UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

OPTIMIZACIÓN DE SOFTWARE DE VISUALIZACIÓN Y DETECCIÓN DE PATRONES DE DRENAJE Y TERRAZAS FLUVIALES EN SUPERFICIES DE TERRENO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

JOSÉ TOMÁS PEFAUR PUMARINO

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
ALEXANDRE BERGEL
SERGIO OCHOA DELORENZI

SANTIAGO DE CHILE
2016

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: JOSÉ TOMÁS PEFAUR PUMARINO
FECHA: 2016
PROF. GUÍA: NANCY HITSCHFELD KAHLER

OPTIMIZACIÓN DE SOFTWARE DE VISUALIZACIÓN Y DETECCIÓN DE PATRONES DE DRENAJE Y TERRAZAS FLUVIALES EN SUPERFICIES DE TERRENO

La Geomorfología fluvial corresponde al estudio de los procesos de formación y sedimentación de los ríos, y de su interacción con el entorno, lo que entrega información sobre la "historia de vida"de un terreno. En este contexto, existen dos elementos de estudio interesantes: las redes de drenaje y las terrazas fluviales.

Runnel es un software que tiene por objetivo visualizar y detectar patrones de drenaje y terrazas fluviales sobre terrenos representados por grillas o triangulaciones de éstas. Si bien el funcionamiento de Runnel es correcto, éste tiene un gran problema: el tiempo de ejecución de sus principales algoritmos. A medida que el tamaño del terreno crece, el tiempo de ejecución aumenta considerablemente. Con el fin de mejorar este problema, se decidió paralelizar los algoritmos de detección de redes de drenaje: Peucker, Callaghan, RWFlood y Ángulo Diedro. Al mismo tiempo, se decidió implementar una triangulación simplificada, la cual disminuye el número de triángulos, y en consecuencia, se disminuye el tiempo de ejecución de los algoritmos que usan la triangulación.

Para lograr la paralelización se utilizó OpenCL, herramienta que permite la ejecución de un código paralelo tanto en GPU como en CPU de forma indistinguible. Para la triangulación simplificada se utilizó un algoritmo basado en la eliminación de uno de los vértices de aquellos triángulos que cumplen una condición predeterminada.

Como resultado de la paralelización se obtuvieron mejoras significativas con los algoritmos de Peucker (speed-up mínimo: 1.19 y speed-up máximo: 13.63), Callaghan (speed-up mínimo: 5.87 y speed-up máximo: 19.82) y Ángulo Diedro (speed-up mínimo: 1.43 y speed-up máximo: 23.53). La triangulación simplificada también entregó mejoras en rendimiento, pero con menor impacto que la paralelización (speed-up mínimo: 1.14 y speed-up máximo: 1.94). El único algoritmo que no resultó en mejoras en su tiempo de ejecución (en la mayoría de los casos de prueba) fue el algoritmo RWFlood (speed-up mínimo: 0.16 y speed-up máximo: 2.93).

Junto con el desarrollo de los algoritmos paralelos se adquirió conocimiento sobre las diferencias de rendimiento de una CPU con una GPU. Se tuvo que ahondar en la arquitectura de cada una y reconocer el tipo de problema que cada una puede resolver de manera óptima.

Se propone como trabajo futuro solucionar el uso excesivo de la memoria en la triangulación, analizar del impacto de la triangulación simplificada en los terrenos, solucionar el problema del tiempo de ejecución de RWFlood y paralelizar otros algoritmos.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

La Geomorfología fluvial corresponde al estudio de los procesos de formación y sedimentación de los ríos, y de su interacción con el entorno, lo que entrega información sobre la "historia de vida" de un terreno. En este contexto, existen dos elementos de estudio interesantes: las redes de drenaje y las terrazas fluviales. Por red de drenaje se entiende el conjunto de ríos, vaguadas, barrancos y huellas impresas en el terreno que deja la circulación constante e/o intermitente del agua. Las redes de drenaje se clasifican en distintos patrones según su forma geométrica. La constatación de cierto patrón de drenaje en un lugar da indicios de ciertas características geológicas de la zona, tales como la presencia de fracturas en la roca, volcanes y otros procesos o eventos geológicos [9]. En cuanto a las terrazas fluviales, estas son zonas planas situadas en torno a la ribera de un río, o de sus valles y llanuras aluviales, usualmente en forma escalonada. Estas terrazas se forman por sedimentación del material transportado por el río a lo largo del tiempo, y la presencia de estas terrazas da cuenta de cambios en la elevación del cauce principal, lo cual puede responder a eventos y procesos tectónicos y climatológicos, entregando una visión histórica de la fisiografía de la zona [3].

Actualmente, para detectar estos elementos en una superficie, los investigadores revisan manualmente la información entregada por imágenes satelitales. Tras identificar sectores de interés, se deben hacer pruebas en terreno para estudiar y verificar lo potencialmente relevante de la zona. La existencia de alguna herramienta automatizada que permita identificar estas zonas de interés evitaría todo este trabajo.

Con el fin de cubrir esta necesidad es que se creó la herramienta llamada Runnel [10]. Runnel es un software que tiene por objetivos visualizar y detectar patrones de drenaje y terrazas fluviales sobre terrenos representados por grillas o triangulaciones de éstas. Se pueden identificar dos etapas de desarrollo las cuales han llevado al estado actual de Runnel. En la primera etapa de desarrollo fueron implementados todos los aspectos de visualización de terrenos y gran parte de los algoritmos de reconocimiento de patrones de drenaje [10]. En la segunda etapa se mejoró la robustez y el desempeño de los algoritmos de reconocimiento de patrones de drenaje ya implementados y se agregaron otros. También se agregó la detección de terrazas fluviales [6].

Si bien actualmente existen aplicaciones que tienen algunas de las funcionalidades de Runnel, estas no permiten la clasificación de los patrones de drenaje y ni la identificación de terrazas

fluviales (e.g., RiverTools [2]). Por otro lado, la mayoría de estas alternativas son de pago.

1.1. Motivación

Runnel tiene las siguientes funcionalidades:

1. Visualización de terrenos (con datos de archivos TIFF o datos de Google Earth). La visualización de un terreno se puede ver en la Figura 1.1 y en la Figura 1.2. La Figura 1.1 muestra el terreno visto desde el eje Z. Se utiliza una escala de grises, en donde mientras más oscuro es un punto indica que es más profundo, mientras más claro sea un punto indica mayor altura. Esto puede ver en la Figura 1.2, en donde se muestra una vista lateral del terreno, quedando en evidencia la relación del color con la altura de cada punto.
2. Visualización e Identificación de ríos mediante el algoritmo Peucker [8], Angulo Diedro [10], Callaghan [8] o RWFlood [11]. Un ejemplo de resultado de la ejecución del algoritmo de Peucker se puede ver en la Figura 1.3.
3. Modelar flujo hídrico desde un punto arbitrario del terreno mediante el algoritmo Callaghan o Gradient[10]
4. Visualización e Identificación de la red de drenaje mediante el algoritmo de Callaghan o Peucker. Un ejemplo de resultado de la ejecución del algoritmo de Callaghan se puede ver en la Figura 1.4.
5. Identificación de patrones de drenaje mediante el algoritmo Zhang Guillbert[13]
6. Visualización e Identificación de terrazas fluviales mediante el algoritmo Normal Vector Similarity[6]. Un ejemplo de resultado de la ejecución del algoritmo Normal Vector Similarity se puede ver en la Figura 1.5.
7. Exageración de terreno
8. Modificación de atributos de configuración. Los atributos que se pueden modificar son los colores usados para las distintas visualizaciones. La ventana de configuración se puede ver en la Figura 1.6.

La principal limitación del software son los tiempos de ejecución de sus principales algoritmos. Si bien el funcionamiento de Runnel es correcto y el tiempo de ejecución (tanto de los algoritmos de reconocimiento de patrones como los de visualización) sobre superficies de prueba es aceptable, el tamaño de las superficies sobre las cuales los usuarios reales del software desearían trabajar son considerablemente mayores, aumentando significativamente el tiempo de ejecución.

Podemos identificar 2 estrategias para abordar este problema:

Paralelización: Debido a la naturaleza de los algoritmos de detección redes de drenaje (la forma de recorrer y procesar el terreno) es posible procesar distintos sectores del terreno en forma independiente y luego unir sus resultados, lo que permite la paralelización de éstos.

Triangulación Simplificada: Runnel recibe como entrada una grilla con puntos en tres dimensiones (x,y,z). A partir de estos puntos es que se crea una triangulación, la cual es usada para la visualización del terreno, en el algoritmo de detección de terrazas fluviales (Normal Vector

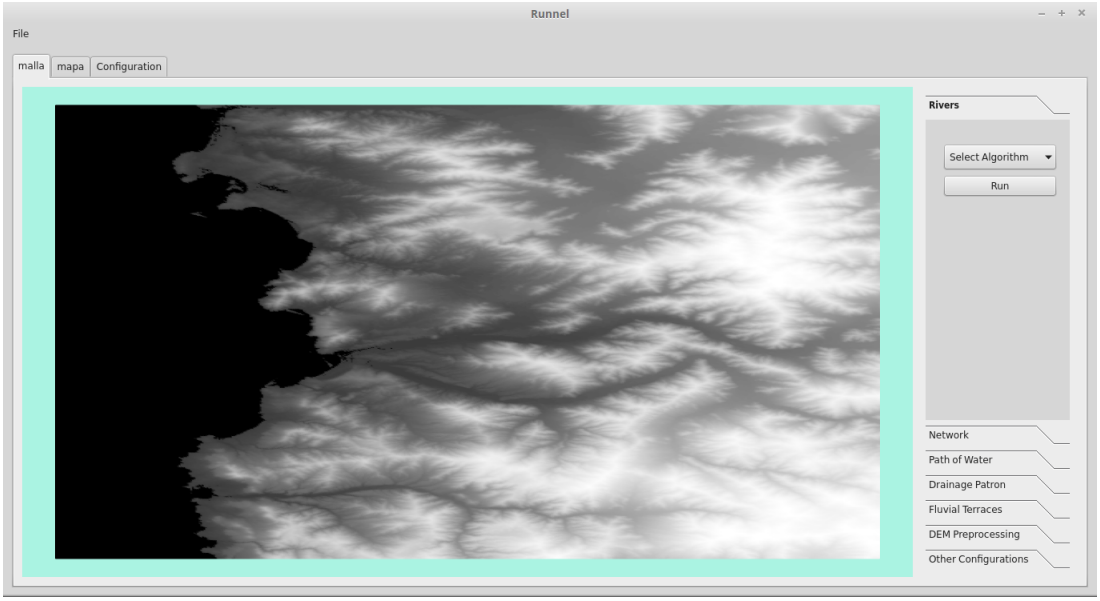


Figura 1.1: Visualización de un terreno en Runnel visto desde el eje Z.

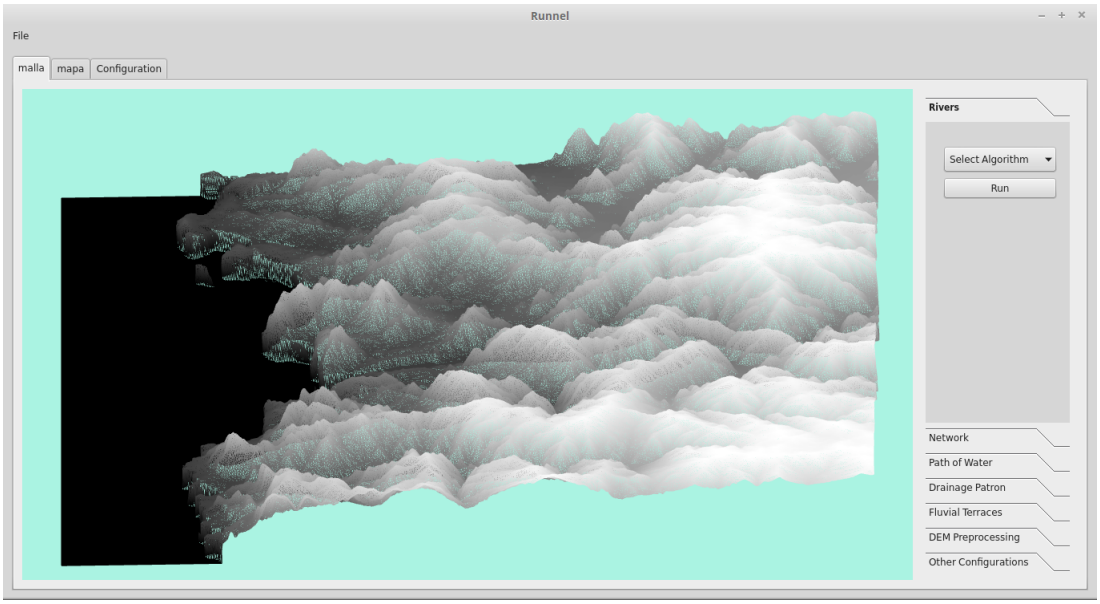


Figura 1.2: Visualización lateral de un terreno en Runnel.

Similarity) y en uno de los algoritmos de detección de redes de drenaje (Ángulo Diedro). Este último algoritmo no da buenos resultados, principalmente porque la triangulación es generada dividiendo cada cuadrado de la grilla en dos triángulos, todos del mismo tamaño y con la misma forma. Esto hace que se agreguen valles o cimas donde no las hay, aumentando los errores de precisión que ya traen estos datos debido a la forma en que son obtenidos y a la resolución considerada.

Con el objetivo de evitar este problema y de eliminar información redundante de una malla (triángulos de zonas casi planas, para así dejar solo las que tienen una diferencia de altura que se considera importante para los algoritmos que se desean aplicar) se aplicará sobre la

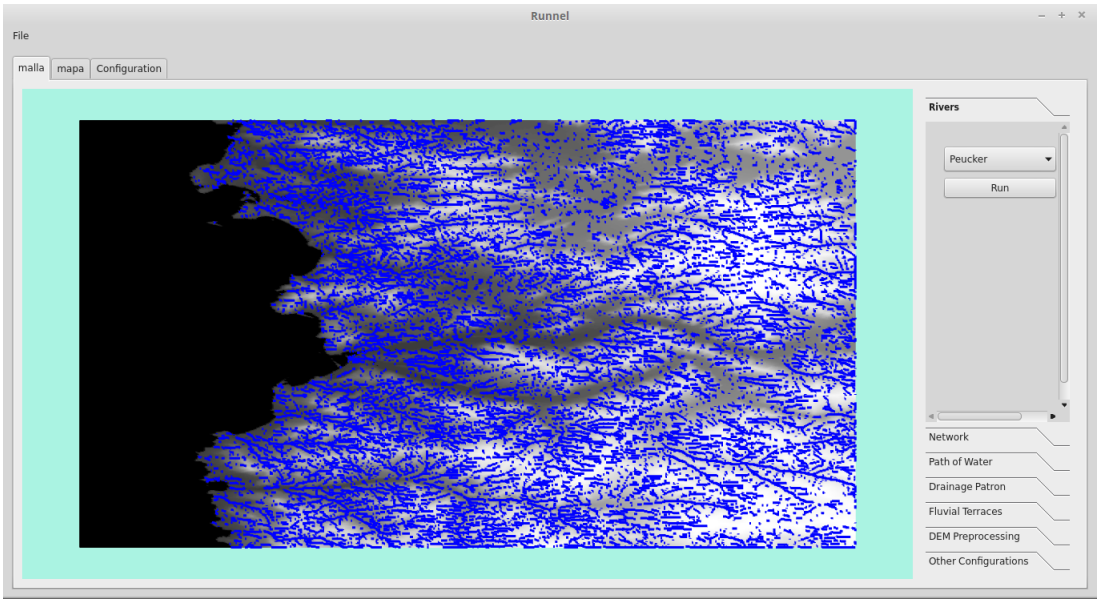


Figura 1.3: Visualización del resultado de la ejecución del algoritmo de Peucker en Runnel.

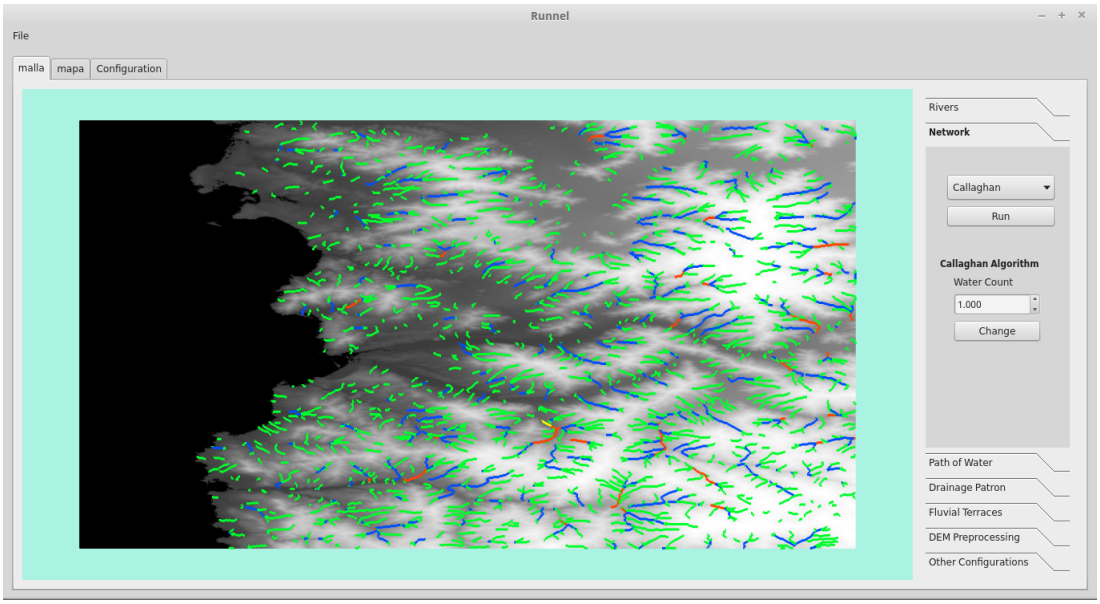


Figura 1.4: Visualización de la ejecución del algoritmo de Callaghan en Runnel.

triangulación de la grilla algoritmos de simplificación de mallas. Estos algoritmos existen en la literatura y han sido aplicados en otros ámbitos pero no en el contexto de patrones de drenaje. La idea es evaluar si estos nos sirven para generar una triangulación más representativa del relieve antes de aplicar el algoritmo de ángulo diedro.

Si bien la principal razón por la cual es deseable implementar esta simplificación es mejorar la precisión de uno de los algoritmos, al mismo tiempo se verá beneficiado el tiempo de ejecución de los algoritmos que trabajan con la triangulación, ya que la simplificación disminuye la cantidad de triángulos a procesar. Por lo mismo hay que considerar que las mejoras que se obtengan en rendimiento con esta estrategia son solo experimentales, ya que se necesitará

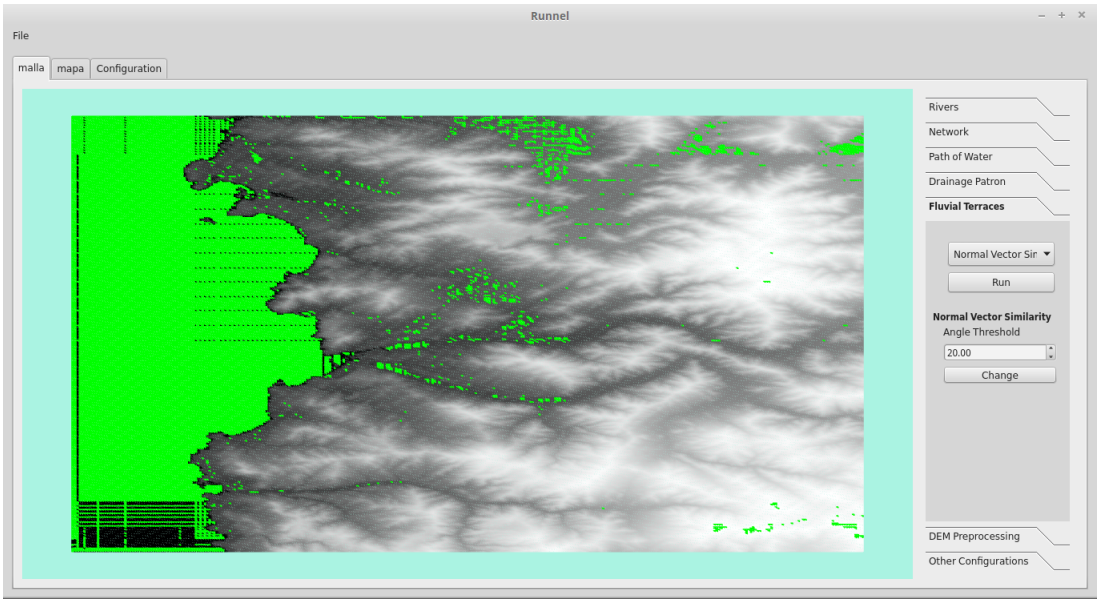


Figura 1.5: Visualización de la ejecución del algoritmo Normal Vector Similarity.

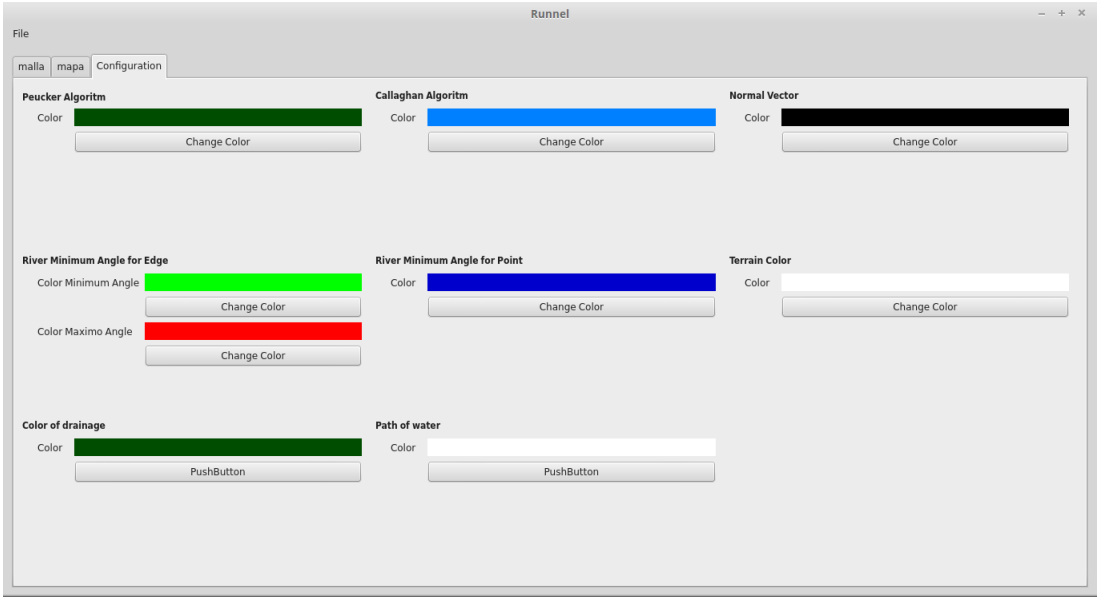


Figura 1.6: Ventana de configuración de Runnel.

un análisis más profundo para verificar si la simplificación no afecta de forma negativa la representación del terreno.

1.2. Objetivos

1.2.1. Objetivo general

El principal objetivo de este trabajo es la optimización de Runnel con el fin mejorar el desempeño computacional de sus principales algoritmos, dado que la principal limitación que tiene el software es el tiempo de ejecución de éstos.

1.2.2. Objetivos específicos

1. Paralelización de algoritmos de detección de redes de drenaje en superficies de terreno:
    - Algoritmos que actúan sobre la grilla:
        - Peucker
        - Callaghan
        - RWFlood
    - Algoritmo que actúan sobre la triangulación:
        - Ángulo diedro
2. Implementación de una triangulación simplificada

1.3. Metodología

El plan de trabajo para poder abordar el proyecto contempló las siguientes etapas: Investigación, Análisis y Desarrollo.

1.3.1. Investigación

El objetivo de esta primera etapa fue comprender la estructura y funcionamiento del código fuente, estudiar las tecnologías en uso e instruirse de nuevas tecnologías que se podrían utilizar.

Dentro de las tecnologías ya usadas en el software se destacan:

- **Qt**: Framework para el desarrollo de software multiplataforma. Es principalmente usado para el desarrollo de aplicaciones de escritorio con interfaz gráfica[4].
- **OpenGL**: API multiplataforma para el renderizado de gráficos 2D y 3D [5].
- **Shaders**: Programas que permiten personalizar las primitivas a ser renderizadas.

Para la paralelización de los algoritmos se decidió utilizar alguna tecnología que nos permitiera la ejecución en la GPU (GPGPU: General-Purpose GPU). Las razones de esta decisión tiene relación con que las restricciones que tiene el uso de una GPU no resultan ser un problema para los problemas que se desean resolver y que dentro de las alternativas de hardware que existen

para paralelizar, las GPU son las unidades de procesamiento más económicas y de fácil acceso. Al mismo tiempo, se podrían ver mejores resultados que al paralelizar en CPU (dependiendo de la naturaleza del problema a resolver).

Las alternativas a considerar son las siguientes:

- **CUDA**: API de bajo nivel. Permite la utilización de tarjetas de video NVIDIA para GPGPU [6]
- **OpenCL**: API de bajo nivel. Permite la programación paralela usando CPU o GPU (entre otros) de forma indistinguible [7].
- **OpenACC**: API de alto nivel. Permite la programación paralela usando CPU o GPU de forma indistinguible mediante directivas para el compilador [8].

1.3.2. Análisis

En esta segunda etapa se hizo un estudio del software, los algoritmos a paralelizar y la implementación de estos, la triangulación implementada y la triangulación simplificada a utilizar. Esto con el fin de tener una idea de como abordar las distintas optimizaciones.

