GPU Accelerated Procedural Terrain Generation

A thesis presented in partial fulfilment of the
requirements for the degree

Master of Science

in

Computer Science

at Massey University, Albany
New Zealand

Richard Changwoo Kim

2021

Abstract

Virtual terrain is often used as the large scale background of computer graphics scenes. While virtual terrain is essential for representing landscapes, manual reproduction of such large-scale objects from scratch is time-consuming and costly for human artists. Many algorithmic generation methods have been proposed as an alternative solution to manual reproduction. However, those methods are still limited when needing them to be employed in a wide range of applications. Alternatively, simulation of the stream power equation can effectively model landscape evolution at large temporal and spatial scales by simulating the land-forming process. This equation was successfully employed by a previous study in terrain generation. However, the unoptimised pipeline implementation of the method suffers from long computation time on the increased simulation size. Graphics processing units (GPUs) provide significantly higher computational throughput for massively parallel problems over conventional multi-core CPUs. The previous study proposed a general parallel algorithm to compute the simulation pipeline, but is design for any multi-core hardware and does not fully utilise the computing power of GPUs. This study seeks to develop an optimised pipeline of the original stream power equation method for GPUs. Results showed that the new parallel GPU algorithm consistently had higher performance (about 300% for GTX 780 and 900% for RTX 2070 Super) recent octa-core CPU (Intel i7 9700k 4.9 Ghz). It also consistently showed a 300% improvement in performance over the previous parallel algorithm on GPUs. The new algorithm significantly outperformed the fastest parallel algorithm available, while still being able to produce the same terrain result as the original stream power equation method. This advancement in computational performance allows the algorithm method to generate precise geological details of terrain while providing reasonable computation time for the method to be employed in a broader range of applications.

Acknowledgement

First of all I would like to thank my supervisor Daniel Playne for his wisdom, endless patience, and encouragement throughout this work. In particular, I would also like to thank my family Marleen Baling, little Juno, my parent Jintae and Hyungok, and my brother Hyunwoo for their continual love and support. Also thank my colleague Dara Quach for his so many valuable advice. Lastly, I wish to acknowledge Massey University for their financial support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer graphics is the creation, storage and manipulation of images and models using computers. There are many applications of computer graphics, and one of the commonly known is to provide experiences of virtual worlds. Many of these virtual worlds can present a realistic world of imagination that cannot be experienced in reality. This attraction has created an increasing demand from movie audiences and video game users for high degrees of realism and for development of artificially crafted objects within the virtual scenes. Computer graphics application has also been employed for manufacturing industries where virtually-rendered images of the products are economical and flexible for presenting products. Construction industries have also applied computer graphics to virtually render their expected construction outcome, such as previewing a building and for pre-inspection of the design and structure. As a result, the increased expectation for computer graphics applications has been demanding the continuous and rapid evolution of electronic devices.

Virtual terrain is essential background for representing landscapes in many computer graphics applications such as movies, video games, architectural design, and simulations. However, virtual terrain is usually a complex large-sized object, and for humans to craft it manually from scratch is a costly approach because human labour is an expensive resource. An alternative method is using computers to generate terrain automatically. Such methods are called procedural content generation in computer graphics, which algorithmically creates digital content with limited or indirect input. Procedural terrain generation methods have been an active area of research in computer graphics for almost four decades. Many proposed methods are typically based on the fractal, example data, and physical simulation. Different procedural terrain methods have their advantages and disadvantages.

Additionally, the purpose of the application will affect the goals for these methods. Most applications want to have reasonable control over the method to create results that they desire

rather than an unexpected random result. However, lack of control is not a concern for some applications. For example, several video games attempt to provide a totally unexpected random experience to players. Computation time should also satisfy the purpose of the application. There is often a trade-off between realism and computation time, where the demand for more realistic details usually means more computation. For applications such as movies and simulations, realism is the most important factor, and how long it takes to compute is often less of a concern. However, there are applications where realistic quality and computation time are both important. Video games require interactions with users, but the user also wants the virtual world to have the highest quality detail for a better experience at the same time. It may be one reason simulation-based terrain generation methods are not very popular for an interactive application like video games that intend to create content on the fly because simulations are usually expensive to compute even though it can add realism to the result.

In order to achieve better realism for applications that require both reasonable computation time and quality detail, parallel computing may offer as a possible solution. Parallel computing exploits multiple processing cores to execute the computation simultaneously to solve a computational problem. A decomposition of the computation usually has to be manually written into a parallel algorithm by the programmer. There are a variety of computing resources available for parallel computing such as a multi-core CPU, network of distributed computers, digital signal processor, field-programmable gate array, and graphics processing unit. Different resources are optimised for different purposes and choosing the most effective combination for the application is crucial.

Graphics processing units (GPUs) are designed for accelerating computer graphics rendering pipeline. In order to process millions of pixels for a scene showing on the screen in real-time, GPUs have thousands of small processing cores. GPUs can also be effective for computing a general problem that requires a large amount of computation if the problem can be decomposed into small subproblems. These are often called massively parallel problems, and GPUs can provide significantly higher computational throughput in such cases as compared to multi-core CPUs. Parallel computing exploiting GPUs may be one solution to generate terrain with high-quality detail while also satisfying the requirement for reasonable computation time.

Research Goal

This study seeks to develop a method that generates large scale virtual terrain by simulating the land forming process to produce realistic geological detail. This study simulates the stream power equation, which was successfully employed by the method proposed in [1]. The previous study

showed that the simulation of the equation can effectively generate large-scale features of terrain such as valleys, mountains, and river networks. However, their implementation of the simulation pipeline is unoptimised, and increasing the simulation size would result in a long computation time.

This study objective was also to improve the computation time of the new method while keeping the quality of terrain to widen the application range that the method can be employed. There has a study that proposed a parallel algorithm for the simulation [2]. The parallel algorithm proposed in [2] was a general algorithm that targeted any multi-core hardware such as multi-core CPUs, and GPUs. However, the parallel algorithm specifically optimised to utilise GPUs' computing power has not yet been proposed. Therefore, this study seeks to develop an optimised pipeline of the original stream power equation method for GPUs.

This thesis is arranged as follows. Chapter 2 briefly introduces the procedural terrain generation and the method proposed in [1]. Chapter 3 presents and discusses the stream power equation and its simulation in detail. Chapter 4 introduces GPU computing, which helps to explain the parallel implementation of the simulation pipeline. Chapter 5 presents and discusses different parallel algorithms and their implementations to compute the simulation pipeline. Chapter 6 presents and discusses the result shown in this research, including computation time and terrain generation result. Finally, Chapter 7 presents an overview of the study and discussion, along with some conclusions and future work.

Chapter 2

Procedural Terrain Generation

Virtual world in computer graphics scenes consists of a variety of digital contents such as vegetations, terrain, vehicles, buildings, living objects, and texture. For the application such as video games, the contents also include levels, maps, rules, stories, items, quests, music, and sound effects. While the creation of such contents is typically the responsibility of the human artist, humans are slow and expensive. Moreover, those contents can be spatially demanding for limited storage size. These limitations drove people to think of the automatic creation of digital contents through computer algorithms. In computer graphics, the algorithmic generation technique of digital contents is called procedural content generation.

Procedural content generation (PCG) refers to the algorithmic generation method of digital contents with limited and or indirect human input. PCG algorithms are typically based on repeating patterns found in the object, exploiting existing dataset of the object, or physical simulations of the forming process of the object. PCG methods were first introduced to overcome the storage limitation in video games by generating contents on the fly instead of storing them in the storage. Manual crafting of the variety of contents from scratch can be time-consuming and challenging for the human artist, which also can be improved with the assistance from algorithms. However, the effectiveness of PCG methods is limited to the contents that the algorithm can define its creation pattern such as maps and levels in video games, or forming process such as terrain, and plants. PCG is not a preferable solution for contents that require unique and artistic design.

Virtual terrain serves as an essential background in many computer graphics applications such as video games, films, architectural design, and simulations. Complex landform features of terrain are a result of interactions between various natural phenomena, and it is challenging for the human to reproduce such features while keeping geological correctness. It is because while our visual system is highly effective at identifying important features of natural terrain, our interpretation of such features does not focus on their forming process. In order to overcome difficulties in virtual

terrain crafting, procedural terrain generation has been an active area of research in computer graphics for decades.

2.1 Procedural Content Generation

Computer graphics applications consist of a variety of digital contents such as texture, sound effect, 3D objects. Procedural content generation (PCG) is an algorithmic creation method of digital contents with the minimal, or indirect input from the user. PCG was first employed in video games to overcome the storage size limitation in the early days. Instead of including large amounts of contents data in the packages, exploiting PCG creates the contents on the fly as the user proceeds in the game world.

In addition to the efficient use of storage, there are also other advantages of PCG. The algorithmic generation is an automatic creation of contents with minimal input required from the user, which can enhance the productivity and creativity of the human artist. Randomness can also be introduced to the procedure using pseudorandom number generators. Random content generation enables game developers to provide less predictable gameplay, which helps with creative game design as a result.

PCG has been receiving increasing attention in many computer graphics application in large industries. A classic example is the dungeon generation in the early 80s game Rogue (1980). The random map generation in Civilisation has been allowing a unique gameplay experience. Diablo (1996) employs PCG all over the game for its contents, such as maps, type, items, and monsters. Borderland, the famous shooting game uses PCG to generate weapons in the game (2009). Minecraft is a good example of extensive use of PCG techniques to generate the whole world and its contents (2009). Although PCG methods were first developed for video games and have been mostly employed for video games, they are also extensively used in other industries such as films, and design. Speed Tree, the interactive data visualisation tool that generates vegetation is employed in many 3D modelling for games, films, architectural design (2003). MASSIVE (Multiple Agent Simulation System in Virtual Environment) is a famous example of PCG used in many famous films. It was first developed for Lord of the Ring, and its key feature is the ability to generate thousands of agents that all act as individuals.

Due to the increasing attention, PCG has been an active area of research across the academia and industry. Research goal in PCG vary depending on the application, but there are some common desirable properties. Firstly, the PCG method must be able to generate believable content reliably. It is not desirable for the contents consumer to identify that the algorithm generated the content. The failure of the content creation due to the unreliability of the algorithm cannot be forgiven

especially for real-time applications such as video games. Secondly, the contents creator usually need the ability to specify some aspects of the content to be generated. PCG methods that are difficult to control are only useful for the limited scope of applications. Control methods are desired to be highly-influencing to the resulting content and not to be too complicated. Lastly, the computation time of the algorithm should satisfy the requirement of the application. Requirements for time vary widely depending on the application, from real-time to days. The desirable properties of a solution are different for each application. There is usually some degree of tradeoffs between the properties involved.

2.2 Procedrual Terrain Generation

Terrain serves as an essential background in many computer graphics applications in large industries such as video games, films, architectural design, and simulations. Features of terrain such as eroded parallel valleys, dendritic mountain ranges, and river networks are a result of complex interactions between a number of natural phenomena such as temperature changes, rain, wind, lightning, tectonic uplift, and plant ecosystem. The human visual system is highly effective at identifying important features of natural terrain and so can quickly recognise errors and inconsistencies in virtual landscapes. However, our interpretation of such complex features does not focus on their forming process, and therefore, it is challenging and time-consuming for a human creator to reproduce realistic-looking landscapes from scratch. In order to overcome the challenge of the terrain crafting, developing methods of procedural terrain generation has been an active area of research in computer graphics.

Procedural terrain generation has been studied for nearly four decades, and there are three common approaches in the fields; fractal and noise, example-base, physical simulation of natural phenomena. Fractal-based methods were inspired by the fact that similar patterns repeat at different scales in terrain features. Using existing terrain dataset to generate new ones is also a popular approach. These methods can exploit real-terrain dataset to reproduce realistic virtual terrain. Methods that generate terrain data by simulating natural phenomena have been received much attention since there is a large body of previous work on modelling of landscapes evolution. Simulation of natural phenomena can reproduce geologically more accurate terrain features at small to large scale. The goal of research in procedural terrain generation is typically to achieve better realism, controllability, and reasonable computation time for its application. Each of these approaches has advantages and disadvantages and which to be used is depending on the application.

Fractal-based methods are built on the observation that similar patterns repeat at different scales in terrain feature. Fractional Brownian motion (fBm), the continuous-time Gaussian process

is a basis of methods [3]. Two commonly known approaches are a random midpoint displacement algorithm [4, 5, 6, 7] and a noise-based method [8]. Both methods are efficient to compute, especially the independent evaluation of the noise function of noise-based methods enable efficient parallel implementations. However, the result of fBm is difficult to control through input parameters which makes fractal-based terrain generation methods suffer from lack of control [9]. Moreover, the self-similar pattern modelled by fractal terrain is only observed in relatively new not eroded terrain, and such terrain does not include geological structures formed by erosion such as networks of parallel valleys [4].

Example-based methods exploit the existing terrain model to generate new ones. The examplar can be real terrain dataset or even generated terrain by other terrain models. A typical approach in example-based methods is based on texture synthesis [10, 11, 12, 13]. This approach takes a user painted input and derives the corresponding terrain model by combining synthesis extracted from the dataset. Another approach is terrain altering [14, 15]. It is a technique that generates user-desired terrain by adjusting provided exemplars. Example-based methods can achieve high-quality realism, control and interactive performance. The difficulty is that the quality of output is heavily dependent on the input data. They also have a difficulty to provide realism at a large spatial scale because the nature of the methods does not consider the geological land forming process.

Erosion is the most influencing geomorphological agent that forms natural terrain and most of the simulations for terrain modelling are based on erosion. Erosion is the transportation action of surface processes that remove material from a location then move to another location. Exploiting erosion simulations have been studied for many years in computer graphics to generate terrain that is geologically more accurate. Erosion processes include a number of natural phenomena such as temperature changes, rain, wind, lightning, tectonic uplift, and plant ecosystem. One that has been received the most attention in the field of procedural terrain generation is a hydraulic erosion. A simple hydraulic and thermal erosion model was coupled with fractal noise function to implement a warping function that can break the unrealistic regularity of the fractal terrain [3]. More detailed hydraulic erosion simulation model later proposed by adopting physical fluid simulation [16, 17, 18, 19].

While methods that are based on hydraulic erosion simulations can effectively generate geologically correct landform features such as complex river network, valleys, and dendritic ridges, they suffer from expensive computational cost and lack of control. Fluid simulation is a well-known computationally expensive problem and also the spatial and temporal scale of many hydraulic erosion simulations is small, which makes it limit the scale of terrain due to the providing tractable computation time. Erosion processes can only be controlled through initial parameters of the simulation, but it is difficult to predict the result from the initial condition before the simulation

completes, which means that if the result is not satisfactory at the end of the simulation, the user has to re-run the simulation with different initial parameters until the simulation generates the desired terrain.

Background landscapes in many computer graphics applications are usually required to be able to represent a large spatial scale. Although hydraulic erosion simulations add high-quality realistic details to terrain features, many of simulation algorithms are difficult to control and limited in scales. The hydraulic erosion simulation method proposed in [1] resolved these limitations by exploiting the simulation of large spatial and temporal scales erosion model, the stream power erosion model [20]. The stream power erosion model efficiently captures the action of hydraulic erosion at large spatial and temporal scales by including fluvial erosion. Fluvial erosion is the process of detachment of materials of the river bed and the sediment and its transportation downhill by running water. At such large scales, terrain evolution cannot be considered without also taking into account the increase of elevation of earth surface caused by collisions between tectonic plates, which is called in term tectonic uplift. A variety of geological processes are involved in the formation of terrain at large scale, but the most significant of which is the interaction between tectonic uplift and fluvial erosion and such interaction is modelled by the stream power equation.

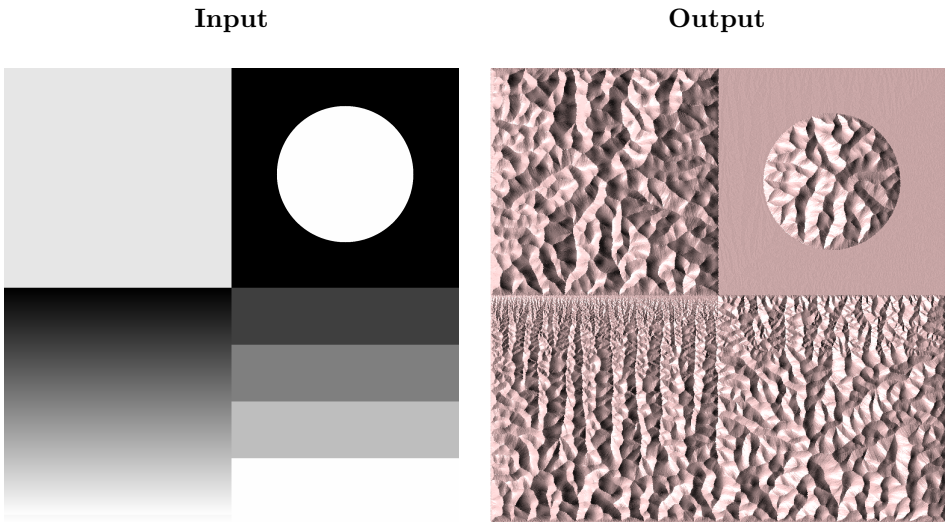
Exploiting the simulation of the stream power erosion model can efficiently generate geologically accurate terrain features at large spatial scales. Moreover, the method in [1] proposes a highly-influencing and straightforward control method. Instead of controlling the hydraulic erosion process, the method in [1] controls the result by feeding a simple user-painted grayscale image that defines tectonic uplift rate over the domain. Figure 6.1 shows the influence of the user-painted input to the output. The simulation of the stream power erosion model is the basis of this study to generate terrain of large spatial scale, which is discussed in detail in the following chapter.

2.3 Summary

Virtual terrain serves as an essential background in many computer graphics applications. This chapter briefly introduced procedural generation methods of the virtual terrain. This study aims to develop the method that can generate geologically more accurate virtual terrain of large spatial scales.

There are three typical approaches in the field of procedural terrain generation; fractal-based, example-based, and erosion-based. Fractal-based and example-based methods are efficient to compute. However, they cannot reliably generate geologically correct terrain features at large spatial scale such as river networks, and valleys because the land forming physical processes are not taken into account. Hydraulic erosion-based generation methods can generate geologically more accurate

Figure 2.1: An example to show the influence of a simple user-painted input to the output. A user-painted greyscale image defines uplift rates over the domain where higher intensity represents the higher uplift rate.



terrain by simulating physical land forming process. However, they are difficult to be employed in many real-world problems due to the long computation time.

The Stream power erosion model is an extension of hydraulic erosion model, which models the interaction between tectonic uplift and fluvial erosion. The method introduced in [1] successfully employed this model to effectively generate virtual terrain of a large spatial scale within a reasonable computation time. The steam power erosion model is the basis of this study, and more details are discussed in the following chapter.

Chapter 3

Stream Power Erosion Model

Chapter 2 described various procedural generation methods of terrain and discussed the effectiveness of the stream power erosion model for mountainous terrain generation. Hydraulic erosion models can reproduce terrain of realistic detail. However, it is difficult to control the erosion process for desired output, and those models are computationally demanding to be used at large spatial and temporal scale. The stream power erosion model were first exploited to generate terrain in the method suggested in [1]. They extended the way hydraulic erosion is modelled in terrain generation methods to capture such erosion interacting with tectonic plates at large spatial and temporal scales, the action of water forming streams that carve the terrain while it vertically grows. The basis of the stream power erosion model is a partial differential equation named the stream power equation, which models key geological processes of mountainous terrain formation. Computational simulation of the stream power equation is exploited to generate terrain in this study.

This chapter introduces the stream power erosion model in detail. The stream power equation and its geological background are introduced in Section 3.1. How the domain of landscapes can be approximated for the simulation is introduced in Section 3.2. Finally, the computational simulation of the equation and the limitation of the previous work is introduced in Section 3.3.5.

3.1 Stream Power Equation

Large scale landscapes such as mountainous terrain contain complex landform features including valleys, ridges, and river networks. A variety of geological processes are involved in the formation of such landscapes, the most significant of which is the interaction between tectonic uplift and fluvial erosion. Tectonic uplift is increase of elevation of earth surface caused by collisions between tectonic plates. Fluvial erosion is the process of detachment of materials of the river bed and the sediment and its transportation downhill by running water. The stream power equation is

a hyperbolic partial differential equation that is used to model the interaction between tectonic uplift and fluvial erosion [20].

The stream power equation models the rate of change in surface elevation over the domain of landscapes, which is controlled by the balance between the tectonic uplift rate and fluvial erosion rate. The equation divides the domain into smaller regions, and the tectonic uplift rate and fluvial erosion rate are computed for each of these local regions. The equation can be written as the following:

$$\frac{dh(p)}{dt} = U(p) - E(p) \tag{3.1}$$

Where p denotes the 2D point of each local region, $h(p)$ denotes the surface elevation at point p, $U(p)$ denotes a function that evaluates the given uplift rate caused by tectonic plates at point p, $E(p)$ denotes a function for the fluvial erosion rate at point p. In the stream power equation, the change in elevation $\frac{dh(p)}{dt}$ at point p is a difference between $U(p)$ and $E(p)$.

The function $E(p)$ for a fluvial erosion rate at position p can be written as the following:

$$E(p) = -kA(p)^m S(p)^n \tag{3.2}$$

Where A(p) denotes an approximation of total water flowing into point p, S(p) denotes a gradient of a discharging path of running water at point p, which is used to approximate the total discharge rate of water. k, m, and n are constant positive values. The constant k is an erosion factor that depends on geological factors such as lithology, channel width, and channel hydrology. The constants m and n are scaling factors that depend on bedrock strength, climate, and the topology of stream networks. While the appropriate values for k, m, and n are still up for debate, observations on depth and channel of streams indicate that ratio of m/n is constrained to be around 0.5 [20]. Values $m = 0.5$ and $n = 1$ are used in most geomorphology studies.

