https://lib.uliege.be https://matheo.uliege.be

# Master thesis : Discrete Element Simulation of Ice Particle Interaction: Migration to GPU Computing and Subsequent Validation

**Auteur :** Bristy, Kaniz Fatema
**Promoteur(s) :** 14964
**Faculté :** Faculté des Sciences appliquées
**Diplôme :** Master : ingénieur civil mécanicien, à finalité spécialisée en "Advanced Ship Design"
**Année académique :** 2021-2022
**URI/URL :** http://hdl.handle.net/2268.2/16560

---

*Avertissement à l'attention des usagers :*

*Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.*

*Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.*

With the support of the Erasmus+ Programme of the European Union

# Discrete Element Simulation of Ice Particle Interaction: Migration to GPU Computing and Subsequent Validation

Submitted on 27th August 2022

By
BRISTY Kaniz Fatema | 5-7 Esp. de la Pierre Percée | 44300 Nantes |
Kaniz-Fatema-.Bristy@eleves.ec-nantes.fr
Student ID No.: 210603Q

**First Reviewer:**
Pierre FERRANT
Professor at
Ecole Centrale de Nantes
1, rue de la noë
44321, Nantes
France

**Second Reviewer:**
Félicien BONNEFOY
Lecturer at
Ecole Centrale de Nantes
1, rue de la noë
44321, Nantes
France

# Table of Contents

# List of Figures

# Acronyms

| | |
|---|---|
| FEM | Finite Element Method |
| DEM | Discrete Element Method |
| CPU | Central processing unit |
| GPU | Graphics processing unit |
| CUDA | Compute Unified Device Architecture |
| API | Application programming interface |

# Abstract

A numerical simulation code using Discrete Element Method has been developed by HSVA, which can generate brash ice and ice ridges as well as analyse ship navigation through ice channels. The current version is simulating the problem in model scale for ease of validation.
This thesis aims to enhance the software's capabilities, reduce the computational time, and to enhance performance and capabilities by modifying internal source code.

To improve the performance GPU programming has been introduced. GPU programming extension CUDA, developed by NVIDIA, has led to numerous advances in computing over the last few years. The CUDA API makes it relatively easy for users to access the graphics card hardware, which allows users to perform parallel computations with thousands of CUDA cores.

The following report investigates the methodology and advantages of using the CUDA API for DEM computations. In order to achieve this, existing CPU code had to be rewritten for the GPU. Both implementations show significant improvements with regard to iteration time, and performance depending on of GPU architecture.

Additionally, it has been demonstrated that the GPU can be sped up by simply varying certain parameters, which boosts the code's performance overall. Another investigation dives into the overhead associated with programming memory intensive scripts to the GPU and shows what effect this has on the total calculation times for the application. Further a more complex Ice-Structure interaction algorithm can improve the quality of results.

In this case a different number of simulations is done varying the element number to find out the dependency of the elements to the computational time. Eventually, several tests were conducted for different types of brash ice and Ice ridge channels to see how the software would react.

**Keywords**: DEM, GPU, CUDA, Time consumption

# Acknowledgment

Working in Hamburgische Schiffbau-Versuchsanstalt GmbH (HSVA) has been a unique experience for me during the past six months.

I would like to thank my thesis supervisor, Quentin Hisette for giving me this opportunity. As a professional, he was the most humane, tolerant, and kind person I have ever met. I will always be grateful to him for his help and valuable inputs while learning Fortran and Implimentation of CUDA.

I would like to thank every professor I have had for the past two years, both at the University of Liege and at Ecole Centrale de Nantes, and especially professor RIGO Philippe for believing in me and guiding me at every step of my master journey.

I would also like to thank my colleagues, my friends for sharing their thoughts, knowledge and support.

The last, but certainly not the least. I dedicate this to my family. The lighthouse that guides me through an ocean of doubts and fears. It would be impossible for me to conduct this humble work without you, and I dedicate it to you.

France, 27th August 2022
Kaniz Fatema Bristy

# Declaration of Authorship

I declare that this thesis and the work presented in it are my own and have been generated by me as the result of my own original research.

Where I have consulted the published work of others, this is always clearly attributed.

Where I have quoted from the work of others, the source is always given. Apart from such quotations, this thesis is entirely my own work.

I have acknowledged all main sources of help.

Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

This thesis has no material that has been submitted previously, in whole or in part, for the award of any other academic degree or diploma.

I cede copyright of the thesis in favour of the University of Ecole Centrale De Nantes, France.

Date: 27th August 2022 Signature:

Chapter 1

# 1 Introduction

## 1.1 Overview

The surface of the sea ice is not continuous and uniform as it varies dramatically over time. Most of the sea ice in the polar region can be classified as deformed ice (i.e., hummocks, broken ice, ice ridges, etc.) or uniformed ice (mostly level ice). Studies show that the shipping activity in the polar region will increase by 50% within the coming few decades [1]. However, the unpredictable, harsh environment remains the main obstacle for artic engineers.

Most sea ice in polar regions can be generalized into two types, (a) level ice that exists as a continuous form and (b) broken ice that consists of discontinuous ice blocks. In addition to breaking level ice, polar ships can interact with broken ice in many different scenarios. Brash ice accumulates in ice channels, sliding ice pieces form when continuous ice breaks, and unconsolidated ice ridges appear and evolve as natural processes compress ice floes [2]. In arctic regions, ice ridges and brash ices are essential features that add to the overall forces acting on a vessel. Therefore, there is much interest in research and development of this issue.

**Figure 1 Vessel navigating through ice [3]**

The current industries are more likely to perform model tests to get more precise results regarding the ship's performance in such a sensitive environment. However, the problem with the model test is that it can be done after the overall design, which is more likely to be called design validation and expensive. An early-stage estimation of ice resistance can help the engineers to reduce the operating cost and to develop a more effective design. This report only focuses on forming Improving performance of the in-house software (***IceDEM***) which is being used in HSVA to perform simulation regarding ice generation.

While developing a simulation tool is more likely to implement a CPU version as it is easier to program and debug as well as cost effective. In 2003, two research groups independently discovered that GPU-based approaches to solving general linear algebra problems on GPUs ran faster than on CPUs[4],[5]. Moreover, Nvidia's CUDA, allows programmers to underlying graphical concepts in favour of more common high-performance computing concepts [6].

Running a sequentially executed code developed following DEM is devastatingly slow for larger number of elements. In order to fully utilize the available computational resources, this thesis aimed to convert the in-house developed CPU-based program to run on the GPU with the help of CUDA, a parallel computing platform [6].

## 1.2 Motivation

The discrete element method (DEM) is the most widely used numerical approach for microscale modelling.

In DEM calculations, an increasing number of discrete objects, combined with small time steps imposed by an explicit calculation scheme, leads to an increased degree of computational complexity, resulting in long simulation times. One of the easiest ways to cope with this problem is to parallelize the computations. The relative simplicity of this approach stems from the nature of the DEM method [7].

Initially, the parallelization strategies in open-source or commercial DEM software packages focused only on the central processing units (CPU) implementation. During the last decade, the increased capabilities and computational power of graphics cards spurred a transition to where more and more computations are transferred to the GPU [8]. Parallel processing is the main advantage of GPU architecture, which allows large amounts of calculations to be performed simultaneously.

One of the challenges related to the DEM simulations on the GPU is the efficient utilization of available computing resources. During the calculations on GPU, the CPU remains idle, and powerful multi-core processors are not used. Computing efficiency can be improved by using CPU-GPU cooperative computing techniques [7].

The benefits of multi-GPU processing are numerous from an investment perspective. With multiple GPUs, running different cases with millions of particles costs much less than buying an equivalent CPU-based machine. GPU-based machines also consume less energy and can be upgraded more easily by adding more cards or buying new ones [9].

## 1.3 Objective

The aim is to further improvement of the capabilities of the existing in-house software, focusing on the performance of and abilities by the changes of internal source code and verifying it with the model performance data.

Aside from modifying software, the thesis focuses on gaining a general understanding of GPU computing and the CUDA implementation for the Fortran programming language. As part of its initial focus, the existing DEM software generates only brash ice channels and ice ridges. After initialization, the whole data has been allocated and GPU cooperative computing techniques are used to perform the time integration loop in GPU.

Later, GPU computation performance will be compared with CPU computation to verify and validate its effectiveness.

## 1.4 Content Description

This thesis work is divided into five different parts. A theoretical overview of ice ridge and brash ice generation is discussed in the beginning. The HSVA ice tank was used as a testing environment for comparing model results to the actual environment.

Later the same work was done with the in-house CPU-based software (***IceDEM***). Though the program provides many accurate results, it takes much time to produce one output. To avoid this complication, CUDA has been introduced, which has been discussed in the fourth part of the thesis. The overall performance has been discussed to show the difference between the results obtained from these procedures. Furthermore, possible improvement to the program has been discussed.

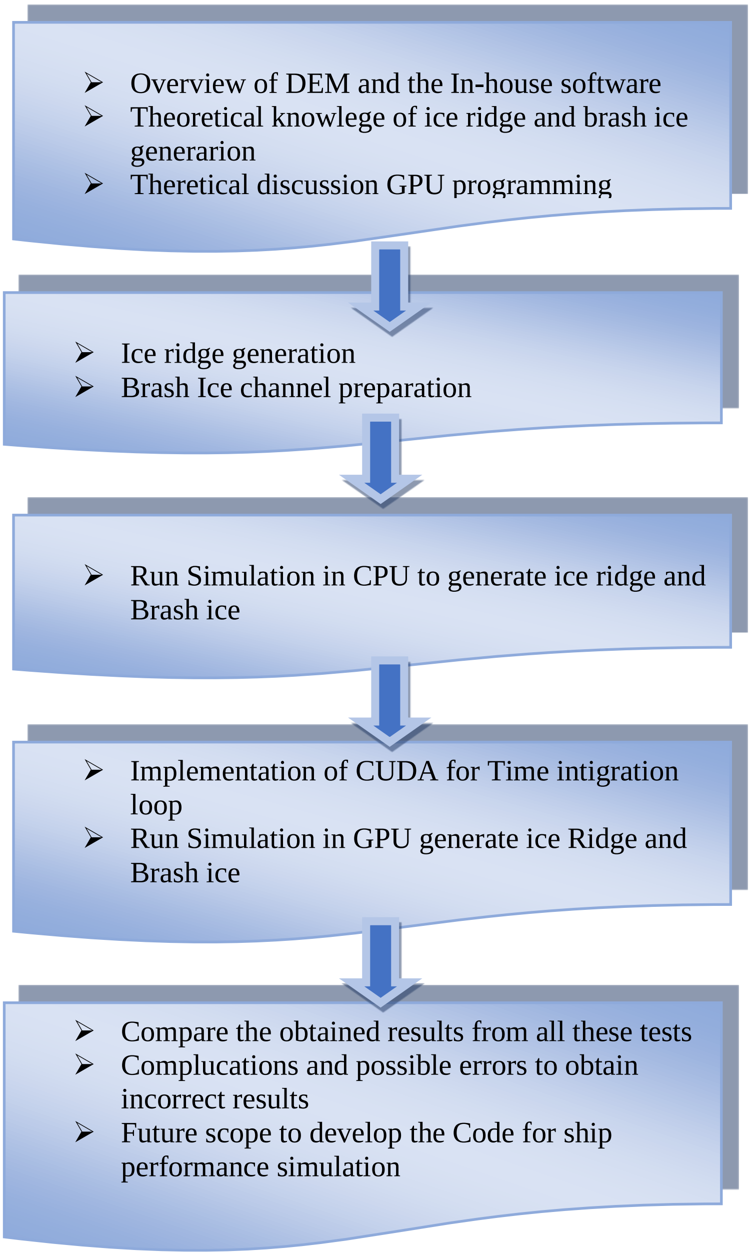


**Figure 2 Content description**

**Chapter 2**

# 2 Theories and Methodology

This chapter discusses the theory and implementation of different types of ice formations. The initial simulation tool in development at HSVA was introduced in terms of its functionalities, capabilities, algorithm description, boundary condition, and set up to generate more realistic brash ice channels and Ice ridges.

Later a summary of relevant theory behind the CUDA framework, including GPU memory architecture, will be presented. Additionally, the theory presented is motivated by the implementation strategy of the code, in addition to outlining the key components of the CUDA framework and parallelism in general.

## 2.1 Ice ridge

The term 'ice ridge' refers to a line or wall of broken ice that forms between relatively large ice floes due to pressure. The age and process of sea ice ridge formation determine their size, strength, and shape.

Depending on their age and formation process, sea ice ridges can be found in many different sizes, strengths, and shapes. In terms of the ice area, the percentage covered by sea ice ridges is small, but their mass can be one-third. Ice-going vessels face many challenges when they encounter these ridges.

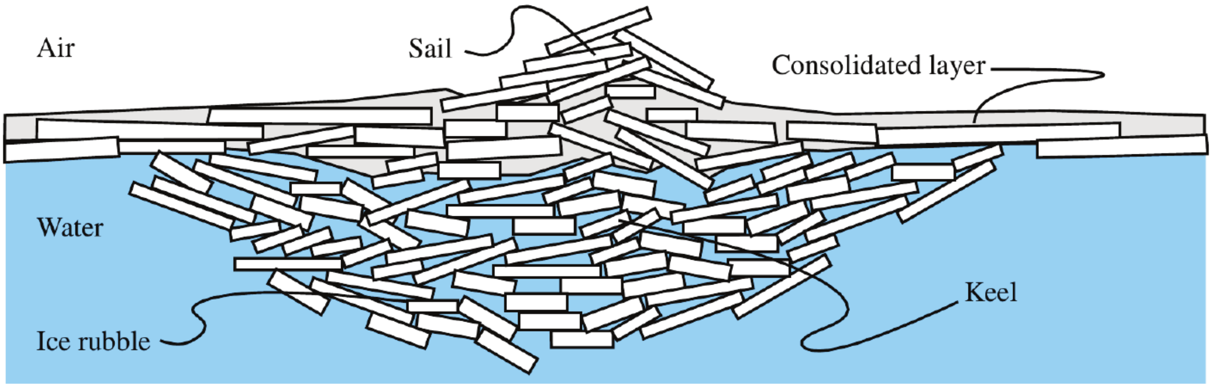


**Figure 3 Cross-section of a ridge[10]**

Pressure ice ridges are formed when ice floes collide or when level ice is stressed. These stresses arise from various external factors, such as currents, wind drag, and thermal expansion. When stresses exceed a certain level of strength, the ice cover breaks, crushes, and bends. As a result, several discrete ice blocks appear between two ice floes or edges of the level ice cover. The above picture represents the overview of ice ridges and their principal parts.

## 2.2 Brash Ice

The term 'brash ice' refers to an accumulation of floating ice fragments that are not wider than 2m. When the icebreaker ship navigates through the level ice, it forms a channel filled with broken ice pieces. If more ships pass through this channel, the pieces of ice are broken further and further, and the broken ice blocks are pushed down by the passing hull.

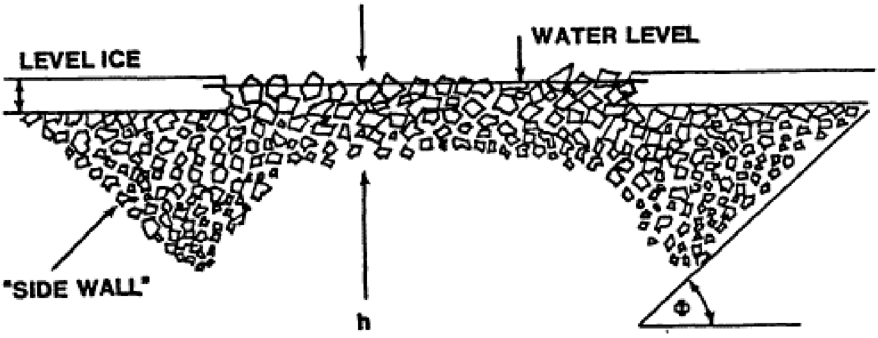


**Figure 4 Cross-section of a brash ice channel[11]**

Ice-class ships without ice-breaking ability follow the same route and break the previously broken ice pieces into even smaller pieces. Due to ship navigation and repeated traffic, ice pieces take shape similar to a sphere because of the rounding off the edges by colliding with passing ships.

## 2.3 Discrete Element Method

"The discrete element method is a numerical method for computing the motion and effect of a large number of small particles."

Individual particles are numerically represented, with the definition of the properties of the particles (location, size, shape, material, initial velocity) and the domain (dimensions, gravitational fields). Movement occurs over a short period and causes some particles to collide with others or the domain boundaries.

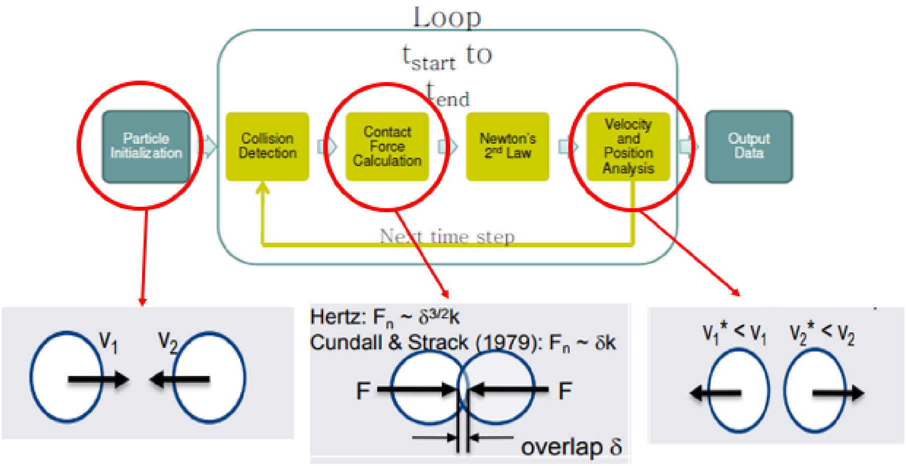


**Figure 5 Example of DEM Simlulation pinciple (Collision of two spheres)[12]**

Behavior of these particles are analyzed in Lagrangian Frame by classical rigid body dynamics under interaction forces of contact plasticity, friction, hydrostatic, electrostatic, magnetic, gravitational etc.

A contact model determines how much reaction force each particle experiences due to collisions. Gravity and hydrostatic forces also exert influence. Summarizing the total force on each particle, and adding external forces, is then done. (Figure 5) Each particle's motion parameters (acceleration, velocity, displacement) are determined using Newton's laws of motion.

