ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería Marítima y Ciencias del Mar

"SIMULACIONES NUMÉRICAS DE FLUIDOS USANDO TARJETAS GRÁFICAS GPU"

INFORME DE PROYECTO INTEGRADOR

Previo la obtención del Título de:

INGENIERO NAVAL

Presentado por:

ANDRÉS JAVIER LÓPEZ PEÑAFIEL

GUAYAQUIL - ECUADOR

Año: 2019

DEDICATORIA

Este proyecto se lo dedico a mis padres María y Teodoro por todo el apoyo incondicional en todas las etapas de mi vida. A mi esposa Erika y mis hijas Ximena e Isabella por ser el motor de mi vida. A toda mi familia (hermanas, primas, etc.) por estar pendientes de mí y A mis excompañeros de aula los cuales me apoyaron indirectamente en este proyecto, en especial a Guillermo, José.

AGRADECIMIENTOS

Porque todas las cosas proceden de él, y existen por él y para él. ¡A él sea la gloria por siempre! Amén. (Romanos 11:36). A Dios por su gran amor y misericordia hacia mí en todo momento y A la Virgen María mi segunda Madre.

DECLARACIÓN EXPRESA

"Los derechos de titularidad y explotación, me (nos) corresponde conforme al reglamento de propiedad intelectual de la institución; Andrés Javier López Peñafiel doy (damos) mi (nuestro) consentimiento para que la ESPOL realice la comunicación pública de la obra por cualquier medio con el fin de promover la consulta, difusión y uso público de la producción intelectual"

Andrés Javier López Peñafiel

EVALUADORES

Nadia Muñoz, Ing.

COORDINADORA DE LA CARRERA

Rubén Paredes, PhD.

PROFESOR TUTOR

RESUMEN

En este trabajo se realizó la simulación de un fluido estático, empleando el método lagrangiano de partículas suavizadas **S**moothed **P**article **H**ydrodynamics (SPH) implementado en lenguaje CUDA. En el desarrollo de este proyecto, se modificó el código fuente del tutorial “Particles” (Green, 2010) y distribuido en el Developer Toolkit de NVIDIA CORPORATION.

El tutorial “Particles” simula la interacción entre partículas asumiendo que están conectadas con resortes. Por lo que para alcanzar el objetivo de este trabajo fue necesario calcular la densidad de cada partícula y agregar subrutinas que estimen las fuerzas de interacción entre partículas usando la función Kernel típica en SPH.

Finalmente se obtuvieron estimaciones preliminares de densidad que varían en función de la profundidad 1000 $\frac{Kg}{m^3}$ y 1002 $\frac{Kg}{m^3}$. Los valores de presión obtenidos por medio de una ecuación de estado estaban entre 1000$\frac{N}{m^2}$ y 23000 $\frac{N}{m^2}$ que varían en función de la profundidad.

Palabras Clave: Fluido Lagrangiano, SPH, CUDA, Particles.

ABSTRACT

In this work, we conducted a simulation of a static fluid, using the method of Smoothed Particle Lagrangian Smoothed Particle Hydrodynamics (SPH) implemented in CUDA language. In the development of this project, modified the source code of the tutorial "particles" (Green, 2010) and distributed in the Developer Toolkit of NVIDIA Corporation.

The tutorial "Particles" simulates the interactions between particles assuming that are connected with springs. Therefore, in order to achieve the objective of this work was necessary to calculate the density of each particle and add subroutines that deem the forces of interaction between particles using the Kernel function typical of SPH.