3.2 Representation of the Domain of Landscapes

Computational simulation of the stream power equation is used to generate terrain in this study. In order to simulate the stream power equation, continuous fields must be discretised. The terrain can be represented by a geometric mesh connecting a discrete set of points sampled over the terrain domain. A sample point represents topographic information such as the position and elevation of a local region in the domain. Meshes for the simulation can be stored in a graph data structure. Each node $N_i \in$ graph G is each sample point over the terrain domain. Each edge $E_i \in G$ represents a connection between sample points, which is used to represent a potential path of running water

in the domain; the path of running water is called as a term stream-flow in the stream power erosion model. Geometric information stored in G can be used to render virtual terrain in 3D space directly without any post-processing.

There are many types of meshes available for the simulation. A regular grid shape could be used, as described in [21]. However, while it provides simplicity in implementation due to its regular structure, it can introduce artefacts since water does not always flow in directions aligned to such structure. Therefore, in order to remove such artefacts, the structure of the mesh must be able to represent stream-flows in a variety of directions. A random mesh generation method suggested in [1] is a preferable solution for this reason. The method consists of two stages, random point generation over the domain, and construction of the set of connections between generated points using Delaunay triangulation.

A method called Poisson disc sampling [22] is used to generate a set of random points. It is a blue noise sampling method based on nonuniform Poisson distribution where all samples are at least distance r apart from each other. Sampling is indirectly controlled using a density parameter, r that defines the minimum distance between samples but not the number of samples directly. This method can generate a set of points that is tightly-packed and in a naturally distributed pattern.

The set of points are then connected by computing a Delaunay triangulation of the points. In computational geometry, for a given set of discrete points P in a plane, a Delaunay triangulation is a triangulation $DT(P)$ such that no point in P is inside the circumcircle of any triangle in $DT(P)$. The association of Poisson disc sampling and Delaunay triangulation produce a quality triangle mesh where most of the edges and angles are quasi-uniformly distributed. This randomly generated mesh can represent stream-flows more randomly distributed angles, which enables stream-flows to be aligned to the pattern of more like natural streams, as shown in Figure 3.1.

There is another useful property of the Delaunay triangulation that can be exploited in the stream power erosion model. A set of Delaunay triangles $DT(P)$ for a given set of points P is a dual graph of a Voronoi diagram $VD(P)$ for P. A Voronoi diagram is a partition of a plane into regions that correspond to each point of a given set of points P. Each of these regions consists of every point in the Euclidean plane whose distance to its corresponding point in P is closer than any other points in P. These regions are called a Voronoi cell. For a given set of points P, a Voronoi Diagram $VD(P)$ can be found by connecting the circumcentres of neighbouring triangles in $DT(P)$. The approximation of the surface area a_i of each local region represented by each node $N_i \in G$ serves as an important property to evaluate the amount of rainfall received. Each Voronoi cell v_i of $VD(P)$ that surrounds the node $N_i \in G$ can provide a good approximation for the local surface area value of a_i by computing the surface area of v_i.

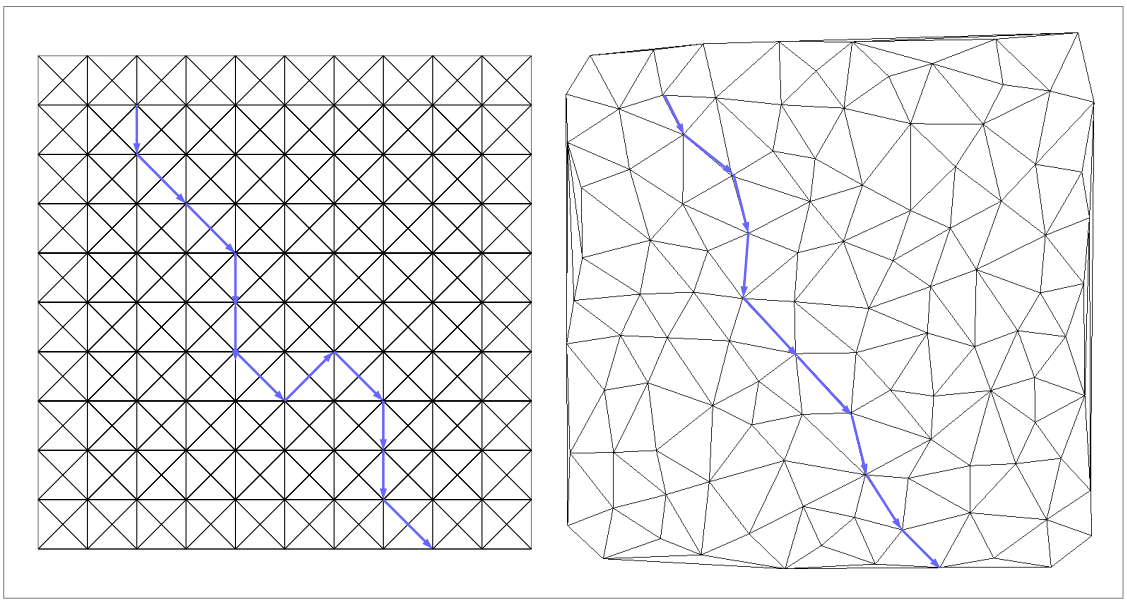


Figure 3.1: A visual example of directions of running water on different types of meshes.
A regular grid (left), Random mesh generated by the method in [1] (right).

3.3 Computational Simulation

In order to simulate the stream power equation, the equation must be computed for discrete timestep for each node $N_i \in G$ until the simulation converges. Each iteration of the simulation requires the following steps:

1. Compute stream connectivity that is represented by a set of trees T covering the graph G.
2. Compute the change in elevation caused by tectonic uplift for each node $N_i \in G$.
3. Compute the change in elevation caused by fluvial erosion for each node $N_i \in G$.
4. Compute the change in elevation caused by thermal erosion for each node $N_i \in G$ for correction of extreme peaks.

Convergence occurs when the change in elevation is below a given threshold. It is difficult to predict the number of iterations required until convergence. The implicit solver suggested in [21] enables solving the equation with large-timestep for the fast convergence, but as the size of the simulation grows, the number of iterations required to converge also increases.

3.3.1 Stream-flow Computation

In the stream power erosion model, a stream-flow is a term for directions of running water between neighbouring regions. Stream-flows are computed for each node $N_i \in G$, and it is to construct a set of directed trees T that covers G. This set of directed trees T is used to represent streams

in the model. The topology of T must be updated every iteration according to the topographic gradient of G that approximates total water accumulation and discharge at each local region.

To define a set of directed trees T, we employ the same single flow direction model used in [1]. Nodes that are connected by the edge in G are a neighbour region of each other. The model assumes that a local stream flows in a single direction into a neighbouring region with the steepest descent slope. This neighbour is called the receiver node denoted by N_{R_i} for each node $N_i \in G$. Based on this model, we define T as follows. N_{R_i} for each node $N_i in G$ can be computed by finding the neighbour that has a lower elevation than N_i and the steepest gradient. For N_{R_i}, N_i is called a donor node $D_{N_{R_i}}$ of N_{R_i}. If a node does not have any neighbour with a lower elevation, water does not flow out of its region. For such cases, the node serves as a root of each tree in T. If the simulation initially starts from a flat surface with no elevation, fluvial erosion process does not occur during the first iteration. This is because all nodes serve as a root of N trees, where N is the total number of nodes in G.

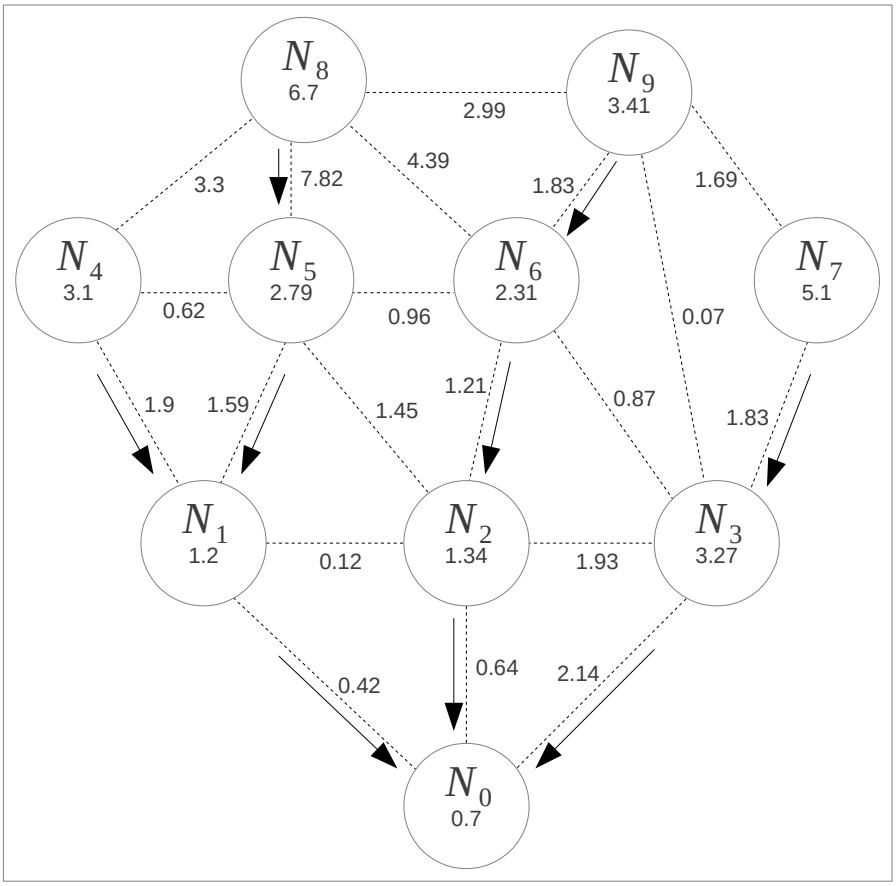


Figure 3.2: An example streasm-flow graph consisting of 10 nodes. Each node N_i and its elevation are represented by a circle. Dashed lines represent an edge of the global graph, and a number next to the edge denotes a slope between nodes. Arrow lines represent a directed edge of the tree.

Figure 3.2 is a visual example of a small system covered by one tree. As shown in the figure, the receive N_{R_i} of each node N_i is the neighbour with lower elevation and the steepest slope. In

Figure 3.2, N_0 serves as a root of the tree because its elevation is lower than all of its neighbours.

3.3.2 Uplift Computation

The uplift rate in the stream power erosion model approximates increase of elevation due to the movement of tectonic plates. The surface uplift $u(p)$ in the equation is a function that defines the uplift rate at point p. The uplift over a timestep Δt for each node $N_i \in G$ can be defined using the following equation:

$$h_i(t + \Delta t) - h_i = u_i \Delta t \tag{3.3}$$

Where h_i denotes the elevation of N_i, u_i is a constant that defines the uplift rate at N_i, Δt denotes the length of timestep. The equation can be used to compute a new elevation $h_i(t+\Delta t)$ of N_i over the timestep caused by tectonic uplift:

$$h_i(t + \Delta t) = h_i + u_i \Delta t \tag{3.4}$$

Boundary Condition

In the stream power erosion model, nodes that are on the boundary of the geometric graph G serves as river mouths such as the region at the sea level, and outflows of the terrain domain. Several boundary conditions are suggested for fluvial erosion simulation such as fixed height boundary condition, reflecting boundary condition, and periodic boundary condition [21]. This study employs the fixed height boundary condition used in the method in [1], which fixed the height of nodes on the boundary to constant. The height for boundary nodes can be fixed to constant by setting their uplift rate to 0.

3.3.3 Erosion Computation

Fluvial erosion is the transportation process of bedrock material downhill caused by the stream of water from rainfall and running downhill. In the stream power equation, such erosion is modelled by water accumulation and discharge rate at a local region. The change in elevation caused by fluvial erosion can be computed using the equation:

$$\frac{dh_i}{dt} = -kA_i^m S_i^n \tag{3.5}$$

Where k, m, and n are erosion constants described in Section ?.?, A_i denotes the total area of nodes that flow into N_i as defined by the set of stream trees T; A_i is called the drainage area of N_i

and is a key property required to compute the water accumulation at the local region, S_i denotes the slope of the local stream between N_i and N_{R_i}; S_i is to approximate how much accumulated water discharges from the region by measuring the steepness of the local stream.

Based on the stream flows computed, the drainage area A_i can be computed using the formula:

$$A_i = a_i + \sum_{d_n \in D_{N_i}} A_{d_n} \tag{3.6}$$

Where a_i denotes the area of N_i, D_{N_i} denotes all donor nodes of N_i. Note that this is a recursive formula, the computation of A_i depends on all $A_{D_{N_i}}$. Therefore the computation of the drainage area for each node has to be computed in reverse order (leaf to root) of trees $\in T$. Such order may be extracted by traversing T.

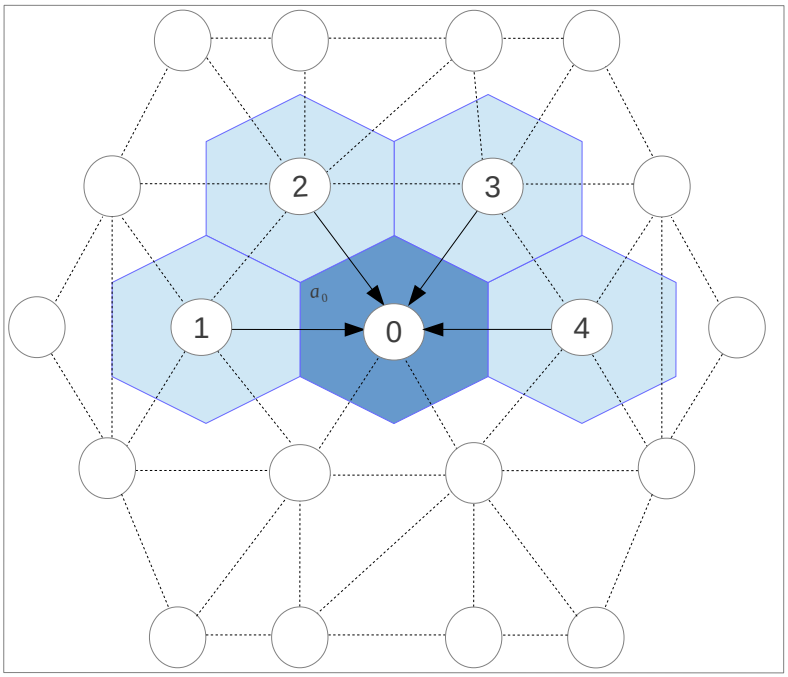


Figure 3.3: An example of the small graph to demonstrate a drainage area computation. Each node N_i is represented by a circle. Dashed lines represent an edge of the global graph. Arrow lines represent a directed edge of the tree. Hexagons represent a local region. In this figure, node 0 has four contributing nodes 1, 2, 3, 4. For node 0, let a_0 is the area of the local region, drainage area $A_0 = a_0 + A_1 + A_2 + A_3 + A_4$.

S_i is the slope between N_i and N_{R_i}, which can be computed using the formula:

$$S_i = \frac{h_i - h_{R_i}}{\Delta d} \tag{3.7}$$

Where h_i and h_{R_i} denote the elevation of N_i and its receiver N_{R_i}, Δd denotes the distance between N_i and N_{R_i} in the horizontal plane.

The implicit scheme suggested in [21] can efficiently solve Equation 3.5. For each node N_i and its receiver N_{R_i}, the Equation 3.5 can be rewritten as:

$$\frac{h_i(t+\Delta t) - h_i(t)}{\Delta t} = -KA_i^m \left(\frac{h_i(t+\Delta t) - h_{R_i}(t+\Delta t)}{\Delta d} \right)^n \tag{3.8}$$

When $n = 1$, equations for each node N_i is linear and can be solved explicitly to compute the new elevation after a timestep Δt:

$$h_i(t+\Delta t) = h_i + \frac{\frac{KA_i^m}{\Delta d} h_{R_i}(t+\Delta t)\Delta t}{1 + \frac{KA_i^m}{\Delta d}\Delta t} \tag{3.9}$$

To compute $h_i(t+\Delta t)$, the equation for $h_{R_i}(t+\Delta t)$ must be solved first, which means that each equation has to be solved in order (root to leaf) of each tree of T. Note that this is similar to computing Equation 5.3 in that they must be computed in reverse order (leaf to root). It means that one extraction of tree order of T can be used to compute both Equation 5.3 and 5.2.

3.3.4 Correction Process

When the simulation is generating high-altitude mountain ranges, the stream power erosion model can efficiently shape river networks and dendritic ridges. However, unrealistic high peaks may appear in regions with low discharge areas. Such unrealistic peaks create undesirable visual artefacts in the resulting terrain. This can be corrected by using a thermal erosion process [MKM89] as suggested in [1].

Thermal erosion combines thermal weathering and downward movement of rocks and sediments on slopes, due to temperature change and force of gravity. The different thermal expansion of water and material creates cracks and small intrusions at the material boundary, which causes the material to break and fall. This process creates regularity in the slope angle of ridges and hill slopes as a result. We apply the following formula at the end of the erosion process for each node N_i:

$$h_i = \begin{cases} h_i & \text{if } S_i \leq S_{max} \\ \tan(s_{max})\Delta d + rH & \text{if } S_i > S_{max} \end{cases} \tag{3.10}$$

Where S_{max} denotes a maximum slope angle allowed between N_i and its receiver N_{R_i}. If the elevation of N_i exceeds the S_{max}, the elevation is corrected to restrict the slope to a prescribed range.

3.3.5 Computational Limitation of the Simulation

Computational performance is one of the desirable properties of procedural generation methods. Terrain generation methods are usually to assist the human artist authoring terrain to create a desired digital asset for computer graphics applications or to generate random landscapes on the fly for background in real-time applications such as video games. For a method to assist the human artist effectively, the method must be able to be computed in a time that it can provide reasonable interactivity. For real-time applications, methods that cannot provide real-time performance would introduce several undesirable loading sessions to end-users. The computational performance of modern hardware now heavily relies on an increasing number of processing cores. In order to exploit modern hardware, a series of computation in the algorithm must be parallelised.

More densely sampled meshes can provide better-approximated simulation results of the stream power equation. Figure 3.4 shows that as the number of points grows, the method can generate terrain of more detail. However, the method in [1] is limited in size due to providing reasonable computational performance. The computational simulation of the stream power equation described in [1] has $O(n)$ complexity, where n is the total number of sample points of the geometric mesh. Although the linear complexity is considered efficient to compute, the computation time of the simulation would grow up to the point that it can not complete the computation in acceptable time for a large number of points such as over millions. This limitation is because their implementation only exploits a single core of the processing unit.

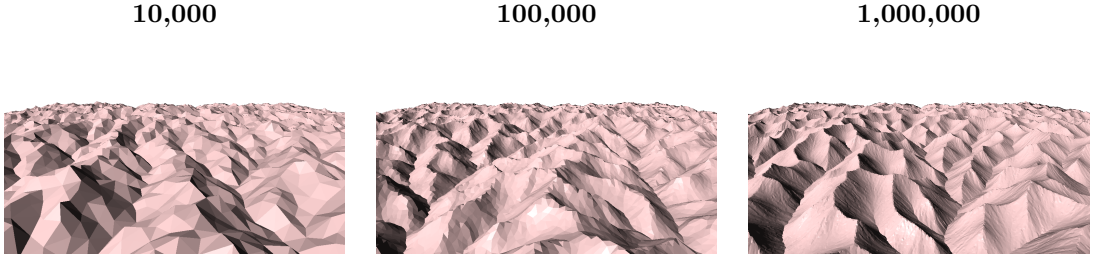


Figure 3.4: The comparison of outputs of different density meshes. Numbers on the top are the total number of points on the mesh. Pictures below numbers are the corresponding output.

Graphics processing units (GPUs) are a widely available computing hardware with a highly parallel structure that consists of thousands of processing cores. GPUs are originally designed to accelerate 3D graphics rendering, but they have been getting much attention for computationally demanding general-purpose applications due to their capability to provide significantly higher computational throughput than CPUs on massively parallel problems. However, the physical architecture and processing mechanism of GPUs are significantly different from conventional CPUs. To exploit GPUs efficiently, a parallel algorithm that specifically targets GPUs must be developed. The goal of this study is the development of the parallel GPU algorithm for the computational

simulation of the stream power equation to overcome the size limitation of the simulation. More details of GPU computing is discussed in the following chapter.

3.4 Summary

Complex landform features in mountainous terrain result from the combined action of tectonic uplift and fluvial erosion. Such land forming process can be modelled by the stream power erosion model introduced in this chapter. The basis of the model is a partial differential equation called the stream power equation. The simulation of the equation is used to generate large scale terrain in this study.

Continuous fields of the system can be approximated by a geometric mesh connecting. The mesh must be able to represent the terrain forming process naturally. Association of random point sampling and Delaunay triangulation can effectively provide such mesh.

Computational simulation of the equation is discussed in detail. The implicit scheme suggested provides an efficient way to solve the equation for a large time step. The algorithm suggested for the simulation shows the linear complexity, but it still suffers from the long computation time as the size of the simulation grows. This computational limitation may be overcome by exploiting parallel hardware such as GPU discussed in detail in the following chapter.

Chapter 4

GPU Computing

Graphics processing units (GPUs) are a widely available computing hardware with a highly parallel structure that contains thousands of processing cores. Although GPUs was originally designed for the needs of accelerating 3D graphics, the parallel processing capability of GPUs have also made them used in many computationally demanding applications other than computer graphics recently such as computational science and image processing.

To fully utilise the computing capability of GPUs properly, a parallel algorithm that specifically targets GPUs must be developed due to their different architecture and processing mechanism. This study proposes the parallel algorithm for the stream power equation simulation, which is specifically designed to run on GPUs to exploit their computing capability more effectively.