## 2.4 Introduction to CPU and GPU

In order to understand CUDA and its implementation, the reader needs a basic understanding of the memory architecture of a normal CPU and GPU. Therefore, this first section aims to provide sufficient background information to facilitate a good understanding of the implementation.

A CPU (central processing unit) consists of just a few cores with lots of cache memory that can handle a few threads of software at once, while GPU computing makes use of a graphics processor as a co-processor to accelerate CPUs for use in general-purpose scientific and engineering applications.

GPUs accelerate CPU-based applications by offloading some of the compute-intensive and time-consuming portions of the code, while the CPU still handles the rest of the simulation. As a result, the application runs faster since it is using the GPU's massive parallel processing power.

## 2.5 CPU and GPU Architecture

CPU and GPU have separate physical locations as well as their own memory locations. The CPU and the GPU cannot convey implicit information, so particular data must be transferred explicitly from the CPU to the GPU or vice versa.

A CPU consists of four to eight CPU cores, while the GPU consists of hundreds of smaller cores. Together, they operate to crunch through the data in the application. This massively parallel architecture is what gives the GPU its high computational performance. Several GPU-accelerated applications provide an easy way to access high-performance computing (HPC).

In order to pass data to the GPU, the programmer will need two copies of the data. There will be one copy on the host side (CPU), which will be configured in some predefined process, and one copy that will be allocated and transferred to the GPU.

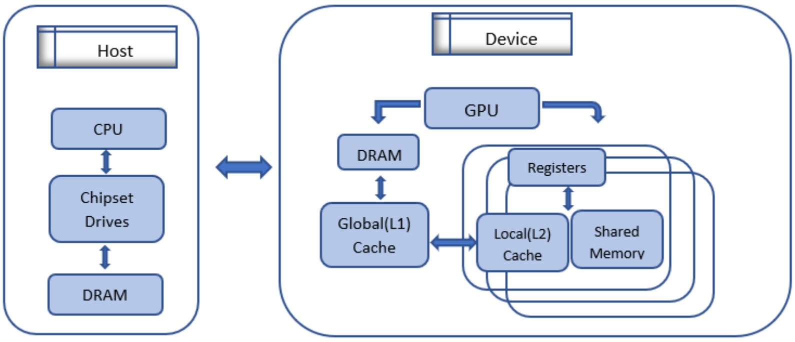


**Figure 6 Memory architecture CPU vs GPU[8]**

Figure 6 shows a schematic view of the memory architecture for the host (on the left) and the device (on the right). The GPU consists of multiple streaming multiprocessors (SM), which are the fundamental building blocks of every GPU. A large number of SMs will generally, simply put, create a more powerful GPU.

Inside an SM, there are a certain number of cores. Inside the cores, the coding instructions take place. Hence, having access to many SMs opens up more cores and thereby more places to execute code. Parallelization is thus limited by the number of SMs.

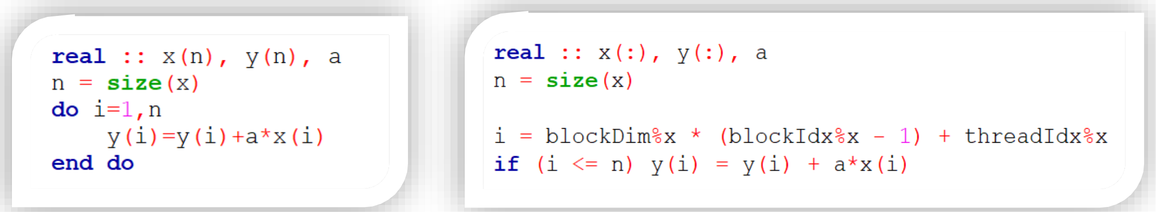
Data from the host is transferred to the GPU through global dynamic random-access memory (DRAM). Recently accessed data requested from an SM passes through two fast access caches. Firstly, the global L1 cache, followed by the local L2 cache. The L1 cache is shared with the whole device, and the L2 cache is local to each SM [8].

As the data reaches the registers or the cores of the GPU, it can either be used for computations or placed in shared memory. By placing the data in shared memory, one can ensure that the data is always accessible with minimal latency.

## 2.6 CUDA

CUDA is a parallel computing platform and programming model for graphics processing units (GPUs). The NVIDIA® CUDA® Toolkit provides a development environment for creating high-performance GPU-accelerated applications.

CUDA supports four fundamental abstractions: cooperating threads organized into thread groups, shared memory and barrier synchronization within thread groups, and coordinated independent thread groups tagged into a grid. A CUDA programmer must divide the program into coarse grain blocks so that they can be executed in parallel. Each block is partitioned into fine-grain threads, which can cooperate using shared memory and barrier synchronization. A properly designed CUDA program will run on any GPU that supports CUDA, regardless of the number of processor cores available.



```
real :: x(n), y(n), a
n = size(x)
do i=1,n
    y(i)=y(i)+a*x(i)
end do
```

```
real :: x(:), y(:), a
n = size(x)

i = blockDim%x * (blockIdx%x - 1) + threadIdx%x
if (i <= n) y(i) = y(i) + a*x(i)
```

**Figure 7 Example of serial computation (CPU) vs parallel computation (GPU)**

In mid-2009, PGI and NVIDIA cooperated to develop CUDA Fortran. CUDA Fortran includes a Fortran 2003 compiler and toolchain for programming NVIDIA GPUs using Fortran. The simple example above shows how a standard Fortran program can be accelerated using CUDA.

## 2.7 Visual Profiler

NVIDIA Visual Profiler provides tools for visualizing and optimizing application performance. In the Visual Profiler, the application's CPU and GPU activity is displayed as a timeline, which makes it easier to identify possible performance improvements. Additionally, the Visual Profiler analyses the application to identify potential performance bottlenecks and directs the coder on how to resolve them [13].

## Chapter 3

# 3 Discrete Element Method Implementation

The DEM is intrinsically advantageous in ice load calculations since it can describe discrete ice structures on macro and micro scales and reasonably model brash ice during ice-vessel or ice-ice interactions. In the Discrete Element Method, the domain is represented by a discontinuous Lagrange treatment of discrete particles which match actual particle sizes and preferably their shapes. The computational focus lies in the detection of particle collisions and the calculation of contact forces.

## 3.1 Ice Particle interaction

The numerical calculation of discrete element method models is divided into three steps according to Cundall and Strack. Figure 8 shows the processes within the loop of a discrete element method [14].

- contact calculation between particles
- integrating the equations of motion
- transition to the next time step.

Two tasks are involved in the contact calculation for the Discrete Elements Method. A contact model calculates the contact forces according to the contact finding and contact force calculation. In case of brash ice simulation, the use of spheres simplifies the process of determining contact or overlap between the particles described above. While defining ice ridges, polyhedral particles have to be used.

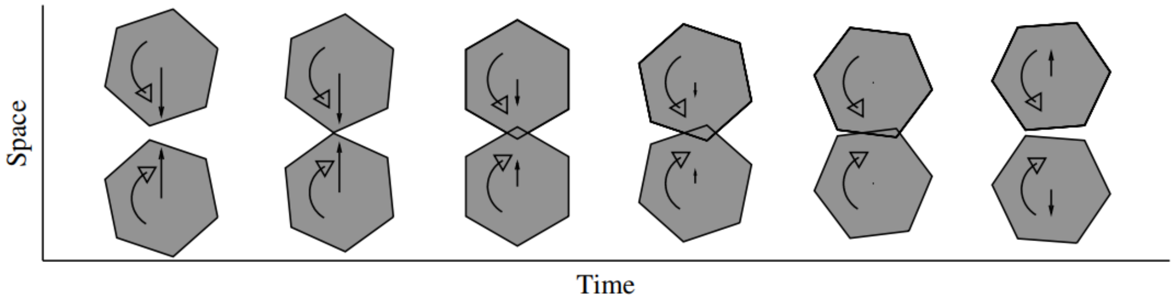


**Figure 8 One-dimensional collision of non-spherical particles[15]**

To calculate the forces acting on the spheres, all spheres in contact must be identified, and the forces acting on the spheres must be calculated after the spheres are found. The external forces are composed of drag, buoyancy, and gravity forces [14].

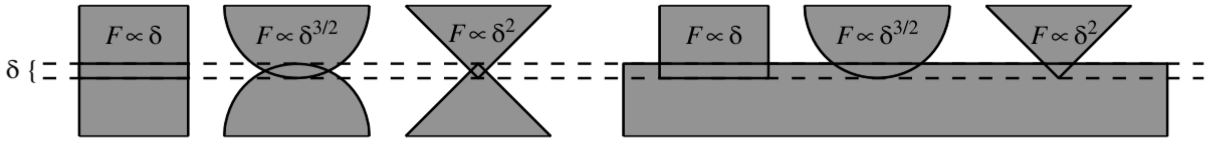


**Figure 9 Contact shapes and proportionality relations of forces for particle–particle and particle–line[15]**

The above figure represents the interaction between two ice particles. From elasticity theory it is known that the elastic force acting between rectangular (‘linear’), spherical (‘Hertzian’) and wedge-shaped contacts should vary with the deformation $\delta$ (or the depth of the overlap in our discrete element approach) as $\delta^1$, $\delta^{3/2}$ and $\delta^2$, respectively.

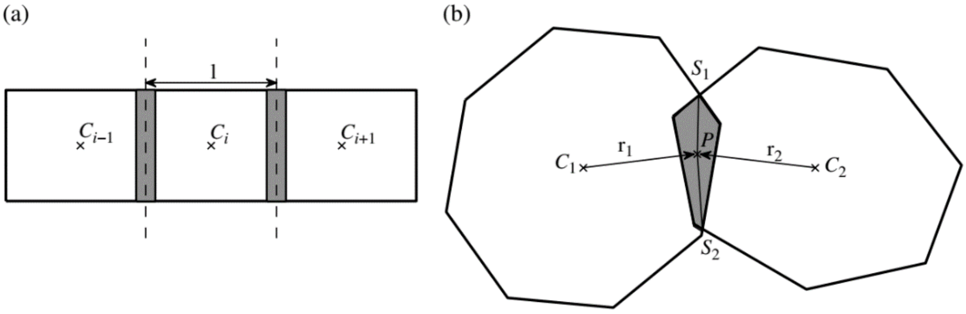


**Figure 10 Ice particle contacts[15]**

The above figure represents the (a) DEM principle for rectangular particle where Particle length with centroids at $C_{i-1}$, $C_i$, $C_{i+1}$. (b) Variables for the force computation for two interacting polygons where the force point P is the centroid of the overlap polygon; $S_1$, $S_2$ are the intersection points of the outlines of the polygons; and r1, r2 are the connecting vectors between the centres of mass $C_1$, $C_2$ of the two polygons.

The equations of motion must be integrated twice to determine the position and velocity of the particles for the next step. The explicit Euler method or the predictor-corrector method are used for the numerical integration of the equation of motion.

## 3.2 Program Flow

The proposed algorithm of DEM modified for consideration of ship and ridge interaction and implemented into developed software is outlined in Figure 11.

The flow chart shows that the main integration loop requires the data for neighbouring particles. After predicting the overlap values, the total force can be obtained. However, finding the contact data for neighbour particles requires a large-scale simulation which takes much time to obtain some results. So, this part is implemented using CUDA.

Initialization needs to run only once and which is outside the time integration loop. So, it does not require a high-performance computer. So, this part is still kept to run for the CPU.

## 3.3 Introducing CUDA for DEM

The DEM method was implemented using Fortran and Compute Unified Device Architecture (CUDA) developed by NVIDIA. The GPU program was formulated using a single-program multiple data (SPMD) technique, where the same program is executed by multiple threads simultaneously. Due to the discrete nature of the particle methods, GPU threads are assigned to each particle. As a result, neighbour searching, force calculation and time integration of the equation of motion can be carried out independently for each particle using different GPU kernel functions.

For efficient use of the GPU memory, parameters that remain the same during simulation, such as material properties, are stored in the constant memory (a type of read-only memory on the

GPU with fast data fetching) while other particle-related information, including positions, velocities, forces and contact histories, are stored in the global memory on the GPU.

Following figure shows the flow chart of the algorithm that runs on a single GPU. The whole program can be divided into two major parts: initialization (Done on CPU) and Main time increment loop (Done in GPU) calculation.

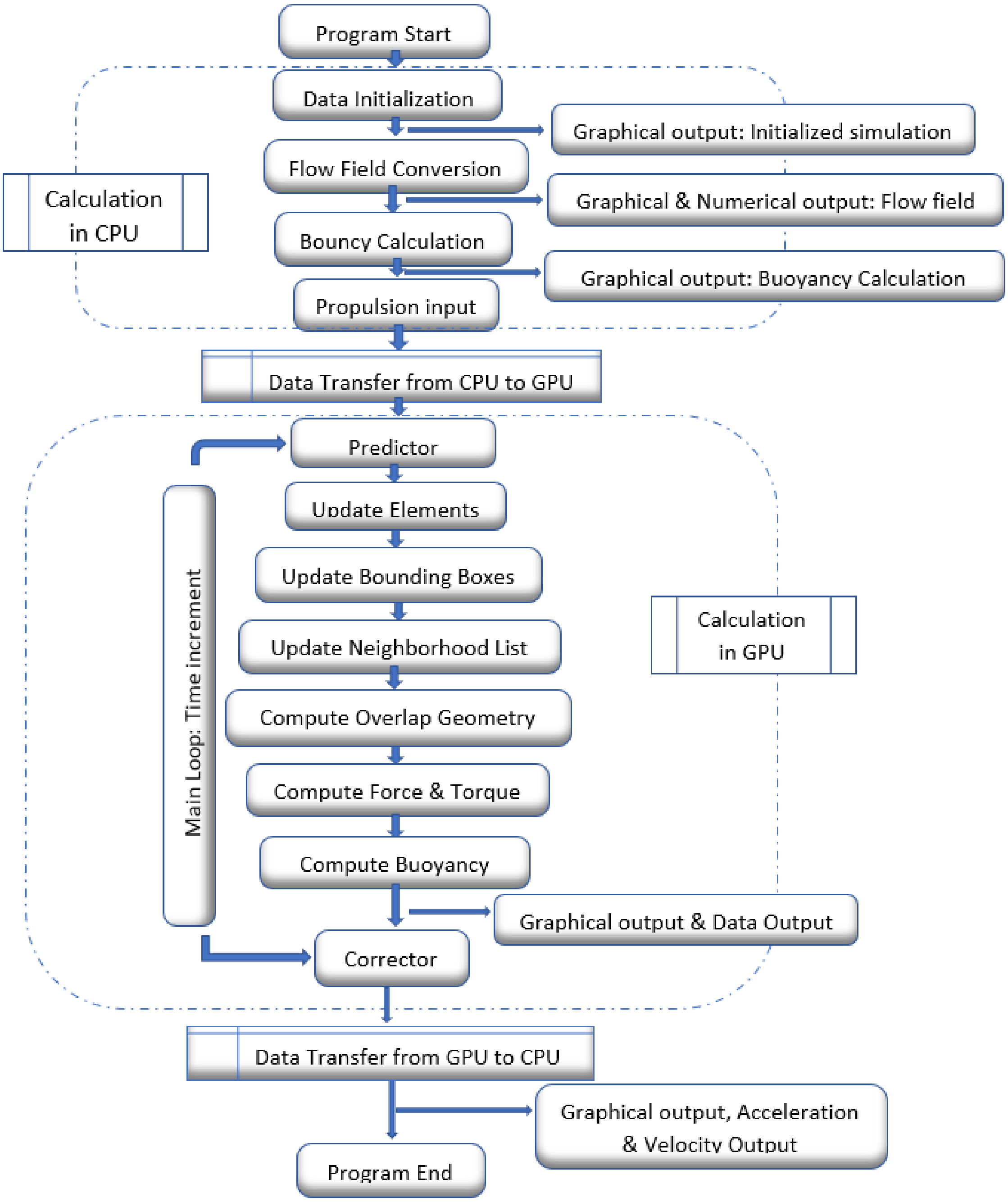


**Figure 11 Program Flowchart**

## Chapter 4

# 4 Ice Ridge Generation

## 4.1 Natural creation process

A pressure ice ridge is formed when ice floes rub against one another under pressure or when level ice is stressed against it. These stresses arise from various external factors, such as currents, wind drag, and thermal expansion. When stresses exceed a certain level of strength, ice cover breaks, crushes, and bends.

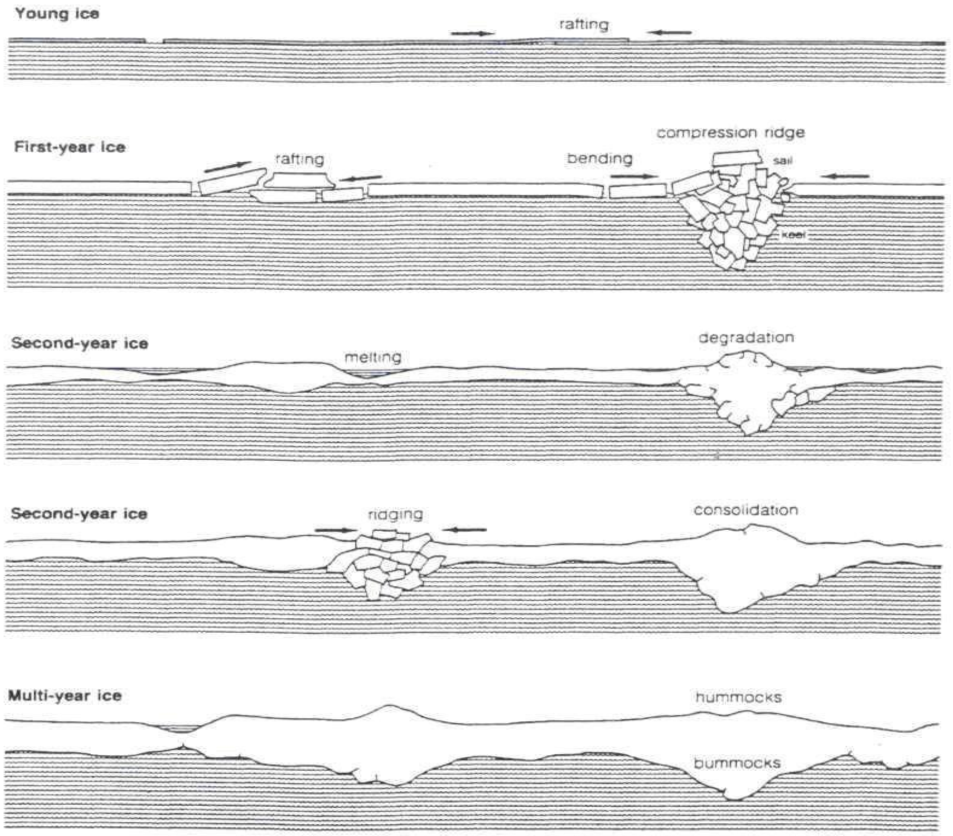


**Figure 12 Ice Ridge creation in nature[16]**

## 4.2 Ice Ridge in ice model basin

Ice ridges are broken ice walls forced up by shear or pressure. Firstly, a parental level ice sheet with pre-defined ice properties is prepared according to HSVA’s standard procedure for preparing an ice ridge in the ice tank. Then the ridges are prepared so that a section of the level ice sheet is cut into narrow strips. Ridges prepared according to HSVA’s standard procedure in the ice tank consist of a sail, a keel, and a consolidated layer and typically show a trapezoidal or triangular cross-section shape.

