UiO : Department of Informatics
University of Oslo

Accelerating Numerical Simulations on Multiple GPUs with Multiple CUDA Streams

Applied on a Sediment-Transport Model for Dual Lithologies

Heidi-Christin Bernhoff-Jacobsen

Master's Thesis Spring 2015

Accelerating Numerical Simulations on Multiple GPUs with Multiple CUDA Streams

Heidi-Christin Bernhoff-Jacobsen

Spring 2015

Contents

List of Figures

List of Tables

List of Acronyms

CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General-purpose computing on Graphics Processing Units
HPC	High Performance Computing
PDE	Partial Differential Equation
PDE’s	Partial Differential Equation’s
CFD	Computational Fluid Dynamics
FDM	Finite Difference Method
f FEM	Finite Element Method
FVM	Finite Volume Method
FTCS	Forward Time Centered Space
DDT	Domain Decomposition Methods
BVP	Boundary Value Problem
BC	Boundary Condition
CUDA	Compute Unified Device Architecture
SM	Stream Multiprocessor
SM’s	Stream Multiprocessor’s
SP	Stream Processor
SP’s	Stream Processor’s
IPCs	Interprocessor Communications
PCB	Printed Circuit Board
I/O	Input/Output
UVA	Unified Virtual Addressing
PCIe	Peripheral Component Interconnect Express (PCI Express)
OpenMP	Open Multi Processing
MPI	Message Passing Interface
SISD	Single Instruction, Single Data stream
SIMD	Single Instruction, Multiple Data stream
MIMD	Multiple Instruction, Multiple Data streams
MISD	Multiple Instruction, Single Data streams
SIMT	Single Instruction, Multiple Threads

Chapter 1

Introduction

Improving the overall computational time is one of the challenges in scientific computing today. Performing simulations and visualizations of large data have been very slow or impossible to do on desktop computers or laptops. This is due to the limited processing capacity of these computers' Central Processing Unit (CPU). This thesis will investigate the field of High Performance Computing (HPC) and the possibility of running the simulations of a sediment transport model in parallel on the many cores of the Graphics Processing Unit (GPU), and observe if these computations will perform faster, and yield results with the same accuracy as a CPU. It will also look at the possibility of coupling multiple GPUs, and observe if that will give an additional speedup.

1.1 Background

In the field of scientific computing/computational science, mathematical models and quantitative analysis techniques are used to solve a big specter of scientific problems. Computers are used to perform simulations of the modeled equations and other forms of computations. With the computer simulation we get a solution that is a numerical approximations to the problem we are aiming to solve. Numerical simulations can be performed in many various fields and with different objectives, such as reconstructing and understanding events like earthquakes and tsunamis, or predicting the future like a weather forecast or predicting unobserved events such as where to find oil. Other fields where this is used include medical applications, various physical phenomenon and rocket science.

Computer programs have traditionally been written for serial computations. Problems have been broken down to a set of instructions that have been executed sequentially, one after another, on a single core on the Central Processing Unit

(CPU). The CPUs have over the last years moved away from having one core to having multiple cores that can run in parallel. In 2006 Intel released a dual-core processor called Intel Core 2 Duo, and CPUs with 4, 8, 16 and more processors has been produced since then. The mainstream desktop computers and laptops sold today have multi-core processors, a single CPU with two or more processing units called cores, that can run multiple instructions simultaneously. To perform a simulation with billions of computations on a CPU with e.g. 8 cores running in parallel will still take a very long time, and we see that for such cases the multi-core CPUs still have their limitations. Coupling multiple CPUs into a cluster has been done to increase processing speed, but the purchase of all the units for such a cluster is often rather expensive. Reducing cost is one of the motifs why this thesis investigates the use of GPUs. Another motif is reducing computational time by running simulations on parallel computers.

For certain problems in the field of scientific computing, the graphics card, has been used to do the computations. The GPU is a circuit board originally designed to rapidly alternate and render graphics on the computer screen. The possibility to use the GPU to other tasks than graphics rendering has made scientists explore the GPU's architecture through General-Purpose computing on Graphics Processing Units (GPGPU).
Accelerators such as GPUs are used to speed up computationally-intensive tasks. These are also implemented in supercomputers around the world used by e.g. researchers.

Compute Unified Device Architecture (CUDA) [*NVIDIA CUDA Toolkit Documentation, v7.0*] is a parallel computing platform and programming model. It allows software developers to access the power of GPUs through the CUDA-accelerated libraries end extensions to standard programming languages such as C, C++, Fortran etc.

The CUDA architecture available on NVIDIA graphics cards will be tested in this thesis, allowing the simulations of ["Application of a dual-lithology, depth-dependent diffusion equation in stratigraphic simulation"] sediment transport model to run in parallel on a single GPU. Coupling of multiple GPUs will also be tested. Both the single- and multi-GPU will be tried with synchronous and asynchronous simulations. The asynchronous version will perform the communication between the GPU and CPU, or between the multiple GPU's simultaneously with the computations, and by that aim to avoid the overhead due to communications (between the nodes). Here the communication between the GPUs will be tested using the Peripheral Component Interconnect Express (PCI Express, also abbreviated PCIe). Computer busses move information between the different parts of the hardware in a computer system including the CPU, RAM and peripheral devices. The PCIe is an expansion bus that in this case moves information between the nodes of an GPU. In addition to running parallel code on multiple GPUs, multiple OpenMP threads will will be spawn on the CPU to reduce kernel launch overhead

The geological phenomenon described in the equation set derived by [*Rivenæs* 1993] is one of many in the field of geology and geophysics. On the path to the numerical solution to the mathematical sediment transport model, the equation set is discretized using a Finite Difference method according the Forward-Time Central-Space (FTCS) scheme.
The software is then implemented for GPU's produced by Nvidia in CUDA C and additional OpenMP threads, but also in plain C for CPU(s). The CPU implementation is used for comparison of speedup.

1.2 Motivation

The major motivation behind this work is to reduce computational time. The results of large systems can take hours, days or moths to compute. For both research and the industry reducing the overall computing time means money saved. For business it may result in getting a product out on the market sooner, and in research it means moving forward towards new findings. Getting results faster is an important goal for everyone.

1.3 Document Structure/Chapter Overview

Chapter 1: Introduction presents the problem and why the filed is interesting to investigate.
Chapter 2: Thesis Domain describes the problem to be investigated more thoroughly, and give a more precise description of the problem to be analized.
Chapter 3: Describes the different methodologies.
Chapter 4: Presents the mathematical model, discretizing schema and boundary- and initial conditions.
Chapter 5: Gives an overview of High Performance Computing.
Chapter 6: Presents Parallel Architectures, Flynn's taxonomy and Foster's methodology.
Chapter 7: Presents the features in CUDA C applied in this thesis.
Chapter 8: Presents details in the various implementations of the sediment transport.
Chapter 9: Presents a performance analysis.

Chapter 10: Results and Conclusion, contribution and future work

Chapter 2

Thesis Domain

2.1 Sediment Transport Model

The mathematical model investigated in this thesis is a sediment transport model evolved/described by [“Application of a dual-lithology, depth-dependent diffusion equation in stratigraphic simulation”]. The model describes an application of a two-dimensional (2D) basin simulation. It is based on topographical diffusion of a dual-lithology mass, as it runs down a dynamic slope and into a basin. The two materials are classified as sand and mud. Each lithology has its own transport-coefficient function, and the effect of compaction is included in the mass balance, as well as depth-dependent transport coefficients (diffusivities).

This physical phenomenon is described by a set of Partial Differential Equations (PDEs). The equation set is a coupled system consisting of a diffusion part and a convection part.

Diffusion describes/is the movement/motion of (mud/silt) molecules as it flows from one region of high concentration to a a region of low concentration along a concentration gradient. Diffusion can also relate to other matters as e.g. pressure along a pressure gradient or temperature along a temperature gradient. The word diffusion comes from the latin word "diffundere", and means "to spread out". We here use the word in the context of mass transportation.

Convection is another transport phenomenon in the field of physics and fluid mechanics. It describes the collective movement of groups/aggregates of molecules within fluids. It describes how the masses (of sand) moves in a circulatory motion as the variation in density (of the mud/silt) changes.

A broader elaboration of these equations will be handled in chapter 4 "The Mathematical Model". A thorough presentation is given on how to get from an equation set representing a physical model to the final computational model that is used for implementations of the system.

To be able to do the computations/simulations on a computer, the area of interest, also called domain, is separated into a grid consisting of discrete grid points. Calculations on our equations are performed for each grid-point. The equation-system consists of various variables that may hold different values on the different grid-points.

Forward-Time Central-Space (FTCS) scheme is the **Finite Difference Method** used to discretize the equations. In the **numerical analysis** the path to an algorithm representing the numerical approximation of the analog equations is displayed.
Other extensive details regarding the analysis of **Computational Fluid Dynamics (CFD)** are presented in the same chapter.

2.2 Area of Interest Used for Testing

The area of interest that will be focused on when testing the numerical algorithms is situated in Florida. More specificly **Lake Okeechobee** and the **Kissimmee River**. The lake is remarkably shallow with a maximum depth of only 4 m. and an average depth of 2.7 m. The Kissimmee River is the single contributor of sediments to the lake, counting for 30% of the (water-)volume added to the lake. High resolution data for the lake's bathymetry are derived from depth sounding, and they are publicly available. These data will be utilized in the simulated model.

2.3 The Computer System

The application is based on the numerical model of a 2D system with time series. In the process of implementing a computer system for the sediment-transport model, there are several considerations to be taken. The simulations of this model are to be run for large data-sets and it will be computing-heavy. If the system is to be computed on a CPU of any desktop computer, even the more powerful ones, it will most probably take terribly long time. The widely used term **Big Data** can be be used when simulating a numerical model as the sediment-transport model and with data-sets as large as the ones for Lake Okeechobee, since this yields a system so large and complex that traditional data processing applications are inadequate. For that reason this thesis will investigate the field of **High Performance Computing (HPC)**, which is the practice of aggregating computing power that delivers a much higher performance than than what is possible on a single CPU.

This system consists of a set of numerical PDE's to be computed for a large set for grid points. At each grid point computations on the same equations will be performed, but the values of the equation's variables may differ from one grid point to the next. In a multiprogramming context, this system classifies under the **Singel Instruction, Multiple Data streams (SIMD)** paradigm. This means that the system may benefit from an implementation made for a parallel computer architecture. In a parallel context this thesis will investigate the use of hardware accelerators such as the GPUs produced by NVIDIA, a bleeding edge technology.

In the process of the thesis work there will be two main implementations. The first is a serial application made for computations on a CPU, the next is a parallel implementation made to run on a single GPU. The development of the parallel code normally demands extended programming skills that can take time to master and take in use. On the other hand, if the parallel system runs much faster than the serial system then there are tremendous benefits. A further extension is to evolve the parallel system to run on multiple GPU's of an arbitrary number. One focus will then be on GPU's sharing the same PCI Express bus for data transfer.

The programing languages that will be used are C for the serial system to run on a CPU, and C with the CUDA extensions/bindings called CUDA C, used for the parallel GPU-version. Multiple technologies such as CUDA C combined withe Open MP threads will be tested for the implementation of the multi-GPU application.

Further details on HPC will be surveyed in the Chapter 5.
Details regarding CUDA C programming will be targeted in the Chapter 7.

2.4 Speedup

Measurements of performance will be made for the different systems, and they will be compared in various ways. The performance enhancement can be measured using speedup as a quantifying metric for relative performance improvement. More on this topic will be surveyed in the Chapter 9: Performance Analysis.

In this research the aim is to see if there are any improvements in computational time comparing a parallel application running on a single GPU to a serial application running on a CPU. An additional focus is to see if there is even more time to save if running the system on multiple GPU's.

Another research aspect to focus on is if the parallel computations yields results with the same accuracy as the serial computations.

If the parallel system achieves a good speedup, it will open up for the possibility to run larger simulations. One possibility could be to do data-simulations over a larger grid representing a larger interest domain or it could be a more refined grid with a higher number of grid points within the same domain size, but yielding a more detailed result. Other possibilities could be the choice of extending the time series to include more time steps, or even a combination of the different options that opens up if good speedup is achieved.

So, will the parallel code run faster and yield results in shorter time than the serial code, and will the parallel code yield results with the same accuracy as the CPU?
The next section will state the questions this thesis will aim to answer.

2.5 Thesis Questions and Problems to be Solved

One of the areas this thesis will investigate is wether there are any benefits in implementing a parallel system. Then another question is; if there are time to save when running the application on one GPU, will there be more time to save when run on multiple GPUs?

Performing simulations and visualizations of large data and complex systems has been very slow or impossible to compute due to the lack of computing power of the CPU. The evolution in the GPU production has been amazing over the last years, towards a circuit board adapted for heavy computations and other tasks than graphics rendering. With the use of GPGPU, a new world has opened up to computer programmers.
Here is a structured presentation of the questions this thesis will aim to answer and the problems this thesis will aim to solve.

Q 1:	Is a parallel application running on a single GPU faster than a serial application running on a CPU? And if so:
Q 2:	Is a parallel application running on multiple GPUs faster than a parallel application running on a single GPU?
Q 3:	Is the application enhancing performance utterly if additional CUDA Streams are utilized?

If the the GPU-implementations yields the results faster, then the further questions will be:

Q 3:	Does the parallel application running on a single GPU yield results with the same accuracy as the serial application running on a CPU?
Q 4:	Does the parallel application running on multiple GPU's yield results with the same accuracy as the application running on a single GPU?

Table 2.1: Questions This Thesis Aims To Answer

2.6 Contribution of This Work

This thesis has reached the goal the of answering the Thesis Questions in table 2.1. The goal was to enhance speedup for sediment-transport simulations, with the use of GPUs. Another goal was to achieve solutions with the same accuracy as the solution from a CPU-simulation. Through this investigation, the enhanced speedup has been more than satisfactory for simulations on both a a single GPU as well as on multiple GPUs. The numerical results yield the same accuracy on GPUs as on CPU. This clearly show the benefits of using GPUs for sediment-transport simulations.

Chapter 3

Methodology

3.1 Paradigms for The Discipline of Computing

This thesis investigates a technology-oriented discipline with fundamentals in mathematics and engineering. The discipline of computing embraces all of computer science and computer engineering. The two fields do not differ in the core material. [*Comer* et al. 1989] suggests paradigms for the discipline of computer science, and emphasizes:

1. **Theory** - based in the mathematical model.

2. **Abstraction** - based in experimental scientific method.

3. **Design** - based in engineering, involves designing and testing a system.

With their view on methodologies for the field, this chapter presents methods utilized on the path to a solution to the problem aimed to be solved in this thesis.

The main problem to be tested is if a computer-system can enhance performance if the simulation is executed on a parallel architecture like the SIMD architecture on a GPU, compared to execution on a serial CPU.

For this purpose the chosen mathematical model is the sediment-transport model by [*Rivenæs* 1993]. This model consists of a coupled system of Partial Differential Equations (PDE's). To be able to compute the set of analog PDE's, the equations has to be discretized and for that a numerical method is needed.

3.2 The Numerical Method For Partial Differential Equations

The approach to find numerical solutions to PDE's has led researchers into the field of numerical analysis, and numerous well established numerical methods has been evolved to help in this matter. Some of these method are, the Finite Difference Method (FDM), Finite Element Method (FEM) and Finite Volume Method (FVM).

On the sediment transport model the **Finite Difference Method** is applied. With this method, functions are represented by their values at discrete points in a grid. Through the differences in these values, derivatives are approximated. From the mathematical model the numerical model is derived using an **Explicit Method**. This is a discretizing method applying a forward difference in time, and central difference in space, called **Forward-Time Central-Space (FTCS)**. It is a first-order method in time and a second-order method in space. For the sediment transport model there are 2 dimensions in space, so second-order derivatives are calculated in both x- and y-direction. The calculations made for time-step u^l uses the values at calculated at timestep u^{l-1}.

3.2.1 Stability

To maintain numerical stability for 1D problems with the FTCS method, the following condition has to satisfied.
$r = \frac{\alpha \delta t}{\delta x^2} \leq \frac{1}{2}$

This means that the size of the time step, δt is restricted by this stability condition. For problems with large diffusivity the δt has to be very small. A small δt can be a major disadvantage of this method, as it will have to perform many time-iterations to simulate a given time period.

For systems with many variables and multiple large data-sets as the sediment-transport system, the simulations can take a long time. Therefore numerous optimization techniques and methods will be tested and investigated in this thesis.

3.3 The Variables

The variables used in this investigation are given by the mathematical model by [*Rivenæs* 1993] as seen in the chapter 4. It is a massive equation system with a myriads of variables that are operated on as the numerical model is simulated through the vey many time steps. Some of these variables holds scalars or constants, while other variables are large data-sets extracted from survey of seismic activity. The data-sets used in the computational domain is gridded for the spatial discretization.

3.4 The Data

The data that will be used in the sediment-transport simulations are extracted from Lake Okeechobee in Florida. The lake is positioned in a shallow geological trough and is of the size 1.900 square kilometers (730 sq. mi). It is exceptionally shallow with an average depth of 3 meters (9 feet). The Kissimmee River gives the influx of sand and mud to the lake.
The massive data-sets used in these computations are the results of seismic measurements of Lake Okeechobee. These data are publicly available [*Gtopo30* 1996].

3.5 Domain Decomposition Method

In the field of numerical analysis and numerical PDE's, simulations are mostly performed over massive data-domains. Computations performed on very large data-set(s) takes time. If they are executed sequentially it takes much longer time then if they are executed in parallel on a single GPU, and even more time can be saved if run on multiple GPU's compared to a single GPU. With the purpose of the enhancement to yield a faster result, a method to run the simulations on multiple GPU's is applied. Solving boundary value problems on multiple GPU's can be done by splitting the data-domain into smaller subdomains, where computations for each subdomain are computed on one of the dedicated GPU's in the multi-GPU system. The Domain Decomposition Method (DDM) is a method that splits a data-domain into independent, adjacent subdomains. The method is ideal for parallel computing on a **Boundary Value Problems (BVP)**.

A 1D problem can only be subdivided in a 1D decomposition.
Let the intervals over the linear domain M be denoted $[m_{start}, m_{end}]$.
For a 1D domain on the interval $[0,1]$, if split into two equally sized subdomains, one is on the interval $[0, \frac{1}{2}]$ and the other is on the interval $[\frac{1}{2}, 1]$.

A 2D-problem is exposed to a variation of decomposition possibilities. The domain can be decomposed in 1D or 2D manners.
A 2D decomposition of a 2D domain is decomposed both horizontally or vertically.

Let the intervals over a MxN domain be denoted $[m_{start}, m_{end}]x[n_{start}, n_{end}]$.
With a $2x2$ decomposition on the interval $[0,1]x[0,1]$, the different subdomains are, when equally sized, on the intervals:
$[0, \frac{1}{2}]x[0, \frac{1}{2}]$,
$[\frac{1}{2}, 1]x[0, \frac{1}{2}]$,
$[0, \frac{1}{2}]x[\frac{1}{2}, 1]$ and
$[\frac{1}{2}, 1]x[\frac{1}{2}, 1]$

A 1D decomposition on a 2D domain can be decomposed horizontally or vertically.
A 1D decomposition into 4 subdomains will with a $1x4$ decomposition on the interval $[0,1]x[0,1]$ give subdomains on these intervals:
$[0,1]x[0, \frac{1}{4}]$,
$[0,1]x[\frac{1}{4}, \frac{1}{2}]$,
$[0,1]x[\frac{1}{2}, \frac{3}{4}]$ and
$[0,1]x[\frac{3}{4}, 1]$.
If decomposed in the other direction; $4x1$ on the same interval, it yields subdomains on these intervals:
$[0, \frac{1}{4}]x[0,1]$,
$[\frac{1}{4}, \frac{1}{2}]x[0,1]$,
$[\frac{1}{2}, \frac{3}{4}]x[0,1]$ and
$[\frac{3}{4}, 1]x[0,1]$.

A 2D decomposition yields a larger number of neighbors that boundary values has to be communicated to than a 1D decomposition. Due to communication-overhead a 2D decomposition is not necessarily a better choice than a 1D decomposition.

3.6 Empirical Research Method

To evaluate the results in this research, empirical research method is used. It is a way to acquire knowledge through experimenting and testing, and it may also be achieved through direct or indirect observations or experience. Empirical evidence is the record of a someone's direct observations or experience and can be analyzed **quantitatively** or/and **qualitatively**.

Implementing, testing and comparing the applications will make up for a large part of the empirical studies in this work. There will of course be iterations of application development with the intent of making improvements.

Based on the problem domain and the questions this thesis aims to solve, both a qualitative and a quantitative analysis will be made.

3.6.1 Quantitative Analysis

For the quantitative analysis of the results, the execution time of the various systems are measured. This time is given by a very direct observation of benchmarking the actual speed of the executed system. Execution time for all the different systems can be compared, and by the comparison it is possible answer some of this thesis' questions.
Through the quantitative analysis of these result it is possible to decide if a parallel implementation is faster than a serial implementation, and if a parallel implementation of multiple GPU's is faster than an implementation of one GPU. It is also possible to decide if an implementation with OpenMP threads yields a faster solution.
Performance and **Speedup** are two "methods" to analyze the measurements and enhancement between to different systems. Both methods are in this way quantitative as both methods derive results form quantitative measurements. More about the these two methods is reviewed in the chapter on Performance Analysis 9.

3.6.2 Qualitative Analysis

For a **qualitative** analysis of the results, this thesis examines the numerical results of the simulations, and makes comparisons between the serial implementation and the various parallel versions. If the different systems parallel systems, through execution, render the same correct numerical result as the serial system, it can be said that the parallel implementations produce a result of equal quality as the serial.

Although speedup and performance are analyzed qualitatively, it would be possible to say that a system that gives a noticeable speedup and improved performance, in addition to a correct numerical result of course, adds to the total quality of the application.

3.7 Enhancement Methods And Techniques

Computer programs have been implemented with the aim and ambition to solve the problems this thesis states. The code has been produced with a variety of features, exploiting the possibilities that lies within the chosen technologies. Extensive empirical testing has been conducted through running the different programs evolved with the different programming details made for optimization. Along with the use of the well-established methods presented here, the research is accomplished with a great specter of enhancing programming techniques and methods. A broader explanation on how these methods and the large variety of enhancement techniques are used, is presented in several of the chapters throughout this thesis.

Chapter 4

The Mathematical Model

4.1 Numerical Analysis of a Dual-Sediment Transport Model

There are several numerical strategies for solving this system, and Clark et al. [*Clark*, *Wei*, and *Cai* 2010] has studied two of the possibilities for solving a coupled system of distinct nonlinear equations governing sediment transport of Lake Okeechobee, Florida. They used high-resolution bathymetry data of the actual lake derived from depth-sounding [*Gtopo30* 1996]. The computations were performed on a multicore-based cluster.

This thesis will investigate the numerical strategy of a fully-explicit numerical scheme, and the computations will be tested on the CUDA architecture on NVIDIA GPUs.

Erosion and deposition of sediment is a process often modeled using a diffusion equation. The speed of diffusion represents the transport efficiency of the sediment. When multiple sediments are involved in the model, one can add another equation for each sediment. Various physical processes like sediment compaction, tectonic movements and carbonate production can be governed by a numerous empirical rules, but has been neglected by Clark et al. in order to investigate the parallel performance of their diffusion equation alone.