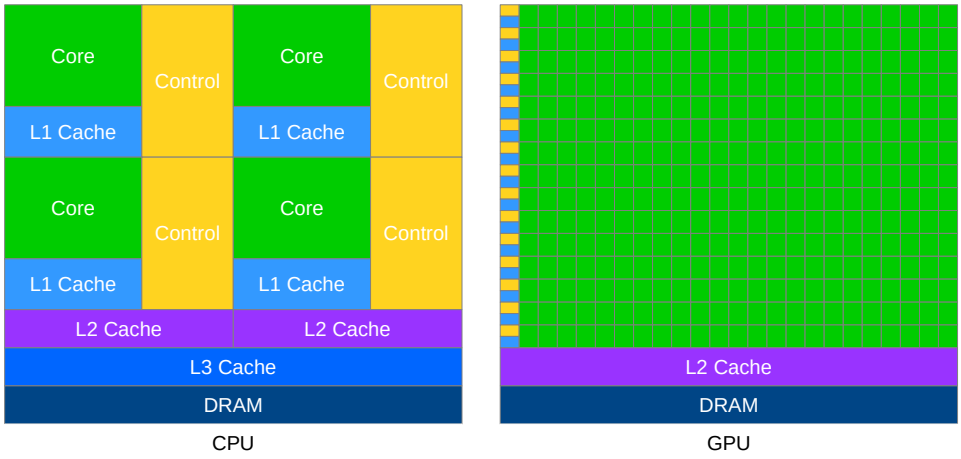
In order to implement the parallel algorithm to run on GPUs, there are two commonly used tools, OpenCL and CUDA. OpenCL (Open Computing Language) is a framework for a writing program that executes on parallel hardware such as CPUs, GPUs, DSPs (digital signal processors), and FPGAs (field-programmable gate arrays). CUDA is a framework developed by NVIDIA for a writing program that executes on their GPUs. NVIDIA is the leading GPU vendor and also known better support for general applications of GPUs than OpenCL. This study uses NVIDIA CUDA framework throughout this study to implement the newly proposed algorithm.

This chapter introduces the NVIDIA technology-based general-purpose GPU computing, which helps understand our parallel implementation of the stream power equation simulation presented in the following chapter. The content presented in this chapter is a summary of important information that is relevant to this study from NVIDIA CUDA Programming Guide [23] and NVIDIA Turing architecture white paper [24].

4.1 NVIDIA GPU Architecture

The CPU and GPU serve different purposes. The GPU architecture is specialised for massively parallel computations, whereas the CPU is designed to focus on the performance of its smaller number of cores on individual tasks. Figure 4.1 gives a high-level comparison of the CPU and GPU architecture comparison. While CPUs consist of few cores and allocates more transistors for the control unit and the cache memory, GPUs devote most of the transistor to data processing cores. The CPU is suited to workloads that require fast memory latency or high per-core performance. The cores on the GPU, due to their lack of cache memory, they can not provide good memory access latency but, the GPU can outperform the CPU on the massively parallel problems that enable the GPU to hide poor memory latency by exploiting its parallel processing capability.

Figure 4.1: High-level comparison diagram of CPU and GPU architectures.



NVIDIA GPUs employ a unique architecture called SIMT (Single-Instruction, Multiple-Thread) architecture. The architecture is built on a scalable array of streaming multiprocessors (SMs) where each processor is designed for the concurrent execution of thousands of threads. A GPU enumerates and distributes threads to the available SMs when the CUDA kernel launches. Each SM can run multiple threads concurrently.

SMs manage threads in groups of 32 parallel threads called warps. An SM partitions threads into warps and each warp get scheduled by a warp scheduler. Threads initialised in the same warp share the same program address. However, each thread has a separate instruction address counter and register state, enabling them to diverge and execute independently.

Threads in the same warp execute one common instruction at a time. The optimal efficiency is only achieved when all threads in a warp are on the same execution path. If the branch divergence occurs within a warp, the warp sequentially executes each branch execution path, disabling threads in the other path. Branch divergence could affect the performance of the application because there will be a decrease in computational throughput due to the deactivated threads.

Continuous enhancement to the architecture has been made through the seven architecture generations since CUDA framework was introduced. At the time of writing Turing architecture is the seventh generation, and this architecture introduced in Section 4.4 is used for all the experiments of this study.

4.2 NVIDIA CUDA Programming Model

General-purpose computing on graphics processing units (GPGPU) exploits the GPU for computationally expensive applications other than computer graphics such as computational science, and image processing. CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for general-purpose applications of their GPUs. CUDA provides programming interfaces in popular high-level languages such as C, C++, Fortran, and Python, which enhances the accessibility for the programmers. This study employs CUDA C++ that extends the standard C++ language. The CUDA programming model requires the programmer to partition the problem into fine-grained sub-problems that can be solved independently in parallel.

CUDA employs heterogeneous programming model that breaks down the application into two sections, which are referred to as a term *host* (CPU) and *device* (GPU). The *host* executes the main sequential part of the program and launches a special user-defined function called kernel for parallel execution of threads on the *device*. CUDA allows concurrent execution of multiple kernels and the *host* program does not necessarily have to wait for the kernels to complete execution before continuing execution.

A CUDA kernel is a user-defined function that is executed N times in parallel by N different threads where N denotes the total number of threads. A *host* side of the application is usually responsible for invoking a kernel function on a *device* side. CUDA threads are organised in a hierarchy of two levels. A grid consists of thread blocks, and a thread block consists of threads. A maximum number of threads per block is up to 1024 threads on a modern GPU. A structure of the threads blocks can be equally shaped into 1D, 2D or 3D arrays of threads, which makes more sense with many parallel applications that involve with multidimensional data such as a vector, matrix, or volume. A scalable programming model of CUDA automatically scales each block of threads to any available SMs on the GPU.

Listing 4.1 is a sample code snippet of a simple kernel definition and invocation. `__global__` declaration specifier is used to define a kernel function. Each thread that executes the kernel is given a unique index that can be accessed through the built-in variable `threadIdx`. A kernel invocation can be made from the *host* to the *device* using a `<<<...>>>` execution configuration

syntax that defines the dimension of the grid and blocks for the execution. The sample code invokes kernel for a block of N threads, and then the kernel function adds two vectors A and B of size N and stored the result into vector C.

Listing 4.1: An example of the simple kernel function `vecAdd` and its invocation by *host* side program. `vecAdd` sums two vectors `A` and `B`, and stores the result to the vector `C`.

```
//Kernel definition
__global__ void vecAdd(float* A, float* B, float* C){
int i = threadIdx.x;
C[i] = A[i] + B[i];
}
int main(){
...
//Kernel invocation with a block of N threads.
vecAdd<<<1, N>>>(A, B, C);
...
}
```

4.3 CUDA Memory

CUDA threads have access to several different types of the *device* memory separated from the *host* side during the execution. Data transfers between the *host* memory and the *device* memory have much lower bandwidth than data transfers between memories in the *device*. Therefore minimising the data transfers between the *host* and the *device* is essential for the high performing solution.

Data transfers in GPU have much higher latency than CPU applications. In order to maximise memory throughput, CUDA supports several types of memories. Depending on the access pattern of each type of memory, the overall throughput of memory accesses dramatically varies. Different types of memories are optimised for different usages. Following are the *device* side memory types that the programmer has access to:

- **Global memory** is accessible from any threads in a grid across the same program. It resides in off-chip DRAM and provides the largest memory space, having the slowest access latency. Although access performance is poor, the overall performance of the program can be significantly improved by coalescing global memory transactions. The *device* memory is accessed by 32, 64 or 128 bytes memory transactions. When threads in the warp access global memory, the hardware coalesce the request into one or more transactions depending on the size of the memory that is each thread accesses and distribution of the memory addresses accessed. For instance, accessing sequentially well-aligned 4-byte words from all threads in the warp (accessing 128 bytes in total) can coalesce into a single 128 bytes memory transaction.

- **Local memory** is a private space for a thread residing in off-chip DRAM and is used as overflow for registers. Thus, it has the same high memory access latency as global memory.

However, local memory access in CUDA is fully coalesced in a way that consecutive 4 bytes words are accessed by consecutive thread indexes.

- **Shared memory** is an on-chip memory shared between threads in the same block. It provides much lower memory access latency than global or local memory. Shared memory is sliced into modules called banks. Access to each bank is one at the time, but it can be made concurrently as long as there is no conflict between two or more threads requesting the same bank. To effectively access shared memory should therefore take account on avoiding conflict between accesses.

- **Constant memory** is read-only and an on-chip caching method of accessing global memory. This memory cached in a constant cache for all threads to access the same value at the same time. Global memory access only occurs for the case of a cache miss. The maximum size of the memory is 64KB.

- **Registers** are an on-chip memory automatically used to store the local variables belonging to each thread. If the registers cannot store all the local variables required by the threads, local memory must be used. The access time for registers is effectively zero extra clock cycles. The programmer cannot explicitly allocate registers for the local variables, and the only consideration for the programmer is how many registers are used by each thread.

- **L1/L2 cache** is to cache all access to local memory for each thread. L1 is on-chip cache memory provided per SM, and the L2 is off-chip cache memory provided for the GPU.

4.3.1 Atomic Memory Operation

CUDA support atomic memory operation to prevent a race condition between threads accessing the same memory address. It is provided as a built-in function and can be used for global and shared memory. An atomic function guarantees to perform a read-modify-write memory operation without interference from other threads. The atomic operation is useful and essential for many parallel applications, but care must be taken for the best performance because a series of atomic operations on the same memory address by threads can only be performed sequentially.

4.4 Turing Architecture

The latest NVIDIA Turing architecture achieves significant performance improvement over previous Pascal architecture due to architectural changes. NVIDIA claims that a new design of Turing SM achieves 50% improvement in performance per CUDA core. Newly introduced GDDR6 device memory provides 20% enhanced power consumption while maintaining the same data transfer rate

as compared to the previous Pascal architecture. This section introduces selected architectural changes of Turing GPUs, which are important for the parallel implementation discussed in the following chapter.

4.4.1 Turing SM

Turing SM is divided into four blocks, each of which has its own warp scheduler, register file, and dispatch unit. There are also newly introduced units that our work does not exploit, such as Tensor core, and RT core, more details of which can be found in the architecture white paper [24]. Figure 4.2 gives a high-level comparison of the Turing and Pascal SM.

Figure 4.2: High-level comparison diagram of the Turing and Pascal SM.

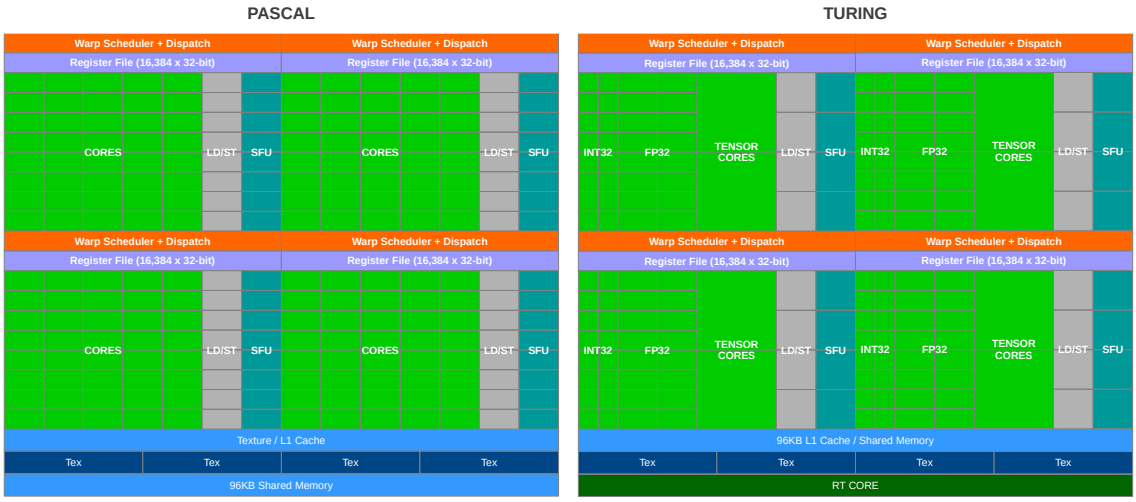


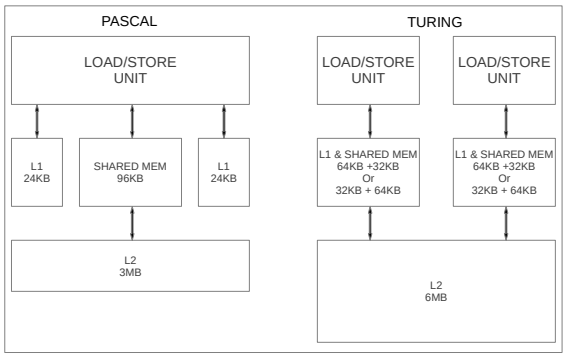
Table 4.1 shows the comparison of compute-cores per SM in Pascal and Turing architectures. Turing SM has a new independent integer datapath feature by allocating half amount of space to 32 bits integer arithmetic unit. Computations in a modern shader are mainly floating-point instructions along with simpler integer pipe instructions such as address math and branching. While it used to be that floating-point instruction pipeline has to stall to execute integer instruction, Turing GPUs enables concurrent execution of floating-point and integer instructions for the first time in the consumer GPU line-up of NVIDIA. The concurrent execution of integer and floating-point pipeline achieves 36% additional throughput for floating-point arithmetic.

Table 4.1: Compute Cores per SM Comparison (Pascal and Turing)

	Pascal SM	Turing SM
FP64 Cores	4	2
FP32 Cores	128	64
INT32 Cores	N/A	64

Turing SM unifies L1 cache and shared memory. Shared memory provides high bandwidth, low latency, and consistent performance (no cache misses), but it requires explicit management by a programmer. The new unified memory design narrows the performance gap between the explicitly tuned shared memory management and direct access to the device memory. The new unified memory design also allows an application to configure memory structure into either 64KB of L1 cache and 32 KB of shared memory or vice versa. This configurability helps achieve better utilisation of expensive SRAM resource depending on applications demands. Figure 4.3 shows the new unified memory design compared to the Pascal architecture.

Figure 4.3: High-level comparison diagram of the new Turing unified memory and the previous Pascal.



4.4.2 Independent Thread Scheduling

In the previous architectures, a single program counter is shared among the threads in the warp. Threads in the same warp execute one common instruction at a time. When threads in the same warp diverge to multiple execution paths, each execution path must be sequentially processed at a time, and threads that are not on the current path get disabled. Diverged threads in the warp later reconverge implicitly once the execution of all branches is complete.

In Volta and Turing, independent thread scheduling allocates program counter and call stacks per thread instead per warp. Diverged threads in the same warp can be executed concurrently and not required to reconverge unless the programmer explicitly converges them. Warp synchronise functions can be called to reconverge diverged threads in the same warp once branch execution

complete for the case of the reconvergence required.

4.5 Summary

This chapter introduced the NVIDIA GPU architecture and CUDA programming to help to understand the parallel implementation of the stream power erosion simulation presented in the following chapter.

The GPU architecture is designed to be effective for massively parallel computation. NVIDIA GPUs used in this study employs SIMT architecture, which consists of an array of SMs. CUDA GPU automatically scales parallel threads to any available SMs.

C++ is used for the implementation in this study out of various high-level programming languages supported by CUDA. The CUDA application is heterogeneous and can be broken down to two sections *host* (CPU) and *device* (GPU) where the *host* invokes the kernel function that executes parallel threads on the *device*.

CUDA supports different types of memories for the *device* side of the application. Each type of memories is optimised for different purposes; therefore, care must be taken for a high-performing application.

The latest Turing GPUs used for all the experiments for this study provides enhanced power efficiency and computational performance compared to the previous architecture. The configurable unified memory enables efficient use of L1 cache for the application that does not exploit shared memory up to its capacity. Independent thread scheduling allows diverged thread in the same warp to be executed concurrently. However, care must be taken for the case that the program requires diverged threads to be explicitly reconverged after branch execution.

Chapter 5

Parallel Implementation

The stream power equation simulation can be used to generate geologically more accurate terrain of large spatial scale. The method introduced in [1] also provides high-level control method. However, while more densely discretised simulations can add high-quality details to the resulting terrain, the unoptimised implementation suggested in [1] can not provide reasonable computation time for simulations of such larges spatial scales. The computational performance of modern hardware now heavily relies on an increasing number of processing cores. A series of computation in the algorithm must be parallelised to exploit the power of modern hardware.

The implementation in [1] is a sequential execution of the algorithm proposed in [21] has a computational complexity that is linear in the number of samples. Although linear complexity is considered efficient to compute, large simulation sizes still suffer from long computation time. The algorithm introduced in [21] also introduced a parallel execution model. However, the design of this model can not fully leverage the power of modern hardware such as multi-core CPUs, or GPUs and imposes limitations on parallelism and scalability. These limitations were later improved by the parallel algorithm proposed in [2], which fully utilise the power of modern CPUs and shows performance improvement on GPUs. However, the algorithm proposed in [2] is a parallel algorithm that targets any multi-core hardware, and therefore, the performance it provides on GPUs is not optimal. This study improves on this previous work by developing a parallel algorithm that specifically targets GPUs.

This chapter introduces the parallel implementation of the newly proposed algorithm written in CUDA. This study also implemented the original terrain generation method [1] using the parallel algorithm proposed in [2]. The implementations of the previous algorithm to generate terrain are written for multi-core CPUs and GPUs. Moreover, the previous parallel algorithm was implemented in OpenACC that is a high-level parallel computing platform for easier learning and accessibility for users. However, the algorithm has never been implemented in CUDA that provides

lower-level access to the GPU to explore an opportunity to achieve better performance. This study is the first one that presents the CUDA implementation of the parallel algorithm proposed in [2].

5.1 Data Structure

This section introduces commonly used data structures for different implementations presented in this chapter. In order to compute the stream power equation simulation, discretised approximation of terrain domain is stored as a graph data structure where each node represents each local region. Each node stores its geometric information such as position at a horizontal plane, elevation, and surface area. Each node requires to have storage to store simulation data that updates every iteration such as stream-flow and drainage area. There are some constant simulation data that each node stores, such as uplift rate, and maximum slope angle limitation for the correction process.

The implementation declares the separate pointer variables to construct dynamic arrays for different data. For the CPU targeted C++ implementation, we use *std::vector* provided in C++ Standard Template Library instead of the dynamic array. In this chapter, an array means either a dynamic array pointed by pointer variable or a *std::vector* container.

5.1.1 Graph Representation

Terrain in the stream power erosion model is represented by a geometric graph. In the implementations, an adjacency list is used to represent a graph. An adjacency list is an array for each node $N_i \in G$, store a list of the nodes adjacent to it. Each node $N_i \in G$ typically has an array of its own adjacency list where the size of an array is a total number of its adjacent nodes. As shown in Figure 5.1, All adjacency list array for each node N_i are stored in the one large array of length $\sum_{i=0}^{n} NN_i$ where n denotes a total number of nodes in the G, NN_i denotes a total number of adjacent nodes of each node N_i. Each adjacency list can be accessed using a stored offset and a length of the list of each node N_i. Because the graph used in the model has an irregular structure that total numbers of neighbouring nodes of each node N_i vary, this representation provides spatial efficiency and simple access pattern.

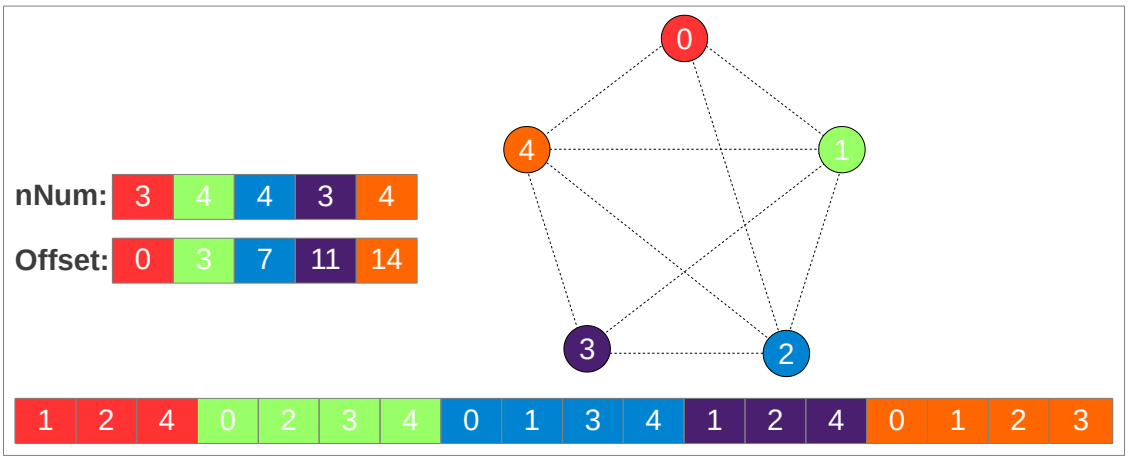


Figure 5.1: An example of the small graph of 5 nodes and how 1D global adjacency list array is constructed. Circles represent a node, and numbers are an identifier. Dashed lines represent an edge of the graph. `nNum` is an array to store a total number of neighbours of each node. `Offset` is an array to store start index at the adjacency list array for adjacency list of each node.

In the code implementation, the following arrays are declared to represent graph:

```
int* neighbour;
int* offset;
int* nNum;
float* distance
```

Where `neighbour` is a 1D array to store the adjacency lists of all nodes, `offset` stores offset into neighbour for each node, and `nNum` stores a total number of neighbours for each node, severa stages of the pipeline needs distance between neighbouring points in the horizontal plane, and it is computed at the initialisation and stored in `distance`. Neighbours of node N_i can be accessed from `neighbour[offset[i]]` to `neighbour[offset[i] + nNum[i] - 1]`.