Afterward, the entire ice sheet, including the strips, is cut free from the side walls of the tank and pushed against the resting ice sheet utilizing the carriage’s pushing board. During the ridge formation, the beam is successively moved forward. The resulting ridge has a pre-defined mass and geometry and is embedded in level ice. The ridge has a typical natural underwater profile.

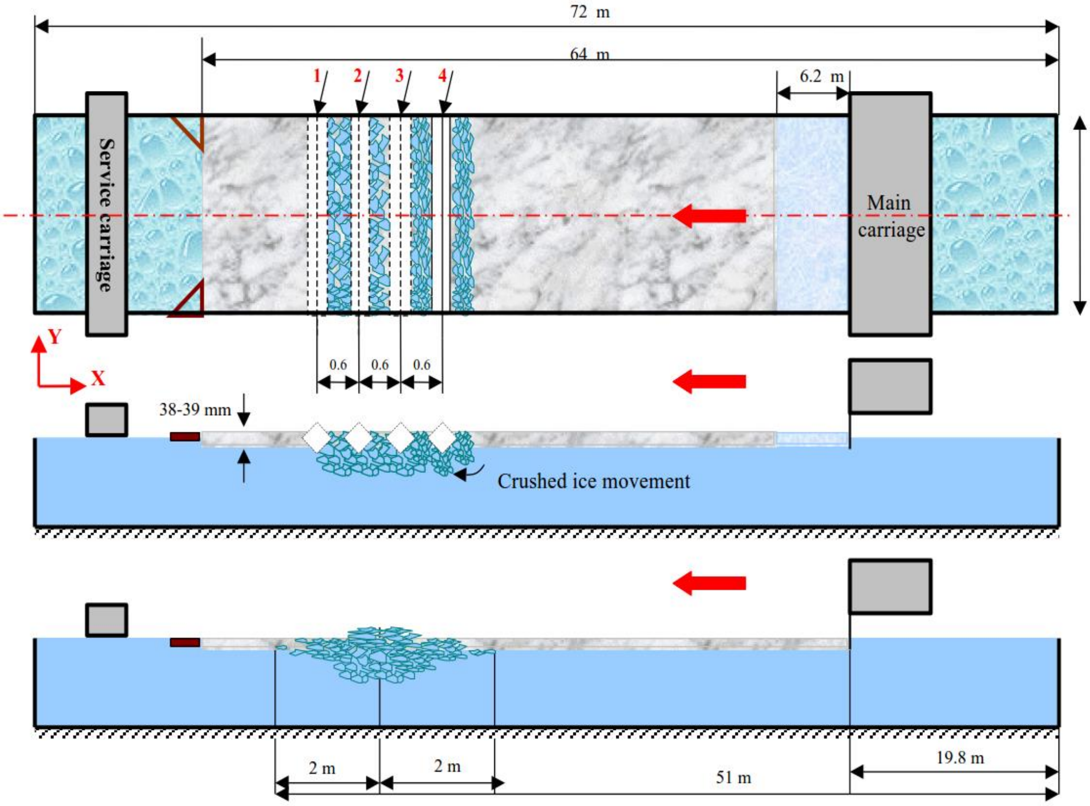


**Figure 13 Ice Ridge preparation Procedure[16]**

The standard procedure at HSVA is to build the ridge at low air temperatures to use the heat flow for consolidation, i.e., ridge fragments freeze to the level ice sheet pushed on top of the ridge keel portion.

## 4.3 Numerical Ridge Simulation in in-house software

The ice ridge creation in the software does not follow the same principles as the natural form described in Figure 12 or in the model basin described in Figure 13. The program creates an ice ridge according to the specifications of the ridge profile introduced by the user, which contains the ridge's geometry and the rubble ice's mechanical properties.

Initially, the program places polyhedrons under the surface of the water to modify ice particles. The particles float up to the free surface by the buoyancy force, and two crossed bars move towards each other and push the ice particles. Ideally, the length should not be too long to influence the boundaries; however, it should not be too short either. The simulation is done within a reasonable time [17].

The typical input parameters to run the following ice ridge formation has been mentioned in Appendix 01: Parameters to generate Ice ridge in the in-house software

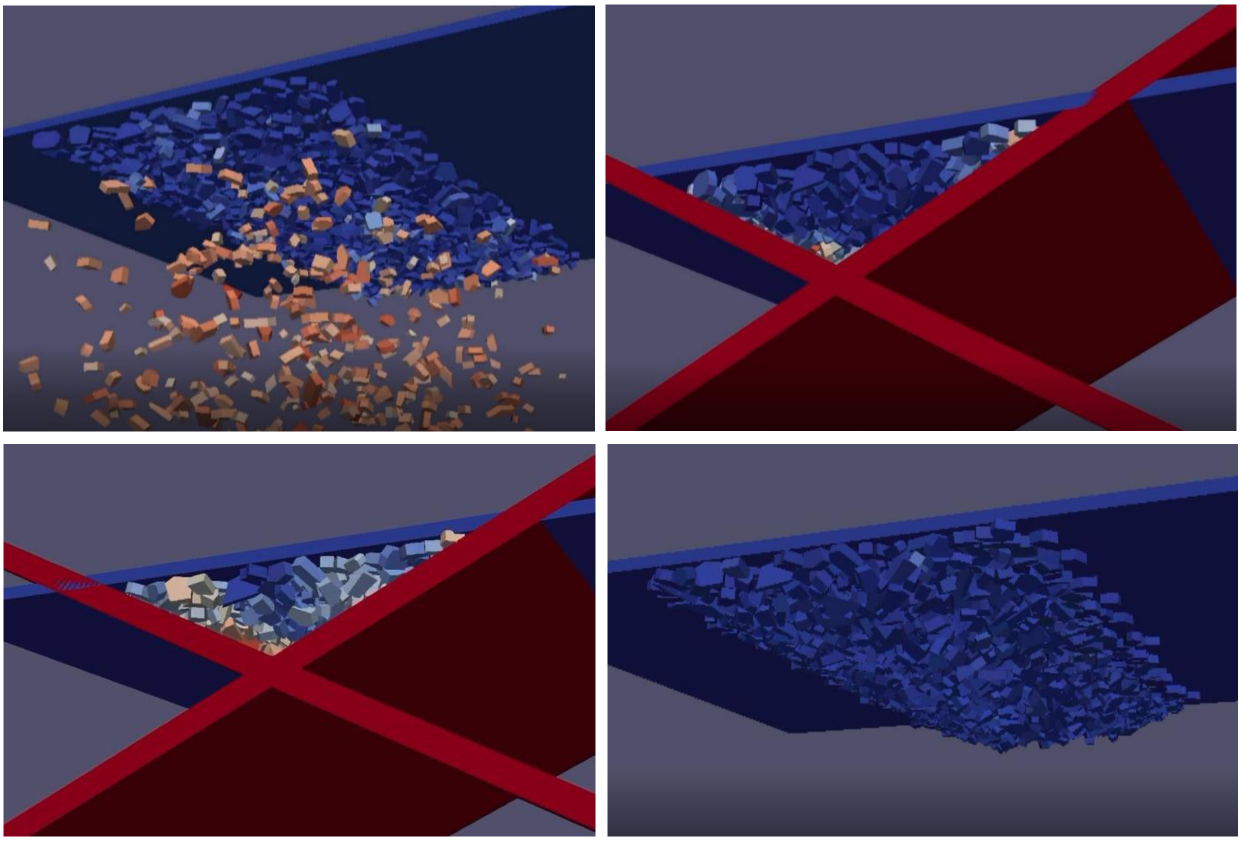


**Figure 14 Steps for creating a Ice ridge in the existing software**

## 4.4 Performance analysis for serial computation (CPU)

Some results have been obtained using the existing in-house software implemented entirely on the CPU. The time required to run the whole simulation has been discussed below.

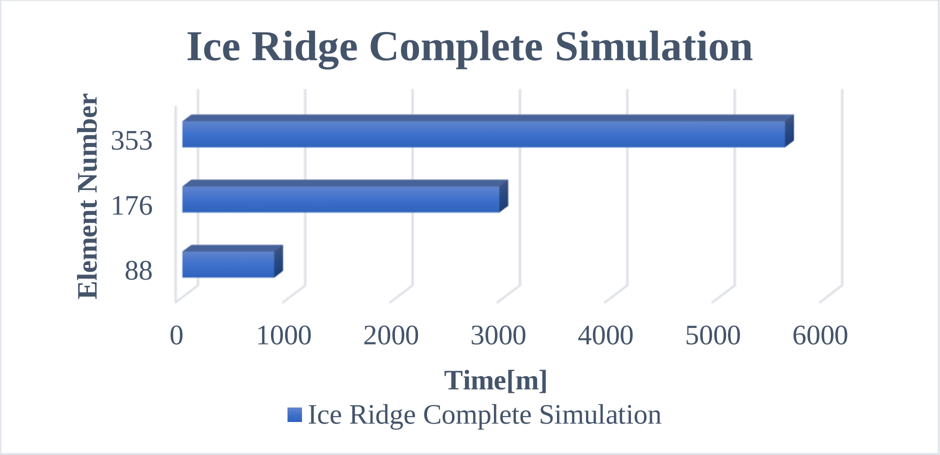


**Figure 15 Brash Ice Simulation in CPU**

The graph clearly shows that the program can do simulations in minutes or hours with more minor elements. Nevertheless, with a higher number of elements, it takes several days to complete the ice generation, which is much less effective. It takes even much more time compare to the brash ice simulation

So, to solve this problem, CUDA was introduced into the existing coding, as discussed in the following sections.

Chapter 5

# 5 Brash Ice Generation

## 5.1 Natural creation process

When one large piece of ice falls off another, brash is generated and can cover large amounts of the sea. From frequent ship passages, brash ice forms in harbours and ship channels, and the resulting freezing–breaking cycles create unique ice formations. The brash ice accumulation over the winter season results from meteorological, thermodynamical, and mechanical processes [18].

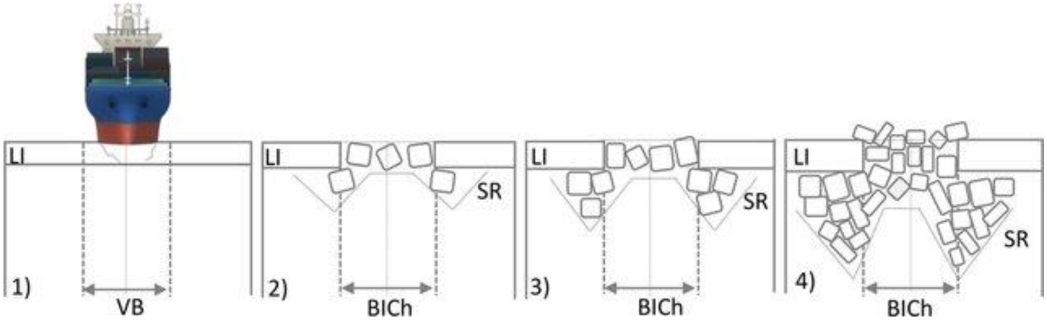


**Figure 16 Formation process of brash ice when ship passes through the channel[18]**

**Figure 17 Brash Ice channel in nature[16]**

## 5.2 Brash Ice in ice model basin

Firstly, a parental level ice sheet with pre-defined ice properties is prepared according to HSVA’s standard procedure for preparing a brash ice channel in the ice tank. Afterward, at a room temperature of around -2°C, an ice channel with straight edges is cut into the ice sheet. After that, the ice stripe between the two cuts is manually broken up into relatively small ice pieces using special ice chisels [17].

To achieve the most realistic appearance of the brash ice channel, sections where the ice pieces remain in a regular pattern, are carefully stirred. Once the first test run is completed, the ice pieces are rearranged in the channel and compacted to perform a second test run with thicker brash ice.

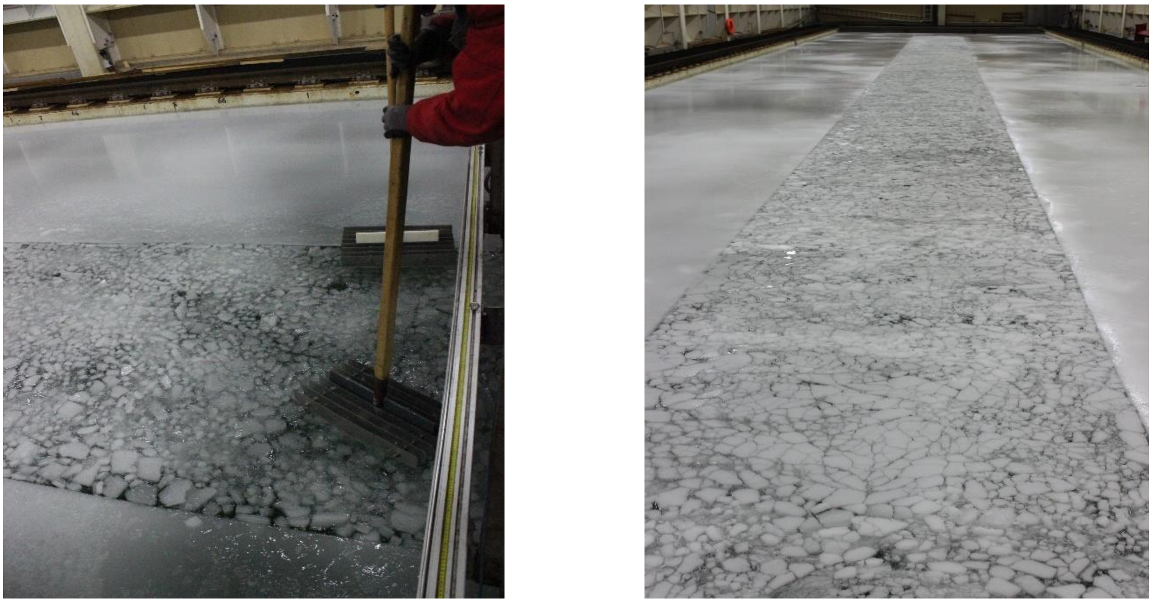


**Figure 18 Brash Ice prepapration procedure[16]**

## 5.3 Brash ice Simulation in in-house software

The brash ice creation in the software does not follow the same principles as the natural form described in Figure 17 or in the model basin described in Figure 18. The program creates a brash ice channel according to the specifications of the ridge profile introduced by the user.

The spheres model is used for the ice particles. The particles float up to the free surface by the buoyancy force. This way, the code generates spheres below the waterline with a random diameter and a random velocity.

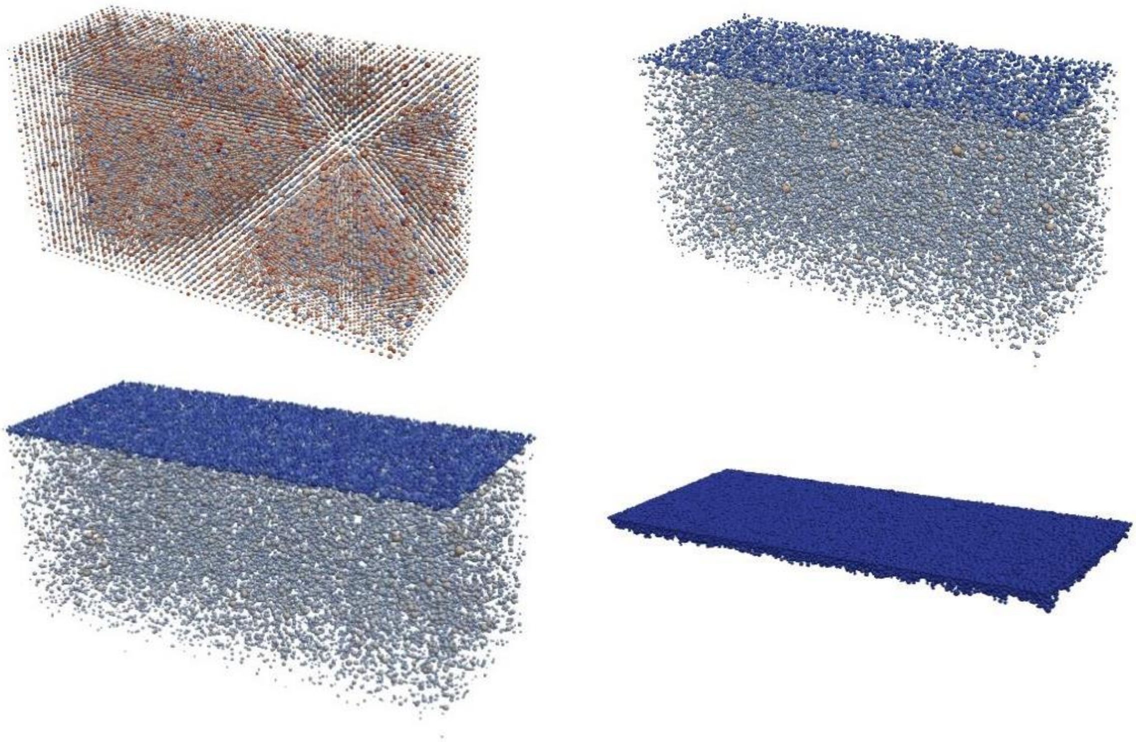


**Figure 19 Steps for creating Brash Ice in the existing software**

## 5.4 Performance analysis for serial computation (CPU)

Some results have been obtained using the existing in-house software implemented entirely on the CPU. The time required to run the whole simulation has been discussed below.

