Increasing the Performance of the Wetland DEM Ponding Model using Multiple GPUs

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Tonghe Liu

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Or

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

Due to the lack of conventional drainage systems on the Canadian Prairies, when excess water runs off the landscape because of the snow-melt and heavy rainfall, the water may be trapped in surface depressions ranging in size from puddles to permanent wetlands and may cause local flooding. Hydrological processes play an important role in the Canadian Prairies regions, and using hydrological simulation models helps people understand past hydrological events and predict future ones. In order to obtain an accurate simulation, higher-resolution systems and larger simulation areas are introduced, and those lead to the need to solve larger-scale problems. However, the size of the problem is often limited by available computational resources, and solving large systems results in unacceptable simulation durations. Therefore, improving the computational efficiency and taking advantage of available computational resources is an urgent task for hydrological researchers and software developers. The Wetland DEM Ponding Model (`WDPM`) was developed to model the distribution of runoff water on the Canadian Prairies. It helps determine the fraction of Prairie basins contributing flows to stream while these change dynamically with water storage in the depressions. In the WDPM, the water redistribution module is the most computationally intensive part. Previously, the WDPM has been developed to run in parallel with one CPU or one GPU that makes the water redistribution module more efficient. Multi-device parallel computing is a common method to increase the available computation resources and could effectively speed up the application with an appropriate parallel algorithm.

This thesis develops a multiple-GPU parallel algorithm and investigates efficient data transmission methods compared to the CPU parallel and one-GPU parallel algorithm. A technique that overlaps communication with computation is applied to optimize the parallel computing process. Then the thesis evaluates the new implementation from several aspects. In the first step, the output summary and the output system are compared between the new implementation and the initial one. The solution shows significant convergence as the simulation processes, verifying the new implementation produces the correct result. In the second step, the multiple-GPU code is profiled, and it is verified that the algorithm can be re-organized to take advantage of multiple GPUs and carry out efficient data synchronization through optimized techniques. Finally, by means of numerical experiments, the new implementation shows performance improvement when using multiple GPUs and demonstrates good scaling. In particular, when working with a large system, the multiple-GPU implementation produces correct output and shows that there is around 2.35 times improvement in the performance compared using four GPUs with using one GPU.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Raymond J. Spiteri, for his significant academic support, writing guidance, and personal advice in the past year and a half. I would also express my sincere thanks to my co-supervisor, Dr. Seok-Bum Ko, for recommending me to this project and for teaching me academic knowledge. It is one of my luckiest things to study under their supervision. I could never finish my M.Sc. degree without their extensive knowledge, patient instructions, and selfless help. I would like to acknowledge my parents, Mrs. Kui Song and Mr. Qiang Liu, and my elder brother, Xuantian Li, for their all-aspect support when I am far from home and for getting me familiar with the university and life when I first came here. I would like to acknowledge Dr. Kevin Shook from the Centre for Hydrology and Dr. Olivier Fisette from the Advanced Research Computing (ARC) team for providing me the documents to support my experiment and the technical knowledge of programming. I would like to thank Dr. Kevin R. Green from the Numerical Simulation Laboratory and Dr. Hao Zhang from the Electrical and Computer Engineering department for their insightful guidance. Finally, I would like to thank other colleagues from the Numerical Simulation Laboratory for their generous help and my dear friends for our lasting friendship.

Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
DEM	Digital Elevation Model
ESRI	Environment System Research Institute
FP	Floating Point
FPGA	Field-programmable Gate Array
GIS	Geographic Information System
GPU	Graphics Processing Units
HDL	Hardware Description Language
I/O	Input/Output
INT	Integer
LiDAR	Light Detection and Ranging
LDU	Load-Store Units
LIRA	Land and Infrastructure Resiliency Assessment
OpenCL	Open Computing Language
SCRB	Smith Creek Research Basin
SFU	Special-Function Units
SIMD	Single Instruction, Multiple Data
SM	Stream Multiprocessors
SW	Shapiro and Westervelt
WDPM	Wetland DEM Ponding Model

Chapter 1

INTRODUCTION

1.1 Motivation

Graphics processing units (GPUs) are a popular platform for executing general-purpose parallel applications. With many GPU programming systems such as CUDA, OpenACC, and OpenCL, programmers can parallelize an application into thousands of threads that execute the same code. Many applications show significant performance improvement when working with GPUs. These applications cover various fields such as physics (ANSYS, which simulates the interaction of liquids and gases with surfaces [4]), biological sciences (Basic Local Alignment Search Tool, which is one of the most widely used bioinformatics tools [5]), chemistry (Vienna Ab-initio Simulation Package, which is used for quantum mechanics and molecular dynamics simulation [6]), and weather forecasting (The Weather Research and Forecasting Model, a numerical weather forecast system [7]). Meanwhile, other studies have explored how to fully utilize the computation ability of GPUs. However, obtaining peak GPU performance requires significant programming effort. In addition to some general methods to optimize the code, programmers need to design the code such that it is compatible with the characteristics of the underlying GPU hardware. Sometimes, this kind of extra work is necessary, but hardware-specific algorithms require programmers to have sufficient hardware knowledge, but then at the same time, the application's portability is reduced. In this case, instead of designing an application that perfectly fits a particular GPU hardware configuration, increasing the degree of parallelism is a promising approach [8]. The focus of this thesis is parallel computing with multiple GPUs.

The Wetland DEM Ponding Model (WDPM) was developed by the Centre for Hydrology at the University of Saskatchewan to model the distribution of runoff water on the Canadian Prairies. It helps determine the fraction of Prairie basins contributing flows to stream while these change dynamically with water storage in the depressions. The model has also been used to demonstrate the extent of flooding on Prairie landscapes [2]. The program was initially written in Fortran and adapted to CPU parallel computing with OpenMP. Later, it was translated to C, and the parallel computation was implemented in OpenCL, a cross platform parallel programming language. With OpenCL, the WDPM can ran faster and is able to run on a GPU. After this evolution, the performance of this program improved significantly. For example, a simulation that originally took a year to complete now only took seven weeks. Large problems require more advanced techniques that accelerate the simulation and allow researchers to solve the problem in real time.

This thesis investigates how the WDPM can be correctly and efficiently executed with multiple GPUs in parallel. This thesis also applies some general GPU programming optimization methods. But as mentioned above, fine-tuning the algorithm does not make a significant difference, and the WDPM has been well developed for several years. So the performance improvement is mainly embodied by applying multiple-GPU parallelism.

1.2 Contributions

Based on an existing code (WDPM version 1.0 [2]) that solves WDPM in parallel with one GPU or one CPU, a new main loop is contributed to apply the application for one host containing multiple GPUs, and an advanced data synchronization method is introduced to reduces the overhead and improves the performance. All these processes are implemented with OpenCL.

The multiple-GPU parallel algorithm is developed referring to the CPU parallel and GPU parallel methods applied to WDPM. The correctness of this parallel method is verified by evaluating the output summaries and the errors of multiple-GPU output maps. A more efficient data synchronization method was worked out to overlap the data communication with computation. The experiments that test the running time of working with different numbers of GPUs (up to four) when solving problems of different scales are performed to evaluate the performance of the new implementation. It is observed that for the small system (about 500×500), using multiple GPUs actually makes the performance worse, around 1.69 times slower when using 2 GPUs compared with using 1 GPU. But as the scales of the problems increase, using multiple GPUs shows better performance. For a system of approximate size 2500×3000, the maximum speedup is 1.61 when using 2 GPUs. For a system of approximate size 4500×4500, the maximum speedup is 1.92 when using 3 GPUs. For a system of approximate size 6000×6000, the maximum speedup is 3.1 when using 4 GPUs. These results verify the significant performance improvement of running the WDPM on multiple GPUs, and they further prove the good scaling of the new implementation. As a result, it can be expected that the performance improvement would be more significant when the systems become even larger.

1.3 Outline

The remaining parts of this thesis are organized as follows. Section 2 introduces some background knowledge This section includes the literature review about CPU and GPU computing technique, the knowledge about programming and hardware architecture, and the numerical methods related to the WDPM. Chapter 3 discusses in detail the parallel computing algorithm and digs into the GPU computing optimization for WDPM. Chapter 4 demonstrates the results and does performance analysis. Finally, Chapter 5 concludes and discusses future research directions.

Chapter 2

Background Knowledge

2.1 ArcGIS

ArcGIS is a geographic information system (GIS) that works with maps and geographic information maintained by the Environmental Systems Research Institute (ESRI). It has wide applications in creating and using maps, compiling geographic data, analyzing mapped information, and discovering geographic information. A GIS file contains a system that recognizes the raster graphics pixel as the smallest individual grid unit building block of an image [9]. Also, the system size, default value of the non-observation point, and pixel size are described in the title of the GIS file. Digital elevation models (DEMs) are geospatial datasets that contain elevation values sampled according to a regularly spaced rectangular grid [10]. A DEM can be stored in several different formats, such as a GIS file. Figure 2.1 shows a sample DEM (Digital Elevation Model) rasters in GIS that present the elevations of terrain.

```
NCOLS 471
NROWS 482
XLLCORNER 313420.000000
YLLCORNER 5632511.000000
CELLSIZE 10.000000
NODATA_VALUE -99999.000000
 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000
509.4391 509.4645 509.7461 509.8247 509.4214 509.0347 509.1991 509.4515 509.4814 509.0794 508.4984 508.4500 5
508.7896 509.0836 508.9834 508.6675 508.3659 508.5509 508.7933 508.7666 508.8533 508.9408 509.0728 509.0650 5
509.8803 510.0865 510.2441 510.1775 509.6836 509.6462 509.7166 509.7319 509.7997 509.7808 509.6991 -99999.00
-99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000
509.7859 509.9478 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999
-99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000
-99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000
-99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000
-99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000 -99999.0000
```

Figure 2.1: Sample GIS file: basin5.asc.

WDPM needs a DEM file and a "water" file as inputs that show the ground elevation and water depth separately. The combination of these two files roughly draws the water distribution of a prairie area. In Figure 2.2, yellow is the ground area, blue is the water coverage area, and black is the non-observation area. The non-observation area refers to the points for which no ground elevation is given and to which no water is added.

The study was conducted in the Smith Creek Research Basin (SCRB), located in southeastern Saskatchewan, Canada, approximately 60 km southeast of Yorkton, SK. The basin area is 393.4 km^2 and is relatively flat with slopes fluctuating from 2% to 5% and elevation ranging from 490 m to 548 m [11]. There are five sub-basins (from "basin1" through "basin5") in this area. The thesis mainly focuses on three sub-basins of different sizes ("basin1", "basin4", and "basin5"). The sub-basin names and the system scales are shown in

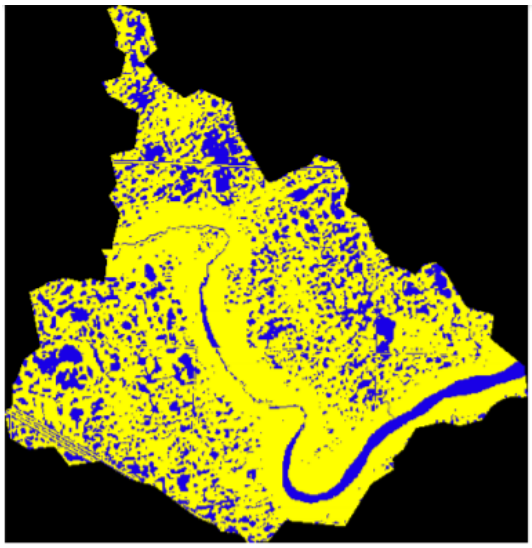

Figure 2.2: Input visualization

Table 2.1. The "smithcreek_dem1m_sb5.asc" is the same area as the "basin5.asc" but with higher resolution. Also, a DEM system outside the SCRB, which is larger than all the sub-domains in the SCRB, is introduced to provide a large enough experiment sample. Thus, the performance efficiency of the new implementation can be better investigated.

Table 2.1: Input System

System	Size
basin5.asc	482×471
basin4_5m.asc	2520×1833
culvert_basin1_5m.asc	3794×3986
smithcreek_dem1m_sb5.asc	4712×4826
patched.asc	5877×5519

2.2 Wetland DEM Ponding model

The purpose of the WDPM is to model the spatial distribution of runoff water on the Canadian Prairies based on an input DEM system [11], where the DEM is a 3D computer graphics representation of elevation data to represent terrain, commonly of a planet (e.g., Earth). Because of the recent post-glacial history, the Canadian Prairies do not have a conventional drainage system [2]. When excess water runs off the landscape, generally due to snow-melt in the spring, it may be trapped in surface depressions ranging in size from puddles to permanent wetlands and may cause local flooding [2]. The WDPM simulates this precipitation, snow melting, evaporation, and drainage system, and it produces the final water distribution map as the

output.

The WDPM has also been introduced to the Land and Infrastructure Resiliency Assessment (LIRA) project to provide improved flood hazard information for Prairie landscapes and the application of the WDPM to LiDAR (Light Detection and Ranging) DEMs have been particularly useful for Prairie landscapes where filling of wetlands is a dominant factor contributing to flooding [12]. The accuracy of spatially distributed runoff information has been verified against ground and aerial photographs, remote sensing imagery, and most importantly community stakeholder experience [13]. By now, at least four LIRA case studies have been successfully completed that show the value of this simple, spatially focused approach to assessing flood hazards across wetland dominated landscapes [13].

There are three modules in WDPM ("add", "subtract", and "drain"). The combination of these modules simulates how water routes in a prairie after rainfall or snow-melt. The input of the WDPM is a rectangle raster system (GIS file), and the elevations of the territory are written in the grid points of the system. The non-observation points in the system are given a no-data value. For example, the no-data value of the system shown in Figure 2.1 is -99999. In WDPM, no water depth is added to the non-observation area, and water does not flow to non-observation grid points either. Therefore, the non-observation area plays the role of a dam that prevents water from leaking out of the map.

The "add" module simulates rainfall and snow-melt by adding a specific depth of water to the input DEM system, and a runoff fraction can also be set to simulate water infiltration to the soil. In most cases, the "add" module can simulate this process and produce the final water distribution maps to meet the requirements. A water redistribution algorithm is performed to route the water from high elevation to low elevation, making the water surface smooth in the DEM system. However, in some situations, the "add" module cannot finish the simulation itself. If a stream or river runs through the target territory, the "add" module routes the water to the stream channel. As mentioned before, water does not flow to the non-observation area, which makes the edge of the territory act as a dam. Therefore, the water in the river or stream cannot flow out of the territory, and the water is backed up over the landscape. However, after heavy rainfall, water should flow out of the landscape through the river instead of being stored in it. In this case, the "drain" module always follows after the "add" module to simulate the drain system.

The "drain" module simulates water runoff from a stream or river by removing water on the DEM system from the lowest point at each iteration. In this way, it can gradually drain backed-up water away [2]. Therefore, the combination of the "add" and the "drain" modules achieves a more realistic simulation. However, the "drain" module needs to move a large volume of water over a long distance, so this module usually takes the longest time to execute.