To model sediment transport in fluvial, on- and off-shore environments, diffusion has been used. Jordan and Flemings [*Jordan* and *Flemings* 1991] suggest a model for sediment transport, approximating a slope-controlled diffusion, and their work gives the basic equation:

$$\frac{\partial h}{\partial t} = \nabla . \left(\kappa \nabla h\right) \tag{4.1}$$

where
$h = h(x,y,t)$ is the height of the basin in the x,y-plane at a given time(step),
t is the time and
$\kappa = \kappa(x,y,t)$ is the transport coefficient or the efficiency of the diffusion.
This equation (4.1) was later modified by Rivenæs [*Rivenæs* 1993], so two types of sediments could be handled in the model, with one transport coefficient for each sediment.

The variables used for mathematical system are:

Variable Name	**Explanation**
h	Bathymetry (Height of the basin)
s	Fraction of sand
$(1-s)$	Fraction of mud/silt
α	Transport coefficient for sand
β	Transport coefficient for mud/silt
A	Layer thickness
C_s	Compaction ratio for sand
C_m	Compaction ratio for mud
f_s	Influx sand
f_m	influx mud
dx	Size of spatial step in x-direction
dy	Size of spatial step in y-direction
dt	Size of time step
l	Temporal discretization step
i	Spatial discretization step in x-direction
j	Spatial discretization step in y-direction

Table 4.1: Varibles for the Mathematical Model

$$\frac{\partial h}{\partial t} = \nabla \cdot (\alpha s \nabla h) + \nabla \cdot (\beta(1-s)\nabla h) \tag{4.2}$$

and

$$A\frac{\partial s}{\partial t} + s\frac{\partial h}{\partial t} = \nabla \cdot (\alpha s \nabla h). \tag{4.3}$$

Here,
$\alpha = \alpha(x,y)$ and $\beta = \beta(x,y)$ are the two diffusion coefficients for sand and silt/-mud, and
$s = s(x,y,t)$ and $(1 - s(x,y,t))$ are the corresponding fractions of the two sediment

types at a particular location. For this case s represents the first material, sand, and $(1-s)$ represents the second; material, silt. Further
A is the layer thickness, representing the height of unsettled sediments in the basin, and in this case it's set to 1 m. Also, A scales the partial derivatives of s with respect to distance and determines how effective changes in h will be affecting the sediment concentration.

When taking into consideration the compaction ratio for sand and silt, the equations 4.2 and 4.3 are represented by:

$$\frac{\partial h}{\partial t} = \frac{1}{C_s}\nabla \cdot (\alpha s \nabla h) + \frac{1}{C_m}\nabla \cdot (\beta(1-s)\nabla h) \tag{4.4}$$

and

$$A\frac{\partial s}{\partial t} + s\frac{\partial h}{\partial t} = \frac{1}{C_s}\nabla \cdot (\alpha s \nabla h). \tag{4.5}$$

where
C_s is the compaction ratio for sand and
C_m is the compaction ratio for mud.

Water discharge rates, water-flow induced shear-stress and drag-coefficient can be neglected due to the separation of the two material types rather than having one diffusion coefficient based on grain-size.

For simplicity the values are not depth dependent, since Clark et al. [*Clark*, *Wei*, and *Cai* 2010] assume that sediment diffusion is not as vigorous in the lake as for the seashore. Therefor they utilize lower values than those suggested by the lakes mean depth of 2.7 m. In this case, they use the coefficients for sand and silt, and the table below shows the different regions with the corresponding transport coefficients in square meters per year (m^2/yr).

Region	Sand (α)	Silt (β)
Kissimmee River	70.000	100.000
Lake Okeechobee	2.100	3.000
Surronds	70	100

Table 4.2: Transport Coefficients for Sand and Mud

4.2 Numerical Scheme

The problem to be solved is a typical initial value problem. Clark et al. [*Clark*, *Wei*, and *Cai* 2010] solve the initial value problem by time integration, and at time step l the latest numerical solutions of h and s, denoted h^{l+1} and s^{l+1}. To have the equations system solved by a computer, the continuous equations need to be transferred into discrete counterparts by using techniques for numerical discretization. Solving the analogue equations will yield a correct result, while solving the discretized scheme will give an approximation to the solution. Various numerical discretization strategies can be chosen, but this thesis will look exclusively at the **Fully-explicit scheme**.

4.2.1 Initial Condition (IC)

At the start of our simulations, at time step 0, we set the initial condition to be:

$$h(x,y,0) = h^0(x,y)$$

$$s(x,y,0) = s^0(x,y)$$

These initial values for h and s that are used at the start of the simulations at time step 0, come from seismic measurements of Lake Okeechobee in Florida [*Gtopo30* 1996].

4.2.2 Temporal Discretization

The time domain $0 < t \leq T$ is divided into a number of equal-distanced discrete time levels with Δt set as the step size. The time level index is noted with l and we use superscript ^{l} such that h^l denotes $h(x,y,l\Delta t)$ and s^l denotes $s(x,y,l\Delta t)$. The temporal derivatives are then approximated as:

$$\frac{\partial h}{\partial t} \approx \frac{h^{l+1} - h^l}{\Delta t}, \qquad \frac{\partial s}{\partial t} \approx \frac{s^{l+1} - s^l}{\Delta t}$$

4.2.3 Fully-Explicit Numerical Scheme

Using a fully explicit discretization scheme, the equations 4.4 and 4.5 are transformed into:

$$\frac{h^{l+1} - h^l}{\Delta t} = \frac{1}{C_s}\nabla \cdot (\alpha s^l \nabla h^l) + \frac{1}{C_m}\nabla \cdot (\beta(1 - s^l)\nabla h^l) \tag{4.6}$$

$$A\frac{s^{l+1}-s^{l}}{\Delta t}+s^{l+1}\frac{h^{l+1}-h^{l}}{\Delta t}=\frac{1}{C_s}\nabla\cdot\left(\alpha s^{l}\nabla h^{l+1}\right) \tag{4.7}$$

It is to be noted that h is must be updated before s during each time-step. This is why the newly computed h^{l+1} from the first equation is immediately used to compute s^{l+1} in the second equation. Another remark is that Wei et al. [*Wei* et al. 2013] suggests s^{l+1} instead of s^{l}, used in the $s\frac{\partial h}{\partial t}$ term on the left side of equation 4.5. They have shown with numerical experiments that this trick improves the numerical stability of this fully-explicit scheme in which both h^{l+1} and s^{l+1} are computed straight forwardly. The values h^{l+1} and s^{l+1} can be computed using simple algebraic operations, and exploit the advantage that no linear systems need to be solved.

The values of α, β and mesh spacing limits the size of Δt in the fully-explicit scheme. The maximum allowed Δt is of order $O\left(\Delta x^2/max\left(\alpha,\beta\right)\right)$

This scheme has 1. order accuracy in time.

4.2.4 Spatial Discretization

The equations 4.4 and 4.5 for respectively diffusion and convection are partially differentiatiated with respect to both x and y as represented by:

$$\begin{aligned}\frac{\partial h}{\partial t}=&\frac{1}{C_s}\left(\frac{\partial}{\partial x}\left(\alpha s\frac{\partial h}{\partial x}\right)\right)+\frac{1}{C_s}\left(\frac{\partial}{\partial y}\left(\alpha s\frac{\partial h}{\partial y}\right)\right)\\&+\frac{1}{C_m}\left(\frac{\partial}{\partial x}\left(\beta\left(1-s\right)\frac{\partial h}{\partial x}\right)\right)+\frac{1}{C_m}\left(\frac{\partial}{\partial y}\left(\beta\left(1-s\right)\frac{\partial h}{\partial y}\right)\right)\end{aligned} \tag{4.8}$$

and

$$\begin{aligned}A\frac{\partial s}{\partial t}+s\frac{\partial h}{\partial t}&=\frac{1}{C_s}\left(\frac{\partial}{\partial x}\left(\alpha s\frac{\partial h}{\partial x}\right)\right)+\frac{1}{C_s}\left(\frac{\partial}{\partial y}\left(\alpha s\frac{\partial h}{\partial y}\right)\right)\\&=\frac{1}{C_s}\left[\left(\frac{\partial}{\partial x}\left(\alpha s\frac{\partial h}{\partial x}\right)\right)+\left(\frac{\partial}{\partial y}\left(\alpha s\frac{\partial h}{\partial y}\right)\right)\right]\end{aligned} \tag{4.9}$$

Motivated by numerical and programming simplicity, [*Wei* et al. 2013] use **Finite Differences** for the spatial discretization.

The 5-point stencil applied on this application is displayed in figure 4.1:

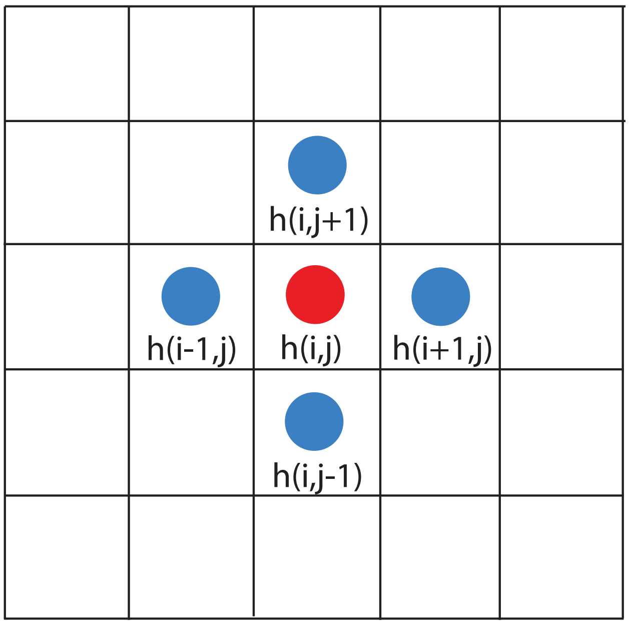

Figure 4.1: 5 Point Stencil

4.2.4.1 Spatial Discretization on Diffusion

Centered Difference in space is a standard way that will be used for the two diffusion terms on the right hand side of 4.4, and this will give a second-order accuracy in space. Applying centered difference to the $\nabla \cdot (\alpha s \nabla h)$ term yields the following discretized form:

$$\nabla \cdot (\alpha s \nabla h) = \frac{\alpha_{(i+1/2,j)} s_{(i+1/2,j)} \left(h_{(i+1,j)} - h_{(i,j)}\right) - \alpha_{(i-1/2,j)} s_{(i-1/2,j)} \left(h_{(i,j)} - h_{(i-1,j)}\right)}{\Delta x^2} + \frac{\alpha_{(i,j+1/2)} s_{(i,j+1/2)} \left(h_{(i,j+1)} - h_{(i,j)}\right) - \alpha_{(i,j-1/2)} s_{(i,j-1/2)} \left(h_{(i,j)} - h_{(i,j-1)}\right)}{\Delta y^2} \tag{4.10}$$

where the subscripts i, j are the indices at a grid point in a 2D uniform mesh with the mesh spacing Δx and Δy . The half-indexed terms in the formula 4.10 are evaluated as:

$$\alpha_{(i+1/2,j)}s_{(i+1/2,j)} = (\alpha_{(i,j)}\ s_{(i,j)} + \alpha_{(i+1,j)}\ s_{(i+1,j)})/2.$$

4.2.4.2 Spatial Discretization on Convection

Equation 4.5 is a convection equation with respect to s, due to the term $\nabla\dot{(}\alpha s\nabla h)$. To obtain numerical stability, one-sided upwind finite difference is preferred over centered difference, although it has a first order accuracy. Therefore, when checking the flow direction, the convection term $\nabla\dot{(}\alpha s\nabla h)$ is moved to the left side of the equation, as customary. By this $-\nabla h$ gives the convection velocity. The approximation of the x-component, $-\partial h/\partial x \approx (h_{i-1,j} - h_{i+1,j})/2\Delta x$, and the sign determines how the x-component of the convection term is discretized by one-sided upwind difference. More specifically, if $h_{i-1,j} > h_{i+1,j}$ approximation with this schema is used:

$$\frac{\partial}{\partial x}\left(\alpha s\frac{\partial h}{\partial x}\right) \approx \left(\frac{\alpha_{ij}s_{ij} - \alpha_{i-1j}s_{i-1j}}{\Delta x}\right) \times \left(\frac{h_{i+1j} - h_{i-1j}}{2\Delta x}\right) \tag{4.11}$$

otherwise this is used:

$$\frac{\partial}{\partial x}\left(\alpha s\frac{\partial h}{\partial x}\right) \approx \left(\frac{\alpha_{i+1j}s_{i+1j} - \alpha_{ij}s_{ij}}{\Delta x}\right) \times \left(\frac{h_{i+1j} - h_{i-1j}}{2\Delta x}\right) \tag{4.12}$$

In the y-direction the discretization is done similarly.

4.2.4.3 Boundary Condition (BC)

In the numerical model most of the boundary has a no-flow condition. The entire boundary is using **Homogeneous Neumann Boundary Condition**, and it is described as follows:

$$\frac{\partial h}{\partial n} = \frac{\partial s}{\partial n} = 0 \tag{4.13}$$

One part of the boundary has a non-zero inflow along a 1.5 km wide channel representing the River Kissimmee. Where a river crosses the boundary of the

solution domain, the **Dirichlet Boundary Condition** is set for s and the non-zero **Inhomogeneous Neumann BC** is set for $\frac{\partial h}{\partial n}$. By this, the system allows specifications of an inflow of 80% mud and 20% sand.
On this part of the boundary the fluxes of sand and mud inflow are prescribed as:

$$-\alpha s \frac{\partial h}{\partial n} = f_s \tag{4.14}$$

$$-\beta(1-s)\frac{\partial h}{\partial n} = f_m \tag{4.15}$$

These boundary conditions model an inflow of sediments due to e.g. a river crossing the boundary of the solution domain.

Treatment of the Boundary Condition (BC):
The boundaries can then be described by two parts.
Most of the boundary has a no-flow condition using a second-order accurate treatment of the Homogeneous Neumann BC:

$$\frac{\partial h}{\partial n} = \frac{\partial s}{\partial n} = 0 \tag{4.16}$$

The standard approach by using one layer of ghost points around the boundary points is applied.
Along domain where the river Kissimmee crosses the boundary, special treatment is needed for the inhomogeneous influx conditions The two conditions, 4.14 and 4.15, can be rewritten into the following equivalent form:

$$\frac{\partial h}{\partial n} = -\frac{f_s}{\alpha} - \frac{f_m}{\beta} \tag{4.17}$$

$$s = \frac{\beta f_s}{\beta f_s + \alpha f_m} \tag{4.18}$$

Here h is an Inhomogeneous Neumann BC and s takes a Dirichlet BC.
The derivation of the formulas for the boundary conditions where we have the influx can be seen i Table 4.3.

Influx Sand	**Influx Mud**
$f_s = -\alpha s \frac{\partial h}{\partial n}$	$f_m = -\beta\left(1-s\right)\frac{\partial h}{\partial n}$
$\implies \frac{f_s \partial n}{-\alpha \partial h} = s$	
substitute sand's expression for s into the mud eqn.	$\implies f_m = -\beta\left(1 - \frac{f_s \partial n}{-\alpha \partial h}\right)\frac{\partial h}{\partial n}$
	$\implies f_m = -\beta\left(\frac{\partial h}{\partial n} + \frac{f_s}{\alpha}\right)$
	$\implies -\frac{f_m}{\beta} = \frac{\partial h}{\partial n} + \frac{f_s}{\alpha}$
	$\implies \frac{\partial h}{\partial n} = -\frac{f_s}{\alpha} - \frac{f_m}{\beta}$
rewrite the expression for sand	
$\implies \frac{f_s}{-\alpha \frac{\partial h}{\partial n}} = s$	
	substitute mud's expression for $\frac{\partial h}{\partial n}$ into the sand eqn.
$\implies s = \frac{f_s}{-\alpha\left(-\frac{f_s}{\alpha} - \frac{f_m}{\beta}\right)}$	
$\implies s = \frac{f_s}{f_s + \frac{\alpha f_m}{\beta}}$	
$\implies s = \frac{\beta f_s}{\beta f_s + \alpha f_m}$	

Table 4.3: Derivation of the Formulas for Boundary Conditions at the Influx

4.2.5 Numerical Scheme for Diffusion

Recalling the temporal and the spatial discretization schema with the superscript l for temporal discretization and the subscrtipts i and j for spatial discretization in respectively x and y direction. These schemas are applied on the derivative of the diffusion equation as seen in equation 4.8 and yields this discretized equation:

$$
\begin{aligned}
\frac{h_{ij}^{l+1}-h_{ij}^{l}}{\Delta t} = \frac{1}{C_s}\Bigg[\Bigg[&\frac{\left(\left((\alpha_{ij}s_{ij}^{l}+\alpha_{i+1j}s_{i+1j}^{l})/2\right)\cdot\left(h_{i+1j}^{l}-h_{ij}^{l}\right)\right)-\left(\left((\alpha_{i-1j}s_{i-1j}^{l}+\alpha_{ij}s_{ij}^{l})/2\right)\cdot\left(h_{ij}^{l}-h_{i-1j}^{l}\right)\right)}{\Delta x^2}\Bigg]+\\
&\Bigg[\frac{\left(\left((\alpha_{ij}s_{ij}^{l}+\alpha_{ij+1}s_{ij+1}^{l})/2\right)\cdot\left(h_{ij+1}^{l}-h_{ij}^{l}\right)\right)-\left(\left((\alpha_{ij-1}s_{ij-1}^{l}+\alpha_{ij}s_{ij}^{l})/2\right)\cdot\left(h_{ij}^{l}-h_{ij-1}^{l}\right)\right)}{\Delta y^2}\Bigg]\Bigg]\\
+\frac{1}{C_m}\Bigg[\Bigg[&\frac{\left(\left((\beta_{ij}(1-s_{ij}^{l})+\beta_{i+1j}(1-s_{i+1j}^{l}))/2\right)\cdot\left(h_{i+1j}^{l}-h_{ij}^{l}\right)\right)-\left(\left((\beta_{i-1j}(1-s_{i-1j}^{l})+\beta_{ij}(1-s_{ij}^{l}))/2\right)\cdot\left(h_{ij}^{l}-h_{i-1j}^{l}\right)\right)}{\Delta x^2}\Bigg]+\\
&\Bigg[\frac{\left(\left((\beta_{ij}(1-s_{ij}^{l})+\beta_{ij+1}(1-s_{ij+1}^{l}))/2\right)\cdot\left(h_{ij+1}^{l}-h_{ij}^{l}\right)\right)-\left(\left((\beta_{ij-1}(1-s_{ij-1}^{l})+\beta_{ij}(1-s_{ij}^{l}))/2\right)\cdot\left(h_{ij}^{l}-h_{ij-1}^{l}\right)\right)}{\Delta y^2}\Bigg]\Bigg]
\end{aligned}
\tag{4.19}
$$

The diffusion is then solved with respect to h^{l+1}, displayed in equation (4.20). This will also be the numerical schema used for the implementation of the diffusion.

$$
\begin{aligned}
h_{ij}^{l+1} = \frac{\Delta t}{C_s}\Bigg[\Bigg[& \frac{\Big(\big((\alpha_{ij}s_{ij}^{l}+\alpha_{i+1j}s_{i+1j}^{l})/2\big)\cdot\big(h_{i+1j}^{l}-h_{ij}^{l}\big)\Big)-\Big(\big((\alpha_{i-1j}s_{i-1j}^{l}+\alpha_{ij}s_{ij}^{l})/2\big)\cdot\big(h_{ij}^{l}-h_{i-1j}^{l}\big)\Big)}{\Delta x^2}\Bigg]+ \\
& \Bigg[\frac{\Big(\big((\alpha_{ij}s_{ij}^{l}+\alpha_{ij+1}s_{ij+1}^{l})/2\big)\cdot\big(h_{ij+1}^{l}-h_{ij}^{l}\big)\Big)-\Big(\big((\alpha_{ij-1}s_{ij-1}^{l}+\alpha_{ij}s_{ij}^{l})/2\big)\cdot\big(h_{ij}^{l}-h_{ij-1}^{l}\big)\Big)}{\Delta y^2}\Bigg]\Bigg] \\
+\frac{\Delta t}{C_m}\Bigg[\Bigg[& \frac{\Big(\big((\beta_{ij}(1-s_{ij}^{l})+\beta_{i+1j}(1-s_{i+1j}^{l}))/2\big)\cdot\big(h_{i+1j}^{l}-h_{ij}^{l}\big)\Big)-\Big(\big((\beta_{i-1j}(1-s_{i-1j}^{l})+\beta_{ij}(1-s_{ij}^{l}))/2\big)\cdot\big(h_{ij}^{l}-h_{i-1j}^{l}\big)\Big)}{\Delta x^2}\Bigg]+ \\
& \Bigg[\frac{\Big(\big((\beta_{ij}(1-s_{ij}^{l})+\beta_{ij+1}(1-s_{ij+1}^{l}))/2\big)\cdot\big(h_{ij+1}^{l}-h_{ij}^{l}\big)\Big)-\Big(\big((\beta_{ij-1}(1-s_{ij-1}^{l})+\beta_{ij}(1-s_{ij}^{l}))/2\big)\cdot\big(h_{ij}^{l}-h_{ij-1}^{l}\big)\Big)}{\Delta y^2}\Bigg]\Bigg] \\
+ h_{ij}^{l} &
\end{aligned}
\tag{4.20}
$$

4.2.6 Numerical Scheme for Convection

For convection the discretization schema is applied on the derivative of the equation, as displayed in 4.9:

$$\begin{aligned}
A\left(\frac{s^{l+1}-s^{l}}{\Delta t}\right)+s^{l+1}\left(\frac{h^{l+1}-h^{l}}{\Delta t}\right) &= \frac{1}{C_s}\left[f\ldots\right]\\
A\left(s^{l+1}-s^{l}\right)+s^{l+1}\left(h^{l+1}-h^{l}\right) &= \frac{\Delta t}{C_s}\left[f\ldots\right]\\
As^{l+1}-As^{l}+s^{l+1}h^{l+1}-s^{l+1}h^{l} &= \frac{\Delta t}{C_s}\left[f\ldots\right]\\
As^{l+1}+s^{l+1}h^{l+1}-s^{l+1}h^{l} &= \frac{\Delta t}{C_s}\left[f\ldots\right]+As^{l}\\
s^{l+1}\left(A+\left(h^{l+1}-h^{l}\right)\right) &= \frac{\Delta t}{C_s}\left[f\ldots\right]+As^{l}\\
\frac{s^{l+1}\left(A+h^{l+1}-h^{l}\right)}{\left(A+h^{l+1}-h^{l}\right)} &= \frac{\frac{\Delta t}{C_s}\left[f\ldots\right]+As^{l}}{\left(A+h^{l+1}-h^{l}\right)}\\
s^{l+1} &= \frac{\frac{\Delta t}{C_s}\left[f\ldots\right]+As^{l}}{\left(A+h^{l+1}-h^{l}\right)}
\end{aligned} \tag{4.21}$$