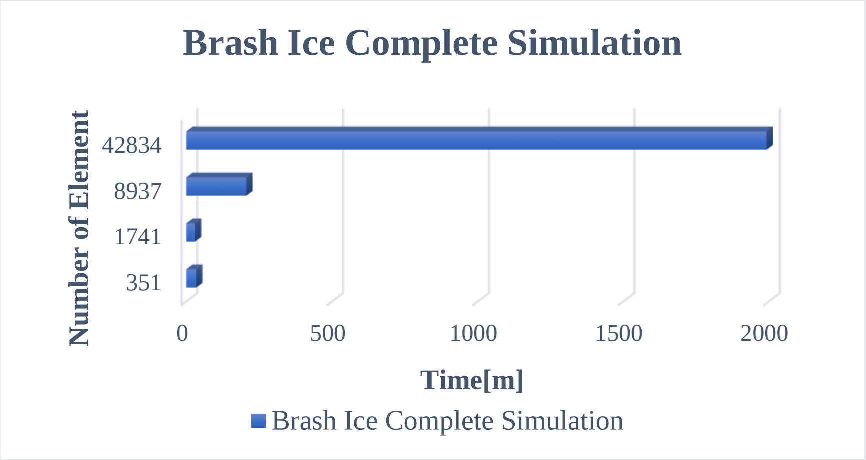


**Figure 20 Brash Ice Simulation in CPU**

The Figure 20 Brash Ice Simulation in CPUclearly shows that the program can do simulations in minutes or hours with more minor elements. Nevertheless, with a higher number of elements, it takes several days to complete the ice generation, which is much less effective.

The typical input parameters to run the following ice ridge formation has been mentioned in Appendix 02: Parameters to generate Brash ice in the in-house software

So, to solve this problem, CUDA was introduced into the existing coding, as discussed in the following sections.

Chapter 6

# 6 Implementation of CUDA

With CUDA, applications can run on hundreds of parallel processing elements and manage thousands of threads simultaneously. CUDA is designed to work with programming languages such as Fortran. This accessibility makes it easier for specialists in parallel programming to use GPU resources.

In this chapter, the implementation of CUDA in the in-house software has been discussed broadly. For instance, only the whole-time integration loop is transferred from CPU to GPU with the help of CUDA.

## 6.1 Defining Kernels

Implementing code on the GPU must be done inside special functions, typically called kernels. Compilers need extra specifiers to distinguish between regular functions and GPU kernels [8].

Based on the Fortran code, it is observed that the kernels are void-typed, and they return nothing. The typical workflow in void-type functions is to modify the result directly in the memory. In order to achieve this, the function must pass in an array with memory allocated for the result as an argument.

```
Kernel <<< BlocksInGrid, ThreadPerBlock >>> (Input1, Input2,····)
```

Calling the kernel from the central part of the program will also appear differently than regular functions. To determine which resources are needed, the parallel discretization parameters must be specified when calling the kernel.

## 6.2 Parallel Discretization

In general, running code on a CPU is done serially, which means that a particular process is performed in sequence concerning the data. In serial discretization of a simulation problem, the domain can be divided into cells and perform calculations for each cell in series.

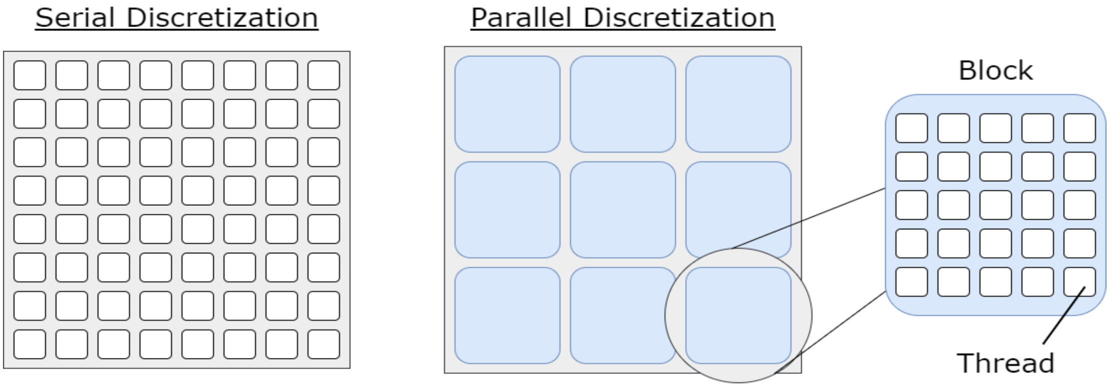


**Figure 21 Example of serial discretization vs parallel discretization[8]**

The discretization process involves two steps for parallel computing. As seen in the figure, the domain is divided into several blocks. The blocks contain threads that perform the calculations, similar to the serial case.

## 6.3 Indexing

The indexing for a parallel discretization is two-layered. An index identifies each block, and within each block, the threads are indexed locally by their position. Thread indices are not unique, but the combination of thread index and block index results in a suitable location in the total domain.

*Global Index = Thread Index + Block Index * Block Size*

Getting the global location of each thread requires some index mapping. The thread and block indices are hidden struct objects which every thread can uniquely access. For example, if the domain is two-dimensional, as in this case, the struct-objects, thread index, block index, and block size have an x and a y component.

In case of implementation in the in-house source code, only one-dimensional domain has been used throughout the coding.

## 6.4 Block Sizes and Grid Sizes

A logical question at this point might be how to be able to decide or determine the best size of the blocks. As it turns out, determining the optimal block size is not a straightforward process. It is both hardware and problem dependent. The general guidelines are a maximum of 1024 threads per block, i.e., 32x32, in two dimensions, and a block size evenly divisible by 32.

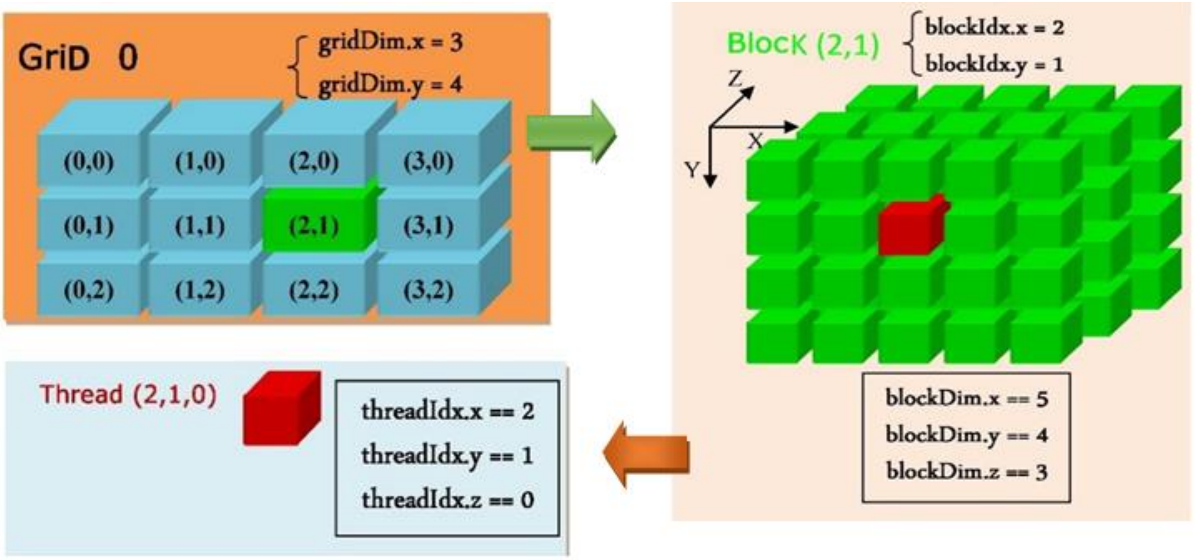


**Figure 22 Grid Hierarchy of Thread Blocks[19]**

It is the block size we care most about for optimization. However, to launch one thread per cell, one must create the correct number of blocks to cover the domain entirely. The set of all blocks in the domain is generally called a grid. The grid size is obtained by dividing the total domain size by the block size in each coordinate direction. Since this needs to be an integer, we cannot round down and thereby risk missing cells. We, therefore, ceil the divisions.

```
Grid Size = Ceil (Domain Size/Block Size)
```

However, this might lead to the kernel launching threads outside the domain, i.e., in unspecified memory. At the beginning of the kernel, one needs to add a conditional to ensure none of those threads are accessed.

## 6.5 Memory declaration and allocation

Memory management on a CUDA device is similar to how it is done in CPU programming. one needs to allocate memory space on the host, transfer the data to the device using the built-in API, retrieve the data (transfer the data back to the host), and finally free the allocated memory.

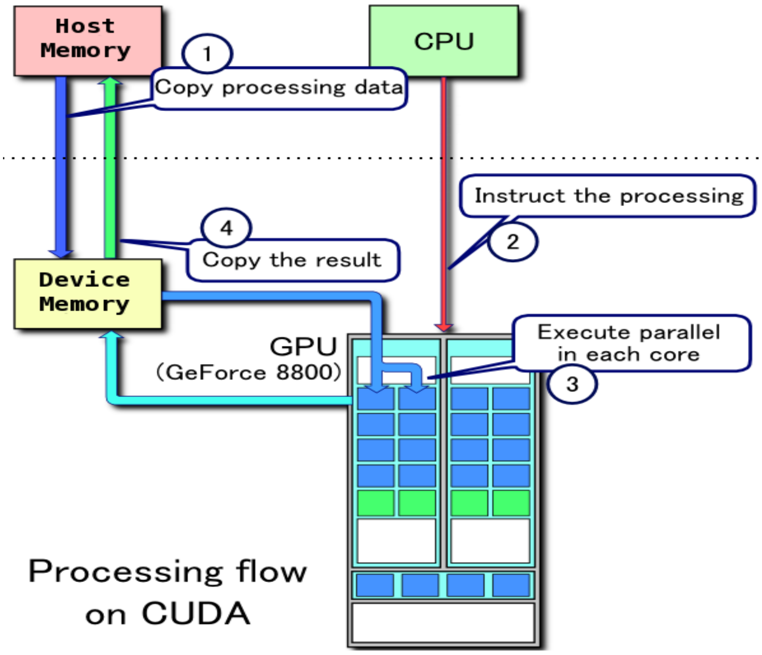
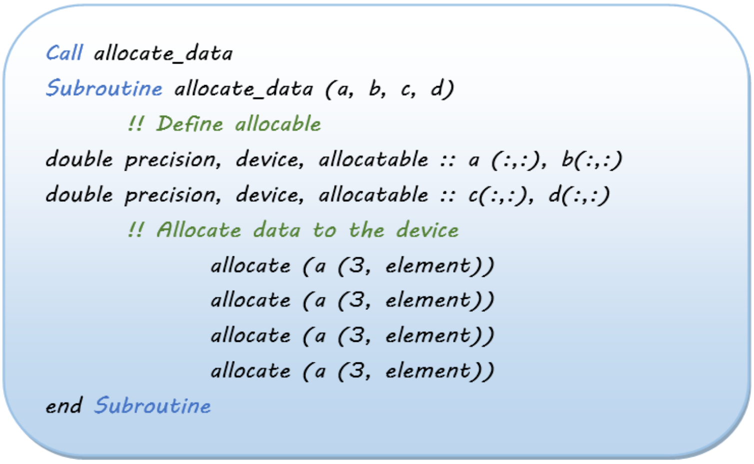


**Figure 23 Processing flow and memory defination on CUDA GPU[20]**

One needs to allocate memory on both host (CPU) and device (GPU) before transferring data between host and device. One can copy the kernel's input (a, b, c, d) from the host to the device by allocating the space in the device. Allocating space to copy the result from the device to the host can also be done later. A CUDA application manages the device space memory through calls to the CUDA runtime, including device memory allocation, deallocation, and data transfer between the host and device memory. Once memory space is allocated, it needs to transfer data back to GPU global memory from the device.



```
Call allocate_data
Subroutine allocate_data (a, b, c, d)
        !! Define allocable
double precision, device, allocatable :: a (:,:), b(:,:)
double precision, device, allocatable :: c(:,:), d(:,:)
        !! Allocate data to the device
                allocate (a (3, element))
                allocate (a (3, element))
                allocate (a (3, element))
                allocate (a (3, element))
end Subroutine
```

**Figure 24 Example of data allocation in device**

## 6.6 Shared memory

The shared memory is a user-managed memory allocation, unlike the caches, which are system-managed allocations based on recently accessed data, which means that the user has complete control of what and how data is stored.
Shared memory is used for memory optimization in many general parallelization cases with the same size as the block. Generally, the optimization revolves around avoiding latency from global memory calls if some values are to be used multiple times in a kernel [8].

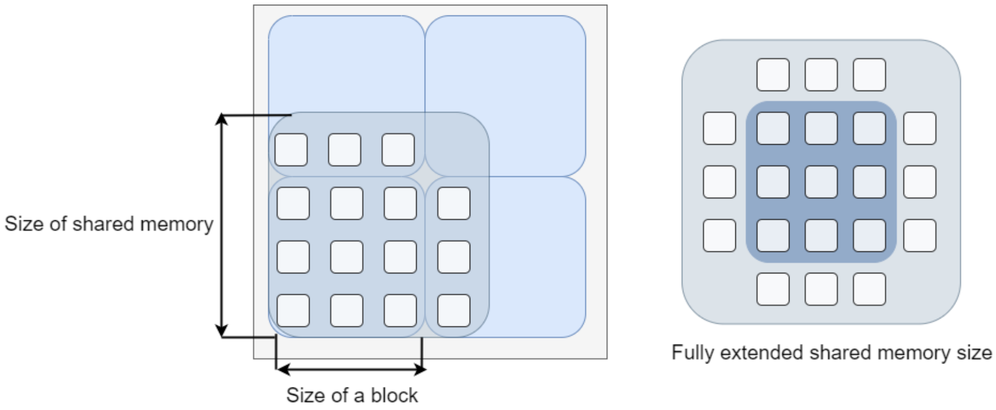


Figure 25 Example of Shared memory[8]

In order to solve the intercommunication problem, one can expand the shared memory to cover an area larger than the block size. The expanded area will cover cells in neighbouring blocks, which means that every thread in the current block will have access to its neighbours.

## 6.7 Synchronization

An important concept when working with shared memory is synchronization, or more particularly, synchronization of threads within a block. The synchronization acts as a barrier, stopping the threads until every thread in that block has reached that point. If several threads are running through code in parallel and this process is writing to shared memory (just like the previous code snippet), not every thread might be in sync at a given location in the code. If a thread tries to access shared memory for a cell for which another thread has not yet finished its calculation, this can cause a memory conflict.

Syntactically, there is no easy way to create a global synchronization. Only the blocks of the threads are valid for synchronization.

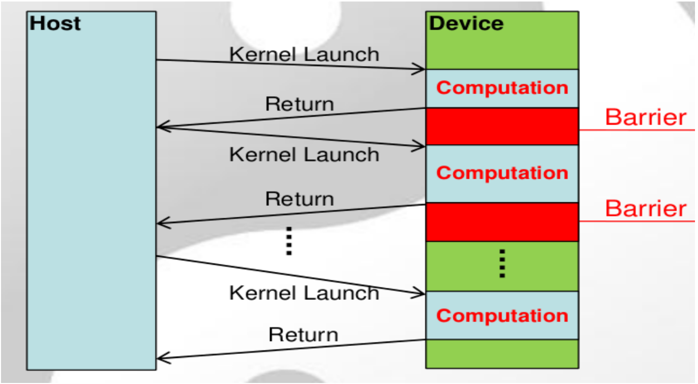


Figure 26 Example of simple data Syncronization[21]

## 6.8 Occupancy

For the maximal performance of the GPU, one needs to maximize the occupancy. GPU usage varies based on the number of SMs used, or how parallelized the code becomes during execution. The code could, for example, run on one SM, which would cause low occupancy, or it could utilize every SM, which should significantly enhance the performance. Utilization of the SM is determined by the size of the blocks.

If the parallel discretization consists of very few blocks, then the code will only load a few of the SMs. By introducing many blocks with significant sizes, the GPU can maximize parallelization.

## 6.9 Nvidia Visual profiler

NVIDIA profiling tools are used for optimizing the performance of CUDA applications. Profile data is collected by default over the entire application run.

The Visual Profiler can collect a trace of the CUDA function calls made by an application. The Visual Profiler displays a timeline of the application's activity on both the CPU and GPU to identify opportunities for performance improvement. In addition, the Visual Profiler will analyze the complete application to detect potential performance bottlenecks and direct the coder on how to take action to eliminate or reduce those bottlenecks. The Visual Profiler does not require application changes; however, by making some simple modifications and additions, one can significantly increase its usability and effectiveness.

In addition to the guided analysis results, one will see a timeline for the application showing the CPU and GPU activity that occurred as the application was executed. Several examples of the profiler has been shown in the annex-01

**Chapter 7**

# 7 Implementation Issues

CUDA is a parallel programming language. By reducing the time and cost of full-scale computation, ship designers can optimize vessel performance. A rewrite or targeting for GPU computations is not always the best way to improve overall performance.

While re-writing the Time integration into parallelized code, several costly and non-evident bugs were introduced by accident. The following section will discuss some of these problems, why we believe they occurred and how they were fixed.

## 7.1 Lack of intercommunication

In parallel kernels, the only possible communication between threads is within a thread block. If there is no intercommunication among cells, performing computations on one cell results in problems for neighbouring cells. The figure illustrates how a lack of intercommunication can cause big problems. As a result, cells on block boundaries cannot access all of their neighbours. In a DEM program, the use of the neighbouring cells is particularly significant.