The "subtract" module simulates water evaporation by subtracting a specific depth of water. This module typically takes the shortest time to finish because a minimal spatial redistribution of water is usually required. It is worth mentioning that the "subtract" module is not the reverse of the "add" module. In the WDPM, each module is performed sequentially. When performing the "add" module, a specific depth of water is

added to all the cells in the input system (except the non-observation points) at first, and an initial system is produced. Then the WDPM iterates to redistribute the water until the water surface is smooth. Finally, an output system is produced. In this case, performing the "subtract" module removes a specific depth of water from the output system of the previous "add" module, not the initial system. So for example, if 10 mm of water is added to a DEM in the "add" module, the original water file cannot be obtained by deleting 10 mm of water with the "subtract" module.

2.3 Relevant Hydrological Models

2.3.1 D8 Algorithm

The D8 method is widely used and implemented in many GIS software packages [14]. It assumes that the water in a cell can only flow into the eight adjacent cells. It uses the steepest slope method to determine the direction of water flow. For example, on a 3×3 DEM grid, it calculates the distance drop between the center grid point and each adjacent grid point. The grid point with the largest distance drop is the outflow grid point. In practice, for the neighbor cells with direction coding 1, 4, 16, and 64, the water distance drop is equal to the water level difference between the center cell and the neighbor cell. For the neighbor cells with direction coding 2, 8, 32, and 128, the water distance drop is computed by the water level difference divided by $\sqrt{2}$. The principle of the so-called steepest gradient method assumes that the surface is impervious to water, and the rainfall is uniform [15]. Then, the water on each grid point always flows to the lowest place.

At present, one of the most widely used technology is watershed feature extraction based on flow direction analysis and confluence analysis. Jenson and Domingue [16] designed an algorithm utilizing this technique. The algorithm includes three processes: flow direction analysis, confluence analysis, and watershed feature extraction.

Flow direction analysis determines the water flow direction for each cell. It uses eight numbers to represent the direction. So given a water level matrix, there is a corresponding direction matrix shown in Figure 2.3 [17].

In this step, there are some special situations that need to be considered, for example, "depression" and "flat" in the DEM. A "depression" means that the elevation of a certain cell is lower than that of its neighboring cells. This phenomenon happens when the river valley width is smaller than the size of one pixel; this commonly occurs in the basin's upper reaches. "Flat" ground means that the adjacent eight units have the same elevation. This situation happens because the size of the pixel is too small or the terrain of the area is flat. These two phenomena are quite common in DEMs. Before the flow direction analysis, the DEM needs to be filled to change the "depression" to "flat" and determine the "flat" flow direction by a complex set of iterative algorithms [16]. This thesis does not study this algorithm in-depth because the Shapiro and Westervelt Algorithm (SW algorithm) is used in the WDPM. Unlike D8 drainage, the SW algorithm is iterative, and this is more suitable for parallel programming [11]. More information about the D8 algorithm

can be found in [16].

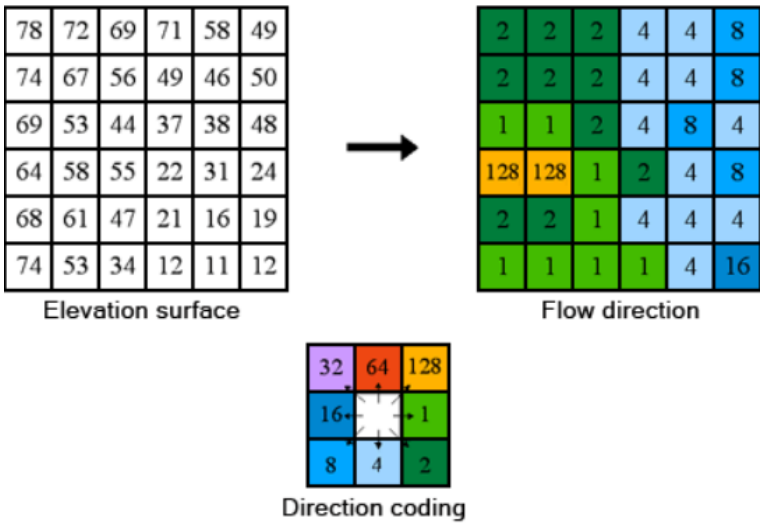

Figure 2.3: D8 algorithm

2.3.2 Shapiro and Westervelt Algorithm

The WDPM is a fully distributed model of wetland storage and runoff that was described by Shook and Pomeroy [18]. The model finds the final spatial distribution of excess precipitation (water) evenly applied over a LiDAR-based DEM using the iterative algorithm of Shapiro and Westervelt [19]. Different from the D8 algorithm, the SW algorithm allows water in the center cell to flow to its eight neighbors. Figure 2.4 visually shows this water redistribution method, where "DEM" refers to the ground elevation and "WATER" refers to the water depth.

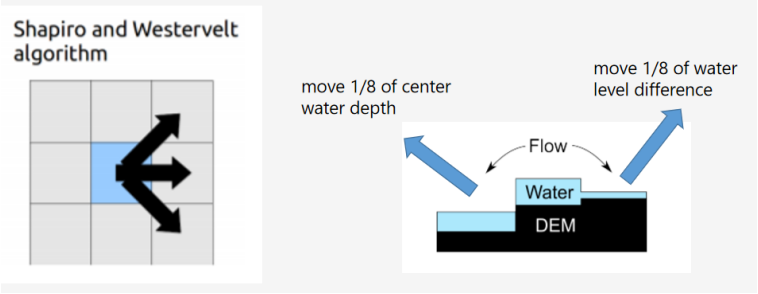

Figure 2.4: SW algorithm [2]

Using the notation shown in Table 2.2, the SW algorithm is described in Algorithm 1. As shown in Figure 2.2, there is non-observation area in the input system that is outside the Smith Creek region. The WDPM does not do any computation on non-observation points. So if a center cell is a non-observation point, the WDPM skips this point and computes the next point. If a neighbor cell is a non-observation point, the SW algorithm does not move water from the center cell to it.

At each iteration, the algorithm is imperfect because the depth of water transferred may result in an inaccurate representation of the final water surface. However, over thousands of iterations, the movement of water results in a realistic water surface [2]. In WDPM, the user-specified elevation tolerance determines the degree of convergence of the simulation. The algorithm runs 1000 iterations without interruption. After that, it calculates the maximum water level difference between two consecutive iterations. If this value is larger than the elevation tolerance set by the user, it moves on to the following 1000 iterations. The termination of the algorithm occurs when the maximum change of the water level is smaller than the elevation tolerance [12]. Compared with the water redistribution method used by other hydrology applications, the SW algorithm does not have strong competitiveness in terms of computational efficiency because the water level changes are small at each iteration, resulting in slow convergence. A significant advantage of WDPM is it allows out-of-order calculations, and this is a good feature for parallel computing.

Table 2.2: Notations in the Algorithm

Notation	Meaning
N_0	The center cell
N_i	The eight neighbor cells
E_{wc}	The center cell water elevation
E_{wni}	The neighbor cell water elevation
E_{dc}	The center cell DEM elevation
D_{wc}	The center cell water depth

Algorithm 1 SW algorithm

if $N_0 \neq missingvalue$ **then**
  for N_i from N_1 to N_8 **do**
    if $(E_{wc} > E_{wni})$ AND $(N_i \neq missingvalue)$ **then**
      if $E_{dc} > E_{wni}$ **then**
        move $\frac{1}{8}D_{wc}$ to N_i
      else
        move $\frac{1}{8}(E_{wc} - E_{wn})$ to N_i
      end if
    end if
  end for
end if

2.4 GPU Computing with OpenCL

C

2.4.1 GPU Computing

In GPU computing, the CPU offloads some of the compute-intensive and time-consuming portions of the code to the GPU. In contrast, the rest of the application still runs on the CPU. The application runs faster from a user's perspective because it uses the GPU's massively parallel computing power to boost performance. A typical contemporary CPU consists of four to eight CPU cores, whereas a typical contemporary GPU consists of thousands of smaller cores. This massively parallel architecture is what gives the GPU its high compute performance [5].

In this thesis, most of the GPU programming and simulations are completed on the Cedar compute cluster of Compute Canada that is equipped with 4 NVIDIA Tesla P100 GPUs. By introducing the specification of this GPU product (shown in Table 2.4), the hardware architecture and the theoretical peak performance of the GPU are further discussed.

Figures 2.5 is a diagram of a Stream Multiprocessor (SM) in the Tesla P100. SMs are the part of the GPU that runs the CUDA kernels. Each SM contains thousands of registers that can be partitioned among threads of execution and several caches that provide shared memory for fast access between GPU threads. Thousands of computation kernels (which do integer and floating-point computation), LDUs (Load-Store Units), and SFUs (Special-Function Units) are also included. The Texture / Processor Cluster is a group made up of several SMs, a texture unit, and some logic controls. The Tesla P100 has 3584 shared FP32 / INT32 CUDA cores and 1792 FP64 CUDA cores in total.

Table 2.3: Abbreviation in the Equation

IPC	instructions per cycle
TPC	transfers per cycle
IPS	instructions per second
FPI	floating-point operations per instruction

Then according to the abbreviations defined in Table 2.3, the data list in Table 2.4, and the relevant formula (described in Chapter C-2 in [20]), the peak FP64 instruction throughput of the Tesla P100 is

$$1 \text{ IPC} \times 1480 \text{ MHz} \times 1792 \text{ cores} \approx 2.68 \times 10^{12} \text{ IPS}. \tag{2.1}$$

Because two FLOPs are processed in one instruction, the peak performance of FP64 can be computed as

$$2.68 \times 10^{12} \text{ IPS} \times 2 \text{ FPI} = 5.36 \text{ TFLOPs}. \tag{2.2}$$

Finally, the peak memory bandwidth for the Tesla P100 can be computed as

$$2 \text{ TPC} \times 715 \text{ MHz} \times 4096 \text{ bits} \times \frac{1}{8} \text{ bytes/bit} \approx 732.16 \text{ GB/s}. \tag{2.3}$$

Nowadays, there are many APIs and development tools for GPU programming. CUDA is a parallel computing platform and programming model that NVIDIA developed for general computing on its own GPUs, the NVIDIA GPU, that is currently the GPU market leader. CUDA is a programmer-friendly GPU programming tool that results in a relatively short development cycle in the industrial environment (compared with OpenCL). Also relevant studies have proved that CUDA has a better performance compared with other APIs. Kamran Karimi and Neil G. Dickson [21] compare the performance of OpenCL and CUDA when working with a Monte Carlo simulation. They do the comparison by timing the kernel execution time and data transfer time. The conclusion is that CUDA is 13% – 63% faster than OpenCL in kernel execution and 16% – 67% in data transfer. Tetsuya Hoshino and Naoya Maruyama [22] compare the performance of CUDA and OpenACC. By doing experiments on fluid dynamics applications, the performance of OpenACC shows approximately 50% lower than CUDA. CUDA and NVIDIA GPUs have been adopted in many areas that need high floating-point computing performance, such as computational finance, climate, ocean modeling, data science, deep learning, medical imaging, etc. The most significant limitation of CUDA is that it only supports NVIDIA GPU. However, in the first quarter of 2020, NVIDIA takes up about 69.19% of the GPU market share, while the remaining 30.81% is taken by AMD [23]. So CUDA is currently the most popular GPU API.

Table 2.4: The Specifications of the NVIDIA Tesla P100 [1]

SMs	56
FP32 CUDA Cores / SM	64
FP32 CUDA Cores / GPU	3584
FP64 CUDA Cores / SM	32
FP64 CUDA Cores / GPU	1792
Memory clock	715 MHz
GPU Boost Clock	1480 MHz
Peak FP32 GFLOPs	10600
Peak FP64 GFLOPs	5300
Memory Size	16 GB
Memory Interface	4096-bit HBM2
L2 Cache Size	4096 KB
Register File Size (SM)	256 KB

OpenCL (Open Computing Language) is another widely used GPU API. Compared with CUDA, the most significant advantage of OpenCL is its cross-platform capabilities that are an industry-standard framework

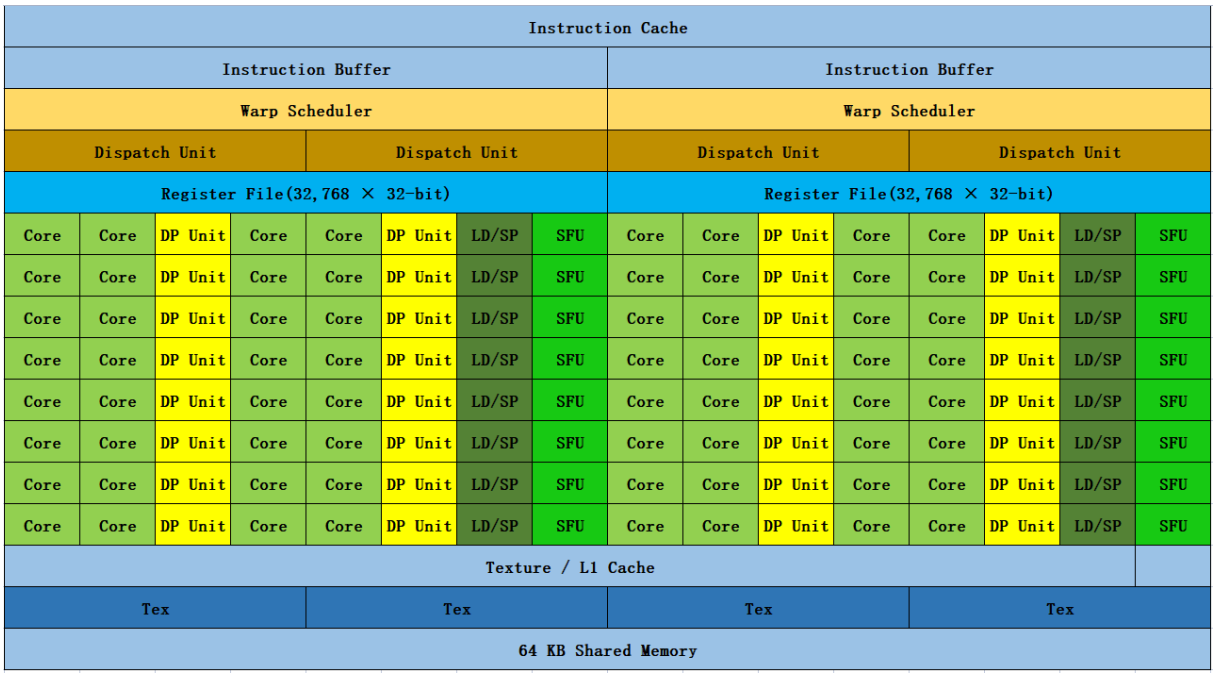

Figure 2.5: Pascal GP100 SM Unit

for programming computers composed of a combination of CPUs, GPUs, FPGAs, and other processors [24]. These so-called heterogeneous systems have become an essential class of platforms, and OpenCL is the first industry-standard that directly addresses their needs [24]. With OpenCL, a single program can run on a wide range of systems, from cell phones to nodes in massive supercomputers. No other parallel programming standard has such a broad reach. To some extent, OpenCL is a low-level language because it provides little abstraction of programming concepts and is close to actual machine instructions. Many processes have to be completed by the programmer manually, such as device detecting, memory allocating, and I/O controlling. The choice is made in this thesis to use OpenCL because it is desired to make WDPM apply to all potential users who might be using an AMD GPU or do not even have access to a GPU and would like to run it faster with parallel CPUs. Chapter 5 discusses the future work on WDPM implementation on FPGA that would fully take advantage of the portability of OpenCL.