1.3.3. Desarrollo

El objetivo de esta última etapa es la implementación de las optimizaciones considerando la información obtenida en la etapa de análisis.

La validación de las distintas optimizaciones se realizará mediante la comparación de los resultados de la ejecución de los algoritmos optimizados con los resultados de la ejecución de los algoritmos originales (tanto el resultado visual como su tiempo de ejecución).

1.4. Contenido de la Memoria

A continuación se indica el contenido de cada uno de los capítulos que siguen en la memoria:

- **Antecedentes**: Se comienza explicando las diferencias de paralelizar en GPU y CPU, para luego dar paso a las razones por la cual se seleccionó OpenCL como herramienta de paralelización. Luego se analiza la arquitectura de OpenCL.
- **Análisis del Software**: Se explican los principales módulos de Runnel y como estos interactúan entre ellos. Se sigue con una revisión de cada uno de los algoritmos a paralelizar. El capítulo termina con la explicación de como se representa un terreno en Runnel y como se crea la triangulación de éste.
- **Diseño**: Se muestra el diseño de la solución. Primero se explica la estrategia que se usará para paralelizar cada uno de los algoritmos. Luego se define el algoritmo que se aplicará para lograr una triangulación simplificada.

- **Implementación**: Se expone la implementación de la solución. Acá se muestra como se llevó a la práctica lo definido en el capítulo anterior, usando partes del código como referencia.
- **Resultados**: Se muestran y analizan los resultados de rendimiento de los algoritmos implementados. Se compara el rendimiento en distintas unidades de procesamiento de los algoritmos paralelizados con respecto a la versión secuencial. Los resultados de la triangulación simplificada son comparados con el rendimiento de la triangulación regular.
- **Conclusión**: Consiste en la conclusión general del trabajo realizado y cuáles son los cambios que se podrían implementar en el futuro con el fin de mejorar el software.
- **Anexo**: Se muestra parte del código que no fue explicado en el capítulo de Implementación.

Capítulo 2

Antecedentes

A continuación se realiza una descripción de los conceptos, tecnologías y algoritmos importantes para la realización de esta memoria. Se comienza explicando las diferencias de paralelizar en GPU y CPU, para luego dar paso a las razones por la cual se seleccionó OpenCL como herramienta de paralelización. El capítulo termina con un análisis de la arquitectura de OpenCL.

2.1. Paralelización: CPU vs GPU

Para entender en qué tipo de problema es conveniente utilizar una GPU o una CPU, es necesario conocer la arquitectura de cada uno de éstos.

La CPU utiliza una parte importante de espacio físico en memorias cache de distintos niveles. Esto con el fin de disminuir el overhead de los accesos a memoria. Como consecuencia, hay menos espacio para alojar unidades de procesamiento, limitando así la cantidad de instrucciones en paralelo que puede ser ejecutadas. Por otro lado, las unidades de procesamiento de una CPU son bastante complejas: pueden recibir señales de interrupción, pueden hacer uso de memoria virtual, tienen una gran variedad de instrucciones y tienen un sistema de predicción de saltos bastante complejo y eficiente.

A diferencia de una CPU, la mayor parte del espacio físico de una GPU es utilizado en unidades de procesamiento, pudiendo así procesar muchas más instrucciones en paralelo que una CPU, pero sus unidades de procesamiento son mucho más lentas y tiene un set de instrucciones mucho menos complejo que consiste principalmente en operaciones de aritmética simple. En consecuencia, el espacio designado para la memoria cache es bastante pequeño, por lo que los accesos a memoria son mucho más costosos que en una CPU. Con el fin de solucionar este problema, las GPU suelen tener más interfaces de memoria, obteniendo así anchos de banda mucho mayores que los de una CPU. Es decir, si el ancho de banda es usado eficientemente, el efecto de la latencia de la memoria se ve disminuido. Una GPU tampoco tiene interrupciones, no tiene memoria virtual y es recomendable evitar saltos condicionales ya que disminuye su rendimiento considerablemente. Es decir, mientras más parecido es lo que hace cada una de sus unidades de procesamiento, mejor será el rendimiento.

Dicho de otra forma, si el problema a resolver tiene una gran cantidad de datos que se puede procesar en paralelo, en cada unidad de procesamiento se hará prácticamente el mismo procedimiento y la principal fuente de latencia son operaciones aritméticas, entonces es un problema ideal para resolver en GPU. En caso que el problema tenga como principal fuente de latencia los accesos a memorias y/o que las instrucciones a paralelizar sean muy complejas, puede ser mejor opción la paralelización en CPU.

2.2. Elección de tecnología para la paralelización

Recordemos que se decidió utilizar alguna herramienta que nos permitiera la ejecución en GPU con el fin de estudiar este tipo de tecnología. Esto es conocido como tecnologías GPGPU: General-Purpose GPU.

La primera tecnología que se consideró fue CUDA. Si bien pareciera ser la tecnología de GPGPU más conocida y más madura, su principal problema es que nos obliga a utilizar una tarjeta de video NVIDIA moderna, lo que es una limitación tanto para el usuario como para el desarrollador.

Como segunda alternativa se consideró OpenCL. El funcionamiento en comparación con CUDA es básicamente igual, el modelo programación solo tiene pequeñas variaciones. La gran ventaja que tiene por sobre CUDA es que no se restringe al uso de una tarjeta de video NVIDIA, sino que soporta una gran variedad de tarjetas de videos de distintas marcas, e incluso permite ejecutar el mismo código tanto en GPU como en CPU sin grandes modificaciones. Esto se debe a que es una tecnología de estándar abierto. Por otro lado, no existe una gran diferencia en cuanto a eficiencia de estas dos tecnologías [5]. El punto negativo de OpenCL es que no es una tecnología tan madura como CUDA, por lo que no existen muchas herramientas de ayuda para el desarrollador.

Por último se consideró OpenACC. La idea de OpenACC es dar instrucciones al compilador en forma de comentarios en el código que se desea paralelizar, evitando así la modificación del código y pasando el trabajo de paralelización al compilador. Pareciera ser una excelente opción debido a su simplicidad.

El gran problema de OpenACC es que ha sido desarrollado para ser usado con compiladores privativos. Si bien hoy en día existe una versión para el compilador GCC, esta se encuentra en una etapa de prueba temprana, lo que implica que es poco confiable y su documentación es escasa.

Considerando las ventajas y desventajas de cada una de las alternativas, se decidió trabajar con OpenCL, debido a la flexibilidad que entrega tanto al desarrollador (hardware y compilador) como al usuario (hardware). Al mismo tiempo, no sabemos de antemano si efectivamente se verán mejores resultados al paralelizar en GPU que al paralelizar en CPU, ya que esto depende de la naturaleza del problema a resolver. Si bien esto se puede deducir al analizar en problema, OpenCL nos permite alternar fácilmente entre GPU y CPU, por lo que tomar una decisión anticipada de donde ejecutar el algoritmo no es necesario, y es posible decidirlo comparando directamente sus tiempos de ejecución.

2.3. Arquitectura OpenCL

Esta sección está basada en la especificación de OpenCL 1.2 [4].

2.3.1. Modelo de plataforma

El modelo de plataforma de OpenCL está definido en la Figura 2.1. Este modelo consiste en un **host** que se conecta a uno o más OpenCL **devices**. Un OpenCL device está dividido en uno o más **compute units** (CUs), los cuales se subdividen en uno o más **processing elements** (PEs). La ejecución ocurre en los processing elements.

El programa OpenCL es ejecutado en un host usando la implementación de la plataforma seleccionada por el host. Lo que se ejecuta en los processing elements de un device depende de los comandos que sean enviados desde el host.

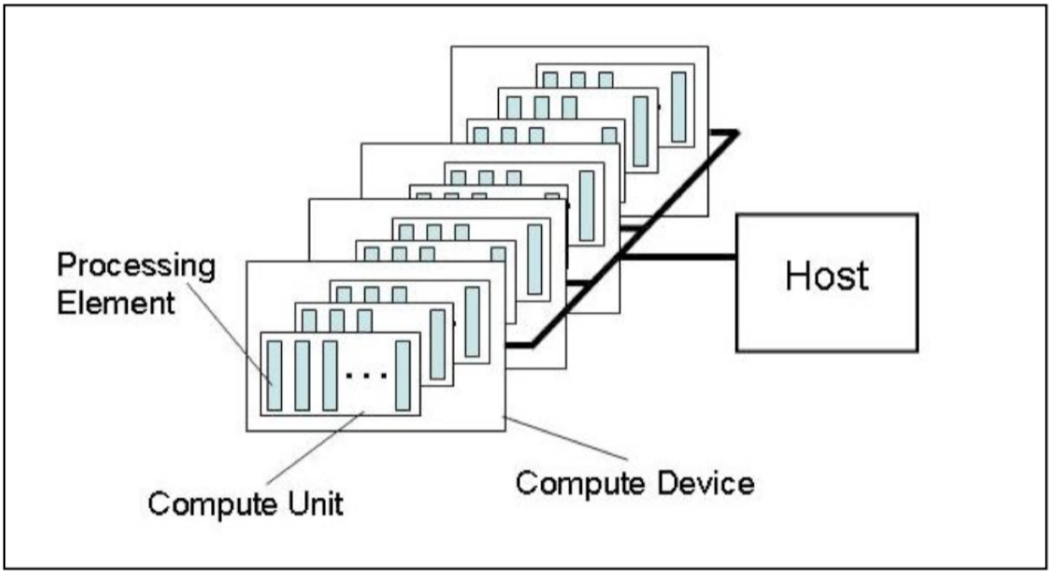


Figura 2.1: Modelo de plataforma de OpenCL.

2.3.2. Modelo de ejecución

El modelo de ejecución de un programa OpenCL consiste en dos partes: **kernels** que son ejecutados en uno o más OpenCL devices y un **host program** ejecutado en el host. El host program define el contexto para los kernels y administra su ejecución.

El core del modelo de ejecución de OpenCL está definido por como se ejecutan los kernels. Cuando el host envía un kernel para su ejecución se define un espacio de índices. Una instancia del kernel es ejecutada por cada punto en este espacio de índices. Esta instancia del kernel es llamada **work-item** y es identificada por su punto en el espacio de índices, el cual entrega un **global id** distinto para cada work-item (en otros modelos, por ejemplo en CUDA, un work-item sería análogo

a un thread). Cada work-item ejecuta el mismo código pero el camino especifico de ejecución puede variar.

Los work-items están agrupados en **work-groups**. Los work-groups entregan una visión más global del espacio de índices. A cada uno se le asigna un **work-group id** distinto que tiene las mismas dimensiones que el espacio de índices usado para los work-items. A los work-items se les asigna un **local id** dentro de su work group. De esta forma cada work-item puede ser identificado por su global id o mediante una combinacion de su local id y su work-group id. Los work-items de un cierto work-group se ejecutan concurrentemente en los processing elements de una misma compute unit.

El espacio de índices en OpenCL es llamado **NDRange**. Un NDRange es un espacio de índices de N-dimensiones, siendo N igual a 1, 2 o 3. Un NDRange se define mediante un arreglo de largo N, especificando el tamaño en cada dimensión del espacio de índices. Cada global id y local id son tuplas de N-dimensiones.

Los ids de los work-groups son asignados de forma similar que los global ids. Un arreglo de largo N define el número de work-groups en cada dimensión. Los work-items son asignados a un work-group y se les asigna un local id.

Un ejemplo de un NDRange se puede ver en la Figura 2.2.

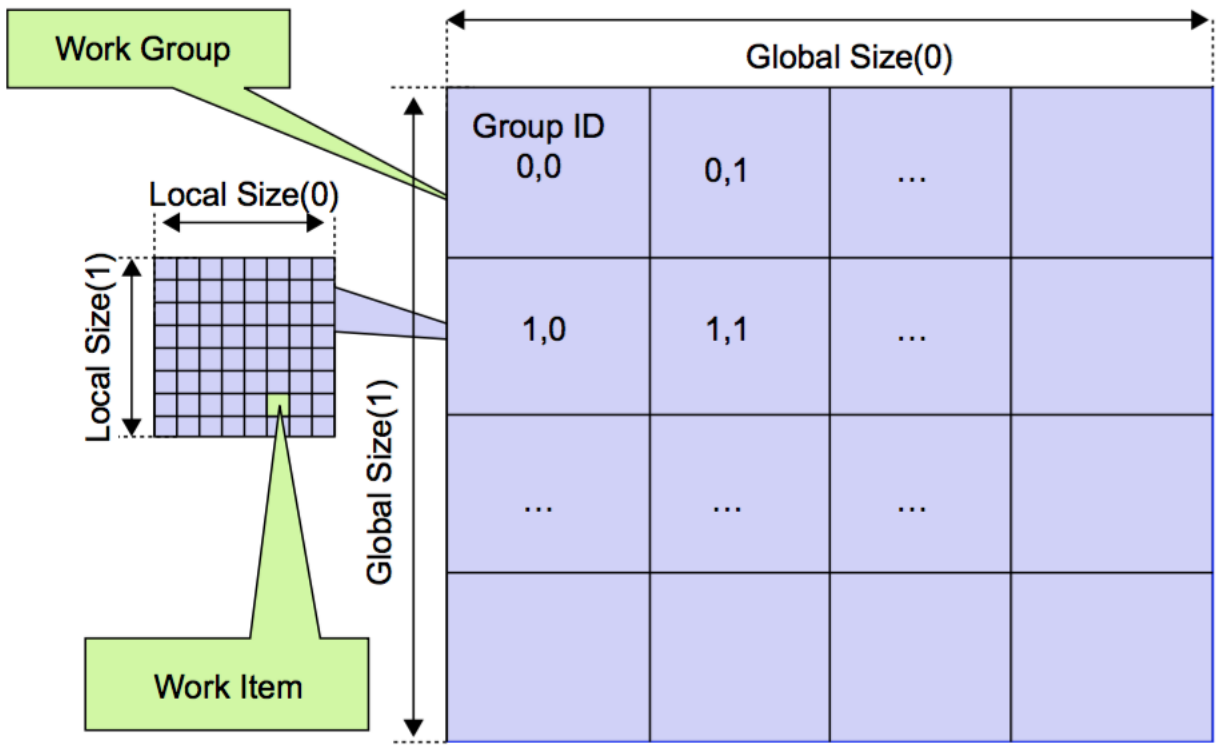


Figura 2.2: Representación de un NDRange.

2.3.3. Modelo de la memoria

El modelo de la memoria se puede ver en la Figura 2.3. Los work-items tienen acceso a cuatro regiones de memoria:

- **Global Memory**: Permite la lectura/escritura de todos los work-items en cualquier work-group.
- **Constant Memory**: Una región de la memoria global que permanece constante durante la ejecución del kernel. Es el host quien la asigna e inicializa.
- **Local Memory**: Memoria de un work-group. Puede ser usada para asignar variables que son compartidas por los work-items del work-group. Puede ser memoria dedicada o parte de la memoria global, depende del device.
- **Private Memory**: Región privada a un work-item. Variables definidas en acá son solo visibles para el work-item que las define.

En la Tabla 2.1 se pueden ver las diferencias de acceso que tiene el host y el device en las distintas regiones de memoria.

Memory Type	Host access	Device access
Global memory	Dynamic allocation. Read/write access	No allocation. Read/write access by all work items in all work groups. Large and slow, but may be cached in some devices
Constant memory	Dynamic allocation. Read/ write access	Static allocation. Read-only access by all work items
Local memory	Dynamic allocation. No access	Static allocation. Shared read/write access by all work items in a work group
Private memory	No allocation. No access	Static allocation. Read/write access by a single work item

Tabla 2.1: Diferencias de acceso entre el host y el device en las distintas regiones de memoria.

2.3.4. OpenCL vs CUDA

Los modelos de programación de OpenCL y CUDA son bastante parecidos. Cada concepto en OpenCL suele tener uno equivalente en CUDA. Con el fin de facilitar la comprensión de OpenCL para quienes ya estén familiarizados con CUDA, en la Tabla 2.2 se pueden ver ciertos conceptos de OpenCL y su análogo en CUDA. [7].

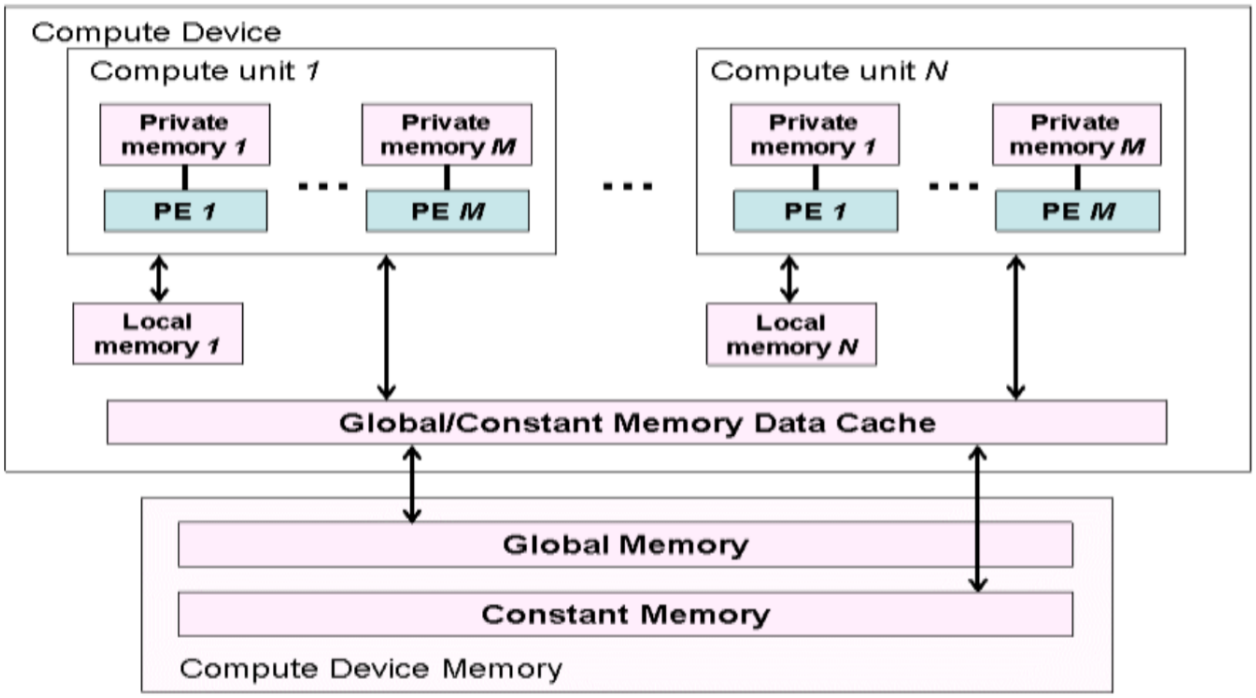


Figura 2.3: Modelo de la memoria en OpenCL.

OpenCL	**CUDA**
host	host
device	device
kernel	kernel
host program	host program
NDRange (index space)	grid
work-item	thread
work-group	block
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	registers and local memory

Tabla 2.2: Conceptos claves de OpenCL y su análogo en CUDA.

Capítulo 3

Análisis del Software

A continuación se describen los principales módulos de Runnel y como éstos interactúan entre ellos. Se sigue con la explicación de cómo se representa un terreno en Runnel. El capítulo termina con una revisión de cada uno de los algoritmos a paralelizar.

3.1. Arquitectura del Software

La Figura 3.1 muestra la arquitectura de Runnel con sus principales módulos. Aquellos marcados en rojo son los relevantes para el proyecto. El rol que cumple cada módulo se describe a continuación:

- RunnelController: Módulo central encargado de coordinar el trabajo del resto de los módulos y de la comunicación entre ellos. Contiene una clase y 190 líneas de código.
- InputData: Encargado de la carga de los distintos formatos de entrada. Se manejan datos de entrada desde archivos y de la web. Contiene 7 clases y 648 líneas de código.
- Builders: Encargado del pre-procesamiento de los datos de entrada. Se crean las estructuras de datos que serán usadas como input para el análisis de los terrenos . Módulo relevante ya que es donde se crea la triangulación (no simplificada). Contiene una clase y 120 líneas de código.
- Painters: Encargado de renderizar la malla del terreno (shaders e interfaz gráfica del terreno). Contiene 12 clases y 1529 líneas de código.
- Primitives: Contiene las primitivas usadas para la modelación del terreno (puntos, triángulos y arcos). Suma 3 clases y 251 líneas de código.
- DrainageAlgorithms: Contiene los algoritmos de detección de redes de drenaje. Módulo relevante ya que es donde se encuentra la mayoría de los algoritmos que se desean optimizar (Peucker, Ángulo Diedro, Callaghan y RWFlood). Suma un total de 6 clases y 906 líneas de código.
- FluvialTerraceAlgorithms: Contiene el algoritmo de detección de terrazas fluviales (Normal Vector Similarity). Consiste en 3 clases y 207 líneas de código.

- PatternsAlgorithms: Contiene el algoritmo de clasificación de una red de drenaje (Zhang-GuilbertAlgorithm). Suma 4 clases y 588 líneas de código.
- UI: Módulo encargado de la interfaz gráfica. Contiene 14 clases y 4110 líneas de código.

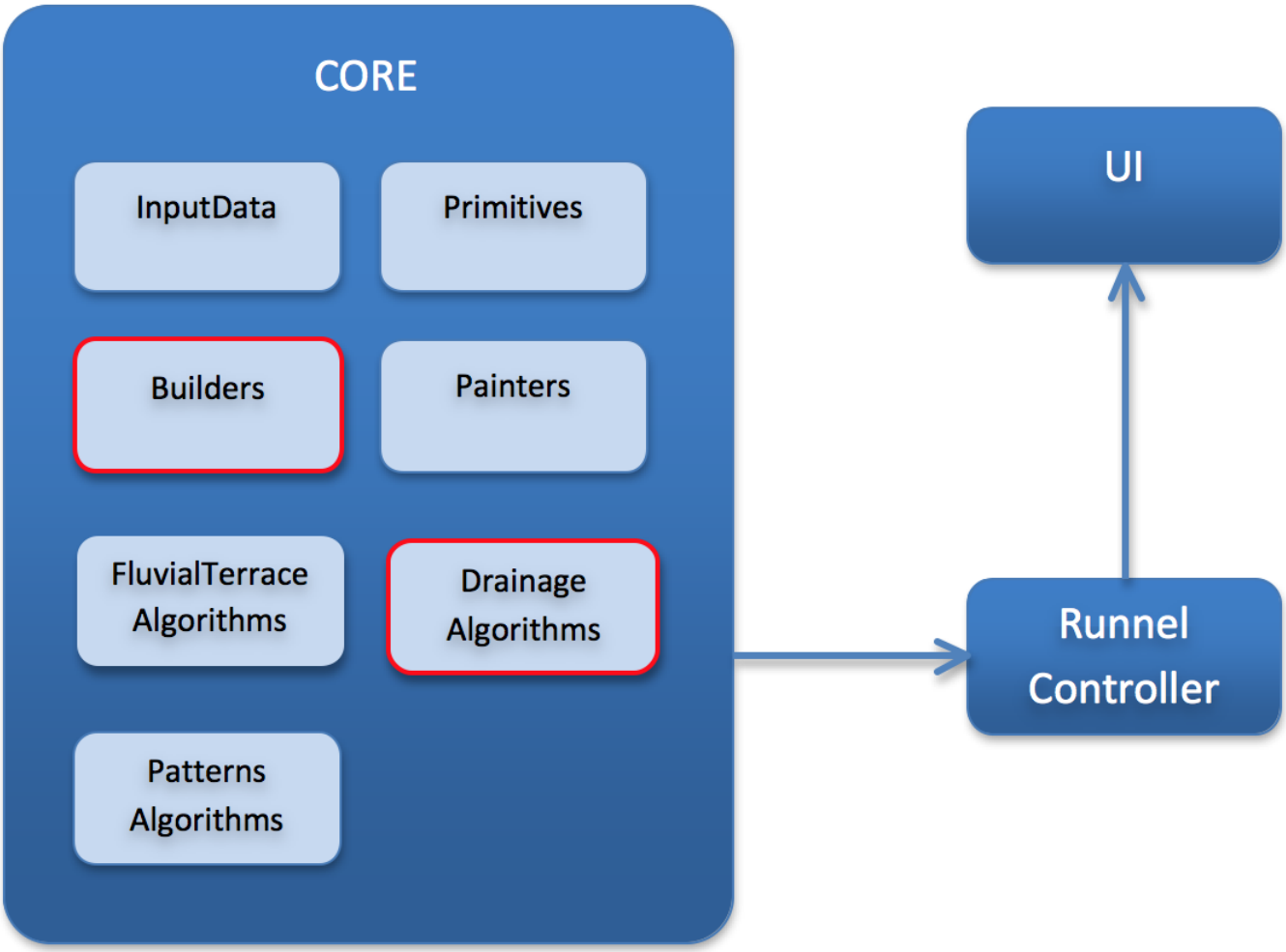


Figura 3.1: Arquitectura de Runnel.

3.2. Representación del Terreno: Grilla y Triangulación

El terreno se representa mediante el uso de tres primitivas: Point, Triangle y Edge.

- Point: Representa un punto del terreno. Cada punto tiene sus propias coordenadas (x,y,z).
- Triangle: Corresponde a los triángulos generados a través de la triangulación. Contiene punteros a los 3 Edges que lo forman.
- Edge: Arcos de un polígono, en este caso, arcos de los triángulos. Contiene punteros a los Triangle que lo utilizan y a los Point que lo forman.

