Ice Simulation Using GPGPU

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

***Abstract*— Simulation of the behaviour of a ship operating in pack ice is a computationally intensive process to which General Purpose Computing on Graphical Processing Units (GPGPU) can be applied. In this paper we present an efficient parallel implementation of such a simulator developed using the NVIDIA Compute Unified Device Architecture (CUDA). We have conducted an experiment to measure the relative performance of the parallel and serial versions of the simulator by running both versions on several different ice fields for several iterations to compare the performance. Our results show speed up of up to 77 times, reducing simulation time for a small ice field from over 88 minutes to about 68 seconds.**

***Index Terms*— GPGPU, CUDA, simulation.**

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 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')."[1] 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.[1] 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 results of measuring the performance of the GPU version of the simulator with respect to the CPU version through an experiment consisting of implementing both serial and parallel version of the simulator and running them both on different ice fields for several iterations 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, ships 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. The goal of this work is to be able to simulate behaviour of ice fields sufficiently quickly to allow the results to be used for planning ice management activities, and so it is necessary to achieve many times faster than real-time simulation.

This project is structured in two components, the *Ice Simulation Engine*, which is the focus of this paper, and the *Ice Simulation Viewer*, which is being developed to display the data produced by the simulation engine. The simulation viewer 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 [3] user interface framework. Fig. 2 shows a screenshot

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

Fig. 1. Ice Floe[2]

of the main interface of the Ice Simulation Viewer with ice field loaded.

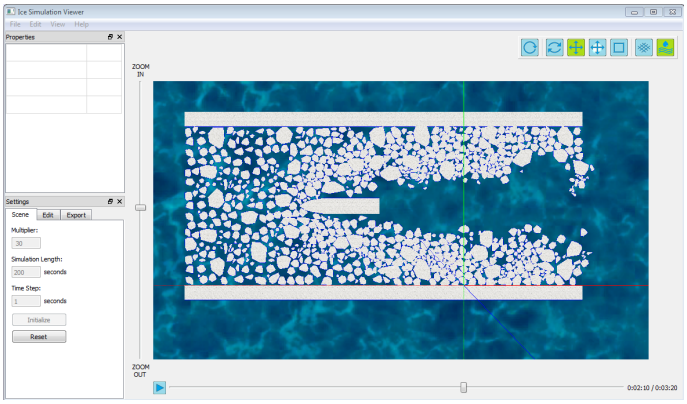

Fig. 2. Ice Simulation Viewer

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 processed in parallel is called a stream. The stream is processed by a series of instructions, called a kernel [4]. 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 access stream processing becomes ineffective [4], [5]. Computer graphics processing is very well suited for stream processing, since vertices, fragments and pixels 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. 3 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).

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

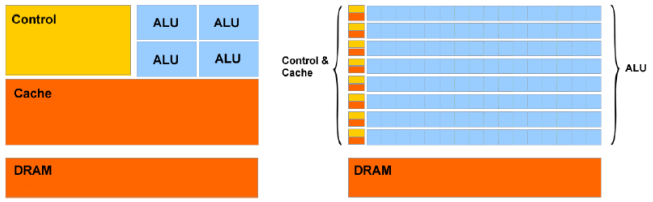

Fig. 3. Simple comparison of a CPU and a GPU [6]

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[7]. An general overview of CUDA is illustrated in Fig. 4.

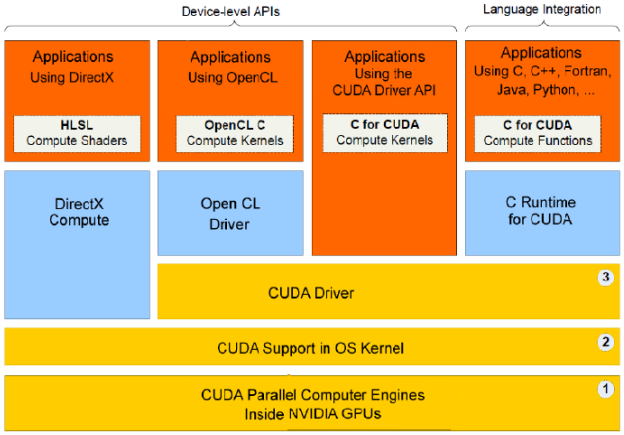

Fig. 4. CUDA overview [8]

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. 4) in which 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 use 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 of this approach is it gives us a lot of control, but a disadvantage is that it 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. 4). Based on [8], 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.