There are many other GPU programming APIs. OpenMP is mainly used in the system with shared memory and CPU parallel computing. C++ Accelerated Massive Parallelism (C++ AMP) accelerates execution of C++ code by taking advantage of data-parallel hardware such as a GPU on a discrete graphics card [25]. However, it only works with the Windows system. OpenACC is another popular GPU programming API. OpenACC intends to simplify the parallel programming of heterogeneous computing (CPU/GPU) systems [26]. Programmers can add comments to C, C++, and Fortran source code to indicate the code segments that need to be accelerated by compiling instructions or other functions. OpenACC is similar to OpenMP in writing style and is easy to get started. However, the relatively poor performance compared to the performance of pure CUDA code [22] is the main disadvantage of OpenACC.

2.4.2 OpenCL

Table 2.5: Part of the Device Information

Platform Name	NVIDIA CUDA
Platform Version	OpenCL 1.2 CUDA
Number of Device	1
Device Name	Quadro P400
Device Type	GPU
Max Clock Frequency	1252 MHz
Max Dimension	3
Max Work Item Size	1024×1024×64
Max Work Group Size	1024

An OpenCL program is composed of a host function that runs on a CPU and one or several kernel functions that run on the OpenCL devices. A host function is the outer control logic that performs the configuration for a GPU-based application. It mainly detects the OpenCL platform, configures the OpenCL environment, compiles the kernel functions, and controls the data flow. Kernel functions run on powerful computing devices that work on the most computationally intensive part of an application.

Figure 2.6 shows the programming flow of OpenCL. The OpenCL platform model defines a high-level representation of any heterogeneous platform used with OpenCL, and a set of computing devices that an application uses to execute code are associated with each platform. **clinfo** is a simple command-line application that enumerates all possible (known) properties of the OpenCL platform and devices available on the system. Inspired by the AMD program of the same name, it is coded in the C programming language. It tries to output all possible information, including that which is provided by platform-specific extensions while trying not to crash on unsupported properties. Table 2.5 show part of the device information of a machine that is equipped with NVIDIA Quadro P400. According to Table 2.5, when working on a two-dimensional system, the maximum number of work-items can be computed is 1024×1024. Contexts are used to manage objects such as command-queues, memory, programs, and kernel objects. Multiple devices can be associated with one context. Kernel functions are compiled inside the host function by building the program. Then operations like executing kernel and data transferring between device and host are performed under the instruction of the command queue.

Similar to the other GPU programming APIs, OpenCL organizes the computing units hierarchically. The qualifier of the kernel variables can be declared to determine the storage location of them. The **private** variables are stored in a work-item's memory, which is similar to a register. It is always defined in the kernel function and can only be read and written by one work-item. The **local** variable stores in the global memory

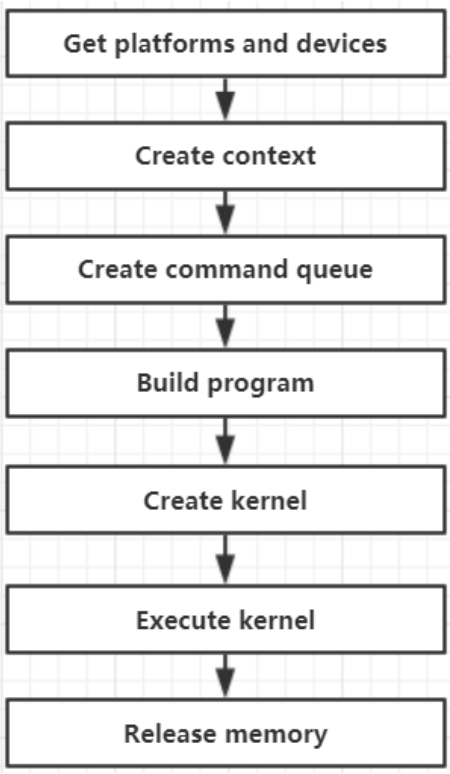

Figure 2.6: OpenCL programming flow

of a work-group. GPU threads are grouped into fixed-size SIMD (single-instruction, multiple-data) warps that are called work-groups in OpenCL. So local memory can provide fast read-write access to the local variables for the work-items in a work-group. Finally, in the device global memory, the readable and writable **global** variables and the read-only **constant** variables are stored [3]. When working on a two-dimensional matrix, data are mapped in the device as shown in Figure 2.7. All of these work items can be executed in parallel. When programming the kernel function, "global work id" and "local work id" are used to target the specific work item. Finally, Figure 2.8 shows a complete memory hierarchy of multiple devices.

2.4.3 Pipe API in OpenCL

As discussed before, the strong cross-platform capability is one of the biggest advantages of OpenCL. The field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence the term "field-programmable". Usually, an FPGA is programmed with a hardware description language (HDL). The C-programming-language-based OpenCL makes it easier to program the FPGA. The pipeline API concept has been introduced to OpenCL GPU computing since OpenCL 2.0 [27]. But it was already supported on FPGA in the previous OpenCL version. Figure 2.9 shows the different parallelism strategies between GPU and FPGA.

The critical difference between kernel execution on GPUs versus FPGAs is how parallelism is handled.

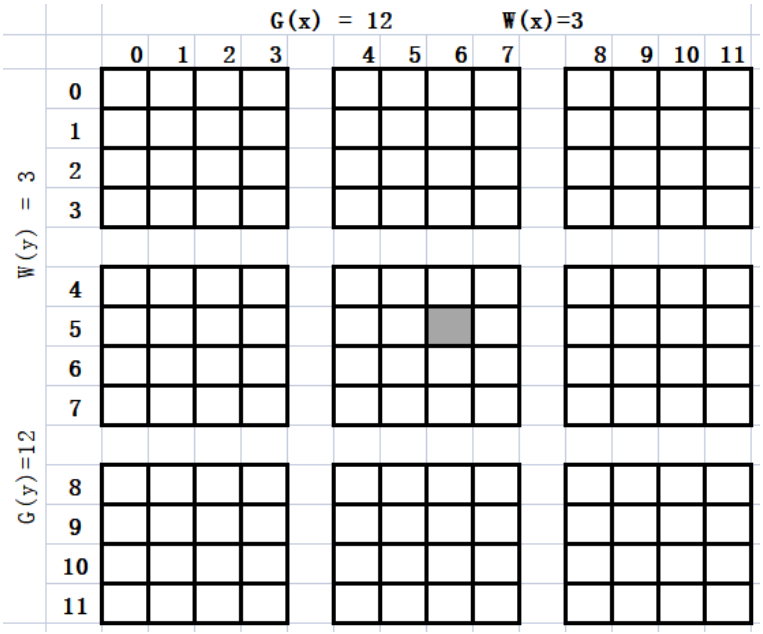

Figure 2.7: An example of how the global IDs, local IDs, and work-group indices are related for a two-dimensional NDRange. Other parameters of the index space are defined in the figure. The shaded block has a global ID of $(g_x, g_y) = (6, 5)$ and a work-group plus local ID of $(w_x, w_y) = (1, 1)$ and $(l_x, l_y) = (2, 1)$. [3]

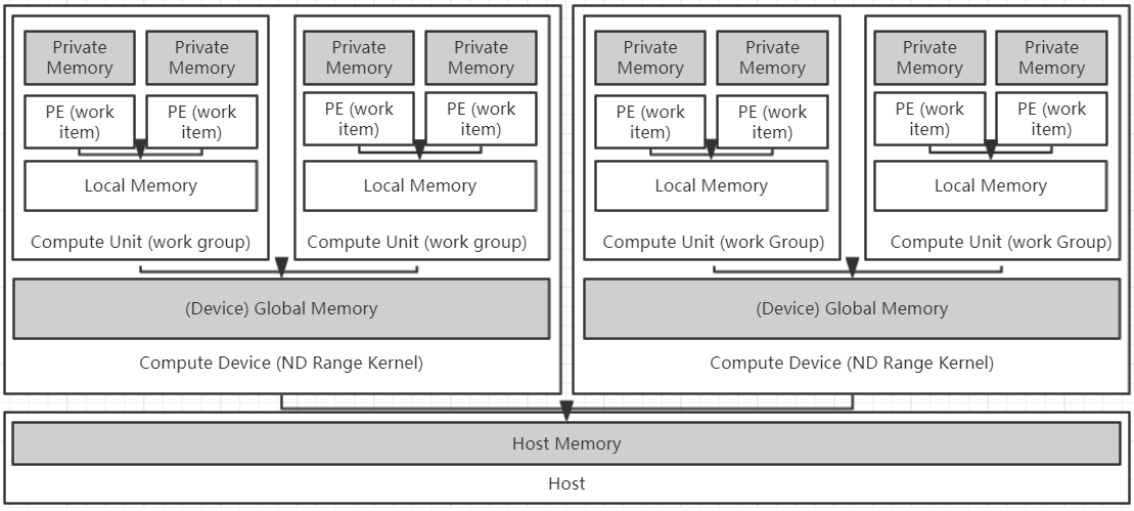

Figure 2.8: Memory hierarchy

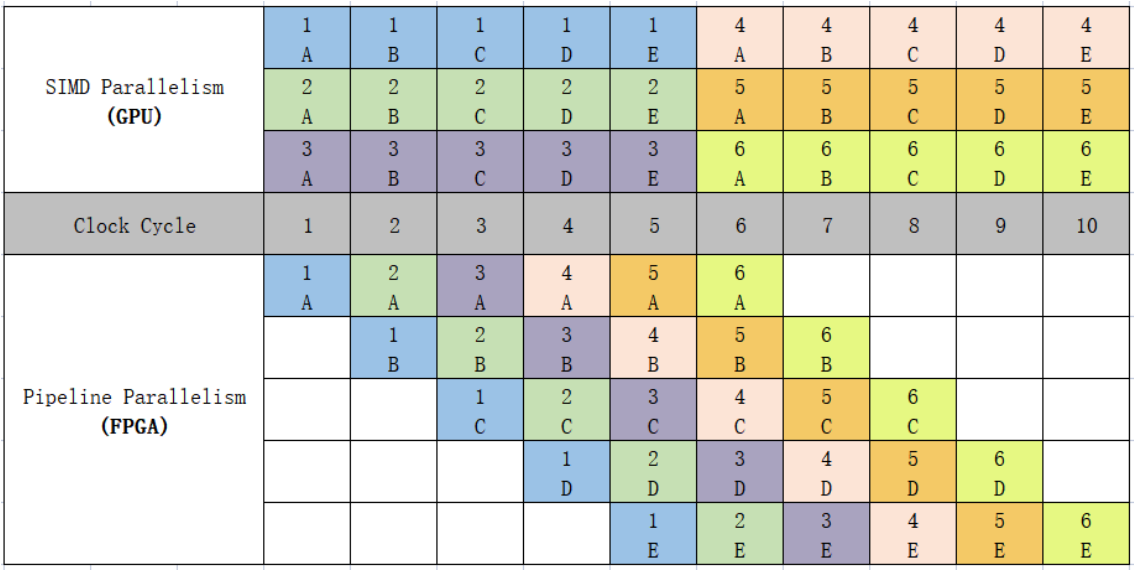

Figure 2.9: Comparison of SIMD parallelism versus pipeline parallelism

GPUs are SIMD devices, where groups of processing elements perform the same operation on their individual work-items. On the other hand, FPGAs exploit pipeline parallelism, where different stages of the instructions are applied to different work-items concurrently. [28]

According to Figure 2.9, the two parallelism strategies could achieve a similar performance when the kernel is simple. But in some cases, like when handling branching conditions, the FPGA pipeline could show its advantages. An SIMD unit (GPU) operates a single instruction at a time; all code-paths taken by the individual work-items must be executed one after another, with individual work-items disabled or enabled based upon how they evaluated the branching condition. As a result, encountering a branching condition with N options could potentially result in execution time equal to the sum of execution times for all N options (for N up to the SIMD width). Branching is less of an issue on FPGAs because all code-paths are already established in hardware. All branch options can be executed concurrently or even computed speculatively in some cases to allow overlap with a branch condition computation [28]. Figure 2.10 shows this process.

FPGA parallel computing is not the focus of this thesis. More information can be found in the paper of Zeke Wang and Bingsheng He [29] that studies how OpenCL applications are optimized on FPGA by applying pipeline and the paper of Ahmed Sanaullah and Martin C. Herbordt [30] that achieves high speedups for 3D Fast Fourier Transforms (FFTs) with FPGA OpenCL.

2.4.4 Overlapping Communication with Computation

For data parallel applications, effective use of accelerators is indispensable for higher system performance and energy efficiency. Communication management between multiple heterogeneous devices is one of the most challenging problems because host CPUs and accelerators generally have disjoint memory spaces. In particular, large-scale super-computing applications suffer from this issue either because the data do not fit

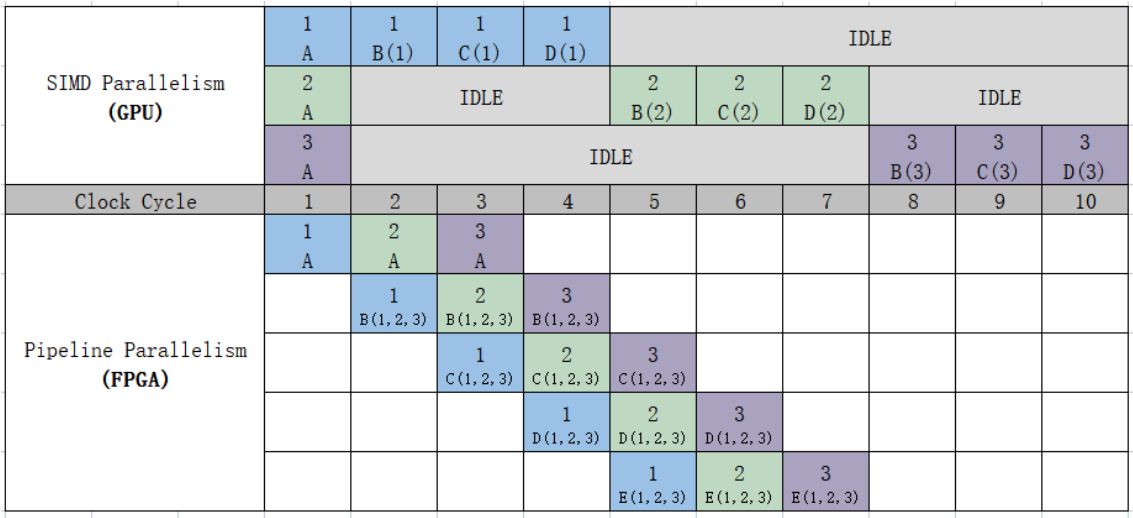

Figure 2.10: Comparison of SIMD parallelism versus pipeline parallelism

into the GPU memory, inter-node communication at the end of each time step requires data to be present at the host, or parts of the application are executed on the CPU. Moreover, because super-computing systems are large investments, even a 10-percent speedup can often save millions of dollars. So it is especially important to save this extra time. To be specific, an appropriate technique should be applied to partially or entirely hide communication delays behind the heterogeneous devices computation. Here, two techniques are introduced that are relevant to the WDPM.