Los archivos de entrada de Runnel solo contienen datos que se pueden representar directamente

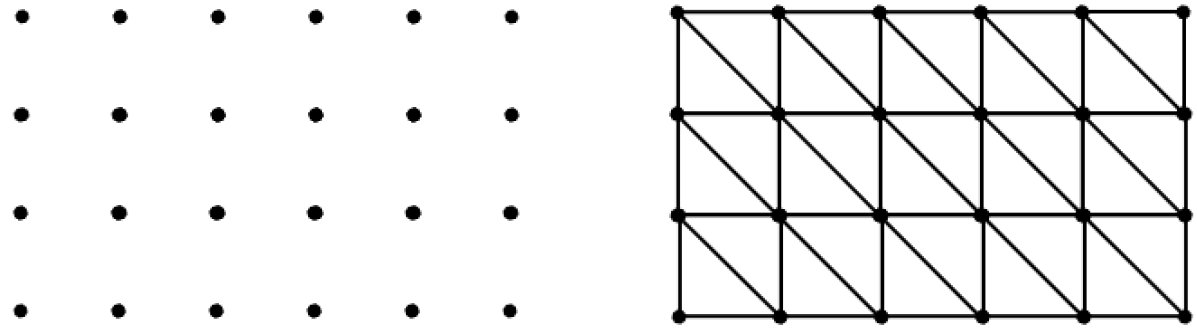
con la primitiva Point, por lo que es necesario generar las instancias de la primitiva Triangle (junto con las instancias de la primitiva Edge) mediante alguna estrategia de triangulación.

Usando la primitiva Point podemos representar el terreno mediante una grilla. Una grilla consiste en una matriz de dos dimensiones, en donde cada elemento de la matriz corresponde a un punto en el terreno, representados por tres coordenadas (x,y,z). En Runnel, cada elemento de la matriz es una primitiva Point, en donde sus coordenadas se obtienen a partir de los datos del archivo de entrada. Cabe destacar que la distancia entre un punto y sus vecinos en el eje X es siempre de una unidad. Lo mismo ocurre con los vecinos del eje Y.

Otra de las formas de representar el terreno es utilizando una triangulación. Se entiende por triangulación como la partición de un dominio de puntos en triángulos, cumpliendo que los vértices de los triángulos son los puntos originales de la nube y que toda dupla de triángulos comparte a lo más un arco [10].

La triangulación es usada para la visualización de los terrenos, en el algoritmo de detección de terrazas fluviales (Normal Vector Similarity) y en uno de los algoritmos de detección de redes de drenaje (Ángulo Diedro).

La estrategia de triangulación de Runnel es la siguiente: Cada cuadrado de la grilla, generado por cuatro puntos del terreno, produce dos triángulos, todos del mismo tamaño y forma (al considerar solamente las coordenadas x e y). Los triángulos se generan trazando una línea desde el punto inferior izquierdo hacia el punto superior derecho del cuadrado. Es decir, la altura de los puntos es completamente ignorada. De esta forma obtenemos como resultado una triangulación exageradamente regular. En la Figura 3.2 se puede ver el resultado de aplicar la triangulación sobre la grilla.



(a) Grilla (b) Triangulación

Figura 3.2: Generación de la triangulación a partir de una grilla.

3.3. Algoritmos a paralelizar

A continuación se describen los distintos algoritmos a paralelizar. Todos corresponden a algoritmos de detección de redes de drenaje, de los cuales tres actúan sobre la grilla y uno sobre la triangulación.

3.3.1. Peucker

Con el fin de facilitar la comprensión del algoritmo de Peucker, el Algoritmo 1 muestra un pseudocódigo es éste. El algoritmo itera sobre la grilla (no hace uso de la triangulación) mediante una ventana cuadrada de 4 puntos. En cada ventana se marca el punto de mayor altura. Luego, se hace lo mismo con los siguientes 4 puntos que corresponden a la siguiente ventana. Se sigue con este procedimiento hasta que se haya pasado por todas las ventanas de la grilla.

Algoritmo 1 Peucker

```
1: procedure PEUCKER(terrain)
2:     for each point p in terrain do
3:         window ← 2x2 window containing p as its top-left point
4:         Flag max height point(s) of window
```

Una vez procesadas todas las ventanas del terreno, los puntos que no han sido marcados corresponden a una red de drenaje.

Este algoritmo es de orden $\mathcal{O}(n^2)$, siendo n el máximo entre el número de columnas y el número de filas de la grilla. Esto se debe a que el total de puntos es de orden $\mathcal{O}(n^2)$ y en cada punto se realiza un conjunto de operaciones de orden constante.

En la Figura 3.3 se puede ver un ejemplo de la ejecución del algoritmo. El cuadrado rojo indica la ventana que se está procesando. Todos los puntos comienzan siendo de color negro y se marcan de color celeste cuando es el punto de mayor altura de una ventana. Terminado el algoritmo, los puntos negros son aquellos que corresponden a la red de drenaje. También se observa que cuando en un mismo cuadrado hay 2 o más valores que tienen la mayor altura, todos estos son marcados.

3.3.2. Callaghan

Con el fin de facilitar la comprensión del algoritmo de Callaghan, el Algoritmo 2 muestra un pseudocódigo es éste. Este algoritmo también trabaja sobre la grilla (no hace uso de la triangulación). La idea es considerar la acumulación de agua en cada punto, en donde cada uno tiene como base una cantidad de agua igual a 1.

Para cada punto, uno de los ocho puntos que lo rodean (o menos si es que es un punto de borde) será aquel hacia el cual el agua se drenará. Este punto será aquel que tenga la mayor diferencia de altura con el punto que estamos procesando y que al mismo tiempo tenga menor altura que éste. Una vez encontrado ese punto (que se asume único), se le sumará la cantidad de agua que tiene el punto que se está procesando. Se realizará este procedimiento en cada punto, recorriéndolos de mayor a menor altura, por lo que el primer paso consiste en ordenar los puntos de acuerdo a su altura.

Una vez terminado este procedimiento, la red de drenaje consiste en todos los puntos que tengan una acumulación de agua mayor a cierta cantidad predefinida.

Siendo n el máximo entre el número de columnas y el número de filas de la grilla, el total de

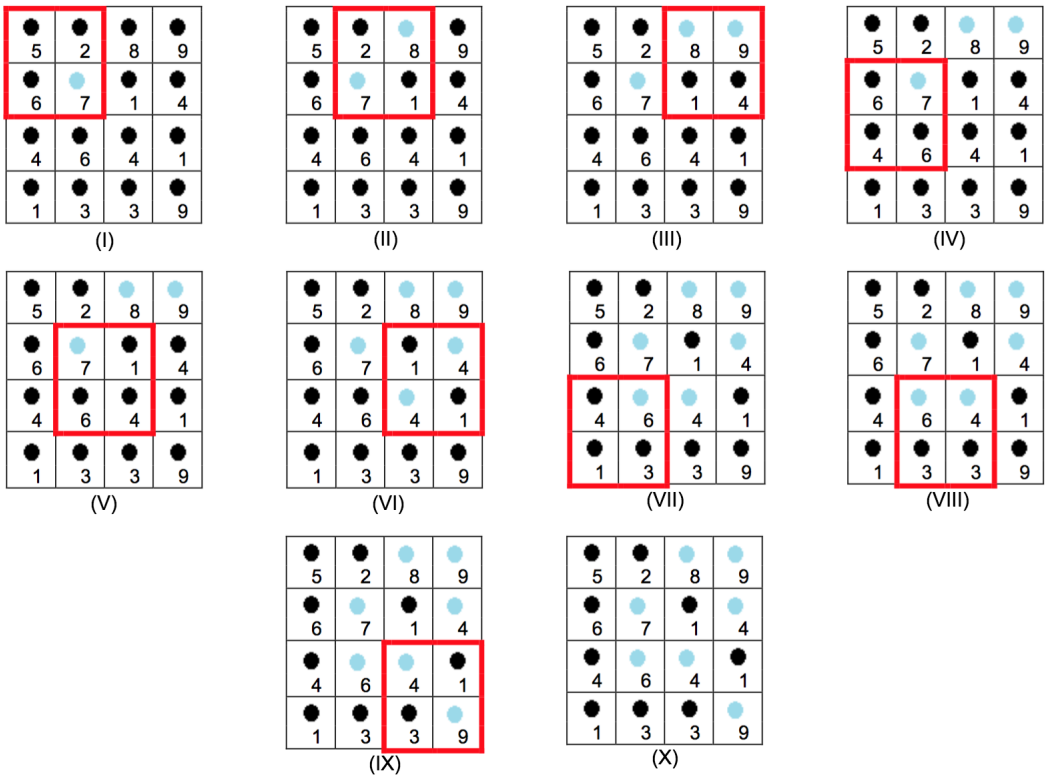


Figura 3.3: Ejemplo de ejecución del algoritmo de Peucker.

puntos en el terreno es del orden de $\mathcal{O}(n^2)$. Es por esto que ordenar nos toma $\mathcal{O}(n^2 \log n)$. Una vez ordenados se realiza a cada punto un conjunto de operaciones de orden $\mathcal{O}(1)$, demorándose así $\mathcal{O}(n^2)$. Sumando $\mathcal{O}(n^2 \log n)$ con $\mathcal{O}(n^2)$ nos queda que el algoritmo es de orden $\mathcal{O}(n^2 \log n)$.

Algoritmo 2 Callaghan

```
1: procedure CALLAGHAN(terrain)
2:     for each point p in terrain do
3:         p.water ← 1
4:     sortedPoints ← sort(points in terrain)
5:     for each point p in sortedPoints do                ▷ from highest to lowest
6:         drainNeighbor ← get the neighbor of p with the steepest descent
7:         drainNeighbor.water = p.water + drainNeighbor.water
```

3.3.3. RWFlood

Con el fin de facilitar la comprensión del algoritmo RWFlood, los Algoritmos 3 y 4 muestran un pseudocódigo es éste. Este algoritmo trabaja sobre la grilla (no hace uso de la triangulación). El algoritmo consiste en dos partes. Primero se determina la dirección de flujo en cada punto del terreno. Luego se calcula el agua acumulada en cada punto debido al flujo. Finalmente, la red de drenaje consiste en todos los puntos que tengan una cantidad de agua mayor a cierta cantidad predefinida.

La idea principal es simular que el terreno es una isla en donde el nivel del agua que la rodea va incrementando.

Cuando el nivel del agua aumenta, la isla se va inundando gradualmente a medida que el agua se encuentra con algún punto de la isla que tiene menor o igual altura que el nivel del agua actual.

Para lograr ésto se utiliza una colección de colas en donde los puntos se insertarán para posteriormente ser procesados. Para cada posible nivel del agua existe una cola asociada, siendo el nivel mínimo del agua igual a la menor altura de los puntos del borde del terreno y el nivel máximo del agua igual a la mayor altura de todos los puntos del terreno. Inicialmente, cada cola se encuentra vacía. La cola en la que un punto se inserta depende de su altura y el nivel actual del mar. Si el punto que se quiere insertar en una cola tiene menor o igual altura que el nivel del agua actual, entonces debe insertarse en la cola que corresponde al nivel del agua actual. Si el punto tiene mayor altura que el nivel del agua actual, entonces deberá insertarse en la cola que esté asociada a un nivel de agua igual a la altura de ese punto.

En un comienzo, el nivel del agua es igual a la menor altura de los puntos de borde del terreno. Todos los puntos del borde son insertados en su cola correspondiente. Posteriormente hay que sacar todos los puntos de la cola asociados al nivel del agua actual. El sacar un punto de una cola significa que el punto se inundó. Una vez que se saca un punto de una cola, es necesario ver la altura de todos los puntos adyacentes a este e insertarlos en la cola que corresponda (o ignorarlo si es que ese punto ya fue inundado previamente). Esto se debe a que puede estos puntos también deban ser inundados (si es que su altura es menor o igual al nivel del agua actual) o que deban ser almacenados para ser procesados posteriormente (ya que su altura es mayor que el nivel del agua actual, pero el agua ya llegó a uno de sus puntos vecinos). Una vez que ya no queden más puntos por inundar debido al nivel del agua actual (que no queden más puntos en la cola asociada al nivel del agua actual), se incrementará el nivel del agua en una unidad y se vuelve a realizar todo este procedimiento. Se sigue así hasta que se llegue a la altura máxima de los puntos del terreno (es decir, hasta que el terreno esté completamente inundado). Parte de este procedimiento se puede ver en la Figura 3.4. En 1(a) el nivel del agua es 70m. En 1(b) muestra como el nivel del agua ha aumentado. Notar que hay puntos de la isla que tienen menor altura que el nivel del agua actual, pero que todavía no han sido inundados ya que el agua no los alcanza. En 1(c), el nivel del agua es 99m, los puntos en la cola asociada al nivel del agua 99m son procesados. Al hacer esto, el punto con altura 100m (la cima de la derecha, vecino del punto de altura 99m) se inserta en la cola asociada a la altura 100m. Una vez que el nivel del agua es 100m, se procesan los puntos en la cola asociada a 100m. En 1(d) vemos la inundación que se produjo al procesar los puntos de la cola asociada a 100m. Finalmente, en 1(e) se ve cuando el nivel del agua es 105m.

Para determinar la dirección del flujo lo que se hace es que a medida que un punto p es sacado de una cola, todos los puntos adyacentes a p y que son insertados en una cola se les asigna una dirección de flujo que apunta a p.

El Algoritmo 3 muestra un pseudocodigo para determinar la dirección del flujo en el terreno.

Una vez que calculamos la dirección del flujo en cada punto, el siguiente paso es calcular el agua acumulada en cada punto debido al flujo.

La idea es procesar la red de flujo generada en la etapa anterior como si fuera un grafo, en

Algoritmo 3 RWFlood

```
1: procedure RWFlood(terrain)
2:     Let Q[terrain.border.minElev...terrain.maxElev] be an array of queues
3:     for each point p in terrain do
4:         p.dir ← NULL
5:     for each point p in terrain.border do
6:         Q[p.elev].insert(p)
7:         p.dir ← OutsideTerrain
8:     for z = minElev ← maxElev do
9:         while Q[z] is not empty do
10:            c ← Q[z].remove()
11:            for each point p neighbor of c where p.dir = NULL do
12:                p.dir ← c
13:                if p.elev <z then
14:                    p.elev ← z
15:                Q[p.elev].insert(p)
```

donde cada punto del terreno es un vértice y el flujo entre un punto y otro indica una arista dirigida entre esos dos puntos. Inicialmente, cada punto tiene una cantidad de agua igual a 1. Luego se itera sobre todos los puntos que aun no hayan sido visitados. Cuando encontramos un punto P el cual no tenga ninguna arista incidente será marcado como visitado y su agua acumulada se suma al agua acumulada del punto al que apunta. Una vez que se termina de procesar P, la arista que conecta P con el punto al que ésta apunta es eliminado. Si al hacer esto el punto al que P apuntaba queda sin aristas incidentes, entonces ese punto se debe procesar de manera similar al punto recién procesado.

El Algoritmo 4 muestra un pseudocodigo para calcular el agua acumulada en cada punto (next(p) corresponde al punto al cual el punto p apunta y Degree corresponde a la cantidad de vecinos que apuntan a un punto y que todavía no han sido procesados).

Considerando ambas etapas, el algoritmo es de orden $\mathcal{O}(n^2)$, siendo n el máximo entre el número de columnas y el número de filas de la grilla.

3.3.4. Ángulo Diedro

Con el fin de facilitar la comprensión del algoritmo de Ángulo Diedro, el Algoritmo 5 muestra un pseudocódigo es éste. Este algoritmo trabaja sobre la triangulación. La idea del algoritmo es que los ángulos agudos dentro de un terreno corresponden a cauces, obteniendo así la red de drenaje a partir del conjunto de cauces.

Para obtener los ángulos del terreno, el algoritmo procesa cada uno de los triángulos, calculando el ángulo formado por un triángulo y cada uno de sus vecinos. Este ángulo, conocido como ángulo diedro, se calcula por medio de las normales de los triángulos.

La siguiente fórmula entrega el calculo un ángulo diedro φ_{AB} para 2 planos A y B, donde n_A

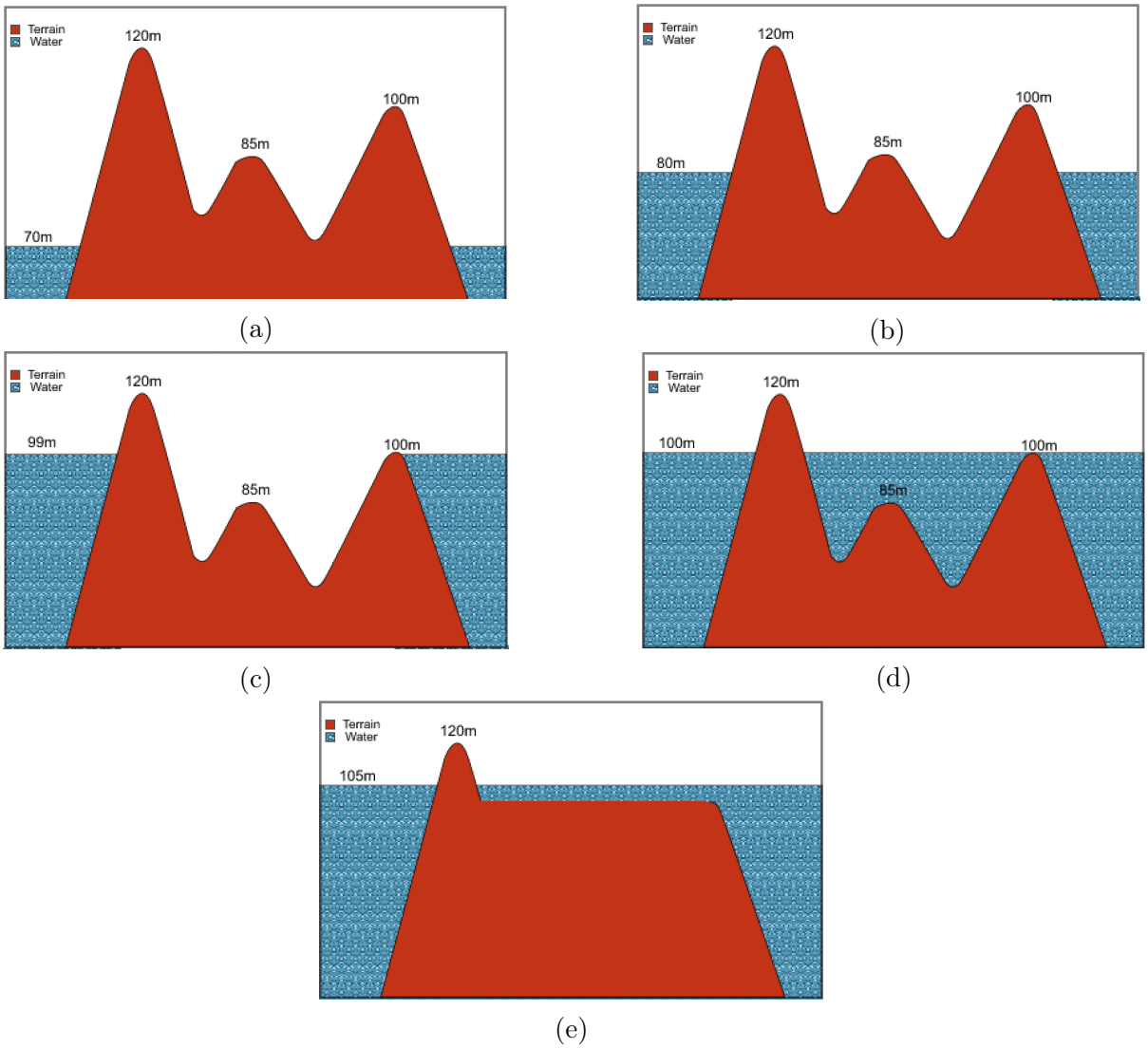


Figura 3.4: Ejemplo de inundación en el algoritmo RWFlood.

es la normal del plano A y n_B es la normal del plano B:

$$\varphi_{AB} = arcos(n_A \cdot n_B)$$

Una vez terminado este procedimiento, la red de drenaje consiste en todos los arcos en donde el par de triángulos que hacen uso de éste forman un ángulo diedro agudo (o está dentro de un intervalo predefinido).

Este algoritmo es de orden $\mathcal{O}(n)$, siendo n el número triángulos en la superficie. Esto se explica ya que se hace una pasada por cada uno de los triángulos en donde el cálculo de el ángulo diedro con todos sus vecino es de orden constante.

La relación entre la cantidad de puntos y triángulos (debido a la forma en que se generan los triángulos explicado en la sección 5.3) es:

Algoritmo 4 Flow Accumulation

```
1: procedure FlowAccumulation(terrain)
2:     for each point p in terrain do
3:         Degree[p] ← 0
4:         Flow[p] ← 1
5:         p.visited ← false
6:     for each point p in terrain do
7:         Degree[next(p)] ← Degree[next(p)] + 1
8:     for each non-visited point p in terrain do
9:         while Degree[p] = 0 do
10:            p.visited ← true
11:            if next(p) is outside the terrain then
12:                break-while
13:            Flow[next(p)] ← Flow[next(p)] + Flow(p)
14:            Degree[next(p)] ← Degree[next(p)] - 1
15:            p ← next(p)
```

$$\#Triangulos = 2 * (\#Columnas - 1) * (\#Filas - 1)$$

Es decir, si quisiéramos representar el orden términos de la cantidad de puntos en el terreno, el orden sería de $\mathcal{O}(n^2)$, siendo n el máximo entre el número de columnas y el número de filas de la grilla.

Algoritmo 5 Ángulo Diedro

```
1: procedure AnguloDiedro(terrain)
2:     for each triangle t in terrain do
3:         for each neighbor triangle n of t do
4:             setAnguloDiedro(t, n)
```

3.4. Profiling de algoritmos a paralelizar

Se decidió hacer uso de un profiler para verificar que no exista una solución más simple y con mejores resultados que la paralelización de los algoritmos. Para esto se utilizó la herramienta Callgrind[1]. Ésta permite hacer un análisis del uso de CPU, el cual muestra a cuantas instrucciones de CPU se traduce la ejecución de un procedimiento en C++. Si bien el contar la cantidad de instrucciones no necesariamente se traduce en tiempo de ejecución, ésto es una buena aproximación. Al mismo tiempo, Callgrind permite hacer un análisis del uso de memoria, entregando la cantidad de accesos a memoria de lectura y escritura, y la cantidad de fallos de cache.

Se muestran dos tipos de tablas. Aquellas relacionadas con las instrucciones de CPU exponen que porcentaje del total de las instrucciones de un álgoritmo fue dedicado a cierta actividad.

Aquellas relacionadas con el uso de memoria muestran el porcentaje de fallo de cache al realizar lecturas y escrituras en los cache de nivel 1 y en el de último nivel.

El primer algoritmo a analizar es Peucker. El principal trabajo de este algoritmo está asociado a los accesos a memoria. Dos de los tres procedimientos mostrados en la Tabla 3.1 están relacionados con accesos a memoria (accesos a elementos de un vector y marcar el punto de mayor altura). La mayor cantidad de instrucciones surge a partir del manejo de iteradores, lo que también es esperable, ya que aparte de las instrucciones de acceso a memoria, existen las instrucciones asociadas a iterar sobre cada uno de los puntos. Ninguno de estos resultados sugiere que exista una ejecución ineficiente de alguno de los procedimientos.

Como muestra la Tabla 3.2 la cantidad de fallos de cache está dentro de lo normal, por lo que concluimos que los accesos a memoria no se hacen de manera ineficiente en el algoritmo de Peucker.

Manejo de iteradores	36.29 %
Accesos a elementos de un vector	31.62 %
Marcar el punto de mayor altura	22.70 %
Otros	9.39 %

Tabla 3.1: Porcentaje de instrucciones usado en los distintos procedimientos del algoritmo de Peucker.

Cache Level 1 Read	0.82 %
Cache Level 1 Write	1.27 %
Cache Last-Level Read	0.086 %
Cache Last-Level Write	1.26 %

Tabla 3.2: Porcentaje de fallos de cache del algoritmo de Peucker.

El segundo algoritmo a analizar es Callaghan. La Tabla 3.3 muestra que la mayor cantidad de instrucciones se ejecuta en la elección del vecino hacia el cual el agua se drenará. La Tabla 3.4 muestra el análisis de ese procedimiento, donde se ve que la mayor cantidad de instrucciones ocurre al insertar elementos en un vector. Si bien éste último punto puede sugerir que se hace un uso ineficiente de los vectores, se decidió solucionar este problema junto con la paralelización del código con el fin de ver cambios importantes en el tiempo de ejecución.

Como muestra la Tabla 3.5 la cantidad de fallos de cache está dentro de lo normal, por lo que concluimos que los accesos a memoria no se hacen de manera ineficiente en el algoritmo de Callaghan.

El tercer algoritmo a analizar es RWFlood. Tal como se muestra en la Tabla 3.6, la mayor cantidad de instrucciones es ejecutada al determinar la dirección de flujo de cada punto. La Tabla 3.7 muestra el análisis de ese procedimiento, donde se puede ver que la mayor parte de las instrucciones ocurren en el cálculo de la vecindad de los puntos. Al ver el código de ese procedimiento nos encontramos con un problema similar al del algoritmo de Callaghan: la principal actividad consiste en insertar elementos en un vector. Usando el mismo argumento que en el algoritmo de Callaghan, se decidió solucionar ese problema en conjunto con la paralelización.

Elección de vecino y cálculo de agua acumulada	82.45 %
Ordenar los puntos	15.64 %
Otros	1.91 %

Tabla 3.3: Porcentaje de instrucciones usado en los distintos procedimientos del algoritmo de Callaghan.

Insertar elementos en un vector	83.91 %
Otros	16.09 %

Tabla 3.4: Porcentaje de instrucciones usado en los distintos procedimientos de la elección de vecino y cálculo de agua acumulada del algoritmo de Callaghan.

Cache Level 1 Read	0.68 %
Cache Level 1 Write	0.043 %
Cache Last-Level Read Miss	0.049 %
Cache Last-Level Write Miss	0.041 %

Tabla 3.5: Porcentaje de fallos de cache del algoritmo de Callaghan.

Como muestra la Tabla 3.8 la cantidad de fallos de cache está dentro de lo normal, por lo que concluimos que los accesos a memoria no se hacen de manera ineficiente en el algoritmo RWFlood.

