulation of the field. It is currently used by the STePS[2] software team to help determine the validity of the data calculated by the simulation and will eventually be used to present results to project partners. The Ice Simulation Viewer is being developed in C++ using the Qt [4] user interface framework.

## II. Methodology

### A. Stream Processing

The basic programming model of traditional GPGPU is stream processing, which is closely related to SIMD[2]. A uniform set of data that can be operated in parallel is called a stream. The stream is processed by a series of instructions, called a kernel [5]. Stream processing is a very simple and restricted form of parallel processing that avoids the need for explicit synchronization and communication management. It is especially designed for algorithms that require significant numerical processing over large sets of similar data (data parallelism) and where computations for one part of the data only depend on 'nearby' data elements. In the case of data dependencies, recursion or random memory accesses stream processing becomes not reasonable [5], [6]. Computer graphics processing is very suitable for this, where vertices, fragments and pixel can be processed independently of each other, with clearly defined directions and address spaces for memory accesses. The stream processing programming model allows for more throughput oriented processor architectures. For example, without data dependencies caches can be reduced in size and the transistors can be used for ALUs instead. Fig. 2 shows a simple model of a modern CPU and a GPU. The CPU uses a high proportion of its transistors for controls and caches while the GPU uses them for computation (ALUs).

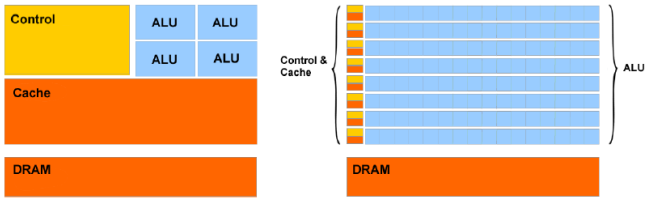


Fig. 2. Simple comparison of a CPU and a GPU [7]

### B. CUDA

Compute Unified Device Architecture (CUDA) is a comprehensive software and hardware architecture for GPGPU that was developed and released by Nvidia in 2007. It is Nvidia's move into GPGPU and High-Performance Computing (HPC), combining huge programmability, performance, and ease of use. A major design goal of CUDA is to support heterogeneous computations in a sense that serial parts of an application are executed on the CPU and parallel parts on the GPU[8]. An general overview of CUDA is illustrated in Fig. 3.

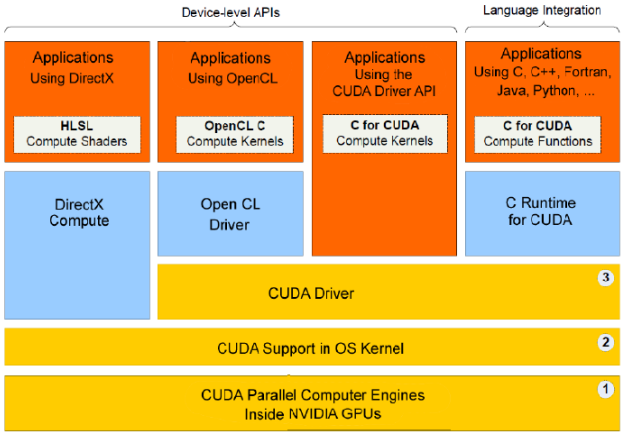


Fig. 3. CUDA overview [9]

Nowadays, there are two distinct types of programming interfaces supported by CUDA. The first type is using the device level APIs (left part of Fig. 3) we could use the new GPGPU standard DirectX Compute by using the high level shader language (HLSL) to implement compute shaders. The second standard is OpenCL which is created by the Khronos Group (as is OpenGL). OpenCL kernels are written in OpenCL C. The two approaches don't depend on the GPU hardware hence they can used to GPUs from different vendors. In addition to that, there is a third device-level approach through low-level CUDA programming which directly uses the driver. One advantage for this approach is it gives us a lot of control but this approach is complicated because it is low-level (it interacts with binaries or assembly code). Another programming interface is the language integration programming interface (right column of Fig. 3). As explained in [9], it is better to use the C runtime for CUDA, which is a high-level approach that requires less code and is easier in programming and debugging. Also, we could use high-level languages e.g. Fortran, Java, Python, or .NET languages through bindings. Therefore, in this work we have used the C runtime for CUDA.

The CUDA programming model, as discussed in [10], suggests a helpful way to solve a problem by splitting it in two steps: Firstly into coarse independent sub-problems (grids) and then into finer sub-tasks that can be executed cooperatively (thread blocks). The programmer writes a serial C for CUDA program which invokes parallel *kernels* (functions written in C). The kernel is usually executed as a grid of *thread blocks*. In each block the threads work together through barrier synchronization and they have access to a shared memory which is only visible to the block. Each thread in a block has a different *thread ID* which

[2]Single Instruction Multiple Data, in the Flynn's taxonomy of computer architectures

can be accessed though **threadIdx**. Each *grid* consists of independent blocks. Each block in a grid has a different *block ID* which can be accessed though **blockIdx**. Grids can be executed either independently or dependently. Independent grids can be executed in parallel provided that the hardware being used supports executing concurrent grids. Dependent grids can only be executed sequentially. There is an implicit barrier that ensures that all blocks of a previous grid have finished before any block of the new grid is started.