Loop indexing: In domain decomposition applications, the domain is usually decomposed into inner and outer regions [31]. Boundary data to be communicated may be produced earlier by updating the outer regions first followed by the inner regions. This technique requires rearrangement of the loop index. A data dependency analysis is required in order to ensure correctness.

Loop distribution: This technique separates independent computation from dependent computation in a single loop into multiple loops [31].

As mentioned before, in the WDPM, the SW algorithm reads and writes to a 3×3 matrix when working on a water redistribution component. So each point in the system is interdependent on the surrounding points. When doing domain decomposition on the input system, boundary data cannot be extracted directly. The idea is to create the outer part of the domain that includes the boundary data by applying loop distribution. Then loop indexing can be used to make sure the outer part finishes computation before the inner part. The detailed parallel programming method and the implementation are discussed in Chapter 3.

Chapter 3

Implementation

3.1 Previous Work

Originally, WDPM was written in Fortran using OpenMP and can run in parallel with multiple CPU cores. Then WDPM was translated to the C programming language, and the parallel computation was implemented in OpenCL to improve the speed and responsiveness and utilize the GPU.

Before further introducing the parallel algorithm, pre-processing for the input system needs to be done. According to the SW algorithm 1, each water redistribution component covers a 3×3 matrix. To make sure each cell in the DEM has eight neighbors, WDPM extends the input matrix as in Figure 3.1. The additional points are defined as the non-observation points, and they are given a no-data value.

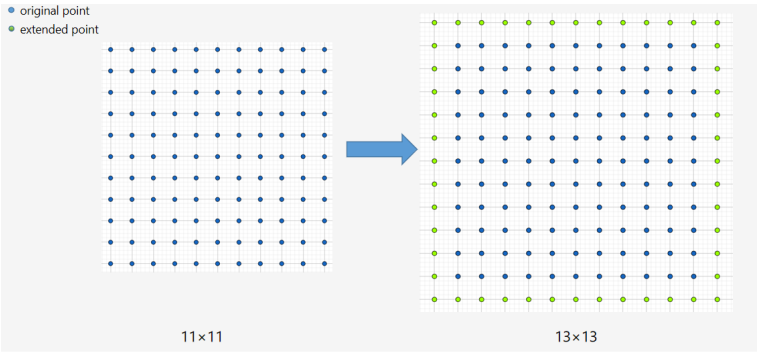

Figure 3.1: Input matrix pre-processing

Before discussing the parallel algorithms, Table 3.1 shows the meaning and the values of the variables used in the algorithms.

Algorithm 2 Parallel CPU implementation

```
continue = true
while continue = true do
  for i = 1 to N_iter do
    OMP PARALLEL (compute sub-matrices)
    for row from 2 to numrows + 1 do
      for col from startcols(thread_num) to endcols(thread_num) do
        if bigwater(row, col) > 0.0 AND bigdem(row, col) > missingvalue then
          do SW algorithm 1
        end if
      end for
    end for
    OMP END PARALLEL
    OMP BARRIER
    OMP PARALLEL (compute the boundaries)
    if numslice > 1 then
      for row from 2 to numrows + 1 do
        for col from boundary_start_col(thread_num) to boundary_end_col(thread_num) do
          if bigwater(row, col) > 0.0 AND bigdem(row, col) > missingvalue then
            do SW algorithm 1
          end if
        end for
      end for
    end if
    OMP END PARALLEL
  end for
  if The maximum water elevation change is smaller than the tolerance then
    continue = false
  end if
end while
```

Table 3.1: Variables in the Algorithms

Variable	Meaning	Value in practice
N_{iter}	the number of iterations	1000
S_g	the size of sliding grid	3
N_d	the number of GPUs	-
i_d	the index of the GPU	-
N_r	the number of rows	-
N_{bc}	the number of boundary columns	-

3.1.1 Parallel Programming on one CPU with OpenMP

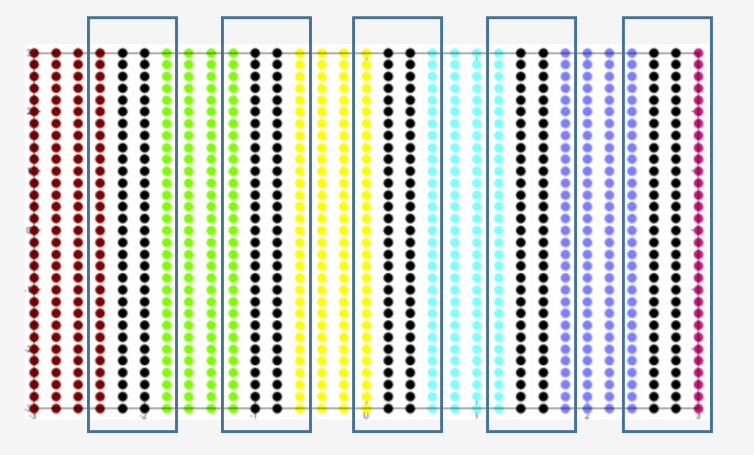

Figure 3.2: CPU parallel computing

The original WDPM Fortran code uses the algorithm that divides the water matrix into slices, and each slice is assigned to a separate thread. Because the SW algorithm works on a 3×3 matrix each time, two columns are maintained between every two slices in order to prevent the boundary data from being written to by two CPU processors simultaneously. In Figure 3.2, all the colored matrices (excluding black) are computed in parallel, and the points with the same color are computed by the same CPU processor. After all slices finish computing, the algorithm then handles the regions in between the slices that are called the "boundaries" (marked by the blue boxes in Figure 3.2).

The CPU parallel implementation is given in Algorithm 2.

3.1.2 Parallel Programming on one GPU with OpenCL

According to the previous section, the parallel CPU programming method is quite straightforward. Basically, domain is divided into slices, and each slice is computed with one CPU core. In each slice, a large proportion

of data is independent, in the sense that the slices do not read and write to other slices, so the race condition can be easily solved by separately handling the boundary of each slice. Unlike CPUs, GPUs have thousands of cores. Dividing the domain into thousands of slices is unrealistic. As discussed before, each cell is inter-dependent on its neighbor cells when using the SW algorithm. So performing the algorithm on two adjacent cells simultaneously results in a race condition. In this case, the sliding window method can solve this problem. In the sliding window method, a window of specified size moves over the data, sample by sample, and computations are performed over the data in the window. The output for each input sample is the computational result of the current sample and the previous samples. In the first time step, to compute the first outputs when the window does not have enough data yet, the algorithm fills the window with a non-observational value. In the subsequent time steps, to fill the window, the algorithm uses samples from the previous data frame. In the WDPM, the sliding window size is set to three, and Figure 3.3, Figure 3.4, and Figure 3.5 demonstrate how to use this method. The black cells shown in each figure are computed in parallel, and by looping over the sliding window index i and j from 1 to 3, all the cells in the input system can execute the SW algorithm without race condition.

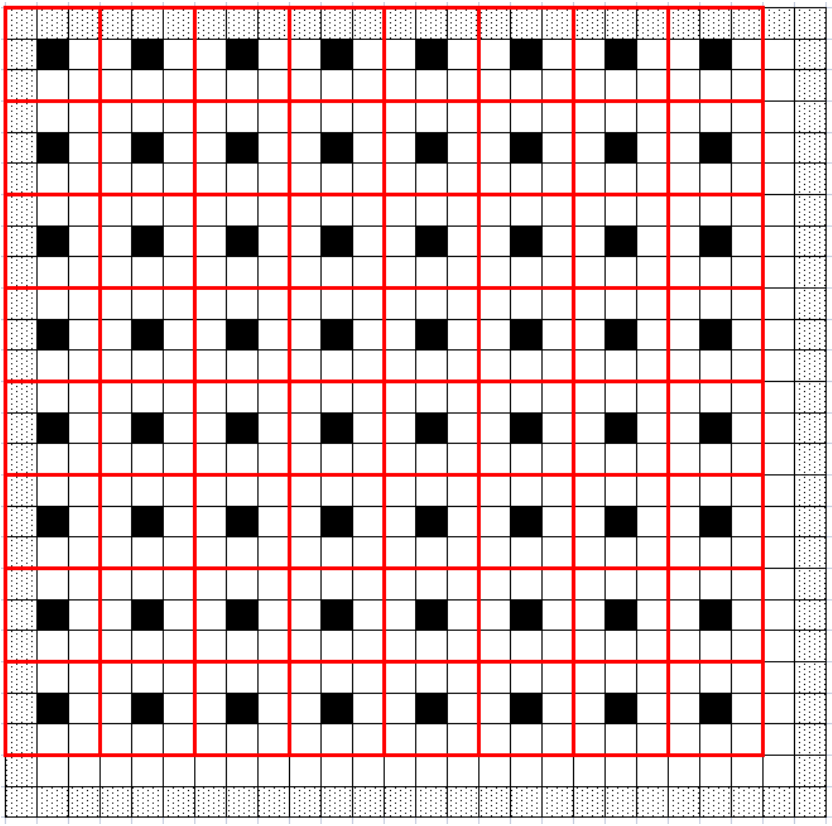

Figure 3.3: The sliding windows when i=1 and j=1

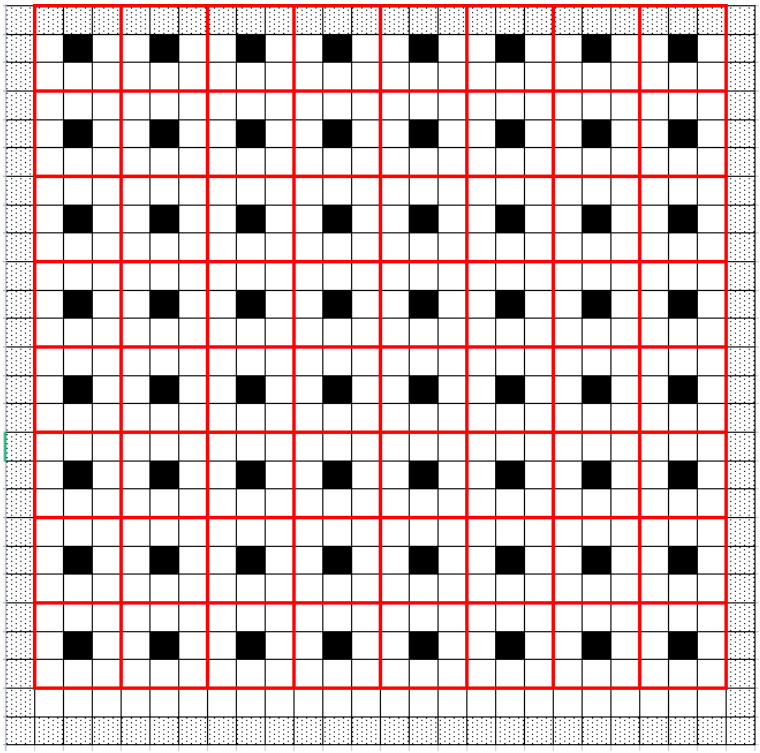

Figure 3.4: The sliding windows when i=1 and j=2

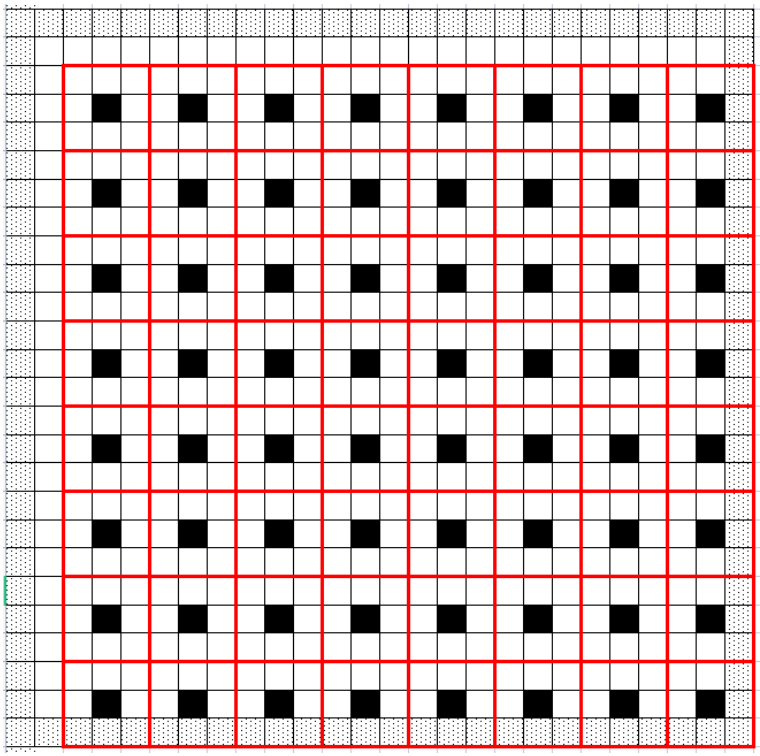

Figure 3.5: The sliding windows when i=3 and j=3

The kernel function that runs on the GPU is given in Algorithm 3, and the host function that runs on the CPU is given in Algorithm 4.

Algorithm 3 Kernel function

$row^{'} = global\_id(0)$
$col^{'} = global\_id(1)$
$row = (j - S_g) + row^{'} \times S_g$
$col = (k - S_g) + col^{'} \times S_g$
if $bigdem(row, col) > missingvalue$ AND $bigwater(row, col) > 0.0$ **then**
    do SW algorithm (Algorithm 1)
end if

Algorithm 4 Host function

continue = true
while continue = true **do**
    for i from 1 to N_{iter} **do**
        for j from 1 to S_g **do**
            for k from 1 to S_g **do**
                Write the j and k to the kernel function
                Execute kernel function (Algorithm 3)
            end for
        end for
    end for
    if The maximum water elevation change < elevation tolerance **then**
        continue = false
    end if
end while

3.2 Multiple-GPU Implementation

3.2.1 Multiple-GPU parallel algorithm

Inspired by the CPU parallel computing method, the domain is divided into slices, and each slice is sent to one GPU. However, the CPU parallel method is based on a shared memory system, where completing data synchronization is relatively easy. Referring to the OpenCL memory hierarchy diagram shown in Figure 2.8, the CPU-GPU architecture is a distributed memory system. When doing data synchronization, the host CPU needs to read data from the GPUs, and the data communication is completed in the host. After that, the

host CPU writes the updated data back to the GPUs, and then the algorithm starts the next iteration. This process is shown as Figure 3.6. Here, D_matrix refers to the sub-matrix computed on the device (GPU), and the H_matrix refers to the boundary part computed on the host (CPU).