Determinar dirección de flujo	86.24 %
Cálculo acumulación de agua	13.76 %

Tabla 3.6: Porcentaje de instrucciones usado en los distintos procedimientos del algoritmo RW-Flood.

Cálculo vecindad	68.93 %
Manejo de queues	14.43 %
Otros	16.64

Tabla 3.7: Insutrcciones RWFlood Determinar dirección de flujo.

El último algoritmo a analizar es RWFlood. La Tabla 3.6 muestra que la mayor cantidad de instrucciones consiste en el cálculo de el ángulo diedro entre dos triángulos. Esta operación consiste principalmente en operaciones trigonométricas, por lo que es esperable que se haga uso de una gran cantidad de instrucciones sin necesidad de que exista una ejecución ineficiente. La segunda operación con mayor cantidad de instrucciones consiste en la inserción de elementos en un vector, al igual que en los algoritmos anteriores.

Cache Level 1 Read	0.78 %
Cache Level 1 Write	0.16 %
Cache Last-Level Read	0.35 %
Cache Last-Level Write	0.14 %

Tabla 3.8: Porcentaje de fallos de cache del algoritmo RWFlood.

Como muestra la Tabla 3.10, la cantidad de fallos de cache está dentro de lo normal, por lo que concluimos que los accesos a memoria no se hacen de manera ineficiente en el algoritmo Ángulo Diedro.

Cálculo de Ángulo Diedro	49.25 %
Insertar elementos en un vector	26.82 %
Acceso a elementos de un vector	11.38 %
Otros	12.55 %

Tabla 3.9: Porcentaje de instrucciones usado en los distintos procedimientos del algoritmo Ángulo Diedro.

Cache Level 1 Read	0.78 %
Cache Level 1 Write	0.16 %
Cache Last-Level Read	0.35 %
Cache Last-Level Write	0.14 %

Tabla 3.10: Porcentaje de fallos de cache del algoritmo Ángulo Diedro.

Capítulo 4

Diseño

A continuación se muestra el diseño de la solución. Primero se explica la estrategia que se usará para paralelizar cada uno de los algoritmos. Luego se define el algoritmo que se aplicará para lograr una triangulación simplificada.

4.1. Paralelización

4.1.1. Peucker

Debido a la naturaleza del algoritmo de Peucker, su paralelización es simple. Se procesará en paralelo cada una de las ventanas de 2x2. Se escogerá el punto de mayor altura y será marcado. A simple vista pareciera ser que podría existir algún tipo de data-race debido a que distintas ventanas pueden compartir puntos, pero en la práctica eso no afecta el resultado final, ya que si es que hay un mismo punto que dos work-item quieren modificar, no importa quien lo marque primero, el resultado será el mismo. Habría problemas si es que el algoritmo también tuviera que marcar aquellos puntos que no son el máximo, pero eso no ocurre ya que en el momento que la ejecución paralela comienza, todos los puntos llegan con un valor inicial que indica que ese punto no es el máximo.

A continuación se presenta un pseudocódigo del kernel que ejecuta cada work-item:

Algoritmo 6 Peucker Kernel

```
1: procedure PeuckerKernel(window)
2:     Flag max height point(s) of window
```

Con respecto al uso de memoria, no existe una diferencia significativa en esta versión en comparación con la versión secuencial.

4.1.2. Callaghan

El principal problema que presenta este algoritmo para lograr su paralelización está relacionado con que su primer paso consiste en ordenar los puntos de mayor a menor altura, para luego procesarlos en orden. Esto pone un obstáculo para paralelizar, ya que pareciera que el resultado de procesar un punto depende de los resultados de los puntos de menor altura. Es por esto que es necesario ir un paso atrás, y ver cual es la razón por la cual es necesario ordenar antes de procesar, para así ver si efectivamente existe tal dependencia.

Lo importante del algoritmo es que la acumulación de agua final en cada punto represente toda el agua que le llegaría desde resto de los puntos. Es por esto que se ordena de mayor a menor, con el fin de que los puntos de altura menor se vean afectados por el agua de los puntos mayores. Si no fuera así y los puntos no se procesaran en orden, lo que ocurriría es que una vez que se procesa un punto solo puede ver el efecto de los puntos que tiene alrededor, y si esos puntos todavía no han sido procesados y no han recibido la acumulación de agua de puntos mayores a ellos, el punto que estamos procesando a lo más podrá tener la acumulación de esos puntos vecinos, ignorando así el efecto que tiene el agua que viene del resto de los puntos hacia este.

La diferencia se puede ver mejor con la utilizando la Tabla 4.1 donde se muestra la dirección de flujo en cada punto de un terreno al utilizar el algoritmo de Callaghan. Las flechas indican la dirección del flujo y el punto negro indica que el agua no fluye hacia algún vecino. El punto 1C es el menor del terreno. Este punto debe recibir el agua de todo el resto de los puntos ya que el flujo de agua de todo el resto de los puntos termina en este punto. Si el primer punto en procesar fuese el 1C, este solo vería el efecto de los puntos vecinos, es decir, de los puntos 1B, 2B y 2C, teniendo así una acumulación de agua final igual a 4 (1 como base y 1 desde cada uno de los puntos vecinos). Si los puntos se procesaran en orden y 1C fuera el último en ser procesado, entonces su acumulación de agua final sería de 9, una unidad por cada punto en el terreno.

	1	**2**	**3**
A	↓	↓	↙
B	↓	↙	←
C	•	←	←

Tabla 4.1: Ejemplo de flujo del algoritmo de Callaghan.

La solución encontrada va de la mano con notar que los resultados de procesar un punto no depende del resultado de procesar los puntos de mayor altura, sino que solo depende de desde cuantos puntos llega el flujo de agua a cada uno de los puntos, es decir, cuantas unidades de agua pasan a través del punto.

La paralelización del algoritmo consiste en lo siguiente: se procesa cada punto en paralelo, partiendo todos con una base de agua igual a cero. Al procesar un punto lo que se hace es sumar 1 a su acumulación de agua, ver cual es su vecino que tenga la mayor diferencia de altura y que al mismo tiempo tenga menor altura que el punto que se está procesando. Una vez encontrado ese punto, se le suma 1 a su acumulación de agua, se elige el vecino que cumpla con las condiciones ya descritas, y así sucesivamente. Se detiene cuando no hay vecinos que cumplan con las condiciones. Importante destacar que dado que este procedimiento se hace partiendo desde cada uno de los

puntos de la malla, efectivamente puede ocurrir data-race al momento de aumentar la acumulación en alguna de las celdas. Es por esto que esa operación debe ser atómica.

También notar que esta no es una buena solución desde el punto de vista teórico si es que el algoritmo se procesara en forma secuencial. Dado que el procedimiento se hace en todos los puntos y desde esos puntos se pasa por todos los puntos por donde pasaría el agua, el algoritmo es de $\mathcal{O}(n^4)$ en el peor caso, siendo n el máximo entre el número de columnas y el número de filas de la grilla.

A continuación se presenta un pseudocodigo del kernel que ejecuta cada work-item:

Algoritmo 7 Callaghan Kernel

```
1: procedure CallaghanKernel(point, maxNeighbors)
2:     while maxNeighbors.of(point) is not NULL do
3:         point.water = point.water + 1
4:         point ← maxNeighbors.of(point)
```

Recordando que para procesar un punto P se necesita saber el vecino que tenga la mayor diferencia de altura y que al mismo tiempo tenga menor altura que P, vemos que el algoritmo recibe como argumento una colección que contiene el vecino que cumple esas condiciones para cada uno de los puntos del terreno. Sin esa información, sería necesario calcular el vecino cada vez que un punto lo necesite.

La colección de vecinos se obtiene mediante otro kernel que es ejecutado previamente, una vez por punto del terreno, del cual se puede ver un pseudocodigo a continuación:

Algoritmo 8 GetSteepestDescentNeighbors Kernel

```
1: procedure GetSteepestDescentNeighborsKernel(point, maxNeighbors)
2:     maxNeighbors.of(point) ← get the neighbor of point with steepest descent
```

Con respecto al uso de memoria, este aumentó en comparación a la versión secuencial del algoritmo. Esto se debe a que se debe pre-calcular el vecino de cada uno de los puntos que cumpla las condiciones ya mencionadas, ya que se utiliza como argumento en ambos kernels.

4.1.3. RWFlood

Como se vio en 5.4.3, el algoritmo consiste en 2 partes: primero se calcula la dirección de flujo en cada punto y luego se calcula el agua acumulada en cada punto. La primera etapa toma aproximadamente el 90 % del tiempo de ejecución, y la segunda solo el 10 %. Es por esto que si queremos ver diferencias importantes en el tiempo de ejecución, es fundamental que la primera etapa sea paralelizada.

Una primera aproximación a la paralelización sería procesar todos los puntos del terreno en paralelo. El problema es que si intentáramos procesar cada punto de forma independiente se hace imposible determinar su dirección de flujo, ya que por definición del algoritmo, esta depende

directamente de cual de sus vecinos se procesa primero. Es imposible determinar el flujo a partir de algún otro dato.

Por otro lado, lo que se puede hacer es procesar todos los puntos en una cola en paralelo, ya que el resultado es independiente del orden en que se procesan los puntos dentro de una misma cola. Por lo tanto, la estrategia para paralelizar es la siguiente: se procesan los puntos dentro de una cola, una vez procesados se verifica que la cola esté vacía (ya que pueden haber nuevos elementos), si está vacía, se sigue a la cola de la siguiente altura, si no está vacía, se procesan en paralelo los puntos que se hayan agregado en la cola. Así sucesivamente hasta que todas las colas estén vacías.

Esto no hace grandes cambios en el algoritmo, y su rendimiento dependerá de la cantidad de puntos que hayan en cada una de las cola al procesarlas (mientras más puntos se procesen en paralelo mejor será el rendimiento).

A continuación se presenta un pseudocódigo del kernel que ejecuta cada work-item. Es importante destacar que siempre se ejecuta un work-item por cada punto en el terreno, pero aquellos que realmente hacen algún tipo de procesamiento son los asociados a un punto contenido en la cola que se está procesando. Ésto se puede ver en la línea 2, en donde se verifica que el punto asociado al work-item esté contenido en la cola que se está procesando:

Algoritmo 9 RWFlood Kernel

```
1: procedure RWFloodKernel(point, queues, currentHeight)
2:     if point is in queues[currentHeight] then
3:         queues[currentHeight].remove(point)
4:         for each point n neighbor of point where n.dir = NULL do
5:             n.dir ← point
6:             if n.elev < z then
7:                 n.elev ← z
8:             queues[n.elev].insert(n)
```

Con respecto al uso de memoria, no existe una diferencia significativa en esta versión en comparación con la versión secuencial.

4.1.4. Ángulo Diedro

Debido a la naturaleza del algoritmo de Ángulo Diedro, su paralelización es simple. Se procesará en paralelo cada uno de los triángulos. Se calculará el ángulo diedro entre cada triangulo y sus vecinos. A simple vista pareciera ser que podría existir algún tipo de data-race debido a que dos work-item pueden estar calculando el ángulo diedro entre los mismos dos triángulos, pero debido a que ambos work-item escribirán el mismo resultado, no existe tal problema.

A continuación se presenta un pseudocodigo del kernel que ejecuta cada work-item:

Otra estrategia para lograr la paralelización consiste en iterar sobre los arcos (ya que en Runnel cada arco sabe que triángulos lo contienen), pero se decidió iterar sobre los triángulos con el fin de seguir la misma estrategia usada en la versión secuencial del algoritmo.

Algoritmo 10 Ángulo Diedro Kernel

```
1: procedure AnguloDiedroKernel(triangle)
2:     for each neighbor triangle n of triangle do
3:         setAnguloDiedro(triangle, p)
```

Con respecto al uso de memoria, no existe una diferencia significativa en esta versión en comparación con la versión secuencial.

4.2. Triangulación Simplificada

En primer lugar se debió elegir algún criterio para eliminar elementos del terreno para así lograr una triangulación simplificada. El criterio elegido tiene relación con el área del los triángulos: si el área de un triángulo es menor que algún número predefinido, entonces el triángulo debe ser eliminado (siempre y cuando ésto no altere las características topográficas del terreno). Es importante destacar que no hay ningún argumento de peso detrás de esta elección, no se ha medido las consecuencias positivas y/o negativas que este criterio podría tener sobre el terreno. El uso de este criterio es con fines experimentales.

Para lograr la simplificación de la triangulación el procedimiento es el siguiente: se iterará sobre todos los triángulos del terreno, se verificará que cumpla el criterio para ser eliminado. Si es que lo cumple, el triangulo se debe intentar eliminar.

En segundo lugar se necesitaba alguna estrategia para deshacerse de un triángulo que se quiere eliminar. Para esto se usó el algoritmo conocido como Vertex-Deletion [12].

El algoritmo de Vertex-Deletion consiste en lo siguiente: Se identifica el arco más corto del triángulo, se elimina uno de los puntos que lo forman junto con los triángulos formados por este arco. Luego, se modifica el resto de los triángulos que hacían uso del punto eliminado, para que el punto eliminado sea reemplazado por el otro punto del arco más corto. Una demostración gráfica de esta transformación puede ser visualizada en la Figura 4.1, en donde el punto a eliminar es el punto A del arco AB.

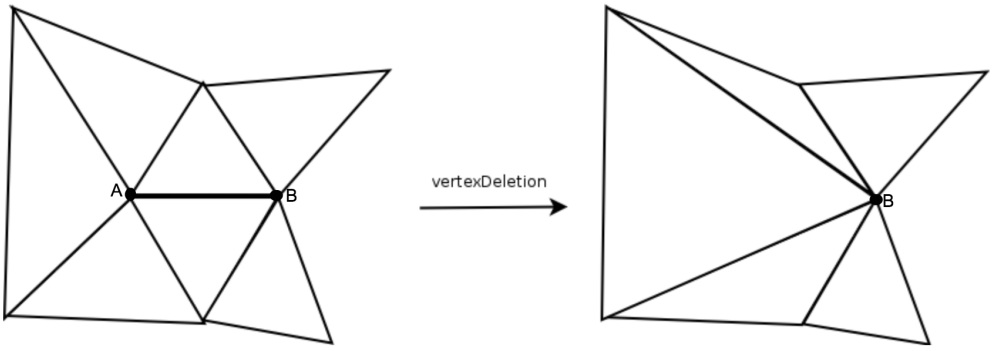


Figura 4.1: Vertex-Deletion.

Un pseudocódigo de Vertex-Deletion se puede ver en el Algoritmo 11. También se incluye una

representación gráfica de las variables usadas en este algoritmo en la Figura 4.2. Notar que las variables CaraBorrar1 y CaraBorrar2 son aquellos triángulos que se eliminarán ya que el arco que comparten será borrado, CaraRemplazar1 y CaraRemplazar2 son aquellos triángulos que contienen el vértice que será eliminado y que comparten uno de sus arcos con alguno de los triángulos que serán eliminados. Por último, CaraModificar corresponde al triángulo que contiene el vértice que se eliminará pero que no comparté ningún arco con los triángulos que serán eliminados. En la práctica, más de un triángulo podría cumplir las propiedades de CaraModificar, pero con el fin de facilitar la comprensión del algoritmo, el ejemplo asume que solamente un triángulo cumple estas propiedades.

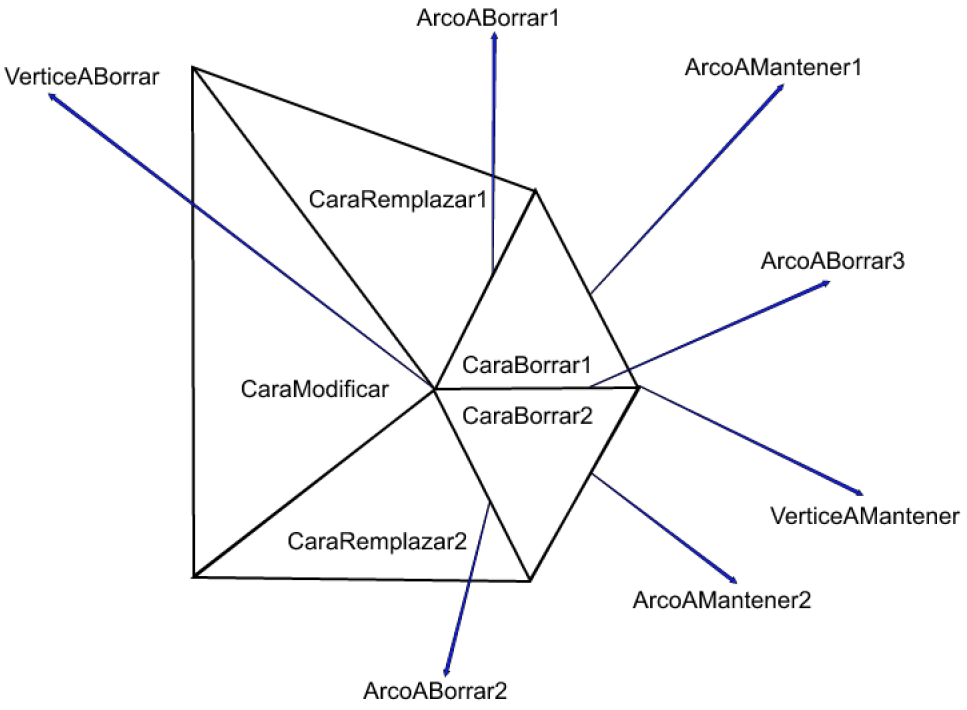


Figura 4.2: Variables usadas en Vertex-Deletion.

Por último, importante destacar que usando el criterio del área del triángulo se pueden obtener resultados interesantes. Debido a la forma en que se realiza la triangulación del terreno, en donde todos los triángulos son iguales al considerar sus coordenadas x e y, el área de los triángulos solamente varía debido a la coordenada z de cada uno de sus puntos. Si todos los puntos de un mismo triángulo tienen la misma coordenada z, entonces será un triangulo plano, el cual tendrá la mínima área posible de un triángulo en el terreno. Si usamos como criterio que los triángulos a eliminar son aquellos que tengan área igual (o similar) a la menor área de todos los triángulos del terreno, estaríamos tratando de deshacernos de triángulos innecesarios en sectores planos, utilizando menos triángulos para representar un mismo sector.

Algoritmo 11 Vertex-Deletion

```
1: procedure VertexDeletion(Terreno, VerticeABorrar, VerticeAMantener,
   CaraBorrar1, CaraBorrar2, ArcoAMantener1, ArcoAMantener2, CaraReemplazar1,
   CaraReemplazar2, CaraModificar, ArcoABorrar1, ArcoABorrar2, ArcoABorrar3)
2:     Terreno.removeArco(ArcoABorrar1)
3:     Terreno.removeArco(ArcoABorrar2)
4:     Terreno.removeArco(ArcoABorrar3)
5:     Terreno.removeCara(CaraBorrar1)
6:     Terreno.removeCara(CaraBorrar2)
7:     for each cara c in {CaraModificar, CaraReemplazar1, CaraReemplazar2} do
8:         if c.containsArco(ArcoABorrar1) then
9:             c.replaceArco(ArcoABorrar1, ArcoAMantener1)
10:        if c.containsArco(ArcoABorrar2) then
11:            c.replaceArco(ArcoABorrar2, ArcoAMantener2)
12:        for each arco a in c do
13:            if a.contains(VerticeABorrar) then
14:                a.replaceVertice(VerticeABorrar, VerticeAMantener)
```

Capítulo 5

Implementación

En este capítulo se detalla la implementación de los algoritmos propuestos. En la sección 5.1 se incluye la implementación en paralelo del algoritmo de Callaghan usando OpenCL. La sección 5.2 consiste en la implementación del algoritmo Vertex-Deletion.

5.1. Paralelización

Antes de ejecutar un kernel en OpenCL es necesario realizar una serie de operaciones relacionadas principalmente con el paso de memoria de la CPU hacia/desde el device. Como referencia, se describe primero la preparación para la ejeción de los kernels de Callaghan, y luego se explican los kernels del algoritmo mismo. Los kernels de los algoritmos de Peucker, RWFlood y Ángulo Diedro se encuentran en el capítulo 8 (Anexo).

La Figura 5.1 muestra la creación del contexto. En un contexto pueden existir varios devices. En nuestro caso solo utilizaremos uno. También muestra la creación de una command queue, la cual es una queue que recibe lo que se ejecutará en el device. Las instrucciones se ejecutarán en el mismo orden en que se insertaron en la queue. Por último, se carga el archivo que contiene los kernels que se desean ocupar.

```
113     cl_context context = clCreateContext(NULL, 1, {device}, NULL, NULL, &error);
114     checkError(error, "Creating context");
115
116     cl_command_queue commandQueue = clCreateCommandQueue(context, deviceId, 0, &error);
117     checkError(error, "Creating command queue");
118
119     const char* kernelSource = loadKernel("callaghan.cl");
120     cl_program callaghanProgram = clCreateProgramWithSource(context, 1, &kernelSource, NULL, &error);
121     checkError(error, "Creating program");
122     free((char*)kernelSource);
```

Figura 5.1: Configuración inicial para la ejecución de un kernel.

En la Figura 5.2 se ve cómo se calcula el tamaño de los datos de entrada y de salida, cómo se obtienen los datos de entrada del algoritmo y cómo se pide memoria para almacenar los resultados en memoria a la que el host tiene acceso. Los datos de entrada y salida consisten en la altura de

cada uno de los puntos del terreno (coordz), el id del punto vecino hacia donde el agua de un punto drenará (maxNeighbours) y la acumulación de agua de cada punto (waterValues).

```
138     int coordszMemorySize = (terrain->width)*(terrain->height)*sizeof(float);
139     int maxNeighboursMemorySize = (terrain->width)*(terrain->height)*sizeof(int);
140     int waterValuesMemorySize = (terrain->width)*(terrain->height)*sizeof(int);
141
142     float* coordsz = terrain->pointsCoordZ.data();
143     int* waterValues = (int*)malloc(waterValuesMemorySize);
144     int* maxNeighbours = (int*)malloc(waterValuesMemorySize);
```

Figura 5.2: Cálculo de memoria de entrada y salida de los kernels.

En la Figura 5.3 se ve como se pide memoria en el device en el cual los kernels serán ejecutados. Es necesario pedir memoria para todo dato de entrada y todo dato de salida. Se puede especificar si los datos pedidos son solo de escritura, solo lectura o ambas, con el fin de que se puedan realizar algunas optimizaciones.

```
152     cl_mem d_coordsz = clCreateBuffer(context, CL_MEM_READ_WRITE, coordszMemorySize, NULL, &error);
153     checkError(error, "Allocating memory in the device\n");
154
155     cl_mem d_maxNeighbours = clCreateBuffer(context, CL_MEM_READ_WRITE, maxNeighboursMemorySize, NULL, &error);
156     checkError(error, "Allocating memory in the device\n");
157     cl_mem d_waterValues = clCreateBuffer(context, CL_MEM_READ_WRITE, waterValuesMemorySize, NULL, &error);
158     checkError(error, "Allocating memory in the device\n");
159
160     cl_mem d_width = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int), NULL, &error);
161     checkError(error, "Allocating memory in the device\n");
162     cl_mem d_height = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int), NULL, &error);
163     checkError(error, "Allocating memory in the device\n");
164     cl_mem d_deltaWater = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float), NULL, &error);
165     checkError(error, "Allocating memory in the device\n");
```

Figura 5.3: Creación de buffers en la memoria del device.

En la Figura 5.4 podemos ver como se escriben los datos de entrada desde la memoria del host hacia la memoria del device. Es acá en donde existe un cuello de botella, ya que este procedimiento es lento. También es importante destacar que si el device y el host son la misma unidad de procesamiento, existen opciones para omitir este paso. El problema de hacer esto es que ese procedimiento no funciona si el device no es igual al host, por lo que se optó por siempre escribir la memoria desde el host hacia el device, cualquiera sea el device, con el fin de mantener la portabilidad del código entre devices.

```
168     error = clEnqueueWriteBuffer(commandQueue, d_coordsz, CL_TRUE, 0, coordszMemorySize, coordsz, 0, 0, NULL);
169     checkError(error, "Writing from cpu to device\n");
170     error = clEnqueueWriteBuffer(commandQueue, d_waterValues, CL_TRUE, 0, waterValuesMemorySize, waterValues, 0, 0, NULL);
171     checkError(error, "Writing from cpu to device\n");
172     error = clEnqueueWriteBuffer(commandQueue, d_width, CL_TRUE, 0, sizeof(int), &(terrain->width), 0, 0, NULL);
173     checkError(error, "Writing from cpu to device\n");
174     error = clEnqueueWriteBuffer(commandQueue, d_height, CL_TRUE, 0, sizeof(int), &(terrain->height), 0, 0, NULL);
175     checkError(error, "Writing from cpu to device\n");
176     error = clEnqueueWriteBuffer(commandQueue, d_deltaWater, CL_TRUE, 0, sizeof(float), &deltaWater, 0, 0, NULL);
177     checkError(error, "Writing from cpu to device\n");
```

Figura 5.4: Paso de datos desde el host al device.