Based on [9], the CUDA programming model suggests a helpful way to solve the problems by splitting it into two steps: The first one 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 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 we have a hardware that 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.

C. Collision Detection

Since we are using a discrete time simulation, for each time step we detect collisions by searching for regions of overlap between ice floes, compute the force that would result from such a collision and adjust the velocity of each floe accordingly. The problem of detecting collisions between ice floes is broken down into two parts: determining if the floes are overlapping, and computing the region of overlap.

To determine whether or not two convex polygons are intersecting we have used the method of separating axes [10]. This method is for determining whether or not two stationary convex objects are intersecting, and we extend it to our moving objects by considering the objects at each time step. 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.

In this method the 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: the normal vectors to the edges of the polygons. The left picture in Fig. 5 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).

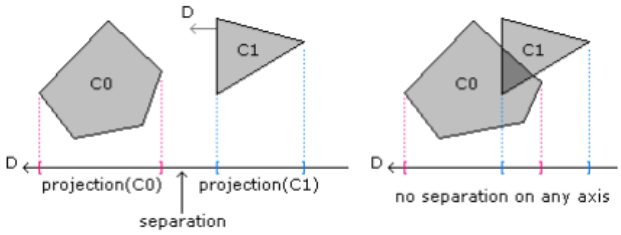

Fig. 5. Nonintersecting convex polygons (left). Intersecting convex polygons (right). [10]

Once it has been determined that two polygons are overlapping, we must find the region of overlap to compute the resultant force. Finding 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 [11], 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 that developed by O'Rourke, Chien, Olson & Naddor [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 simpler than the one that we have used in this work.

The basic idea of the algorithm is as illustrated in Algorithm 1[12]. Here, we assume the boundaries of the two polygons P and Q are oriented counterclockwise, and let A and B be directed edges on each. The algoirthm has A and B chasing one another.

D. Implementation Discussion

As we have developed our implementation we have progressed through three different general structures of the GPU solution, as follows. The relative performance of these is illustrated in Fig. 7.

In the first implementation, we used two CUDA kernels: One, executed using one thread per polygon, finds the list of all pair-wise collisions by determining which pairs of polygons (ice floes) are overlapping. The second kernel, executed using one thread per overlapping polygon pair, computes the collision response (forces) for each pair. This approach resulted in speed-up of up to 10 times as compared