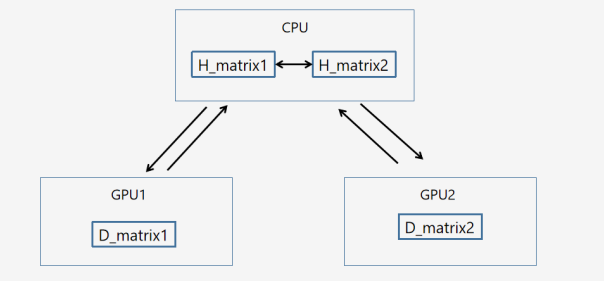

Figure 3.6: Data flow during data synchronization

Data transmission is a time-consuming operation; in particular, WDPM does data synchronization in each iteration, and thousands of iterations need to be done in the whole simulation. By a rough estimate, half of the running time is spent on data synchronization if transferring all of the data between the devices and the host, and this results in a longer running time with two GPUs than with one GPU. Fortunately, OpenCL allows users to manually control the data transmission by modifying some OpenCL function parameters. Furthermore, it is unnecessary to transfer all the data because only the boundary data need to be handled during data synchronization. As a result, the communication cost can be minimized by transferring the boundary data only between host and devices.

OpenCL has built-in support for processing image data (two-dimensional data). Using image objects, image data that resides in host memory can be made available for processing in a kernel executing on an OpenCL device [3]. Image objects simplify the process of representing and accessing image data because they offer native support for a multitude of image formats [3]. However, many OpenCL devices do not support the image feature. To check for image support from the host, the `clGetDeviceInfo` function can be called with the "CL_DEVICE_IMAGE_SUPPORT" option. If the result is "CL_FALSE", the device does not support images [32]. Because the WDPM developers expect the software to work for all OpenCL users, the two-dimensional input system is stored in a one-dimensional buffer. In this case, the data are stored in column-major order (Figure 3.7). The multiple-GPU parallel algorithm is given in Algorithm 5.

3.2.2 Overlapping Communication with Computation

According to Algorithm 5, the boundary data are handled in serial CPU after the GPUs have finished all their computation. The GPUs are idle when the host CPU is working on computing the boundaries. So, if CPU computation and GPU computation can be done simultaneously, the data communication cost can be reduced.

In order to work with the distributed memory of multiple GPUs, data communication requires boundary data to be present at the host. Loop indexing, which has been introduced in Chapter 2, is a suitable technique

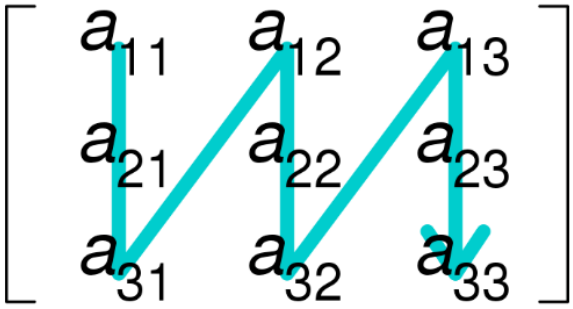

Figure 3.7: 2D to 1D

Algorithm 5 Multiple-GPU parallel programming

```
continue = true
while continue = true do
  for i from 1 to N_iter do
    if i ≠ 1 then
      Write the boundary data from CPU to GPUs
    end if
    for i_device from 0 to N_d − 1 do
      for j from 1 to S_g do
        for k from 1 to S_g do
          Write the j and k to the kernel function
          Execute kernel function (Algorithm 3)
        end for
      end for
    end for
    Block function
    if i ≠ N_iter then
      Read the boundary data from GPUs to CPU
      Do SW algorithm (Algorithm 1) in serial to the boundary data
    end if
  end for
  if The maximum water elevation change < elevation tolerance then
    continue = false
  end if
end while
```

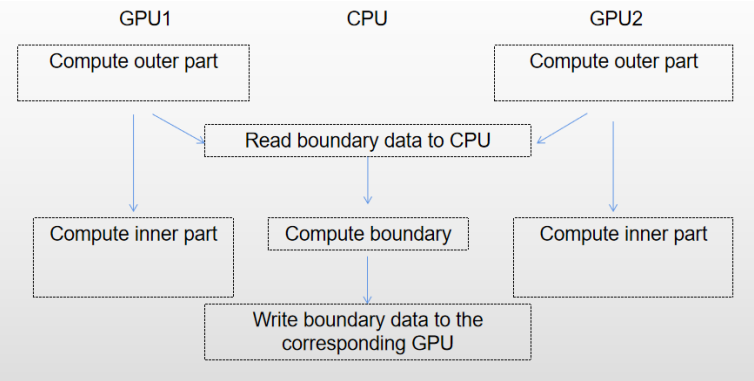

Figure 3.8: Programming flow

to overlap this communication with GPU computation. As discussed previously, loop indexing requires the domain in each GPU to be divided into two parts: the inner part and the outer part. The inner part contains the independent data and does not read and write to the domains in other GPUs, whereas the outer part includes the boundaries. By managing the loop order, the outer part of the domain is computed in GPU at first, and the host CPU reads the boundary data back after that. In this way, the inner part of the GPU computation and the boundary part CPU computation can be computed simultaneously. However, because of the event schedule mechanism of OpenCL, the data transfer between CPU and GPUs cannot be overlapped by any computation.

3.2.3 Optimization Algorithm

Without loss of generality, consider a system with three GPUs. One sub-domain is sent respectively to each of the three GPUs.

Figure 3.9 shows the outer parts (marked in black) of the sub-domains. In one iteration, GPUs first compute these outer parts. So that after this step, the boundary columns (8, 9, 10, 11 and 18, 19, 20, 21) are not read or written any more. Then, the GPU computation is paused, and the host CPU reads the boundary columns back. As shown in Figure 3.10, the boundary columns are merged into boundary matrices that are computed simultaneously with GPU computation of the inner part. At the end of this iteration, the host takes the boundary matrices apart and writes the updated boundary columns back to each sub-domain.

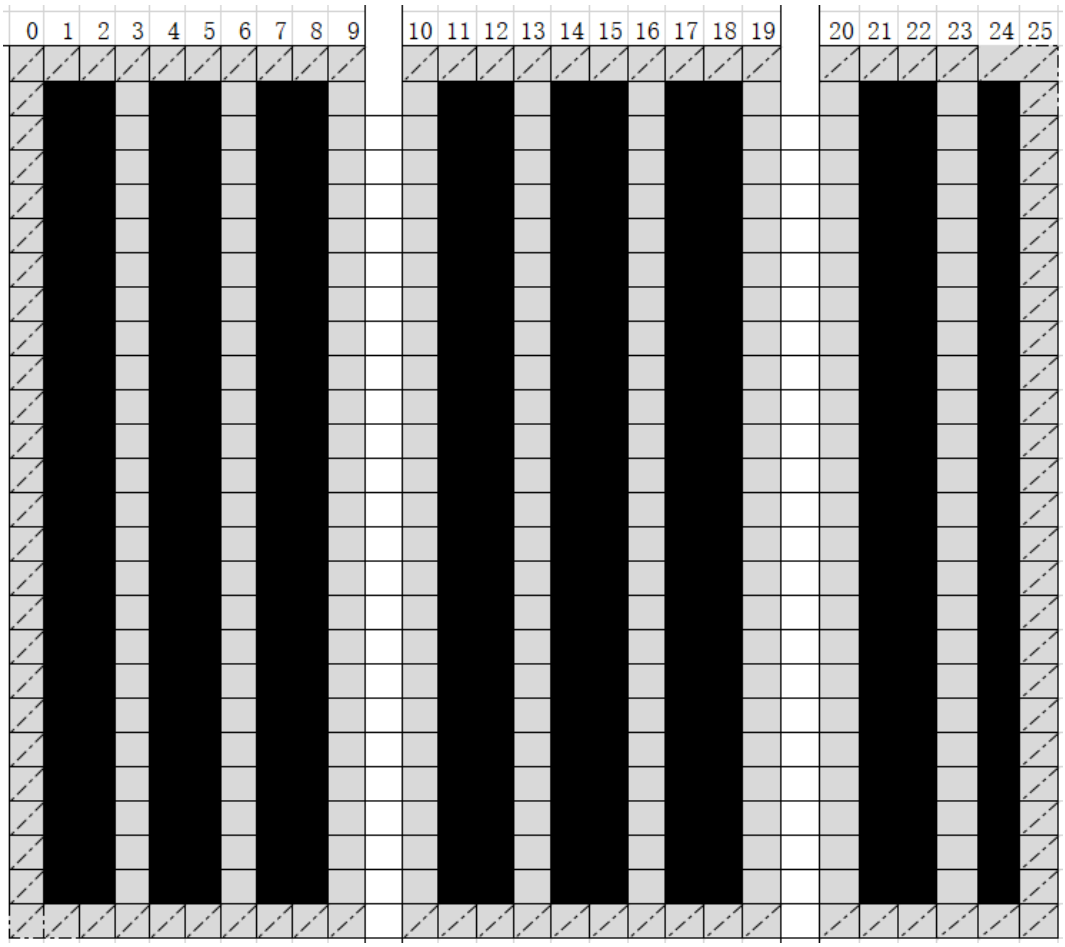

Figure 3.9: Computing the outer part

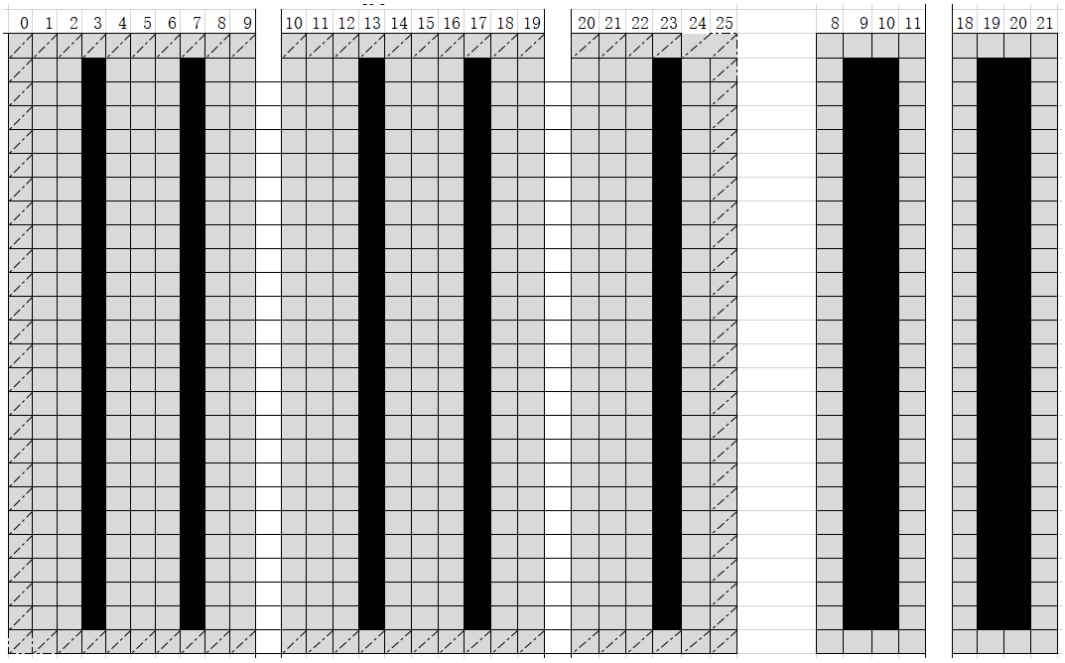

Figure 3.10: Computing the inner part and the boundary

Algorithm 6 Overlapping communication and computation

```
continue = true
while continue = true do
  for i from 1 to N_iter do
    if i ≠ 1 then
      Write the boundary data from CPU to GPUs
    end if
    for i_device from 0 to N_device − 1 do
      for j from 1 to S_g − 1 do
        for k from 1 to S_g do
          Write the j and k to the kernel function
          Execute kernel function (Algorithm 3)
        end for
      end for
    end for
    Block function
    if i ≠ N_iter then
      Read the boundary data from GPUs to CPU
    end if
    for i_d from 0 to N_d − 1 do
      Set k = S_g
      for j from 1 to S_g do
        Write the j and k to the kernel function
        Execute kernel function (Algorithm 3)
      end for
    end for
    for i from 1 to N_r do
      for j from 1 to N_bc do
        Do SW Algorithm 1
      end for
    end for
    Block function
  end for
  if maximum water elevation change < elevation tolerance then
    continue = false
  end if
end while
```

Chapter 4

Experiment and Analysis

4.1 Output Comparison

In this chapter, the sample systems used for the experiments are described in Table 2.1. The WDPM needs a DEM file and a water file as inputs that separately show the ground elevation and water depth, and the WDPM produces the final water distribution map as the output. Before discussing the parallel performance of the WDPM multiple-GPU implementation, the final water distribution compared with running with one GPU is evaluated to ensure the new implementation produces the correct output.

When a simulation is over, WDPM generates a brief output summary. The summary includes the output system information, such as the maximum water depth, the mean water depth, etc. Table 4.1 shows the sample summaries of the three modules (add, subtract, and drain) separately. Then to make sure it is easier to see the output difference between using one GPU and multiple GPUs, a large input value is set, which is adding 300 mm water. The output systems of adding 300 mm water to the DEM and draining the water away from the river using two GPUs are produced to check the output system difference. Figure 4.1 and Figure 4.2 show the water distribution after adding 300 mm water to the domain and then drain the water away from the river when running with two GPUs. In general, the multiple-GPU implementation produces the correct final (steady-state) results, but the intermediate outputs are not exactly the same.