En la Figura 5.5 se ve como se crean los kernels y como se asignan sus argumentos. El output de los kernels deberán ser pasados como argumentos ya que es la única forma de recuperarlo desde el host.

```
181     cl_kernel setMaxNeighbourKernel = clCreateKernel(callaghanProgram, "setMaxNeighbour", &error);
182     checkError(error, "Creating setMaxNeighbour Kernel");
183     error = clSetKernelArg(setMaxNeighbourKernel, 0, sizeof(cl_mem), (void*)&d_coordsz);
184     checkError(error, "Setting first argument of the setMaxNeighour kernel");
185     error = clSetKernelArg(setMaxNeighbourKernel, 1, sizeof(cl_mem), (void*)&d_maxNeighbours);
186     checkError(error, "Setting first argument of the setMaxNeighour kernel");
187     error = clSetKernelArg(setMaxNeighbourKernel, 2, sizeof(cl_mem), (void*)&d_width);
188     checkError(error, "Setting third argument of the setMaxNeighour kernel");
189     error = clSetKernelArg(setMaxNeighbourKernel, 3, sizeof(cl_mem), (void*)&d_height);
190     checkError(error, "Setting fourth argument of the setMaxNeighour kernel");
191     error = clSetKernelArg(setMaxNeighbourKernel, 4, sizeof(cl_mem), (void*)&d_deltaWater);
192     checkError(error, "Setting fifth argument of the setMaxNeighbour kernel");
193
194     cl_kernel setWaterPathKernel = clCreateKernel(callaghanProgram, "setWaterPath", &error);
195     checkError(error, "Creating setWaterPath Kernel");
196     error = clSetKernelArg(setWaterPathKernel, 0, sizeof(cl_mem), (void*)&d_maxNeighbours);
197     checkError(error, "Setting first argument of the setWaterPath kernel");
198     error = clSetKernelArg(setWaterPathKernel, 1, sizeof(cl_mem), (void*)&d_waterValues);
199     checkError(error, "Setting first argument of the setWaterPath kernel");
200     error = clSetKernelArg(setWaterPathKernel, 2, sizeof(cl_mem), (void*)&d_width);
201     checkError(error, "Setting third argument of the setWaterPath kernel");
202     error = clSetKernelArg(setWaterPathKernel, 3, sizeof(cl_mem), (void*)&d_height);
203     checkError(error, "Setting fourth argument of the setWaterPath kernel");
```

Figura 5.5: Creación de kernels y asignación de sus argumentos.

En la Figura 5.6 se muestra cómo se calcula el tamaño del work-group que se usará. Dependiendo del device y de la naturaleza del kernel, OpenCL recomendará que el tamaño sea múltiplo de un número específico. Es necesario realizar pruebas para encontrar el múltiplo adecuado. En este caso se eligió el menor múltiplo posible. A partir del tamaño del work-group elegido, se calcula el tamaño del global del problema. Esta misma configuración se utilizó en cada uno de los kernels implementados y para cada uno de estos el número recomendado por OpenCL tiene como fin maximizar el uso de todos los cores del device.

```
206     size_t localWorkSizeX;
207     clGetKernelWorkGroupInfo(setWaterPathKernel, deviceId,
208                              CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE, sizeof(size_t), &localWorkSizeX, NULL);
209     int globalWorkSizeX = terrain->height*terrain->width;
210     globalWorkSizeX = globalWorkSizeX - (globalWorkSizeX%localWorkSizeX) + localWorkSizeX;
211     const size_t globalWorkSize [] = { globalWorkSizeX, 0, 0 };
212     const size_t localWorkSize [] = { localWorkSizeX, 0, 0 };
```

Figura 5.6: Configuración del tamaño del work-group.

En la Figura 5.7 Se muestra la ejecución de los kernels y como se leen los resultados desde el device hacia el host. También es posible ejecutar kernels en paralelo, siempre y cuando uno no dependa del otro.

```
213     error = clEnqueueNDRangeKernel(commandQueue, setMaxNeighbourKernel, 1, NULL, globalWorkSize, localWorkSize, 0, NULL,  NULL);
214     checkError(error, "Running setMaxNeighbour kernel\n");
215
216     error = clEnqueueReadBuffer(commandQueue, d_maxNeighbours, CL_TRUE, 0, maxNeighboursMemorySize, maxNeighbours, 0, NULL, NULL);
217     checkError(error, "Reading from device to cpu");
218
219     error = clEnqueueNDRangeKernel(commandQueue, setWaterPathKernel, 1, NULL, globalWorkSize, localWorkSize, 0, NULL,  NULL);
220     checkError(error, "Running setWaterPath kernel\n");
221
222     error = clEnqueueReadBuffer(commandQueue, d_waterValues, CL_TRUE, 0, waterValuesMemorySize, waterValues, 0, NULL, NULL);
223     checkError(error, "Reading from device to cpu");
```

Figura 5.7: Ejecución de kernels y lectura de resultados.

La Figura 5.8 se muestra como se libera la memoria que se pidió en el device.

La Figura 5.9 muestra el kernel encargado de la elección del punto vecino que tenga la mayor

```
232     error = clReleaseMemObject(d_coordsz);
233     checkError(error, "Releasing memory from device");
234     error = clReleaseMemObject(d_waterValues);
235     checkError(error, "Releasing memory from device");
236     error = clReleaseMemObject(d_maxNeighbours);
237     checkError(error, "Releasing memory from device");
238     error = clReleaseMemObject(d_width);
239     checkError(error, "Releasing memory from device");
240     error = clReleaseMemObject(d_height);
241     checkError(error, "Releasing memory from device");
242     error = clReleaseMemObject(d_deltaWater);
243     checkError(error, "Releasing memory from device");
```

Figura 5.8: Liberación de la memoria de los buffers del device.

diferencia de altura con el punto que estamos procesando y que al mismo tiempo tenga menor altura que éste. Dado que en el device no se pueden manejar las estructuras de datos complejas que existen en el host (ya que pueden tener direcciones de memoria distinta, por lo que los punteros no funcionan), no es posible pasar como argumento la estructura de datos de un punto, por lo que se pasa solo el dato necesario de cada punto, en este caso, su altura.

Las alturas se pasan en un arreglo, en donde el índice del arreglo es equivalente al índice del punto en el arreglo de puntos en el host. El output usa esta misma estrategia.

```
99  __kernel void setMaxNeighbour(__global float* coordsz, __global int* maxNeighbours,
100                             __global const int* pwidth, __global const int* pheight,
101                             __global const float* pdeltaWater) {
102
103         int id = get_global_id(0);
104         int width = *pwidth;
105         int height = *pheight;
106         float deltaWater = *pdeltaWater;
107
108         if(id < width*height) {
109                 int neighbourPosition[8];
110                 getNeighboursPositions(neighbourPosition);
111
112                 float maxHeightDifference = -1;
113                 int idMaxNeighbour = -1;
114
115                 for (int i = 0; i < 8; ++i)
116                 {
117                         if (neighbourPosition[i]>=0)
118                         {
119                                 float currentPointHeight = coordsz[id];
120                                 float neighbourPointHeight = coordsz[neighbourPosition[i]];
121                                 if (neighbourPointHeight < currentPointHeight + deltaWater) {
122                                         float heightDifference = currentPointHeight - neighbourPointHeight;
123                                         if (maxHeightDifference < heightDifference)
124                                         {
125                                                 maxHeightDifference=heightDifference;
126                                                 idMaxNeighbour = neighbourPosition[i];
127                                         }
128                                 }
129                         }
130                 }
131                 maxNeighbours[id] = idMaxNeighbour;
132         }
133 }
```

Figura 5.9: Primer kernel del algoritmo de Callaghan.

En la Figura 5.10 se muestra el kernel encargado de obtener la acumulación de agua en cada punto. Nuevamente el índice de los arreglos de entrada y salida corresponden al índice del punto en el host. También podemos ver que en este caso fue necesario utilizar un lock al escribir los datos ya que pueden haber más de un work-item alterando la acumulación de agua para un mismo punto.

```
84  _kernel void setWaterPath(__global int* maxNeighbors, __global int* waterValues,
85                            __global const int*  pwidth, __global const int*  pheight){
86
87          int id = get_global_id(0);
88          int width = *pwidth;
89          int height = *pheight;
90
91          if(id < width*height) {
92                  while(id != -1) {
93                          atomic_add(&(waterValues[id]), 1);
94                          id = maxNeighbors[id];
95                  }
96          }
97  }
```

Figura 5.10: Segundo kernel del algoritmo de Callaghan.

5.2. Triangulación

La Figura 5.11 muestra el algoritmo de triangulación simplificada. Se puede ver que a medida que pueda existir algún arco que se pueda colapsar el algoritmo se vuelve a realizar. Esto se hace ya que una vez que el algoritmo se ejecutó, este modificó otros triángulos que ahora pueden cumplir la condición, por lo que es necesario iterar hasta que no existan triángulos que hayan sido modificados.

También se excluyen aquellos triángulos que si bien cumplían la condición para ser eliminados, no fue posible eliminarlos. La condición para intentar eliminar el triángulo consiste en que éste tenga un área menor al 102 % del área mínima de todos los triángulos del terreno.

Existe otra condición que consiste en que el triángulo no haya sido eliminado del terreno. Esta condición pareciera no tener sentido, pero es necesaria. Lo que ocurre es que idealmente iteraríamos sobre la estructura del terreno que contiene los triángulos. El problema es que queremos modificar esa estructura ya que vamos a eliminar triángulos, y el eliminar elementos de una estructura sobre la cual estamos iterando trae problemas. Por lo tanto se crea una copia de esa estructura del terreno, por lo que iteraremos siempre sobre el total de triángulos pero siempre chequeando que ese triángulo pertenezca al terreno.

```
62  void BuilderTerrain::runSimplifedTriangulation(Terrain* ter){
63
64      bool thereMightBeAnEdgeLeftToCollapse = true;
65      std::unordered_set <runnel::Triangle*> excluded;
66
67      std::cout << "Amount of triangles before simplification: " << ter->struct_triangle.size() <<  std::endl;
68      std::cout << "Average triangles area: " << ter->trianglesAverageArea << std::endl;
69      std::cout << "Min triangles area: " << ter->trianglesMinArea << std::endl;
70
71      std::unordered_set <runnel::Triangle*> trianglesCopy = ter->struct_triangle;
72      while(thereMightBeAnEdgeLeftToCollapse) {
73          thereMightBeAnEdgeLeftToCollapse = false;
74          for(runnel::Triangle* triangle : trianglesCopy) {
75              if(ter->struct_triangle.find(triangle) ==  ter->struct_triangle.end()) continue;
76              //This will simplify the triangulation on the flat zones of the terrain
77              if(excluded.find(triangle) == excluded.end() && triangle->getArea() < ter->trianglesMinArea*1.02){
78                  if(simplifyTriangle(triangle, ter)) thereMightBeAnEdgeLeftToCollapse = true;
79                  else excluded.insert(triangle);
80              }
81          }
82      }
83      std::cout << "Amount of triangles after simplification: " << ter->struct_triangle.size() <<  std::endl;
84  }
```

Figura 5.11: Triangulación simplificada.

La Figura 5.12 muestra que se elige el arco más corto para ser eliminado.

```
86  bool BuilderTerrain::simplifyTriangle(runnel::Triangle* triangle, Terrain* ter){
87      assert(("Triangle to simplify was not part of the terrain",ter->struct_triangle.find(triangle) != ter->struct_triangle.end()));
88      runnel::Edge* shortestEdge = triangle->getShortestEdge();
89      bool edgeWasCollapsed = edgeCollapse(shortestEdge, ter);
90      return edgeWasCollapsed;
91  }
```

Figura 5.12: Eliminar un triángulo deshaciéndose del arco más corto.

La Figura 5.13 muestra la elección del punto que se eliminará. El punto a eliminar no puede pertenecer al borde el terreno.

```
93  bool BuilderTerrain::edgeCollapse(runnel::Edge* edge, Terrain* ter){
94
95      if(ter->isBorderEdge(edge)) {
96          return false;
97      }
98
99      runnel::Point* pointToDelete = edge->point1;
100     runnel::Point* pointToKeep = edge->point2;
101
102     if(ter->isBorderPoint(pointToDelete)) {
103         runnel::Point* aux = pointToKeep;
104         pointToKeep = pointToDelete;
105         pointToDelete = aux;
106     }
107
108     bool pointWasDeleted = pointDeletion(ter, edge, pointToDelete);
109     return pointWasDeleted;
110 }
```

Figura 5.13: Antes de eliminar el arco, verifica que no se vaya a eliminar un punto del borde.

La Figura 5.14 muestra ciertas condiciones que no permiten que un punto sea eliminado. En caso de no cumplir esas condiciones, se procede a eliminar el punto. Las primeras dos condiciones están relacionadas a que si el punto es eliminado, existirán dos triángulos que tendrán las mismas coordenadas, es decir, quedarán sobrepuestos. La última verifica que ninguno de los triángulos que se van a modificar tenga una rotación de más de 90 grados.

Las Figuras 5.15 y 5.16 muestran el algoritmo conocido como Vertex-Deletion el cual ya fue explicado en el capítulo de Diseño en la sección Triangulación Simplificada.

```
bool BuilderTerrain::pointDeletion(Terrain* ter, runnel::Edge* edge, runnel::Point* pointToDelete){

    assert(("Point to delete is not part of the edge to delete", edge->point1 == pointToDelete || edge->point2 == pointToDelete));
    runnel::Point* pointToKeep = edge->point1 == pointToDelete ? edge->point2 : edge->point1;

    int intersectingPoints = getIntersectingNeighbourPoints(pointToKeep, pointToDelete, ter);
    if(intersectingPoints > 4) {
        return false;
    }

    runnel::Triangle* triangleToDelete1 = edge->neighbour_triangle[0];
    runnel::Triangle* triangleToDelete2 = edge->neighbour_triangle[1];

    runnel::Triangle* triangleToDelete1NeighbourToModify = getTriangleOpposedToPoint(triangleToDelete1, pointToKeep);
    runnel::Triangle* triangleToDelete1NeighbourToKeep = getTriangleOpposedToPoint(triangleToDelete1, pointToDelete);
    runnel::Triangle* triangleToDelete2NeighbourToModify = getTriangleOpposedToPoint(triangleToDelete2, pointToKeep);
    runnel::Triangle* triangleToDelete2NeighbourToKeep = getTriangleOpposedToPoint(triangleToDelete2, pointToDelete);
    if(areNeighbour(triangleToDelete1NeighbourToModify, triangleToDelete1NeighbourToKeep) ||
            areNeighbour(triangleToDelete2NeighbourToModify, triangleToDelete2NeighbourToKeep)) {
        //They would be the same triangle if the edge was deleted
        return false;
    }

    std::vector<runnel::Triangle*> trianglesToModify = getTrianglesToModify(edge, pointToDelete, ter);
    for(runnel::Triangle* triangle : trianglesToModify) {
        if(triangleWouldFlipIfModified(pointToDelete, triangle, pointToDelete->coord)) return false;
    }

    removeEdgeFromTerrain(ter, edge, pointToKeep, pointToDelete);
    return true;

}
```

Figura 5.14: Elimina un punto siempre y cuando cumpla ciertas condiciones.

```
void BuilderTerrain::removeEdgeFromTerrain(Terrain* ter, runnel::Edge* edge, runnel::Point* pointToKeep, runnel::Point* pointToDelete) {
    assert(("Point to delete is not part of the edge to delete", edge->point1 == pointToDelete || edge->point2 == pointToDelete));
    assert(("Point to delete is not part of the edge to delete", edge->point1 == pointToKeep || edge->point2 == pointToKeep));
    runnel::Triangle* triangleToDelete1 = edge->neighbour_triangle[0];
    runnel::Triangle* triangleToDelete2 = edge->neighbour_triangle[1];

    runnel::Edge* edgeToDelete1 = getEdgeOpposedToPoint(triangleToDelete1, pointToKeep);
    runnel::Edge* edgeToDelete2 = getEdgeOpposedToPoint(triangleToDelete2, pointToKeep);

    runnel::Triangle* triangleToDelete1NeighbourToModify = getTriangleOpposedToPoint(triangleToDelete1, pointToKeep);
    runnel::Triangle* triangleToDelete2NeighbourToModify = getTriangleOpposedToPoint(triangleToDelete2, pointToKeep);

    //Delete edges from terrain
    ter->struct_edge.erase(edgeToDelete1);
    ter->struct_edge.erase(edgeToDelete2);
    ter->struct_edge.erase(edge);
    //Delete triangles from terrain
    ter->struct_triangle.erase(triangleToDelete1);
    ter->struct_triangle.erase(triangleToDelete2);
    for(runnel::Point* point : triangleToDelete1->points) {
        ter->trianglesContainingPoint[point].erase(triangleToDelete1);
    }
    for(runnel::Point* point : triangleToDelete2->points) {
        ter->trianglesContainingPoint[point].erase(triangleToDelete2);
    }

    //Update edgesToKeep neighbour triangles
    runnel::Edge* edgeToKeep1 = getEdgeOpposedToPoint(triangleToDelete1, pointToDelete);
    runnel::Edge* edgeToKeep2 = getEdgeOpposedToPoint(triangleToDelete2, pointToDelete);
    edgeToKeep1->neighbour_triangle.erase(std::remove(edgeToKeep1->neighbour_triangle.begin(), edgeToKeep1->neighbour_triangle.end(), triangleToDelete1));
    edgeToKeep2->neighbour_triangle.erase(std::remove(edgeToKeep2->neighbour_triangle.begin(), edgeToKeep2->neighbour_triangle.end(), triangleToDelete2));
    edgeToKeep1->neighbour_triangle.push_back(triangleToDelete1NeighbourToModify);
    edgeToKeep2->neighbour_triangle.push_back(triangleToDelete2NeighbourToModify);

```

Figura 5.15: Primera parte de Vertex-Deletion.

```
179     //Update points and edges of triangles to modify
180     std::vector<runnel::Triangle*> trianglesToModify = getTrianglesToModify(edge, pointToDelete, ter);
181     for(runnel::Triangle* triangle : trianglesToModify) {
182         std::replace(triangle->points.begin(), triangle->points.end(), pointToDelete, pointToKeep);
183         ter->trianglesContainingPoint[pointToKeep].insert(triangle);
184         std::replace(triangle->edges.begin(), triangle->edges.end(), edgeToDelete1, edgeToKeep1);
185         std::replace(triangle->edges.begin(), triangle->edges.end(), edgeToDelete2, edgeToKeep2);
186         for(runnel::Edge* edge : triangle->edges) {
187             if(edge->point1 == pointToDelete) edge->point1 = pointToKeep;
188             if(edge->point2 == pointToDelete) edge->point2= pointToKeep;
189         }
190         triangle->refresh();
191     }
192     delete triangleToDelete1;
193     delete triangleToDelete2;
194     delete edge;
195     delete edgeToDelete1;
196     delete edgeToDelete2;
197 }
```

Figura 5.16: Segunda parte de Vertex-Deletion.

Capítulo 6

Resultados

En el siguiente capítulo se muestran los resultados de rendimiento de los algoritmos implementados. Primero se explica como se validó la solución. Luego se compara el rendimiento en distintas unidades de procesamiento de los algoritmos paralelizados con respecto a la versión secuencial. Los resultados obtenidos al utilizar una triangulación simplificada son comparados con los de la triangulación regular.

Todas las pruebas se hicieron sobre un mismo terreno que se puede ver en la Figura 6.1. La elección de este terreno se debe a que también ha sido usado en trabajos pasados [10] [6], por lo que se considera un buen terreno de prueba. Al mismo tiempo Runnel permite cambiar la precisión del terreno modificando la cantidad de puntos que representan el terreno, por lo que solamente un terreno de prueba es suficiente para tener terrenos de diferentes tamaño.

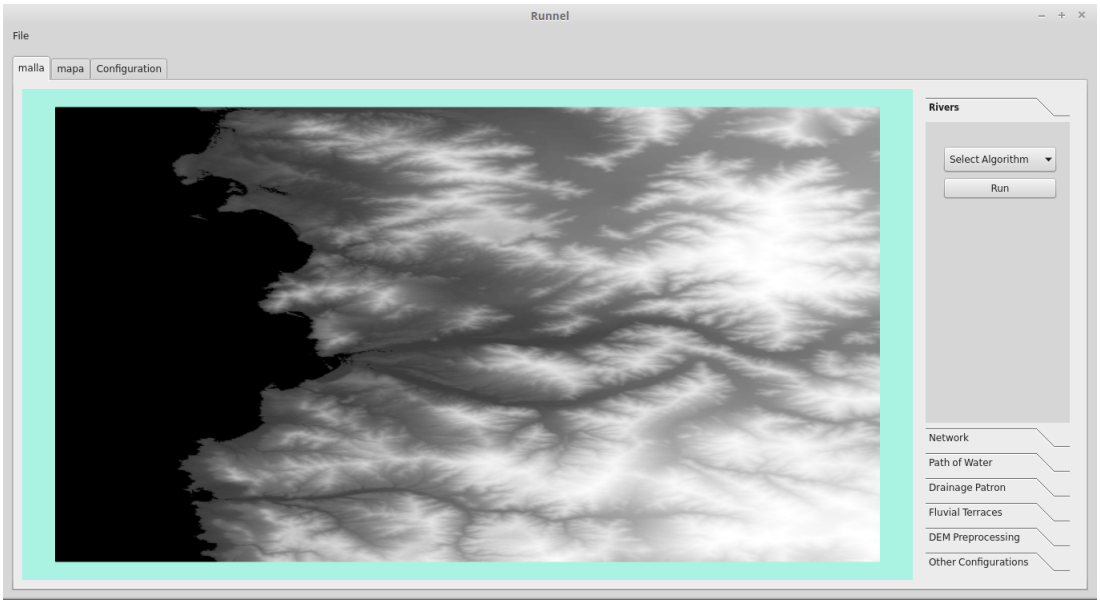


Figura 6.1: Visualización del terreno de prueba.

6.1. Validación de la implementación

Los algoritmos paralelizados fueron validados comparando sus resultados con los resultados de su versión secuencial. Dependiendo del caso, se decidió aceptar que sus resultados no fuesen exactamente igual, siempre y cuando se conociera el origen de esta diferencia y que ésta no tuviera un gran impacto en el resultado.

Para los algoritmos de Peucker y Callaghan se verificó que las redes de drenaje obtenidas con sus versiones paralelas fuesen exactamente iguales a las de sus versiones secuenciales. En cambio para los algoritmos RWFlood y Ángulo Diedro se aceptó cierto margen de error, ya que se esperaban variaciones respecto a sus versiones secuenciales.

El algoritmo de triangulación simplificada fue implementado de forma experimental, por lo que su correcta validación deberá ser realizada en un trabajo futuro, analizando de forma detallada las consecuencias que trae sobre el terreno la ejecución de este. Es por esto que en este trabajo se decidió hacer una comparación visual de la ejecución de distintos algoritmos (Ángulo Diedro, Normal Vector Similarity y Peucker) una vez utilizada la triangulación simplificada, con el fin de poder comparar sus resultados con la ejecución del mismo algoritmo pero usando la triangulación regular original.

Por último, es importante destdacar que tanto el testing como el debugging en OpenCL son complejos. Esto se debe a que, sumado a lo complejo que son estas actividades en algoritmos paralelos de por si, no existen herramientas para OpenCL que faciliten este trabajo. En consecuencia, la principal herramienta para testear y debuggear el código consiste en insertar manualmente ciertas instrucciones en el código con el fin de obtener información que sea útil en el tiempo de ejecución, lo que conlleva a errores y otro tipo de complicaciones. Es por esto que el tiempo dedicado a estas actividades es mayor de lo común.

6.2. Paralelización

Todos los algoritmos fueron ejecutados en 3 unidades de procesamiento distintas, de las cuales se pueden ver sus carateristicas en la Tabla 6.1.

	Intel Core i7-2620M	**NVIDIA GeForce GTX 460M**	**NVIDIA GeForce GTX 680**
Cores	2 (4 threads)	192	1536
Clock Speed	2700MHz	675MHz	1006MHz
Memory	16GB(MAX)	1.5GB	2GB
Memory Bandwidth	21.3GB/s	60GB/s	192.2GB/s

Tabla 6.1: Características de las unidades de procesamiento.

El Intel Core i7-2620M fue utilizado para medir el tiempo de ejecución del algoritmo original (secuencial) y también para ejecutar el nuevo algoritmo paralelizado.

Se utilizaron dos GPU, la primera (NVIDIA GeForce GTX 460M) fue utilizada para ver el rendimiento que se tiene en una GPU de gama media, ya que sus características son más cercanas a la GPU que puede tener un usuario común. La otra GPU (NVIDIA GeForce GTX 680) fue utilizada con el fin de ver el rendimiento de una GPU de gama alta.

Para cada algoritmo se muestra una tabla de resultados, en donde para cada unidad de procesamiento se muestra el tiempo de ejecución promedio, la mediana del tiempo de ejecución, el speed-up y la variación del tiempo en terrenos de distinto tamaño. Tanto el promedio como la mediana son calculados de un total de 10 ejecuciones. El tiempo se encuentra en mili-segundos y el tamaño del terreno corresponde a la cantidad de puntos. El speed-up corresponde al resultado de dividir el tiempo de ejecución del algoritmo secuencial en el Intel Core i7-2620M por el tiempo de ejecución en cada una de las unidades de procesamiento. La variación del tiempo corresponde al resultado de restar el tiempo de ejecución del algoritmo secuencial en el Intel Core i7-2620M a el tiempo de ejecución de cada una de las unidades de procesamiento, dividido en el tiempo de ejecución del algoritmo secuencial en el Inter Core i7-2620M. Tanto para el speed-up como para la variación del tiempo se usan los tiempos de ejecución promedios. Destacar que solo se hará alusión a los al tiempo de ejecución promedio y al speed-up, los otros datos sirven solo como referencia.

Junto con esto, se muestran dos figuras para cada algoritmo. Ambas corresponden a las redes de drenaje encontradas, siendo la primera el resultado de la ejecución secuencial y la segunda el resultado de la ejecución paralela del algoritmo.

Importante destacar que los tamaños de terreno utilizados para el algoritmo de Ángulo Diedro son mucho menores que para el resto de los algoritmos. Esto se explica con que es el único algoritmo que hace uso de la triangulación y uno de los grandes problemas que se encontró durante la realización de este trabajo fue el uso excesivo de memoria en la triangulación, lo que hace imposible probar el algoritmo de Ángulo Diedro en terrenos más grandes.