5.1.2 Simulation Data

In order to simulate the stream power erosion model, each node $N_i \in G$ have geometric information stored in the following arrays:

```
glm::vec2* point;
float* height;
float* area;
```

Where `point` stores the geometric position of each node N_i in the horizontal plane, `height` stores the current elevation of each node N_i, and `area` stores the local surface area of each node N_i.

A stream-flow is used to approximate the total water accumulation and discharge rate at each local region represented by each node $N_i \in G$. A set of stream-flows T for each node is updated every iteration, and it is stored in the following arrays:

```
int* receiver;
int* dNum;
int* donor;
```

Where `receiver` stores the index of receiver node N_{R_i} for each node N_i, `dNum` stores the total number of donors for each node N_i, and `donor` is an array with the same size as `neighbour`, which stores all the donor indexes of each node N_i.

In order to compute a new elevation for each node $N_i \in G$, the following arrays stores key values:

```
float* uplift;
float* drainageArea;
float* maxSlope;
```

Where `uplift` stores the uplift rate u_i for each node N_i, `drainageArea` stores the drainage area A_i for each node, and `maxSlope` stores the maximum slope angle S_{max_i} for each node N_i.

Lastly, we define the following constants for the simulation parameters:

```
#define SIZE_GRID 5.0e+4f
#define MAX_UPLIFT 5.0e-4f
#define KEQ 5.61e-7f
#define MEQ 5.0e-1f
#define DT  2.5e+5f
#define NO_FLOW -1
```

Where `SIZE_GRID` defines the horizontal and vertical size of the plane in metres, `MAX_UPLIFT` defines the maximum uplift rate in my^{-1}, `KEQ` defines the constant k in y^{-1}, `MEQ` defines the constant m, `DT` defines the timestep Δt in y where m denotes a metre, and y denotes a year. The value -1 is used in `NO_FLOW` to define no receiver status.

5.2 Initialisation

This section introduces the CUDA implementation of the simulation initialisation process introduced in [1]. The initialisation consists of three stages; random point generation using the Poisson disc sampling, graph construction by performing the Delaunay triangulation, and assigning the local surface area value for each node $N_i \in G$ by computing the Voronoi diagram of the constructed graph. All data are stored in the GPU *device* memory to avoid expensive data transfer between *host* and *device* throughout the simulation.

5.2.1 Random Point Generation

The first stage of the initialisation is a random point generation over the domain. Poisson disc sampling is used to generate a set of random points, which is a blue noise sampling method based on nonuniform Poisson distribution where all samples are at least distance r apart from each other. Sampling is indirectly controlled using a density parameter, r that defines the minimum distance between samples but not the number of samples directly. This method can generate a set of points that is tightly-packed and in a naturally distributed pattern. An efficient linear time sequential algorithm introduced in [25] is the most commonly used approach, but it demands long

computational time to generate a large set of samples such as over million. Therefore, Poisson disc sampling for GPUs is implemented in CUDA for faster initialisation. The CUDA implementation of Poisson disc sampling is based on but not exactly the same as the algorithm introduced in [26].

In order to generate random points, the algorithm first covers the horizontal plane with a regular grid that its size of each grid cell is $\frac{r}{\sqrt{2}}$ where r denotes a density parameter that defines the minimum distance between samples. Each grid cell is given a status that states *accepted*, *rejected*, and *active*, where *accepted* states that the cell is accepted and included in the set of point being generated, *rejected* states that the cell is rejected and not included in the set, *active* states that the cell has not yet been accepted, and it still needs to be validated. Three status are stored as an integer value; $active = 0$, $accepted = 1$, and $rejected = 2$. An initial status for all grid cell is the *active*.Each grid cell is also given a unique priority to determine which one is on priority when two are too close to each other. The algorithm then repeats the following sample generation and validation process for k times; generate the random samples in the grid cells that are not validated to be accepted yet, and validate the samples (whether they are not too close to each other) in each grid. In the implementation, the sampling converges at around 30 iterations, so it uses $k = 30$.

The size properties of a grid can be computed using the grid cell size $\frac{r}{\sqrt{2}}$ as the following C++ code:

```
const float RADIUS = r;
const float CELL_WIDTH = RADIUS / std::sqrt(2.0f);
const int COLS = std::ceil(SIZE_GRID / CELL_WIDTH);
const int ROWS = std::ceil(SIZE_GRID / CELL_HEIGHT);
const int SIZE = COLS * ROWS;
```

Where *COLS* is a number of columns in the grid, *ROWS* is a number of rows in the grid, and *SIZE* is a total number of cells in the grid.

Initial data can be copied into the following arrays of the size of the grid in the *device* memory:

```
curandState* state;
glm::vec2* grid_points;
int* priority;
int* status;
```

Where `state` stores the sequence of pseudo-random numbers for CUDA random number generator, `grid_points` is stores the randomly generated sample for each grid cell, `priority` is stores randomly generated unique priority value for each grid cell, and `status` stores the status of acceptance of the sample in each grid cell.

The sampling process can then begin. As shown in Listing 5.1, both sample generation and validation process run on the GPU. Both kernels are repeatedly launched for k times. Each stage is implemented in the separate kernels because the generation has to finish before the validation and vice-versa. Therefore, `cudaDeviceSynchronize()` function must be called between kernel launch to ensure for them to wait for the other kernel to complete before launching.

Listing 5.1: An example of the *host* side implementation of Poisson disc sampling that launch sample generation and validation kernel for 30 times. `generatePoints` is a kernel to generate a new sample in the grid cells that are not accepted yet. `validatePoints` is a kernel to validate samples in the grid cells that are still on active status.

```
for(int i = 0; i < 30; ++i){
  generatePoints <<<RAND_BLOCK, THREAD>>>(grid_points, status, state);
  cudaDeviceSynchronize();
  validatePoints <<<BLOCK, THREAD>>>(grid_points, priority, status, i);
  cudaDeviceSynchronize();
}
```

The kernel function `generatePoints` creates a thread that generates a sequence of pseudo-random numbers used to produce samples for grid cells that have not yet been accepted. Each thread is given an equally divided number of grid cells to produce a new sample for the next validation. Listing 5.2 shows the implementation of `generatePoints`. Each thread iterates through the given grid cells to generate a new sample. A new sample is only generated for the cell that is on *active* status. Note that random number is generated even for the cell that is not *active* status, or when it is out of range. This dummy generation is because the parallel random number generator (PRNG) by CUDA requires all threads given a sequence of random numbers have to generate the random number. Otherwise, the whole sequence may break.

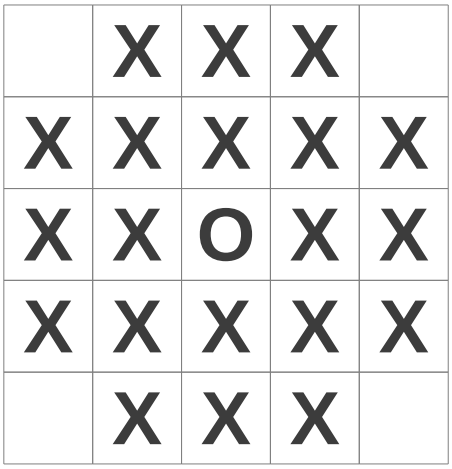
Listing 5.2: The CUDA kernel implementation of sample generation stage. Each tread is given a sequence of pseudo-random numbers to generate a new sample for an equally divided number of grid cells.

```
__global__ void generatePoints(glm::vec2* points, int* status, curandState* state){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int s_idx = idx * POINTS_PER_GENERATOR;
  int e_idx = s_idx + POINTS_PER_GENERATOR - 1;
  for(int i = s_idx; i <= e_idx; ++i){
    if(i < SIZE){
      if(status[i] == 0){
        int row = i / COLS;
        int col = i % COLS;
        float minX = col * CELL_WIDTH;
        float maxX = min(minX + CELL_WIDTH, SIZE_GRID);
        float minY = row * CELL_WIDTH;
        float maxY = min(minY + CELL_WIDTH, SIZE_GRID);
        float x = randFloat(state + idx, minX, maxX);
        float y = randFloat(state + idx, minY, maxY);
        points[i] = glm::vec2(x, y);
      }else{
        randFloat(state + idx, 0.0f, 1.0f);
        randFloat(state + idx, 0.0f, 1.0f);
      }
    }else{
      randFloat(state + idx, 0.0f, 1.0f);
      randFloat(state + idx, 0.0f, 1.0f);
    }
  }
}
```

Each grid cell is validated by checking the distance between the samples and the samples in the 20 surrounding cells, as shown in Figure 5.2. Listing 5.3 shows the implementation of the kernel function `validatePoints`. The validation only proceeds for the cells that are on the *active* status, and the validation process is as follow; For each grid cell given, if the surrounding cell has

not yet been rejected, compute the distance between samples, it then compares priority between the cell and the current checking surrounding cell, if the priority is lower than the current cheking surrounding cell, it terminates the kenel (Validation fail). If the validation process success without fail until it checks the last surrounding cell, the cell gets accepted. The sample gets rejected if the validation process fails at the last iteration (30^{th}) of the sampling process.

Figure 5.2: An visual example to show which surrounding cells are to be checked to validate a sample in the cell. The grid cell with a circle inside has the sample that is currently on validation. The grid cells with an X inside have samples that need to be checked for validation.



Listing 5.3: The listing presents CUDA kernel implementation of sample validation stage. Each thread is given a grid cell to validate a sample inside the grid cell.

```
__global__ void validatePoints(glm::vec2* points, int* priority, int* status, const int N){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= SIZE) return;
  if(status[idx] == 1) return;

  glm::vec2 thread_p = points[idx];
  unsigned int thread_priority = priority[idx];

  int col = idx % COLS;
  int row = idx / COLS;

  for(int i = -2; i <= 2; ++i){
    int n_row = row + i;
    if(n_row < 0 || n_row >= ROWS) continue;
    int abs_i = abs(i);
    for(int j = (abs_i == 2 ? -1 : -2); j <= (abs_i == 2 ? 1 : 2); ++j){
      int n_col = col + j;
      if((i == 0 && j == 0) || n_col < 0 || n_col >= COLS ) continue;
      int n_idx = n_row * COLS + n_col;
      int n_status = status[n_idx];
      if(n_status != 2) {
        glm::vec2 n_p = points[n_idx];
        float distance = glm::distance(thread_p, n_p);
        if(distance < RADIUS){
          if(n_status == 1){
            if(N == 29) status[idx] = 2;
            return;
          }
          unsigned int n_priority = priority[n_idx];
          if(thread_priority < n_priority){
            if(N == 29) status[idx] = 2;
            return;
          }
        }
      }
    }
  }
  status[idx] = 1;
```

```
}
```

A final post-processing step is required after the sampling process because the valid and invalid samples are stored in `grid_point` in arbitrary order. In order to copy only valid samples out of the array, it first sorts `status` and `grid_point` as a key-value pair. A parallel radix-sort can be performed efficiently since the maximum value of the sort key uses only two bits. Once `grid_point` is sorted, it launches kernel that each thread checks values in `status` at its thread index and the next index to find the boundary index between sets of accepted and rejected samples, which is to identify how many samples have been accepted. For each thread index i, i is the boundary index if the $status_i = 1, status_{i+1} = 2$. All accepted samples can then be copied into `point`.

5.2.2 Graph Construction

The second stage of initialisation is the construction of the geometric graph from the generated random points. The graph is computed by performing Delaunay triangulation for the generated points. In computational geometry, for a given set of discrete points P in a plane, a Delaunay triangulation is a triangulation $DT(P)$ such that no point in P is inside the circumcircle of any triangle in $DT(P)$. In order to perform fast Delaunay triangulation on the GPU, the algorithm and its implementation introduced in [27] is employed. A geometric graph can be stored according to the structure of the set of triangles computed.

In order to store the graph, the total number of neighbours for each node can first be computed by counting Delaunay triangles that each point lies on. The total number of neighbours for nodes can be two cases; Let the total number of triangles that the point lies on be t, if the node is at the boundary of the set of Delaunay triangles, the total number of neighbours is $t+$; otherwise, the total number of neighbours is t. To reduce the computational cost of the process, it allocates $t + 1$ space for the neighbour list for all node. Listing 5.4 shows the CUDA kernel implementation of the procedure. `count` is first initialised all elements of the array with 1, and then `countTriangles_kernel` is launched. The result then can be copied into `nNum`.

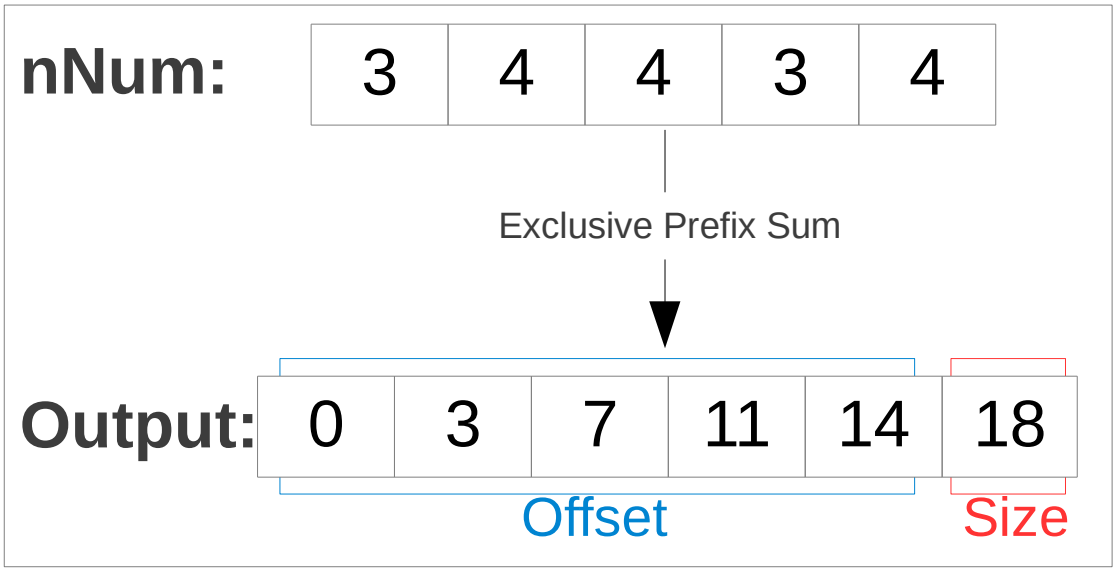
Listing 5.4: The CUDA kernel implementation to construct `nNum` array. `countTriangles_kenel` function is to count triangles of each node is part of. `increaseAllCount_kernel` function is to increase all counts by one.

```
__global__ void countTriangles_kernel(Tri* triangles, unsigned int* count){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= D_SIZE_TRI) return;
  Tri t = triangles[idx];
  atomicInc(count + t._v[0], D_SIZE);
  atomicInc(count + t._v[1], D_SIZE);
  atomicInc(count + t._v[2], D_SIZE);
}
```

To compute the index offset into the adjacency list for each node and the size of `neighbour`

that stores the adjacency lists for all nodes, an exclusive-prefix-sum of `nNum` can be computed. The implementation exploits the `ExclusiveSum` function provided in the CUB library for efficient parallel execution on the GPU. Let n be the size of the `nNum`, performing the exclusive-prefix-sum of the array produce an array of size $n+1$ where the elements at $[0, n-2]$ are the index offset for each node, and the last element at $n-1$ is the size of `neighbour`. Figure 5.3 provides an example of the computation for the small system of 5 nodes.

Figure 5.3: The figure is to visually demonstrate how the index offset and the size of the adjacency list are computed by performing an exclusive-prefix-sum.



The final step is the computation of the neighbour lists (adjacency list). `neighbour` is first constructed with the size computed in the previous step, and then all elements are initialised with -1. The value -1 indicates an empty slot in the later steps. The kernel shown in Listing 5.5 is then launched to compute and store the neighbour list. The kernel function `constructNeighbourList` create a thread that is given one Delaunay triangle. For three vertexes of the given triangle, each thread calls the `addNeighbour` function to store the other two vertexes as a neighbour to each vertex.

Listing 5.5: The CUDA kernel implementation that construct the 1D global adjacency list array.

```
__global__ void constructNeighbourList(Tri* triangles, int* neighbour, unsigned int* offset,
    unsigned int* nNum)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= D_SIZE_TRI) return;

  Tri t = triangles[idx];
  int* n0 = neighbour + offset[t._v[0]];
  int* n1 = neighbour + offset[t._v[1]];
  int* n2 = neighbour + offset[t._v[2]];

  unsigned int nN0 = nNum[t._v[0]];
  unsigned int nN1 = nNum[t._v[1]];
```

```
  unsigned int nN2 = nNum[t._v[2]];

  addNeighbour(n0, nN0, t._v[1]);
  addNeighbour(n0, nN0, t._v[2]);
  addNeighbour(n1, nN1, t._v[0]);
  addNeighbour(n1, nN1, t._v[2]);
  addNeighbour(n2, nN2, t._v[0]);
  addNeighbour(n2, nN2, t._v[1]);
}
```

Listing 5.6 present the implementation of the `addNeighbour` function, which iterates through the neighbour list to add a new neighbour index (`n_idx`) to an empty slot in the list. There are two cases to check for each iteration of the loop. Firstly, if the *n_idx* has already been stored, it terminates the function to avoid duplicated storing. Secondly, if the slot is empty, it performs an atomic compare and swap operation to store a new neighbour to the slot. The atomic operation is required because there are groups of triangles that share one or two vertexes with each other, and there is a race condition when two threads try to access the same slot at the same time. `atomicCAS` function lets the first arrived thread write to the empty slot, and indicates to other threads that the slot is no longer empty.

Listing 5.6: The CUDA *device* function implementation that adds a new adjacency to the empty slot in the neighbour list.

```
__device__ bool addNeighbour(int* neighbour, unsigned int nNum, int n_idx){
  for(int i = 0 ; i < nNum; ++i){
    int result = neighbour[i];
    if(result == n_idx) return true;
    if(result == -1) {
      result = atomicCAS(neighbour + i, -1, n_idx);
      if(result == -1 || result == n_idx) return true;
    }
  }
}
```

5.2.3 Area Computation

The final stage is to assign a local surface area value to each node. Approximation of the local surface area for each node is an important property to compute the stream power equation. A set of Delaunay triangles $DT(P)$ for a given set of points P is a dual graph of a Voronoi diagram $VD(P)$ for P. A Voronoi diagram is a partition of a plane into regions that correspond to each point of a given set of points P. Each of these regions consists of every point in the Euclidean plane whose distance to its corresponding point in P is closer than any other points in P. These regions are called a Voronoi cell. Each Voronoi cell in Voronoi diagram $VD(P)$ for the given set of points P provides a reasonable and well-distributed approximation for the local surface. In computational geometry, for the given set of points P, $VD(P)$ is a dual graph of $DT(P)$. A set of vertexes v_i of each Voronoi cell V_i for each node N_i is circumcentre of the set of triangles in $DT(P)$ that have N_i as one of the vertexes.

To compute a surface area of each Voronoi cell V_i for each node N_i, V_i is divided into the small

triangles by connecting each edge of V_i with N_i. It is efficient to sort the set of vertexes v_i of V_i with the angle to their centre N_i to identify the edges. Parallel radix sort can be performed efficiently for the case. In the implementation, the v_i are sorted into the clockwise order. In order to sort v_i for each N_i, the key and value 1D global array pair for all nodes is constructed. Each element in the key array consists of the major and minor key where the major key is the index of N_i, and the minor key is the angle of each Voronoi vertex in v_i.

To construct the key and the value array pair, an exclusive-prefix-sum of the `count` that stores triangle count for each node N_i is performed. Let n be the size of the resulting array, the elements at $[0, n-2]$ are the index offset for each node, and the last element at $n-1$ is the size of the key and the value array pair. The CUDA kernel function `constructSortKeyValue_kernel` shown in Listing 5.7 to compute the key and the value array pair.

In the `constructSortKeyValue_kernel` function in Listing 5.7, each thread computes the circumcentre of the given triangle. Note that the circumcentre computation must be performed in the double-precision otherwise, it may not produce a geometrically correct Voronoi diagram due to floating-point arithmetic errors. Then for each vertex of the given triangle, each thread loads the index offset, and the total numbers of triangles and calls the addKeyValuePair function to fill in the key and the value array pair.

Listing 5.7: The CUDA kernel implementation to construct the key and the value array pair.

```
__global__ void constructSortKeyValue_kernel(Tri* tri, unsigned int* offset,unsigned int* tNum,
      glm::vec2* point, unsigned long long int* keys, glm::vec2* values){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= D_SIZE_TRI) return;
  Tri t = tri[idx];
  int idx0 = t._v[0];
  int idx1 = t._v[1];
  int idx2 = t._v[2];

  //Get circumcentre
  glm::dvec2 circum = computeCircumcentre(point[idx0], point[idx1], point[idx2]);

  int offset0 = offset[idx0];
  int offset1 = offset[idx1];
  int offset2 = offset[idx2];

  addKeyValuePair(idx0, keys + offset0, values + offset0, p0, circum, tNum[idx0]);
  addKeyValuePair(idx1, keys + offset1, values + offset1, p1, circum, tNum[idx1]);
  addKeyValuePair(idx2, keys + offset2, values + offset2, p2, circum, tNum[idx2]);
}
```

Listing 5.8 presents the implementation of the `addKeyValuePair` function, which iterates through the key array to add a new key to an empty slot. The key allocates the lower 10 bits to the angle and the upper bits to the index of the node. In order to find the empty slot, atomic compare and swap operation is required to avoid a race condition between threads that share one or two vertexes.