Table 4.1: The Output Summary Comparison

Module	Add		Subtract		Drain	
number of GPUs	1	2	1	2	1	2
Initial volume (m^3)	0.0	0.0	0.0	0.0	110036.0	110036.0
Final volume (m^3)	56550681.0	56550681.0	160215.7	160215.7	97577.6	97577.6
Water coverage	7.4%	7.4%	7.0%	7.0%	10.1%	10.1%
Mean depth (mm)	40.6	40.6	60.3	60.3	87.4	87.4
Max depth (mm)	1791.4	1791.5	1124.9	1124.8	1038.3	1038.2

An explanation for this observation is the computation order. Rotating the input system can change the computation order. With rough experiment, the output summaries of the same system with different

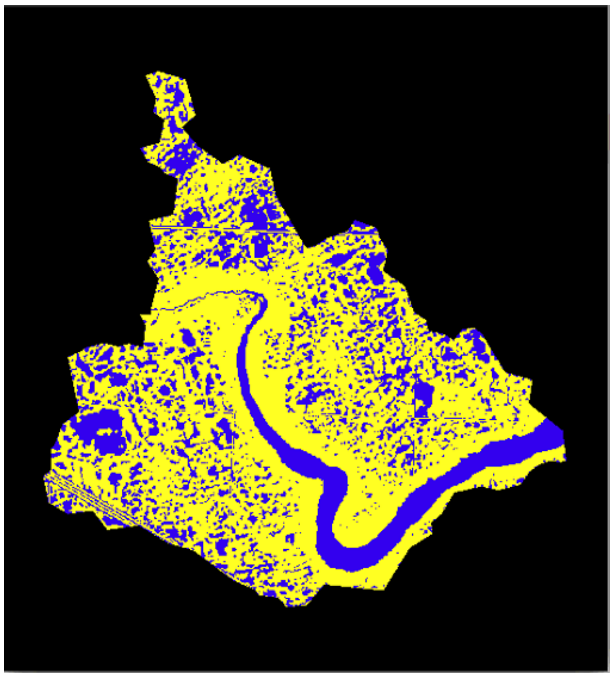

Figure 4.1: The water distribution after adding 300.0 mm water

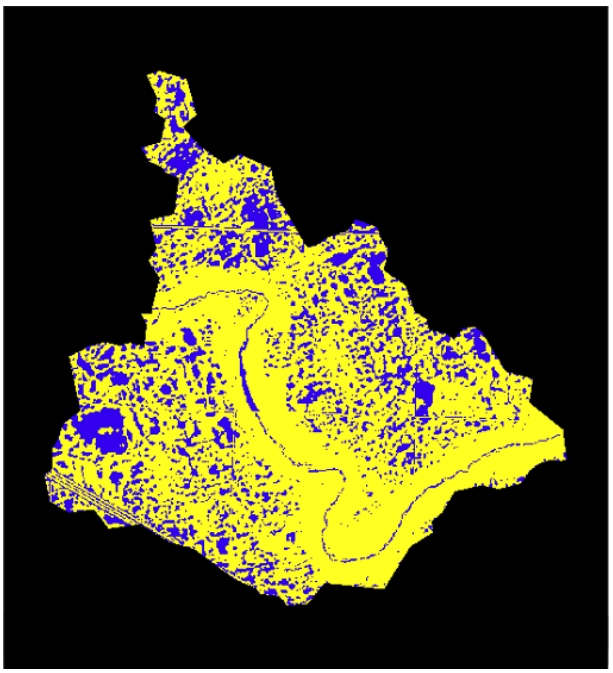

Figure 4.2: The water distribution after draining the water away from the river

rotation angles are different. In this case, a further experiment is introduced to see if the solution convergent as the simulation goes further. The water depth difference of the output system is calculated to check the differences in final water distribution between running with different numbers of GPUs. As described in Section 2.3.2, users need to input an elevation tolerance to determine the degree of the convergence of the simulation. The water surface is close to flat when given a small water elevation tolerance. It also provides a method to verify the convergence of the solution. The test object is "basin5.asc". Running with two GPUs and one GPU, the experiment adds 300 mm depth of water and then drains the water from the system. The water depth errors of all the pixels are calculated, and the error map is plotted (shown as Figure 4.3 and 4.4). A significant solution convergence is shown when using a small water elevation tolerance; the maximum error is about 4×10^{-4} with 1.0 mm tolerance and about 5×10^{-5} with 0.1mm tolerance. It further proves that the multiple-GPU implementation is correct.

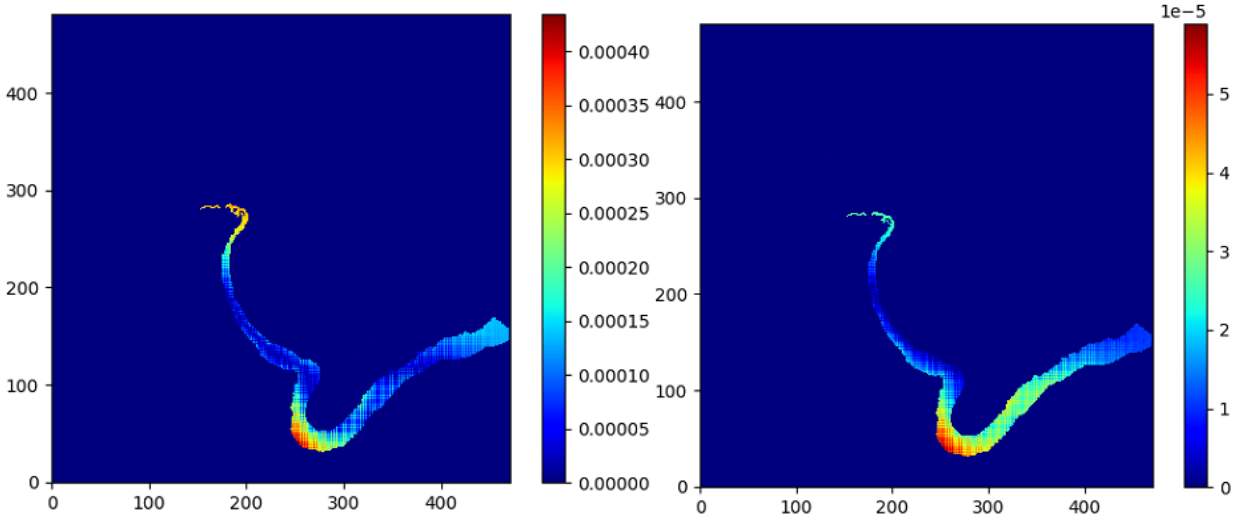

Figure 4.3: The error map when the elevation tolerance is 1 mm

Figure 4.4: The error map when the elevation tolerance is 0.1 mm

4.2 Profiling Results

On Compute Canada, profiling a CUDA program or an OpenACC program is more straightforward than profiling an OpenCL program. Compute Canada provides the PGI profiler that is a powerful and simple tool for analyzing the performance of parallel programs written with OpenMP, MPI, OpenACC, or CUDA [33]. Using the PGI profiler usually consists of two steps: data collection and analysis. Data collection is performed by running the application with the PGI compiler with profiling enabled. The analysis is performed by visualizing the data produced in the first step. Users can save the performance data into a file. Then the performance data file can be uploaded to a performance analysis tool, like Nsight System, to visually investigate bottlenecks and pursue optimizations with a higher probability of performance gains [33]. However, the NVIDIA profiling tool cannot profile the OpenCL program, and at present Compute Canada does not have AMD GPU profiling tools that are better for OpenCL profiling. Fortunately, OpenCL has an event-based profiling tool. It provides this mechanism by having the "cl_event" objects hold timing information. This timing information can be captured using an OpenCL function.

4.2.1 Method

In OpenCL, the "command queue" contains instructions to inform the devices of the "context" (that is, the group of devices that have been chosen for use) to execute a particular command. Basically, the commands include reading data from the device to the host, writing data from the host to the device, and executing the kernel function. In the new implementation, each "command queue" controls one device. OpenCL also has an event scheduler that allows the programmer to target the process associated with one OpenCL command. The event scheduler is used for two purposes. First, the running time of one command can be recorded with the

device time counters associated with the specified event. Second, the blocking functions (`clWaitForEvents`) can be associated with the specified event to stop the execution until one command finishes. In this way, the running time of each segment can be tested.

By using the OpenCL function `clGetEventProfilingInfo`, getting the running time of one OpenCL command is straightforward. As described in Section 3.2.2, the overlapping communication and computation technique is applied to optimize the multiple-GPU parallel algorithm, and the OpenCL profiling function only works for timing an OpenCL command. A rigorous method to test both GPU computing and CPU computing needs to be found. Section 3.2.2 discussed that the sub-domain in each device is divided into the outer part (includes the boundary data) and the inner part (without boundary data). The overlapping technique is applied to the inner part GPU computing and boundary data CPU computing. In practice, the OpenCL blocking function is placed after the CPU computation so that the boundary data computation can start without waiting for the end of the GPU computation. This programming flow modification is shown in Figure 4.5. To verify the effect of the overlapping technique, the position of the blocking function needs to be changed so that the inner part GPU computation and the boundary CPU computation are executed sequentially. In this way, the success of the overlapping technique is verified if the running time of the first programming flow in Figure 4.5 is shorter than the second one.

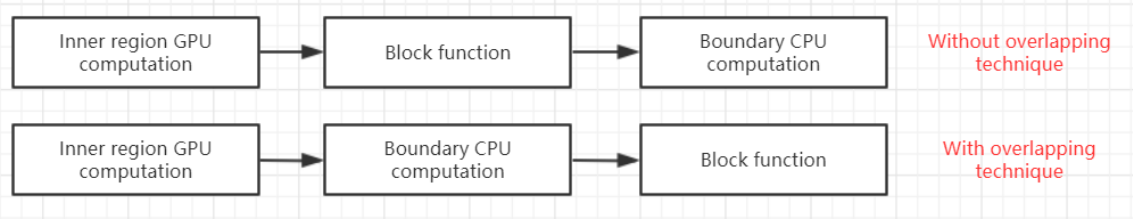

Figure 4.5: The experiment programming flow

4.2.2 Results

As reference, Table 4.2 shows the profiling result of WDPM running with one GPU. Table 4.3, Table 4.4, and Table 4.5 respectively show the profiling results of different systems when running with two GPUs, three GPUs, and four GPUs. As discussed in the previous section, each "event" is associated with a "command queue", and each "command queue" is associated with a GPU. So the running time of each GPU is tested separately. The experimental results in the table are the cumulative running time of 1000 iterations of the "add" module. Here are some explanations about the tables. First, "read time" is the running time for reading data from a GPU by the host CPU. Second, "write time" is the running time of writing data from the host CPU to a GPU. Third, "outer time" is the running time of executing the outer region in a GPU. Fourth, "inner time" is the running time of executing the inner region on a GPU. Fifth, "boundary time" is the running time of computing the boundary data in serial on the host CPU. Finally, "wall clock time" represents the running time in practice. There are two wall clock times reported, one associated with the outer time and one

associated with the inner and boundary times. Because kernel functions in different GPUs are expected to execute in parallel, the “wall clock time” of the “outer time” is the running time from the first kernel function that starts to run to the last kernel function that runs, corresponding to the "outer time". The “wall clock time” reported in the rightmost column is the total running time of the inner region computation plus the boundary serial computation.

According to the profiling result in Table 4.2, 4.3, Table 4.4, and Table 4.5, we make the following observations.

- The “read time” and the “write time” are relatively long for the GPUs in the middle (GPU2 when using three GPUs, GPU2 and GPU3 when using four GPUs); that is because the sub-domains in the middle have two more boundary columns that increase the amount of data that needs to be transferred.
- The workload of each GPU is unbalanced. In any system, there are non-observation points on which WDPM does not do any computation. So each sub-domain does not necessarily have the same size. But according to the running time of “outer time”, computations on multiple GPUs can nonetheless be parallelized well.
- The overlapping communication and computation method performs well when working with a large system. In such cases, the running time when using the overlapping technique is significantly shorter than running the GPU computation and CPU computation sequentially. For example, when using two GPUs to work with “patched.asc” (shown in Table 4.3), the boundary data computation (2.09979 s) is almost fully hidden behind the inner region computation on the GPU computation (9.47106 s) because the total wall clock time of the inner and boundary time is 9.48504 s. However, the experiment performed on “basin5.asc” with four GPUs (shown in Table 4.5) shows that the boundary data computation (0.41139 s) takes much longer than the inner GPU computation (0.04385 s), and hence the overlapping technique does not offer any computational advantage.

Table 4.2: The Profiling Result of Using One GPU

	read time (s)	write time (s)	kernel time (s)
system	basin5.asc (471×482)		
GPU 1	0.00014	0.00030	0.18487
system	basin4_5m.asc (2520×1833)		
GPU 1	0.00276	0.00602	4.46162
system	culvert_basin1_5m.asc (3794×3986)		
GPU 1	0.00896	0.01968	29.26780
system	smithcreek_dem1m_sb5.asc (4712×4826)		
GPU 1	0.01347	0.02958	31.90098
system	patched.asc (5877×5519)		
GPU 1	0.01924	0.04291	55.75925

Table 4.3: The Profiling Result of Using Two GPUs

	read time (s)	write time (s)	outer time(s)	inner time (s)	boundary time (s)
system	basin4_5m.asc (2520×1833)				
GPU 1	0.00345	0.00719	1.74791	0.87262	0.38743
GPU 2	0.00343	0.00747	1.20557	0.59462	
wall clock time			1.84967	0.88447	
system	culvert_basin1_5m.asc (3794×3986)				
GPU 1	0.00653	0.01674	8.25735	4.12102	1.89684
GPU 2	0.00617	0.01722	11.11832	5.54391	
wall clock time			11.28418	5.57077	
system	patched.asc (5877×5519)				
GPU 1	0.00832	0.02986	18.90109	9.47106	2.09979
GPU 2	0.00792	0.02986	18.19111	9.09789	
wall clock time			18.99330	9.48504	

Table 4.4: The Profiling Result of Using Three GPUs

	read time (s)	write time (s)	outer time (s)	inner time (s)	boundary time (s)
system	basin4_5m.asc (2520×1833)				
GPU 1	0.00361	0.00599	1.14232	0.57013	0.88357
GPU 2	0.00680	0.10730	1.02638	0.50787	
GPU 3	0.00342	0.00639	0.64678	0.32639	
wall clock time			1.25262	0.97548	
system	culvert_basin1_5m.asc (3794×3986)				
GPU 1	0.00650	0.01335	3.94846	1.97606	3.13170
GPU 2	0.01224	0.02091	8.84756	4.39119	
GPU 3	0.00596	0.01379	6.52940	3.26627	
wall clock time			9.01920	4.43428	
system	patched.asc (5877×5519)				
GPU 1	0.00835	0.02284	11.08195	5.54000	4.24901
GPU 2	0.01646	0.03159	15.72594	7.86060	
GPU 3	0.00793	0.02330	10.10486	5.05297	
wall clock time			15.88408	7.89801	

Table 4.5: The Profiling Result of Using Four GPUs

	read time (s)	write time (s)	outer time (s)	inner time (s)	boundary time (s)
system	basin5.asc (471×482)				
GPU 1	0.00177	0.00227	0.08768	0.04385	0.41139
GPU 2	0.00345	0.00450	0.09065	0.04491	
GPU 3	0.00352	0.00454	0.08777	0.04352	
GPU 4	0.00178	0.00232	0.08013	0.03996	
wall clock time			0.11070	0.57250	
system	basin4_5m.asc (2520×1833)				
GPU 1	0.00352	0.00546	0.64573	0.32040	1.35556
GPU 2	0.00688	0.00943	1.07178	0.52759	
GPU 3	0.00710	0.01017	0.47532	0.23279	
GPU 4	0.00178	0.00585	0.37852	0.18788	
wall clock time			1.25001	1.43666	
system	culvert_basin1_5m.asc (3794×3986)				
GPU 1	0.00620	0.01170	2.18561	1.09400	4.37358
GPU 2	0.01247	0.01849	5.91543	2.94556	
GPU 3	0.01274	0.01930	6.60300	3.28824	
GPU 4	0.00606	0.01211	4.44874	2.20622	
wall clock time			7.83544	4.48975	
system	smithcreek_dem1m_sb5.asc (4712×4826)				
GPU 1	0.00754	0.01531	4.02102	2.00870	4.64050
GPU 2	0.01418	0.02311	8.48651	4.22610	
GPU 3	0.01408	0.02399	6.32994	3.15144	
GPU 4	0.00690	0.01572	2.39133	1.18867	
wall clock time			8.56870	4.74995	
system	patched.asc (5877×5519)				
GPU 1	0.00872	0.01936	7.21244	3.59759	6.08359
GPU 2	0.01642	0.02812	11.53983	5.75371	
GPU 3	0.01642	0.02879	12.23909	6.09307	
GPU 4	0.00794	0.01975	5.82081	2.89596	
wall clock time			12.47135	6.24318	

4.3 Performance Evaluation

4.3.1 Performance Comparison of Previous Study

This section tests the OpenMP CPU parallel code and OpenCL GPU parallel code with the systems in Table 2.1. It is worth mentioning that the serial C code is faster than the serial Fortran code. As a result, the performance improvement of running in parallel with CPU and GPU are compared separately. The CPU is Intel(R) Core(TM) i7-2600 3.40GHz. The running times and the speedup are shown from Table 4.6 to Table 4.9. Running with multiple CPU cores has a decent speedup compared with running in serial. However, limited by the number of cores equipped on the machine, it is hard to achieve a greater speedup. The GPU is an NVIDIA Tesla P100, and the test results are given in Table 4.10. From this table, it can be observed that running on GPU gains significant speedup. Also, the speedup is not ideal when using too many CPU cores because of the large overhead of data synchronization. The previous CPU and GPU parallel methods provide inspiration on how to implement multiple-GPU parallelization and also a reminder of the need for minimizing overhead due to data communication methods.