When the expression for the function as shown in the section on Spatial Discretization of Convction 4.2.4.2 is filled in between the hard brackets with respect to x and y in solution (4.21), the first line of the equation (4.21) will result in this discretized expression:

$$A\left(\frac{s_{ij}^{l+1}-s_{ij}^{l}}{\Delta t}\right)+s_{ij}^{l+1}\left(\frac{h_{ij}^{l+1}-h_{ij}^{l}}{\Delta t}\right)=\begin{cases}\frac{1}{C_s}\left[\left(\frac{\alpha_{ij}s_{ij}^{l}-\alpha_{i-1j}s_{i-1j}^{l}}{\Delta x}\right)\times\left(\frac{h_{i+1j}^{l+1}-h_{i-1j}^{l+1}}{2\Delta x}\right)\right], & h_{i-1j}>h_{i+1j}\\ \frac{1}{C_s}\left[\left(\frac{\alpha_{i+1j}s_{i+1j}^{l}-\alpha_{ij}s_{ij}^{l}}{\Delta x}\right)\times\left(\frac{h_{i+1j}^{l+1}-h_{i-1j}^{l+1}}{2\Delta x}\right)\right], & h_{i-1j}<h_{i+1j}\end{cases}$$

$$+\begin{cases}\frac{1}{C_s}\left[\left(\frac{\alpha_{ij}s_{ij}^{l}-\alpha_{ij-1}s_{ij-1}^{l}}{\Delta y}\right)\times\left(\frac{h_{ij+1}^{l+1}-h_{ij-1}^{l+1}}{2\Delta y}\right)\right], & h_{ij-1}>h_{ij+1}\\ \frac{1}{C_s}\left[\left(\frac{\alpha_{ij+1}s_{ij+1}^{l}-\alpha_{ij}s_{ij}^{l}}{\Delta y}\right)\times\left(\frac{h_{ij+1}^{l+1}-h_{ij-1}^{l+1}}{2\Delta y}\right)\right], & h_{ij-1}<h_{ij+1}\end{cases} \tag{4.22}$$

The last line in equation (4.21) shows this result when it is fully discretized with respect to s^{l+1}:

$$s_{ij}^{l+1}=\begin{cases}\dfrac{\frac{\Delta t}{C_s}\left[\left(\frac{\alpha_{ij}s_{ij}^{l}-\alpha_{i-1j}s_{i-1j}^{l}}{\Delta x}\right)\times\left(\frac{h_{i+1j}^{l+1}-h_{i-1j}^{l+1}}{2\Delta x}\right)\right]+As_{ij}^{l}}{\left(A+h_{ij}^{l+1}-h_{ij}^{l}\right)}, & h_{i-1j}>h_{i+1j}\\ \dfrac{\frac{\Delta t}{C_s}\left[\left(\frac{\alpha_{i+1j}s_{i+1j}^{l}-\alpha_{ij}s_{ij}^{l}}{\Delta x}\right)\times\left(\frac{h_{i+1j}^{l+1}-h_{i-1j}^{l+1}}{2\Delta x}\right)\right]+As_{ij}^{l}}{\left(A+h_{ij}^{l+1}-h_{ij}^{l}\right)}, & h_{i-1j}<h_{i+1j}\end{cases}$$

$$+\begin{cases}\dfrac{\frac{\Delta t}{C_s}\left[\left(\frac{\alpha_{ij}s_{ij}^{l}-\alpha_{ij-1}s_{ij-1}^{l}}{\Delta y}\right)\times\left(\frac{h_{ij+1}^{l+1}-h_{ij-1}^{l+1}}{2\Delta y}\right)\right]+As_{ij}^{l}}{\left(A+h_{ij}^{l+1}-h_{ij}^{l}\right)}, & h_{ij-1}>h_{ij+1}\\ \dfrac{\frac{\Delta t}{C_s}\left[\left(\frac{\alpha_{ij+1}s_{ij+1}^{l}-\alpha_{ij}s_{ij}^{l}}{\Delta y}\right)\times\left(\frac{h_{ij+1}^{l+1}-h_{ij-1}^{l+1}}{2\Delta y}\right)\right]+As_{ij}^{l}}{\left(A+h_{ij}^{l+1}-h_{ij}^{l}\right)}, & h_{ij-1}<h_{ij+1}\end{cases} \tag{4.23}$$

Chapter 5

High Performance Computing (HPC)

Before describing the implementation of both the serial and the parallel applications, a survey on ideas and concepts regarding High Performance Computing (HPC) will be presented in this chapter. HPC is the use of parallel processing technologies for running applications efficiently. The efficiency can be measured in different ways.

Clock speed/Clock rate is the time used by a microprocessor to execute an instruction. All computers contains a clock that regulates the speed. Clock rates are expressed in megahertz (MHz) or gigahertz (GHz).
Throughput is a measure of how many units of information a (computer-) system can process in a given time-period. The throughputs are often described as **FLoating-point OPerations per Second (FLOPS)**. This is a benchmark measurement for the speed of microprocessors.
The earliest supercomputers had a throughput measured in
kiloFLOPS (KFLOPS) = 10^3 FLOPS and
megaFLOPS (MFLOP) = 10^6 FLOPS.
For supercomputers today we are talking about throughput measured in teraFLOPS (TFLOPS) = 10^{12} FLOPS and
petaFLOPS (PFLOPS) = 10^{15} FLOPS.
Eager scientists are even looking towards exa-scale computing for a throughput on the size of exaFLOPS (EFLOPS) = 10^{18} FLOPS.

The term HPC applies to systems performing above a teraflop (10^{12}) floating point operations per second.

5.1 Supercomputing in a Historical Perspective

The basic components used for electronics in radios, televisions, radars, telephones and computers was the vacuum tube (electron tube), a device that was used to control electrical current through a vacuum in a sealed container. The ones with a control grid could also be used as amplifiers. The vacuum tubes was invented in 1907 and are mostly used until the first half of the twentieth century.

In 1947 came the first transistor, a semiconductor device used to amplify and switch electronic signal and electrical power, invented by the American physicists John Bardeen, Walter Brattain, and William Shockley. The transistor revolutionized the field of electronics, and the inventors were rewarded the Nobel Prize in Physics in 1956.

Among the advantages that have allowed the transistors to replace the vacuum tube processors are lower power dissipation and higher energy efficiency.

The first electronic general purpose computer was **Electronic Numerical Integrator and Computer (ENIAC)**. It was Turing-complete (computationally universal) and reprogrammable to solve a large class of numerical problems. The press called it "Giant Brain" when it was announced in 1946. It was primarily designed to calculate artillery firing tables for the US Army.

The **Turing machine** is named after the British pioneering computer scientist, mathematician, cryptanalyst and logician, Alan Mathison Turing (1912-1954). With the development of the Turing machine, a theoretical or hypothetical device, he formalized the concepts of "algorithm" and "computation". An algorithm is a step-by-step set of operations/instructions to be performed, and computation is a calculation or use of computing technology in information processing.

In the 1960's various supercomputers were introduced.

As early as November 1959 the **IBM** 7090 was installed, designed for large-scale scientific and technological applications. This was a transistorized version of the earlier IBM 709, a vacuum tube computer. The processing speed was around 100 KFLOP/s and about six times faster than the 709.

The first **Atlas** supercomputer was commissioned in 1962 and installed at the University of Manchester. The machine used discrete germanium transistors to lower the voltage.

Another supercomputer was introduced by Seymour Cray, and in 1976 the 80 MHz **Cray**-1 system was installed with a single CPU. The 80 MFLOPS Cray-1 was in 1982 succeeded by the 800 MFLOPS Cray X-MP, the first multi-processing computer. This was a shared-memory parallel vector processor with a clock cycle of 9.5 nano seconds (ns) (105 MHz) compared to Cray-1's clock cycle of 12.5 ns. A vector processor, or array processor, is a CPU that has implemented an instruction set able to operate on one-dimensional arrays called vectors, in contrast to a scalar processor that only operates on single data units. In 1984 an improved model of the Cray X-MP came with up to a four-processor system.

Twice a year the TOP500 list of supercomputers [*Top500List* 2015] is published, and one machine has been on the top of the list the last times the list has come out. China's **Tianhe-2** supercomputer, which translates to "Milky Way 2", has been the fastest in the world since June 2013. It runs at 33.86 petaFLOPS (PFLOPS), or 33.86 quadrillion floating point operations per second. Or in other words; 33.86 thousand million million floating point operations per second.

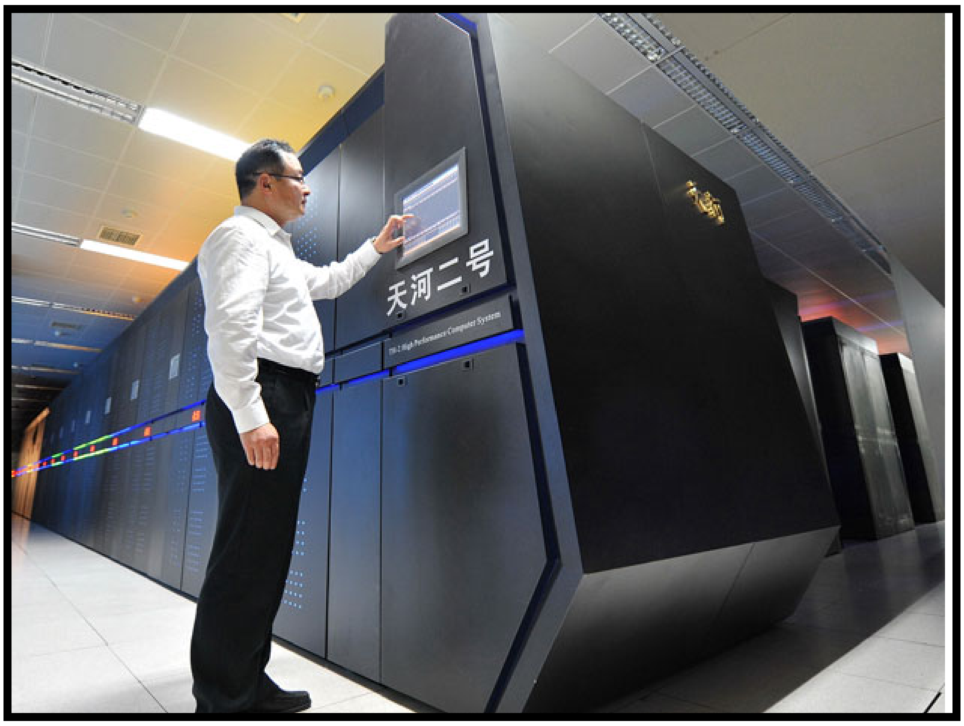

Figure 5.1: Tianhe-2, Chinese Supercomputer

Chapter 6

Parallel Computer Architectures

6.1 Parallel Computing

Workstations today are a hundred times faster than the ones made a decade ago. Yet scientists and engineers still experience that "fast" isn't fast enough. Waiting for hours, days or even weeks for larges computations to finish can be costly. The use of a parallel computers to solve a computational problem is called parallel computing, and is widely used to solve large problems. A parallel computer is a multi-processor computer system where computations on different processors are run in parallel. The use of parallel architectures to solve computational problem often saves a lot of time, but it is important to make the right choices of computer architecture and algorithm design, to achieve the most optimal result.

6.1.1 Flynn's Taxonomy - A Scheme for Parallel Architectures

Operations processed in a computer is a set of instructions that operates on a set of data operands. These processes can be executed as a stream of instructions operating on a stream of data. The possibility of multiplicity in the streams of instructions and data lies in the computer architecture.

A well-known classifications scheme for parallel computer architectures is Flynn's Taxonomy, proposed by Michael J. Flynn in 1966 [*Quinn* 2003]. In this scheme a computer is categorized on it's ability to exhibit parallelism in the instruction stream and the data stream., see figure 6.1

	Single Instruction	*Multiple Instructions*
Single Data	**SISD** Uniprocessors	**SIMD** Processor arrays Pipelined vector processors
Multiple Data	**MISD** Systolic arrays	**MIMD** Multiprocessors Multicomputers

Table 6.1: Flynn's Taxonomy for Parallel Computer Architecture

Single Instruction, Single Data stream (SISD)
This is the architecture of a sequential computer which exploits no parallelism neither in the instruction nor data streams. An example is the **uniprocessor**, a system where all processing tasks share a single CPU.

Single Instruction, Multiple Data stream (SIMD)
Computers with this architecture employ a single instruction stream but multiple data streams. **Processor arrays** and **pipelined vector processors** are examples in this category, and both technologies employs a single control unit executing one instruction or instruction stream at the time. A processor array has multiple processor elements operating in parallel on multiple data elements in a data stream, and the same instruction is performed on each data element. The vector processor has a single processor element that operates in sequence on the data elements.

Multiple Instruction, Single Data streams (MISD)
The MISD architecture employs a pipeline of multiple independently executing instructions performed on a single data stream. The **systolic array** exemplifies this kind of computer, forwarding results from one function to the next on a single data element.

Multiple Instruction, Multiple Data streams (MIMD)
The MIMD class of parallel architectures is for computers consisting of multiple instruction streams performed on multiple data streams. **Multiprocessors** and **multicomputers** fit into this category, and both architectures are based on multi-

ple CPU's where the different CPU's simultaneously execute different instructions streams on multiple data elements. This has possibly been the most familiar form of parallel architectures.

6.1.2 Designing a Parallel Algorithm

6.1.2.1 The Task/Channel Model for Parallel Architectures)

When designing a parallel algorithm, Ian Fosters [*Quinn* 2003] **task/channel methodology** can be used. The model is intended for MIMD architectures, and not for SIMDs. MIMD architectures are multi-processors and multi-computers witch consists of multiple CPU's with a shared memory. The MIMD architecture typically have fewer processors than the SIMD does.

In the case of constructing the algorithm for the numerical model for sediment transport, the channel/task method will be presented here to enlighten a few topics around choices for designing parallel algorithms.
The task/channel model represents a parallel computation as a set of tasks that may interact with each other by sending messages through a channel. A **task** is a program's instructions and private data on the local memory and a collection of Input/Output (I/O) ports, and a task can use the ports to send local data to other tasks. One task's output port can connect with another task's input port through a message queue called a **channel**.

In the task/channel model any task can send data to other tasks and continue it's instructions regardless if the receiving task has received the data. Sending is a non-blocking process and is said to be an asynchronous operation. On the other hand a task can not receive data from another task before the other task has sent it, so the task is blocked until the value is received. Receiving is then said to be a blocking process, and is a synchronous operation.
In this model:

- The **execution time** of a parallel algorithm is the period of time the parallel tasks are active.
- The **starting time** of a parallel algorithm is when all tasks simultaneously starts executing.
- The **finishing time** of a parallel algorithm is when the last task has stopped executing.

Ian **Foster's design methodology** proposes a four-step process for designing parallel algorithms. These design steps are **partitioning**, **communication**, **ag-**

glomeration and **mapping**.

Partitioning
In the search for how much parallelism is possible in a system, the methodology starts with a partitioning step. This process divides data and computations into pieces, and can take either a data-centric or a computation-centric approach.

The data-centric design approach for parallel algorithms is where data is divided into pieces and then computations are associated with the data pieces. This is called **domain decomposition** and results in sets of data and a number of operations to be performed on the data. Some examples of domain decomposition is when data is divided in different dimensions (1D, 2D and 3D).

The computation-centric design approach is where computations are divided into tasks and then it's decided which data are associated with the individual computations. This is called **functional decomposition**, and the collection of tasks can often achieve concurrency through pipelining.

Communication
The next step is to determine the communication pattern between the tasks. If data needs to be shared between the tasks, then communication is needed.
The two possible communication patterns for parallel algorithms are **local communication** and **global communication**.
Communication is not needed for a sequential algorithm, and the extra time it takes to perform the communication among tasks in a parallel algorithm is called **overhead**. An important goal of parallel algorithm design is to minimize the overhead. The overhead may be strongly affected by how the tasks are assigned to the processors, especially if the number of tasks greatly exceeds the number of processors. To help evaluate the communication structure of the parallel system, Foster has a checklist:

- Balance communication operations among the tasks.
- Each task communicates only with a small number of neighbors.
- Tasks can perform their communications concurrently.
- Tasks can perform their computations concurrently.

Agglomeration
After achieving as much parallelism as possible through the first two design steps, it is time to group the primitive tasks into larger tasks to improve performance,

simplify programming and maintain scalability of a program. This process s called agglomeration. Reducing communication overhead is one of the goals of agglomeration. One way to reduce the communication to zero is to agglomerate all primitive tasks that communicates with each other. This is called **increasing the locality**. Replicating computations on different processors may also take less time than communicating the data over a channel. This replication must be small enough to allow a system to scale.

Mapping
Mapping is the process of assigning the different tasks to the various processors in a way that maximizes processor utilization and minimizes interprocessor communication costs, but these two goals are often conflicting. Processor utilization is the average percentage of time a systems's processors actively performs tasks to solve a problem. Computation time may vary for different tasks. When computations are balanced evenly, the processor utilization is maximized.
Different characteristics in the parallel algorithm will lead to different mapping strategies. Mapping can be part of a static or dynamic task allocation. One example on when to choose dynamic load-balancing algorithms is in the case where tasks are created and destroyed runtime and has frequent communications between the tasks. An example of when to choose static load-balancing algorithms to minimize the communication overhead, is when you have a static number of tasks with unstructured communication patterns.

6.1.3 General-Purpose Parallel Computer Architecture

The insatiable market demand for real-time, high-definition 3D-graphics has driven producers of GPU's to evolve the general-purpose programmable, highly parallel, multithreaded, many-core processors with extreme computational power and high bandwidth. The GPU's are devoting more transistors to data processing than to data caching and flow control as the CPU's are. This gives the GPU an advantage over the CPU for data-parallel computations, where the same instruction is executed on numerous data elements in parallel. See figure 6.1

6.1.3.1 CUDA - Compute Unified Device Architecture

Reasons why there is such a large performance gap between CPU's and GPU's lies in the fundamental differences in the design of the hardware.
The Compute Unified Device Architecture (CUDA) [*NVIDIA CUDA Toolkit Documentation, v7.0*] is a GPGPU architecture that was introduced by NVIDIA Corporation in November 2006. Since then different generations of GPU's has been evolved, and for different purposes, whether it's been made for graphics rendering

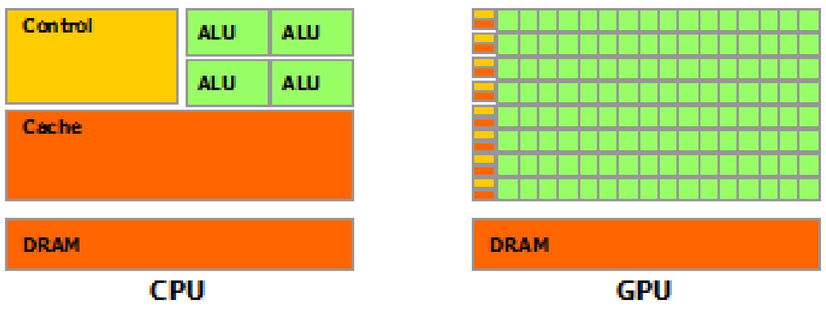

Figure 6.1: GPU devoting more transistors to data processing than CPU [*NVIDIA CUDA Toolkit Documentation, v7.0*]

or general purpose tasks. CUDA also came with a programming platform including adapted libraries, middleware and extensions to programming languages. This opened up for the possibility to make parallel systems solve complex computational problems more efficiently than on a CPU. A figure of the GPU Computing Applications can be seen in figure 6.2

A CUDA capable architecture holds one or multiple **Stream Multiprocessors (SM's)**. Each SM contains several **Streaming Processors (SP's)**.
The **memory hierarchy** spans from high speed to low speed in this order:

- Registers
- Shared Memory
- Constant Memory
- Texture Memory
- Global memory/Device memory

Table 6.2: CUDA Memory Hierarchy

where registers are the fastest to access and global memory the slowest.
GPU's keep a high memory bandwidth that greatly exceeds CPU's. The number of SM's, SP's and memory details varies with the different generations of GPU's. The CUDA architecture is displayed in the figure 6.3.

The CUDA architecture is supplemented with a software environment that allows developers to use C, C++, FORTRAN or other programming languages, apply programming interfaces or directive-based approaches.

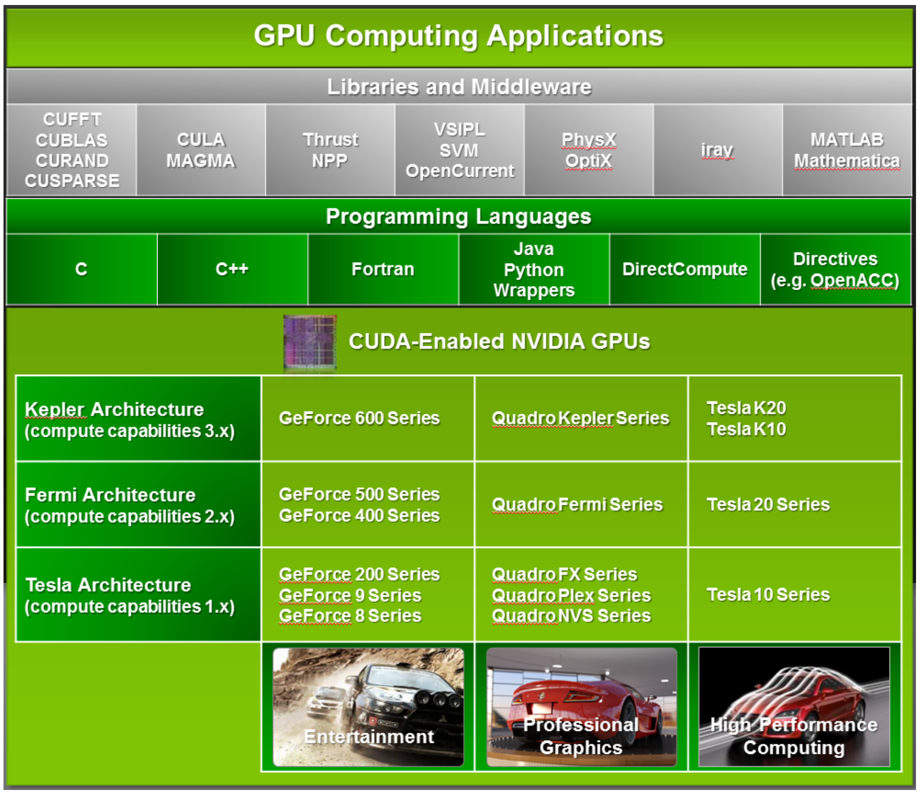
GPU Computing Applications
Libraries and Middleware
CUFFT CUBLAS CURAND CUSPARSE
CULA MAGMA
Thrust NPP
VSIPL SVM OpenCurrent
PhysX OptiX
iray
MATLAB Mathematica
Programming Languages
C
C++
Fortran
Java Python Wrappers
DirectCompute
Directives (e.g. OpenACC)
CUDA-Enabled NVIDIA GPUs
Kepler Architecture (compute capabilities 3.x)
GeForce 600 Series
Quadro Kepler Series
Tesla K20 Tesla K10
Fermi Architecture (compute capabilities 2.x)
GeForce 500 Series GeForce 400 Series
Quadro Fermi Series
Tesla 20 Series
Tesla Architecture (compute capabilities 1.x)
GeForce 200 Series GeForce 9 Series GeForce 8 Series
Quadro FX Series Quadro Plex Series Quadro NVS Series
Tesla 10 Series
Entertainment
Professional Graphics
High Performance Computing

Figure 6.2: GPU Computing Applications [*NVIDIA CUDA Toolkit Documentation, v7.0*]

The three key abstractions presented in the CUDA programming model are the hierarchy of thread groups, shared memories and barrier synchronization. This gives the platform for the coarsely grained data parallelism and task parallelism and within that the fine-grained data parallelism and thread parallelism. The developer may then partition a problem into groups of smaller problems solved in blocks of threads. A CUDA program can execute on a various number of processor cores and therefore supports automatic scaling of a problem. See fig: 6.2

6.1.3.2 CUDA C Programming Model

CUDA C is an extension to the C programming language, evolved to allow parallel execution of programs on the GPU. It let's developers familiar to the C programming language exploit the technology without the effort of learning a completely new programming language. The CUDA C programming model is benefiting from the CUDA-kernels that let the programmer (re)define c-functions. When a kernel is called it executes N times in parallel by N different CUDA threads, while