6.2.1. Peucker

La Tabla 6.2 muestra los resultados del algoritmo de Peucker. Se puede ver un speed-up tanto en su ejecución en paralelo en CPU como en GPU. Si bien el tiempo de ejecución en GPU es menor que en CPU para ambas GPU, al hacer la comparación con la GPU de uso común (NVIDIA GeForce GTX 460M) el rendimiento es similar. Esto se debe a que la principal limitación del algoritmo son los accesos a memoria y las operaciones por acceso son mínimas. Es decir, la principal fuente de latencia son los accesos a memoria, lo que no es el mejor caso para la GPU y si es amigable para la CPU. Si bien se podría pensar que debido a esto la ejecución en CPU debería ser mejor que en GPU, no hay una gran cantidad de accesos a memoria como para lograr este efecto, no existe una gran cantidad de saltos condicionales que afecten negativamente a la GPU, las instrucciones realizadas son simples y además los accesos a memoria tienen la localidad suficiente como para no tener demasiada influencia en los resultados. Por último, el paralelismo que permite la GPU junto con la superioridad de su ancho de banda de la memoria por sobre el de la CPU disminuye el efecto de la latencia provocada por los accesos.

Al comparar las Figuras 6.2 y 6.3 es posible ver que la red de drenaje entregada por ambas versiones del algoritmo es exactamente igual, es decir, si paralelización no provoca cambios en el resultado.

Terrain Size \ Processing Unit	Intel Core i7-2620M Secuencial				Intel Core i7-2620M Paralelo				NVIDIA GeForce GTX 460M				NVIDIA GeForce GTX 680			
	Pro.	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**
$\mathbf{1000^2}$	155.4	158	1	0	129.8	130.5	1.19	-0.16	22	24	7.06	-0.86	11.4	11.5	13.63	-0.93
$\mathbf{2000^2}$	375	379	1	0	161.4	159.5	2.32	-0.57	80.2	81.5	4.67	-0.79	41.6	41	9.01	-0.89
$\mathbf{3000^2}$	694.4	694.5	1	0	226.2	222	3.06	-0.67	159	160.5	4.36	-0.77	92.2	92	7.53	-0.87
$\mathbf{4000^2}$	1137.4	1137	1	0	322.4	332	3.52	-0.72	251.4	251.5	4.52	-0.78	159.4	159	7.13	-0.86
$\mathbf{5000^2}$	1713.6	1715	1	0	454.6	461	3.76	-0.73	357	357.5	4.8	-0.79	245	245.5	6.99	-0.86
$\mathbf{6000^2}$	2395.2	2392.5	1	0	615.6	621	3.89	-0.74	535.2	509.5	4.47	-0.78	352.2	352	6.8	-0.85
$\mathbf{7000^2}$	3192.2	3189.5	1	0	801.4	811	3.98	-0.75	688.8	682	4.63	-0.78	477.4	477	6.68	-0.85
$\mathbf{8000^2}$	4133.8	4134	1	0	1028.6	1027.5	4.01	-0.75	878.8	878	4.7	-0.79	624.6	624	6.61	-0.85

Tabla 6.2: Resultados del algoritmo de Peucker (Pro: Tiempo de ejecución promedio, Med: Mediana del tiempo de ejecución, Spdu: Speed-up, Var: Variación del tiempo de ejecución).

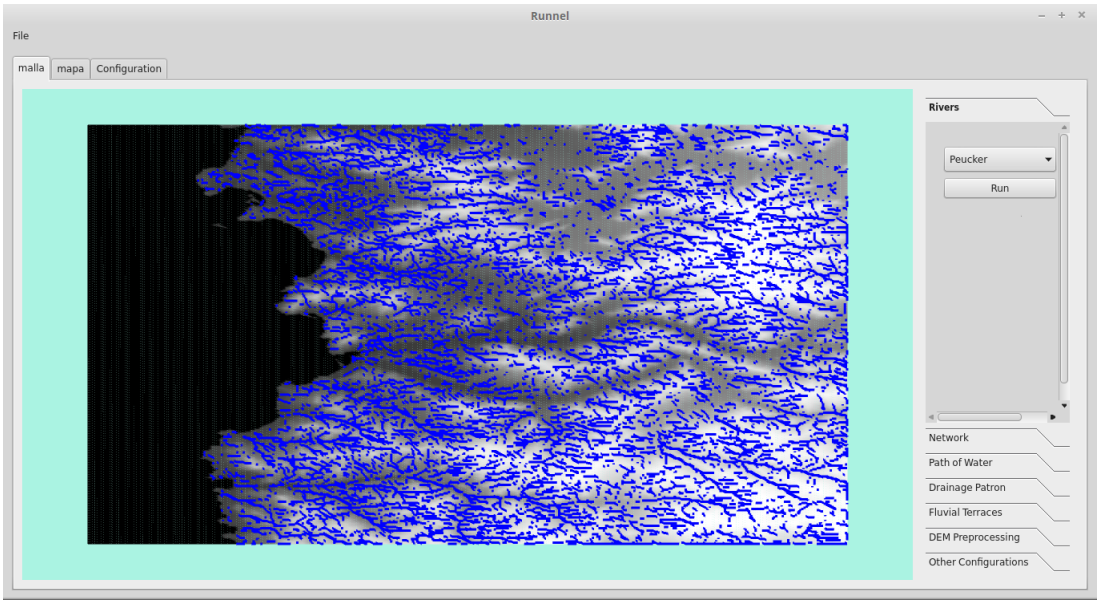


Figura 6.2: Ejecución del algoritmo de Peucker secuencial.

6.2.2. Callaghan

La Tabla 6.3 muestra los resultados del algoritmo de Callaghan. Se puede observar que existe un mayor speed-up que en Peucker, tanto en CPU como en GPU. Esto se explica con que el algoritmo no depende exclusivamente de los accesos a memoria, hay un poco más de procesamiento en comparación con el algoritmo de Peucker, lo que genera mejoras en el tiempo de ejecución en ambos tipos de unidades de procesamiento. También se puede ver que el tiempo de ejecución en GPU es significativamente menor que en CPU. Si bien hay saltos condicionales en donde la GPU se podría ver afectada, el paralelismo logra disminuir ese efecto.

Como se mencionó en la sección 4.1.2, esta versión paralela del algoritmo de Callaghan es teóricamente mas lenta en el peor caso, pero el peor caso es poco probable, ya que implica un

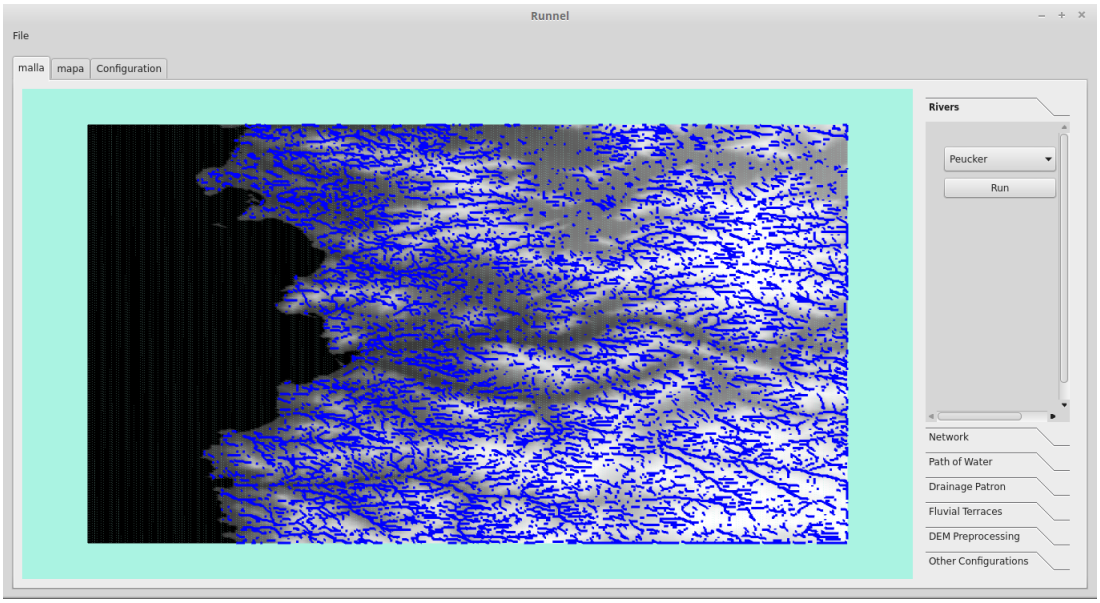


Figura 6.3: Ejecución del algoritmo de Peucker paralelo.

orden especifico de todos los puntos en el terreno. Por lo tanto el algoritmo no se ve gravemente afectado por ésto.

Al comparar las Figuras 6.4 y 6.5 es posible ver que la red de drenaje entregada por ambas versiones del algoritmo es exactamente igual, es decir, la paralelización tampoco provoca cambios en el resultado.

Terrain Size \ Processing Unit	Intel Core i7-2620M Secuencial				Intel Core i7-2620M Paralelo				NVIDIA GeForce GTX 460M				NVIDIA GeForce GTX 680			
	Pro.	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**
1000²	2193.6	2224	1	0	373.5	359.5	5.87	-0.83	245	231.5	8.95	-0.89	145.5	142	15.07	-0.93
2000²	9207.4	9288.5	1	0	1127.5	1112.5	8.16	-0.88	886.5	852	10.38	-0.90	555	568.5	16.58	-0.94
3000²	20767.2	20535.5	1	0	2518	2521.5	8.24	-0.88	2100.5	2141.5	9.88	-0.90	1220	1230	17.02	-0.94
4000²	39085.6	39104	1	0	4692	4651	8.33	-0.88	3638.5	3645	10.74	-0.91	2154.5	2121	18.14	-0.94
5000²	62772.6	62871	1	0	8159.5	8166	7.69	-0.87	5813.4	5901	10.79	-0.91	3335.5	3325.5	18.81	-0.95
6000²	94203	94174	1	0	12308.5	12221	7.65	-0.87	8679	8648.5	10.85	-0.91	4862	4844	19.37	-0.95
7000²	126248.6	126256.5	1	0	18455.5	18402	6.84	-0.85	12380	12126	10.19	-0.90	6609	6601.5	19.10	-0.95
8000²	172203.2	173064.5	1	0	26728.5	26700.5	6.44	-0.84	16428.5	16337.5	10.48	-0.90	8684	8666.5	19.82	-0.95

Tabla 6.3: Resultados del algoritmo de Callaghan (Pro: Tiempo de ejecución promedio, Med: Mediana del tiempo de ejecución, Spdu: Speed-up, Var: Variación del tiempo de ejecución).

6.2.3. RWFlood

Los resultados del algoritmo RWFlood se pueden ver en la Tabla 6.4. Este es el único algoritmo en donde se vieron resultados negativos. El tiempo de ejecución aumentó significativamente en todas las unidades de procesamiento, obteniendo así speed-ups menores que 1.

El gran problema del algoritmo en GPU es la pésima relación entre operaciones realizadas por accesos a memoria. La estrategia que se usó para ejecutar los puntos de una misma cola en paralelo

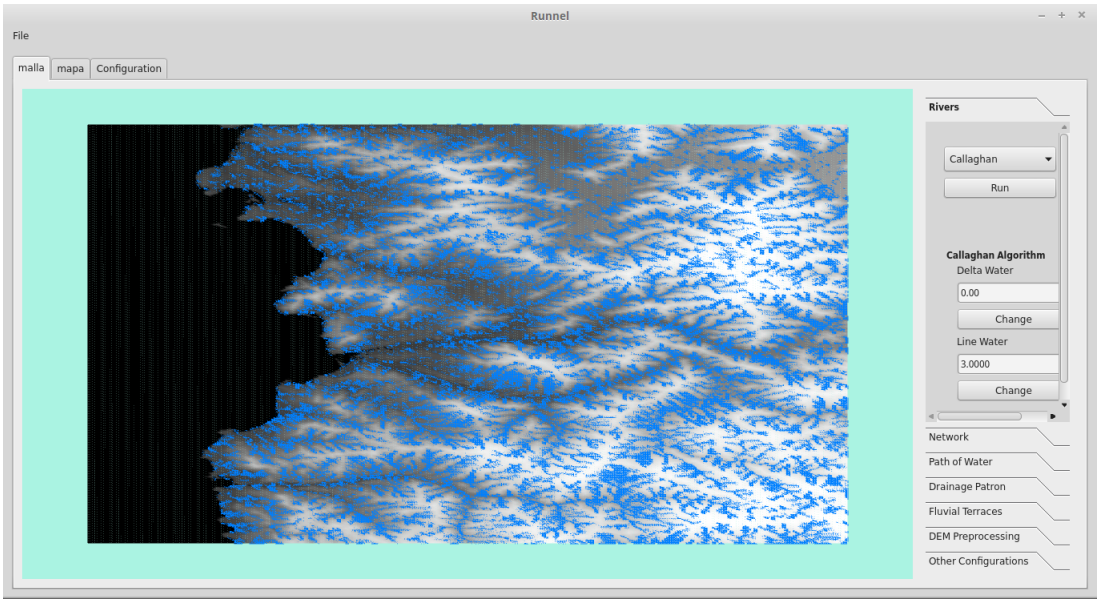


Figura 6.4: Ejecución del algoritmo de Callaghan secuencial.

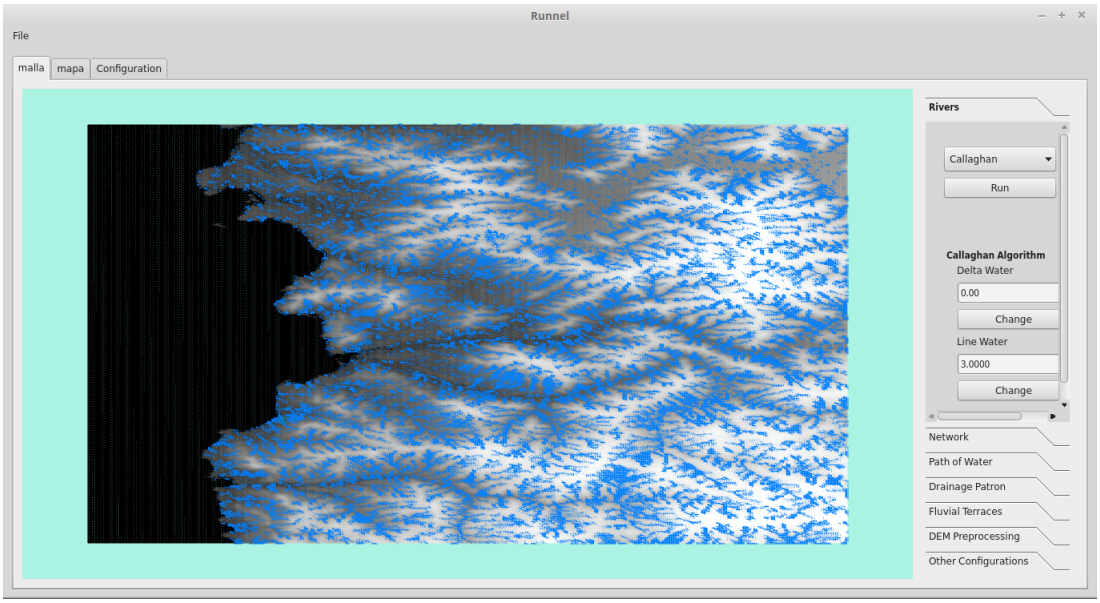


Figura 6.5: Ejecución del algoritmo de Callaghan paralelo.

(explicada en 4.1.3) hace que el procesamiento que realiza cada work-item sea mínimo, y a la vez todos tendrán al menos un acceso a memoria.

En cuanto a la ejecución en CPU, si bien los accesos a memoria son más barato, la localidad de los datos es nula, por lo que los cache no ayudan demasiado para disminuir la latencia de los accesos.

Al comparar las Figuras 6.6 y 6.7 es posible ver que la red de drenaje entregada por ambas versiones del algoritmo es bastante parecida, solo con pequeñas variaciones, principalmente en las zonas planas del terreno. Esto se debe a que el algoritmo secuencial sigue cierto orden al procesar los puntos de una cola, en cambio en la versión paralela este orden es desconocido y puede ir

variando.

Terrain Size \ Processing Unit	Intel Core i7-2620M Secuencial				Intel Core i7-2620M Paralelo				NVIDIA GeForce GTX 460M				NVIDIA GeForce GTX 680			
	Pro.	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**
1000²	1701.8	1702	1	0	1602.8	1602.5	1.06	-0.06	2513	2522	0.67	0.47	582.6	577	2.93	-0.66
2000²	6832.8	6838.5	1	0	10739.2	10766	0.63	0.57	16898	16899.5	0.4	1.47	3640.4	3642.5	1.87	-0.47
3000²	15372	15362	1	0	34022.4	34192.5	0.45	1.21	52169	51944.5	0.29	2.39	11349.6	11338	1.35	-0.26
4000²	27666.2	27586.5	1	0	34030.2	34231	0.81	0.23	94914	94971.5	0.29	2.43	20628	20689.5	1.34	-0.25
5000²	43500.2	43671	1	0	151307	150882.5	0.28	2.48	160194	160888.5	0.27	2.68	50310	50255.5	0.86	0.16
6000²	62561	62545.5	1	0	257906	254702	0.24	3.12	265146	260669.5	0.23	3.24	85825	85674	0.72	0.37
7000²	86852.8	87448	1	0	406207	404514.5	0.21	3.68	434466	431285	0.19	4.00	137591	135789.5	0.63	0.58
8000²	115238.2	115113.5	1	0	600220	601270.5	0.19	4.21	702171	705001	0.16	5.09	204086	206396	0.56	0.77

Tabla 6.4: Resultados del algoritmo RWFlood (Pro: Tiempo de ejecución promedio, Med: Mediana del tiempo de ejecución, Spdu: Speed-up, Var: Variación del tiempo de ejecución).

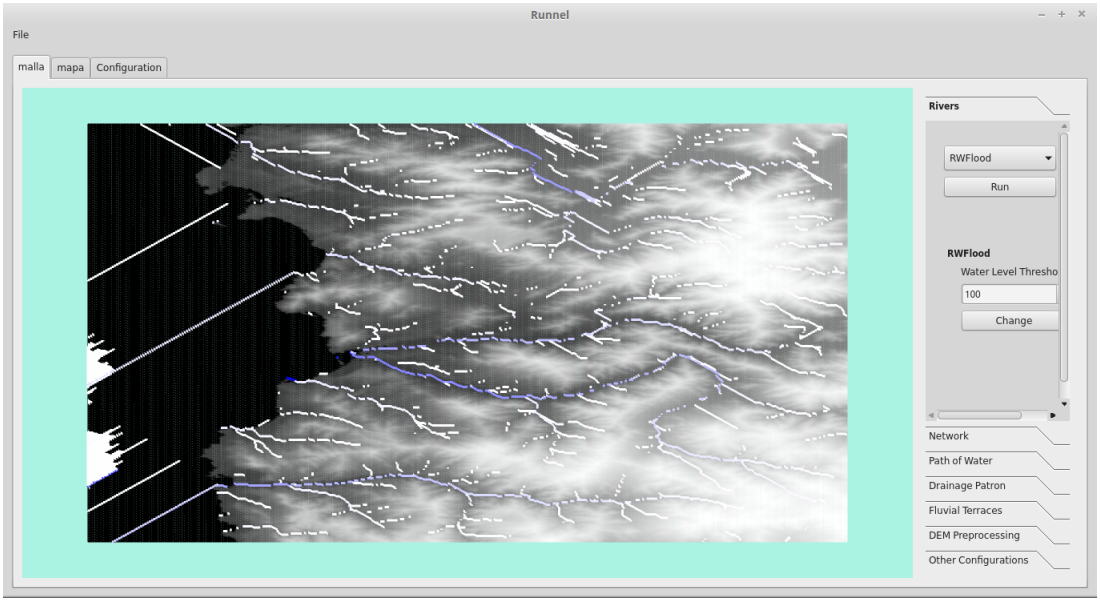


Figura 6.6: Ejecución del algoritmo RWFlood secuencial.

6.2.4. Ángulo Diedro

Los resultados del algoritmo Ángulo Diedro se pueden ver en la Tabla 6.5. Este es el algoritmo con mayor speed-up en GPU. Esto se debe a que es un algoritmo ideal para ser ejecutado en GPU: todos los work-items hacen un procedimiento igual, pocos saltos condicionales, y si bien existen accesos a memoria, la cantidad de operaciones aritméticas son mucho más (cálculo del angulo diedro), por lo que es posible disminuir el efecto de la latencia significativamente.

Si bien en CPU también se ve un buen rendimiento, la superioridad de la GPU se debe a lo ideal del algoritmo para este tipo de unidades de procesamiento.

Al comparar las Figuras 6.8 y 6.9 es posible ver que la red de drenaje entregada por ambas versiones del algoritmo es bastante parecida, solo con pequeñas variaciones. Las variaciones se deben a que la precisión y aproximaciones de números con decimales en OpenCL son distintas a las de g++ (compilador usado para Runnel), por lo que sus resultados tienen pequeñas diferencias.

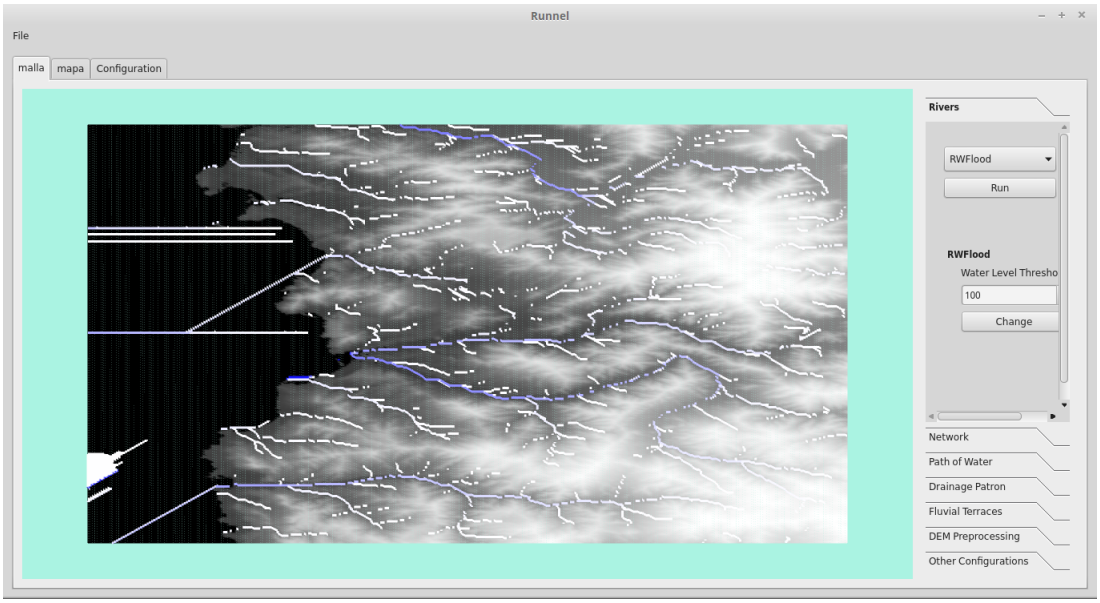


Figura 6.7: Ejecución del algoritmo RWFlood paralelo.

Con respecto al uso de memoria, el algoritmo de Ángulo Diedro tiene una complicación que no existe en el resto de los algoritmos. Éste hace uso de información contenida en los triángulos, arcos y puntos. Los triángulos contienen punteros a sus arcos y los arcos punteros a sus puntos. Estos punteros no se pueden usar directamente ya que las direcciones de memoria en el host son distintas a las del device. La solución que se decidió utilizar tiene la desventaja de que ignora el hecho de que la mayoría de los triángulos comparte sus ejes y puntos con al menos un triángulo. Ésto tiene como consecuencia que la memoria que se usa para la información de los puntos sea 6 veces lo que debería y la de los arcos es el doble de lo que debería.

Terrain Size \ Processing Unit	**Intel Core i7-2620M Secuencial**				**Intel Core i7-2620M Paralelo**				**NVIDIA GeForce GTX 460M**				**NVIDIA GeForce GTX 680**			
	Pro.	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**
350^2	470.66	462	1	0	328.75	328	1.43	-0.30	32.33	32	14.55	-0.93	20	22	23.53	-0.96
700^2	1614.33	1586	1	0	445.25	446	3.62	-0.72	116.66	117	13.83	-0.93	69	68	23.39	-0.96
1400^2	6226	6317	1	0	920.74	922.5	6.76	-0.85	390.33	390	15.95	-0.94	266.5	263	23.36	-0.96

Tabla 6.5: Resultados del algoritmo Ángulo Diedro (Pro: Tiempo de ejecución promedio, Med: Mediana del tiempo de ejecución, Spdu: Speed-up, Var: Variación del tiempo de ejecución).

6.3. Triangulación Simplificada

En esta sección identificamos tres triangulaciones distintas:

- **Triangulación Regular**: Triangulación original sin simplificación.
- **Triangulación Simplificada - Área Triángulo Mínima**: Simplificación en que el criterio para intentar eliminar un arco es que el triángulo tenga un área menor al 102 % del área del triángulo con área mínima del terreno.