Table 4.6: OpenMP Performance Test with Basin5.asc

basin5.asc (471×482)							
threads	1	2		4		8	
system	run time(s)	run time(s)	speedup	run time(s)	speedup	run time(s)	speedup
add 5mm	1074.32	632.25	1.70	465.60	2.31	420.53	2.56
subtract 5mm	4.47	2.62	1.71	2.47	1.81	21.12	0.21
drain	39.33	23.70	1.66	19.81	1.99	61.10	0.64

Table 4.7: OpenMP Performance Test with Basin4_5m.asc

basin4_5m.asc (2520×1833)							
threads	1	2		4		8	
system	run time(s)	run time(s)	speedup	run time(s)	speedup	run time(s)	speedup
add 0.5mm	4565.74	2800.6	1.63	1968.75	2.32	1242.05	3.68
subtract 3mm	99.27	52.15	1.90	39.46	2.51	37.89	2.62
drain	121.87	62.80	1.94	46.52	2.62	42.36	2.88

Table 4.8: OpenMP Performance Test with Culvert.asc

culvert.asc (3794×3586)							
threads	1	2		4		8	
system	run time(s)	run time(s)	speedup	run time(s)	speedup	run time(s)	speedup
add 0.5mm	3490.48	1758.71	1.98	1136.90	3.07	808.15	4.32
subtract 5mm	126.82	65.42	1.94	49.91	2.54	46.20	2.75
drain	502.41	286.63	1.75	201.12	2.50	188.35	2.67

Table 4.9: OpenMP Performance Test with Patched.asc

patched.asc (5877×5964)							
threads	1	2		4		8	
system	run time(s)	run time(s)	speedup	run time(s)	speedup	run time(s)	speedup
add 0.5mm	6415.66	3290.08	1.95	1748.14	3.67	1255.51	5.11
subtract 5mm	301.11	153.63	1.96	103.12	2.92	87.53	3.44
drain	1256.78	694.35	1.81	453.71	2.77	402.81	3.12

Table 4.10: Test Result of Running on GPU

	Serial C (s)	GPU (s)	speedup
system	basin5.asc (471×482)		
add 5mm	852.65	6.8	125.39
subtract 5mm	2.74	0.72	3.81
drain	24.75	1.83	13.52
system	basin4_5m.asc (2520×1833)		
add 5mm	3162.61	37.05	85.36
subtract 5mm	17.22	3.45	4.99
drain	619.48	37.05	16.72
system	culvert.asc (3794×3586)		
add 5mm	9068.72	61.65	147.10
subtract 5mm	101.18	8.97	11.28
drain	551.46	28.28	19.50

4.3.2 Performance Comparison of Using Multiple GPUs

This section evaluates the performance of the new implementation when using different numbers of GPUs. The experiment was performed on the Cedar cluster of Compute Canada that is equipped with four NVIDIA Tesla P100 GPUs, so four GPUs at most are used in the experiment. Through the experiment, running with multiple GPUs does not necessarily improve the run time. Specifically, according to Figure 3.10, four-column boundary data matrices are produced when using multiple GPUs to do data synchronization between two adjacent sub-domains, and the boundary data are handled sequentially in the host CPU. In this case, if the input system is small, the boundary part accounts for a large proportion of the overall data, and hence the overhead of data communication becomes relatively large. Specifically, every time when using one more GPU, there is one more four-column boundary data matrix being produced. The performance is quite poor when using four GPUs to run a small system (basin5.asc). However, the purpose of using multiple GPUs is to shorten the running time when solving large problems, and the WDPM itself has been well developed to solve general problems. The rest of this chapter demonstrates the performance improvement of working with large systems.

Chapter 3.2 discusses the domain decomposition method in WDPM. The input system is divided into several slices, and each of them is sent to one GPU. In a real system, there are non-observational areas on which no computations are done. In this case, the computation scales of different sub-domains are different. As a result, the workload of each GPU is unbalanced. To see an ideal acceleration, some artificial systems are produced manually with a random number generator. The sizes of the systems are 1000×1000, 2000×2000, 3000×3000, 4000×4000, 5000×5000, and 6000×6000. The running times of the three modules are recorded for each system, and each task is performed three times to give some idea of the stability of the results. The minimum value (t_{min}) is taken as the final result.

Then, the relative speedup compared with running with one GPU is computed as

$$S = \frac{t_{min}^{(1)}}{t_{min}^{(n)}} \quad (n = 2, 3, 4) \tag{4.1}$$

The complete experimental data can be found in Appendix A and Appendix B, and the performance improvements can be calculated according to Equation 4.1. The test results are shown in Figure 4.6, Figure 4.7, and Figure 4.8.

First, by analyzing the acceleration of the "add" module shown as Figure 4.6, it is observed that the speedup is more significant when the input system is large. Specifically, about three times speedup is achieved when working on a 6000×6000 system with four GPUs.

According to Figure 4.7 and Figure 4.8, the speedup of the "subtract" module and the "drain" module are less significant than the "add" module. This is because the computational amounts of different modules are different. By profiling the serial CPU code working with "basin5.asc", when running the "add" module, the SW function is called 110, 036, 000 times every thousand iterations on average. When running the "subtract"

module, the SW function is called 5, 992, 000 times every thousand iterations on average. And when running the "drain" module, the SW function is called 7, 029, 997 times every thousand iterations on average. So there is a large difference in the calculation scales between in different modules. In WDPM, the SW algorithm only works on the pixels whose water depth is larger than 0. These pixels are called the active points. The "add" module adds a specific depth of water to all the observation points (however, the non-observation points are given a no-data value). So the active points of the "add" module are all the observation points. However, for the "subtract" and the "drain" modules, only a small proportion of the observation points are active points. So working on the "subtract" module and the "drain" module is, to some extent, working with a relatively small system. As discussed previously, the multiple-GPU implementation shows its efficiency when working with a large system because of data synchronization overhead. So that explains why the "add" module has the most significant speedup when using multiple GPUs.

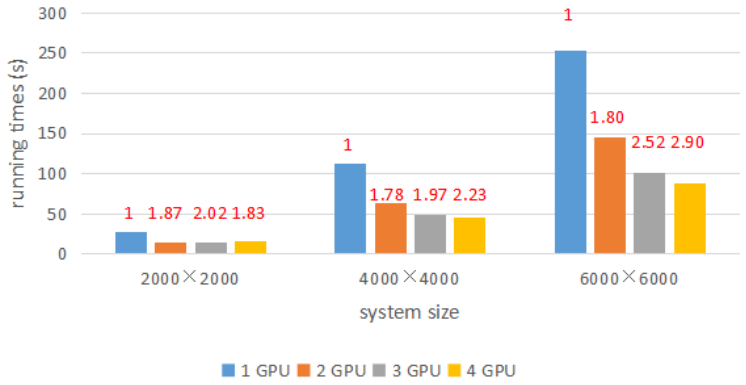

Figure 4.6: The running time and relative speed of the "add" module

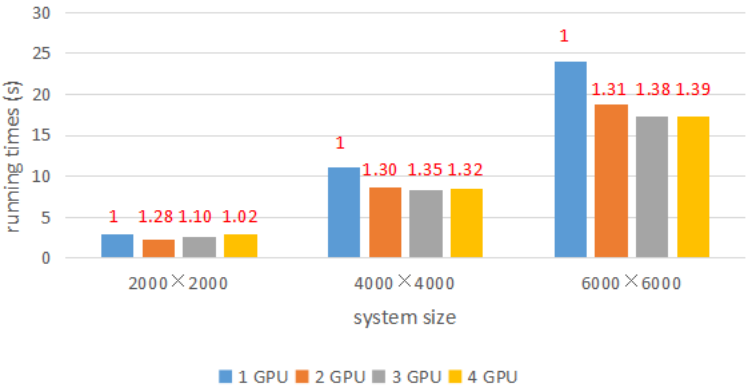

Figure 4.7: The running time and relative speed of the "subtract" module

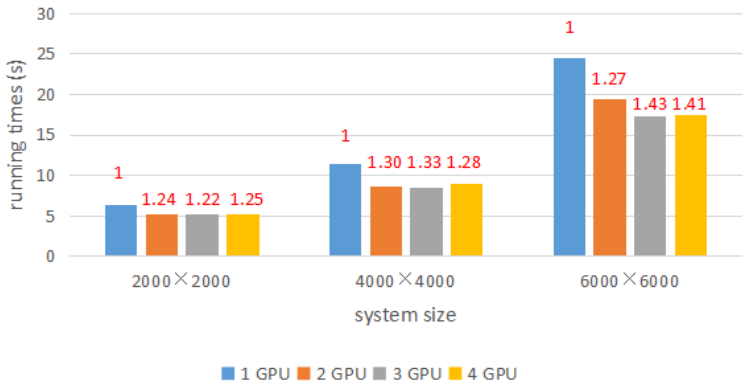

Figure 4.8: The running time and relative speed of the "drain" module

Finally, the experiment is performed on the real systems that listed in Table 2.1. Table 4.11 shows the running time of different systems when using different numbers of GPUs. As reference, the running time when running with Intel(R) Xeon(R) CPU E5-2650 (2.20 GHz) in serial and in parallel CPU (32 threads) is recorded. By comparison, it is clear that running with GPUs makes the simulation much faster. Also, the speedup using different numbers of GPUs compared with using one GPU is shown in Figure 4.9. As mentioned previously, because the proportions of the non-observational area in each sub-domain are different in the real systems, the workloads across the devices are unbalanced. Therefore, the performance improvement is not significant as it is when experimenting with the artificial system. Currently, the largest real system is the "patched.asc" with size 5877×5519. For this experiment, the speedup reaches its best (2.35 times) when using four GPUs.

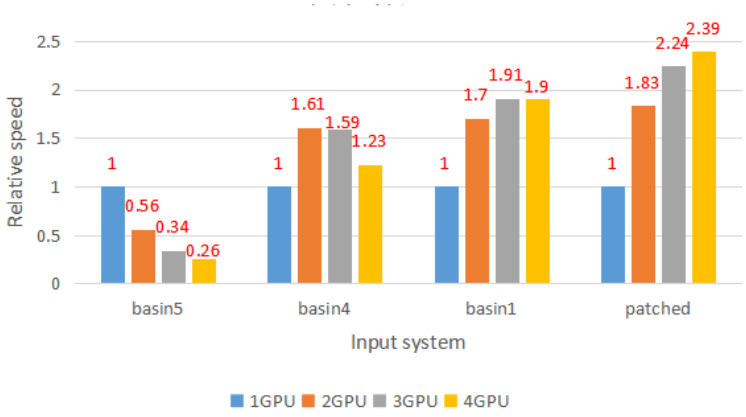

Figure 4.9: The performance comparison of real system experiment

Table 4.11: The Test Results of the Real Systems (the "Add" Module) (Group 1)

		Add				
System		basin5 (+10 mm)	basin4_5m (+5 mm)	culvert_basin1_5m (+3 mm)	smithcreek_dem1m_sb5 (+3 mm)	patched (+3 mm)
Size		471×482	2520×1833	3794×3986	4712×4826	5877×5519
CPUs	1	1132.47	55958.42	360950.93	413045.16	≈ 1115604.75
	32	77.86	2409.92	13758.98	18168.44	44624.19
GPUs	1	8.23	500.70	3203.29	4296.58	11149.74
	2	14.77	311.79	1889.50	2863.56	6079.70
	3	24.04	315.86	1680.17	2320.75	4976.20
	4	32.12	405.81	1683.01	2329.13	4673.24

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The Wetland DEM Ponding Model plays an important role in Canadian prairie hydrology simulation. It models the spatial distribution of runoff water on the Canadian Prairies based on an input DEM system. It has completed many case studies and has been adapted to a few applications (e.g., LIRA). Due to the desire to work with larger systems, it is important to make the water redistribution and routing simulation more efficient to run within a reasonable period of time. For this purpose, an implementation capable of running across multiple GPUs is performed to increase the performance. In particular, this thesis contributes to research in the WDPM in the following aspects.

1. By studying the previous CPU parallel algorithm and the single GPU parallel algorithm, a multiple-GPU parallel algorithm is developed. The correctness of the implementation is verified in Section 4.1 by evaluating the output summaries and the errors of multiple-GPU output maps. Specifically, the experiment produces the error raster system with one GPU output system and multiple-GPU output systems. By simulating with more iterations, we see a significant convergence of the solutions. This result verifies that the multiple-GPU implementation produces the correct result.

2. In the experiment, the event-based technique is used to profile the OpenCL program. As described in Section 4.2.2, although the workload is not the same in each GPU, multiple GPUs can be parallelized well. Furthermore, to fully utilize the computation resource, the new implementation uses overlapping communication and computation techniques to do CPU computation and GPU computation simultaneously. By profiling the simulation when working with different sizes of systems, the overlapping technique is demonstrated to show its efficiency when working with a large system, where the CPU computation can be mostly hidden behind the GPU computation.

3. The running time of working with different numbers of GPUs (up to four) is tested when solving problems of different sizes. Synthetic systems of different sizes are also produced to calculate an ideal performance improvement when working with multiple GPUs. By observation, for small systems, such as "basin5.asc" (471×482), using multiple GPUs actually worsens the performance, which is around 1.59 times slower when using 2 GPUs compared with using 1 GPU. But as the sizes of the problems

increase, using multiple GPUs shows better performance. For the "basin4_5m.asc" (2520×1833), the maximum speedup is about 1.61 when using 2 GPUs. For the "smithcreek_dem1m_s.asc" (4712×4826), the maximum speedup is about 1.92 when using 3 GPUs. For the "patched.asc" (5877×5519), the maximum speedup is 2.35 when using 4 GPUs. Furthermore, a 6000 ×6000 DEM synthetic system is produced to experiment with an ideal (load-balanced) system, and the maximum speedup is about 3.1 when using 4 GPUs. These results verify the significant performance improvement of running the WDPM on multiple GPUs, and they further prove the good scaling of the new implementation.