Finally, we obtained preliminary estimates of population density that vary depending on the depth of 1000 $\frac{Kg}{m^3}$ *and 1002* $\frac{Kg}{m^3}$ *. The pressure values obtained by means of an equation of state were among 1000* $\frac{N}{m^2}$ *and 23000* $\frac{N}{m^2}$ *which vary depending on the depth.*

Keywords: *Lagrangian Fluid, SPH, GPU, Meshless Method*

ÍNDICE GENERAL

EVALUADORES....5
RESUMEN....I
ABSTRACT....II
ÍNDICE GENERAL....III
ABREVIATURAS....VI
SIMBOLOGÍA....VII
ÍNDICE DE FIGURAS....VIII
ÍNDICE DE TABLAS....X
CAPÍTULO 1....11
1. INTRODUCCIÓN....11
1.1. Descripción del problema....12
1.2. Justificación del problema....12
1.3. Objetivos....12
1.3.1. Objetivo General....12
1.3.2. Objetivos Específicos....12
1.4. Marco teórico....13
1.4.1. Tarjetas Gráficas GPU, antecedentes.....13
1.4.2. Características de las tarjetas gráficas GPU....14
1.4.3. Programación Paralela....16
1.4.4. Lenguaje de programación de las tarjetas gráficas GPU....17
1.5. Potenciales aplicaciones de las tarjetas Gráficas GPU.....19
CAPÍTULO 2....21
2. METODOLOGÍA....21
2.1. Descripción Lagrangiana de Fluidos....21
2.2. Método SPH....22
2.2.1. Función Kernel $\boldsymbol{W(ri - rj, h)}$....24

2.2.2. Ecuación de Estado....27
2.3. Tutorial de Partículas “Particles” 27
2.3.1. La Integración....30
2.3.2. Construcción de Red de Datos....32
2.3.3. Proceso de Colisión....33
2.4. Implementación de SPH en CUDA....34
2.4.1. Parámetros Iniciales....36
2.4.2. Densidad entre partículas....37
2.4.3. Densidad Cell....39
2.4.4. Densidad_Update....41
2.4.5. CollideSPH....43
CAPÍTULO 3....47
3. RESULTADOS Y ANALISIS....47
3.1. Equipo empleado en el desarrollo del proyecto....47
3.2. Modificación del tutorial Particles....47
3.2.1. Distribución Inicial de Partículas.....47
3.2.2. Modificación de la Interfaz gráfica....49
3.2.3. Opciones de menú principal.....50
3.2.4. Cubo de colisión.....50
3.3. Cálculo de la masa inicial del sistema....52
3.4. Cálculo de las densidades....55
3.5. Cálculo de las Presiones....56
CAPÍTULO 4....57
4. CONCLUSIONES Y RECOMENDACIONES....57
CONCLUSIONES....57
RECOMENDACIONES....58
BIBLIOGRAFÍA....59
ANEXOS....61

ABREVIATURAS

CUDA Compute Unified Device Architecture
SPH Smoothed Particle hydrodynamics
CPU Central Processing Unit
GPU Graphics Processing Unit
MDA Monochrome Display Adapter
OpenGL Open Graphics Library
FLOP Floating Point Operations

SIMBOLOGÍA

M	Masa
P	Presión
A	Aceleración
V	Velocidad
Z	Altura
ρ	Densidad
W	Kernel
α	Alfa
Vol	Volumen
R	Distancia relativa
$f_{i.s}$	Fuerza de resorte
$f_{i.d}$	Fuerza de amortiguación
V_{ij}	Velocidad relativa de las partículas