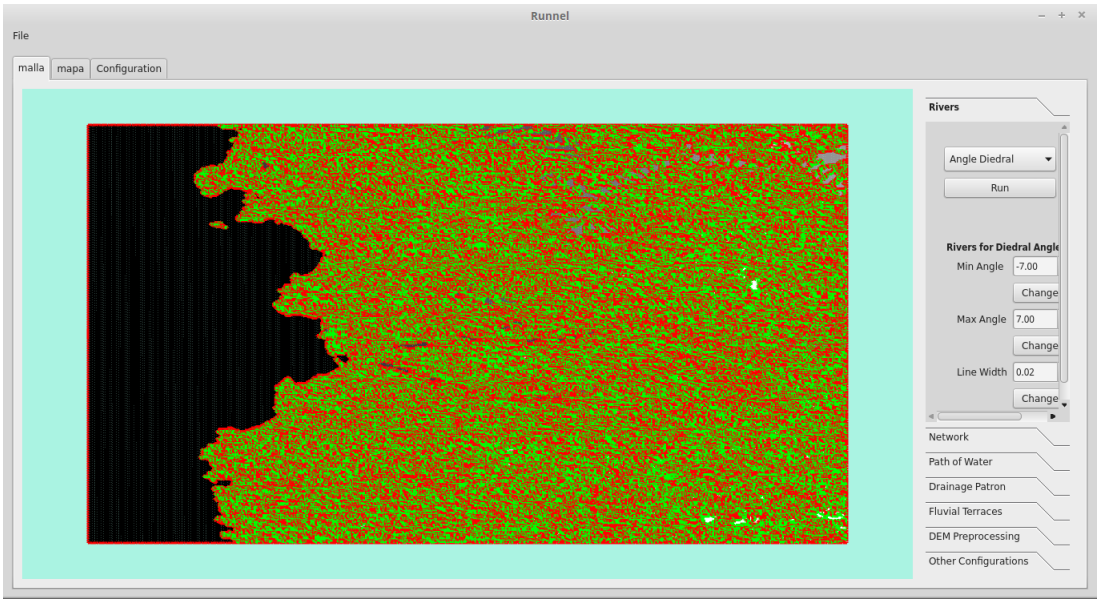


Figura 6.8: Ejecución del algoritmo Ángulo Diedro secuencial.

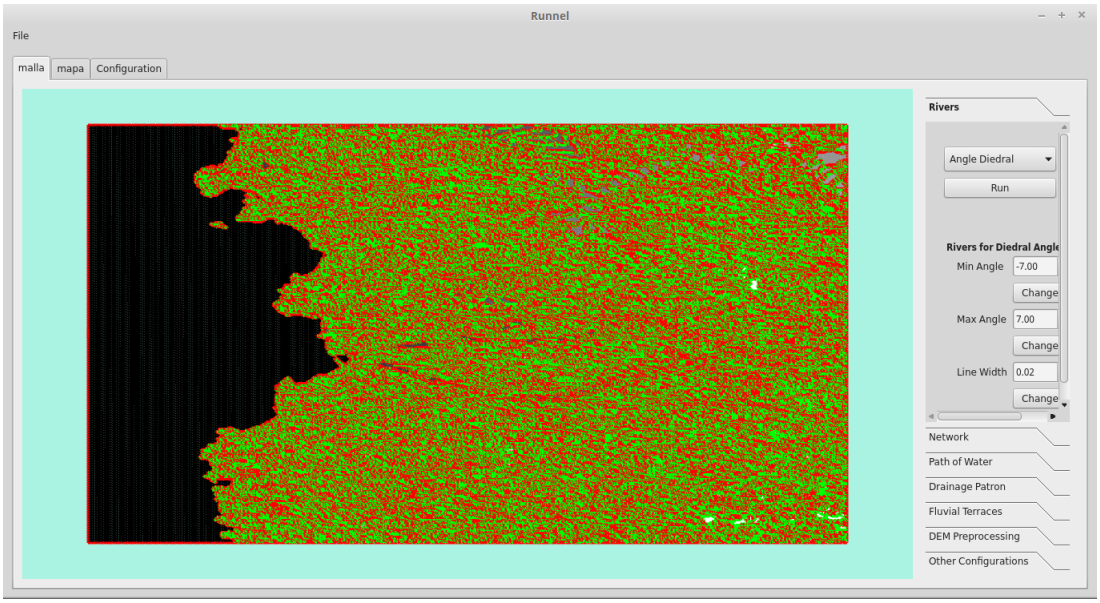


Figura 6.9: Ejecución del algoritmo Ángulo Diedro paralelo.

- **Triangulación Simplificada - Área Triángulo Promedio**: Simplificación en que el criterio para intentar eliminar un arco es que el triángulo tenga un área que sea menor al área promedio de todos los triángulos del terreno.

Si bien se decidió que Runnel usará la Triangulación Simplificada - Área Triángulo Mínima (por razones explicadas en la sección Triangulación Simplificada del capítulo Diseño), la Triangulación Simplificada - Área Triángulo Promedio se utilizó con fines experimentales.

El algoritmo que se ejecutó para medir sus cambios en el tiempo de ejecución fue el Ángulo Diedro. Si bien hay otros algoritmos que usan la triangulación, se decidió usar éste ya que fue estudiado previamente.

Al igual que en la paralelización del algoritmo de Ángulo Diedro, el tamaño de los terrenos de prueba son pequeños debido al uso excesivo de memoria en Runnel para la triangulación. Todos los algoritmos fueron ejecutados en una CPU Intel Core i7-2620M.

La Tabla 6.6 muestra los resultados del tiempo de ejecución del algoritmo Ángulo Diedro bajo las distintas triangulaciones en terrenos de distintos tamaños. Si bien podemos ver que el algoritmo de Ángulo Diedro muestra mejoras en tiempo de ejecución con ambas simplificaciones, estas mejoras están lejos de las logradas con el paralelismo.

Las mejoras en rendimiento se explican con la disminución en la cantidad de triángulos que necesitan ser procesados. La Tabla 6.7 muestra la cantidad de triángulos en las distintas triangulaciones y la proporción entre la cantidad de triángulos en las distintas triangulaciones por sobre la cantidad de triángulos en la triangulación regular.

A medida que el terreno crece, la cantidad de triángulos y tiempo de ejecución tiene una directa relación. Por ejemplo, podemos ver que en una superficie de tamaño 1400^2 la cantidad de triángulos al usar la Triangulación Simplificada - Área Triángulo Promedio disminuye en la mitad, y en consecuencia, su tiempo de ejecución disminuye en un 50 %.

La Tabla 6.8 muestra el tiempo que demoran las triangulaciones simplificadas en terrenos de distintos tamaños. Vemos que a pesar de que la simplificación tiene efectos positivos en el rendimiento de los algoritmos que usan la triangulación, esta viene acompañada con un overhead importante.

Con el fin de entender el tipo de cambios que provoca la triangulación simplificada, en la Figura 6.10 y la Figura 6.11 se muestra un zoom en terrenos distintos después de aplicar la simplificación.

Es interesante ver los efectos que tienen las distintas triangulaciones sobre la visualización. Las Figuras 6.12, 6.13 y 6.14 muestran como se ve el terreno después de las triangulaciones. Acá se puede ver que los cambios más notorios se encuentran en las zonas planas del terreno.

Las Figuras 6.15, 6.16 y 6.17 muestran el efecto del algoritmo Ángulo Diedro sobre terrenos con las distintas triangulaciones. Este es el algoritmo en donde se ven más cambios visuales. Es difícil determinar el verdadero impacto de estos cambios ya que no es muy claro cual es la información que muestra la visualización del algoritmo Ángulo Diedro.

Las Figuras 6.18, 6.19 y 6.20 muestran el efecto de aplicar el algoritmo Normal Vector Similarity sobre terrenos con las distintas triangulaciones. Sus resultados son coherentes con lo esperado, ya que Normal Vecotor Similarity marca terrenos planos y en las distintas triangulaciones marca prácticamente los mismos sectores, los cuales coinciden con los más afectados por la simplificación.

Por último podemos ver en las Figuras 6.21, 6.22 y 6.23 los efectos de la simplificación en la visualización del algoritmo de Peucker. Este resultado también es interesante ya que se puede apreciar que debido a que la simplificación afecta principalmente los sectores planos, algoritmos que no trabajan sobre estos sectores, como es el caso se Peucker, no se ven significativamente afectados, la visualización es similar.

Triangulación / Terrain Size	Triangulación Regular				Triangulación Simplificada - Área Triángulo Mínima				Triangulación Simplificada - Área Triángulo Promedio			
	Pro.	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**	**Pro.**	**Med.**	**Spdu**	**Var.**
$\mathbf{350^2}$	470.66	462	1	0	411	415.5	1.14	-0.13	303	304.5	1.55	-0.36
$\mathbf{700^2}$	1614.33	1586	1	0	1360.33	1363	1.18	-0.16	966.33	967.5	1.67	-0.40
$\mathbf{1400^2}$	6226	6317	1	0	4999	4970	1.24	-0.20	3200.66	3193.5	1.94	-0.49

Tabla 6.6: Resultados del algoritmo Ángulo Diedro usando distintas triangulaciones (Pro: Tiempo de ejecución promedio, Med: Mediana del tiempo de ejecución, Spdu: Speed-up, Var: Variación del tiempo de ejecución).

Triangulación / Terrain Size	Triangulación Regular		Triangulación Simplificada - Área Triángulo Mínima		Triangulación Simplificada - Área Triángulo Promedio	
	# Triángulos	**Proporción**	**# Triángulos**	**Proporción**	**# Triángulos**	**Proporción**
$\mathbf{350^2}$	242208	1	194680	0.8	121952	0.5
$\mathbf{700^2}$	974408	1	772198	0.79	479528	0.49
$\mathbf{1400^2}$	3908808	1	3023406	0.77	1830422	0.46

Tabla 6.7: Resultados de la cantidad de triángulos en las distintas triangulaciones (# Triángulos: Cantidad de triángulos, Proporción: Proporción de triángulos respecto a la triangulación regular).

Triangulación / Terrain Size	Triangulación Simplificada - Área Triángulo Mínima		Triangulación Simplificada - Área Triángulo Mínima	
	Promedio	**Mediana**	**Promedio**	**Mediana**
$\mathbf{350^2}$	2425	2377	4168	4151
$\mathbf{700^2}$	13667	13717	19349	18992
$\mathbf{1400^2}$	71743	71555	92864	93106

Tabla 6.8: Resultados del tiempo de ejecución de las distintas triangulaciones simplificadas (Promedio: Tiempo de ejecución promedio, Mediana: Mediana del tiempo de ejecución).

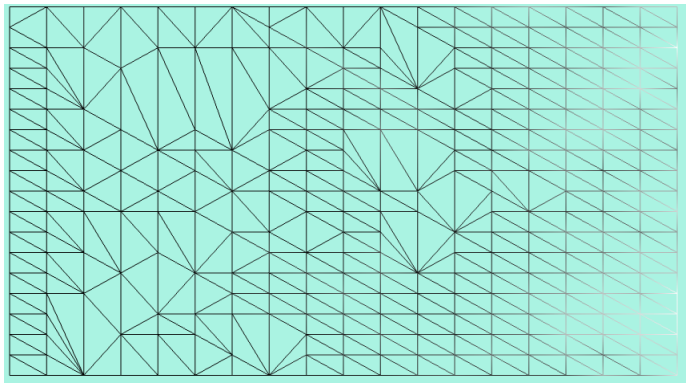


Figura 6.10: Zoom del resultado de la ejecución de la triangulación simplificada en un terreno.

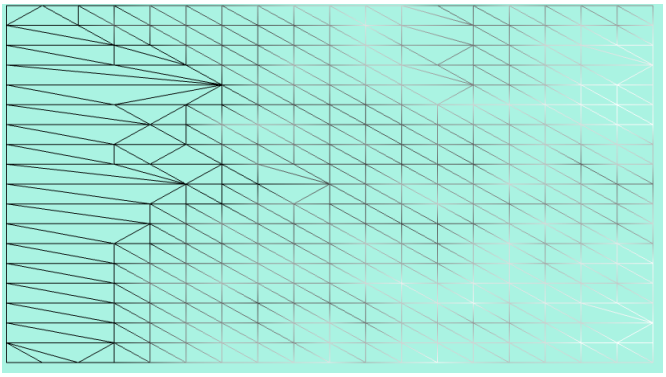


Figura 6.11: Zoom del resultado de la ejecución de la triangulación simplificada en un terreno

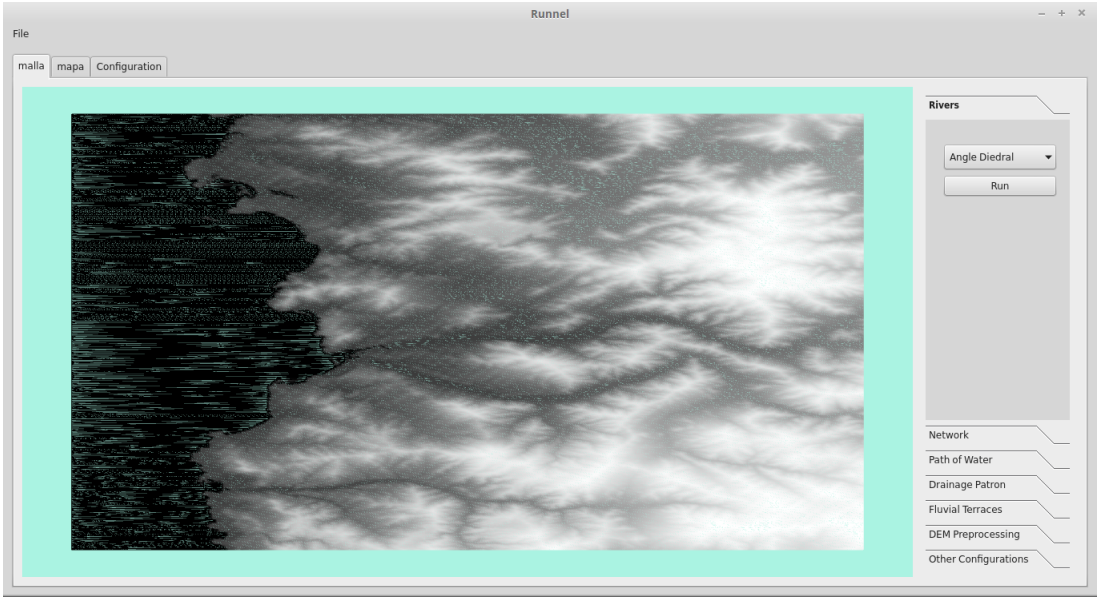


Figura 6.12: Terreno utilizando la Triangulación Simplificada - Área Triángulo Mínima.

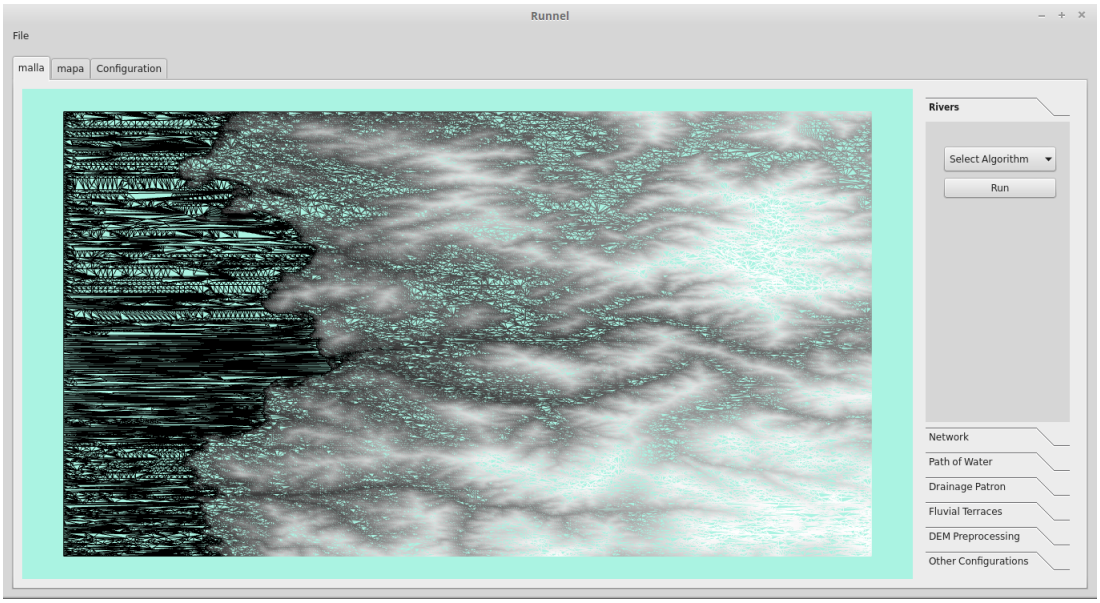


Figura 6.13: Terreno utilizando la Triangulación Simplificada - Área Triángulo Promedio.

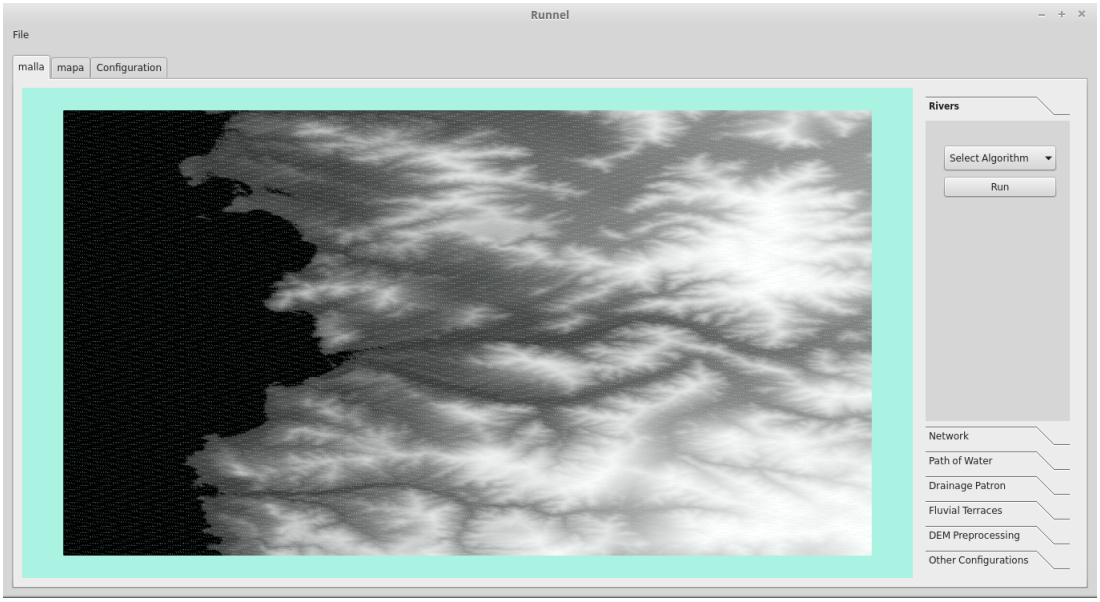


Figura 6.14: Terreno utilizando la Triangulación Regular.

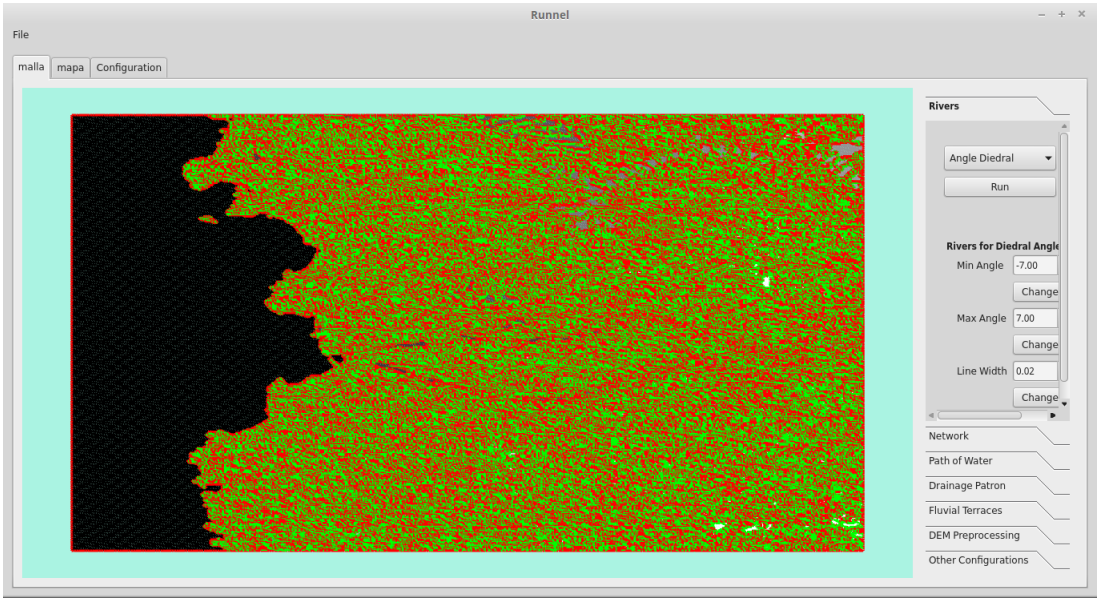


Figura 6.15: Ejecución del algoritmo Ángulo Diedro usando la Triangulación Regular.

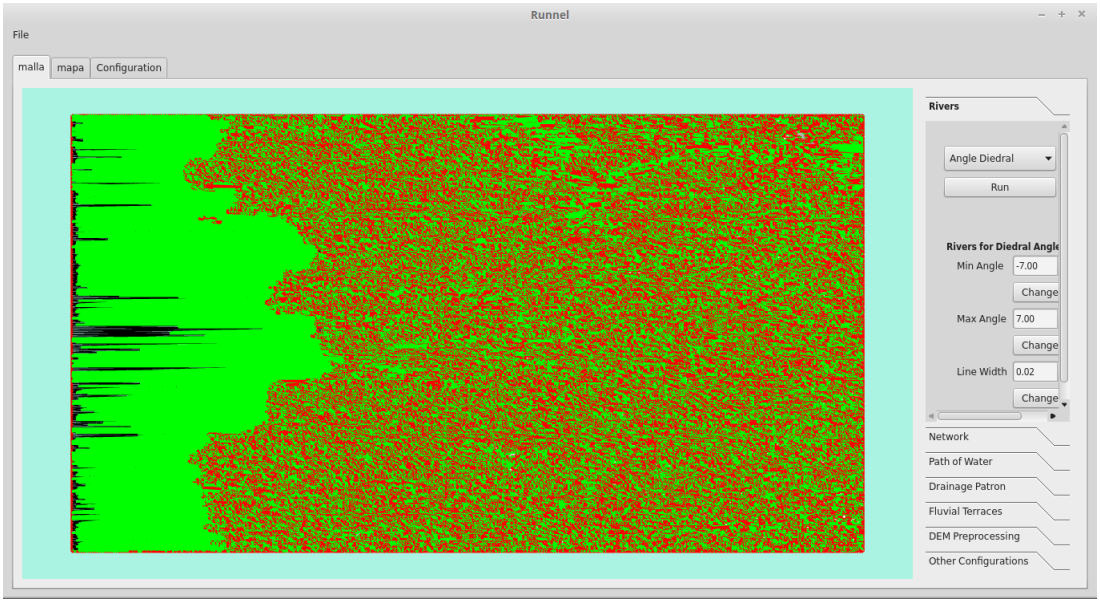


Figura 6.16: Ejecución del algoritmo Ángulo Diedro usando la Triangulación Simplificada - Área Triángulo Mínima.

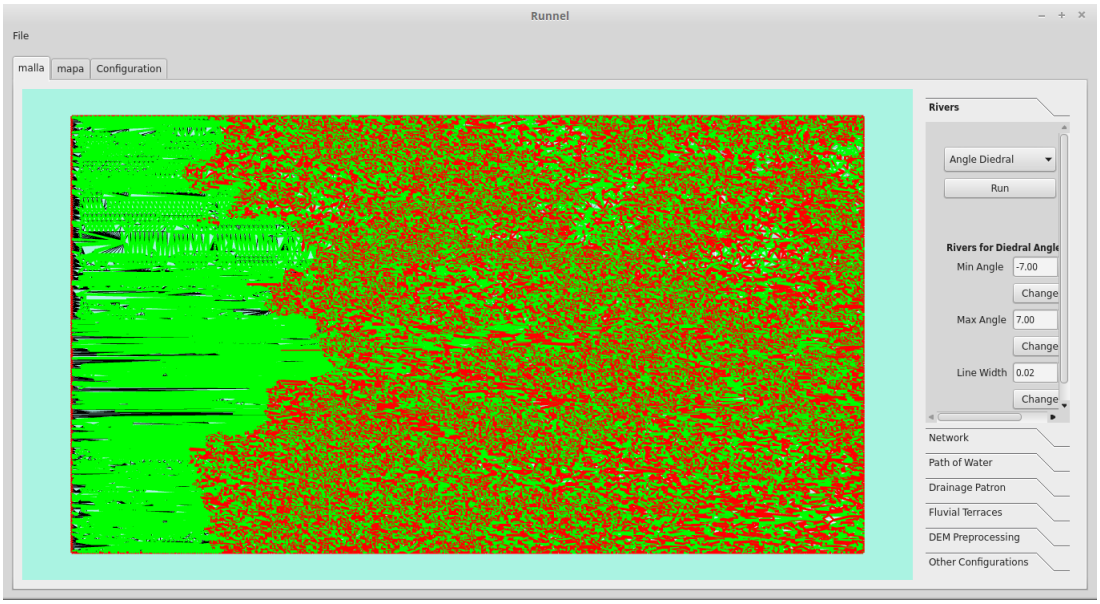


Figura 6.17: Ejecución del algoritmo Ángulo Diedro usando la Triangulación Simplificada - Área Triángulo Promedio.