Listing 5.8: The implementation of the *device* function to add a new key and value pair to the empty slot of the key and the value array pair.

```
__device__ bool addKeyValuePair(int idx, unsigned long long int* key, glm::vec2* value, const
    glm::vec2& p, const glm::dvec2& circum, unsigned int tNum){
  float angle = atan2f((p.y - circum.y), (p.x - circum.x));
  angle = angle < 0 ? angle += M_PI * 2.0f : angle;
  unsigned int bit_angle = angle * 1e+2;

  for(int i = 0; i < tNum; ++i){
    if(atomicCAS(key + i, 0, ((unsigned long long int)idx << 10) | (unsigned long long int)
        bit_angle) == 0){
      value[i] = glm::vec2(circum);
      return;
    }
  }
}
```

A final step is using the sorted v_i the local surface area a_i for each node N_i can be computed on the GPU by launching the CUDA kernel presented in Listing 5.9. In the kernel function `computeArea_kernel`, firstly, the N_i that lies on the boundary of graph is ignored otherwise, each thread iterates through the v_i to compute the a_i. The set of v_i can be accessed using the index offset for N_i. In each iteration, let idx be the index that is currently being processed, the area of the triangle of v_{idx}, v_{idx+1}, and N_i is computed and accumulated to the sum.

Listing 5.9: The CUDA kernel implementation that compute area for each node.

```
__global__ void computeArea_kernel(glm::vec2* point, glm::vec2* circum, unsigned int* offset,
    float* area, unsigned int* tNum){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= D_SIZE) return;
  glm::vec2 p0 = point[idx];
  if(isBoundary(p0, D_RADIUS)) return;

  glm::vec2* thread_circum = circum + offset[idx];
  unsigned int thread_tNum = tNum[idx];
  float areaSum = 0.0f;

  for(int i = 0; i < (thread_tNum == 2 ? 1 : thread_tNum); ++i){
    glm::vec2 p1 = thread_circum[i];
    glm::vec2 p2 = thread_circum[(i + 1) % thread_tNum];
    areaSum += computeTriArea(p0, p1, p2);
  }
  area[idx] = areaSum;
}
```

5.3 Stream-Flows Computation

A stream-flow represents water running between neighbouring nodes in the stream power erosion model. The stream-flow can be used to approximate the total accumulation and discharge rate of water in each node. The stream-flow model used for the simulation is a single direction flow model where water accumulated in each node N_i flows into the neighbour with the steepest descent slope called the receiver node N_{R_i}. Each node N_i is called the donor node $N_{D_{R_i}}$ of the receiver node N_{R_i}. Stream-flows between each node N_i and its receiver node N_{R_i} form a set of directed trees T, which cover the graph G and is a key property to compute the fluvial erosion rate for each node N_i in the later stage of the pipeline. A node that is not higher than any of its neighbours does not have the receiver node and serves as a root node of each tree. Finding the receiver node N_{R_i} for each node N_i is the first step in simulating the stream power equation. This section presents

parallel algorithms and their implementations for this stage of the pipeline.

5.3.1 Finding Receiver Nodes

In order to find the receiver node N_{R_i} for each node N_i, the slope angles between each node N_i and its neighbours need to be computed and compared between all neighbours. The slope angle is only computed for neighbours that have a lower elevation than N_i because it is searching for the steepest descent. Once the relevant slope angles to neighbours are computed, they can be compared, and the neighbour with the steepest slope becomes the receiver node by storing its index. There is no dependency between the procedures to find the receiver node N_{R_i} for each node. Therefore, the procedures can be done per node in parallel. The previously proposed parallel algorithm creates a thread per node to executes the procedure to find the receiver node N_{R_i} in parallel.

For the implementation of the algorithm, in order to search through neighbours, the lists of neighbours are stored in `neighbour`. The lists of neighbours can be accessed for each node using the offset index into the list and the total number of neighbours, which are stored in `offset` and `nNum`. Some nodes may have neighbours stored as -1, indicating an empty space in the list and should be ignored. The slope between the nodes is computed based on their distance on the horizontal plane, and their elevation, which are stored in `distance` and `height`. Lastly, once the receiver node N_{R_i} found, it is stored in `receiver`.

Listing 5.10 presents C++ / OpenMP implementation of the parallel algorithm to find the receiver node N_{R_i} for each node N_i. The implementation uses two variables `maxS` and `maxN` to keep track of the neighbour with the steepest slope throughout the procedure where `maxS` keeps track of the slope of the current steepest, `maxN` keeps track of the index of the current steepest. OpenMP `parallel for prgma` that executes each iteration of the loop in parallel is used for parallelisation of the procedure that finds the receiver node N_{R_i} for each node N_i.

Listing 5.10: The C++ / OpenMP implementation of the stream flows computation.

```
#pragma omp parallel for
for(int i = 0; i < SIZE; ++i){
  float h = height[i];
  int offs = offset[i];
  int nN = nNum[i];

  int maxN = NO_FLOW;
  float maxS = 0.0f;

  for(int j = 0; j < nN; ++j){
    int n = neighbour[offs + j];
    if(n != -1){
      nh = height[n];
      if(nh < h){
        float slope = std::abs(h - nh) / distance[offs + j];
        if(maxS < slope){
          maxS = slope;
          maxN = n;
        }
      }
    }
  }
  receiver[i] = maxN;
```

```
}
```

The GPU implementation follows the same general algorithm parallelised by launching one thread per node. Each thread will search through the neighbour of N_i to find the neighbour with the steepest descent. This kernel is shown in Listing 5.11.

Listing 5.11: The CUDA implementation of the stream-flow computation.

```
__global__ void CR_kernel(float* distance, float* height, unsigned int* offset, unsigned int*
    nNum, int* neighbour, float* distance, int* receiver, const size_t SIZE){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= SIZE) return;
  float thread_h = height[idx];
  unsigned int thread_offset = offset[idx];
  unsigned int thread_nNum = nNum[idx];

  int maxN = NO_FLOW;
  float maxS = 0.0f;

  for(int i = 0; i < thread_nNum; ++i){
    int n_idx = neighbour[thread_offset + i];
    if(n_idx != -1){
      float n_h = height[n_idx];
      if(n_h < thread_h){
        float n_d = distance[thread_offset + i];
        float slope = abs(thread_h - n_h) / n_d;
        if(maxS < slope){
          maxS = slope;
          maxN = n_idx;
        }
      }
    }
  }
  receiver[idx] = maxN;
}
```

The algorithm to compute the later stages of the pipeline also requires the list of donors (denoted as D_{N_i}) for each node N_i. To store the donor list for each node N_i, N_i is added into the donor list $D_{N_{R_i}}$ of the receiver node N_{R_i}. The total number of donors for each node N_i must be counted each iteration because the total number of donors will change through the simulation. The total number of donors for each node can not exceed the total number of neighbours because only neighbours of the node can become a donor of the node.

The procedure to add N_i into the $D_{N_{R_i}}$ will have a race condition when nodes that share the same N_{R_i} are trying to read and increment the donor count for N_{R_i} at the same time. The parallel algorithm proposed in the previous work creates a thread the given each node N_i to add itself into the $D_{N_{R_i}}$ with protection for the race condition of the donor count increment. For the implementation of the algorithm, The lists of donors for each node are stored in `donor`, and each list can be accessed using an offset index into the list stored in `offset`. For a donor node to add itself to the donor list, an index of the receiver node is stored in `receiver`, and as a donor gets added, the donor counts for the node stored in `dNum` need to be incremented.

Listing 5.12 presents C++ / OpenMP implementation of the parallel algorithm to store each node N_i into the donor list of its receiver node $D_{N_{R_i}}$. `dNum` must be initialise to 0 because the total number of donors for each node N_i will change throughout the simulation and is used to add elements to index and also counts donors. `parallel for pragma` is used to parallelised the

procedure, and each node is given to each thread. `atomic capture pragma` is used to protect the section with the race condition that is between nodes that share the same N_{R_i}.

Listing 5.12: The C++ / OpenMP implementation of the list of donors computation.

```
#pragma omp parallel for
for(int i = 0; i < SIZE; ++i) dNum[i] = 0;

#pragma omp parallel for
for(int i = 0; i < SIZE; ++i){
  int r = receiver[i];

  if(receiver[i] != NO_FLOW){
    #pragma omp atomic capture
    unsigned int dN = dNum[r]++;

    donor[offset[r] + dN] = i;
  }
}
```

The GPU implementation follows the same general algorithm with a separate thread to add N_i into the $D_{N_{R_i}}$. The race condition can be protected using the atomic function `atomicInc`. This kernel is shown in Listing 5.13.

Listing 5.13: The CUDA implementation of the list of donors computation.

```
__global__ void CD_kernel(int* receiver,unsinged int* offset,unsigned int* dNum,int* donor,
    const int SIZE){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= SIZE) return;

  int r = receiver[idx];

  if(r == NO_FLOW) return;

  unsigned int r_offset = offset[r];
  unsigned int r_dNum = atomicInc(dNum + r, SIZE);

  donor[r_offset + r_dNum] = idx;
}
```

5.3.2 A New GPU Algorithm

In order to find the receiver node N_{R_i}, the previous method parallelises the procedure by creating one thread per node N_i. This approach may underutilise the GPU because it ignores a further parallelisation opportunity. In the previous work, the thread is created per node N_i to search through its neighbours sequentially. However, there is no dependency between the procedures of searching through each neighbour except for the comparing and updating the current steepest neighbour. The new algorithm further parallelises the procedure by creating one thread per edge to compute slope angle between each node N_i and a single neighbour. This approach better utilises the parallel processing capability of the GPU, which would show better computation throughput. However, it does require additional storage for each neighbour in the list of neighbours for each node N_i to identify which node they belong to.

The new algorithm consists of two steps where each step is implemented in the separate CUDA kernel. The first step creates a thread given each neighbour in the neighbour lists to compute

the slope angle between N_i and a single neighbour to find the steepest. For a thread to access data of N_i that the given neighbour belongs to, the procedure requires additional data that is the same size array as `neighbour`, which stores the index of N_i at the corresponding position of each neighbour stored at the `neighbour`. This array is called the `mask`, and Figure 5.4 shows how data is stored. The steepest neighbour then can be found by comparing the slope angle computed for all neighbours of each node N_i, which needs to be performed sequentially and it can be done using an atomic operation. The result computed for each node N_i needs to be accessed in the next step, which also requires another additional data storage.

nNum:	3	4	4	3	4												
Offset:	0	3	7	11	14												
Distance:	0.17	0.25	1.32	1.42	0.13	0.27	0.31	0.99	0.87	3.21	4.11	2.13	9.7	1.3	14.2	1.2	2.1	1.3
Mask:	0	0	0	1	1	1	1	2	2	2	2	3	3	3	4	4	4	4

Figure 5.4: The visual example of how data stored in the *distance* and *mask* arrays in the small system of 5 nodes.

For the implementation of the first step of the new algorithm, the kernel creates one thread per edge stored in `neighbour`, and the node that is a given neighbour belongs to can be identified from the corresponding element stored in `mask`. Some neighbours may be stored as -1, indicating an empty space, and should be ignored. In order to compute the slope of the given edge, the distance between them is stored in `distance`, and their elevation is stored in `height`. The current steepest is stored in the `maxS` by comparing and updating it using the atomic operation. The kernel implementation of the step is presented in Listing 5.14. Note that it rounds the value of slope angle computed to the integer with 10^6 for the comparison because the atomic operation CUDA does not support floating-point value. The steepest slope angle computed and the index of the receiver node N_{R_i} found are stored in the upper and lower 32-bit of the `maxS` for each node N_i.

Listing 5.14: The CUDA implementation of the massively parallel computation of the slope and the receiver node.

```
__global__ void computeSlope_kerenl(float* height, int* neighbour, unsigned int* mask,unsigned
     long long int* maxS, float* distance){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= D_SIZE_N) return;

  int n_idx = neighbour[idx];
  if(n_idx == -1)return;

  unsigned long long int c_idx = mask[idx];

  float n_h = height[n_idx];
  float c_h = height[c_idx];
  float dist = distance[idx];

  unsigned long long int slope = (c_h > n_h ? (abs(c_h - n_h) / dist) * 1e+6f : 0);

  if(slope != 0) atomicMax(maxS + c_idx, (slope << 32) | n_idx);
```

```
}
```

This stage of the pipeline in the previous work requires to store the list of donors for each node N_i for the later stage of the pipeline. The new algorithm proposed does not require to store the list of donors but the total number of donors for the later stage of the pipeline. The final step of the stage stores N_{R_i} and counts the total number of donors for each node N_i. The index of N_{R_i} can be extracted from the result from the previous step stored in $maxS$. The total number of donors for each node N_i can be counted by N_i increment the donor count of N_{R_i}. A thread can be created for each node N_i to store the receiver node N_{R_i} and increment the donor count of N_{R_i}. There may be a race condition between nodes that shares the same receiver node, which can be prevented using an atomic operation.

For the implementation of the final step of the new algorithm, the computation results from the previous step stored in `maxS` can be extracted and stored can be stored in `receiver`. The slope computed stored in the upper 32-bit, and if it is 0, it indicates that the node does not have a receiver node. Otherwise, the index of the receiver is extracted from lower 32-bit and stored in `receiver`. Lastly, the corresponding donor count stored in `dNum` of the found receiver node can be increment by performing an atomic operation. The implementation of the kernel to execute the procedure is presented in Listing 5.15.

Listing 5.15: The CUDA implementation of the final step to compute stream-flows

```
__global__ void computeReceiver_kernel_level_labeling(unsigned long long int* maxS, int*
    receiver, unsigned int* dNum){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= D_SIZE) return;
  unsigned long long int maxN = maxS[idx];
  int r_idx  = ((maxN & 0xFFFFFFFF00000000) == 0) ?  NO_FLOW : (maxN & 0x00000000FFFFFFFF);
  if(r_idx != NO_FLOW) atomicInc(dNum + r_idx, D_SIZE);
  receiver[idx] = r_idx;
}
```

Although the new algorithm costs additional memory space, the new algorithm proposed in this study to compute this stage of the pipeline better utilise the GPU over the previous method by further parallelising the execution.

5.4 Ordering

Solving the stream power equation is an iterative process based on the order of dependency hierarchy between each node $N_i \in G$ defined by a set of directed trees T computed in the previous step. In order to solve the stream power equation efficiently, such order of computation may be extracted by performing a traversal of T. In this section, we present the parallel algorithms and their implementation of the node ordering process.

5.4.1 RB Algorithm

The parallel algorithm proposed in [2] is building an ordering queue to store computation order by traversing the tree. Traversing each tree in depth-first search (DFS) order was initially suggested in [21]. This approach explores each tree branch by branch and generates a tree by tree computation order queue for nodes $\in G$, which is based on the idea that there is no dependency between nodes that are in different trees. Therefore, each tree can be allocated onto separate parallel threads to compute the later stages of the pipeline. However, [21] does not consider that not all computation of nodes in the same tree is dependent on each other. For nodes in the same level (depth), their computation to solve the stream power equation is also independent of each other, even though they are in the same tree. In order to increase the parallelism, breadth-first search (BFS) order traversal that can build a queue in a level by level order instead of a tree by tree order was suggested by the previous work in [2]. A visual example of the BFS order queue is presented in Figure 5.5. In order to build the ordering queue, the algorithm traverses the set of stream-flow trees T in BFS order, and stores the node into the queue. The algorithm also tracks an offset index into levels in the order queue during traversal for the later stages of the pipeline.

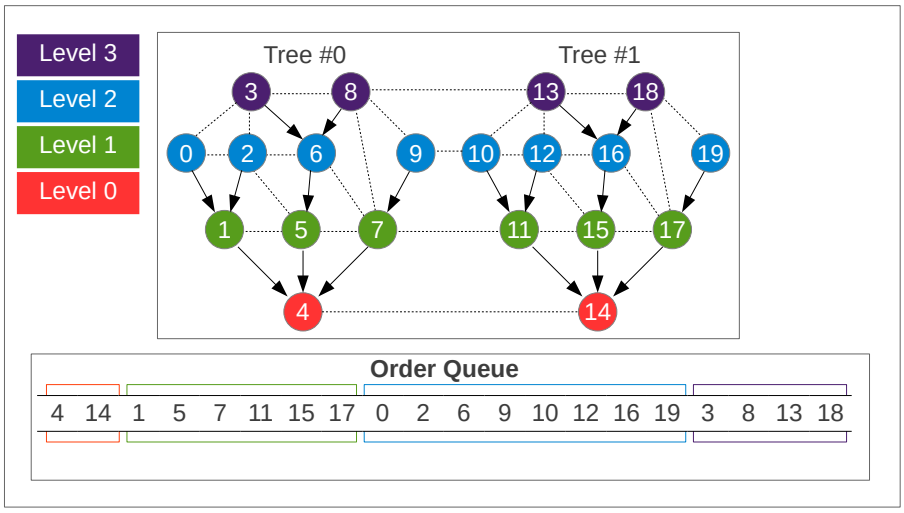


Figure 5.5: An example of the small system of 20 nodes covered by two trees to show how computation order will be in the queue in the BFS order where circles represent a node, dashed lines represent an edge of the graph arrow lines represent a directed edge of the tree. Nodes that are on the same level can split onto different parallel threads.

Listing 5.16 presents our CPU implementation of BFS order queue generation algorithm used in [2]. The algorithm first initialises variables where n is a counter for the *order* queue array, *levels* is an array to store an index offset of each level in the queue, nL is counter for the *levels* array. In the first *for* loop, traversal of T begins with finding all root nodes of each tree $\in T$ and insert those root nodes into the queue. The while loop then expands the search a level by level. The nested loop in the while loop; the outer loop iterates through all nodes at the current deepest level, and

inner loop iterates through all donor of those nodes and insert them into the queue. It then stores a new index offset in the *levels* array once all nodes at the next level are inserted in the queue. The while loop repeats until the whole T traversed.

The algorithm is implemented as a sequential program. The parallel approach of the algorithm in [2] is to parallelise the first for loop and the outer loop of the second for loop and to have a queue insertion and queue counter increment operation as a critical section. For the CPU application, the multi-core implementation of this approach shows slower computational performance than a sequential program.

The CPU implementation is written in a sequential program because the parallel computation of the queue generation showed worth performance than the sequential implementation on the CPUs. For the implementation, the extracted order is stored in `order`, and an index offset into each level also need to stored in `levels`. Nodes at the next level are the donors of the nodes at the current level. In order to traverse to the next level, the donors stored in `donor` need to be accessed using the index offset into the list stored in `offset` and donor count stored in `dNum`. Listing 5.16 presents the CPU implementation of the queue generation procedure. The variable `n` and `nL` are counters used to track of the current last index of queue and current last level. `levels` must be initialised to 0 because it will change throughout the simulation. The procedure first finds all root nodes of trees and then starts traversing by expanding the donor lists of nodes at the current level. During the traversing, nodes are added to the queue one by one, and the level offset when the traversing of the current level finishes.

Listing 5.16: The C++ implementation to generate a computation order queue.

```
int n = 0;
std::memset(levels.data(), 0, sizeof(int) * SIZE_LEVEL);
levels[0] = 0;
nL = 1;

for(int i = 0; i < SIZE; ++i) if(receiver[i] == NO_FLOW) order[n++] = i;

levels[nL++] = n;
int bL = -1;
int tL = 0;

while(bL < tL){
  bL = tL;
  tL = n;
  for(int sI = bL; sI < tL; ++sI){
    int idx = order[sI];
    int _offset = offset[idx];
    int _dNum = dNum[idx];
    for(int i = 0; i < _dNum; ++i){
      order[n++] = donor[_offset + i];
    }
  }
  levels[nL++] = n;
}
nL--;
```

Unlike the CPU implementation, it is inefficient to launch a single thread for GPU implementation to achieve reasonable performance. It is also inefficient to copy a chunk of data back into *host* to generate the queue on the *host* side. Therefore, the CUDA implementation of the procedure

explores parallel execution suggested in [2]. The parallel implementation of the procedure consists of two steps which are implemented as a separate kernel. The steps are as follow; expands the search to add nodes at the next level to the queue in parallel, launch a single thread to label the offset for the next level. In order to iteratively expand the search, the *host* side application needs to keep track of the current last index of the queue, which has to be copied from the *device* side.

Listing 5.17 presents the *host* side implementation of the procedure. The root nodes of the trees are added to the queue first by launching the kernel `addRoot`, and the level offset is stored by launching `setNewBoundary`. It then iteratively expands the search a level by level. Each iteration a new level offset is stored by launching `setNewBoundary`. Note that in order to launch kernel iteratively, the current last index of the queue n need to be copied back to *host*.

Listing 5.17: The *host* side implementation of the queue generation algorithm.

```
nL = 1;

cudaMemset(n, 0, sizeof(unsigned int));
cudaMemset(levels, 0, sizeof(unsigned int) * SIZE_LEVEL);
cudaMemcpy(nL, &nL, sizeof(unsigned int), cudaMemcpyHostToDevice);

addRoot<<<BLOCK, THREAD>>>(receiver, n, d_order,SIZE); cudaDeviceSynchronize();
setNewBoundary<<<1, 1>>>(levels,d_n,d_nL); cudaDeviceSynchronize();

unsigned int h_n;
int bL = -1;
int tL = 0;

while(bL < tL){
  cudaMemcpy(&h_n, n, sizeof(unsigned int), cudaMemcpyDeviceToHost);

  bL = tL;
  tL = n;
  const unsigned int LS = tL - bL;
  const unsigned int BLOCK = LS <= THREAD ? 1 : std::ceil(LS / float(THREAD));

  addNextLevel<<<BLOCK, THREAD>>>(dNum,offset,n,donor,order,bL,tL); cudaDeviceSynchronize();
  setNewBoundary<<<1, 1>>>(levels, d, d); cudaDeviceSynchronize();
}
cudaMemcpy(&nL, d, sizeof(unsigned int), cudaMemcpyDeviceToHost);
cudaMemcpy(levels.data(), d_levels, sizeof(unsigned int) * SIZE_LEVEL, cudaMemcpyDeviceToHost);
nL--;
```

Listing 5.18 presents the the CUDA kernel implementation of `addRoot`. The kernel creates one thread per node, which adds the nodes that are root into the queue. Reading and incrementing `n` is a race condition between threads that found root nodes, which is prevented by an atomic operation.