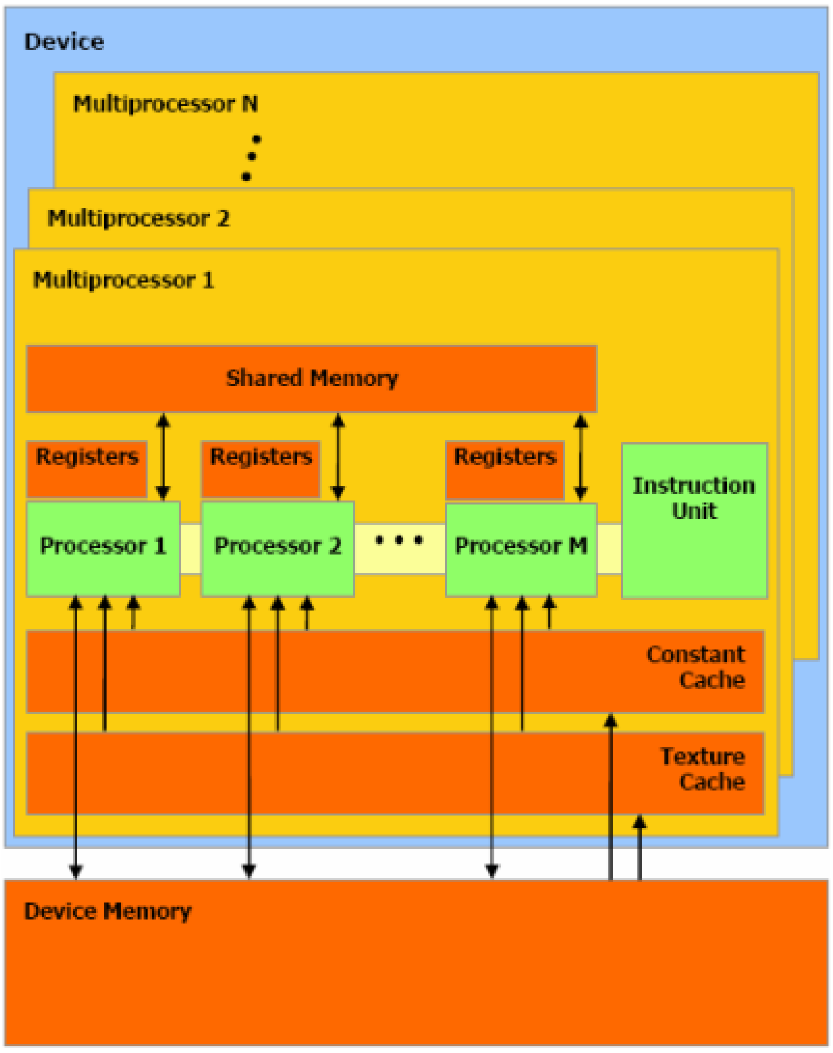

Figure 6.3: The CUDA Architecture [*NVIDIA CUDA Toolkit Documentation, v7.0*]

a C-function only executes once. The kernel-call must have defined:

- the number of threads on each block
- the number of thread-blocks in a grid

A kernel call uses a new `<<<...>>>` execution configuration syntax, and the invocation of a kernel may look like the code listing 6.1

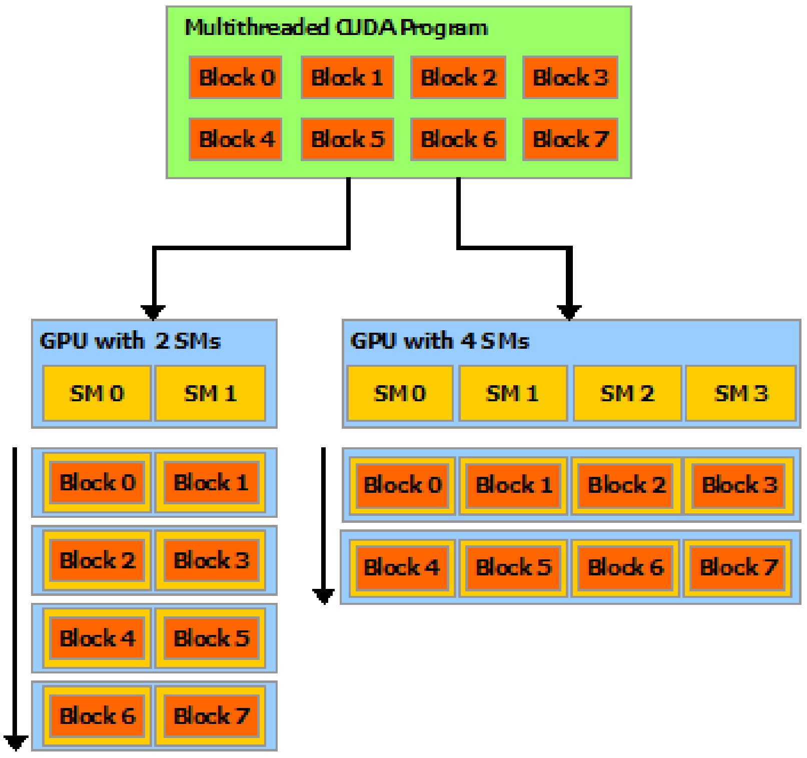

Figure 6.4: Automatic scalability in the CUDA architecture [*NVIDIA CUDA Toolkit Documentation, v7.0*]

Listing 6.1: CUDA Kernel Call

```
kernelName<<<numBlocks, numThreads>>>(argument_1, ...);
```

When the kernel is called from a C-function it must have the `__global__` declaration specifier, and it may be defined as shown the code listing: 6.2

Listing 6.2: CUDA Kernel Declaration

```
__global__ void kernelName(argument_1, ...)
{
        ...
}
```

The number of thread-blocks in the grid and the number of threads per block is for the programmer to decide through the variables specified in the kernel-call. In the code-example 6.1 these variables are called numBlocks and numThreads, and they may be of type `int` or `dim3`. The blocks of threads are organized into a 1D, 2D or 3D grid. Figure 6.5 displays threads on blocks and blocks in a grid. The index of a block can be accessed inside the kernel through the built-in variable **blockIdx**, and the size of the block through the variable **blockDim**.

A unique **thread-ID** is given each thread that executes the kernel, and the thread-ID is also accessible within the kernel through the built-in variable **threadIdx**. The threadIdx defines a threads position in a 1D, 2D or a 3D thread block, and is therefore a 3-component vector. Defining the unique thread-ID is based on the thread's indexed position. For a thread in a 2D system of dimensions (D_x, D_y), the ID of the thread at index (x, y) is $(x + yD_x)$. For a 3D block of size $D_x, D_y, D_z)$ the ID of a thread with the index (x, y, z) is $(x + yD_x + zD_xD_y)$.

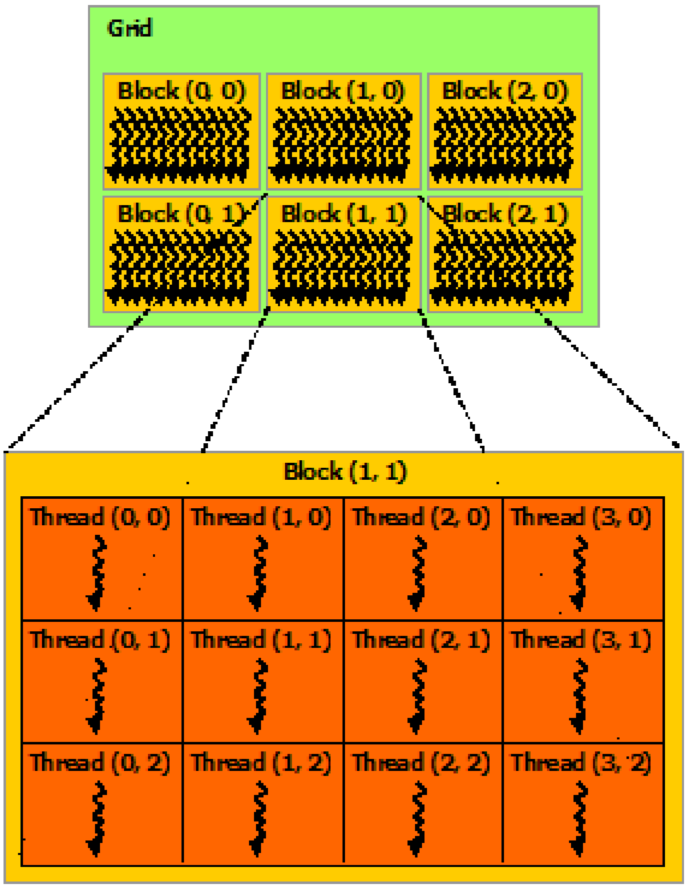

Figure 6.5: Grid of thread block in the CUDA architecture [*NVIDIA CUDA Toolkit Documentation, v7.0*]

Thread-blocks can be executed in any order. The threads of a thread block execute concurrently on one multiprocessor, and a multiprocessor may also execute mul-

tiple thread blocks concurrently. When thread blocks terminate execution, new blocks will be launched for execution on the vacant multiprocessors.
The unique architecture that NVIDIA calls **Single-Instruction, Multiple-Thread (SIMT)** allows a multiprocessor to execute hundreds of threads concurrently. The SIMT architecture is akin to SIMD (Single-Instruction, Multiple Data streams), but the SIMD architecture operates a single instruction to control multiple data on a vector processor, whereas the SIMT architecture operates on single or multiple threads. The SIMT instructions specifies the execution and branching behavior of each single thread. SIMT is intended to limit instruction fetching overhead and the instructions are pipelined for instruction-level parallelism within a single thread. With simultaneous hardware multi–treading, SIMT also offers extensive **thread-level parallelism**. **Data parallelism** [*Kirk* and *Wen-mei* 2012] is achieved with the SIMD architecture where data is distributed across different parallel computing nodes executing the single, same instruction on each data element. Data parallelism can also be achieved with the SIMT architecture as a single instruction can operate on multiple threads in parallel, where each thread operates on a data element.

The CUDA-threads can access data from the various memory spaces. Each thread can access it own (per-thread) **local memory**. All threads within a block can share data through the (per-block) **shared memory**. The **global memory** can be accessed by all threads. **Texture** and **constant memory** spaces can also be accessed by all threads, but these memory spaces are read-only. An organization of the CUDA memory hierarchy is revealed in figure 6.6. In addition to the global memory, the constant and texture memories are persistent during an applications kernel launch.

The CUDA programming model assumes that the C-code is executed on the host (CPU) and all the CUDA-kernels with the CUDA-threads executes on a device (GPU), a physically separate coprocessor. This model also expects that both the host and the device maintains their respective memory spaces in DRAM. The program must manage memory allocations and deallocations, and data transfer between the host and the device memory.

More details on CUDA will be conveyed in Chapter 7 Heterogeneous Computing with CUDA C.

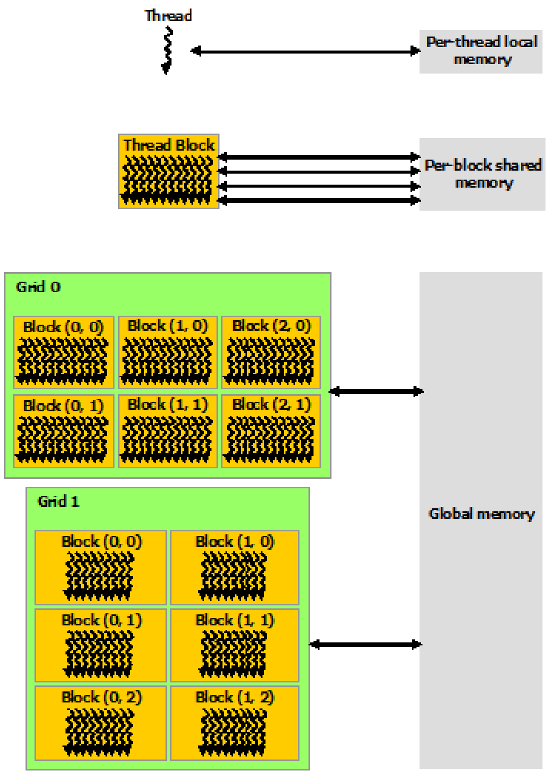

Figure 6.6: Memory Hierarchy of the CUDA architecture [REF: NVIDIA]

Chapter 7

Heterogeneous Computing with CUDA C

7.1 Heterogeneous Computing

In the world of HPC, **General-Purpose computing on Graphics Processing Units (GPGPU)** is considered to be a "bleeding edge technology". In this research, GPU's from NVIDIA will be investigated for their compute capabilities. NVIDIA GPUs are built on the CUDA Architecture. The hardware is constructed to perform **general-purpose** tasks in addition to **traditional graphics-rendering** tasks.

To construct computer-programs to be running general-purpose computations on CUDA GPUs, it's convenient use the CUDA C programming language that extends the C programming language. The CUDA C language is a powerful tool for parallel programming. CUDA C is easy to combine with C by making a few changes, and will be used for the parallel implementation of the sediment-transport application. The C-functions must then be rewritten into CUDA-kernels. In such applications the kernels will run on a GPU, while the CPU handles the scheduling of the kernels. It's typically referred to as **heterogeneous computing** when a system exploits more than one kind of processor.

This parallel application takes advantage of the GPU for it's massive processing power, but uses the CPU to run the operating system. It is heterogeneous for a mixed serial-parallel programming, serial program with parallel kernel. In this parallel context, a heterogeneous system consists of a **host** and a **device** or multiple devices, each with separate processor-capacities and separate memory spaces that they maintain. The memories are referred to as a **host memory** and **device**

memory respectively.

7.2 Parallel Programming Features in CUDA C

CUDA's extension set to the C language, CUDA C, has the runtime implemented in the **cudart** runtime library. It initializes the first time a function is called, and needs no explicit initialization. The entry points are prefixed `cuda`.

7.2.1 Parallel kernel

The CUDA-kernels runs on the the CUDA device, a GPU that is physically separated from the CPU. The CUDA kernel is like a C-function, but the code is executed in parallel. The massive data-load is sectioned into grids consisting of thread blocks, and each thread on the block performs computations for one data-element at one grid point. For explicit synchronization of the threads the intrinsic function

```
__syncthreads()
```

can be called. The function acts as a barrier, and waits for all the threads on a block to finish before it can proceed.

Kernels can operate out of device on memories through functions provided by the runtime library. These functions lets the system perform allocations, copies and deallocations of data. It also permits transfer of data between host and device or between the different devices in a multiple GPU system. It can also directly load and store data between GPUs.

7.2.2 Scalability

A parallel kernel are composed of very many lightweight threads, grouped into thread blocks. CUDA is exceptionally **scalable** [*Nickolls* et al. 2008] and has a **hierarchical execution model:**

- Decompose problem into sequential steps such as kernels
- Decompose kernel into computing parallel blocks
- Decompose block into computing parallel threads

Table 7.1: CUDA's Hierarchical Execution Model

The hardware distributes independent blocks to Streaming Multiprocessors (SMs) as available, and schedules independent threads of execution. The blocks can run concurrently or sequentially, they are independent thus scalable. A block is a virtualized multiprocessor. When a kernel is launched it fits processors to data and computation.

There are different levels of parallelism:

- Thread parallelism
  - each threads an independent thread of execution
- Data parallelism
  - across threads in a block
  - across blocs in a kernel
- Task parallelism
  - different blocks are independent
  - independent kernels

Table 7.2: Levels of Parallelism in CUDA

7.2.3 CUDA Device Memory

The CUDA Architecture handles it's own memory, and the CUDA-kernels can only access the memory on the device. As shown in figure 6.6 we recall that:

- Global memory - can be accessed from all the blocks in the grid
- Shared memory - can be accessed by all the threads on a block
- Local memory - can be accessed by a single thread

The device memory can be allocated as a linear memory or as CUDA-arrays.
The device memory allocation and deallocation has to be handled separately using **`cudaMalloc()`** and **`cudaFree()`**. The transfer of data between the host and device memory also needs to be handled explicitly, and can be done with **`cudaMemcpy()`**.

The various **`cudaMemcpy*()`** functions takes the parameter: **`cudaMemcpyKind`**

which is used as a flag describing the kind of copying that is being performed, defining if the copy goes from a CPU or GPU and to a CPU or GPU. The choices for the "kind" is then:

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice
- cudaMemcpyDefault

Table 7.3: Different "kinds" of Memory Copy in CUDA

7.2.3.1 Linear memory

Linear memory for 2D or 3D objects is recommended to be allocated using **`cudaMallocPitch()`** and **`cudaMalloc3D()`**. The function **`cudaMalloc3D()`** allocates at least $width \times height \times depth$ bytes of linear memory. A 2D linear memory is allocated with **`cudaMallocPitch()`**. This allocates the memory of a size that is at least $width$ (in bytes) $\times height$.
These functions also make sure to pad the allocated memory to meet the hardware requirements, if needed. By this it ensures the best performance when accessing row addresses or copying between regions of device memory. Both the 2D and the 3D allocation mentioned, are padding the memory to a size being a multiple of the warp-size. Since data are fetched with the size of a warp, the memories are then optimized for this. The size of the padded width is returned in a variable called pitch, and holds the size of the stride in number of bytes.
Both the 2D and the 3D allocation, returns a a pointer to the memory.

7.2.3.2 Page-Locked Host Memory

The runtime provides functions with the possibility for a page-locked host memory, also referred to as **pinned memory**, which can be allocated using **`cudaHostAlloc()`** or **`cudaMallocHost()`** and freed with **`cudaFreeHost()`**.

Page-locked memory guarantees that the operating system will never page out this memory to disk, and ensures it's space on the physical memory. A page-locked host memory benefits form the possibility to perform copies between the device memory and the page-locked host memory concurrently with kernel execution.

Mapped Memory - Zero Copy
Pinned memory can also be **mapped** directly into the host's memory space from

some devices, a feature that eliminates memcpy between host and device. The device code will then transfer data implicitly as needed. For a mapped memory, the setup must be done on the host with proper flags, and the memory must be allocated with the function `cudaHostAlloc()` with the flag `cudaHostAllocMapped`. This is also referred to as **Zero copy**. For integrated systems that utilize the CPU memory, Zero copy will be faster. Also for systems where data are read/written from/to global memory once during the application the Zero copy is fast.

If data has to be read/written often throughout the simulations, a mapped memory will run slow due to massive synchronization of different devices trying to read and write to the same data concurrently.

Portable Memory
An other option is to utilize a pinned memory allocation using a portable memory. A block of page-locked memory is only available to the device that was current when the the block was allocated, and to all the devices that shares the same **Unified Virtual Address** space **(UVA)**. To make all the devices in such a multi-GPU system have the benefits of accessing the same specific memory block, it must be allocated with `cudaHostAlloc()`, and he flag `cudaHostAllocPortable` must be passed when the memory block is allocated.

When creating a system for multi GPU's with a portable memory, it makes a single address space available to all the devices sharing the UVA, and copying of data between the GPU's is performed with the `cudaMemcpy()` function. The parameter `kind` which passes the flag `cudaMemcpyKind` can then be set to `cudaMemcpyDefault`.

7.2.4 Synchronous Execution

Kernel execution and memory operations are launched and executed sequentially, one task at the time. The CPU controls the scheduling of the tasks, and the control is not returned from the device to the host thread before the device has finished executing it's task. Synchronous executions are blocking and no further tasks can be launched for execution until the last call is done.

In the CUDA context, a kernel's parallelism embodies the possibility to perform a task on many data-elements concurrently, the so-called thread parallelism, still being synchronous. As opposed to synchronous execution there are possibilities for asynchronous execution. This and other features for enhanced parallelism has been enabled with CUDA, and will be conveyed in later sections.

7.2.5 Asynchronous Concurrent Execution

In addition to the thread-parallelism, CUDA has establish the possibility for concurrent execution between host and device. The control is then given back to the CPU before the kernel or memory operation has has finished it's execution on the GPU.
This is facilitated through some asynchronous function calls. Copies between page-locked memory and device memory can be performed concurrently with kernel execution for devices that supports this behavior. It is possible to check for this capability by the **`asyncEngineCount`** in the device property, if it is set to a value larger than zero.
Concurrent kernel execution is possible for some devices with compute capability higher than 2.0. Up to 32 concurrent kernel launches are possible on devices of compute capability 3.5 or higher. The capability can be checked with a query to the **`concurrentKernels`** i device property. It is set to 1 for devices that supports this.

It is possible for the programmer to block asynchronous behavior by setting the environment variable **`CUDA_LAUNCH_BLOCKING`** to 1.

7.2.6 CUDA Streams

CUDA **Streams** are great for accelerating parallel applications, and manage the application's concurrency. A stream represents a queue of GPU operations, like kernel launches and memory operations, that are executed in a certain order. The commands in the stream are operated by the host thread. It is also possible to utilize more than one host thread, as will be explained later in the section on OpenMP threads.

Different streams can be executed concurrently. They execute their commands separate from each other, and may therefore be executed concurrently and out of order with respect to each other. A stream is created with the instruction **`cudaStreamCreate()`**, and destroyed with **`cudaStreamDestroy()`**. If kernel launches and memory operations are issued without an explicitly declared stream parameter, the tasks are issued to the default stream.

7.2.6.1 Streams and Events on a Multi-GPU application

In a multi-GPU application each GPU can handle multiple streams and multiple events. These streams and events are only valid within the scope where they are declared. The scope of a stream is determined by the device that was current at the time of the streams creation. CUDA streams are per device, and calls to a stream can only be issued when the device is current. This means that streams associated with one device, are not considered by other devices. If a task is issued to a stream that is not associated with the current device, the operation will fail. If the input-event and the input-stream to an operation are associated to different devices, the operation will fail. This can be monitored with the function **`cudaEventRecord()`**. The function returns **`cudaSuccess`** if the event succeeded. Otherwise it yields a **`cudaError`**. Any synchronization between streams has to be handled explicitly.

7.2.6.2 Synchronization

Streams can be synchronized with each other. There are some explicit synchronization calls that handles synchronization on different levels.

`cudaDeviceSynchronize()`
waits until all host threads have completed all their commands in all their streams.

`cudaStreamSynchronize()`
waits until all the preceding commands in a stream are completed. This function takes the specified stream as a parameter.

`cudaStreamQuery()`
provides the application with a way to know if all the foregoing tasks in a stream have completed.

7.2.7 CUDA Events

CUDA events are markers for synchronization that can:

- Yield a fine grained synchronization in a specified stream
- Time (asynchronous) tasks in streams
- Yield an inter stream synchronization, where one stream waits for an event in an other stream

The events are created with **`cudaEventCreate()`**,
and destroyed with **`cudaEventDestroy()`**.
Some of the different event functions are:

`cudaEventRecord()`
Sets a point for where it starts to record a given event.

`cudaStreamWaitEvent()`
this function delays all the tasks that are called for after this function call until the given event in the given stream has finished executing (from where it stared recording). It takes both a stream and an event as parameters. This synchronization does not stall the host, but only waits for the operations to finish within a stream and an event. The stream may be on an other device, and this function can be used to set a barrier until a event in an other stream on an other device is is done.

`cudaSynchronizeEvent()`
waits for an event to complete.

`cudaEventElapsedTime()`
is a function for timing. It takes two CUDA events as parameters, one for start time and one for stop time.

7.2.8 CUDA Featureas for a Multi-GPU System

A CPU may be connected to multiple GPU's. For a system with multiple devices, the host needs to be able to distinct the devices from each other. CUDA-enabled devices can be **enumerated** and **counted**. The following code extracts the number of a devices in the system.