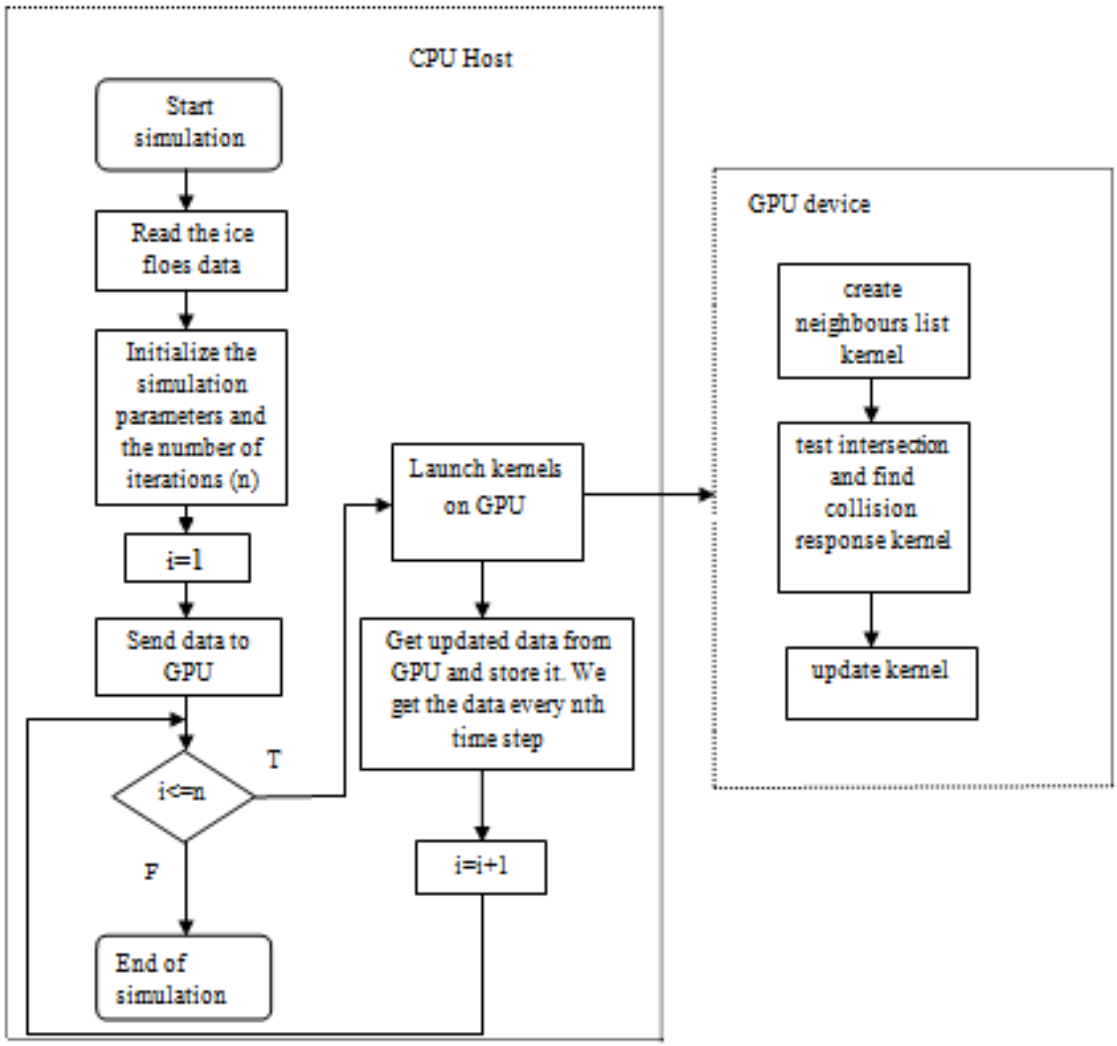

Fig. 6. Ice Simulator Framework

with the CPU implementation, achieving real-time results in only particular cases and so is insufficient for our needs.

In our second implementation we merged the two kernels in one kernel. One thread for each polygon to check the collision with other polygons and calculate the response. This approach was a little faster than the first, but still insufficient for the general case.

In the third implementation we take advantage of the fact that polygons that are widely separated are unlikely to overlap any time soon, and so we can dramatically reduced the number of polygons to be checked for collision by eliminating those that are beyond some distance away. To do this we add another kernel that finds the list of neighbours for each polygon that are within the region where they might overlap with it soon. Therefore, instead of checking the collisions with every other object we just check with those in this list. The list is re-created periodically, but not every time step, so that the total number of computations is significantly reduced. This approach is significantly faster than the other two approaches as we see in Fig. 7 and achieves substantially better than real-time for small ice fields. We are optimistic that through further development and optimization we will achieve our goals for larger ice fields.

E. Simulator Framework

Fig. 6 shows the high-level flow of the ice simulator. At the beginning the CPU reads the ice floe data (position and velocity) and initializes the simulation parameters. The initial data is transferred from the CPU to the GPU. Then, the GPU takes over the main work of the simulation. First, the "create neighbours list" kernel is launched to find the list of polygons that might overlap with each ice floe. Then, the "test intersection and find collision response" kernel is launched to determine the list of ice floes that have overlap with each ice floe and to calculate the collision response for each ice floe. Last, the update kernel is launched to update the position and velocity for all ice floes. After that, the ice floes data is transferred back to the CPU. This process is repeated until

Algorithm 1 :Intersection of convex polygons

▷ Assume that P and Q overlap

Choose A and B arbitrarily.
repeat
  if A intersects B **then**
    The point of intersection is a vertex.
    One endpoint of each of A and B is a vertex.
  end if
  Advance either A or B, depending on geometric conditions.
until both A and B cycle their polygons
if no intersections were found **then**
  One polygon must be entirely within the other.
end if