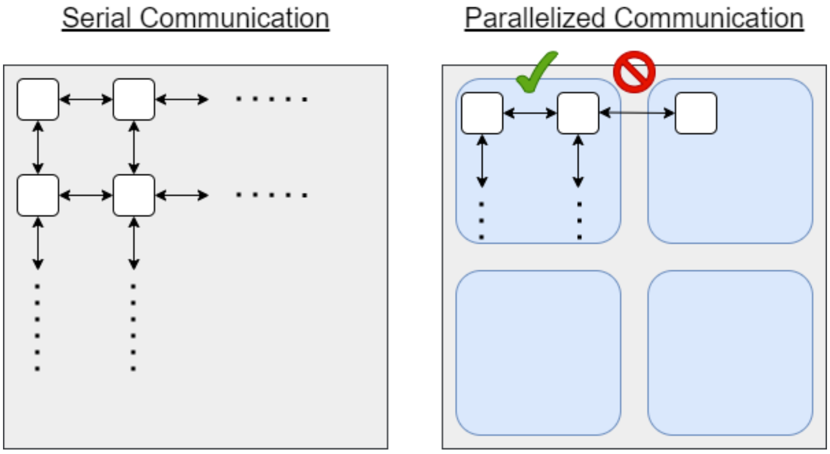
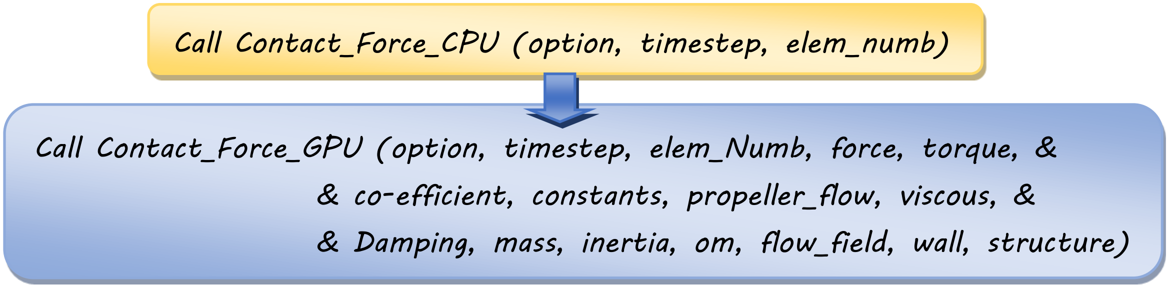


Figure 27 Data sharing mechanism CPU vs GPU

One of the ways to resolve this problem is to use a resource called shared memory, which can uniquely store a certain amount of data inside each block.

## 7.2 Redefining the original Function

Initially (for CPU), the code was split into several files for improved structure and readability, where each kernel got its file. A long-term issue was to allocate the data to GPU and to define the shared memory size in a way that was adaptive to each kernel. This issue is because, in Fortran, the shared memory size must either explicitly be defined earlier or remain constant. So, the variables need to be defined globally, a scope of them being treated as constants.



```
Call Contact_Force_CPU (option, timestep, elem_numb)
```

```
Call Contact_Force_GPU (option, timestep, elem_Numb, force, torque, &
                        & co-efficient, constants, propeller_flow, viscous, &
                        & Damping, mass, inertia, om, flow_field, wall, structure)
```

**Figure 28 Sample of redefining the call function for CPU to GPU**

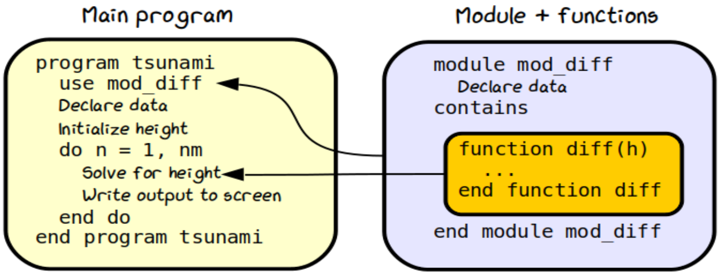
However, in GPU, it is necessary to redefine each function used in the separate file to ensure it receives all the data it needs. To define the shared memory's size in an adaptive way, every kernel needs to be in a file where the block size was created in the global scope. Unfortunately, this was accomplished by adding all kernels to the main Fortran file, which severely harms the code's readability.

The example shows the difference between an old and new function definition and how it needs to be redefined after transferring data for GPU calculation.

## 7.3 Defining new kernels

Kernels cannot handle arrays with only parts passed into them. Some coefficients were collected into a typical three-dimensional array where the third dimension separated the coefficients. For some kernels, only a few of the coefficients were needed; in those cases, they were passed in individually by accessing the third dimension. Due to this process, the script became increasingly slow for large domains.

It is observed that the reason for this is that accessing the variable like inbuild basic Fortran functions (i.e., Cross product, normalization vector) are unable to access by CUDA. So, it was necessary to regenerate the entire array for the device and access the coefficient from within the kernel to solve the issue (i.e., a memory location). At the same time, these new functions cannot perform analysis for complex equations. Instead, those functions can only work with more exact values, making the work more time-consuming.



```
program tsunami
  use mod_diff
  Declare data
  Initialize height
  do n = 1, nm
    Solve for height
    Write output to screen
  end do
end program tsunami
```

```
module mod_diff
  Declare data
contains
  function diff(h)
    ...
  end function diff
end module mod_diff
```

**Figure 29 Example of defining new function[22]**

Passing large arrays into a kernel one time does not result in a performance hit since they are passed in by reference. Nevertheless, defining the kernel for each type of array and allocating the memory repeatedly can slow down the computational time.

## 7.4 Debugging difficulties

Another aspect in determining whether a parallel implementation will be worth the time and effort is the debugging difficulties associated with writing kernels. A kernel cannot write to an output file or to the console. To write an output file or display anything on the console, the data first needs to be transfer from the device to the host (CPU) and then written by the CPU.

Although it is easy to implement the general workflow, debugging intermediate computations with a CUDA kernel is notoriously tricky. Serial code allows breakpoints slowly step over lines to check for unexpected events. A kernel does not support this type of debugging. In order to access intermediate results computed within a kernel, the programmer must find other creative ways.

One effective way in this project was to write intermediate results to one input arguments of the Kernel and terminate the kernel early.

## 7.5 Graphical output

Another issue encountered during the implementation of CUDA is the GPU processor's inability to provide the graphical output of the parallel calculations made in the GPU. Getting the analysis's complete visualization requires transferring all video output data back to the CPU and generating output files (i.e., .vtk file). The re-assigning process is also time-consuming, affecting the software's overall performance.

## 7.6 CUDA Fortran compiler

C++, Fortran, and Open ACC directives are all supported by the NVIDIA HPC SDK compilers for HPC modelling and simulation applications. With support for NVIDIA GPUs and Arm, Open POWER, or x86-64 CPUs running Linux, the HPC SDK provides the tools one needs to build NVIDIA GPU-accelerated HPC applications [23].

It is excellent and works flawlessly on Linux without any hassle. Even the installation process is easier for the PGI CUDA Fortran compiler, which was the HPC-SDK predecessor (for CUDA Fortran, at least). However, the HPC-SDK package is not available for Windows. The free (community edition) versions of the PGI compiler used to have downloadable versions for Windows and C programming language [24].

Unfortunately, writing the whole codebase in C is not an option. Fortran paired with C or CUDA Fortran, and the move to Linux workspace is the choice.

Chapter 8

# 8 Result Analysis

This chapter represents the validation of the results that have been obtained after implementing CUDA. To do so the CPU and GPU simulation results compared to the tank test results. Therefore, the main results of interest are the average iteration times compared between the GPU and CPU. Two separate cases (Brash ice and ice ridge creation) are taken to obtain the average iteration time.

Though it takes considerable time to complete each simulation of brash ice and ice ridge in both CPU and GPU, a fair comparison is taken by considering 15-20 iterations for both serial and parallel implementation. In the first section of the result, brash ice simulation performs successfully in both CPU and GPU. In other sections of the result, only the initial iteration has been considered for further analysis. However, the main parallel and serial implementation patterns are still very evident. Later, the result obtained from CUDA code is compared with tank test data and CPU to check the correct implementation of CUDA code conversion.

However, the target result is not obtained from the CUDA analysis. The possible reason behind this failure in result has been discussed in each section.
The computer specification used for the analysis is discussed in the Appendix 05: Computer Specification.

## 8.1 Validation of performance

### 8.1.1 Ice channel formation

In this section, the performance of the software before and after implementing CUDA are compared. The performance obtain in both scenerio stays similar. On observing the graphical output (Figure 30), it is clear that it has been showing similar technique to generate brash ice or ice ridge in both cases. The file is obtained for both sceneiro of before implemention and after implemenetion of CUDA to the software. In both cases, floating-up techniques is used to generate an ice channel.

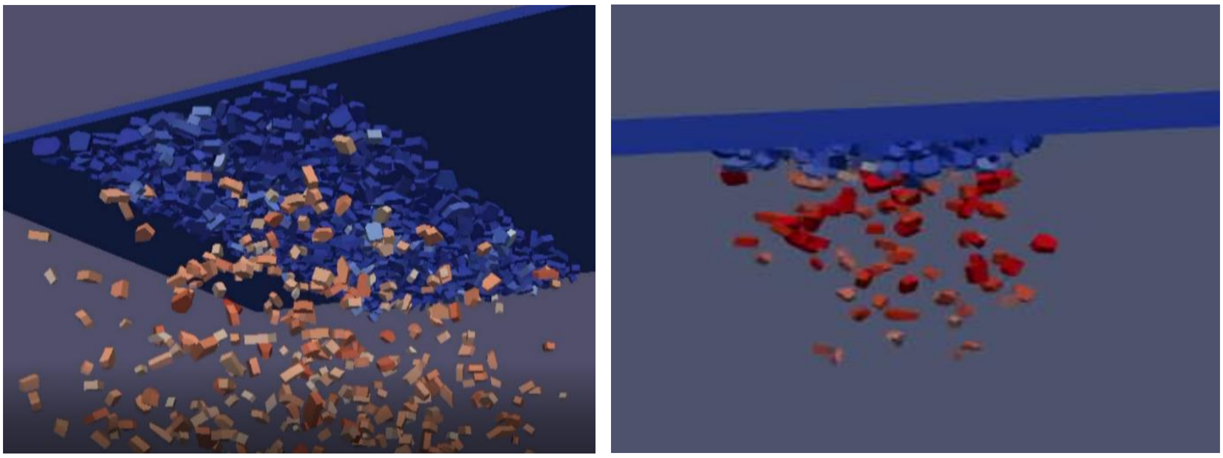


**Figure 30 Floating up performance for Ice ridge CPU(on left) vs GPU (on right)**

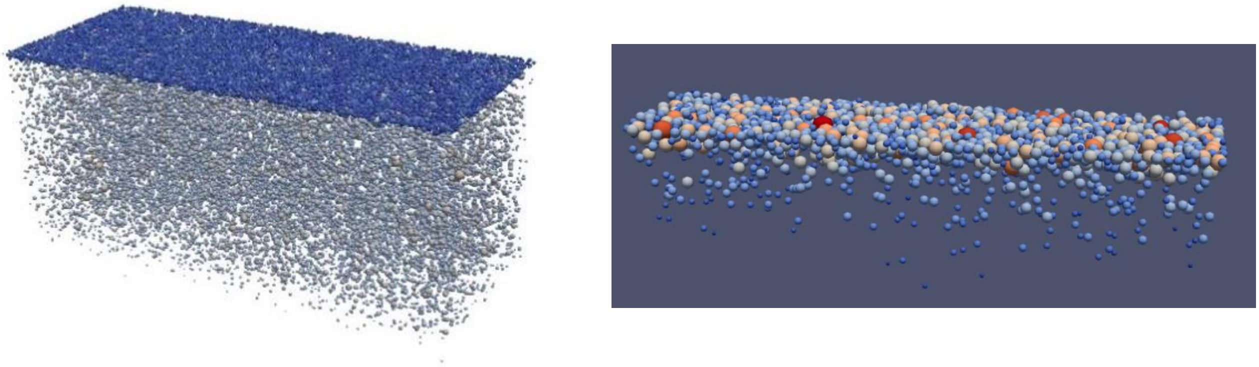


**Figure 31 Floating up performance for brash ice CPU(on left) vs GPU (on right)**

### 8.1.2 Geometrical Characteristics

Initially, only 10-15 iterations are analyzed.The comparision of brash ice radius or ice-ridge geometrical characteristics (wall, area, thickness, length) at initial stage shows similar characteristics for both CPU and GPU.

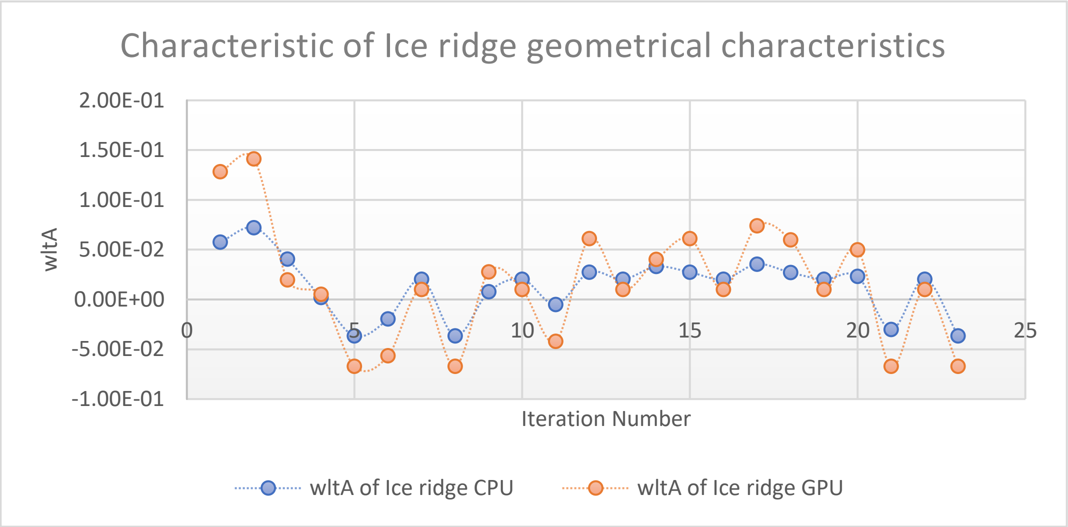


**Figure 32 Geometrical characteristics of Ice ridge (CPU vs GPU)**

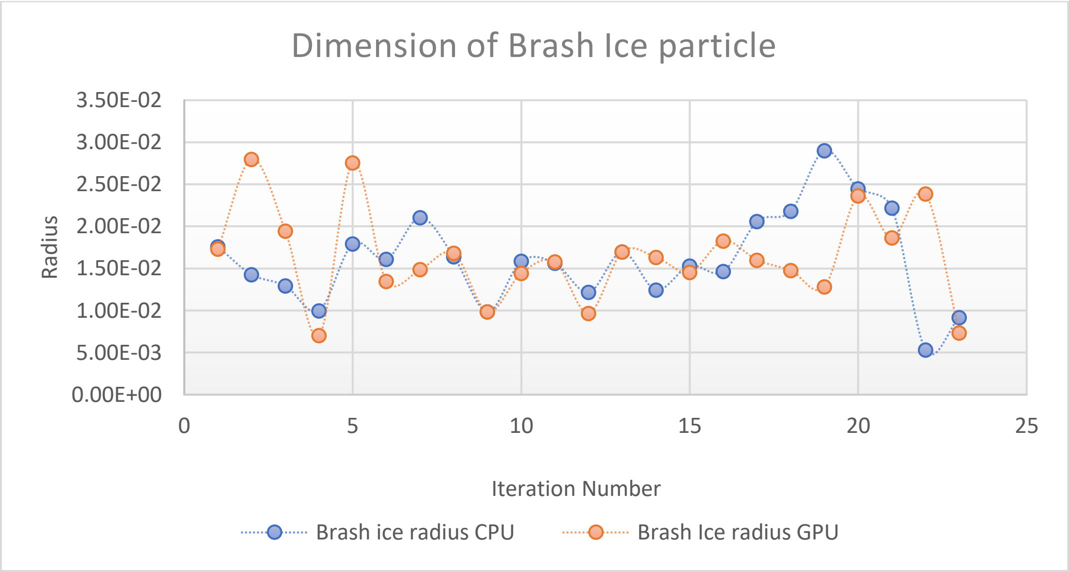


**Figure 33 Dimension of Brash ice particle (CPU vs GPU)**

## 8.2 Validation of performance considering computational time

### 8.2.1 For ice ridge

From the Figure 34 it can be seen that, all the values obtained for CPU and GPU show the same trend of increase with the increase of element number. So, it can be said that the performance obtained from both implementations gives similar values.

But it is clear that computation time required to run the simulation in GPU is 2 times more than the time required for CPU. The software shows better performance compared to the results obtained for brash ice generation. But it can also observe that the time required to run the simulation is fully dependent on the element number. With the increase of element number, the computation time increases and it's the same case for the CPU as well.

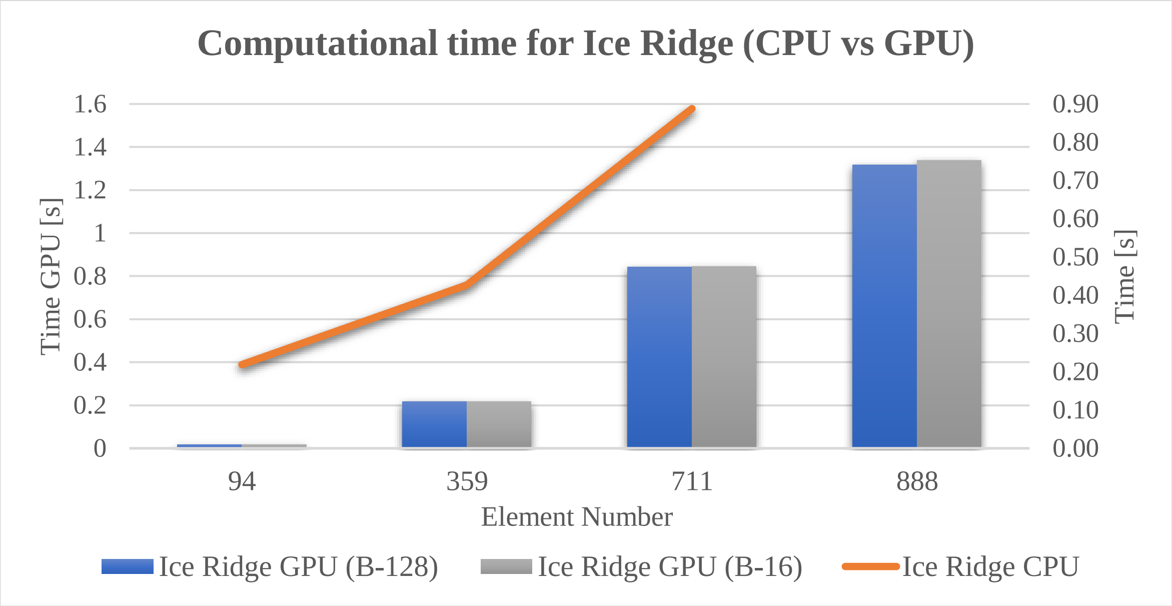


**Figure 34 Computational time for Ice ridge (CPU vs GPU)**

Changes in Block size for computation didn't affect much in the iteration time for small number of iterations. Though higher iteration may show some significant difference in the computation time.