5.2 Future Work

Here are some possible directions that can be followed to extend the studies in this thesis:

1. Section 2.4.3 discusses the different parallel strategies of GPU and FPGA. Briefly, GPUs are good at space parallelism that uses thousands of threads doing the same computation simultaneously. In OpenCL, this method is implemented with the ND-Range kernel function, whereas the FPGA exploits pipeline parallelism, where different stages of the instructions are applied to different work-items concurrently. Also, FPGA has higher energy efficiency and bandwidth compared with GPU. So FPGA is a promising accelerator for WDPM to achieve better performance. It also helps WDPM adapt to more types of platforms and hardware.

   Currently, an experimental version of WDPM has been developed to run on FPGA with ND-Range kernel function, but it shows performance that is worse than running in serial on a CPU. The SW algorithm decomposition might be necessary in order to implement the water redistribution model in pipeline.

2. The Prairie Region Inundation Mapping (PRIMA) Model aims to provide an accurate and comprehensive storage dynamics simulation and inundation mapping in the prairies. Compared with WDPM, the convergence of PRIMA requires fewer iteration to complete a simulation [34]. A study has verified that PRIMA is more efficient than WDPM when concentrating more water in smaller areas and moving more water downstream to the river (near the outlet) [34]. Originally, the PRIMA can only run on a serial CPU. Currently, PRIMA is investigated to be adapted to GPU parallel computing. With a similar parallel algorithm as the WDPM, PRIMA may significantly speed up when running on a GPU. However, more investigation is needed to analyze the errors compared with running in serial to verify the GPU parallel algorithm produces the correct result for the PRIMA. Also, the GPU parallel implementation only reproduces the water redistribution and routing model in PRIMA. As future study, the water travel time computing function will be added to the parallel program. Furthermore, when the PRIMA is completely transferred to the GPU code, a comparison between the WDPM and the PRIMA in terms of algorithm efficiency, parallel performance, and how well they simulate the actual water distribution can be investigated.

Bibliography

[1] Denis Foley and John Danskin. Ultra-performance Pascal GPU and NVLink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[2] Kevin R Shook, Robert Armstrong, Oluwaseun Sharomi, Ray Spiteri, and John W Pomeroy. The WDPM user's guide, 2014.

[3] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.

[4] Géraud P Krawezik and Gene Poole. Accelerating the ANSYS direct sparse solver with GPUs. In *Symposium on Application Accelerators in High Performance Computing, SAAHPC*, 2009.

[5] Panagiotis D Vouzis and Nikolaos V Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 2011.

[6] Mohamed Hacene, Ani Anciaux-Sedrakian, Xavier Rozanska, Diego Klahr, Thomas Guignon, and Paul Fleurat-Lessard. Accelerating VASP electronic structure calculations using graphic processing units. *Journal of Computational Chemistry*, 33(32):2581–2589, 2012.

[7] John Michalakes and Manish Vachharajani. GPU acceleration of numerical weather prediction. *Parallel Processing Letters*, 18(04):531–548, 2008.

[8] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317, 2011.

[9] Wikipedia contributors. ArcGIS. https://en.wikipedia.org/wiki/ArcGIS, 2021.

[10] University of Toronto. Working with Digital Elevation Models in ArcGIS. https://mdl.library.utoronto.ca/technology/tutorials/working-digital-elevation-models-arcgis.

[11] John W Pomeroy, Kevin R Shook, and Stacey Dumanski. *Improving and testing the prairie hydrological model at Smith Creek Research Basin*. Centre for Hydrology, University of Saskatchewan, 2014.

[12] Robert Armstrong, Cameron Kayter, Kevin R Shook, and Harvey Hill. Using the wetland DEM ponding model as a diagnostic tool for prairie flood hazard assessment. *Putting Prediction in Ungauged Basins into Practice*, pages 255–270, 2013.

[13] Darian Brown. Land & Infrastructure Resiliency Assessment (LIRA).

[14] Jiahu Wang, Li Li, Zhenchun Hao, and Jonathan J Gourley. Stream guiding algorithm for deriving flow direction from DEM and location of main streams. *IAHS-AISH Publication*, 346:198–206, 2011.

[15] Richard Jones. Algorithms for using a DEM for mapping catchment areas of stream sediment samples. *Computers & Geosciences*, 28(9):1051–1060, 2002.

[16] Susan K Jenson and Julia O Domingue. Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogrammetric engineering and remote sensing*, 54(11):1593–1600, 1988.

[17] Jiahu Wang, Li Li, Zhenchun Hao, and Jonathan J Gourley. Stream guiding algorithm for deriving flow direction from DEM and location of main streams. *IAHS-AISH Publication*, 346:198–206, 2011.

[18] Kevin R Shook and John W Pomeroy. Memory effects of depressional storage in Northern Prairie hydrology. *Hydrological Processes*, 25(25):3890–3898, 2011.

[19] Michael Shapiro and Jim Westervelt. R. MAPCALC. In *An algebra for GIS and image processing*. United States Army Construction Engineering Research Laboratory Champaign, 1992.

[20] David A. Patterson and John L. Hennessy. *Computer organization and design: The hardware/software interface*. 2009.

[21] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*, 2010.

[22] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 136–143, 2013.

[23] Jon Peddie. JPR reports global board market in Q1'20. https://www.jonpeddie.com/press-releases/global-add-in-board-market-increased-year-to-year-in-q120-led-by-amd-report/, 2020.

[24] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous computing with openCL: revised openCL 1.2*. Newnes, 2012.

[25] Microsoft. Parallel Programming in Visual C++. https://docs.microsoft.com/en-us/cpp/parallel/amp/cpp-amp-overview?view=msvc-160#introduction, 2021.

[26] Wikipedia contributors. OpenACC. https://en.wikipedia.org/wiki/OpenACC, 2021.

[27] Khronos OpenCL Working Group. *opencl specification version 2.1*. 2018.

[28] Intel and Acceleware. OpenCL on FPGAs for GPU Programmers (parallel programming targeting Altera FPGA). https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-201406-acceleware-opencl-on-fpgas-for-gpu-programmers.pdf, 2014.

[29] Zeke Wang, Bingsheng He, Wei Zhang, and Shunning Jiang. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 114–125, 2016.

[30] Ahmed Sanaullah and Martin C Herbordt. Fpga hpc using opencl: Case study in 3d fft. In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 1–6, 2018.

[31] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, 1992.

[32] Matthew Scarpino. OpenCL in action: how to accelerate graphics and computations. 2011.

[33] Compute Canada. PGPROF. https://docs.computecanada.ca/wiki/PGPROF, 2019.

[34] Mohamed Ahmed, Amin Elshorbagy, and Alain Pietroniro. A Novel Model for Storage Dynamics Simulation and Inundation Mapping in the Prairies. *Environmental Modelling & Software*, 133:104850, 08 2020.

Appendix A

The Experiment Data of Synthetic Systems

Table A.1: The Running Times (s) of the Synthetic System (Group 1)

Module	Add				Subtract				Drain			
GPUs	1	2	3	4	1	2	3	4	1	2	3	4
1000×1000	6.51	4.69	4.65	5.73								
2000×2000	28.22	15.1	13.94	15.45	2.89	2.26	2.64	2.83	4.94	3.54	4.05	4.64
3000×3000	62.85	35.99	27.05	28.25	6.21	4.84	4.83	5.14	6.41	5.15	5.24	5.12
4000×4000	113.01	62.73	48.62	44.68	11.15	8.55	8.25	8.54	11.41	8.7	8.46	8.94
5000×5000	172.39	99.33	69.33	64.56	16.5	12.96	12.25	12.65	16.99	13.08	12.21	12.7
6000×6000	252.62	144.72	100.31	87.13	24.03	18.85	17.28	17.26	24.53	19.38	17.33	17.6

Table A.2: The Running Times (s) of the Synthetic System (Group 2)

Module	Add				Subtract				Drain			
GPUs	1	2	3	4	1	2	3	4	1	2	3	4
1000×1000	6.71	4.19	4.75	5.66								
2000×2000	28.45	15.82	14.34	15.66	3.04	2.45	2.63	2.74	4.97	4.00	5.58	4.35
3000×3000	63.26	34.98	28.01	28.07	6.67	5.17	5.23	5.05	6.43	5.16	5.43	5.19
4000×4000	113.80	63.31	46.97	44.79	11.63	9.06	8.57	8.66	12.02	9.38	8.62	10.04
5000×5000	173.94	96.00	70.55	64.50	17.30	13.88	12.65	12.73	17.04	14.31	12.86	16.22
6000×6000	251.82	141.38	100.44	86.89	24.13	20.00	17.79	17.33	26.36	20.50	18.32	17.45

Table A.3: The Running Times (s) of the Synthetic System (Group 3)

Module	Add				Subtract				Drain			
GPUs	1	2	3	4	1	2	3	4	1	2	3	4
1000×1000	6.62	4.18	4.48	5.72								
2000×2000	28.56	15.42	13.88	15.53	3.17	2.28	2.58	2.62	5.05	4.62	4.29	4.34
3000×3000	63.09	34.65	27.20	28.29	6.39	5.32	4.97	5.01	6.60	6.06	5.09	5.28
4000×4000	113.95	63.16	46.57	45.08	11.67	9.17	8.57	8.54	11.73	10.19	8.68	8.66
5000×5000	173.60	95.75	70.55	66.93	16.87	13.84	12.89	12.42	17.90	13.84	12.91	12.72
6000×6000	253.79	139.97	100.74	87.41	24.26	21.70	18.05	17.18	25.79	20.24	18.14	17.86

Appendix B

The Experiment Data of Real Systems

Table B.1: The Running Times (s) of the Real Systems (Add Module) (Group 1)

		Add				
System		basin5 (+10 mm)	basin4_5m (+5 mm)	culvert_basin1_5m (+3 mm)	smithcreek_dem1m_sb5 (+3 mm)	patched_YQ (+3 mm)
Size		471×482	2520×1833	3794×3986	4712×4826	5877×5519
GPUs	1	8.72	516.8	3239.15	4307.72	11189.83
	2	14.26	310.33	1888.08	2860.79	6049.24
	3	23.75	314.12	1664.7	2303.22	4938.27
	4	32.68	401.51	1691.02	2324.29	4585.54

Table B.2: The Running Times (s) of the Real Systems (Subtract Module) (Group 1)

		Subtract				
System		basin5 (–10 mm)	basin4_5m (–5 mm)	culvert_basin1_5m (–3 mm)	smithcreek_dem1m_sb5 (–3 mm)	patched_YQ (–3 mm)
Size		471×482	2520×1833	3794×3986	4712×4826	5877×5519
GPUs	1	0.89	40.56	71.66	1324.44	28.01
	2	1.07	36.22	58.06	1173.04	24.58
	3	1.36	44.06	64.91	1288.16	24.55
	4	2.26	54.80	74.82	1477.24	26.05

Table B.3: The Running Times (s) of the Real Systems (Drain Module) (Group 1)

		Drain				
System		basin5	basin4_5m	culvert_basin1_5m	smithcreek_dem1m_sb5	patched_YQ
Size		471×482	2520×1833	3794×3986	4712×4826	5877×5519
GPUs	1	2.69	8.44	14.65	1788.13	29.83
	2	3.32	6.80	12.24	1446.01	28.67
	3	4.26	8.08	14.53	1672.78	26.66
	4	6.01	9.25	15.69	1768.10	26.67

Table B.4: The Running Times (s) of the Real Systems (Add Module) (Group 2)

		Add				
System		basin5 (+10 mm)	basin4_5m (+5 mm)	culvert_basin1_5m (+3 mm)	smithcreek_dem1m_sb5 (+3 mm)	patched_YQ (+3 mm)
Size		471×482	2520×1833	3794×3986	4712×4826	5877×5519
GPUs	1	8.04	492.32	3184.36	4292.94	11129.12
	2	15.18	313.92	1886.87	2864.99	6100.16
	3	24.34	320.49	1694.97	2329.64	5009.25
	4	32.03	408.29	1675.91	2329.33	4735.54

Table B.5: The Running Times (s) of the Real Systems (Subtract Module) (Group 2)

		Subtract				
System		basin5 (–10 mm)	basin4_5m (–10 mm)	culvert_basin1_5m (–10 mm)	smithcreek_dem1m_sb5 (–10 mm)	patched_YQ (–10 mm)
Size		471×482	2520×1833	3794×3986	4712×4826	5877×5519
GPUs	1	0.84	39.68	71.43	5336.18	27.37
	2	0.88	37.56	64.37	4585.33	26.45
	3	1.46	45.03	70.44	5301.78	26.53
	4	2.01	53.73	76.68	5643.63	26.43

Table B.6: The Running Times (s) of the Real Systems (Drain Module) (Group 2)

		Drain				
System		basin5	basin4_5m	culvert_basin1_5m	smithcreek_dem1m_sb5	patched_YQ
Size		471×482	2520×1833	3794×3986	4712×4826	5877×5519
GPUs	1	2.68	8.49	14.86	6701.48	29.44
	2	2.87	7.23	13.15	5521.70	28.23
	3	4.79	8.06	13.51	6756.42	27.49
	4	5.96	9.10	13.55	6485.01	27.17

Table B.7: The Running Times (s) of the Real Systems (Add Module) (Group 3)

		Add				
System		basin5 (+10 mm)	basin4_5m (+5 mm)	culvert_basin1_5m (+3 mm)	smithcreek_dem1m_sb5 (+3 mm)	patched_YQ (+3 mm)
Size		471×482	2520×1833	3794×3986	4712×4826	5877×5519
GPUs	1	7.92	492.99	3186.35	4289.07	11130.21
	2	14.86	311.12	1893.54	2864.91	6089.70
	3	24.04	312.97	1680.85	2329.38	4981.07
	4	31.66	407.64	1682.11	2333.76	4698.65

Table B.8: The Running Times (s) of the Real Systems (Subtract Module) (Group 3)

		Subtract				
System		basin5 (–10 mm)	basin4_5m (–10 mm)	culvert_basin1_5m (–10 mm)	smithcreek_dem1m_sb5 (–10 mm)	patched_YQ (–10 mm)
Size		471×482	2520×1833	3794×3986	4712×4826	5877×5519
GPUs	1	1.06	40.48	71.03	5345.59	27.17
	2	1.13	37.20	63.72	4548.95	26.18
	3	1.34	45.08	71.42	5199.47	26.08
	4	2.22	53.81	81.25	6523.38	27.50

Table B.9: The Running Times (s) of the Real Systems (Drain Module) (Group 3)

		Drain				
System		basin5	basin4_5m	culvert_basin1_5m	smithcreek_dem1m_sb5	patched_YQ
Size		471×482	2520×1833	3794×3986	4712×4826	5877×5519
GPUs	1	3.15	7.99	14.69	6702.39	40.60
	2	6.16	6.98	12.66	5460.61	27.89
	3	9.08	9.24	13.54	6501.6	27.65
	4	6.52	9.26	13.95	6523.38	27.50