Listing 7.1: Get Device Count with/in CUDA C

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
```

To operate on a specific device, the context for the device needs to be set. A host thread sets the device it operates on with a call to **`cudaSetDevice()`**. This function takes the device, an integer, as a parameter. The kernel launches and memory operations are then handled by that specific device.

7.2.8.1 Peer-to-Peer (P2P) Memory Access

In a multi GPU system, P2P comunication allows for a direct communication between the GPUs. In order to have the ability for a P2P memory communication between two devices, a **Unified Virtual Access space (UVA)** has to be available and utilized by the devices. The direct communication eliminates system memory allocation and copy overhead.
Direct communication between two GPUs can be established by **Direct Transfer** or **Direct Access**. (Direct Transfer will be addressed in the next section.)

On some multi GPU systems, a kernel executing on one device can dereference a pointer to the memory of an other device. This is called P2P memory access. Data is cached in the L2 of target GPU.

Direct Access
gives one GPU the possibility to load/read or store/write data-values directly from or onto the memory of an other GPU.

`cudaDeviceCanAccessPeer()`
is a function that checks if a system supports P2P between two devices when called. It requests if peer access is possible between a set of two devices, and returns `true` if so.

`cudaDeviceEnablePeerAccess()`
must be called to enable P2P direct access from a kernel on GPU to the memory of another GPU

7.2.8.2 Peer-to-Peer (P2P) Memory Copy

Data can be copied directly between the memories of two different devices, without the need to pass through the host, as it works transparently with UVA.

Direct Transfer
allows one GPU to copy data directly to or from an other GPU's memory.

Memory copies can be performed **synchronously** or **asynchronously**. For synchronous P2P memcpy functions CUDA offers:

`cudaMemcpyPeer()`

`cudaMemcpy3DPeer()`

For asynchronous behavior CUDA offers the functions:

`cudaMemcpyPeerAsync()`

`cudaMemcpy3DPeerAsync()`

`cudaMemcpyKind`
is used as a parameter describing the kind of copying that is being performed, whether it is from/to CPU/GPU. This parameter becomes unnecessary when a

Unified Virtual Address space (UVA) is available, so this "default" parameter is used instead

`cudaMemcpyDefault`.

7.2.8.3 Unified Virtual Address Space (UVA)

With UVA, only one address space is needed for all CPU and GPU memory. The application needs to be running as a 64-bit's process and the devices must be of compute capability 2.0 or higher. It simplifies the library functions and the **`cudaMemcpyDefault`** can be utilized instead of any of the other "kinds" of **`cudaMemcpyKind`**. To benefit from this feature, the host memory must be allocated through CUDA with **`cudaHostAlloc()`** and is automatically portable across all the devices using the UVA.

7.2.8.4 Peripheral Component Interconnect Express (PCIe)

The PCIe is a computer bus for maximum high throughput. This kind of bus is acquired for fast P2P communication between devices. To carry out P2P copy between devices, the memory must be allocated using UVA.

Chapter 8

Implementation of the Sediment Transport Model

This chapter will embroider different aspects regarding the implementations of the numerical model for dual-sediment transport, by [*Rivenæs* 1993]. On the path to the numerical solution, the equation set has been discretized and initial conditions and boundary conditions have been set before the software is written for a both serial and parallel implementations. The numerical aspects of the model was derived in Chapter 4. Recalling that the system for the sediment-transport model runs time-series over a set of partial differential equations. Each time step calculates the bathymetry of Lake Okeechobee in the variable h, and the fractions of the two sediments sand and silt as the values of s and $1-s$, all represented as 2D data-grids. The **Finite Difference scheme** called **Forward-Time, Central-Space (FTCS)** is used to approximate a solution to the PDE's.

The serial implementation runs on the host (CPU), and the parallel on the device (GPU) or multiple devices. For the serial implementation the ANSI C programming language has been used. The parallel programs have been developed for a SIMD/SIMT architecture on NVIDIA's CUDA architecture coded with the CUDA C extensions. Most of the CUDA C features that was presented in Chapter 7 have been employed, and will be elaborated in the context of the implementations and how they have been utilized. Parallel applications have been developed for both a single GPU, and for multiple GPU's. For multiple GPU's additional CUDA features have been explored. Also a set of multiple OpenMP-threads are spawned to reduce overhead on the CUDA-kernel launches. The use of OpenMP will also be presented here.

Prior to revealing details in the program code for of the model, an analysis for a parallel architecture is conveyed in this chapter.

8.1 Analysis of a Parallel Architecture for the Numerical Model

When considering an architecture for a parallel application, it is important to analyze the characteristics of the problem to be solved. It is important to look at what kind of an algorithm/system is to be implemented and the details that it evolves around.

8.1.1 The Task/Channel approach for a Parallel Architecture

As a tool for analyzing which parallel architecture to use in this specific case, elements from the **task/channel** model is used. See section 6.1.2. Although this tool ay be intended for a MIMD architecture, it has features that also a SIMD application can benefit from.

Reviewing details in the model, the following analysis has been the basis for the choice of a parallel architecture.

8.1.1.1 Partitioning

A system for the sediment-transport model does calculations on extremely large datasets. It holds multiple data-variables in 2D data-matrices with a data-item at each grid-point. Each data-item is updated with a few functions to perform the calculations, and this procedure is done for every time step. A few tasks/functions are performed at each time step. The tasks are performed on the same data-elements, first one task on all the elements in parallel, then the next task performs on all the elements in parallel and so on, one function after an other. Therefore the system does not yield **functional parallelism**, as would be the case if the different functions could be run in a parallel. This system does yield **data parallelism** since one task can be performed on multiple data-elements simultaneously.

With the SIMT architecture on a single GPU, the thread parallelism naturally leaves a decomposition of the data-domain, a so-called data-decomposition, into each thread handling a single data-element. Therefore a data-domain can contain only one data-element and be associated with one task at the time.

8.1.1.2 Partitioning for a Multi-GPU system

When the computer system contains multiple GPU's it is likely to yield additional parallelism. This application is still restricted to perform data-parallelism, and since it deals with very large data-sets it is natural to choose a decomposition over the data-domain. With a multi-GPU system it is possible to divide the data onto the different GPU's. The complete data-load is divided into N chunks, each to be calculated on one of the N GPU's in the multi-GPU system. In addition to maintaining the parallelism in each single GPU as described, where each thread performs tasks on a data-element, it is possible to benefit from enhanced acceleration when performing that same thread parallelism on multiple GPU's. Then each chunk of data on a the separate GPU's performs the same thread parallelism as a single GPU do. The data-load can be decomposed in a 1D, 2D or a 3D manner.

8.1.1.3 Communication

When data-sets are decomposed, tasks are performed separately on the different decomposed data-domains. A data-domain can contain one single data-element at a grid-point, or many data-elements can be grouped together. When the value of element $h_{j+1,i}$ in one domain is needed in an other domain to perform a task on element $h_{j,i}$, then the element $h_{j+1,i}$ must be communicated through a channel to the domain needing the data-element. If a data-domain holds only one data-element and needs to compute a stencil with four neighboring elements and each element belongs to four separate neighbors, then it needs to perform communication over four channels for accessing all neighboring elements needed in the computations. see figure

Considering the SIMT architecture provided on NVIDIA's GPU's the communication of data-values between threads on a single GPU does not have to be handled explicitly. NVIDIA has organized for memories to be shared between the threads, and with a high bandwidth access.

8.1.1.4 Communication for a Multi-GPU system

When the data is partitioned between the multiple GPU's, the data along the boundaries must be communicated to and from the the neighboring GPU. When calculations are performed on a data-domain at one time step, the data along the boundaries are needed by the neighbor for it's calculations at the next time step. Therefore the data-domains containing single data-elements have to be agglomerated into fewer and larger data-domains containing more data-elements.

To perform tasks on data-elements along the border to a neighbor of the decomposed data-domains, communication must be performed between the different domains. Each data-element along a border of the decomposition has to be communicated to the neighboring data-domain, but the data are communicated as a block of data-elements through one channel, rather than through one channel per data-element. With fewer neighbors there will be less communication and less overhead. The communication only has to be performed along the border of the domains, and larger data-domains will yield a reduction in the communication overhead.

8.1.1.5 Agglomerating and Mapping

It would be impossible to compute all tasks on all data-elements at the same time. One reason is that the values calculated in one time step is used to calculate the new values in the next time step. A second reason is that all the tasks/functions performed at each time step have to follow a certain consecutive order for the calculations to yield correct result. The tasks are listed as the functions in the code example 8.4. Although there are multiple tasks to be performed in this algorithm, they all have dependencies to one another and can not be performed simultaneously on a an architecture for Multiple-Instruction in parallel like the MIMD architecture. Since these functions have to be performed sequentially, one after an other, the system can be considered to benefit from an architecture of a "Single Instructions"-type. The multiple data elements in this finite difference application can be calculated on simultaneously, but by only one function at a the time. By this it can be concluded that this is a system with "Multiple-Data streams" that can be computed on simultaneously by one task at the time, and therefore benefiting from an architecture for Single-Instructions. This yields a SIMD architecture when recalling Flynn's taxonomy for parallel computer architectures, and SIMD is then the preferred choice when implementing the sediment-transport.

In this application akin of SIMD architecture, namely the SIMT architecture will be exploited, executing one data-element per thread, with multiple threads in parallel. With the stencil for the sediment-transport model, there will be a need to communicate the data-elements to and from four neighbors as the decomposition of the data-domain is made per data-element. This could easily result in a major overhead due to the all the communication of data-elements performed over channels, but the CUDA architecture omits this to happen. It could be the case where each data-element in each function at every time-step needed to be communicated. Luckily the CUDA architecture provided on the GPU's produced by NVIDIA employ the SIMT parallel architecture. CUDA features many options for parallelism and some have already been mentioned, like the parallelism performed with the multiple threads and the high memory bandwidth. The data access between the threads on a single GPU is optimized through the great bandwidth to the memory hierarchy, and by this it allows high-speed access and computing on neighboring data-elements. The extended number of memory interfaces (also called channels), is a hardware feature that the GPU's profit on. When computations are executed on a single GPU, the communication overhead is not to be considered between the threads as the multiple threads can share memory spaces and have extremely fast access to the same data.

Although the CUDA architecture performs fast on single threads, it has a feature where threads can be placed into blocks. Placing groups of threads onto thread-blocks increases the speed of the system. The reason is that the multiprocessor executes threads in groups of 32 parallel threads called warps. All threads in a warp start together at the the same program address. The size of the thread-blocks are optimally a multiple of the warp-size. On a single GPU the data-elements are agglomerated into such thread-blocks.

In the world of parallel computing this problem is considered to be **embarrassingly parallel**. It could also be called "perfectly parallel" or "pleasingly parallel". The calculations could be performed in parallel on all the grid points in the data-domain at one time step by one function, and have no dependency to other tasks nor dependencies to data that needs to be communicated during the calculations.

8.1.1.6 Agglomerating and Mapping for a Multi-GPU system

It has already been mentioned that in the multi-GPU system, the decomposition of the data-domain lets each GPU handle one chunk of data. To obtain a good **load-balance**, the total data-domain is decomposed into N (more or less) equally sized data-sets, one for each of the N GPUs. To reduce communication, a **1D decomposition along x** has been chosen.

8.2 General Details Regarding Both Serial and Parallel Implementations

Implementations of the systems have been developed with stencil computations of the coupled system of Partial Differential Equations (PDEs) with time series, according to the numerical model. Operations are executed on the sets of 2D grid-data and the values are updated according to the stencils by a series of arithmetic instructions. The application for the sediment-transport model is an Initial value problem, and has been programmed with the equations **??** for diffusion and 4.20 for convection in addition to the functions handling the boundaries. The spatial discretization of these equations requires on layer of ghost points around the boundaries of each (sub-)domain. The simulations are executed through discrete time steps, and all the data in the large 2D grids for h and s are updated at every time step.

Both the serial and the parallel systems are implemented in C, although the parallel system exploits the CUDA C extensions. CUDA C is basically C-programming with CUDA extensions to perform memory-handling and thread-operations on the CUDA parallel architecture.

8.2.1 Variables for Implementation

The numerical model represents how the bathymetry of the lake and the fractions of the two sediments, sand and silt, are distributed when the masses are coming down the River Kissimmee and into Lake Okeechobee. The lake is represented as a 2D domain, and the physical size of the lake is set to 62130 by 69500 meters. The change in bathymetry and the fractions of sand and silt over time, are simulated using so-called time series over the 2D-domain. To simulate the distribution of the sediments entering the lake from the river, the calculations for both **diffusion** and **convection** are performed on the discrete data for each grid-point in the 2D data-domain, at each time step. The numerical model with the discretized equations from chapter 4 are implemented, and equation 4.20 is used for diffusion and equation 4.23 is used for convection. In addition the updates of the boundaries are computed.

There are various variables used for the calculations at each grid-point and at each time step in the numerical model, as can be recalled from the variables listed in the table 4.1 in the chapter explaining the numerical model 4. Some of the variables are scalar constants used for all the grid points and all the time steps, and some are 2D-matrices with a discrete values for each grid-point in the 2D data-domain. The values for h, s, α and β are represented in such 2D-matrices. The matrices α and β are constant matrices, which means that their data-values remains the same for every time step without any updates.
The values in the 2D data-grids representing h and s are updated at each discrete grid point in space and for each time-step. These matrices defines the bathymetry of the lake, and the fraction of sand (and silt represented as $1 - s$) at a given point in time and space.

The variables α and β from the numerical schema derived in Chapter 4, cor-

responds to variables $alpha$ and $beta$ in the implemented code.
For the simulations some assisting variables are needed. As the new value for h are calculated at time step $l+1$, the old values calculated at time step l are used in the calculations. For this reason, two values are needed for h. By this the matrix for the new time step at $l+1$, representing h^{l+1} in the mathematical model is put into the variable *h_new*, and the variable for the previous time step h^l is kept as h for simplicity when reading the program code on the right hand side.
In addition to this a third variable for h has to be available at the end of each time step, when the the pointer to the matrices needs to we swapped for the next iteration. This variable is called h_tmp. The same set of variables are declared for s with *s_new* and s_tmp.

Variable Name	**Explanation**
h	The "old" values of h, calculated at the previous time step
s	The "old" values of s, calculated at the previous time step
h_new	Newly calculated values of h
s_new	Newly calculated values of s
h_tmp	Temporary grid for h during swapping of pointers
s_tmp	Temporary grid for s during swapping of pointers
dx	Spatial step in x-direction (Equivalent to Δx in the numerical scheme)
dy	Spatial step in y-direction (Equivalent to Δy in the numerical scheme)
dt	Time step (Equivalent to Δt in the numerical scheme)
t	Value incremented by dt at each time step
dx_R	$= 1/dx$ (Reciprocal variable)
dy_R	$= 1/dy$ (Reciprocal variable)
$dx2\_R$	$= 1/(dx * dx)$ (Reciprocal variable)
$dy2\_R$	$= 1/(dy * dy)$ (Reciprocal variable)
as_center	$= alpha[j][i] * s[j][i]$
bs_center	$= beta[j][i] * s[j][i]$
M	Number of elements in x-direction
N	Number of elements in y-direction
$M2$	$= M + 2$ (Number of elements in x-direction incl. 2 ghost points)
$N2$	$= N + 2$ (Number of elements in y-direction incl. 2 ghost points)
num_gpus	Number of GPUs
N_dev	$= N/num\_gpus$ (Num elem in y-dir on each GPU after domain decomp)
$N2\_dev$	$= N\_dev + 2$ (Num elem in y-dir on each GPU incl. 2 ghost points)
$nStreams$	Number of CUDA Streams
$nEvents$	Number of CUDA Events
$nThreads$	Number of OpenMP threads
tr	Number of threads on a thread block
bl	Number of blocks in a grid
$influx$	Influx percentage of sand and mud together (set to 50% at the start)
$influx\_start$	Start position in the grid for the influx
$influx\_stop$	Stop position in the grid for the influx

Table 8.1: Additional Variables for the Programmed Code

Other variables are also introduced to enhance computational performance. The reciprocal variables are set so the stencil computations can avoid as many repeating divisions as possible which would reduce the speed. Also the calculations of $as\_center = alpha[j][i] * s[j][i]$ which are repeated several times in a stencil is put in a separate variable to enhance speedup. The same is done with $bs\_center = beta[j][i] * [j]s[i]$. Layer thickness A is set to $1m$. The variables added for the implementation to the ones that are already presented with the numerical model, are displayed with explanations in Table 8.1.

8.2.1.1 Spatial and Time Intervals - dx, dy and dt

The spatial intervals *dx* and *dy* have been calculated based on the physical size of the domain from Lake Okeechobee's seismic measurements. The size of *dx* is based on the physical length in *x*-direction divided on the number of intervals in *x*-direction, and likewise for *dy* which is based on the physical length in *y*-direction divided on the number of intervals in *y*-direction. How these variables in addition to the *dt* are set, can be seen in the code Listings 8.1.

Listing 8.1: Setting the variables dx, dy and dt

```c
#define X_LEN 62130 //Length of the spatial domain in x-direction
#define Y_LEN 69500 //Length of the spatial domain in y-direction

#define imin(a,b) (a<b ? a:b) //Inline function - returns the smallest of two values
#define imax(a,b) (a>b ? a:b) //Inline function - returns the largest of two values

dx = X_LEN/(M+2-1.); //The size of the the spatial intervals in x-direction (spacing between the x_i and x_(i+1))
dy = Y_LEN/(N+2-1.); //The size of the the spatial intervals in y-direction (spacing between the y_j and y_(j+1))

/* Assert that dt is small enough for the CFL Stability Chriterion
 * Find the smallest dt according to the chriterion:
 * dt <= (dx^2 * dy^2) / (2 * C_max *(dx^2 + dy^2))
 * Call to inline function to return the smallest value (dt or the stability chriterion for dt) */
//dt = imin( dt, ( ((dx*dx)*(dy*dy)) / (2*coeff_max*((dx*dx)+(dy*dy))) ) ); // From hpl's book + article(inkl. coeffs)
setMaxDt(&h_alpha_storage, &h_beta_storage, &dt, dx, dy, M, N);


/* Set the largest possible value for dt, to obtain stability
 * With a fully explicit scheme it is limited by the values of alpha and beta
 */
void setMaxDt(double **h_alpha_storage, double **h_beta_storage, double *dt, double dx, double dy, size_t M, size_t N) {
        int i;
        double alpha_max = (*h_alpha_storage)[0];
        double beta_max  = (*h_beta_storage)[0];
        for (i=1; i<(M*N); i++) {
                alpha_max = imax(alpha_max, (*h_alpha_storage)[i]);
                beta_max  = imax(beta_max, (*h_beta_storage)[i]);
        }
        //*dt = (dx*dy) / imax(alpha_max, beta_max);
        *dt = 0.01;
}
```

The time interval *dt* was first set according to the **Courant-Friedrichs-Levy (CFL) Stability Criterion**, but was later set to 0.01 year for both diffusion and convection according to the suggestions in [*Clark*, *Wei*, and *Cai* 2010].

8.2.1.2 Ghost Points

The grid of the 2D data has been added 1 row of ghost points all around on the boundary as presented in figure 8.1. The size of the grid without the ghost points is $M \times N$, and with the ghost points $M2 \times N2$.

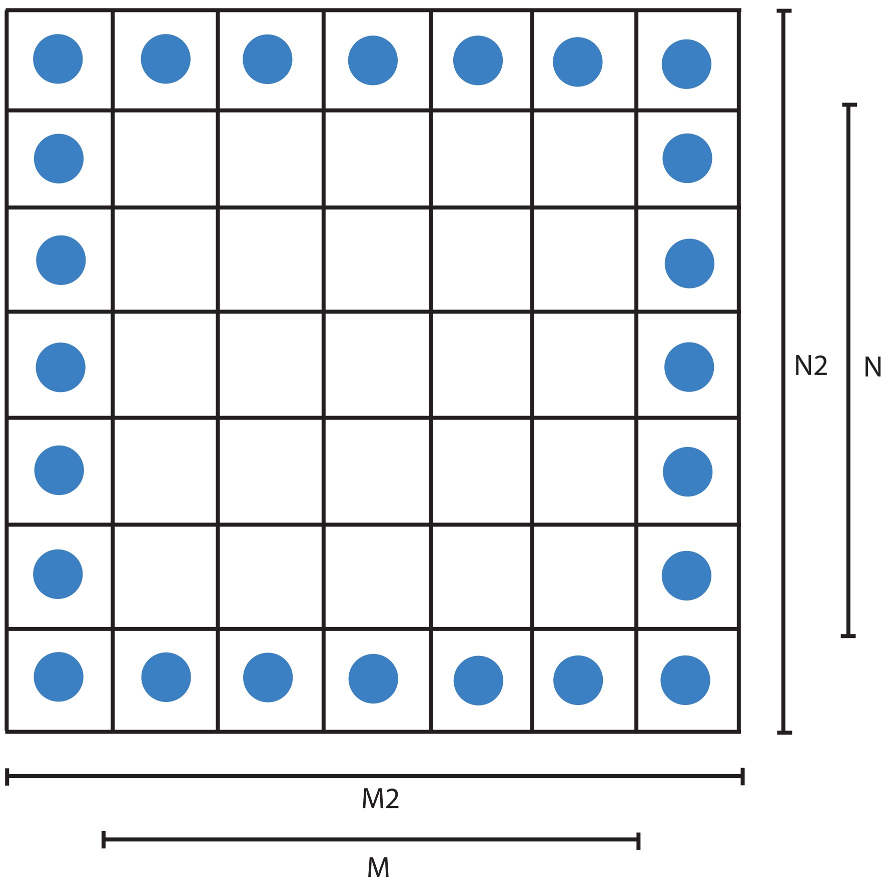

Figure 8.1: One Row of Ghost Points Around the Boundary

8.2.1.3 Influx

Where the River Kissimmee enters Lake Okeechobee, there is an **`influx`** of sand and mud, which together is set to be 50% of the total mass. Of this mass there is a total of 20% sand and 80% mud. The variables **`influx_start`** and **`influx_stop`** indicates the position along the border where the influx starts and stops. The influx is along x at $j = 0$, and the variables are used to set the Dirichlet- and Inhomogeneous Neumann Boundary Condition. An overview of the variables can be viewed in the code listing 8.2.

Listing 8.2: Variables for the influx from River Kissimmee

```
double influx = 50;                          //Influx-percentage
double f_s=influx*0.2, f_m=influx*0.8;  //Influx-value of sand (20%) and mud (80%)
int influx_start=755, influx_stop=809;  //Grid Points for influx- start and stop
```

8.2.2 Seismic Data Read Into The System With The NetCDF-Library

The numerical model implemented for the sediment transport model contains numerous of variables as displayed. The variables need values, and some of the variables regarding in the sediments-transport of Lake Okeechobee, have their data-values at the simulation's start point coming from seismic measurements. The variables that are read into the system from seismic surveys [*Gtopo30* 1996] are: h, s, α and β.

These data exists in a **Network Common Data Form (NetCDF)** [*Unidata* 2015] file format, as a **.nc** file, a machine-independent data format for array-oriented scientific data. These datasets are first opened and read into the system with the **NetCDF-library** for the C-language, and then stored into the memory allocated on the CPU. The data in the .nc files are read into the system with the code in the code listing 8.3. The size of the grid comes from variables read from the file, in addition to the data for the large grids.

The initial values in the s-grid , $s(i, j, 0)$, are set to 0.5, as the initial sand-silt volume fraction are assumed to be at 50% according to [*Clark*, *Wei*, and *Cai* 2010]..

Listing 8.3: Reading files of the format .nc with the NetCDF-library