### 8.2.2 For brash ice

Figure 35 represents the time required to form brash ice channel obtained for CPU and GPU show the same trend of increase with the increament of element number. So, it can be said that the performance obtained from both implementations gives similar values.

From the Figure 35 , time required to run the simulation in GPU is 100 times more than the time required for CPU. But it can also observe that the time required to run the simulation is fully depended on the element number. With the increase of element number, the computation time increases and it's the same case for the CPU as well.

Changes in Block size for computation didn't affect much in the iteration time for small number of iterations. Though higher iteration may show some significant difference in the computational time.

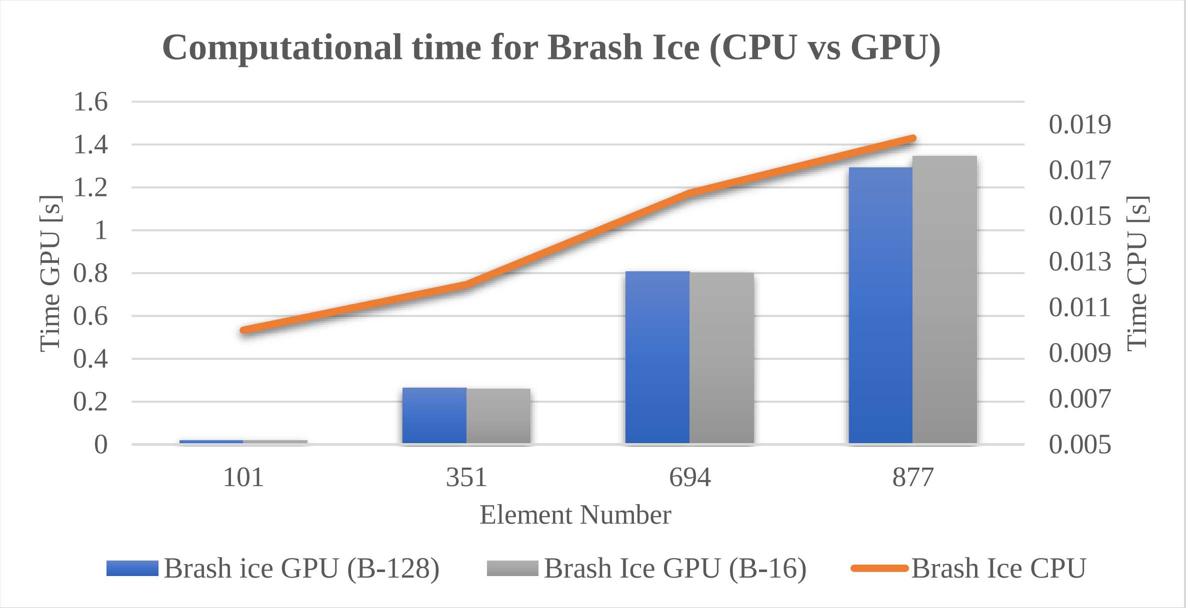


**Figure 35 Computational time for Brash Ice (CPU vs GPU)**

## 8.3 Performance analysis for Different Element Number

Figure 36 and Figure 37 shows that the average time of integration for both brash ice and ice ridge formation. To obtain the quired time for each iteration, the initial element number has been considered So that the simulation can be performed for exact number of elements for CPU and GPU each time.

### 8.3.1 Ice ridge generation

Two different points can be found in Figure 36. Firstly, with the higher element number, the performance ratio for contact detection is increased significantly. Also, around 99% of the whole calculation time is dedicated to find the contact detection between the particles.

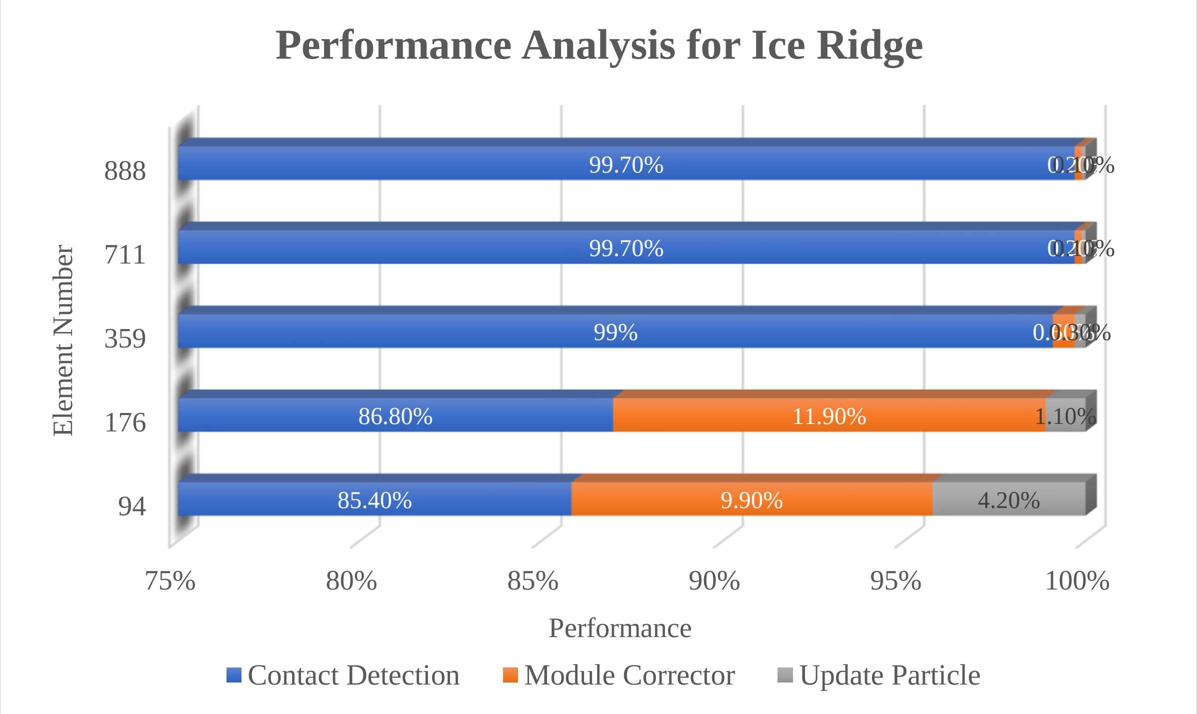


**Figure 36 Performance Analysis for Ice ridge**

Secondly, for the lower number of elements, the module corrector and update particle also pay a significant amount of time for the iteration, which is not visible for higher element numbers.

To find out the overall performance, Nvidia Visual Profiler has been used. Details of the performance are described in Appendix 04: Profile visualization (Ice Ridge)

### 8.3.2 Brash ice formation

Two different points can be found in the above performance graph (Figure 36). Firstly, with the higher element number, the performance ratio for contact detection is increased significantly. For higher element numbers, around 99% of the whole calculation time is dedicated to finding the contact detection between the particles which is similar for ice ridge creation.

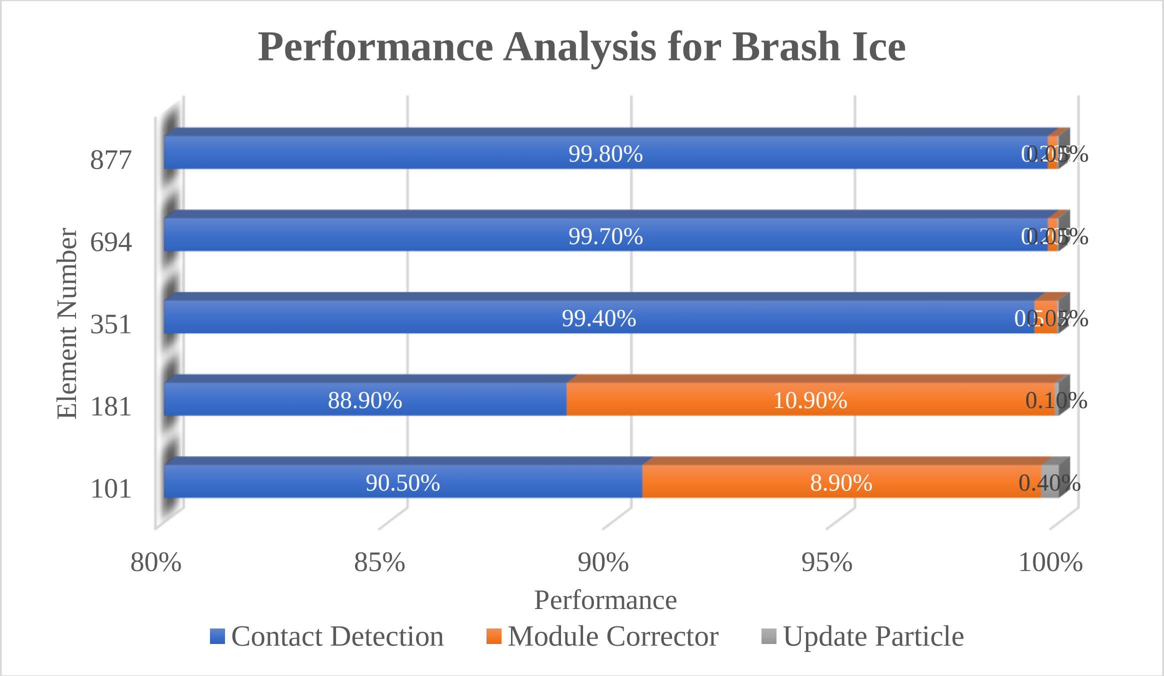


**Figure 37 Performance analysis for Brash Ice**

Secondly, for the lower number of elements, the module corrector also pays a significant amount of time for the iteration, which is not visible for higher element numbers. However, the brash ice generation update particle did not require many contributions to the simulation.

## 8.4 Performance analysis for Different number of Blocks

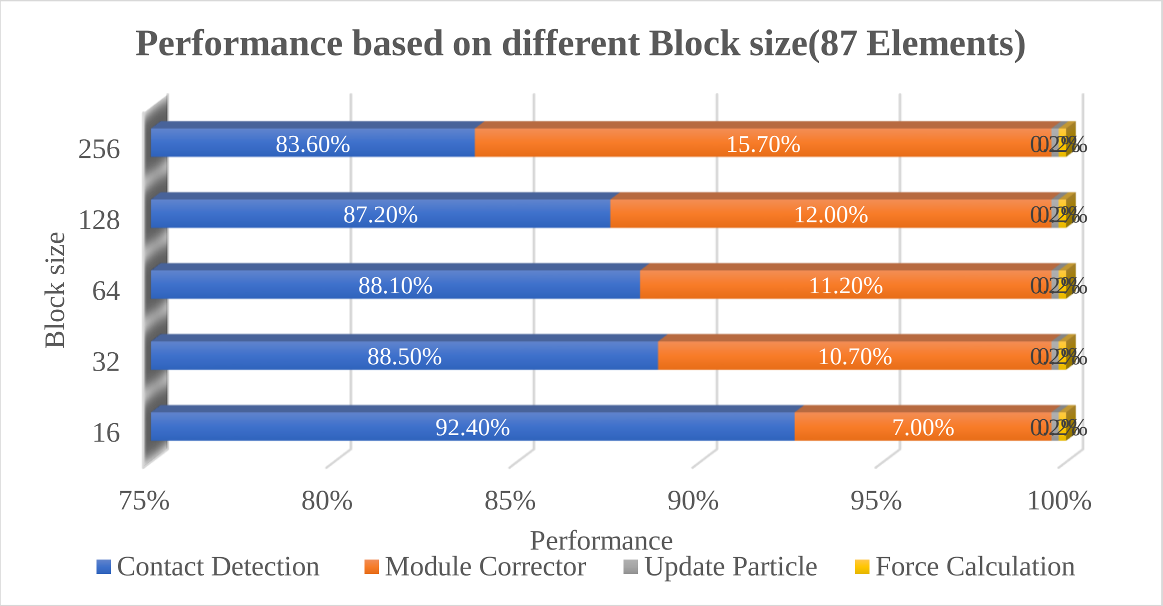


**Figure 38 Performance based on different block size (87 elem)**

Figure 38 and Figure 39 is the visual representation where performance based on the number of blocks are observed . Brash ice and Ice ridge show similar characteristics, so here only brash ice generation condition has been considered.

Figure 38 represents that contact detection caused much more time for the small block size for higher element numbers. It also represents that contact detection takes a lower time to compute within the case of block number, but the corrector takes around 15% of the total computational time.

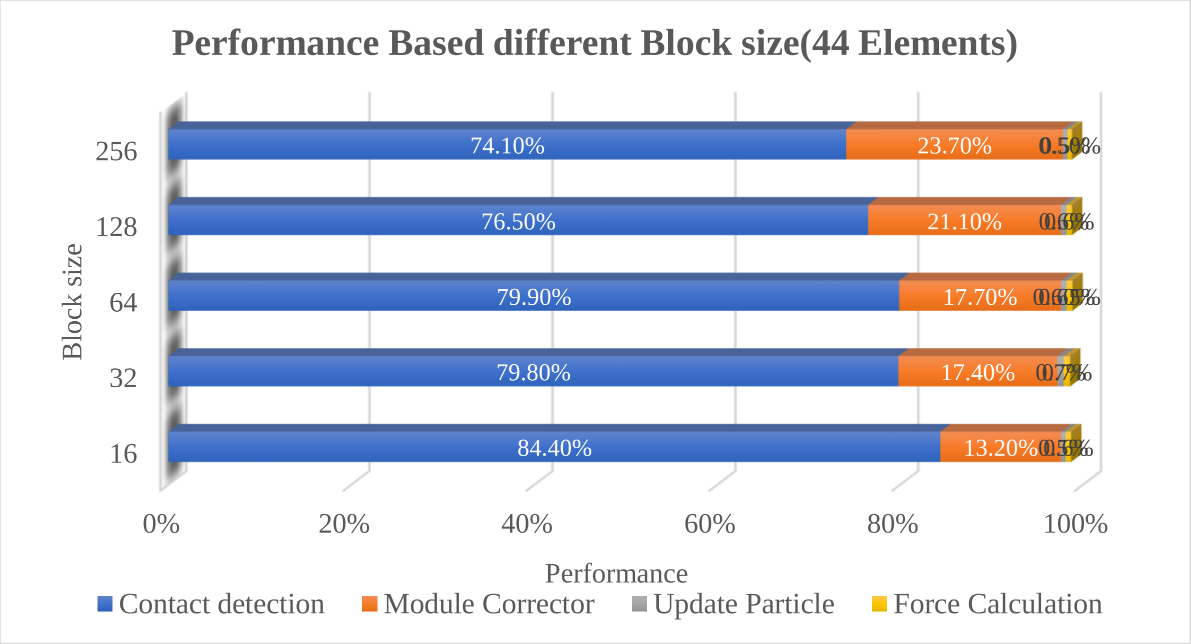


**Figure 39 Performance based on deferent Block size (44 elem)**

Figure 39 shows the small number of elements can show better performance with increased block size. It also represents that contact detection takes a lower time to compute within the case of block number, but the corrector takes around 23% of the total computational. Other factors do not make many contributions to the overall computation.

## 8.5 Time consumption for Data Migration

In this section the time consumption to data transfer from CPU to GPU has been discussed. To get the appropriate values regarding Data migration, Nvidia visual profiler has been used. For the detailed idea please refer to Appendix 03: Profile visualization (Brash Ice)Appendix 04: Profile visualization (Ice Ridge)

### 8.5.1 Ice ridge generation

In Figure 40 the time consumption for data transformation compared to the total computational time has been discussed. Comparing to the total computation time, data migration takes a substantial amount of time when the element number is minor. As the number of elements increases, the time spent on data transfer decreases. So, it can be said that GPU performs better with a higher number of elements.

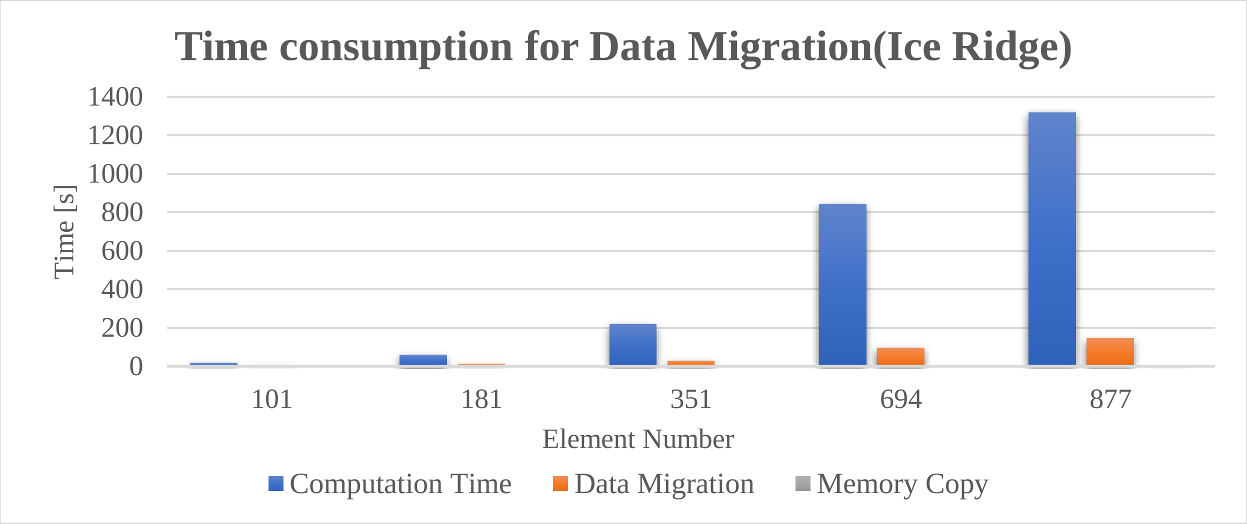


**Figure 40 Time consumption for Data migration (Ice ridge)**

### 8.5.2 Brash ice generation

In the case of brash ice generation, similar characteristics can be seen in the data transformation it can be observed in the previous section. Using a higher element number the effect of data transfer on the whole computation time reduces, as shown in the following figure.