In our work, we have two kernels: one to detect the intersection between polygons and find the list of pairs that have intersection. Another one, to find the intersection between all pairs.

### C. Separation Of Convex Polygons in 2D

We have used the method of separating axes [11] to determine whether or not two convex polygons are intersecting. This method is for determining whether or not two stationary convex objects are intersecting. The ideas can be extended to handle moving convex objects and are useful for predicting collisions of the objects and for computing the first time of contact. This method is a fast generic algorithm that can remove the need to have collision detection code for each type pair (any type of convex polygons) thereby reducing code and maintenance.

Based on this method, a test for nonintersection of two convex objects is simply stated: If there exists a line for which the intervals of projection (the lowest and highest values of the polygon projection on the line) of the two objects onto that line do not intersect, then the objects do not intersect. Such a line is called a separating line or, more commonly, a separating axis.

For a pair of convex polygons in 2D, only a finite set of direction vectors needs to be considered for separation tests. That set includes the normal vectors to the edges of the polygons. The left picture in Fig. 4 shows two nonintersecting polygons that are separated along a direction determined by the normal to an edge of one polygon. The right picture shows two polygons that intersect (there are no separating directions).

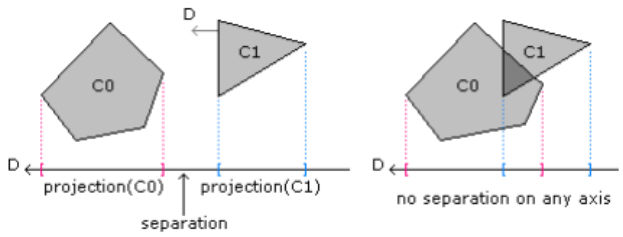


Fig. 4. Nonintersecting convex polygons (left). Intersecting convex polygons (right). [11]

### D. Intersection Of Convex Polygons

The intersection of two arbitrary polygons of $n$ and $m$ vertices can have quadratic complexity, $\Omega(nm)$. But the intersection of two convex polygons has only linear complexity, $O(n+m)$. Intersection of convex polygons is a key component of a number of algorithms, including determining whether two sets of points are separable by a line. The first linear algorithm was found by Shamos (1978), and since then a variety of different algorithms have been developed, all achieving $O(n+m)$ time complexity. In our work, we have used the algorithm developed by O'Rourke, Chien, Olson & Naddor since it is claimed to be the simplest algorithm available.[12] Based on the research that we have done to find an algorithm for calculating the intersection between two convex polygons, we haven't found any simpler than the one that we have used in this work.

The basic idea of the algorithm is straightforward, but the converting of the idea into code is somewhat delicate. Assume the boundaries of the two polygons $P$ and $Q$ are oriented counterclockwise, let $A$ and $B$ be directed edges on each. The algoirthm has $A$ and $B$ chasing one another. The basic structure of the algorithm is illustrated in Algorithm 1[12].

**Algorithm 1** :Intersection of convex polygons

1) Choose $A$ and $B$ arbitrarily.
2) repeat
   a) if $A$ intersects $B$ then
      i) Check for termination.
      ii) Update an $inside$ flag.
   b) Advance either $A$ or $B$,
      i) depending on geometric conditions.
3) until both $A$ and $B$ cycle their polygons
4) Handle $P \cap Q = \Phi$ and $P \subset Q$ and $P \supset Q$ cases.

### E. Experimental Procedure

The problem explored in this paper is to detect and locate the intersection between polygons. We have implemented the serial and parallel algorithms to compute the intersection between polygons. Then, we run both algorithms using 25 data sets of polygons: five different set size (100, 500, 1000, 2000, 3000) and each size has five data sets. Finally, we measured the speed-up (ratio of time for serial algorithm to that for parallel algorithm).

The GPU card that we have used in our work is **Tesla C2050** which is shown in Fig. 5. This card has 448 processor cores, 1.15 GHz processor core clock and 144 GB/sec memory bandwidth.

## III. Results

Fig. 2 shows the CPU and GPU time to detect and locate the intersection between polygons for all five data sets. As we see in Fig. 6 we can tell that the GPU time is less than the CPU time and as we increase the number of polygons the CPU time gets much higher than GPU time. Therefore, we conclude the GPU is more efficient when we have huge number of polygons.


Fig. 5. Tesla C2050.