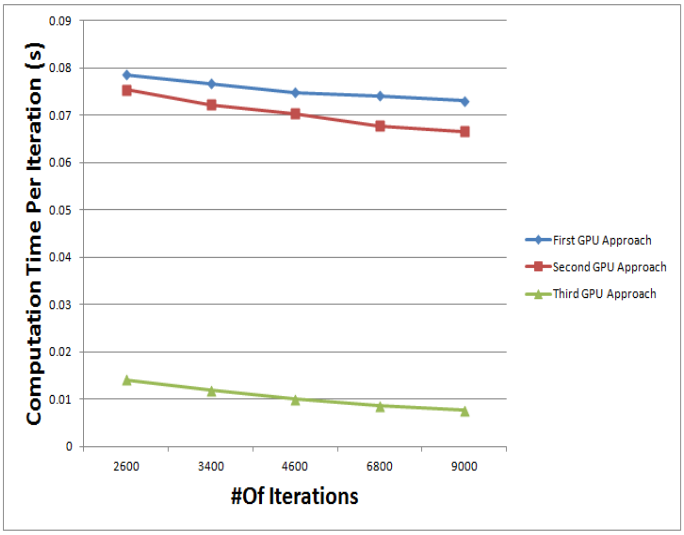

Fig. 7. Computation Time Per Iteration Of The Three GPU Approaches For The First Ice field.

the simulation is completed.

F. Experiment Procedure

We have implemented a serial and parallel version of the simulator and tested both versions on two different ice fields and different (real-time) durations. The first ice field has 456 ice floes and the second one has 824 ice floes. The computation time step (Δt) that we have used in the simulations is 0.1s. We have used 0.1s to maintain the accuracy in the ice mechanics. The distance of the region that we have used in the simulations to generate the neighbor list is 70m. Finally, we measured the speed-up (ratio of time for serial algorithm to that for parallel algorithm).

We have used Intel(R) Xeon(R) CPU @2.27GHz (2 processors) and a GPU Tesla C2050 card which is shown in Fig. 8. This card has 448 processor cores, 1.15 GHz processor core clock and 144 GB/sec memory bandwidth.

III. Results

Fig. 9 shows the CPU and GPU computation time per iteration to simulate the behaviour of the ship in the first ice field which has 456 ice floes for all five different durations (numbers of iterations). As we see in Fig. 9 we can tell that the GPU time is much lower than the CPU time. Moreover, the simulation is super real-time since the computation time per iteration is less than the computation time step ($\Delta t = 0.1s$).

Fig. 8. Tesla C2050.

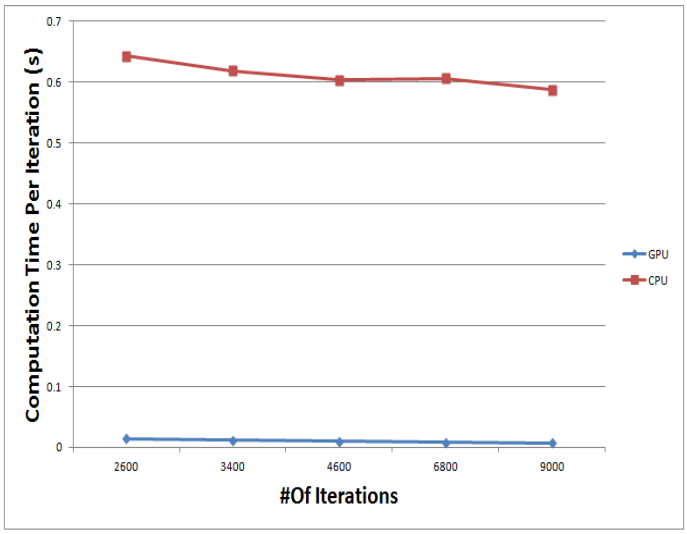

Fig. 9. Computation Time Per Iteration For The First Ice field.

Fig. 10 shows the speed up (ratio of time for serial algorithm to that for parallel algorithm) in all five different cases for the first ice field.

Fig. 11 shows the CPU and GPU computation time per iteration to simulate the behaviour of the ship in the second ice field which has 824 ice floes for all five different durations. As we see in Fig. 11 we can tell again that the GPU time is much lower than the CPU time. Moreover, the simulation is super real-time since the computation time per iteration is less than the computation time step ($\Delta t = 0.1s$).

Fig. 12 shows the speed up (ratio of time for serial algorithm to that for parallel algorithm) in all five different cases for the second ice field.