```
#include <netcdf.h>
        ...

        double *alpha;
        int retval, ncid, varid, xid, yid;
        size_t xlen, ylen;

        if ((retval = nc_open("alpha.nc [2]", NC_NOWRITE, &ncid)))
                ERR(retval);
        if ((retval = nc_inq_dimid(ncid,"x",&xid)))
                ERR(retval);
        if ((retval = nc_inq_dimlen(ncid,xid,&xlen)))
                ERR(retval);
        if ((retval = nc_inq_dimid(ncid,"y",&yid)))
                ERR(retval);
        if ((retval = nc_inq_dimlen(ncid,yid,&ylen)))
                ERR(retval);
```

```

    if ( (alpha = (double*)malloc(xlen*ylen*sizeof(double))) == NULL){
            printf("Failed to allocate the 2D array for alpha.n");
            return -1;
    }

    if ((retval = nc_inq_varid(ncid, "z", &varid)))
            ERR(retval);
    if ((retval = nc_get_var_double(ncid, varid, alpha)))
            ERR(retval);
    if ((retval = nc_close(ncid)))
            ERR(retval);
```

8.2.3 Time Series

This system simulates over the 2D data domain, although represented as 1D coalesced, in time series. In addition to performing the updates for **Diffusion** on h and **Convection** on s, there is also a need to update the boundaries for both h and s at each time step. The boundary conditions for h are updated after the diffusion-step, first by a **Homogeneous Neumann BC**, and then by a **Inhomogeneous Neumann BC**. The BCs need to be updated on h before the convection can be calculated on s, as the newly calculated values for h are used in the calculation of s. The BCs on s are updated after the convection-step, first updated by the **Homogeneous Neumann BC**, before it takes a **Dirichlet BC**. Since the Homogeneous Neumann BC is 0, it means copying the value of the neighboring index on the inner side of the data-domain to the boundary-points. Inhomogeneous Neumann BC and Dirichlet BC are set along the border, only where the Kissimmee River gives an influx. At each time step t is updated with an incrementation of dt. Pseudo-code for the functions in the loop of the time series and the swapping of the matrix-pointers can be seen in the code listing 8.4.

Listing 8.4: Pseudo Code of the Main Function Loop

```
while (t<max_t) {
  t+=dt;

  //compute diffusion and update boundaries on h
  diffusion();
  update_boundary_neumann_homogeneous();
  update_boundary_neumann_inhomogeneous();

  //compute convection and update boundaries on s
  convection();
  update_boundary_neumann_homogeneous();
  update_boundary_dirichlet();

  //swap matrix-pointers
  h_tmp = h_old;
  h_old = h_new;
  h_new = h_tmp;

  s_tmp = s_old;
  s_old = s_new;
  s_new = s_tmp;
}
```

8.2.4 Coalesced Memory

One major feature is added to reduce the time spent to access the data at run-time. The optimization lies in the structuring of the data. Instead of using a 2D data-structure for the 2D-matrices, this application benefits from a 1D **coalesced memory** on both the CPU and the GPU. Using a coalesced 1D data-structure is an advantage when data is to be accessed, as they are read from the same memory address, and not from multiple address spaces. This improves the performance of the application as less time is used on data accessing.

8.3 Serial Implementation with ANSI C

The implementation of the sequential system is done pretty straight forward using the C programming language with the built-in libraries that lies within the ANSI-C standard [REF]. The stencil computations are performed using loops over the time series and the 2D spatial data domain.

8.3.1 Allocation of Coalesced Memory for the Serial Code

Some of the host data that are handled by the host alone has been allocated with **`malloc()`**. For the 2D-grids, 1D coalesced data-structures has been allocated for fast access of the data. The feature of 2D-pointers are created for the 2D data-sets. The 2D-pointers helps indicate which row the data-element belongs to, and makes the coded stencils more readable. This way to create 2D-indexing helps on the comprehension of the code and what lies underneath the numerics, for anybody reading or working with the computer program. The allocations of the coalesced memory and allocations of the corresponding pointer to the rows created on the CPU, together with how to access the data, is presented in the code listing 8.5.

Listing 8.5: 1D Coalesced Memory for Fast Access on the CPU.

```
/* Allocating the 1D coaleasced memory storage for a 2D-dataset */
if (( u_storage = (double *) malloc( sizeof(double)*M*N ) ) == NULL ) {
    fprintf(stderr, "Out of memory\n");
    exit(0);
 }

/* Allocate the pointer to access data in a 2D manner */
 if (( u = (double **) malloc( sizeof(double)*N ) ) == NULL ) {
    fprintf(stderr, "Out of memory\n");
    exit(0);
 }

/* Set the 2D-pointer to point correctly into the 1D coaleasced
 * memory storage where each new line begins */
for (i=0; i<N; i++) {
    u[i] = &(u_storage[i*M]);
}

/* Accessing data-elements using 2D-indexing */
for(j=0; j<N; j++) {
  for(i=0; i<M; i++) {
    ...
    u[j][i] = ...;
    ...
  }
}
```

8.4 Parallelization of Serial Code using the CUDA C Extension

The parallelization carried out by [Clark2010] was applied using Message Passing Interface (MPI). With MPI their code was executed on CPUs while this thesis has investigated the benefit from the extraordinary processing power of NVIDIA's GPUs.

* In addition to aim to answer this thesis' questions stated in 2.1, this thesis also intend to investigate the following problems.

* The first aim was to see if the CUDA application could outperform the MPI application. If so; in addition to yield the benefits of enhanced speedup, it is also great in the aspect of knowing that a few GPUs can be less costly and more energy saving than a CPU cluster.

* Another aim was to see if the features presented by [Sourouri et al] would enhance the the speedup utterly. They have tested some techniques for enhancements for parallelizing code on GPUs using CUDA Streams in combination with

OpenMP threads. The question was if their techniques applied on the sediment-transport application would yield an utterly enhanced speedup.

All computer code has been written from scratch, including the serial code for the CPU, but the MPI code has been viewed in the process of working with this thesis. This thesis has extend the serial code to run in parallel on one or multiple GPUs rather than on a CPU cluster, using the CUDA C Extension for the CUDA Parallel Architecture. The CUDA features mentioned in chapter 7 has been deployed in the procedure, and miscellaneous optimization techniques has been tested out. Here is an enumerated list of the implementations with descriptions and features that has been examined in this thesis:

1. Parallelizing with use of CUDA kernels on 1 GPU
2. Coupling multiple GPU's using Synchronous computations and communications between GPUs via host.
3. Coupling multiple GPU's using Synchronous computations and P2P communications between GPUs over the PCI Express bus.
4. Coupling multiple GPU's using Asynchronous computations and communications between GPUs via host, with CUDA Streams for concurrency.
5. Coupling multiple GPU's using Asynchronous computations and P2P communications between GPUs over the PCI Express bus, with CUDA Streams for concurrency.
6. Coupling multiple GPU's using Asynchronous computations and P2P communications between GPUs over the PCI Express bus, with CUDA Streams for concurrency, and in addition create a set of OMP-threads for each GPU, and let one OMP-thread on the CPU launch one CUDA-kernel, and an other OMP-thread handle asynchronous communication. With this, observations of additional speedup can be expected, due to a reduction of kernel-launch overhead.

Table 8.2: Implementations with Various Features for Parallel Architecture

8.4.1 Allocation and Copying of Coalesced Memory for the Parallel Code

The device memory has been allocated using the function **`cudaMallocPitch()`**. This is a function, which allocates a linear memory that is at least *width* (in bytes)

$\times height$ in size. This function will make sure that the data are padded appropriately to meet the alignment requirements for coalescing the data. This function is recommended by [NVIDIA:CUDA] for best performance when threads in a warp are accessing the addressed row or performing copies with 2D arrays. The length of rows in the 2D array should then be a multiple of the warp size (32). as these data are fetched runtime in the size of a warp, and **`cudaMallocPitch()`** therefore pads the row in the memory accordingly. The function returns a value **`pitch`**, witch is the length of the rows including the padding, and it is measured in the size of bytes. The pitched data must be handled correctly so the right data are accessed right. When copying data between host and device, and the device-array is padded, the function **`cudaMemcpy2D()`** can be used to handle the copying correctly. A small example of the allocation with **`cudaMallocPitch()`** and copying of data from host to a device can be seen in code listing 8.6.

Listing 8.6: Allocation of Coalesced Memory and Copy between Host and Device.

```
double *h, *s;   //grid-data/matrices
size_t pitch;    //size of paddded grid

/* Allocate liear memory for device-data (1D-vectors) */
HANDLE_ERROR( cudaMallocPitch(&h_dev, &pitch, sizeof(double)*M2, N2) );
HANDLE_ERROR( cudaMallocPitch(&s_dev, &pitch, sizeof(double)*M2, N2) );

/* Copy vetors FROM HOST-memory TO DEVICE-memory */
HANDLE_ERROR( cudaMemcpy2D(h_dev, pitch, h_host, sizeof(double)*M2,
                           sizeof(double)*M2, N2, cudaMemcpyHostToDevice) );
HANDLE_ERROR( cudaMemcpy2D(s_dev, pitch, s_host, sizeof(double)*M2,
                           sizeof(double)*M2, N2, cudaMemcpyHostToDevice) );
```

8.4.2 Threads and Blocks in a Grid

When computing on the GPU and exploiting the parallelism of a CUDA-kernel, it is necessary to decide for the number of threads on a thread-block, **`tr`**, and the number of thread-blocks in a grid, **`bl`**. Both these sizes are in 2D for our 2D data, and the variables **`tr`** and **`bl`** are therefore declared of the type **`dim3`**. The computer-code for this can be seen in the code listing 8.7. The code has one line commented out which is setting the size of the grid for an arbitrary number of GPUs, and the variable N_dev is the size of the decomposed domain. The two last lines for the boundaries launches fewer threads as it handles less data. The program optimizes the speed by only launching the necessary number of threads (in a multiple of warps), and not the size of a full grid.

Listing 8.7: Threads on a Block and Blocks in a Grid.

```
/* Number of threads on a thread-block in (x- and y direction) */
int num_x_threads_per_block = 16, num_y_threads_per_block = 4;
dim3 tr(num_x_threads_per_block , num_y_threads_per_block);

/* Number of thread-blocks in a grid */
dim3 bl( (M+tr.x+1)/tr.x, (N+tr.y+1)/tr.y );          /*** FOR 1 GPU   ***********/
//dim3 bl( (M+tr.x+1)/tr.x, (N_dev+tr.y+1)/tr.y );  /*** FOR MULTIPPLE GPU'S ***/
dim3 bl_BC_HomNeumann(16, 4); /*** For Homogeneous Neumann BC ***/
dim3 bl_BC_influx(1, 1);      /*** For BC at the influx ********/
```

8.4.3 Application for a Single GPU

The first parallel application (1), listed in table 8.2 is described here. The application performing on a (single) GPU has some of the same features as the serial code. Instead of the C-functions, the CUDA-kernels are called from a loop handling the time series. It updates diffusion and convection on all the grid points except on the boundaries. Update of the boundaries are performed in separate kernels. The loop handling the kernel calls for 1 GPU can be seen in the code Listing 8.8. Note that for the update of the boundaries, the kernels are called for a smaller set of threads than for the diffusion or convection to reduce time, as mentioned in section 8.4.2.

Listing 8.8: Calling the CUDA-Kernels in Time Series on 1 GPU.

```
HANDLE_ERROR( cudaSetDevice(device[0]) );
while (t<t_max) {
        t += dt;

        /* Compute Diffusion on H */
        diffusion_Solve_H_GPU<<<bl, tr>>>(h, s, h_new, alpha, beta, pitch, dx, dy
            , dt, c_s, c_m, M2, N2);

        /* Update Boundary Conditions on H */
        updateBC_Neuman_Homogeneous_H_GPU<<<bl_BC_HomNeumann, tr>>>(h_new, pitch,
             dx, dy, M2, N2);
        updateBC_Neuman_Inhomogeneous_H_GPU<<<bl_BC_influx, tr>>>(h, h_new, alpha
            , beta, pitch, influx_start, influx_stop, f_s, f_m, dy);

        /* Compute Convection on S */
        convection_Solve_S_GPU<<<bl, tr>>>(h, s, h_new, s_new, alpha, pitch, dx,
            dy, dt, c_s, c_m, A, M2, N2);

        /* Update Boundary Conditions on S */
        updateBC_Neuman_Homogeneous_S_GPU<<<bl_BC_HomNeumann, tr>>>(s_new, pitch,
             dx, dy, M2, N2);
        updateBC_Dirichlet_S_GPU<<<bl_BC_influx, tr>>>(s_new, alpha, beta, pitch,
             influx_start, influx_stop, f_s, f_m, dy);

        /* Swap array-pointers before the next timestep */
        h_tmp  = h;
        h      = h_new;
        h_new  = h_tmp;

        s_tmp  = s;
        s      = s_new;
        s_new  = s_tmp;
}
```

8.4.4 Applications for Multiple GPUs

The various applications for multi GPUs that are described in 8.2, are presented in this section. Different features that has been investigated regarding the implementations, is described.
When programming for multiple GPUs there are several possibilities regarding how to structure the program, various ways to communicate data between the devices, and different features can be employed, all with one major objective; to enhance the performance of the parallel system.

When dealing with multiple devices it's important to set the right context for the kernel-calls or memory handling. Since each device handles it's own memory, it is crucial that operations are performed on the correct set of data allocated on a given device. Each device is given a device-id, and the right context is set for each device with the function **`cudaSetDevice()`** as displayed in the code listing 8.9.

Listing 8.9: Sets The Device-ID, and the Current Device's Context.

```
int d;  //iterator over GPUs
int num_gpus; //number of GPUs returned by the intrinsic function

/* Check for the number of available devices */
HANDLE_ERROR( cudaGetDeviceCount( &num_gpus ));
if (num_gpus < 2) {
  printf("We need at least two compute 1.0 or greater devices, but only found %d\
      n", num_gpus);
}

/* Set device-ids for each GPU */
int device[num_gpus];
for (d=0; d<num_gpus; d++) {
  device[d] = d;
  HANDLE_ERROR( cudaSetDevice(device[d]) );
}
```

8.4.4.1 Data Decomposition

When the computations are performed on multiple devices, the data must be decomposed into smaller data domains, one for each of the multiple devices, and memory must be allocated on each device.
For the sediment transport application the data domain has been decomposed in only 1 direction to reduce the number of neighbors, and by that reducing the number of communication channels. Establishing the communication is a time consuming operation, so in order to minimize communication overhead, a 1D decomposition is preferred. The data domain is decomposed along x in equally sized data blocks, and placed on the number of available GPUs, or the number of GPUs that is set for the simulation (*num_gpus*).

The variable N being the size of the non-decomposed domain in y-direction, has then been discarded for the decomposed size of N. The decomposed size of N is set in the variable *N_dev* and equals the number of N divided by the number of GPUs (*num_gpus*). The handling of ghost-point for the decomposed domains are done equally for every GPU. The decomposed domain including one row of ghost points around the boundary, has like the variable N (for non-decomposed), been added 2 rows in y-direction, one on each side of the decomposed domain. The decomposed size including the ghost points is then given in the variable *N2_dev* which equals the value of $N\_dev + 2$. An illustration of the decomposed data-domain with boundary-points and ghost-points spread on the multiple GPUs is unveiled in figure 8.2.

Since the decomposition is performed along x, the domain is not divided in the x-direction. Therefore the values for the domain size in x-direction remain the same, namely N and $N2$ which respectively represent the grid size without and with ghost-points.

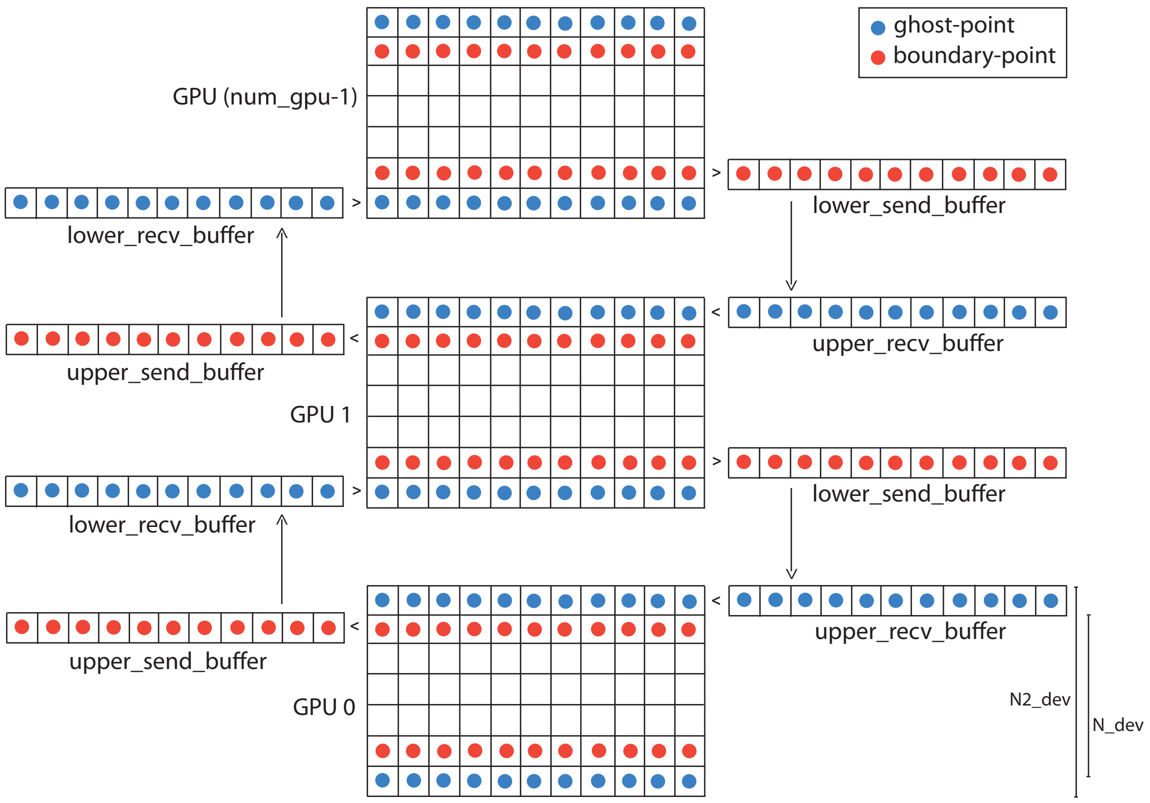

Figure 8.2: Domain Decomposed on GPUs

The data communicated between neighboring devices are copied into "tranfer-buffers" for sending and receiving data. The newly calculated boundary points along x in one domain are transferred to the ghost points in the neighboring domain. The upper boundary on GPU 0, being the $upper\_send\_buffer$ is transferred to the ghost points on GPU 1, being the upper neighbor, via $lower\_recv\_buffer$. This is performed like vise in the opposite direction, where GPU 1 is sending it's lower boundary in $lower\_send\_buffer$ to GPU 0's upper ghost points in $upper\_recv\_buffer$. This is applied for all neighbors on all the working GPU's. Fgure 8.2 also shows the transferred data buffers.

The various implementations handles this data transfer in different ways, as explained in later sections in this chapter.

When handling multiple GPUs, the application loops over all the devices, sets the context of the device, and the variables are declared and allocated on that specified GPU. The declaration and allocation of the decomposed data sets for each device can be seen in listings 8.10

Listing 8.10: Allocating Device Memory for the Decomposed Data-Domains on All the Devices.

```
    /*** Device data ***/
    size_t N_dev = N/num_gpus; //height of data-domain after domain decomposition
    size_t N2_dev = N_dev+2;   //height of data-domain including ghost-points
    size_t pitch_byt; /* pitch size in number of bytes = num_elements * sizeof(
        double) */

    double *h_dev[num_gpus], *h_new_dev[num_gpus], *h_tmp_dev[num_gpus]; /*
        Coaleased 1D storage */
    double *s_dev[num_gpus], *s_new_dev[num_gpus], *s_tmp_dev[num_gpus];
    double *alpha_dev[num_gpus], *beta_dev[num_gpus];

    double *h_first_row_send[num_gpus], *h_first_row_recv[num_gpus];
    double *s_first_row_send[num_gpus], *s_first_row_recv[num_gpus];
    double *h_last_row_send[num_gpus],  *h_last_row_recv[num_gpus];
    double *s_last_row_send[num_gpus],  *s_last_row_recv[num_gpus];

    for (d=0; d<num_gpus; d++) {
      HANDLE_ERROR( cudaSetDevice(device[d]) );

      /* Allocate liear memory for device-data(1D-vectors) on GPU's for the new
          data to be computed */
      /* padded/pitched */
      HANDLE_ERROR( cudaMallocPitch(&h_new_dev[d], &pitch_byt, sizeof(double)*M2,
            N2_dev) );
      HANDLE_ERROR( cudaMallocPitch(&s_new_dev[d], &pitch_byt, sizeof(double)*M2,
            N2_dev) );
      HANDLE_ERROR( cudaMallocPitch(&h_tmp_dev[d], &pitch_byt, sizeof(double)*M2,
            N2_dev) );
      HANDLE_ERROR( cudaMallocPitch(&s_tmp_dev[d], &pitch_byt, sizeof(double)*M2,
            N2_dev) );

      /* Allocate padded memory for device-data to be copied from the existing
          cpu/host-arrays */
      HANDLE_ERROR( cudaMallocPitch(&h_dev[d],     &pitch_byt, sizeof(double)*M2,
            N2_dev) );
      HANDLE_ERROR( cudaMallocPitch(&s_dev[d],     &pitch_byt, sizeof(double)*M2,
            N2_dev) );
      HANDLE_ERROR( cudaMallocPitch(&alpha_dev[d], &pitch_byt, sizeof(double)*M2,
            N2_dev) );
      HANDLE_ERROR( cudaMallocPitch(&beta_dev[d],  &pitch_byt, sizeof(double)*M2,
            N2_dev) );

      /* Allocate 1D-arrays for transferring data between this GPU and the
          previous GPU, and this GPU and the next GPU */
      HANDLE_ERROR( cudaMallocPitch(&h_first_row_send[d], &pitch_byt, sizeof(
          double)*M2, 1) );
      HANDLE_ERROR( cudaMallocPitch(&h_first_row_recv[d], &pitch_byt, sizeof(
          double)*M2, 1) );
      HANDLE_ERROR( cudaMallocPitch(&h_last_row_send[d],  &pitch_byt, sizeof(
          double)*M2, 1) );
      HANDLE_ERROR( cudaMallocPitch(&h_last_row_recv[d],  &pitch_byt, sizeof(
            double)*M2, 1) );

        HANDLE_ERROR( cudaMallocPitch(&s_first_row_send[d], &pitch_byt, sizeof(
            double)*M2, 1) );
        HANDLE_ERROR( cudaMallocPitch(&s_first_row_recv[d], &pitch_byt, sizeof(
            double)*M2, 1) );
        HANDLE_ERROR( cudaMallocPitch(&s_last_row_send[d],  &pitch_byt, sizeof(
            double)*M2, 1) );
        HANDLE_ERROR( cudaMallocPitch(&s_last_row_recv[d],  &pitch_byt, sizeof(
            double)*M2, 1) );