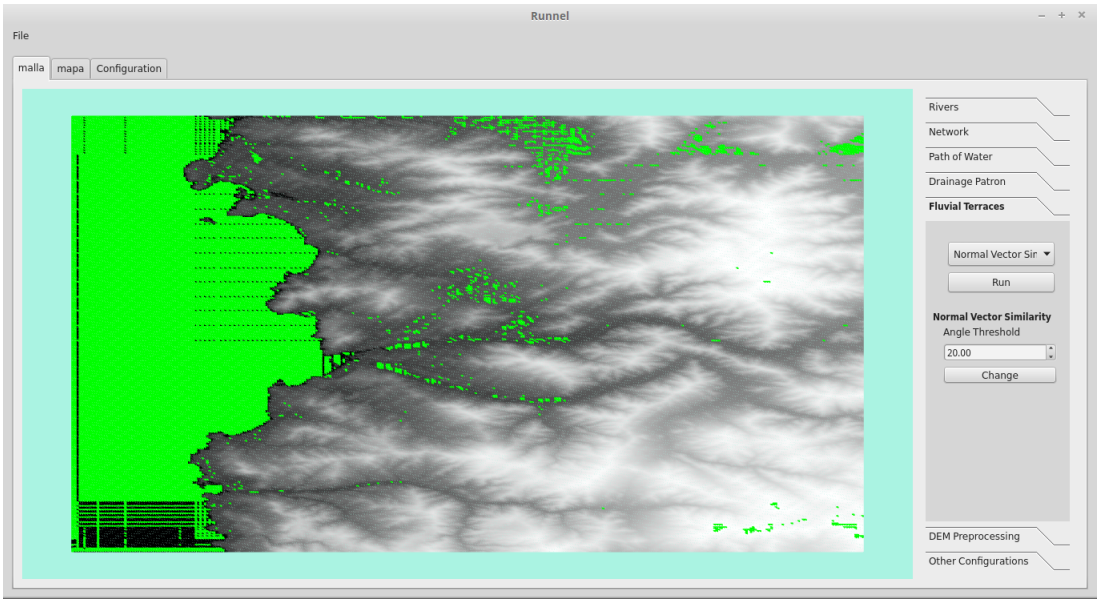


Figura 6.18: Ejecución del algoritmo Normal Vector Similarity usando la Triangulación Regular.

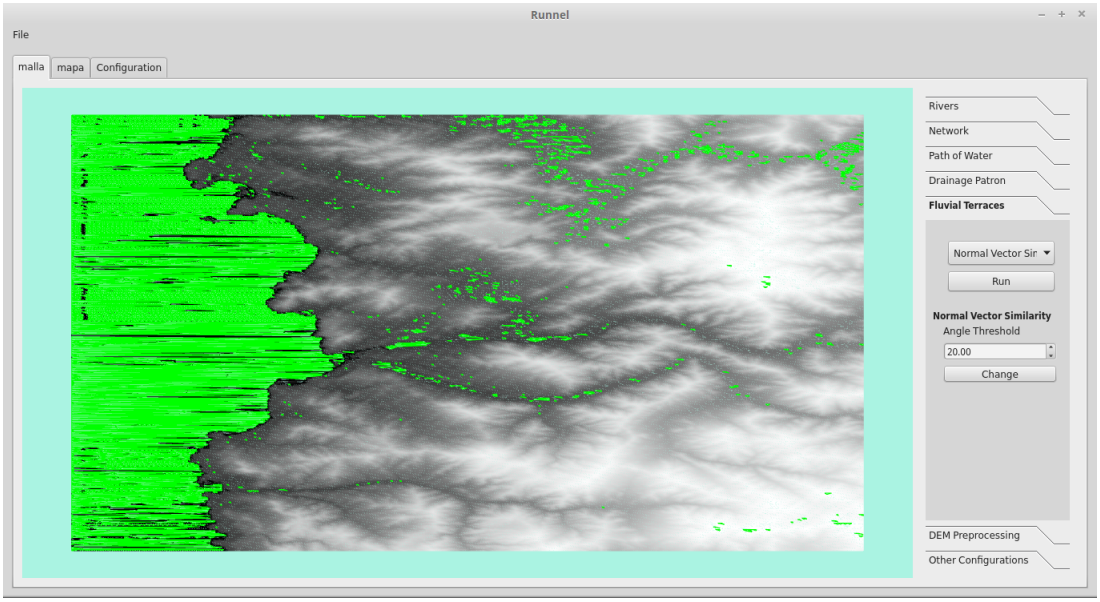


Figura 6.19: Ejecución del algoritmo Normal Vector Similarity usando la Triangulación Simplificada - Área Triángulo Mínima.

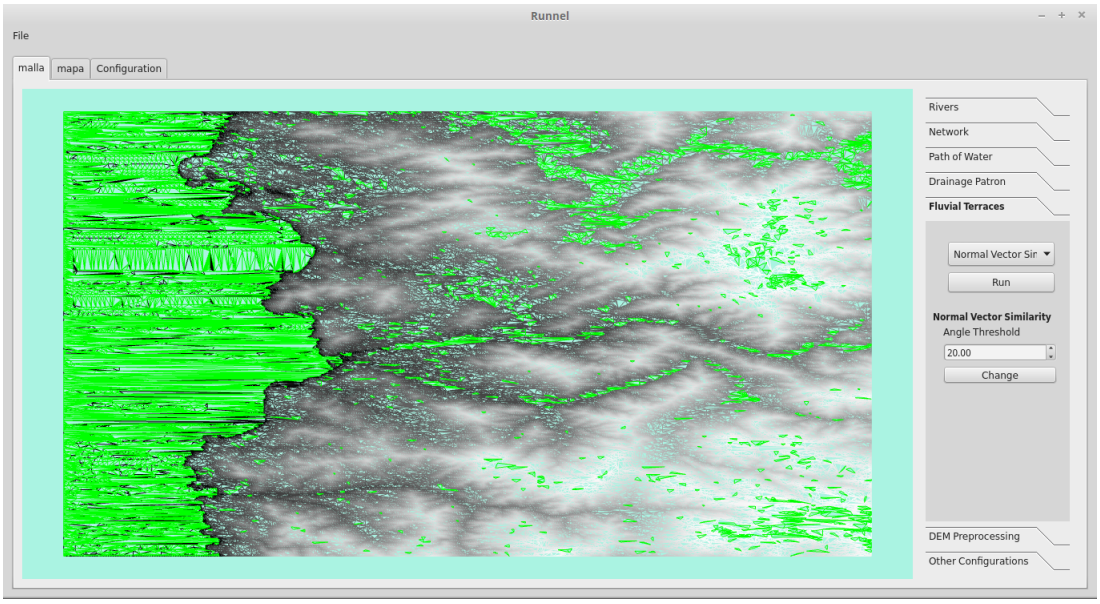


Figura 6.20: Ejecución del algoritmo Normal Vector Similarity usando la Triangulación Simplificada - Área Triángulo Promedio.

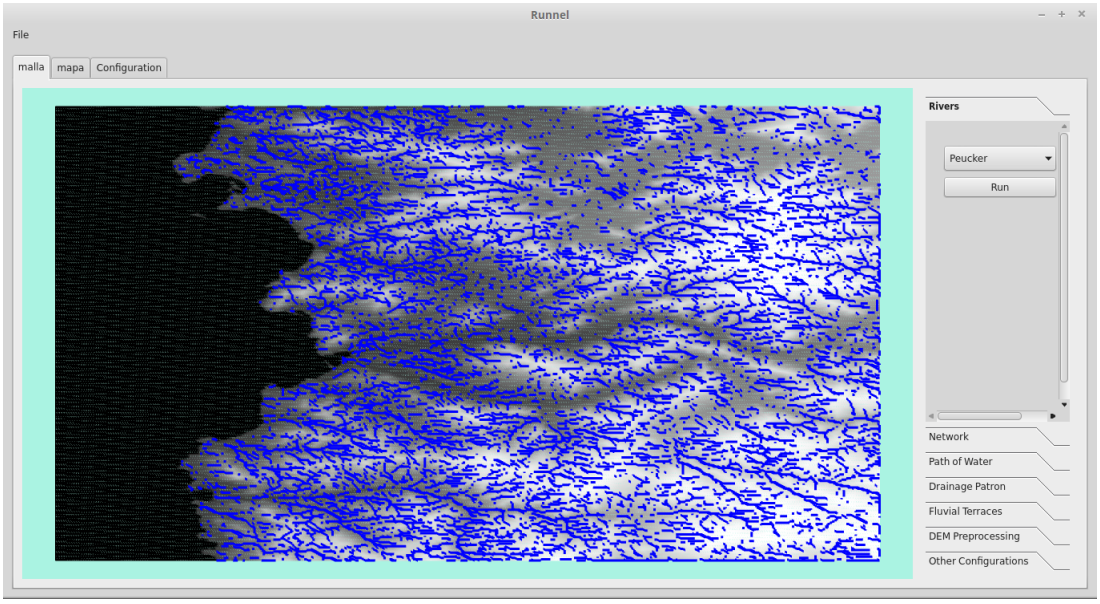


Figura 6.21: Ejecución del algoritmo de Peucker usando la Triangulación Regular.

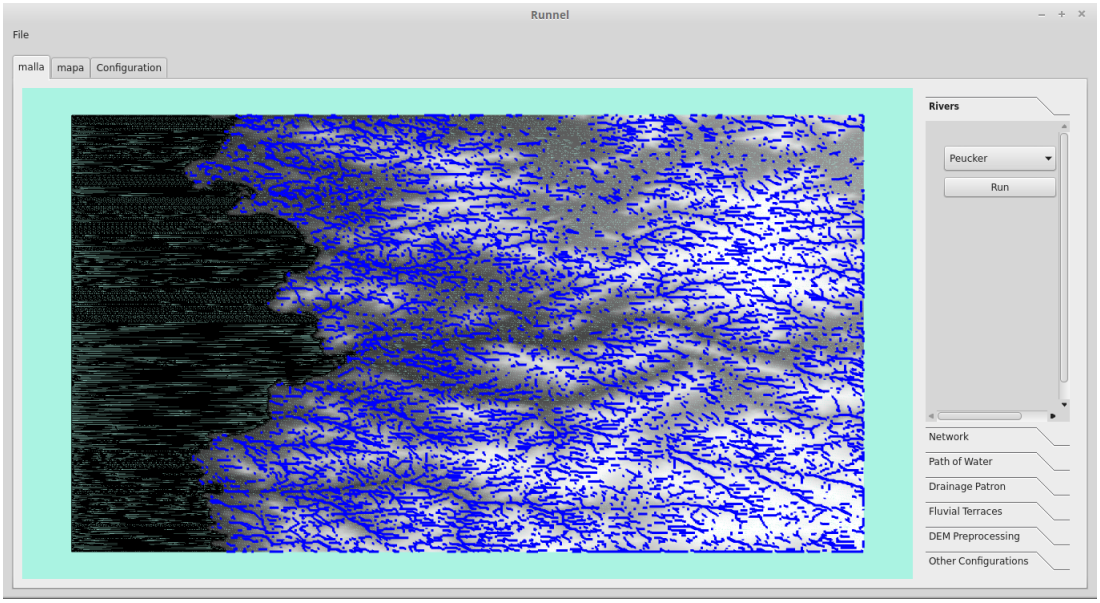


Figura 6.22: Ejecución del algoritmo de Peucker usando la Triangulación Simplificada - Área Triángulo Mínima.

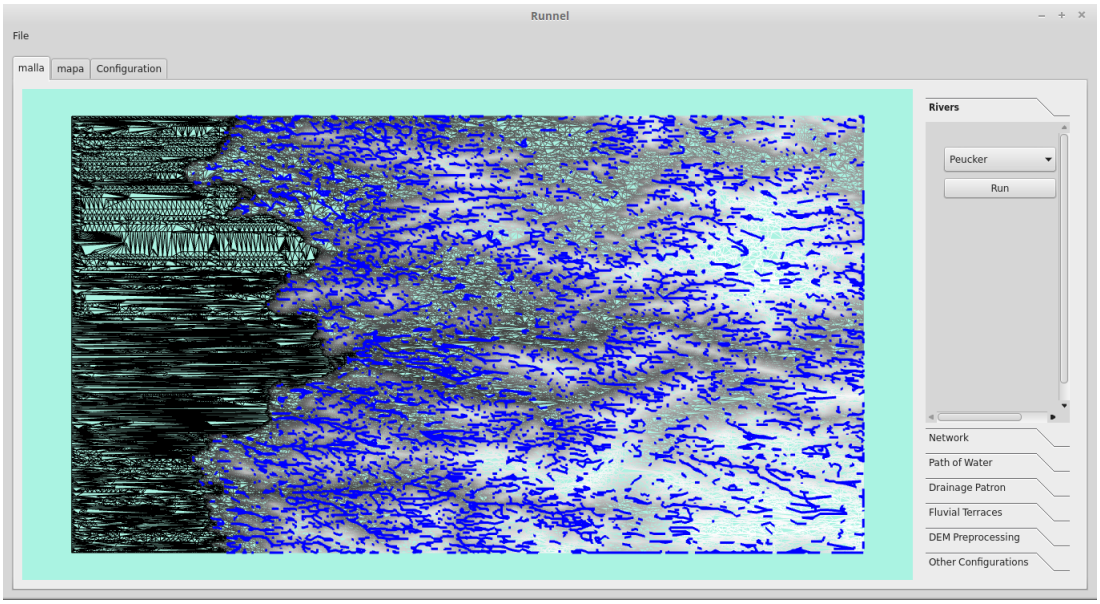


Figura 6.23: Ejecución del algoritmo de Peucker usando la Triangulación Simplificada - Área Triángulo Promedio.

Capítulo 7

Conclusión y Trabajo Futuro

En el presente trabajo se logró obtener conocimiento respecto a la paralelización de algoritmos tanto en CPU como en GPU. Esto fue gracias a la flexibilidad que entrega el uso de OpenCL para escoger el dispositivo en donde se desea ejecutar un algoritmo paralelo. A cambio de esta flexibilidad, se obtuvo un desarrollo más complejo, en donde tanto el testing como el debugging fueron engorrosos. Si bien el testing y el debugging suele tener un grado de dificultad mayor en algoritmos paralelos que en algoritmos secuenciales, esta dificultad en OpenCL aumenta debido a que existen pocas herramientas de ayuda.

Otra complejidad es que dado la flexibilidad que permite OpenCL, se pueden hacer muchas combinaciones en cuanto al diseño del algoritmo. No se puede optimizar pensando en un solo tipo de unidad de procesamiento, si no que se debe encontrar una solución lo más amigable para todo tipo de arquitectura. Encontrar esta configuración es un procedimiento lento y complejo, ya que los resultados solo se obtienen realizando pruebas de distintas configuraciones en distintas unidades de procesamiento.

Usando OpenCL se logró la paralelización de los algoritmos de detección de redes de drenaje, obteniendo un speed-up significativo en tres de los cuatro algoritmos paralelizados. En el desarrollo de estos algoritmos se adquirió conocimiento sobre las diferencias de desempeño de una CPU con una GPU. Cuando un problema puede ser resuelto haciendo un uso eficiente de la arquitectura de una GPU, su rendimiento suele ser significativamente mejor. Un ejemplo de esto son los resultados del algoritmo Ángulo Diedro. Al mismo tiempo, si bien las GPU tiene una gran capacidad de procesamiento paralelo, no es posible solucionar problemas que existen cuando el diseño del algoritmo lo hacen poco amigable con su arquitectura. Un ejemplo de esto es el algoritmo RWFlood.

Por otro lado, se logró implementar una triangulación simplificada, lo que era uno de los principales objetivos del trabajo. Los resultados en cuanto a las mejoras en tiempo de ejecución de los algoritmos que usan la triangulación son positivos, pero es necesario hacer un estudio más detallado con el fin de ver el verdadero impacto que tiene la simplificación sobre el terreno y cual es su configuración óptima.

Por otro lado, la ejecución de la triangulación simplificada suma una cantidad de tiempo importante al tiempo ya tomado en preparación del terreno para luego ser analizado. De todas formas

hay que la triangulación simplificada se implementó con el fin de eliminar información redundante y mejorar la precisión del terreno, las mejoras de rendimiento eran una consecuencia secundaria, por lo que el resultado puede ser útil a pesar del tiempo que demora la simplificación.

Si bien los resultados de esta memoria ayudan al desarrollo de Runnel, este tambien deja trabajos pendientes junto con otros problemas ya existentes. Se proponen los siguientes trabajos futuros:

- Optimizar el uso de memoria en la triangulación. Esto es uno de los grandes problemas del software ya que limita el tamaño máximo del terreno que se puede analizar.
- Analizar en detalle el impacto de la triangulación simplificada y encontrar el mejor criterio para simplificar.
- Explorar alternativas, variaciones u otro tipo de optimizaciones del algoritmo RWFlood, ya que es el único algoritmo que no obtuvo mejoras.
- Re-diseño de la interfaz para usuarios finales. Actualmente Runnel es un software de prueba para desarrolladores. Está lejos de ser amigable para usuarios finales reales.

Capítulo 8

Anexo

```
__kernel void markHighestPoint(__global  float* coordsz, __global char* flags,
                               __global const int*  pwidth, __global const int*  pheight) {
    int x = get_global_id(0);

    int width = *pwidth;
    int height = *pheight;

    if(x < width*height) {

        if((x+1)%width == 0) return;
        if(x >= width*(height-1)) return;

        //window to process
        float window[4];
        window[0] = coordsz[x];
        window[1] = coordsz[x + 1];
        window[2] = coordsz[x + width];
        window[3] = coordsz[x + width + 1];

        //Obtain the highest point in the current window
        float maxHeight = getWindowMaxHeight(window);

        if(window[0] == maxHeight) flags[x]=1;
        if(window[1] == maxHeight) flags[x+1]=1;
        if(window[2] == maxHeight) flags[x+width]=1;
        if(window[3] == maxHeight) flags[x+width+1]=1;
    }

}
```

Figura 8.1: Kernel del algoritmo de Peucker que marca los puntos de mayor altura en una ventana.

```
__kernel void iterateQueues(__global char* flags, __global float* coordsz,
                            __global int* pwidth, __global int* pheight,
                            __global float* queueArray, __global int* pcurrentHeight,
                            __global char* pshouldRepeat) {

    int id = get_global_id(0);
    int width = *pwidth;
    int height = *pheight;
    int currentHeight = *pcurrentHeight;

    if(id < width*height) {
        if((int)queueArray[id] == currentHeight) {

            queueArray[id] = -1.0;
            int neighborId;

            for(int i = 0; i<3; i++){

                neighborId = id - width -1 + i;

                if(neighborId > -1 && neighborId < width*height) {
                    if(flags[neighborId] == 0) {
                        setDirectionTowardsAdjacentPoint(flags, neighborId, id, width);
                        if(coordsz[neighborId] <= currentHeight)
                        {
                            queueArray[neighborId] = currentHeight*1.0;
                            *pshouldRepeat = 1;
                        }
                        else{
                            queueArray[neighborId] = coordsz[neighborId];
                        }
                    }
                }
            }
```

Figura 8.2: Parte 1 del kernel del algoritmo RWFlood que calcula la dirección de flujo de un punto.

```
    neighborId = id - 1;
    if(neighborId > -1 && neighborId < width*height) {
        if(flags[neighborId] == 0) {
            setDirectionTowardsAdjacentPoint(flags, neighborId, id, width);
            if(coordsz[neighborId] <= currentHeight)
            {
                queueArray[neighborId] = currentHeight*1.0;
                *pshouldRepeat = 1;
            }
            else{
                queueArray[neighborId] = coordsz[neighborId];
            }
        }
    }

    neighborId = id + 1;
    if(neighborId > -1 && neighborId < width*height) {
        if(flags[neighborId] == 0) {
            setDirectionTowardsAdjacentPoint(flags, neighborId, id, width);
            if(coordsz[neighborId] <= currentHeight)
            {
                queueArray[neighborId] = currentHeight*1.0;
                *pshouldRepeat = 1;
            }
            else{
                queueArray[neighborId] = coordsz[neighborId];
            }
        }
    }
```

Figura 8.3: Parte 2 del kernel del algoritmo RWFlood que calcula la dirección de flujo de un punto.

```
            for(int i = 0; i<3; i++){

                neighborId = id + width -1 + i;
                if(neighborId > -1 && neighborId < width*height) {
                    if(flags[neighborId] == 0) {
                        setDirectionTowardsAdjacentPoint(flags, neighborId, id, width);
                        if(coordsz[neighborId] <= currentHeight)
                        {
                            queueArray[neighborId] = currentHeight*1.0;
                            *pshouldRepeat = 1;
                        }
                        else{
                            queueArray[neighborId] = coordsz[neighborId];
                        }
                    }
                }
            }

        }
    }
}
```

Figura 8.4: Parte 3 del kernel del algoritmo RWFlood que calcula la dirección de flujo de un punto.

```
__kernel void calculateNeighbourByEdges(__global int3* trianglePointsId,
                                        __global int2* triangleEdgesId,
                                        __global int* edgeNeighbourTriangles,
                                        __global float3* edgeNeighbourTrianglesNormal,
                                        __global float3* edgeVectors,
                                        __global int* pTrianglesSize,
                                        __global float3Packed* angles) {

    int id = get_global_id(0);
    int trianglesSize = *pTrianglesSize;

    if(id < trianglesSize) {

        float3 value;
        int triangleEdge = id*3;
        int currentEdgeNeighbourTrianglesNormal = triangleEdge*2;
        int anglesIndex = id*3;

        for(int edgeIndex = 0; edgeIndex<3; edgeIndex++, currentEdgeNeighbourTrianglesNormal+=2, triangleEdge++) {

            float diedralAngle = 180;

            if(edgeNeighbourTriangles[triangleEdge] == 2) {
                diedralAngle = acospi(dot(edgeNeighbourTrianglesNormal[currentEdgeNeighbourTrianglesNormal],
                                          edgeNeighbourTrianglesNormal[currentEdgeNeighbourTrianglesNormal+1]))*180.0;
                float3 crossProduct = cross(edgeNeighbourTrianglesNormal[currentEdgeNeighbourTrianglesNormal],
                                            edgeNeighbourTrianglesNormal[currentEdgeNeighbourTrianglesNormal+1]);

                if(dot(crossProduct, edgeVectors[triangleEdge]) >= 0){
                        diedralAngle = -diedralAngle;
                }

            }

            if(trianglePointsId[id].x == triangleEdgesId[triangleEdge].x &&
                    trianglePointsId[id].y == triangleEdgesId[triangleEdge].y) {
                value.x = diedralAngle;
            }
```

Figura 8.5: Parte 1 del kernel del algoritmo Ángulo Diedro que calcula el ángulo diedro entre un triángulo y sus vecinos.

```
90
91              if(trianglePointsId[id].y == triangleEdgesId[triangleEdge].x &&
92                      trianglePointsId[id].x == triangleEdgesId[triangleEdge].y) {
93                  value.x = diedralAngle;
94              }
95
96              if(trianglePointsId[id].z == triangleEdgesId[triangleEdge].x &&
97                      trianglePointsId[id].y == triangleEdgesId[triangleEdge].y) {
98                  value.y = diedralAngle;
99              }
100
101             if(trianglePointsId[id].y == triangleEdgesId[triangleEdge].x &&
102                     trianglePointsId[id].z == triangleEdgesId[triangleEdge].y) {
103                 value.y = diedralAngle;
104             }
105
106             if(trianglePointsId[id].z == triangleEdgesId[triangleEdge].x &&
107                     trianglePointsId[id].x == triangleEdgesId[triangleEdge].y) {
108                 value.z = diedralAngle;
109             }
110
111             if(trianglePointsId[id].x == triangleEdgesId[triangleEdge].x &&
112                     trianglePointsId[id].z == triangleEdgesId[triangleEdge].y) {
113                 value.z = diedralAngle;
114             }
115         }
116
117         angles[anglesIndex].x = value.x;
118         angles[anglesIndex].y = value.y;
119         angles[anglesIndex].z = value.z;
120
121         angles[anglesIndex+1].x = value.x;
122         angles[anglesIndex+1].y = value.y;
123         angles[anglesIndex+1].z = value.z;
124
125         angles[anglesIndex+2].x = value.x;
126         angles[anglesIndex+2].y = value.y;
127         angles[anglesIndex+2].z = value.z;
128     }
129 }
```

Figura 8.6: Parte 2 del kernel del algoritmo Ángulo Diedro que calcula el ángulo diedro entre un triángulo y sus vecinos..

Capítulo 9

Bibliografía

[1] Callgrind. `http://valgrind.org/docs/manual/cl-manual.html`. Última visita: Octubre, 2016.

[2] RIVIX. `http://rivix.com/index.php`. Última visita: Octubre, 2015.

[3] Joaquín Gómez de Llarena. Crónica de historia natural. 31. terrazas fluviales, Munibe Ciencias Naturales, Volumen 7 fascículo I, 1955.

[4] Khronos OpenCL Working Group. The OpenCL Specification, 2012. Version 1.2.

[5] Ana Lucia Varbanescu y Henk Sips Jianbin Fang. *A Comprehensive Performance Comparison of CUDA and OpenCL*, 2011. Parallel and Distributed Systems Group Delft University of Technology Delft, the Netherlands.

[6] Diego Andrés Gajardo Jiménez. *Reconocimiento de patrones de drenaje y detección de terrazas fluviales sobre modelos de terreno de cuencas*, Tesis para optar al grado de Magister en Ciencias de la Computación, 2016.

[7] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors*. 2nd edition, 2012.

[8] M. Van Kreveld. *Digital Elevation Models: Overview and selected TIN Algorithms*, 1996. Department of Computer Science, Utrecht University, Netherlands.

[9] Michael Pidwirny. *Fundamentals of Physical Geography*. 2nd edition, 2006. `http://www.physicalgeography.net/fundamentals/contents.html`. Última visita: Agosto, 2016.

[10] Pillippa Ignacia Iris Pérez Pons. *Visualización de mallas de terreno e identificación de patrones de drenaje en cuencas*, Memoria para optar al grado de Ingeniero Civil en Computación, 2014.

[11] W. Randolph Franklin Salles V. G. Magalhães, Marcus V. A. Andrade and Guilherme C. Pena. *A new method for computing the drainage network based on raising the level of an ocean surrounding the terrain*, 2012. Department of Informatics (DPI) - Univ. Fed. Viçosa,

Viçosa, Brazil.

[12] Nicolás Silva. *Modelamiento del crecimiento de árboles usando mallas de superficie*, Memoria para optar al grado de Ingeniero Civil en Computación, 2007.

[13] Z. Ling y E. Guilbert. *Automatic drainage pattern recognition in river networks*, Department of Land Surveying and Geo-Informatics, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong, 2013.