IV. Related Work

There are several researchers who have developed particle system simulation on GPUs. Kipfer et al. [13] described an approach for simulating particle systems on the GPU

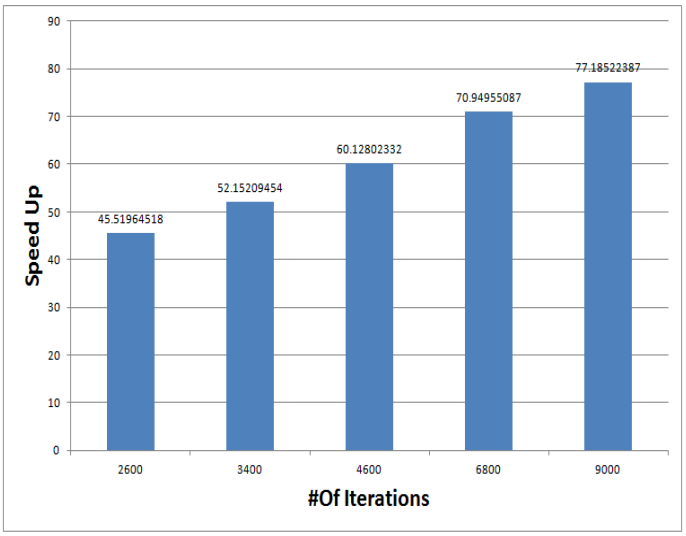

Fig. 10. Speed Up For The First Ice Field.

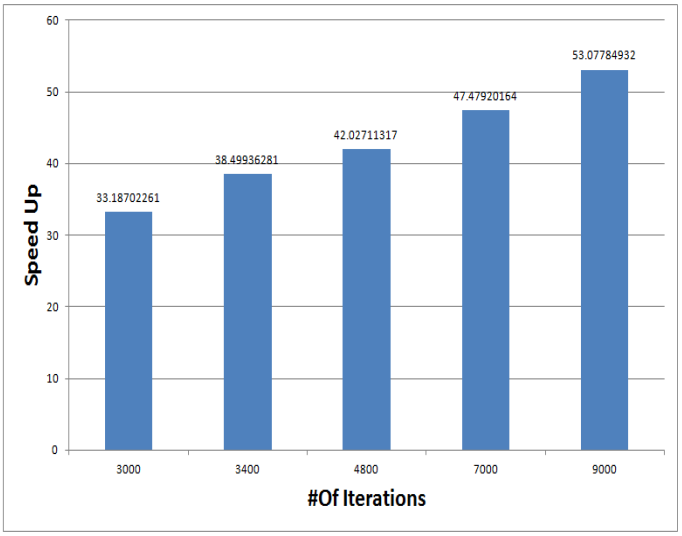

Fig. 12. Speed Up For The Second Ice Field.

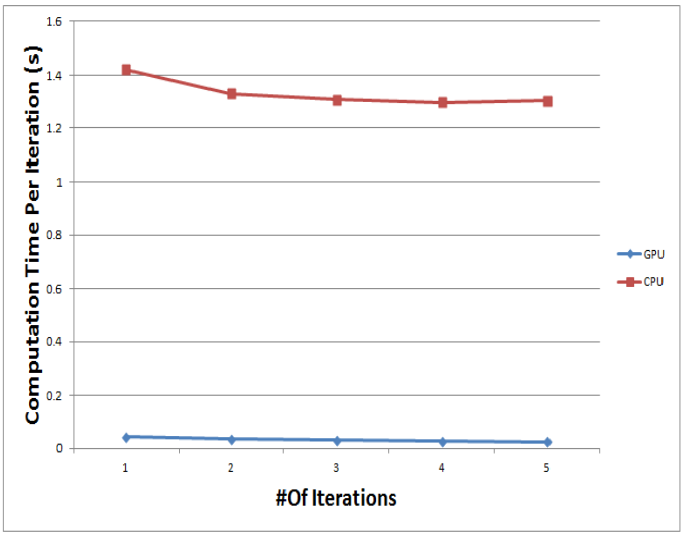

Fig. 11. Computation Time Per Iteration For The Second Ice Field.

including inter-particle collisions by using the GPU to quickly re-order the particles to determine potential colliding pairs. Kolb et al. [14] described a GPU particle system simulator that provides a support for accurate collisions of particles with scene geometry by using GPU depth comparisons to detect penetration. A simple GPU particle system example is provided in the NVIDIA SDK [15].