        /* Copy vetors FROM HOST-memory TO DEVICE-memory */
        HANDLE_ERROR( cudaMemcpy2D(h_dev[d],      pitch_byt, h_old_storage_host[d],
            sizeof(double)*M2, sizeof(double)*M2, N2_dev, cudaMemcpyDefault) );
        HANDLE_ERROR( cudaMemcpy2D(s_dev[d],      pitch_byt, s_old_storage_host[d],
            sizeof(double)*M2, sizeof(double)*M2, N2_dev, cudaMemcpyDefault) );
        HANDLE_ERROR( cudaMemcpy2D(alpha_dev[d], pitch_byt, alpha_storage_host[d],
            sizeof(double)*M2, sizeof(double)*M2, N2_dev, cudaMemcpyDefault) );
        HANDLE_ERROR( cudaMemcpy2D(beta_dev[d],  pitch_byt, beta_storage_host[d],
            sizeof(double)*M2, sizeof(double)*M2, N2_dev, cudaMemcpyDefault) );
    }
```

8.4.4.2 Pinned Memory on Host

For some of the host data that has to be copied or mapped from host to device during runtime, in addition to being transferred between multiple devices using P2P and/or asynchronous behavior, the CUDA-function **`cudaHostAlloc()`** has been used to allocate data on host, and **`kind`** is set to **`cudaHostAllocPortable`**. This ensures for page-locked or so-called pinned memory on the host. The pinned memory space is shared between the host and device or between multiple devices, and this memory has pointers for host and device. An example on how pinned host memory is allocated for the decomposed data, in addition to allocations of 2D-pointers, can be seen in 8.11.

Listing 8.11: Allocation of Pinned/Page-Locked Memory Space)

```
/* Decomposed host data (divided by number of gpu's) */
double *h_old_storage_host[num_gpus], *h_new_storage_host[num_gpus];
double *s_old_storage_host[num_gpus], *s_new_storage_host[num_gpus];
double *alpha_storage_host[num_gpus], *beta_storage_host[num_gpus];
double **h_old_ptr_host[num_gpus], **h_new_ptr_host[num_gpus];
double **s_old_ptr_host[num_gpus], **s_new_ptr_host[num_gpus];
double **alpha_ptr_host[num_gpus], **beta_ptr_host[num_gpus];

int start_row;  // The row in the "global grid"(holding all the partial grids)
    where a partial grid starts

for (d=0; d<num_gpus; d++) {

        /* Allocate 1D coalesced storage of data */
        cudaHostAlloc(&h_old_storage_host[d], sizeof(double*)*M2*N2_dev,
            cudaHostAllocPortable);
        cudaHostAlloc(&h_new_storage_host[d], sizeof(double*)*M2*N2_dev,
            cudaHostAllocPortable);
        cudaHostAlloc(&s_old_storage_host[d], sizeof(double*)*M2*N2_dev,
            cudaHostAllocPortable);
        cudaHostAlloc(&s_new_storage_host[d], sizeof(double*)*M2*N2_dev,
            cudaHostAllocPortable);
        cudaHostAlloc(&alpha_storage_host[d], sizeof(double*)*M2*N2_dev,
            cudaHostAllocPortable);
        cudaHostAlloc(&beta_storage_host[d],  sizeof(double*)*M2*N2_dev,
            cudaHostAllocPortable);

        /* Allocate 2D poiters for each row */
        cudaHostAlloc(&h_old_ptr_host[d], sizeof(double*)*M2*N2_dev,
            cudaHostAllocPortable);
        cudaHostAlloc(&h_new_ptr_host[d], sizeof(double*)*M2*N2_dev,
            cudaHostAllocPortable);
        cudaHostAlloc(&s_old_ptr_host[d], sizeof(double*)*M2*N2_dev,
            cudaHostAllocPortable);
        cudaHostAlloc(&s_new_ptr_host[d], sizeof(double*)*M2*N2_dev,
            cudaHostAllocPortable);
        cudaHostAlloc(&alpha_ptr_host[d], sizeof(double*)*M2*N2_dev,
            cudaHostAllocPortable);
        cudaHostAlloc(&beta_ptr_host[d],  sizeof(double*)*M2*N2_dev,
            cudaHostAllocPortable);

        /* Set the row-pointer */
        for (j=0; j<N2_dev; j++ ) {
                h_old_ptr_host[d][j] = &(h_old_storage_host[d][j*M2]);
                h_new_ptr_host[d][j] = &(h_new_storage_host[d][j*M2]);
                s_old_ptr_host[d][j] = &(s_old_storage_host[d][j*M2]);
                s_new_ptr_host[d][j] = &(s_new_storage_host[d][j*M2]);
                alpha_ptr_host[d][j] = &(alpha_storage_host[d][j*M2]);
                beta_ptr_host[d][j]  = &(beta_storage_host[d][j*M2]);
        }

        /* Initialize the decomposed matrices */
        start_row=d*N_dev;
        for (j=0; j<N2_dev; j++) {
                for (i=0; i<M2; i++) {
                        h_old_ptr_host[d][j][i] = h_old[start_row+j][i];
                        s_old_ptr_host[d][j][i] = s_old[start_row+j][i];
                        alpha_ptr_host[d][j][i] = alpha[start_row+j][i];
                        beta_ptr_host[d][j][i]  = beta[start_row+j][i];
                }
        }
}
```

8.4.4.3 Implementation 2.

The multi GPU system described as implementation (2) in 8.2 is coupling multiple GPU's using synchronous computations and communications between the devices. This means that communication does not overlap with computations. In this app the communication between devices is performed via host, using the function **`cudaMemcpy()`** with the **`kind`** set to **`cudaMemcpyDeviceToHost`** moving data from device memory to host memory. When all the data arrays are copied to host, swapping of pointers to the arrays are done before the data are copied to the neighboring device with the same function, but in the opposite direction, and the

kind parameter is set to **cudaMemcpyHostToDevice**. A system according to this description is displayed in listing 8.12.

Listing 8.12: Multi GPU Synchronous Copy via Host.

```
while (t<t_max) {
        t+=dt;

        /*****************************/
        /*** Compute Diffusion on H ***/
        /*****************************/
        for (d=0; d<num_gpus; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
        diffusion_Solve_H_GPU<<<bl, tr>>>(h_dev[d], s_dev[d], h_new_dev[d],\
                        alpha_dev[d], beta_dev[d], pitch_byt, dx, dy, dt,\
                        c_s, c_m, M2, N2_dev);
        }
        /* Update Boundary Conditions on H */
        for (d=0; d<num_gpus; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
        updateBC_Neuman_Homogeneous_H_GPU<<<bl, tr>>>(h_new_dev[d],\
                        pitch_byt, dx, dy, M2, N2_dev);
        }
        /* This update is only where we have the influx */
        HANDLE_ERROR( cudaSetDevice(device[0]) );
        updateBC_Neuman_Inhomogeneous_H_GPU<<<bl, tr>>>(h_dev[0], h_new_dev[0],\
                        alpha_dev[0], beta_dev[0], pitch_byt,\
                        influx_start, influx_stop, f_s, f_m, dy);

        /*** Section copying data from device to host - swapping pointers on host
             - copy from host to device ***/
        /* Copy Border-data from Device to Host */
        for (d=1; d<num_gpus; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
                HANDLE_ERROR( cudaMemcpy(h_first_row_send_host[d],\
                                        h_new_dev[d]+M2+1, M2,\
                                        cudaMemcpyDeviceToHost) );
        }
        for (d=0; d<num_gpus-1; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
                HANDLE_ERROR( cudaMemcpy(h_last_row_send_host[d],\
                                        h_new_dev[d]+(N_dev*M2)+1, M2,\
                                        cudaMemcpyDeviceToHost) );
        }

        /* Swap boundary pointers on host */
        for (d=0; d<num_gpus-1; d++) {
                h_last_row_recv_host[d] = h_first_row_send_host[d+1];
                h_first_row_recv_host[d+1]= h_last_row_send_host[d];
        }

        /* Copy Border-Arrays from host to matrix on device */
        for (d=0; d<num_gpus-1; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
                HANDLE_ERROR( cudaMemcpy(h_new_dev[d]+((N_dev+1)*M2+1),\
                                        h_last_row_recv_host[d], M2,\
                                        cudaMemcpyHostToDevice) );
        }
        for (d=1; d<num_gpus; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
```

```
                HANDLE_ERROR( cudaMemcpy(h_new_dev[d],\
                                                h_first_row_recv_host[d], M2,\
                                                cudaMemcpyHostToDevice) );
        }

        /******************************/
        /*** Compute Convection on S ***/
        /******************************/
        for (d=0; d<num_gpus; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
                convection_Solve_S_GPU<<<bl, tr>>>(h_dev[d], s_dev[d],\
                        h_new_dev[d], s_new_dev[d], alpha_dev[d], pitch_byt,\
                        dx, dy, dt, c_s, c_m, A, M2, N2_dev);
        }

        /* Update Boundary Conditions on S */
        for (d=0; d<num_gpus; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
                updateBC_Neuman_Homogeneous_S_GPU<<<bl, tr>>>(s_new_dev[d],\
                        pitch_byt, dx, dy, M2, N2_dev);
        }
        /* This update is only where we have the influx */
        HANDLE_ERROR( cudaSetDevice(device[0]) );
        updateBC_Dirichlet_S_GPU<<<bl, tr>>>(s_new_dev[0], alpha_dev[0],\
                beta_dev[0], pitch_byt, influx_start, influx_stop, f_s, f_m, dy);

        /*** Section copying data from device to host - swapping pointers on host
             - copy from host to device ***/
        /* Copy data Border-Arrays from device to host for S */
        for (d=1; d<num_gpus; d++) {
                HANDLE_ERROR( cudaSetDevice(device[d]) );
                HANDLE_ERROR( cudaMemcpy(s_first_row_send_host[d],\
                                                s_new_dev[d]+M2+1, M2,\
                                                cudaMemcpyDeviceToHost) );
        }
        for (d=0; d<num_gpus-1; d++) {
                HANDLE_ERROR( cudaSetDevice(device[d]) );
                HANDLE_ERROR( cudaMemcpy(s_last_row_send_host[d],\
                                                s_new_dev[d]+(N_dev*M2)+1, M2,\
                                                cudaMemcpyDeviceToHost) );
        }

        /* Swap boundary pointers on host */
        for (d=0; d<num_gpus-1; d++) {
                s_last_row_recv_host[d] = s_first_row_send_host[d+1];
                s_first_row_recv_host[d+1]= s_last_row_send_host[d];
        }

        /* Copy from Border-Arrays to from host to matrix on device for S */
        for (d=0; d<num_gpus-1; d++) {
                HANDLE_ERROR( cudaSetDevice(device[d]) );
                HANDLE_ERROR( cudaMemcpy(s_new_dev[d]+((N_dev+1)*M2+1),\
                                                s_last_row_recv_host[d], M2,\
                                                cudaMemcpyHostToDevice) );
        }
        for (d=1; d<num_gpus; d++) {
                HANDLE_ERROR( cudaSetDevice(device[d]) );
                HANDLE_ERROR( cudaMemcpy(s_new_dev[d],\
                                                s_first_row_recv_host[d], M2,
                                                cudaMemcpyHostToDevice) );
                //HANDLE_ERROR( cudaMemcpy(s_new_dev[d]+(M2+1),
                    s_first_row_recv_host[d], M2, cudaMemcpyHostToDevice) );
        }

        /* Swap array-pointers before the next timestep */
        for (d=0; d<num_gpus; d++) {
                HANDLE_ERROR( cudaSetDevice(device[d]) );
                h_tmp_dev[d] = h_dev[d];
                h_dev[d]     = h_new_dev[d];
                h_new_dev[d] = h_tmp_dev[d];
        }
        for (d=0; d<num_gpus; d++) {
                HANDLE_ERROR( cudaSetDevice(device[d]) );
                s_tmp_dev[d] = s_dev[d];
                s_dev[d]     = s_new_dev[d];
                s_new_dev[d] = s_tmp_dev[d];
        }
}
```

8.4.4.4 Implementation 3.

This implementation, is also coupling multiple GPU's using synchronous computations and communications, but the communications between GPUs are performed using P2P over the PCI Express bus. For this the memory has been pinned, meaning that the host memory has been allocated through the function **`cudaHostAlloc()`** as shown in code listing 8.11.

Since the ghost- and boundary-data of *h* and *s* need to be communicated between the GPUs, the transfer of the data have been deployed by copying the boundary data from the padded 2D-matrix on the device into an unpadded array also on the same device. How padded data are accessed and copied into unpadded data, and vice versa, can be viewed in the code listing 8.13.

Listing 8.13: Copy Boundary-Points to the Neighbors Ghost-Points.

```
/*** Copy boundary-data from data-domain to the transfer-array ***/
__global__ void copyMatrixDataToTransferArrays(double *h_new, double *h_row_send,
      int M2, int row_no, size_t d_pitch) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  if (j==row_no && i>=0 && i<M2) {
    double* h  = (double*)((char*)h_new + j*d_pitch);
    h_row_send[i] = h[i];
  }
}

/*** Copy ghost-data from the transfer-array the data-domain ***/
__global__ void copyTransferArraysToMatrixData(double *h_new, double *h_row_recv,
      int M2, int row_no, size_t d_pitch) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  if (j==row_no && i>=0 && i<M2) {
    double* h  = (double*)((char*)h_new + j*d_pitch);
    h[i] = h_row_recv[i];
  }
}
```

After the copying the padded data to the unpadded arrays, transfer of arrays between the devices is performed utilizing the synchronous P2P copy function **`cudaMemcpyPeer()`**. When the neighboring device has received the array, it is copied into the 2D padded matrix on this new device. For this functionality, the sections under diffusion and convection regarding copy and transfer of boundary/ghost data as seen in code listing 8.12, can be changed with the code block as revealed in listing 8.14. This code handles *h*, but the same is applied on *s*.

Listing 8.14: Kernel Calls to Copy Data and Synchronous Transfer of Data P2P.

```
/* Copy from 2D Padded Matrix Data to
 * 1D Unpadded Arrays before P2P transfer for H */

for (d=1; d<num_gpus; d++) {  //copy second first row in the lower grid
        HANDLE_ERROR( cudaSetDevice(device[d]) );
        copyMatrixDataToTransferArrays<<<bl, tr>>>(h_new_dev[d],
                h_first_row_send[d], M2, 1, pitch_byt);
}
for (d=0; d<num_gpus-1; d++) { //copy second last row in upper grid
        HANDLE_ERROR( cudaSetDevice(device[d]) );
        copyMatrixDataToTransferArrays<<<bl, tr>>>(h_new_dev[d],
                h_last_row_send[d], M2, N_dev, pitch_byt);
}

/* Peer-To-Peer-copy of Border-Arrays between devices for H */
for (d=0; d<num_gpus-1; d++) { //copy from grid 1 to 3, to grid 0 to 2
        HANDLE_ERROR( cudaSetDevice(device[d+1]) );
        HANDLE_ERROR( cudaMemcpyPeer(h_first_row_recv[d+1], device[d+1],
                                h_last_row_send[d], device[d], pitch_byt));
}
for (d=0; d<num_gpus-1; d++) { //copy from grid 0 to 2, to grid 1 to 3
        HANDLE_ERROR( cudaSetDevice(device[d]) );
        HANDLE_ERROR( cudaMemcpyPeer(h_last_row_recv[d], device[d],
                                    h_first_row_send[d+1], device[d+1],
                                        pitch_byt));
}

/* Copy from 1D Unpadded Arrays to
 * 2D Padded Matrix Data after P2P transfer for H */
for (d=0; d<num_gpus-1; d++) { //copy into last row of lower grid
        HANDLE_ERROR( cudaSetDevice(device[d]) );
        copyTransferArraysToMatrixData<<<bl, tr>>>(h_new_dev[d],
                 h_last_row_recv[d], M2, N_dev+1, pitch_byt);
}
for (d=1; d<num_gpus; d++) { //copy into first row in upper grid
        HANDLE_ERROR( cudaSetDevice(device[d]) );
        copyTransferArraysToMatrixData<<<bl, tr>>>(h_new_dev[d],
                 h_first_row_recv[d], M2, 0, pitch_byt);
}
```

8.4.4.5 Implementationn 4.

What differ this program from implementation (2), is that this one performs communication and computations asynchronously. Memory has been pinned and communication is performed via host. For asynchronous computations and communications between GPUs, CUDA Streams are applied for concurrency. A stream operates a sequence of commands and can only operate in one given context as on one given GPU. Although one GPU may operate multiple streams. In this application 2 streams has been created on each device as displayed in code listing 8.15. The code also shows the creation of CUDA events that will be utilized in a later application.

Listing 8.15: Creating 2 Streams and 2 Events on Each Device.

```
/* Create CUDA Streams and Events for each GPU */
int nStreams = 2, nEvents = 2;

cudaStream_t stream[num_gpus][nStreams];
cudaEvent_t event[num_gpus][nEvents];

for (d=0; d<num_gpus; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
        HANDLE_ERROR( cudaStreamCreate(&stream[d][0]) );
        HANDLE_ERROR( cudaStreamCreate(&stream[d][1]) );
        HANDLE_ERROR( cudaEventCreate(&event[d][0]) );
        HANDLE_ERROR( cudaEventCreate(&event[d][1]) );
}
```

As CUDA kernels are called or a calls to CUDA memory operations are carried out, the kernels and the memory functions have to take a given stream as parameter. This is displayed in listing 8.16.

Listing 8.16: Loop With 2 Streams Yielding Asynchrone Behavior.

```
while (t<t_max) {
        t += dt;

        /****************************/
        /*** Compute Diffusion on H ***/
        /****************************/
        for (d=0; d<num_gpus; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
        diffusion_Solve_H_GPU<<<bl, tr, 0, stream[d][0]>>>(h_dev[d], s_dev[d],\
                        h_new_dev[d], alpha_dev[d], beta_dev[d],\
                        pitch_byt, dx, dy, dt, c_s, c_m, M2, N2_dev);
        }
        /* Update Boundary Conditions on H */
        for (d=0; d<num_gpus; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
        updateBC_Neuman_Homogeneous_H_GPU<<<bl, tr, 0, stream[d][0]>>>(\
                         h_new_dev[d], pitch_byt, dx, dy, M2, N2_dev);
        }
        /* This update is only where we have the influx on the "global grid's"
            row 0, on H */
        HANDLE_ERROR( cudaSetDevice(device[0]) );
        updateBC_Neuman_Inhomogeneous_H_GPU<<<bl, tr, 0, stream[0][0]>>>(\
                         h_dev[0], h_new_dev[0], alpha_dev[0], beta_dev[0],\
                         pitch_byt, influx_start, influx_stop, f_s, f_m, dy);

        /* Copy Transfer-data from Device to Host */
        for (d=1; d<num_gpus; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
                HANDLE_ERROR( cudaMemcpyAsync(h_first_row_send_host[d],\
                                              h_new_dev[d]+M2+1, M2,\
                                                  cudaMemcpyDeviceToHost,\
                                                  stream[d][0]) );
        }
        for (d=0; d<num_gpus-1; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
                HANDLE_ERROR( cudaMemcpyAsync(h_last_row_send_host[d],\
                                                  h_new_dev[d]+(N_dev*M2)+1, M2,\
                                                  cudaMemcpyDeviceToHost,\
                                                  stream[d][1]) );
        }

        /* Swap boundary pointers on host */
        for (d=0; d<num_gpus-1; d++) {
                h_last_row_recv_host[d] = h_first_row_send_host[d+1];
                h_first_row_recv_host[d+1]= h_last_row_send_host[d];
        }

        /* Copy from Border-Arrays to Matrix Data after P2P transfer for H */
        for (d=0; d<num_gpus-1; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
                HANDLE_ERROR( cudaMemcpyAsync(h_new_dev[d]+((N_dev+1)*M2+1),\
                                                  h_last_row_recv_host[d], M2,\
                                                  cudaMemcpyHostToDevice,\
                                                  stream[d][0]) );
        }
        for (d=1; d<num_gpus; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
                HANDLE_ERROR( cudaMemcpyAsync(h_new_dev[d]+(M2+1),\
                                                  h_first_row_host[d], M2,\
                                                  cudaMemcpyHostToDevice,\
                                                  stream[d][1]) );
        }

        /*****************************/
        /*** Compute Convection on S ***/
        /*****************************/
        // same pattern is made for convection
}
```

8.4.4.6 Implementation 5.

This implementation is similar to implementation (4), but it performs asynchronous computations and P2P communications between GPUs over the PCI Express bus. The concurrency is deployed with CUDA Streams and synchronized with CUDA

Events. The creation of 2 events per device can be viewed in listing 8.15.
The P2P communication is set up as earlier, but the code is asynchronous, and therefore employs the function **`cudaMemcpyPeerAsync()`**, whitch also takes a dedicated stream as an argument. This application is also handling synchronization with the function **`cudaStreamWaitEvent()`** witch is waiting until all calls before the setting of an event-record in a given stream has finished executing and reports completion. It takes both a stream and an event as arguments. The event record is set with the function **`cudaEventRecord()`**, and takes an event as an argument. The function **`cudaStreamWaitEvent()`** can wait for events in streams on other devices.This cross device synchronization is efficient as it is not stalling the host it can enhance performance compared to other barriers. Example on use of asynchronous P2P copy, and kernel calls waiting on events is exhibited in listing 8.17.

Listing 8.17: Creating 2 Streams and 2 Events on Each Device.

```
for (d=0; d<num_gpus; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
        diffusion_Solve_H_GPU<<<bl, tr, 0, stream[d][0]>>>(h_dev[d], s_dev[d],
            h_new_dev[d], alpha_dev[d], beta_dev[d], pitch_byt, dx, dy, dt, c_s,
            c_m, M2, N2_dev);
        HANDLE_ERROR( cudaEventRecord(event[d][0]) );
}

for (d=0; d<num_gpus-1; d++) {
        HANDLE_ERROR( cudaSetDevice(device[d]) );
        HANDLE_ERROR( cudaStreamWaitEvent(stream[d+1][0], event[d+1][0], 0) );
        HANDLE_ERROR( cudaMemcpyPeerAsync(h_last_row_recv[d], device[d],
            h_first_row_send[d+1], device[d+1], pitch_byt, stream[d][0]) );
        //HANDLE_ERROR( cudaEventRecord(event[d][1]) );
}
```

8.4.4.7 Implementation 6.

This application, the last application described in 8.2, is coupling multiple GPU's using asynchronous computations and P2P communications between GPUs over the PCI Express bus. It applies CUDA Streams for concurrency.In addition it creates a set of OMP-threads for each GPU, and let one OMP-thread on the CPU launch one CUDA-kernel, and an other OMP-thread handle asynchronous communication. This is applied according to [Sourouri at al's] suggestions with 4 CUDA Streams and 2 OpenMP threads per device. With these features, observations of additional speedup can be expected and a reduction of kernel-launch overhead. Creation of OpenMP threads in presented in the listing 8.18.

Listing 8.18: Create Open MP Threads.

```
/*** Set the number of OpenMP threads ***/
    unsigned int num_OMP_threads = num_gpus*2;
    omp_set_num_threads(num_OMP_threads);

#pragma omp parallel
    {
    unsigned int nThreads = omp_get_num_threads();
    }
```

Other details in the application are unveiled in an extensive example listed in appendix.

8.4.4.8 CUDA Kernels

The CUDA kernels for diffusion and convection are as follows in listings 8.19 and 8.20.

Listing 8.19: CUDA Kernel for Diffusion.