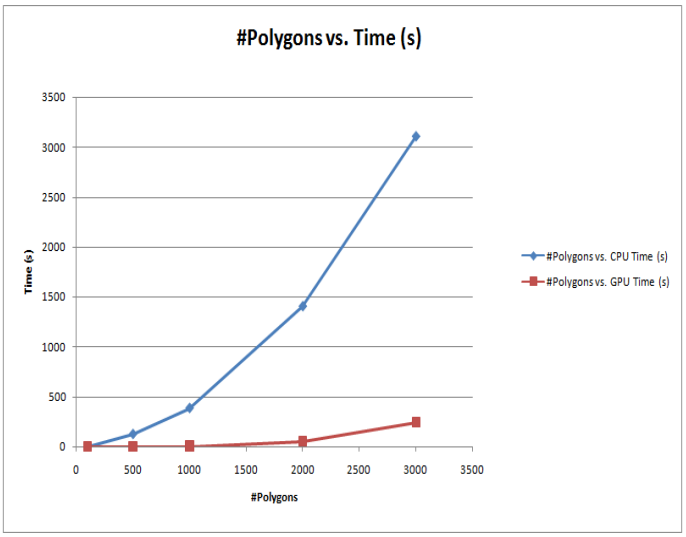


Fig. 6. Compute Time.

Fig. 7 shows the speed up (ratio of time for serial algorithm to that for parallel algorithm) in all five different cases. We notice that the highest speed up is when the number of polygons is 500. We believe this is due the number of processor cores (448) on the card that we have used. Each polygon is approximately handled by one core, but in cases where there are more than 448 polygons one core must handle more than one polygon.

## IV. Conclusion

The paper introduced the basics of GPGPU. The stream processing programming model and the traditional GPGPU approach was presented. CUDA was introduced, including the programming model. The experiment proved performance benefits for detecting and locating the intersection between polygons. It is clear that GPGPU has the potential of significantly improving the processing time of highly data parallel algorithms.

## V. Future Work

Clearly a next step in this research will be to validate and enhance physical models that we are going to use for

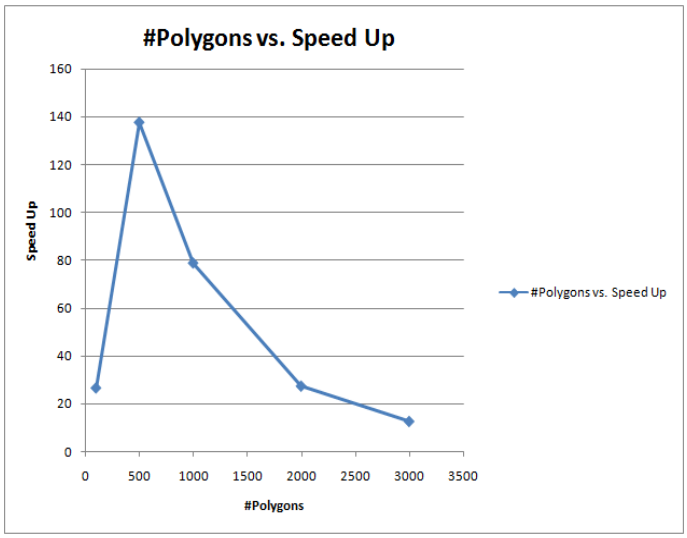


Fig. 7. Speed Up.

simulating the ice floe behaviour using the GPGPU. Adding more model characteristics (driving forces, 3D, floe splitting) will allow us to provide a more useful simulation.

## VI. Acknowledgments

This research has been done under STePS$^2$ project, under the leadership of Drs. Claude Daley and Bruce Colbourne, and was supported by: ABS, Atlantic Canada Opportunities Agency, BMT Fleet Technology, Husky Oil Operations Ltd, Research and Development Council, Newfoundland and Labrador and Samsung Heavy Industries.

## References

[1] Hubert Nguyen, *Gpu gems 3*, Addison-Wesley Professional, first edition, 2007.

[2] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron Lefohn, and Timothy J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[3] Haxon, "Ice floe at oslofjord," March 2009, http://www.panoramio.com/photo/19618780.

[4] Jasmin Blanchette and Mark Summerfield, *C++ GUI Programming with Qt 4 (2nd Edition) (Prentice Hall Open Source Software Development Series)*, Prentice Hall, 2 edition, Feb. 2008.

[5] John Owens, "Streaming architectures and technology trends," in *GPU Gems 2*, Matt Pharr, Ed., chapter 29, pp. 457–470. Addison Wesley, Mar. 2005.

[6] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan, "Brook for GPUs: stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, New York, NY, USA, 2004, pp. 777–786, ACM Press.

[7] Nvidia, "Cuda programming guide v2.3.1," 2009.

[8] Nvidia, "Cuda development tools v2.3. getting started," 2009.

[9] Nvidia, "Cuda architecture overview v1.1. introduction & overview," 2009.

[10] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, pp. 40–53, March 2008.

[11] David Eberly, "Intersection of convex objects: The method of separating axes," *Geometric Tools, LL*, 2008.

[12] Joseph O'Rourke, *Computational Geometry in C*, Cambridge University Press, New York, NY, USA, 2nd edition, 1998.