Listing 5.18: The CUDA kernel implementation to find and add all root nodes to the queue.

```
__global__ void addRoot(int* receiver, unsigned int* n, unsigned int* order, const unsigned int
    SIZE){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= SIZE) return;

  int r = receiver[idx];
  if(r != NO_FLOW) return;

  unsigned int my_idx = atomicInc(n, SIZE);
  order[my_idx] = idx;
}
```

Listing 5.19 presents the implementation of the CUDA kernel `setNewBoundary` that is to insert

a new index offset of the deepest level inserted so far. This kernel only creates a single thread for the insertion. Purpose of the kernel is to avoid data transfer between the *host* and the *device*.

Listing 5.19: The CUDA kernel implementation to insert a new index offset into the *levels* array.

```
__global__ void setNewBoundary(unsigned int* levels, unsigned int* n, unsigned int* nL){
  levels[*nL] = *n;
  (*nL) += 1;
}
```

Listing 5.20 presents implementation of the CUDA kernel `addNextLevel` that is to insert nodes that at the next level into the queue. Ther kernel creates one thread per node at the current deepest level. Insertion to the queue is again a race condition. Each thread loads the count of donors of the given node, and it sums the loaded count to the counter of the queue `n` by performing an atomic add operation. `atomicAdd` function returns an old value of the `n` and adds up the number of donors to the `n` can secure the spot at $[n, n + dNum - 1]$ of the queue to insert all donors of the given node without a race condition. As a result, this approach increases the parallelism of the execution.

Listing 5.20: The CUDA kernel implementation to add all nodes at the next level into the queue.

```
__global__ void addNextLevel(unsigned int* dNum, unsigned int* offset, unsigned int* n, int*
    donor, unsigned int* order, const int bL, const int tL){
  int idx = blockIdx.x * blockDim.x + threadIdx.x + bL;
  if(idx >= tL) return;

  unsigned int thread_idx = order[idx];
  unsigned int thread_dNum = dNum[thread_idx];

  if(thread_dNum == 0) return;

  unsigned int thread_offset = offset[thread_idx];
  unsigned int thread_n = atomicAdd(n, thread_dNum);
  for(int i = 0; i < thread_dNum; ++i) order[thread_n++] = donor[thread_offset + i];
}
```

5.4.2 A New GPU Algorithm

Although the tree traversal has an efficient linear time complexity to the size of the graph, it is challenging to implement queue generation algorithm efficiently for GPU because the algorithm only focuses on parallelism, not the architecture of targeting hardware. Although, a single kernel invocation only takes fractions of a second, launching kernel hundreds and thousands of times at the stages of the pipeline for every iteration of the simulation does affect performance and is a costly approach. Especially implementation of this stage of the pipeline also requires data transfer between the *host* and *device* for every iterative kernel launching, which should be avoided for the optimal performance for the GPU application. This study proposes a new algorithm to perform the ordering, instead of building a queue, it sorts the nodes by their level.

To sort nodes by their level, the procedure as following; label the level of all nodes, sort the nodes using the level labelled as a key and rearrange data. Each step is implemented in separate

CUDA kernels.

The first step is to label a level for each node. This step is implemented by making all nodes traverse the directed edge constructed by the stream-flows in parallel to find their level on the tree. The procedure first finds the root nodes of the trees and label their level as 0. It then launches a kernel that creates a thread that is given each node and traverses a directed edge constructed by the stream-flows until it finds the node that has been labelled. Once it finds the labelled node, the level can be computed by adding traverse step and the level of the found node. For each node to traverse toward the root of the tree, it needs to access the receiver node, which is stored in `receiver`. The levels labelled for each node is used for a sort key, which can be stored in `level`.

Listing 5.21 presents the implementation of the CUDA kernels to label level. The kernel function `initialiseLevel` label the root nodes to begin the procedure. The kernel function `labelLevel` launch threads per node to perform parallel traversing. Each thread traverse the tree towards the root of tree through the receiver node until it finds the one that has been labelled.

All nodes in the graph can then be sorted using the `level` as a key. Construction of a value array of the size N to be paired is also required. The array is filled with indexes $[0, N-1]$, which is named to `sort_id`. The array is later used to construct a `renumber` array that is used to rearrange the data in sorted order. Sorting operation is performed by parallel radix sort from CUB library. In order to achieve the optimal performance, the maximum number of bits required for the sort key has to be known. The maximum value in the `level` can be efficiently found on the GPU using the function called `DeviceReduce::Max` from CUB library.

Listing 5.21: The CUDA kernel implementation to label a level of each node.

```
__global__ void initialiseLevel(int* level, int* receiver){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= D_SIZE) return;
  level[idx] = receiver[idx] == NO_FLOW ? 0 : -1;
}
__global__ void labelLevel(int* level, int* receiver){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= D_SIZE) return;

  int r_idx = receiver[idx];
  if(r_idx == NO_FLOW) return;

  int r_level = level[r_idx];
  int cnt = 1;

  while(r_level == -1){
    cnt++;
    r_idx = receiver[r_idx];
    r_level = level[r_idx];
  }
  level[idx] = r_level + cnt;
}
```

Sorting operation arranges the indexes stored in `sort_id` in an order that is similar to the ordering queue. To rearrange the node data in this order `renumber` that indicates a new position of data need to be constructed. Listing 5.22 presents the CUDA kernel implementation to construct the array. Each thread is given each element of the `sort_id`. Let `i` be a thread index, the data

position at `sort_id[i]` moves to `i`. Therefore each thread stores `i` at `renumber[sort_id[i]]`.

Listing 5.22: The CUDA kernel implementation to construct the renumbering array.

```
__global__ void constructRenumberArray(unsigned int* sort_id, unsigned int* renumber){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= D_SIZE) return;
  renumber[sort_id[idx]] = idx; [4] = 0
}
```

Data can then be rearranged as the CUDA kernel presented in Listing 5.23. The implementation is abstracted to give an example of how a data array can be rearranged. In the actual implementation, all arrays of node data need to be rearranged.

Listing 5.23: The abstracted CUDA kernel implementation to rearrange data.

```
__global__ void rearrangeData(...){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= D_SIZE) return;
  int new_idx = renumber[idx];
  data_out[new_idx] = data_in[idx];
}
```

The index information of the graph representation data also needs to be updated into a new index. Listing 5.24 present the CUDA kernel implementation to rearrange the graph representation. For the graph representation, instead of reposition the data, it just updates the value of the index.

Listing 5.24: The abstracted CUDA kernel implementation to rearrange the list of neighbours.

```
__global__ void rearrangeNeighbourList_kernel_level_labeling(unsigned int* renumber, int*
    neighbour, unsigned int* d_neighbour_mask){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= D_SIZE_N) return;

  int old_idx = neighbour[idx];
  unsigned int old_mask = d_neighbour_mask[idx];

  if(old_idx != -1){
    int new_idx = renumber[old_idx];
    neighbour[idx] = new_idx;
  }

  d_neighbour_mask[idx] = renumber[old_mask];
}
```

5.5 Uplift Computation

Computation of the change in elevation caused by tectonic uplift is one of two parts of the stream power equation. Each node $N_i \in G$ is given a constant uplift rate u_i that defines the growing speed of the elevation per year at each local region. A new elevation of each node caused by tectonic uplift can be computed using the equation:

$$h_i(t + \Delta t) = h_i + u_i \Delta t \tag{5.1}$$

Where h_i denotes an elevation of $N_i \in G$, u_i denotes the constant uplift rate of $N_i \in G$, Δt denote

the length of timestep. Computation for each node are completely independent of each other and therefore the uplift rate for each node can be computed on n threads without any communication required between threads where n denotes the total number nodes.

This stage of the pipeline is the one example of embarrassingly parallel problems, which of each computation is obviously independent of each other. In multi-core CPU implementation, such parallel problems can be computed using SIMD instruction set for it to exploit vector processing capability of CPU efficiently. The function `computeUplift` in Listing 5.25 is the OpenMP implementation that uses the `omp simd` pragma for each iteration of the for loop to be executed in parallel by using SIMD instructions.

Although the multi-core CPUs can exploit SIMD instructions for fast parallel execution of the uplift computation, the vector processing capability of GPUs significantly higher than multi-core CPUs for this kind of problems. `computeUplift_CUDA` function presented in Listing 5.25 is the kernel that each thread is given each node to compute the uplift rate for each node in parallel, which may show faster execution time over multi-core CPUs as the size of G grows.

Listing 5.25: The multi-core implementation (`computeUplift`) and the CUDA kernel implementations (`computeUplift_CUDA`) of the uplift computation.

```
void computeUplift(){
  #pragma omp simd
  for(int i = 0; i < SIZE; ++i){
    height[i] += uplift[i] * DT;
  }
}

__global__ void computeUplift_CUDA(float* uplift, float* height){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx >= D_SIZE) return;
  height[idx] += uplift[idx] * DT;
}
```

5.6 Erosion Computation

The last stage of the pipeline is computing a new elevation changed by fluvial erosion over the discrete timestep for each node $N_i \in G$. In this section, we present parallel algorithms and their implementation to compute elevation changes by erosion.

The new elevation for each node can be computed using the following equation discussed in Chapter 3:

$$h_i(t + \Delta t) = h_i + \frac{\frac{KA_i^m}{\Delta d} h_{R_i}(t + \Delta t)\Delta t}{1 + \frac{KA_i^m}{\Delta d}\Delta t} \tag{5.2}$$

The computation must be in the order of T that is extracted from the ordering stage. It is because the equation for $h_{R_i}(t + \Delta t)$ must be computed first as shown in the Equation 5.2. The drainage area A_i for each node also has to be computed in order to compute Equation 5.2. A drainage area is the total surface area of regions that flow water into a region represented by each

node, and it is used to approximate the total water accumulation in each region. The drainage area A_i for each node can be computed using the following equation discussed in Chapter 3:

$$A_i = a_i + \sum_{d_n \in D_{N_i}} A_{d_n} \tag{5.3}$$

Equation 5.3 is similar to Equation 5.2 where it needs to be computed in the order of T because $A_{D_{N_i}}$ must all be computed first before A_i where D_{N_i} denotes all contributing donor nodes of N_i. Therefore, A_i for each node $N_i \in G$ can be computed in the order of T, the only difference is that it has to be in reverse order since donor nodes have to be computed before the receiver nodes.

Thermal erosion process for each node is computed to correct artefacts caused by unrealistic high peaks. Thermal erosion process corrects the elevation of each node by restricting the slope angle between a receiver node and donor node, and it can be computed using the following function discussed in Chapter 3:

$$h_i = \begin{cases} h_i & \text{if } S_i \leq S_{max} \\ \tan(s_{max})\Delta d + rH & \text{if } S_i > S_{max} \end{cases} \tag{5.4}$$

5.6.1 RB Algorithm

In order to compute the drainage area and erosion for each node in parallel, computation dependency between nodes only occurs for the case that the nodes are connected through the streamflows. The parallelism that the parallel algorithm proposed in [2] to compute this stage of the pipeline is to compute nodes in the same level of T in parallel. The algorithm creates threads per node in the same level iteratively a level by level in order of computation.

The first step of this stage of the pipeline is to compute drainage area A_i for each node N_i. Computation order of drainage area computation is from the top level to bottom (root) where each node N_i in the level adds the A_i to the drainage area of the receiver node $A_{N_{R_i}}$. For the implementation, the lists of donors for each node are stored in `donor`, which can be accessed using the index offset into the list and total number of donors stored in `offset` and `dNum`. The surface area and drainage area to compute drainage area for each node are stored in `area` and `drainageArea`. Listing 5.26 shows the C++ / OpenMP implementation for multi-core CPU. `drainageArea` is first initialised to 0 because the computation result from the previous iteration is stored. The implementation iteratively executes `omp parallel for pragma` for each level to compute drainage area for nodes in the same level in parallel. Note that the parallel threads are only created when there are at least 500 nodes in the level because it is not enough parallelism to outperform sequential execution.

Listing 5.26: The C++ / OpenMP implementation of the drainage area computation for multi-core CPU.

```
memcpy(drainageArea.data(), area.data(), sizeof(float) * SIZE);
for(int lI = nL - 3; lI >= 1; --lI){
  const unsigned int lS = levels[lI];
  const unsigned int lE = levels[lI + 1];
  const unsigned int lSize = lE - lS;
  #pragma omp parallel for default(none) shared(lI) if(lSize > 500)
  for(int sI = lS; sI < lE; ++sI){
    int idx = order[sI];
    unsigned int _dNum = dNum[idx];
    unsigned int _offset = offset[idx];
    for(int i = 0; i < _dNum; ++i){
      drainageArea[idx] += drainageArea[donor[_offset + i]];
    }
  }
}
```

The same parallel algorithm can be implemented in CUDA by substituting the parallel for loop with a CUDA kernel that creates one thread per node in the current level. The *host* side application iteratively launches kernel a level by level. Unlike the implementation for the CPUs, it launches the kernel for all levels even though the total number of nodes is not large enough. It would be more costly to transfer data to *host* to compute it sequentially, and computation throughput provided by a single CUDA thread is not great. As the size of G grows, the total number of nodes at each level of T would grow beyond the efficient parallel processing capability of multi-core CPUs, enabling GPUs to outperform in computation time over CPUs. Listing 5.27 shows the CUDA kernel implementation to compute drainage area for nodes in the current level.

Listing 5.27: The CUDA kernel implementation of the drainage area computation.

```
__global__ void CDA_kernel(unsigned int* order, unsigned int* dNum, unsigned int* offset, int*
    donor, float* drainageArea, const int lS, const int lE){
  int idx = blockIdx.x * blockDim.x + threadIdx.x + lS;
  if(idx >= lE) return;
  int thread_idx = order[idx];
  unsigned int thread_dNum = dNum[thread_idx];
  unsigned int thread_offset = offset[thread_idx];
  for(int i = 0; i < thread_dNum; ++i){
    drainageArea[thread_idx] += drainageArea[donor[thread_offset + i]];
  }
}
```

The second step is to compute a new elevation changed by erosion over the discrete timestep for each node $N_i \in G$. Computation order of erosion computation is from the bottom (root) level to the top level, which is the reverse order of drainage area computation order. It is because to compute a new elevation for each node N_i, it must have updated elevation of the receiver node N_{R_i}. For the implementation, in order to compute erosion for each node, an index of the receiver node is stored in `receiver`, the drainage area A_i stored in `drainageArea`, the elevation of the node and its receiver node stored in `height`, the distance between the node and its receiver node stored in `distance` are required along with some constant parameters for the simulation. Listing 5.28 shows the C++ / OpenMP implementation for multi-core CPU. The implementation iteratively executes `omp parallel for pragma` for each level to compute erosion for nodes in the same level

in parallel. Note that the parallel threads are only created when there are at least 500 nodes in the level same as the drainage area computation. It is for the same reason that when there is not enough parallelism to outperform sequential execution.

Listing 5.28: The C++ / OpenMP implementation of the erosion computation for multi-core CPUs.

```
void computeErosion(){
  for(int lI = 1;  lI < nL - 1; ++lI){
  const int lS = levels[lI];
  const int lE = levels[lI + 1];
  const size_t lSize = lE - lS;
    #pragma omp parallel for if(lSize > 500)
    for(int sI = lS; sI < lE; ++sI){
      int idx = order[sI];
      int rIdx = receiver[idx];
      float _distance = glm::distance(point[idx], point[rIdx]);
      float ka_DIV_distance = (KEQ * std::pow(drainageArea[idx], MEQ) * DT) / _distance; //
          reduce redundunt computation
      float rH = height[rIdx];
      float h = (height[idx] + rH * ka_DIV_distance) / (1 + ka_DIV_distance);
      GLdouble slope = std::atan(std::abs(h - rH) / _distance);

      if(slope > maxSlope[idx]){
        h = std::tan(maxSlope[idx]) * _distance + rH;
      }
      height[idx] = h;
    }
  }
}
```

The same parallel algorithm can be implemented in CUDA by substituting the parallel for loop with a CUDA kernel that creates one thread per node in the current level the same as the drainage area computation. The *host* side application iteratively launches kernel a level by level. Unlike the implementation for the CPUs, it also launches the kernel for all levels even though the total number of nodes is not large enough because it would be more costly to compute it sequentially. As the simulation size grows, the total number of nodes at each level of T would grow beyond the efficient parallel processing capability of multi-core CPUs, which may enable GPUs to outperform in computation speed over CPUs. Listing 5.29 shows the CUDA kernel implementation to compute erosion for nodes in the current level.

Listing 5.29: The CUDA implementation of the erosion computation.

```
__global__ void CER_kernel(unsigned int* order, int* receiver, glm::vec2* point, float* height,
     float* drainageArea, float* maxSlope, const int lS, const int lE){
  int idx = blockIdx.x * blockDim.x + threadIdx.x + lS;
  if(idx >= lE) return;
  int thread_idx = order[idx];
  int r_idx = receiver[thread_idx];
  glm::vec2 thread_p = point[thread_idx];
  float thread_h = height[thread_idx];
  glm::vec2 r_p = point[r_idx];
  float r_h = height[r_idx];
  float distance = glm::distance(thread_p, r_p);
  float ka_DIV_distance = (KEQ * pow(drainageArea[thread_idx], MEQ) * DT) / distance;
  float h = (thread_h + r_h * ka_DIV_distance) / (1 + ka_DIV_distance);
  float slope = atan((h - r_h) / distance);
  float thread_maxSlope = maxSlope[thread_idx];

  if(slope > thread_maxSlope){
    h = tan(thread_maxSlope) * distance + r_h;
  }
  height[thread_idx] = h;
}
```

5.6.2 A New GPU Algorithm

The parallel algorithm proposed in [2] fully utilises the parallel processing capability of modern CPUs. The same general parallel algorithm can scale its parallelism to utilise computing hardware of highly parallel structure such as GPU. However, the algorithm can be further improved to utilise the computing power of GPUs better. Firstly, the maximum level of the set of stream-flow trees T can grow over hundreds and thousands as the simulation size grows. Launching kernels that many times for every iteration of the simulation is a costly approach. Secondly, some levels near the top may not have enough nodes to benefit from GPUs' parallel processing capability. This study proposes a new algorithm to compute this stage of the pipeline. The new algorithm launches kernel only once to compute drainage area and erosion for all nodes $N_i \in G$.

In order to transform the iterative kernel launching into a single kernel, the kernel creates a single thread per node, and each thread waits for dependent computation for the given node to complete. CUDA built-in warp vote function `__all_sync()` can be exploited to implement such behaviour inside the kernel. This function receives two arguments; the first argument is a mask to indicate which threads in the warp to be included in the vote, and the second argument is a condition. The function returns true if all threads in the vote satisfy the condition. The CUDA kernel implementations of this stage of the pipeline have a `while` loop statement block that iterates until `__all_sync()` function returns true where the condition is whether the thread has completed computation for the given node. Lastly, this is not a usual approach to write a CUDA kernel, and it is not recommended to make a thread wait in the loop until something completes. When CUDA kernel launches, threads are getting launched sequentially from the first to the end. For this approach to work appropriately, all nodes must be given to threads in computation order, which can minimise the waiting time of thread by the time they launch otherwise the kernel crashes or completes after a significantly long time.

There are two specific implemented features for Turing architecture compatibility. Firstly the volatile variables are for the variables that must be immediately reloaded once value changes to avoid unexpected caching behaviour showed at Turing architecture GPUs. Those GPUs loads values from the cache memory even when the actual value stored at the global memory has changed. They reload data from memory after a long time, making threads wait too long for warp vote function to return true. Secondly, the *___syncwarp()* function right after the conditional statement block is to reconverge threads in the warp explicitly due to the independent thread scheduling feature newly introduced in Turing architecture.

Listing 5.30 shows the CUDA kernel implementation to compute the drainage area A_i for each node N_i. The kernel arranges nodes to threads in reverse order of stored data because the order of computation is from top to bottom. Each thread waits until the `dNum` of the given node become

0, which indicates that the drainage area computation for the node has completed. Each thread then adds the A_i to A_{R_i}, and it then decreases the `dNum` of its receiver node N_{R_i}. These two computations in the block must be protected by atomic operation because there is a race condition between threads share the same N_{R_i}.

Listing 5.30: The CUDA kernel implementation of the drainage area computation of our new algorithm.

```
__global__ void computeDrainageArea_kernel(float* drainageArea, volatile float*
     vol_drainageArea, int* receiver, int* level, unsigned int* dNum, volatile unsigned int*
     vol_dNum){
  int idx = (D_SIZE - blockIdx.x * blockDim.x) + threadIdx.x;
  if(idx >= D_SIZE) return;
  int thread_level = level[idx];

  if(thread_level < 2) return;

  unsigned int mask = __activemask();
  bool done = false;

  while(!__all_sync(mask, done)){
    if(!done && (vol_dNum[idx] == 0)){
      int r_idx = receiver[idx];
      float da = vol_drainageArea[idx];
      atomicAdd(drainageArea + r_idx, da);
      atomicDec(dNum + r_idx, D_SIZE);
      done = true;
    }
    __syncwarp(mask);
  }
}
```

Listing 5.31 shows the CUDA kernel implementation to compute the erosion for each node N_i. The kernel arranges nodes to thread in order of stored data because the order of computation is from bottom to top. Each thread waits until the `erodded` of the receiver node N_{R_i} of the given node, which indicates that the erosion computation for N_{R_i} has completed. Each thread then computes the new elevation changes caused by erosion for the given node.