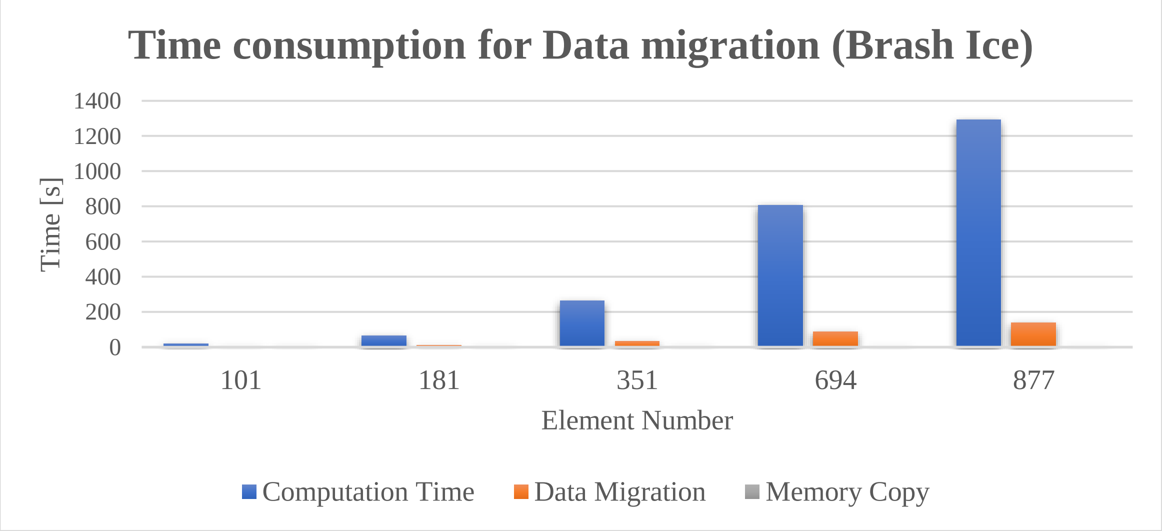


**Figure 41 Consumption for Data migration (Brash Ice)**

Chapter 9

# 9 Discussion

In general, discussing the results of this project will mainly cover two areas. First, a short investigation; while it will be better to code only on CPU or not based on their benefits and disadvantages, it can be a better choice as it is much easier to debug the overall program.

The second part involves a few major implementation problems while it was faced during CUDA implementation. Moreover, to come in a decision that will be a better choice than CPU or not.

## 9.1 When CPU a better choice

A CPU is flexible and resilient, and it can handle a variety of tasks other than graphics processing. The CPU can multitask across multiple activities in the computer because of its serial processing capabilities. For the same reason, a strong CPU can provide more speed for typical computer use than a GPU. In specific situations, the CPU will outperform the GPU. For example, the CPU is significantly faster when handling several systems (random access memory, mid-range computational operations, managing an operating system).

CPU works perfectly fine when there is no need for parallel implementation, and the simulation is done for only a smaller number of elements. In such a case, it does not require any time to transfer and allocate data which takes much longer while implementing the calculation in GPU. In Performance analysis for serial computation (CPU) and Performance analysis for serial computation (CPU) also, we have observed similar results as it takes only minutes to complete the simulation for 351 elements, whereas it takes days to complete the 42834 elements.

Local cache memory in CPUs enables them to handle a more significant number of linear instructions and, therefore, more complex computations. So, it can work efficiently without requiring data allocation and definition separately. It improves the overall capability of the CPU for the simulation of smaller element numbers.

CPUs cannot handle parallel processing like GPUs, so large tasks requiring thousands or millions of identical operations will clog the CPU's processing capacity. In that case, it is observed that neighbouring calculation, which requires parallel implementation, does not work so effectively as it requires multi-core processing.

## 9.2 When CUDA is worth implementing

The primary difficulty in having high-performance GPU code is that one has a ton of cores, and one wants them to all be utilized to their full potency as much as possible. Problems with irregular memory access patterns or not having high arithmetic intensity make this problematic: Either the coder need to spend a long-time for communicating results , or for fetching stuff from memory. Of course, the potential for concurrency in code is critical to its ability to be implemented well on GPU.

Considering the same idea when we tried to implement simple examples in one of the parts of the main calculation, like updating the bounding box, which is a straightforward algorithm and done only once during the initialization, it makes the whole computation time higher than the usual one. The CPU transfers data to the GPU and then back to the CPU at the end of the computation, which consumes the most significant amount of time. So, it is not efficient when one needs to transfer data between CPU to GPU. These simple cases can be ignored to implement in CUDA.

The benefit of GPU programming vs. CPU programming is that one can gain massive speedup for some highly parallelizable problems. If the whole simulation is run several times and has many elements, then it is always a better choice to switch to parallel implementation. From Time consumption for Data Migration, it is also noticed that transferring data between CPU to GPU is much less than the time required for implementation for a higher number of iterations. So, it makes sense to switch to GPU for more significant numbers of elements.

**Chapter 10**

# 10 Future Scopes

This chapter aims to focus on the future possibilities for the calculation that could have been done to improve the software's execution time.

The basic coding shows that the neighbour detection algorithm is implemented in the way of a serial computation algorithm. Even if it is asked to run this part of the code in a divide, it runs serially in a single thread. It is assumed that this is why CUDA implementation could not improve the overall performance. Changing this algorithm into a parallel computation algorithm could solve the overall performance of the software.

Another basic idea could be implementing the full software in a parallel implementation system. Right now, only the time integration loop is working in GPU, where initialization is working in CPU, which could be the possible solution to improve the performance.

Due to lack of time, it was impossible to run the whole simulation considering the ship or model passing through the channel. The code can improve further to simulate for any model test to check the capabilities of the structure.

As it has been discussed earlier, CPU remains idle while the GPU is performing the whole-time integration simulation. It affects the overall performance of the software. So, while GPU is working on calculating neighbours or detecting force, at the same time, the CPU can also perform more minor calculations without affecting the simulation running in GPU. Finally, a CPU-GPU combined implementation can be used to overcome this problem.

# Chapter 11

# 11 Conclusion

This chapter discusses the conclusion we have obtained by analyzing the results above. Firstly, the subroutine running only once during initialization should be done in CPU as it will not affect much in improving the performance. It does not affect if serial computation or parallelization is introduced.

Another factor has been noticed that around 99% of the computation time for a more significant element number is to find the neighbor detection, which makes the GPU performance very low. It can also be possible to run this calculation on the CPU. It can be possible to improve the overall performance of the whole software.

In some cases of the Time integration loop, there is no requirement for parallel implementation. Introducing them with parallelization will require more time to implement in the coding. However, in some consideration, it will not affect the overall performance of the software.

Data Allocation plays a vital role during the whole simulation. Data allocation can be done at the beginning of the process rather than in each kernel. From section7, it can be seen that in some iteration cases, the data allocation time can be similar to the simulation execution time. In this GPU version of coding, we have already introduced a separate kernel for data allocation, which was implemented at the beginning of the coding, which changes the execution time much more.

One last conclusion can draw that, block definition plays a vital role during simulation. It can be seen that, smaller block size means it takes much more thread to complete the simulation or can be a problem in memory allocation. We have also seen that for this particular case, a block size of 128 seems to be the optimum size for maximum performance for this simulation over several GPUs. This particular block size is enough to run the whole simulation without showing any memory allocation problem.

# References

[1] “European Federation for Transport and Environment AISBL.” [Online]. Available: https://www.transportenvironment.org/challenges/ships/arctic/

[2] L. H. Fang Li, “A Review of Computational Simulation Methods for a ShipAdvancing in Broken Ice,” Jan. 2022, [Online]. Available: https://www.researchgate.net/publication/358139815_A_Review_of_Computational_Simulation_Methods_for_a_Ship_Advancing_in_Broken_Ice

[3] C. Jallal, “Ice navigation: the expert’s view,” Dec. 03, 2020. [Online]. Available: https://www.rivieramm.com/news-content-hub/news-content-hub/ice-navigation-the-expertrsquos-view-62159

[4] Jens Krüger, Rüdiger Westermann, “Linear algebra operators for GPU implementation of numerical algorithms,” Jul. 2003, [Online]. Available: https://dl.acm.org/doi/10.1145/882262.882363

[5] Jeff Bolz, Ian Farmer, Eitan Grinspun, Peter Schröder, “Sparse matrix solvers on the GPU: conjugate gradients and multigrid,” Jul. 2003, [Online]. Available: https://dl.acm.org/doi/10.1145/882262.882364

[6] Du, Peng; Weber, Rick; Luszczek, Piotr; Tomov, Stanimire; Peterson, Gregory; Dongarra, Jack (2012), “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming,” Aug. 2012, [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0167819111001335?via%3Dihub

[7] Vasyl Skorych,Maksym Dosta, “Parallel CPU–GPU computing technique for discrete element method,” Jan. 24, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.6839

[8] David Andersson,Robert Ranman,Shisheer Shetty,Frowin Winkes, “GPU Accelerated CFD Using CUDA.” 2022.

[9] ROCKY, “More GPUs = faster processing with Rocky DEM 4.5,” Aug. 2021, [Online]. Available: https://rocky.esss.co/blog/multi-gpu/

[10] Høyland, K V; Jenson, A; Liferov, P; Heinonen, J; Evers, K-U; Løset, S; Määttänen, M, Physical Modeling of First-Year Ice Ridges - Part I: Production, Consolidation and Physical Properties. 2001. [Online]. Available: https://trid.trb.org/view/1391804

[11] P. Greisman, “‘Brash Ice Behavior’, United States Coast Guard, Research and Development Center, Groton, Conneticut,” 1981, [Online]. Available: https://www.researchgate.net/publication/349058266_Simulation_of_Brash_Ice_Behavior_in_the_Gulf_of_Bothnia_Using_Smoothed_Particle_Hydrodynamics_Formulation

[12] “Simulation-Based Engineering Lab: High-Performance Computing for Applications in Engineering,.” 2011. [Online]. Available: http://sbel.wisc.edu/Courses/ME964/2011/Lectures/lecture0224.pdf

[13] NVIDIA, “Profiler User’s Guide.” Aug. 03, 2022. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[14] Jonas Behnen, “Numerical Simulation of Ship Performance in Brash Ice - Assessment of the Mechanical Behaviour in the Numerical Model with the Mechanical Behaviour of Brash Ice in Nature.” Oct. 2019.

[15] J. C. Hans-Georg Matuttis, *Understanding the Discrete Element Method: Simulation of non spherical particles for granular and multi body systems*. 2014.

[16] The Hamburg Ship Model Basin, “HSVA Model Ice Preparation and Property Determination.”

[17] HSVA, “Discrete Element Simulation of Ships Navigating Through Brash Ice Channels and Ice Ridges Instruction Manual.”

[18] Vasiola Zhaka, Robert Bridges, Kaj Riska, Andrzej Cwirzen, “A review of level ice and brash ice growth models,” Dec. 2021, [Online]. Available: https://www.researchgate.net/publication/357253691_A_review_of_level_ice_and_brash_ice_growth_models

[19] Noureddine Ait Ali, Soufiane Hamida, Bouchaib Cherradi, Yasser Lamalem, Ahmed El Abbassi, “A Computational Performance Study of Unsupervised Data Clustering Algorithms on GPU.”

[20] Rick van der Zwet, “Exploratory research on embedding CUDA code into hetrogeneous MP-SOC achitectures programmed with the Daedalus framework.” Sep. 2011.

[21] Stackoverflow, “Does __syncthreads() synchronize all threads in the grid?,” Mar. 06, 2013. https://stackoverflow.com/questions/15240432/does-syncthreads-synchronize-all-threads-in-the-grid

[22] MEAP, “Organizing your Fortran code using modules.” [Online]. Available: https://livebook.manning.com/book/modern-fortran/chapter-4/v-13/7

[23] “A Comprehensive Suite of Compilers, Libraries and Tools for HPC.” https://developer.nvidia.com/hpc-sdk

[24] “Is there a free CUDA Fortran compiler for Windows.” https://www.reddit.com/r/CUDA/comments/nw9owb/is_there_a_free_cuda_fortran_compiler_for_windows/

# Appendix

## Appendix 01: Parameters to generate Ice ridge in the in-house software

| | | | |
|---|---|---|---|
| Ridge width | 1.20 m | 1.20 m | 1.20 m |
| Ridge keel width | 0.0 m | 0.0 m | 0.0 m |
| Ridge keel height | 0.3 m | 0.3 m | 0.3 m |
| Ridge length | 2.5 | 1.00 m | 0.5 |
| Void fracture | 55% | 55% | 55% |
| RubbleLength | 0.1 m | 0.1 m | 0.1 m |
| RubbleWidth | 0.1 m | 0.1 m | 0.1 m |
| Rubble thickness | 0.04 m | 0.04 m | 0.04 m |
| Ice density | 867 kg/m³ | 867 kg/m³ | 867 kg/m³ |
| Ice Young's modulus | 1.0 x $10^6$ Pa | 1.0 x $10^6$ Pa | 1.0 x $10^6$ Pa |
| Ice Poisson's ratio | 0.3 | 0.3 | 0.3 |
| Cohesion coefficient | 0.0001 | 0.0001 | 0.0001 |
| Viscous damping coefficient | 2 | 2 | 2 |
| Normal damping force coefficient | 0.2 | 0.2 | 0.2 |
| Tangential dissipation force coefficient | 0.2 | 0.2 | 0.2 |
| Ice friction coefficient | 1 | 1 | 1 |
| *frac* | 0.2 | 0.2 | 0.2 |

## Appendix 02: Parameters to generate Brash ice in the in-house software

| | | | | |
|---|---|---|---|---|
| ChannelWidth | 2.0 m | 2.0 m | 2.0 m | 1.0 m |
| ChannelLength | 0.1 m | 0.5 m | 2.5 m | 0.05 m |
| ChannelPorosity | 0.35 m | 0.35 m | 0.35 m | 0.35 m |
| Brash ice -Thickness | 0.075m | 0.075m | 0.075m | 0.075m |
| Brash ice -DistType | 2 | 2 | 2 | 2 |
| Brash ice -DistParamOne | -4.0942 | -4.0942 | -4.0942 | -4.0942 |
| Brash ice -DistParamTwo | 0.287041 | 0.287041 | 0.287041 | 0.287041 |
| Brash ice -MaxInitSpeed | 0.5 | 0.5 | 0.5 | 0.5 |
| Ice density | 867 kg/m³ | 867 kg/m³ | 867 kg/m³ | 867 kg/m³ |
| Ice Young's modulus | 1.0 x $10^6$ Pa | 1.0 x $10^6$ Pa | 1.0 x $10^6$ Pa | 1.0 x $10^6$ Pa |
| Ice Poisson's ratio | 0.3 | 0.3 | 0.3 | 0.3 |
| Cohesion coefficient | 0.0001 | 0.0001 | 0.0001 | 0.0001 |
| Viscous damping coefficient | 1.0 | 1.0 | 1.0 | 1.0 |
| Normal damping force coefficient | 0.2 | 0.2 | 0.2 | 0.2 |
| Tangential dissipation force coefficient | 0.2 | 0.2 | 0.2 | 0.2 |
| Ice friction coefficient | 1.0 | 1.0 | 1.0 | 1.0 |
| *frac* | 0.2 | 0.2 | 0.2 | 0.2 |