Several groups have used the GPU to successfully simulate fluid dynamics. A couple of papers described solutions of the Navier-Stokes equations (NSE) for incompressible fluid flow using the GPU [16], [17], [18], [19]. Harris [20] introduced an introduction to the NSE and a complete description of a basic GPU implementation. Harris et al. [18] used GPU-based NSE solutions with partial differential equations (PDEs) for thermodynamics and water condensation and light scattering simulation to develop visual simulation of cloud dynamics. A simulation of the dynamics of ideal gases in two and three dimensions using the Euler equations on the GPU was described in [21].

Recent works shows that the rigid body simulation for computer games performs very well on GPUs. Havok [22], [23] explained an API for rigid body and particle simulation on GPUs, which has all features for full collisions between rigid bodies and particles, and provides support for simulating and rendering on separate GPUs in a multi-GPU system. Running on a PC with dual NVIDIA GeForce 7900 GTX GPUs and a dual-core AMD Athlon 64 X2 CPU, Havok FX achieves more than a 10x speedup running on GPUs compared to an equivalent, highly optimized multithreaded CPU implementation running on the dual-core CPU alone.

Lubbad et al. [24] described a numerical model to simulate the process of ship-ice interaction in real-time. PhysX is used to solve the equations of rigid body motions for all ice floes in the calculation domain. They have validated their results of the simulator against experimental data from model-scale and full-scale tests. The validation tests showed a adequate agreement between the model calculations and experimental measurements. The goal of our work is to be able to simulate behaviour of ice fields sufficiently quickly by using GPGPU to allow the results to be used for planning ice management activities, and so it is necessary to achieve many times faster than real-time simulation.

V. 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 simulating the complex mechanics of a ship operating in pack ice. It is clear that GPGPU has the potential of significantly improving the processing time of highly data parallel algorithms.

VI. Future Work

The physical models do not as yet consider driving forces (e.g., current, wind) and don't model floe splitting, both of which are necessary for fully functional models and so adding these will be a next step. Also, while the results so far are promising, we have yet to reach the point where the simulation is fast enough to be practically used for planning ice management activities in realistic size ice fields. Further development and optimization are necessary to achieve this.

VII. 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] 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.

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

[3] 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.

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

[5] 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.

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

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

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

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

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

[11] Michael I. Shamos, *Computational geometry*, PHD thesis, Yale University, New Haven, 1978.

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

[13] Peter Kipfer, Mark Segal, and Rüdiger Westermann, "UberFlow: a GPU-based particle engine," in *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, New York, NY, USA, 2004, pp. 115–122, ACM.

[14] A. Kolb, L. Latta, and C. Rezk-Salama, "Hardware-based simulation and collision detection for large particle systems," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, New York, NY, USA, 2004, HWWS '04, pp. 123–131, ACM.

[15] Simon Green, "Nvidia particle system sample," 2004, http://download.developer.nvidia.com/developer/SDK/.

[16] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 917–924, July 2003.

[17] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys, "A multigrid solver for boundary value problems using programmable graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Aire-la-Ville, Switzerland, Switzerland, 2003, HWWS '03, pp. 102–111, Eurographics Association.

[18] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra, "Simulation of cloud dynamics on graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Aire-la-Ville, Switzerland, Switzerland, 2003, HWWS '03, pp. 92–101, Eurographics Association.

[19] Jens Krüger and Rüdiger Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 908–916, July 2003.

[20] Mark Harris, "Fast fluid dynamics simulation on the gpu," in *ACM SIGGRAPH 2005 Courses*, New York, NY, USA, 2005, SIGGRAPH '05, ACM.

[21] Trond R. Hagen and Jostein R. Natvig, "Solving the Euler Equations on Graphics Processing Units," *Comp. Sci. - ICCS*, vol. 2006, pp. 220–227.

[22] Andrew H. Bond, "Havok fx: Gpu-accelerated physics for pc games," in *Proceedings of Game Developers Conference 2006*, mar 2006, http://www.havok. com/content/view/187/77/.

[23] Simon Green and Mark J. Harris, "Game physics simulation on nvidia gpus," in *Proceedings of Game Developers Conference 2006*, mar 2006, http://www. havok.com/content/view/187/77/.

[24] Raed Lubbad and Sveinung Lset, "A numerical model for real-time simulation of shipice interaction," *Cold Regions Science and Technology*, vol. 65, no. 2, pp. 111 – 127, 2011.