Listing 5.31: The CUDA kernel implementation of the erosion computation of our new algorithm.

```
__global__ void computeErosion_kernel(glm::vec2* point, float* height, volatile float*
     vol_height, float* drainageArea, float* maxSlope, int* receiver, int* eroded, volatile int*
      vol_eroded, const int N, float* distance){
  int idx = blockIdx.x * blockDim.x + threadIdx.x ;
  if(idx >= D_SIZE) return;
  int r_idx = receiver[idx];
  unsigned int mask = __activemask();
  bool done = false;
  if(r_idx == NO_FLOW) {
    eroded[idx] = N;
    done = true;
  }
  __syncwarp(mask);
  while(!__all_sync(mask, done)){
    if(!done && (vol_eroded[r_idx] == N)){
      glm::vec2 thread_p = point[idx];
      float thread_h = height[idx];
      glm::vec2 r_p = point[r_idx];
      float r_h = vol_height[r_idx];

      float dist = glm::distance(thread_p, r_p);
      float ka = KEQ * pow(drainageArea[idx], MEQ) * DT / dist;
      thread_h = (thread_h + r_h * ka) / (1 + ka);
      float slope     = atan(abs(thread_h - r_h) / dist);
      float maxS    = maxSlope[idx];
      height[idx] = slope > maxS ? (tan(maxS) * dist + r_h) : thread_h;
      __threadfence();
      eroded[idx] = N;
      done = true;
```

```
    }
    __syncwarp(mask);
  }
}
```

Extra Sort Key The new ordering method provides improved memory access over [2] by sorting data in order at the memory. However, while the new method can provide coalesced memory access patterns for threads given nodes at the same level, accessing data of nodes that are connected with given nodes through the stream flows or graph edges still not coalesced access. It is because sorting does not take account of such connections.

Memory access pattern can be improved by having indexes of N_{R_i} of all nodes $N_i \in G$ as a minor key for the sorting. Using the minor key sorts nodes into an order that provides coalesced memory access when each thread are accessing N_{R_i} or $N_{D_{N_i}}$. However, it requires the double amount of bits to sort nodes, making simulation slower overall because it takes longer at the ordering stage. We have tried to find another minor key that requires fewer bits and sort nodes into a similar order; such as sorting nodes with their position at the horizontal plane or sorting nodes in the same tree together. None has provided better memory access pattern. Computational performance of these approaches is also presented in the next chapter for comparison.

Size Limitation The new algorithm proposed minimises data transfer between *host* and *device* for optimal performance and therefore all simulation data are stored in the *device* side memory. Off-chip DRAM capacity is not very high compared to the size of RAM. This memory bottleneck limits the size simulation that the GPU can store. Multi-GPU computing could be one solution that extends data storage by adding more DRAM of the additional GPU to overcome the storage limitation. The Multi-GPU computing solution for the new algorithm is beyond the scope of this study, which can be a potential extension to this study.

5.7 Summary

Although the parallel algorithm proposed in [2] can efficiently exploit the parallel processing capability of modern hardware such as multi-core CPUs and GPUs, the algorithm can still be improved to utilise GPUs better because the algorithm does not specifically target GPUs. The new parallel GPU algorithm proposed by this study is optimised to improve GPU utilisation to compute the stream power equation simulation.

The stream flows computation stage has been improved by increasing the parallelism. The new ordering method by sorting node data removes the inefficient iterative kernel launching and data transfer between *host* and *device*. The drainage area and erosion computation of the previous algorithm also iteratively launch the kernel level by level, which becomes costly as the depth of the

stream-flows tree grows. The new algorithm transforms this iterative kernel launching algorithm into a single kernel launching by exploiting the warp voting function to make a thread wait for dependent computation to be completed. Performance results comparison between the previous work and the new algorithm are presented in the following chapter.

Chapter 6

Result

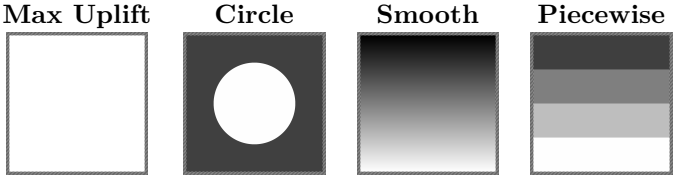
The different parallel implementations' performance results have been measured to compare their computation time for a range of initial uplift rates, sizes, and architecture. The different parallel algorithms for computing the stream power equation have been implemented using C++ / OpenMP and Nvidia's CUDA. In this chapter, the parallel algorithm proposes in the previous work [2] is referred to as `RB` and the new GPU algorithm is referred to as `New GPU`.

Terrain generation results of the new GPU algorithm have also been collected to verify that the new algorithm produces the same result as the original method proposed in [1]. The generate terrain is rendered in 3D using a C++ / OpenGL pipeline.

6.1 Performance Result

The machine used to test the CPU implementations is equipped with a 4.9 GHz Intel Core i7-9700k with 8 processing-cores and a 12MB cache and 32GB of DDR4-3200 RAM. Two GPUs are used to test CUDA implementations: an Nvidia GTX 780 (Kepler architecture, 3GB DRAM) and an Nvidia RTX 2070 Super (Turing architecture, 8GB DRAM). Although the GTX 780 is quite an outdated piece of hardware at the time of writing, it has been included in the test to observe how New GPU algorithm works on an old previous-generation GPU. The operating system used by this test machine is Ubuntu 18.04 LTS.

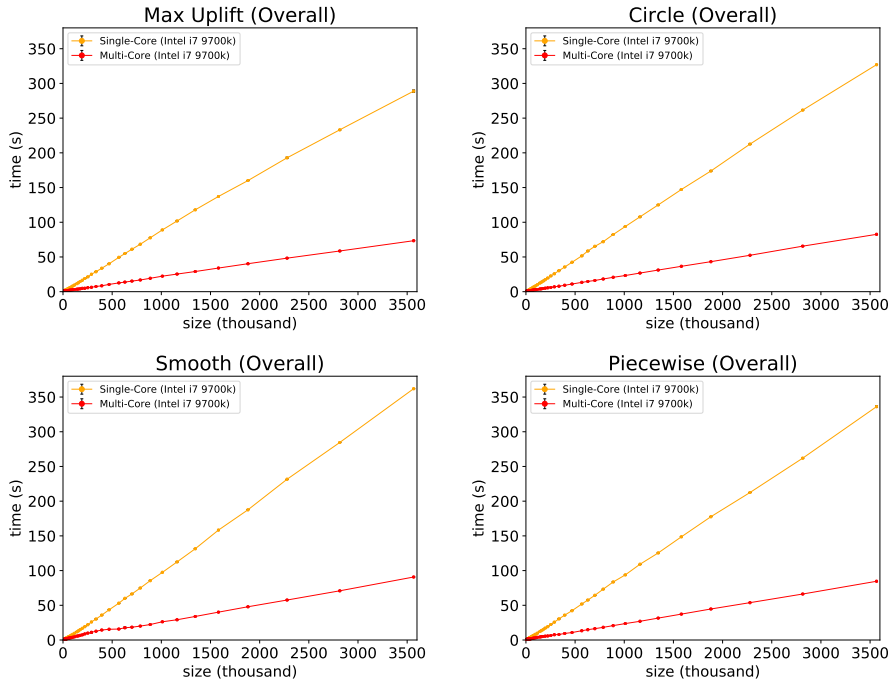
Figure 6.1: Uplift maps used in the tests.



The performance test measures the time duration to compute $1,000$ iterations for different

simulation sizes for different implementations. A randomly generated set of points with a minimum distance r ranges from 400 to 20 on the $5,0000$ X $5,0000$ size plain used in tests. The sizes range from $8,956$ up to $3,565,808$, and the same geometric graph (set of points) is used for the same size. The test for each size was conducted for 4 different uplift maps shown in Figure 6.1 to observe whether different uplift maps affect performance. Each size test was conducted 30 times to observe whether the results are consistent.

Figure 6.2: Overall performance comparison of the CPU implementations for four different uplift maps.

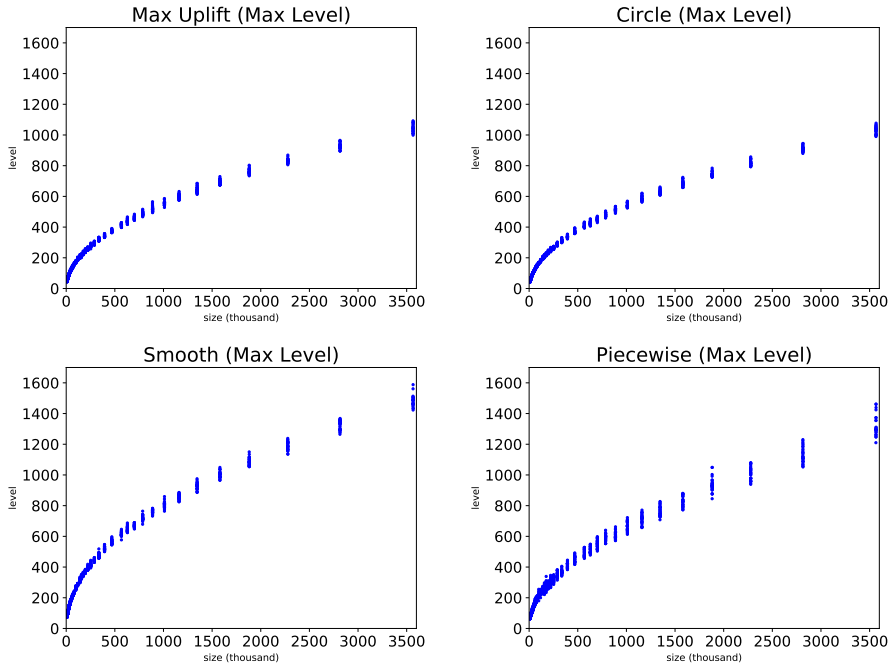


The first performance comparison is between the RB single-core and multi-core CPU algorithm implementations. Figure 6.2 shows that multi-core implementation consistently outperforms the single-core implementation across all sizes. The performance improvement of the multi-core over single-core is approximately 400% for all sizes and uplift maps. The simulation performance of CPU implementations linearly scales with the simulation size. However, results from different uplift maps show different computation times, which take longer in order of the max uplift, circle, piecewise, and smooth uplift maps, where the max uplift map is the fastest.

The inconsistent performance for the different uplift maps is believed to be due to the ordering, drainage area, and erosion stages and their relation to the maximum level (depth) of the set of stream-flows T. The ordering stage traverses each node in T a level by level to build the ordering queue. The drainage area computation stage iteratively processes T a level by level from bottom to top, and the erosion stage processes T the same way in reverse order. Figure 6.3 shows the overall maximum level of T throughout the simulation for the different uplift maps and simulation

sizes. The tests were conducted 30 times with different graph each time generated from the same minimum distance between points ranges from 400 to 20. While all uplift maps show a similar trend that the maximum level grows as the simulation size grows, the smooth uplift map's maximum level is significantly higher than others followed by the piecewise uplift map.

Figure 6.3: Maximum tree level thoughout the simulation.



In the stream power erosion model, the topology of T represents a running water path that flows from peak to river mouths (boundary) through the descendant pathâĂŤthe depth of T increases as the distance of water flowing path increases. While Max Uplift and Circle's peak would be located around the centre of the domain and flow water to the boundary from there, Smooth and Piecewise's peak would be located around the bottom of the domain, and water would flow towards to the top of the domain through the descendent path. Especially for Smooth, uplift rates gradually increase from the bottom to top of the domain, making nodes have more tight stream-flow connections throughout the domain.

Figure 6.4 shows a performance comparison between the RB multi-core and GPU. The RTX 2070 Super results show that the GPU provides better performance than multi-core as the simulation size grows over about 0.45 million. The GTX 780 results for the max uplift, circle, and piecewise uplift maps also shows better performance than multi-core CPU as the simulation size grows over about 2 million. However, the GTX 780 always shows slower performance than the multi-core CPU for the smooth uplift map. It is because the ordering, drainage area, and erosion stages launch kernel iteratively D times to compute the stage where D is the maximum level of the set of stream-flows T. Ordering stage especially also require the data transfer between *host*

and *device* every iteration of kernel launching.

Figure 6.4: Overall performance comparison of the multi-core CPU and RB GPU implementations for four different uplift maps.

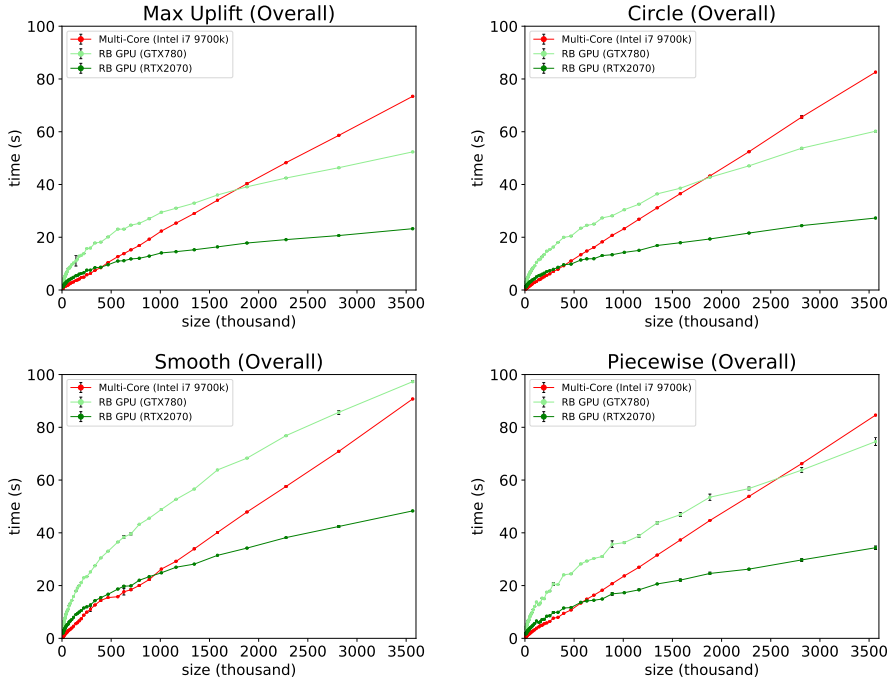
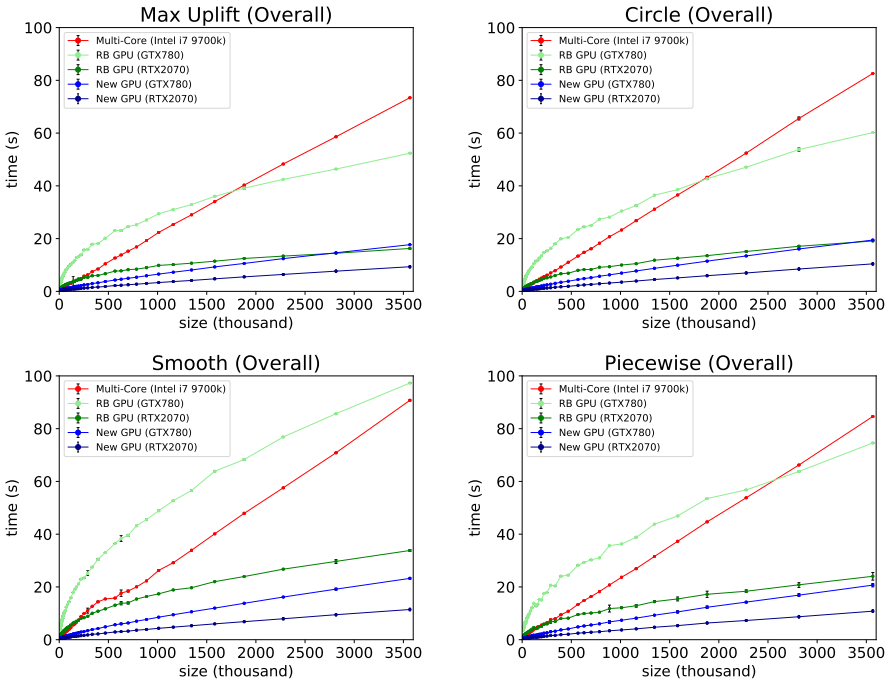


Figure 6.5 shows the performance comparison between the RB algorithm and the New GPU algorithm. The results for both GTX 780 and RTX 2070 Super shows significantly better performance than the RB multi-core CPU and at least 300% improved performance over the RB GPU. The New GPU algorithm's significant performance improvement can be found from the result for the GTX 780. According to the technical specification, while theoretical max computational throughput that the RTX 2070 Super provides is 8.92 TFLOPS, the GTX 780 only provides 3.97 TFLOPS computation throughput. However, the New GPU algorithm running on the GTX 780 outperforms the RB algorithm running on the RTX 2070 Super for most test cases. Performance improvement on the RTX 2070 super from the GTX 780 is also greater in the New GPU. While the RTX 2070 Super provides over the GTX 780 is about 200% improved performance with the RB algorithm, the New GPU algorithm runs with 300% improved performance on the RTX 2070 Super over the GTX 780.

Performance results of each stage of the pipeline have been collected for the case initialised with the smooth uplift map to observe how a higher maximum level of T affects each pipeline stage. Figure 6.6 shows the running time of each stage of the pipeline for the RB CPU and GPU, and New GPU algorithms.

The plots for single-core implementation shows that the stages that take the largest running time are the stream-flow, drainage area, and erosion stages. The stream-flow stage searches through

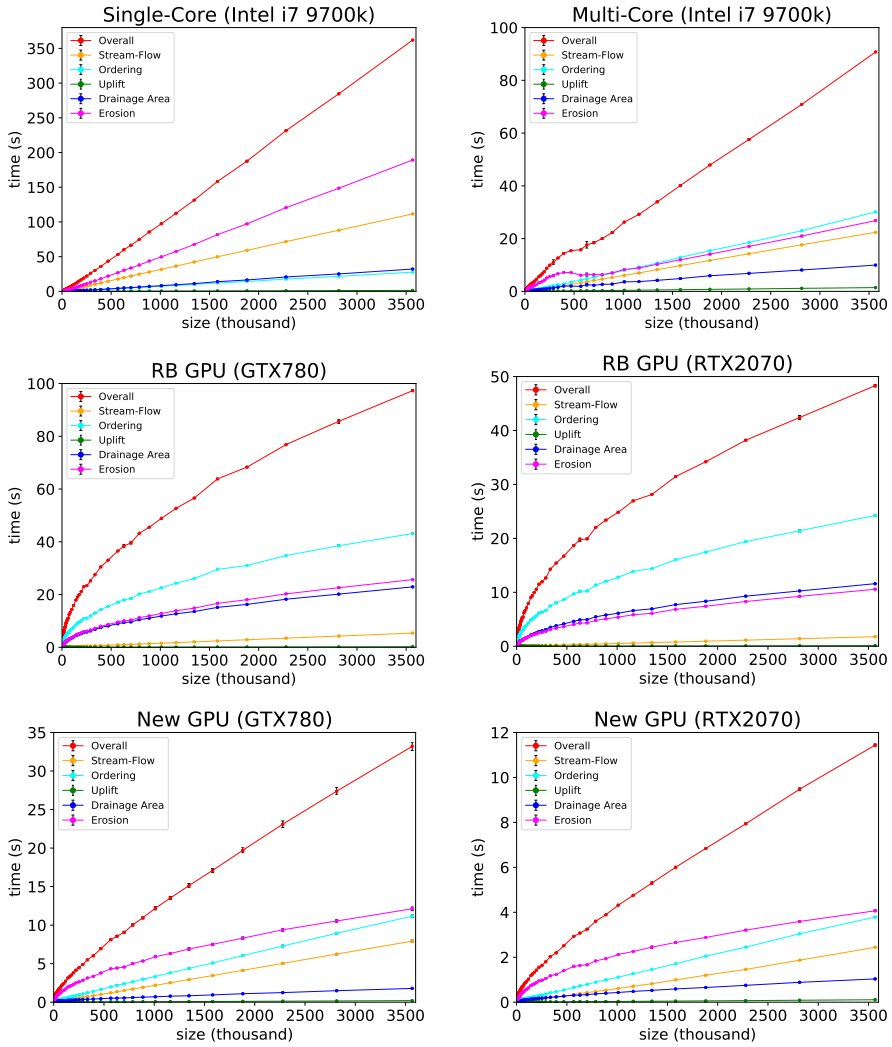
Figure 6.5: Overall performance comparison of the multi-core CPU, RB GPU, and New GPU implementations for four different uplift maps.



a list of neighbours for each node to find the receiver node. The drainage area stage iterates through each node in order of T and adds its drainage area to its receiver node's drainage area. The erosion stage iterates through each node in order of T and computes each node's erosion rate. The parallel multi-core implementation significantly improves these stages. The parallel implementation of the stream-flow stage creates a thread per node, and each thread search through a neighbour list to find the receiver node for the given node. The parallel implementation of the drainage area and erosion stage iteratively creates a thread per node at each level by level, and each thread computes the stage for the given nodes. However, parallelisation of the ordering stage for the multi-core CPU shows slower performance than the sequential implementation. Therefore the ordering stage is not parallelised for the multi-core implementation, making the stage take the largest running time.

The RB GPU plots show that the ordering stage takes the largest running time, followed by the drainage area and erosion stages. Those stages iteratively launch kernel a level by level, which can be costly if it requires hundreds and thousands of kernel launching. This costly approach has resulted in that for this particular case, the overall running time for the GTX 780 is slower than the multi-core CPU. The ordering stage also requires data transfer between *host* and *device* for every iteration of kernel launch. That is why the ordering stage takes significantly longer than other stages, making the overall performance. The stream-flow takes the shortest running time as it only launches a kernel just once for all nodes in the graph.

Figure 6.6: All stage performances comparison of the single-core, multi-core, RB GPU, and New GPU for the smooth uplift map.