# Appendix 03: Profile visualization (Brash Ice)

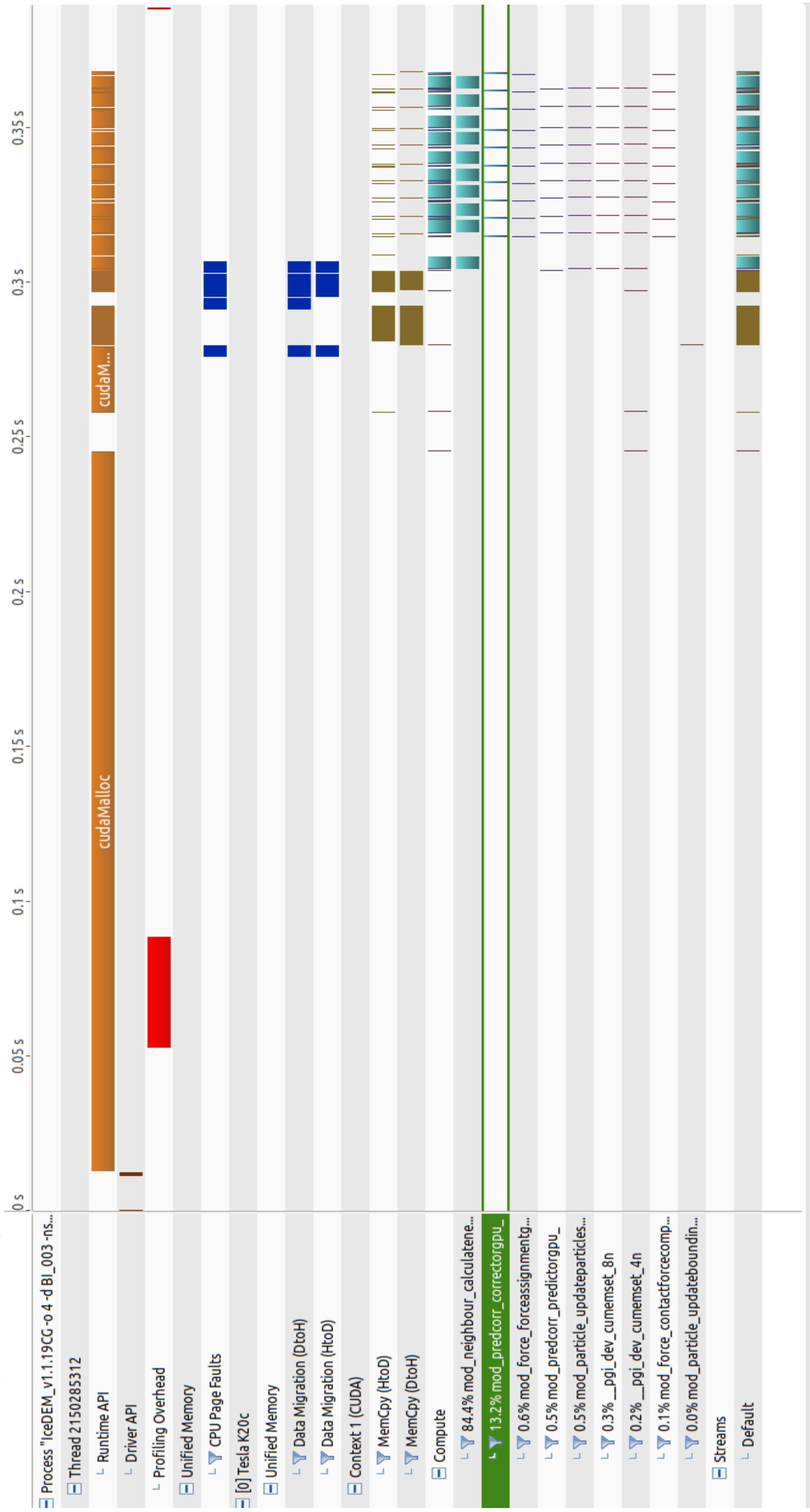


Figure 42 Number of element 44, Number of block 16

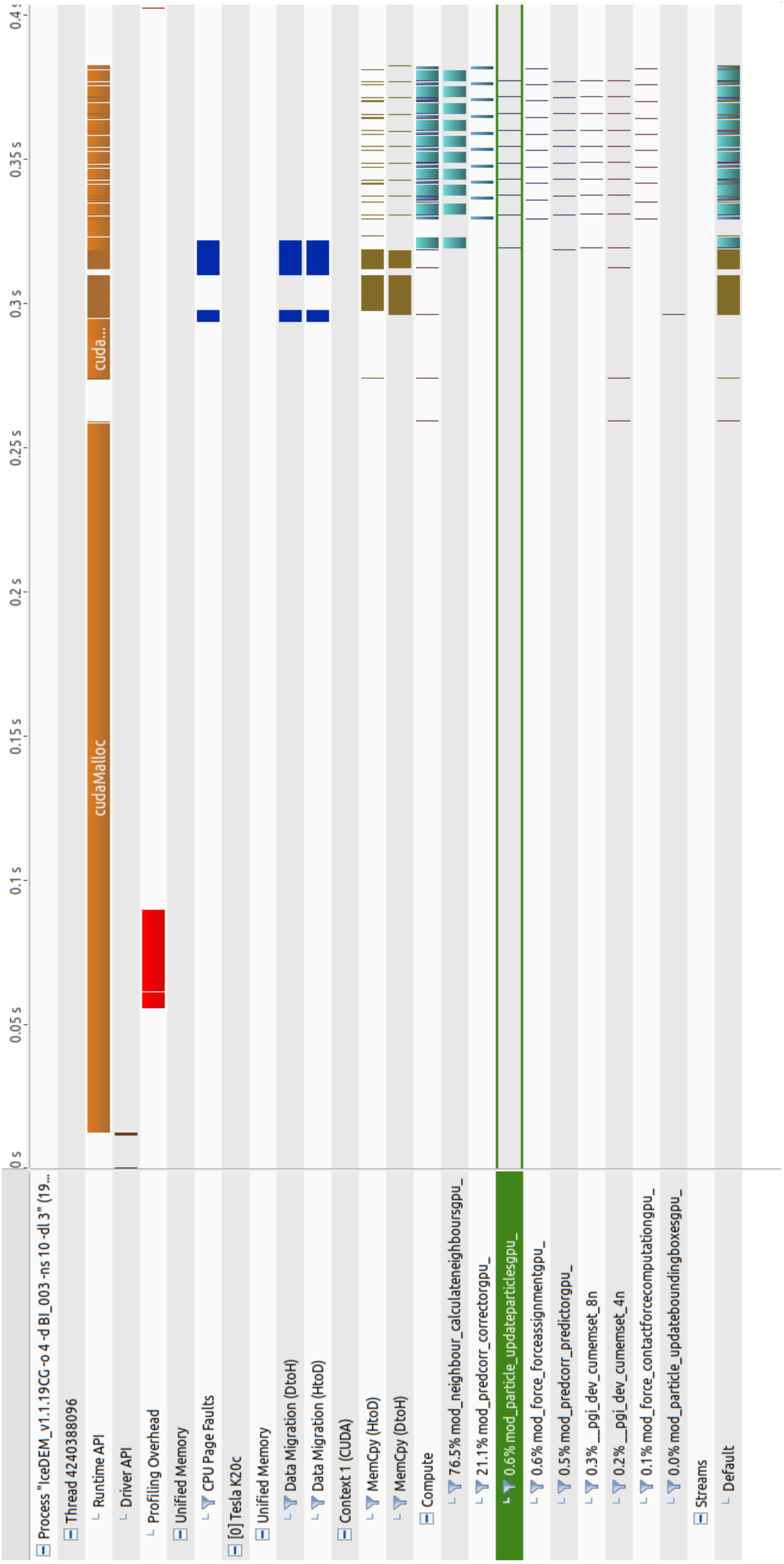


**Figure 43 Number of element 44, Number of block 128**

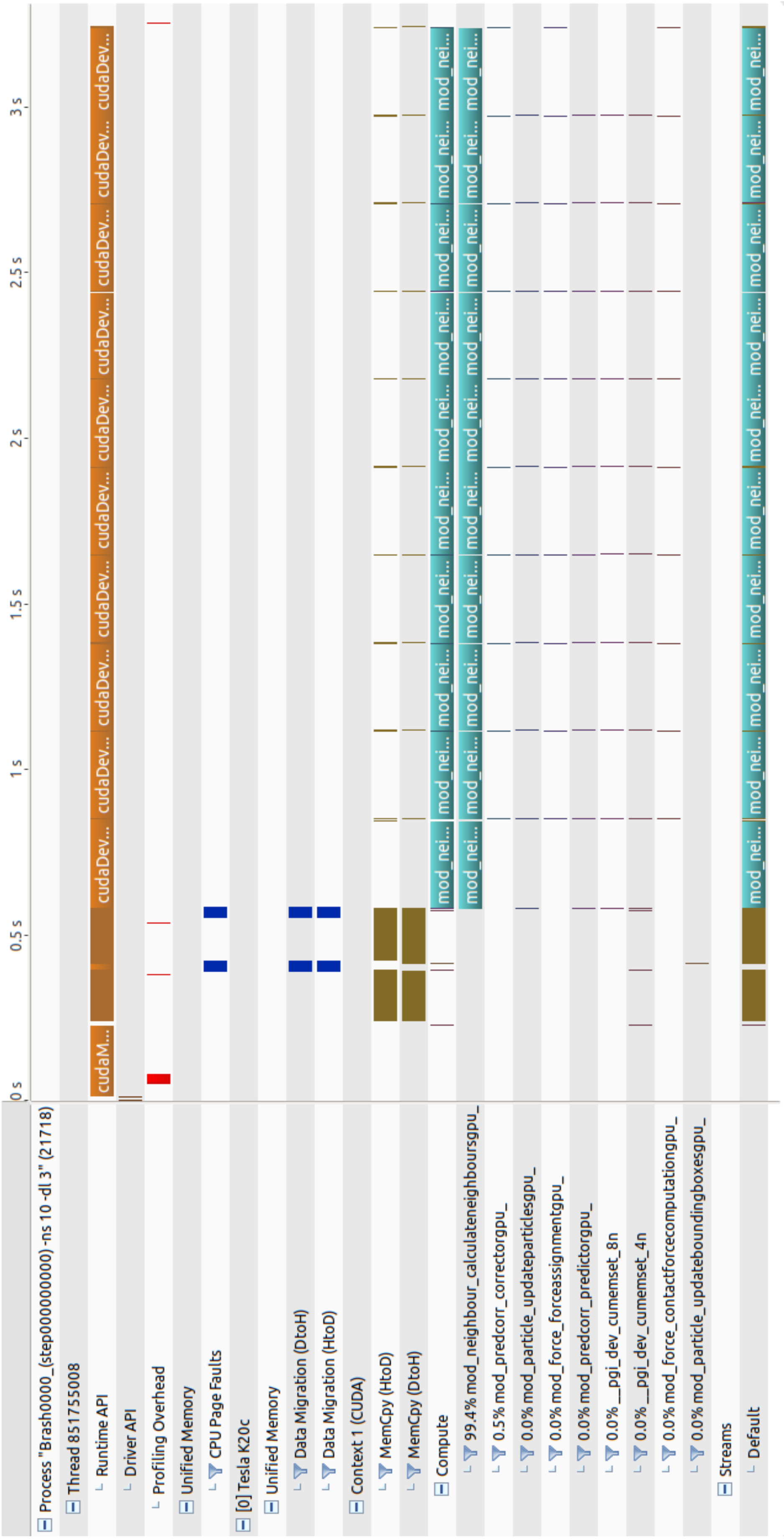


**Figure 44 Number of element 351, Number of block 128**

## Appendix 04: Profile visualization (Ice Ridge)

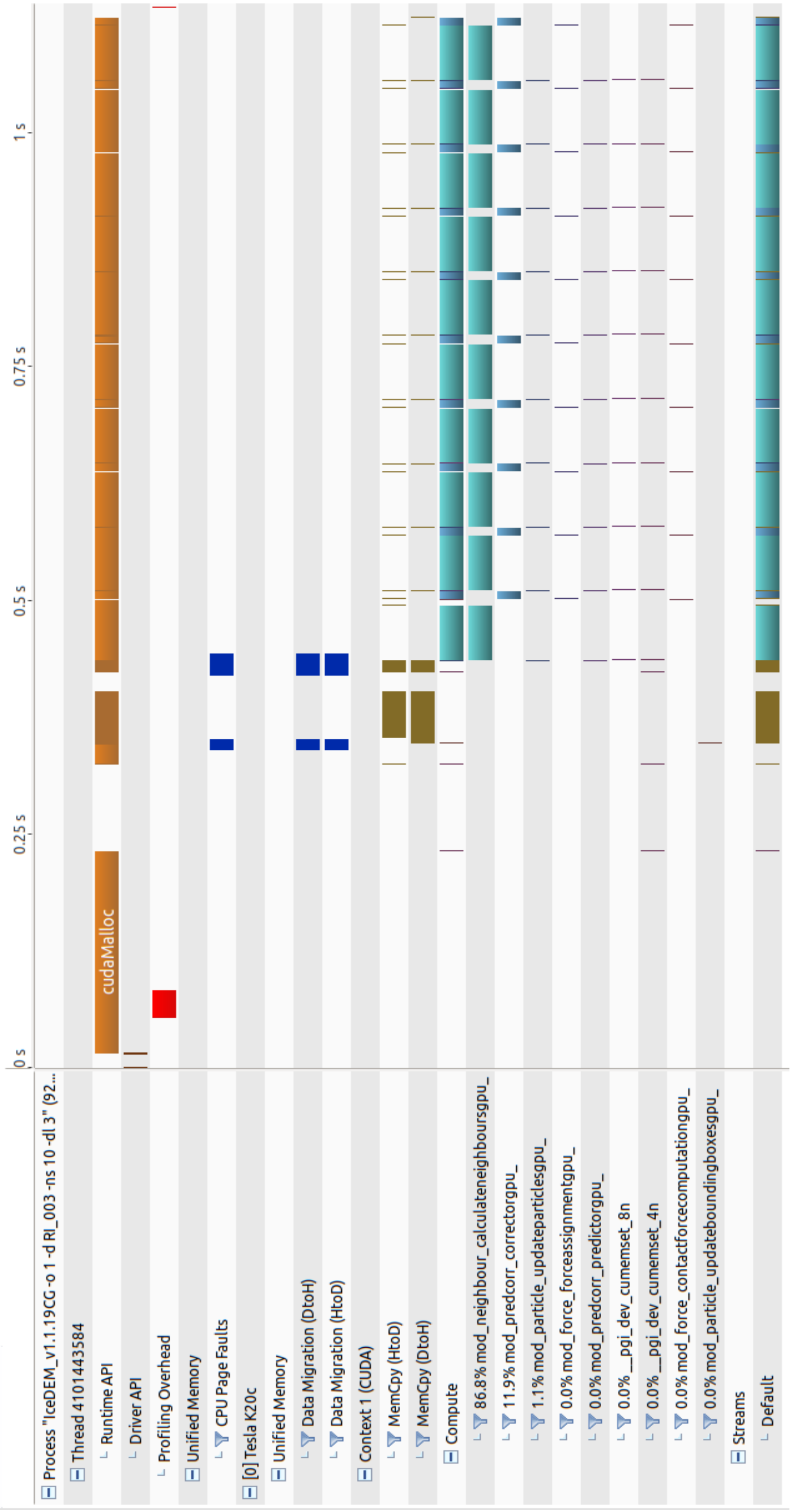
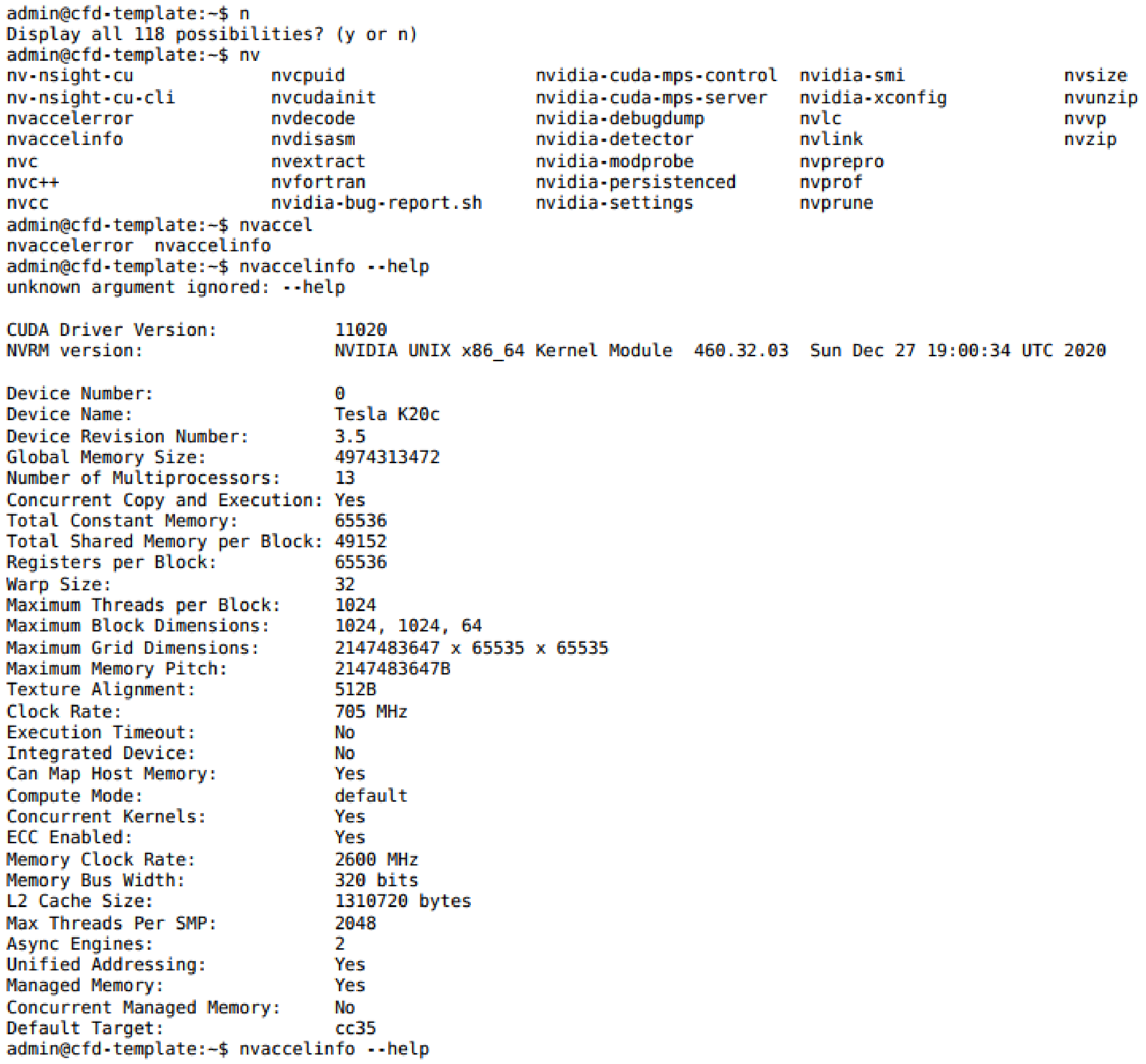


Figure 45 Number of element 359, Number of block 128

# Appendix 05: Computer Specification



```
admin@cfd-template:~$ n
Display all 118 possibilities? (y or n)
admin@cfd-template:~$ nv
nv-nsight-cu             nvcpuid                  nvidia-cuda-mps-control  nvidia-smi               nvsize
nv-nsight-cu-cli         nvcudainit               nvidia-cuda-mps-server   nvidia-xconfig           nvunzip
nvaccelerror             nvdecode                 nvidia-debugdump         nvlc                     nvvp
nvaccelinfo              nvdisasm                 nvidia-detector          nvlink                   nvzip
nvc                      nvextract                nvidia-modprobe          nvprepro
nvc++                    nvfortran                nvidia-persistenced      nvprof
nvcc                     nvidia-bug-report.sh     nvidia-settings          nvprune
admin@cfd-template:~$ nvaccel
nvaccelerror  nvaccelinfo
admin@cfd-template:~$ nvaccelinfo --help
unknown argument ignored: --help

CUDA Driver Version:           11020
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  460.32.03  Sun Dec 27 19:00:34 UTC 2020

Device Number:                 0
Device Name:                   Tesla K20c
Device Revision Number:        3.5
Global Memory Size:            4974313472
Number of Multiprocessors:     13
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    705 MHz
Execution Timeout:             No
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            Yes
ECC Enabled:                   Yes
Memory Clock Rate:             2600 MHz
Memory Bus Width:              320 bits
L2 Cache Size:                 1310720 bytes
Max Threads Per SMP:           2048
Async Engines:                 2
Unified Addressing:            Yes
Managed Memory:                Yes
Concurrent Managed Memory:     No
Default Target:                cc35
admin@cfd-template:~$ nvaccelinfo --help
```