```
/*** Solve the Diffusion term (h) ***/
__global__ void diffusion_Solve_H_GPU(\
                                double *d_h, double *d_s,\
                                double *d_h_new, double *d_alpha,\
                                double *d_beta, size_t d_pitch,\
                                double dx, double dy, double dt,\
                                double c_s, double c_m, int M, int N) {

          int i = blockDim.x * blockIdx.x + threadIdx.x;
          int j = blockDim.y * blockIdx.y + threadIdx.y;

          //printf("c_s=%.4f\t dy=%.4f\n", c_s, dy);

          /* Calculate the value for each inner grid point */
          if(i>0 && i<M-1 && j>0 && j<N-1) {
                  //if (i<M && j<N) {

                  /* Pointers to rows in the the different pitched arrays */
                  double* h     = (double*)((char*)d_h + j*d_pitch);
                  double* s     = (double*)((char*)d_s + j*d_pitch);
                  double* h_new = (double*)((char*)d_h_new + j*d_pitch);
                  double* alpha = (double*)((char*)d_alpha + j*d_pitch);
                  double* beta  = (double*)((char*)d_beta  + j*d_pitch);
                  int stride    = d_pitch/sizeof(double);
                  //printf("In kernel: h=%.4f\n", h[3050]);

                  double dx2_R=1/(dx*dx), dy2_R = 1/(dy*dy);
                  double as_center = alpha[i]*s[i], bs_center = beta[i]*(1-s[i]);

                  h_new[i]=((dt/c_s)*(\
          (((((as_center +alpha[i+1]*s[i+1])*.5)*(h[i+1]-h[i]))-\
          (((alpha[i-1]*s[i-1]+as_center)*.5)*(h[i]-h[i-1])))*dx2_R)+\
   (((((as_center+alpha[i+stride]*s[i+stride])*.5)*(h[i+stride]-h[i]))-\
          (((alpha[i-stride]*s[i-stride]+as_center)*.5)*(h[i]-h[i-stride])))*dy2_R)
                ))+\
          ((dt/c_m)*(\
  (((((bs_center+beta[i+1]*(1-s[i+1]))*.5)*(h[i+1]-h[i]))-\
    (((beta[i-1]*(1-s[i-1])+bs_center)*.5)*(h[i]-h[i-1])))*dx2_R)+\
 (((((bs_center+beta[i+stride]*(1-s[i+stride]))*.5)*(h[i+stride]-h[i]))- \
   (((beta[i-stride]*(1-s[i-stride])+bs_center)*.5)*(h[i]-h[i-stride])))*dy2_R)))
        +\
         h[i];
         }
}
```

Listing 8.20: CUDA Kernel for Convection.

```
/*** Solve the Convection term (s) ***/
__global__ void convection_Solve_S_GPU(
                        double *d_h, double *d_s,\
                        double *d_h_new, double *d_s_new,\
                        double *d_alpha, size_t d_pitch,\
                        double dx, double dy, double dt,\
                        double c_s, double c_m,\
                        int A, int d_M, int d_N) {

        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int j = blockIdx.y * blockDim.y + threadIdx.y;

        if(i>0 && i<d_M-1 && j>0 && j<d_N-1) {

                /* Pointers to rows in the the different pitched arrays */
                double* h      = (double*)((char*)d_h + j*d_pitch);
                double* s      = (double*)((char*)d_s + j*d_pitch);
                double* h_new  = (double*)((char*)d_h_new + j*d_pitch);
                double* s_new  = (double*)((char*)d_s_new + j*d_pitch);
                double* alpha  = (double*)((char*)d_alpha + j*d_pitch);

                double s_x, s_y, h_x, h_y, h_xx, h_yy;
                double as_center = alpha[i]*s[i]; //g1
                double dx_R=1/dx, dy_R = 1/dy;
                double dx2_R=1/(dx*dx), dy2_R = 1/(dy*dy);
                int stride     = d_pitch/sizeof(double);

                /* Calculate the value for each inner grid point */
                if ( (h_y = (h_new[i+stride] - h_new[i-stride]) * 0.5 * dy_R)>0)
                        s_y = (alpha[i+stride]*s[i+stride] - as_center) * dy_R;
                else
                        s_y = (as_center - alpha[i-stride]*s[i-stride]) * dy_R;

                if ( (h_x = (h_new[i+1] - h_new[i-1]) * 0.5 * dx_R) > 0 )
                        s_x = (alpha[i+1]*s[i+1] - as_center) * dx_R;
                else
                        s_x = (as_center - alpha[i-1]*s[i-1]) * dx_R;

                h_xx = (h_new[i+1]       - 2*h_new[i] + h_new[i-1])       * dx2_R;
                h_yy = (h_new[i+stride] - 2*h_new[i] + h_new[i-stride]) * dy2_R;

                s_new[i] = ( (dt/c_s) * (s_x * h_x + s_y * h_y + as_center \
                                * (h_xx + h_yy) ) + A*s[i] ) / (A+h_new[i]-h[i]);

        }
}
```

Chapter 9

Performance Analysis

Implementing a parallel algorithm can be difficult, and in order to find out if it is worth the trouble to implement it, some tools are used help to predict the enhancement in performance of a parallel system. How much faster is a parallel algorithm compared to a sequential algorithm?
To answer this question some formulas for speedup and efficiency will be investigated.

9.1 Speedup

Viewed in the light of parallel computing, speedup is the ratio between the sequential execution time and the parallel and hopefully improved execution time. The Speedup can be defined by the following formula:

$$Speedup = \frac{Sequential\ execution\ time}{Parallel\ execution\ time}$$

When the number of parallel processors is denoted n, the number of serial processors is 1, the *speedup* is denoted S and *time* is T, we can rewrite the formula for speedup:

$$S = \frac{T(1)}{T(n)}$$

From this model a more specified model can be derived:

$$\psi(n,p) \leq \frac{\sigma(n)+\phi(n)}{\sigma(n)+\phi(n)/p+\kappa(n,p)}$$

9.2 Efficiency

Efficiency is defined by speedup divided by the number of processors used:

$$Efficiency = \frac{Sequential\ execution\ time}{Processors\ used \times Parallel\ execution\ time}$$

$$\varepsilon(n,p) \leq \frac{\sigma(n)+\phi(n)}{p\big(\sigma(n)+\phi(n)/p+\kappa(n,p)\big)}$$

$$\varepsilon(n,p) \leq \frac{\sigma(n)+\phi(n)}{p\sigma(n)+\phi(n)+p\kappa(n,p)}$$

We have that $0 \leq \varepsilon(n,p) \leq 1$, as all terms are greater than or equal to zero.

9.3 Amdahl's law

$$\psi(n,p) \leq \frac{\sigma(n)+\phi(n)}{\sigma(n)+\phi(n)/p+\kappa(n,p)}$$

Amdahl's law raises some issues regarding parallel computations, It ignores overhead related to communication time!

9.4 Gustafson's law

Both Gustafson'-Barsis's law and Amdahl's law ignores the term for parallel overhead: $\kappa(n,p)$, and can by that overestimate the speedup achieve by a parallel system.

9.5 The Karp-Flatt Metric

Takes into consideration the overhead term.

$$T(n,p) = \sigma(n)+\phi(n)/p+\kappa(n,p)$$

9.6 The Isoefficiency Metric

Scalability is a measure of the ability to increase performance as the number of processors increases in a parallel system.

Chapter 10

Conclusion

This chapter concludes this thesis, presenting the results from the investigation and contribution of this research.

10.1 Summary

High Performance Computing has been the he main focus in this work. The aim has been to enhance the performance of simulations by having the computations performed in parallel on GPUs with the CUDA architecture rather than serially on a single CPU core. The model of choice for the conduction of the tests has been a sediment transport model. The PDEs representing the model have been discretized into to a fully explicit finite difference scheme and implemented accordingly. The sediment transport model has been implemented with ANSI-C programming language for the serial implementation and CUDA C for the parallel implementation.

Various parallel features have been applied and tested in the conduction of this work. Simulations have been made with numerous optimization techniques for enhanced speedup on the parallel architecture. The tests have been performed on 2 different computers, both holding multiple NVIDIA GPUs with the CUDA architecture. The two computers have GPUs with different specifications and different numbers of available GPUs as shown in table 10.1.

GPUs	GPU Type	Architecture
4	GeForce GTX 590	Fermi
2	Tesla K20m	Kepler

Table 10.1: Available GPUs for Testing

Specification of Implementations
Both the serial and parallel implementations benefit from a few optimizing strategies like coalesced memory and some intermediate variables to reduce the number of operations. The "Generic Code" has been for written for an arbitrary number of GPUs. It can run on all the GPUs the system has available, and decomposes the data-domain accordingly.

The different implementations and their characteristics are listed in table 10.2. The communication between devices for the multi-GPU applications has been tested for both synchronous and asynchronous communication, as is shown under the column named "Comm", and for asynchronous communication, additional CUDA streams have been deployed. Communication between devices can be performed P2P over the PCIe bus or via host, implied with a "Yes" or a "No" in the table. Also the right number of streams are launched according to the number of devices the simulation is set for.

Application	**#GPUs**	**Comm**	**P2P**	**#Streams**
1 CPU	N/A	N/A	N/A	N/A
1 GPU	1	N/A	N/A	1
Hardcoded for 2 GPUs	2	Sync	Yes	1
Generic Code	Multi	Sync	Yes	1
Generic Code	Multi	Sync	No	1
Generic Code	Multi	Async	No	2

Table 10.2: Implementation Specifications

10.2 Results

Diffusion-based evolution of Lake Okeechobee has been simulated for 1000 years and the solutions can be viewed in the figures 10.1, 10.2.

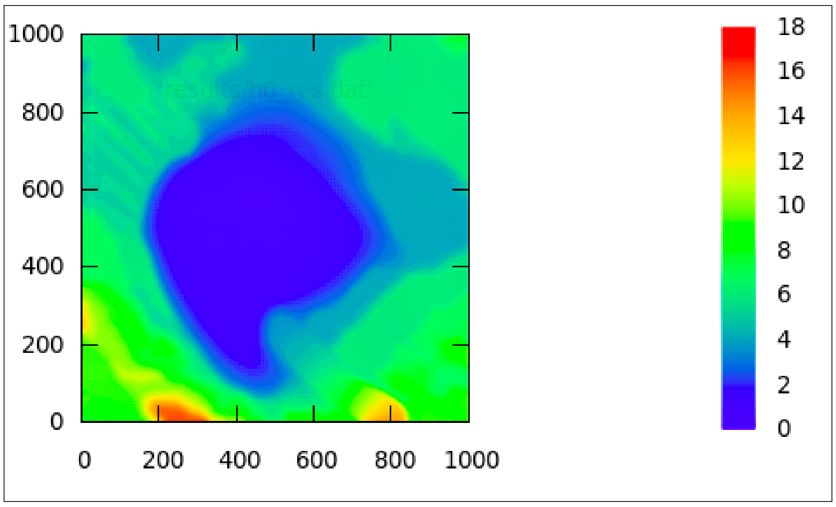

Figure 10.1: Solution of h after 1000 years

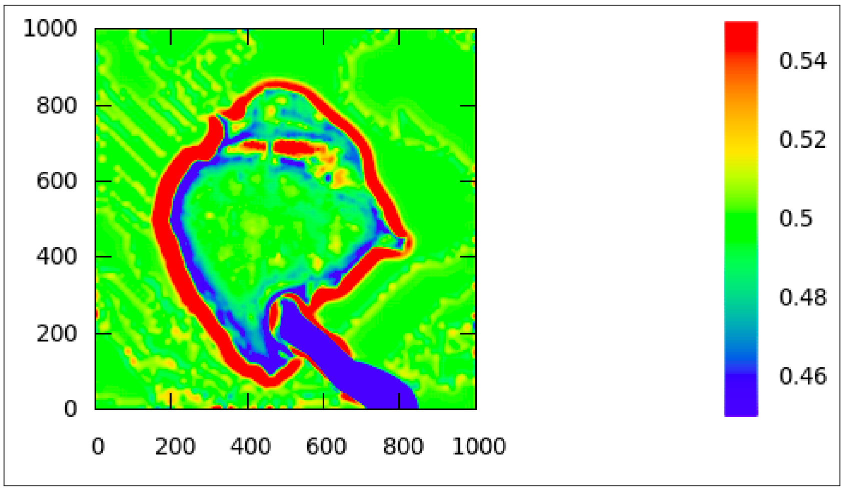

Figure 10.2: Solution of s after 1000 years

All the simulations have been tested with data of **double precision**. Results from the different tests that were carried out in this investigation are unveiled in table 10.3 for GeForce GTX 590 and table 10.4 for Tesla K20m. The results are for a 10 year simulation.

All the implementations have been tested on a data domain with a grid size of 1000x1000. Tests on different multi-GPU implementations have been carried out for both 2 and 4 GPUs on GeForce GTX 590 where 4 GPUs are applicable.

# GPUs	Domain Size	Time	Speedup	Application
0	1000x1000	40.55 s	N/A	Serial code
1	1000x1000	2.125 s	19.08	Parallel code, 1STREAM
2	1000x1000	1.222 s	33.18	Hardcoded for 2 GPUs, SYNC, P2P, 1STREAM
2	1000x1000	1.244 s	32.60	Generic code for Multi-GPU, SYNC, P2P, 1STREAM
4	1000x1000	0.920 s	44.08	Generic code for Multi-GPU, SYNC, P2P, 1STREAM
2	1000x1000	1.509 s	26.99	Generic code for Multi-GPU, SYNC, copyViaHOST, 1STREAM
4	1000x1000	1.221 s	33.36	Generic code for Multi-GPU, SYNC, copyViaHOST, 1STREAM
2	1000x1000	1.350 s	30.17	Generic code for Multi-GPU, ASYNC, copyViaHOST, 2STREAMS
4	1000x1000	0.696 s	58.52	Generic code for Multi-GPU, ASYNC, copyViaHOST, 2STREAMS

Table 10.3: Speedup on GeForce GTX 590 for the maximum of 4 GPUs

# GPUs	Domain Size	Time	Speedup	Application
0	1000x1000	38.01 s	N/A	Serial code
1	1000x1000	1.156 s	32.88	Parallel code, 1STREAM
2	1000x1000	0.763 s	49.82	Hardcoded for 2 GPUs, SYNC, P2P, 1STREAM
2	1000x1000	0.812 s	46.81	Generic code for Multi-GPU, SYNC, P2P, 1STREAM
2	1000x1000	0.953 s	39.88	Generic code for Multi-GPU, SYNC, copyViaHOST, 1STREAM
2	1000x1000	0.769 s	49.43	Generic code for Multi-GPU, ASYNC, copyViaHOST, 2STREAMS

Table 10.4: Speedup onTesla K20 for the maximum of 2 GPUs

10.2.1 Analysis

The tests have been carried out on 2 different computers, both holding multiple NVIDIA GPUs with the CUDA architecture.

The optimal size of thread blocks has proven to be 16×4 on Fermi, and 16×8 on Tesla on a grid of size 1000×1000.

Analysis of the results from GeForce GTX 590, Fermi, seen in table 10.3 and for Tesla K20m, Kepler as seen in table 10.4, show a clear enhancement on using GPU's for simulations of the sediment transport model vs. CPU. All the implementations of so-called generic code for multi-GPUs have been tested for both 2 and 4 GPUs on Fermi.

1 GPU, 1 STREAM:
The speedup from the serial computations on a CPU to the parallel computations on 1 GPU shows that the speedup is 19.08 on Fermi and 27.2 on Kepler. This is a remarkable speedup, considering that large simulations can take weeks and months to carry out.

Multi-GPU hardcoded, SYNC, P2P, 1 STREAM:
For multi-GPU implementations that are synchronous with P2P communication a hardcoded version has been implemented in addition to the "generic code". The implementation of a hardcoded parallel version for 2 GPUs runs faster than the generic code on 2 GPUs with the same features. This is due to the reduction of programming structures created to handle an arbitrary number of GPUs, and the tests show that the hardcoded version is more optimized with respect to speedup.

With respect to results from 1 GPU an "ideal scaling" would be if timing the results for 1 GPU $/2$ equals the results for 2 GPU. The simulations for 1 GPU indicate an ideal scaling from 1 to 2 GPUs as:
$2.125/2 = 1.0625$ on Fermi.
The results for 2 GPUs hardcoded is 1.222, which is a little over the ideal, but not too bad considering the overhead due to communication between GPUs. This implementation yields a good speedup on both Fermi and Kepler, with a speedup of respectively 33.18 and 49.82.

Multi GPU, Generic code, SYNC, P2P, 1 STREAM:
This version has been tested on both 2 and 4 GPUs. On 2 GPUs it yields a slightly poorer result than the hardcoded version, as expected. With respect to 4 devices the generic code yields a very acceptable speedup, although it does not scale ide-

ally, again due to communication overhead. This version shows a nice speedup from 2 to 4 GPUs from 33.6 to 44.08. The communication between the GPUs is synchronous, although P2P, and is launched sequentially by the CPU so a kernel launch overhead is to be expected in this implementation. The more GPUs involved, the more kernel launches and memory copies are issued, since the set of instructions is launched on every GPU.

Multi GPU, Generic code, SYNC, copy via HOST, 1STREAM:
This implementation is almost the same as the one above, but this copies data between devices via host, and not P2P. Without the benefit of the PCIe transfer of the data between devices, it shows in the table 10.3 a reduced speed on both 2 and 4 GPUs, with respect to the P2P implementation, as was expected.

Multi GPU, Generic code, ASYNC, copy via HOST, 2 STREAMS:
This implementation is almost the same as the one above, but it deploys asynchronous kernel executions and memory copies. The results clearly show that asynchronous execution yields very good speedup compared to synchronous. The asynchronous execution is performed with 2 CUDA Streams to yield the asynchronous behavior. The simulations on Fermi yields a speedup of 58.52 on 4 GPUs, and 49.43 on 2 K20m GPUs.

The results for the different implementations show the same tendencies on Fermi and Kepler, although Kepler is faster on 2 GPUs than Fermi, due the enhanced specifications of the GPU.

Contribution
With respect to the Thesis Questions that were stated at the beginning of this investigation, as can be seen in table 2.1, the results as described above, answers and enlightens the topics that this thesis aimed to solve and answer. This thesis' questions are answered in table 10.5. This research has proved that:

A 1:	A parallel application running on a single GPU is faster than a serial application running on a CPU core.
A 2:	A parallel application running on multiple GPU's is faster than a parallel application running on a single GPU.
A 3:	The application does enhance performance utterly if additional CUDA Streams are utilized.
A 3:	A parallel application running on a single GPU yields the exact same results as the serial application running on a CPU core, hence the same accuracy.
A 4:	A parallel application running on multiple GPU's does yield results with the same accuracy as the application running on a single GPU.

Table 10.5: Answers to the Thesis Questions

Looking at the overall results, with a speedup of 58.52 on Fermi's 4 GPUs, and 49.43 on Kepler's 2, it is clear that taking the effort of implementing applications like this sediment-transport model on a parallel architecture is worth while. With respect to the accuracy of the results, the simulations on GPUs give the exact same result as the CPU, also for double precision data, as was deployed in this research. Others, [*Tutkun* and *Edis* 2012] and [*Michéa* and *Komatitsch* 2010], have found that parallel applications of similar kinds yield a speedup between 9 and 60 for GPUs of different kinds and also in combination with Message Passing Interface (MPI). Although, in this matter, it is difficult to compare these results with others, it is not unlikely to think that the results presented in this thesis are quite satisfying.

10.3 Future Work

As for future work different optimizations are possible to carry out.

One is to expand the implementations to take more CUDA streams for utterly parallelism.

In addition to more streams, OMP threads could be combined. The OMP threads may reduce kernel launch overhead as suggested by [*Sourouri* et al. 2014]. The OMP threads could be tested with 2 threads as for one of the already existing implementations, or for implementations with more streams and P2P communications.

Another suggestion for future work would be to see if the use texture memory for constant variables like the matrices holding the coefficients α and β is enhancing speedup.

If it would be possible to have more (high end) GPUs available, it would be nice to test how well these systems scale as the number of GPUs increases.

Appendix A

List of Code Examples

Bibliography

[1] *NVIDIA Corporation*. *NVIDIA CUDA Toolkit Documentation, v7.0*. URL: `https://docs.nvidia.com/cuda/index.html` (cit. on pp. 2, 39–44).

[2] *Jan C. Rivenæs*. "Application of a dual-lithology, depth-dependent diffusion equation in stratigraphic simulation". In: *Basin Research* 4.2 (). ISSN: 1365-2117. URL: `http://dx.doi.org/10.1111/j.1365-2117.1992.tb00136.x` (cit. on pp. 2, 5).

[3] *Jan C. Rivenæs*. "A computer simulation model for siliclastic basin stratigraphy". University of Trondheim, 1993 (cit. on pp. 3, 11, 12, 18, 57).

[4] *D. E. Comer* et al. "Computing As a Discipline". In: *Commun. ACM* 32.1 (Jan. 1989). Ed. by *Peter J. Denning*, pp. 9–23. ISSN: 0001-0782. DOI: `10.1145/63238.63239`. URL: `http://doi.acm.org/10.1145/63238.63239` (cit. on p. 11).

[5] *U. G. Survey*. *Gtopo30*. In: 1996 (cit. on pp. 13, 17, 20, 67).

[6] *S.R. Clark*, *Wenjie Wei*, and *Xing Cai*. "Numerical Analysis of a Dual-Sediment Transport Model Applied to Lake Okeechobee, Florida". In: *Parallel and Distributed Computing (ISPDC), 2010 Ninth International Symposium on*. 2010, pp. 189–194. DOI: `10.1109/ISPDC.2010.29` (cit. on pp. 17, 19, 20, 65, 67).

[7] *T. E. Jordan* and *P. B. Flemings*. "Large-scale stratigraphic architecture, eustatic variation, and unsteady tectonism: A theoretical evaluation". In: *Journal of Geophysical Research: Solid Earth* 96.B4 (1991), pp. 6681–6699. ISSN: 2156-2202. DOI: `10.1029/90JB01399`. URL: `http://dx.doi.org/10.1029/90JB01399` (cit. on p. 17).

[8] *Wenjie Wei* et al. "Balancing efficiency and accuracy for sediment transport simulations". In: *Computational Science Discovery* 6.1 (2013), p. 015011. URL: `http://stacks.iop.org/1749-4699/6/i=1/a=015011` (cit. on pp. 21, 22).

[9] *Top500List*. *Top 500 Supercomputers*. 2015. URL: `\url{http://www.top500.org/},urldate={2015-06-01}` (cit. on p. 33).

[10] *Michael J Quinn*. *Parallel Programming*. Vol. 526. TMH CSE, 2003 (cit. on pp. 35, 37).

[11] *David B Kirk* and *W Hwu Wen-mei*. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012 (cit. on p. 45).

[12] *John Nickolls* et al. “Scalable parallel programming with CUDA”. In: *Queue* 6.2 (2008), pp. 40–53 (cit. on p. 48).

[13] *Unidata*. *Network Common Data Form NetCDF*. 2015. URL: http://www.unidata.ucar.edu/software/netcdf/ (visited on 05/01/2015) (cit. on p. 67).

[14] *Bulent Tutkun* and *Firat Oguz Edis*. “A GPU application for high-order compact finite difference scheme”. In: *Computers & Fluids* 55 (2012), pp. 29–35 (cit. on p. 100).

[15] *David Michéa* and *Dimitri Komatitsch*. “Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards”. In: *Geophysical Journal International* 182.1 (2010), pp. 389–402 (cit. on p. 100).

[16] *M. Sourouri* et al. “Effective multi-GPU communication using multiple CUDA streams and threads”. In: *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*. 2014, pp. 981–986. DOI: 10.1109/PADSW.2014.7097919 (cit. on p. 100).