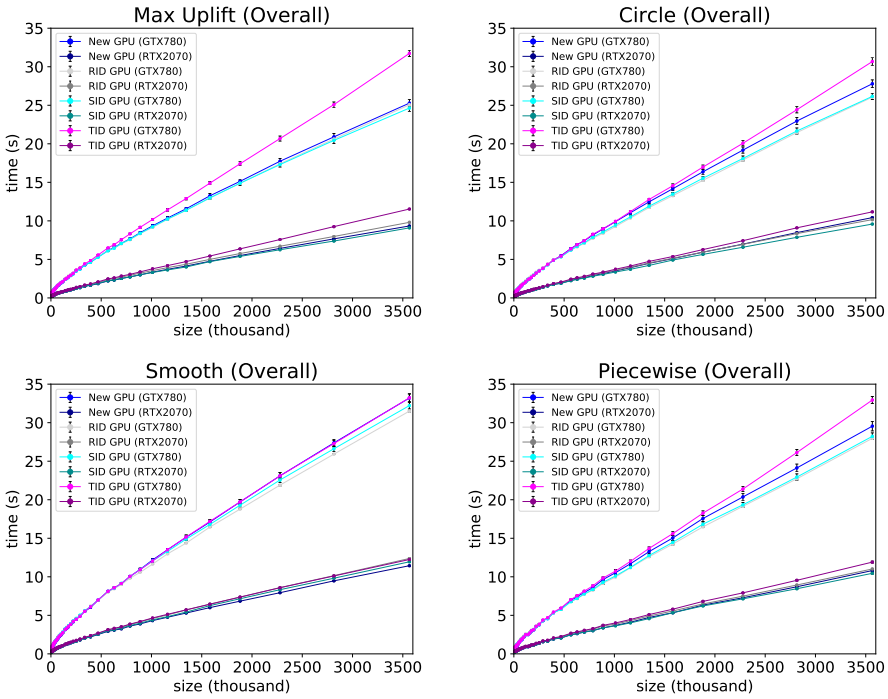


The New GPU plots show that the erosion stage takes the largest running time, followed by the ordering, stream-flow, and drainage area stages. The New GPU algorithm does not iteratively launch kernel to compute the ordering, drainage area, and erosion stages. The results for those stages shows the significantly faster running time than RB GPU. Only the stream-flow stage shows slightly slower performance than RB GPU because of the ordering stage that sorts nodes instead of building a queue affects the memory access pattern in the stream-flow stage. However, that does not affect the overall performance comparison because the ordering stage running of the RB GPU already takes longer than the overall running time of the New GPU.

6.1.1 Minor Sort Key

The potential for further improvement of memory access patterns of the pipeline computation by sorting nodes in the same level using the minor key has been discussed in Chapter 5. The New

Figure 6.7: Overall performance comparison of the New GPU and different minor key approaches for four different uplift maps.

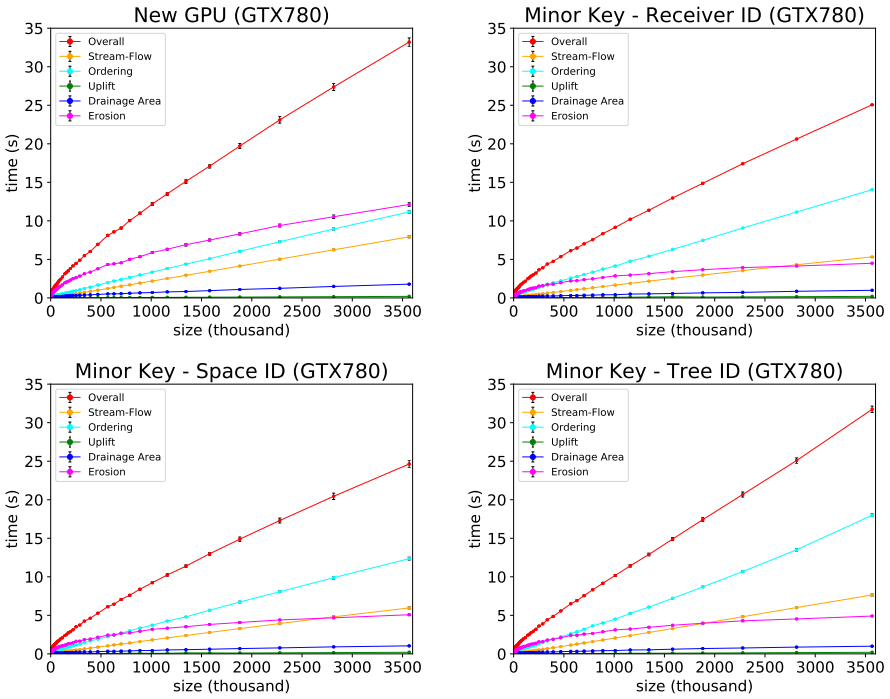


GPU sort nodes by the level can provide coalesced memory access pattern for loading data from nodes that are in the same level. However, the stages of the pipeline also need to load data of the given node's neighbour nodes, including the receiver node, which there will be no guarantee on coalesced access. The idea is that although adding a minor key to the soring would make the ordering stage slower if it can improve the memory access pattern for other stages of the pipeline, overall performance may be better than the original New GPU.

Performance results of different minor key approaches have been collected. Three approaches have been tried. The first approach uses an index of the receiver node for each node, which is referred to as `RID`. The second approach uses an index of the tree that each node belongs to, which is referred to as `TID`. The last approach uses an index of the space on the grid that each node belongs to, which is referred to as `SID`.

Figure 6.7 shows the performance comparison between the New GPU and different minor key approaches for different uplift maps. The results for the TID approach shows the worst performance of all. The poor performance could be that the TID approach either does not improve memory access pattern or takes too long at the ordering stage even though it improves the memory access pattern. The RID approach results only show slightly better performance for the case with the circle uplift map, which states that it may improve the memory access pattern, but the ordering takes too long. The SID approach results show slightly better performance for most cases except for the smooth uplift map. The results state that the SID approach's performance trade-off at the

Figure 6.8: All stage performances comparison of the New GPU and different minor key approaches running on the GTX 780 for the max uplift map.



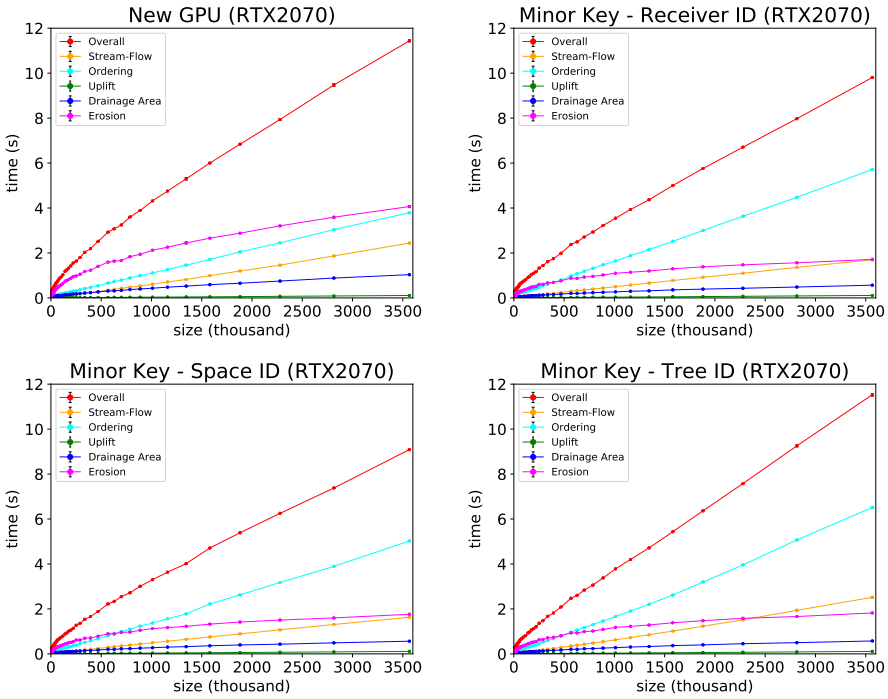
ordering stage is not as bad as long as if the maximum level of T is not too high.

Performance results of each stage of the pipeline have been collected for the case initialised with the max uplift. The ordering stage for the minor key approaches takes longer time than the original New GPU algorithm. The max uplift shows the smallest overall maximum level throughout the simulation, which will have less effect on the ordering stage's performance for each approach. Firue 6.8 and 6.9 shows the running time of each stage of the pipeline for the original New GPU algorithm and minor key approaches.

The RID approach results show that unlike the original New GPU algorithm, the ordering stage takes the largest running time. The stream-flow, drainage area, and erosion stages show improved performance over the original algorithm making RID approach outperform the original algorithm for this particular case. The RID approach results state that the memory access pattern for the pipeline stages is improved enough to show better performance over the original algorithm at the time-consuming pipeline.

The SID approach results also show improved performance at the stream-flow, drainage area, and erosion stages, but the improvement is not as good as the RID approach. The running time at the ordering stage is not as long as the RID approach. As a result, this approach outperforms the original algorithm for this particular case even though the memory access pattern's improvement is not as significant.

Figure 6.9: All stage performances comparison of the New GPU and different minor key approaches running on the RTX 2070 Super for the max uplift map.



The TID approach results show the worst performance of all. The results show that this approach does improve the memory access pattern as it shows shorter running times than the original algorithm for the stream-flow, drainage area, and erosion stages. However, although this approach improves the memory access pattern, it takes a long time at the ordering stage, making the overall performance slower than the original algorithm.

The study has not found a minor key that improves the memory access pattern as good as the RID approach and does not take a too long time at the ordering stage as the SID approach. If such the minor key can be found, the algorithm using that minor key may outperform the original algorithm for all cases.

6.2 Terrain Generation Result

The New GPU algorithm shows significantly improved performance over the RB algorithm and can correctly generate the terrain as the original method proposed in [1]. The result of the stream power equation simulation is a digital elevation map (DEM). Terrain result is rendered using C++ / OpenGL. The purpose of the rendering a result is to verify that the result produced by the New GPU algorithm is correct. Therefore terrain is rendered with a simple material Phong shading without any additional feature and texture.

Figure 6.10: Examples of rendered terrain generation result for different uplift maps.

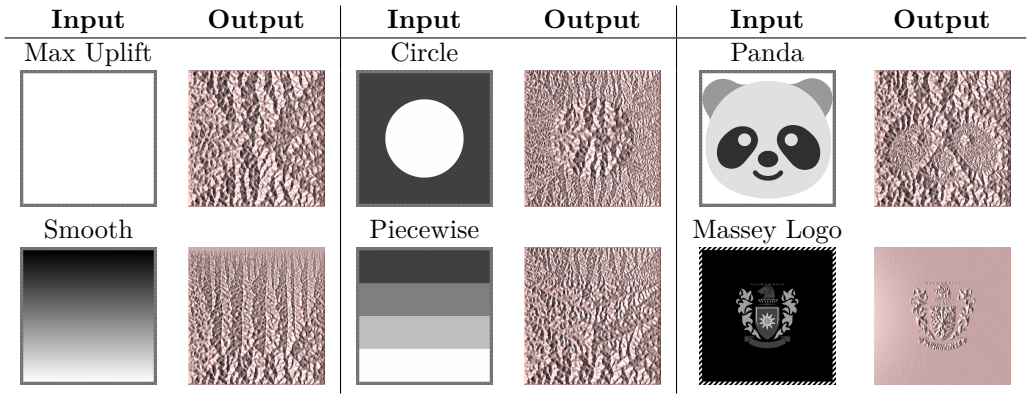


Figure 6.10 presents the rendering results of terrain generated from 6 different uplift maps. Those results are exactly the same as those generated by the simulation pipeline of the previously proposed method. For procedural terrain generation methods, it is important to be able to generate desired looking terrain. If the method only generates unpredictable random terrain and can not produce the desired output, it is not useful for many applications. In the results, controllability of the uplift map input to the output can also be observed. Rendered results with Circle, Panda, and Massey Logo uplift maps are formed into terrain that is almost identical to the shape of the uplift map. Smooth and Piecewise uplift maps distribute the uplift rates over the domain in a gradient pattern. The altitude distribution of terrain of mountain ranges follows the same gradient pattern.

6.3 Summary

The New GPU algorithm has been compared with the RB CPU and GPU algorithm. The New GPU algorithm shows significant performance over any RB algorithm implementation running on different hardware. Especially the New GPU algorithm even shows better performance on the GTX 780 over the RB GPU algorithm running on the RTX 2070 Super. It is a remarkable result considering while theoretical max computational throughput that the RTX 2070 Super provides is 8.92 TFLOPS, the GTX 780 only provides 3.97 TFLOPS computation throughput.

The approaches that use an extra minor key to sort nodes to improve the memory access pattern to neighbours of each node has been tried, and their performance results have been compared with the original New GPU algorithm. The trade-off between the running time of the ordering stage and improvement of the memory access pattern has to be optimal for these approaches to outperform the original algorithm. The optimal solution has not been found yet, and it requires further study.

Terrain generation results using the New GPU algorithm has been verified by rendering them in 3D. The New GPU algorithm successfully produces terrain as same as the original method in [1].

Controllability of the uplift map input to the output has also been presented. The controllability of procedural generation methods is important for them to be used in many applications.

Chapter 7

Conclusion

Virtual terrain is often an object of large spatial scale in backgrounds of the computer graphics scenes. This study developed a method to reproduce large scale terrain by investigating the different procedural terrain generation methods. Many methods have limitation generating terrain of such large scales. Fractal-based is lack of realism, and example-based methods are not reliable. It is because both of those methods do not consider land-forming processes. Hydraulic erosion simulation methods based on computational fluid dynamics are too expensive to compute despite several attempts to optimise the pipeline. The stream power erosion model can effectively model landscape evolution at large temporal and spatial scales. The original method has shown that employing simulation is effective for the purpose of generating large scale terrain. However, increasing simulation size may result in a long computation time using the pipeline of sequential implementation.

Graphics processing units (GPUs) were exploited to overcome the computational limitations of the original method of the stream power erosion model. Graphics processing units are very effective for massively parallel problems as compared to conventional CPUs. However, for GPUs to be effective on such problems, a parallel algorithm that specifically targets GPUs is required due to their different architecture. A previous study proposed a parallel algorithm to compute the stream power equation simulation. However, that proposed algorithm was a general parallel algorithm that targeted any multi-core hardware such as CPUs and GPUs. Therefore, there was an opportunity to improve the algorithm to utilise GPUs better.

This study proposes a new parallel GPU algorithm to compute the stream power erosion model equation. The new algorithm proposed several changes to each stage of the simulation pipeline to optimise its performance on GPUs. The stream-flow computations stage created a thread per graph edge instead of per node to find the receiver node for each node. The ordering stage sorted the nodes by level, rather than building an ordering queue. The drainage area and erosion computation

stages launched kernel only once instead of iteratively launching a kernel level by level.

The new parallel GPU algorithm's terrain generation was verified to generate the same terrain as the original stream power erosion model method. It also showed significantly better performance than the previously proposed algorithm across all tested simulation sizes, uplift maps, and hardware. The new algorithm consistently outperformed the multi-core CPU by about 300% on the GTX 780, and 900% on the RTX 2070 Super. It consistently showed about 300% improvement in performance over GPU implementation of the previous algorithm. In addition, the new algorithm running on the GTX 780, which is outdated hardware, outperformed the previous algorithm that ran on the significantly newer RTX 2070 Super. This result is very significant when considering that while theoretical max computational throughput that the RTX 2070 Super provides is 8.92 TFLOPS, the GTX 780 only provides 3.97 TFLOPS computation throughput.

The newly proposed GPU algorithm significantly outperforms the fastest parallel algorithm available in computing the stream power equation simulation, while being able to produce the same terrain result as the original method. This computational performance advancement allows the method to generate terrain with more precise geological detail while keeping the same computation time.

Future Work

Some possible extensions of this study have been identified. Firstly, the new algorithm stores all simulation data in DRAM to minimise the data transfer between *host* and *device*. However, simulation size is limited to the size of DRAM because simulation data is expected to be stored in the GPU. This memory bottleneck may be overcome by developing the multi-GPU computing solution for the new algorithm to extend data storage by adding more DRAM of the additional GPU to the computation.

Secondly, the ordering stage of the pipeline sorts nodes by the level, which provides coalesced memory access pattern for loading data from nodes in the same level. However, edges for each node will be stored in an arbitrary order, therefore accessing neighbours for each node has no guarantee for them to be coalesced. There have been several attempts to use a minor key to sort nodes within the same level align them in order of nodes that share the neighbours. Sorting by minor keys can improve memory access patterns for the stream-flow, drainage area, and erosion stages, but at an increased cost for the ordering stage. No one minor key shows a consistent performance improvement from this trade-off in all test cases but may warrant further investigation.

Bibliography

[1] G. Cordonnier, J. Braun, M.-P. Cani, B. Benes, E. Galin, A. Peytavie, and E. Guérin, "Large scale terrain generation from tectonic uplift and fluvial erosion," in *Computer Graphics Forum*, vol. 35, pp. 165–175, Wiley Online Library, 2016.

[2] R. Barnes, "Accelerating a fluvial incision and landscape evolution model with parallelism," *Geomorphology*, vol. 330, pp. 28–39, 2019.

[3] F. K. Musgrave, C. E. Kolb, and R. S. Mace, "The synthesis and rendering of eroded fractal terrains," in *ACM Siggraph Computer Graphics*, vol. 23, pp. 41–50, ACM, 1989.

[4] A. Fournier, D. Fussell, and L. Carpenter, "Computer rendering of stochastic models," *Communications of the ACM*, vol. 25, no. 6, pp. 371–384, 1982.

[5] G. S. Miller, "The definition and rendering of terrain maps," in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pp. 39–48, 1986.

[6] J. P. Lewis, "Generalized stochastic subdivision," *ACM Transactions on Graphics (TOG)*, vol. 6, no. 3, pp. 167–190, 1987.

[7] M. F. Barnsley, R. L. Devaney, B. B. Mandelbrot, H.-O. Peitgen, D. Saupe, R. F. Voss, Y. Fisher, and M. McGuire, *The science of fractal images*. Springer, 1988.

[8] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, J. C. Hart, and S. Worley, *Texturing & modeling: a procedural approach*. Morgan Kaufmann, 2003.

[9] J. Schneider, T. Boldte, and R. Westermann, "Real-time editing, synthesis, and rendering of infinite landscapes on gpus," in *Vision, modeling and visualization*, vol. 2006, pp. 145–152, Citeseer, 2006.

[10] H. Zhou, J. Sun, G. Turk, and J. M. Rehg, "Terrain synthesis from digital elevation models," *IEEE transactions on visualization and computer graphics*, vol. 13, no. 4, pp. 834–848, 2007.

[11] F. P. Tasse, J. Gain, and P. Marais, "Enhanced texture-based terrain synthesis on graphics hardware," in *Computer Graphics Forum*, vol. 31, pp. 1959–1972, Wiley Online Library, 2012.

[12] J. Gain, B. Merry, and P. Marais, "Parallel, realistic and controllable terrain synthesis," in *Computer Graphics Forum*, vol. 34, pp. 105–116, Wiley Online Library, 2015.

[13] O. Argudo, C. Andujar, A. Chica, E. Guérin, J. Digne, A. Peytavie, and E. Galin, "Coherent multi-layer landscape synthesis," *The Visual Computer*, vol. 33, no. 6, pp. 1005–1015, 2017.

[14] J.-D. Génevaux, E. Galin, A. Peytavie, E. Guérin, C. Briquet, F. Grosbellet, and B. Benes, "Terrain modelling from feature primitives," in *Computer Graphics Forum*, vol. 34, pp. 198–210, Wiley Online Library, 2015.

[15] F. P. Tasse, A. Emilien, M.-P. Cani, S. Hahmann, and N. Dodgson, "Feature-based terrain editing from complex sketches," *Computers & graphics*, vol. 45, pp. 101–115, 2014.

[16] N. Chiba, K. Muraoka, and K. Fujita, "An erosion model based on velocity fields for the visual simulation of mountain scenery," *The Journal of Visualization and Computer Animation*, vol. 9, no. 4, pp. 185–194, 1998.

[17] B. Beneš and R. Forsbach, "Visual simulation of hydraulic erosion," 2002.

[18] B. Beneš, V. Těšínskỳ, J. Hornyš, and S. K. Bhatia, "Hydraulic erosion," *Computer Animation and Virtual Worlds*, vol. 17, no. 2, pp. 99–108, 2006.

[19] J. Vanek, B. Benes, A. Herout, and O. Stava, "Large-scale physics-based terrain editing using adaptive tiles on the gpu," *IEEE computer graphics and applications*, vol. 31, no. 6, pp. 35–44, 2011.

[20] K. X. Whipple and G. E. Tucker, "Dynamics of the stream-power river incision model: Implications for height limits of mountain ranges, landscape response timescales, and research needs," *Journal of Geophysical Research: Solid Earth*, vol. 104, no. B8, pp. 17661–17674, 1999.

[21] J. Braun and S. D. Willett, "A very efficient o (n), implicit and parallel method to solve the stream power equation governing fluvial incision and landscape evolution," *Geomorphology*, vol. 180, pp. 170–179, 2013.

[22] R. L. Cook, "Stochastic sampling in computer graphics," *ACM Transactions on Graphics (TOG)*, vol. 5, no. 1, pp. 51–72, 1986.

[23] N. Corporation, "*NVIDIA CUDA Toolkit Documentation*," 2020.

[24] N. Corporation, "*NVIDIA Turing Architecture White Paper*," tech. rep., 2018.

[25] R. Bridson, "Fast poisson disk sampling in arbitrary dimensions.," *SIGGRAPH sketches*, vol. 10, p. 1, 2007.

[26] X. Ying, S.-Q. Xin, Q. Sun, and Y. He, "An intrinsic algorithm for parallel poisson disk sampling on arbitrary surfaces," *IEEE transactions on visualization and computer graphics*, vol. 19, no. 9, pp. 1425–1437, 2013.

[27] T.-T. Cao, A. Nanjappa, M. Gao, and T.-S. Tan, "A gpu accelerated algorithm for 3d delaunay triangulation," in *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 47–54, 2014.