ÍNDICE DE FIGURAS

Figura 1.1 Procesador Gráfico Desarrollado por NVIDIA. 13
Figura 1.2 Estructura de la GPU.....14
Figura 1.3 Comparación GPU vs CPU. 15
Figura 1.4 Tarjeta Gráfica GEFORCE GT 740 (izquierda) y NVIDIA TITAN V (derecha) Desarrollada por NVIDIA Corporation. 15
Figura 1.5 Configuración de CPU y GPU. 16
Figura 1.6 Programación en C y CUDA.....17
Figura 1.7 Configuración de Memoria en CUDA en una GPU (Cámara, Represa, & Sánchez, 2016) 18
Figura 1.8 Simulación de Erupción volcánica y Ruptura de presa.....19
Figura 1.9 Simulación de Amarre. 20
Figura 2.1 Representación de un Fluido Lagrangiano.....21
Figura 2.2 Movimiento de una Partícula en un Fluido Lagrangiano.....21
Figura 2.3 Representación del Método SPH. 23
Figura 2.4 Diagrama de Flujo del método SPH 24
Figura 2.5 Función Kernel de (Lucy, 1977).....26
Figura 2.6 Diagrama de Flujo del Tutorial Particles.....28
Figura 2.7 Captura de Pantalla de la Simulación Particles.....30
Figura 2.8 Método de Euler para la Resolución de EDO.....31
Figura 2.9 Método de Clasificación de las Partículas en el Tutorial Particles.....32
Figura 2.10 Actualización de Velocidades y Posiciones del Tutorial Particles.....34
Figura 2.11 Implementación del Método SPH en CUDA. 36
Figura 2.12 Flujograma de la Función Densidad entre Partículas. 38
Figura 2.13 Implementación de la Subrutina Densidad entre Partículas. 39
Figura 2.14 Enmallado del sistema (García Garino, 2013).....39
Figura 2.15 Flujograma de la Función Densidad cell.....40
Figura 2.16 Implementación de la Subrutina Densidad cell.....41
Figura 2.17 Flujograma de la Función Densidad_Update. 42
Figura 2.18 Implementación de la Subrutina Densidad_Update.....43
Figura 2.19 Flujograma de la función CollideSPH 45
Figura 2.20 Implementación de la Subrutina para el Cálculo de la Presión. 46

Figura 3.1 Función initGrid modificada 48
Figura 3.2 Configuración de Color de Cada Pixel (Cámara, Represa, & Sánchez, 2016) 49
Figura 3.3 Datos modificados para obtener la tonalidad del proyecto 50
Figura 3.4 Funciones del Cubo de Colisión 51
Figura 3.5 Resultado de las Modificaciones del Tutorial Particles. 51
Figura 3.6 Modelado del Sistema 51
Figura 3.7 Vista Frontal del Sistema 52
Figura 3.8 Distribución de presión Hidrostática de las Partículas. 53
Figura 3.9 Subrutina del cálculo de las masas iniciales 53
Figura 3.10 Valores Obtenidos del Programa Modificado. 54
Figura 3.11 Resultados del valor de la Densidad 55
Figura 3.12 Valores de Densidad Almacenados 56

ÍNDICE DE TABLAS

Tabla 1-1 Características de las primeras Tarjetas Gráficas Según su Serie 14
Tabla 1-2 Especificadores para funciones.....18
Tabla 1-3 Aplicaciones de las Tarjetas Graficas (NVIDIA, 2018) 19
Tabla 2-1. Tipos y Características de Funciones Kernel. 25
Tabla 2-2 Principales Códigos del Tutorial. 29
Tabla 2-3 Parámetros Iniciales del Tutorial 33
Tabla 2-4 Valores iniciales de la función CollideSpheres 43
Tabla 2-5 Nuevos parámetros de CollideSpheres 44
Tabla 2-6 Valores Eliminados del Tutorial inicial 44
Tabla 3-1 Características del Computador Usado.....47
Tabla 3-2 Datos ingresados para el calculo 48
Tabla 3-3 Menú Principal.....50
Tabla 3-4 Valores Iniciales Mediante Presión Hidrostática.....54
Tabla 3-5 Comparación de Valores Masa 54
Tabla 3-6 Comparación de Valores Densidad 55
Tabla 3-7 Comparación de Valores de Presión 56

CAPÍTULO 1

1. INTRODUCCIÓN

En la actualidad, la ciencia computacional se combina con el área de la ingeniería para modelar problemas físicos complejos y permite optimizar el proceso de diseño, principalmente porque los resultados numéricos son considerados por la industria tan confiable como aproximaciones o experimentales. Esto ha sido posible gracias al incremento de su capacidad de cálculo, que ha permitido acelerar el progreso tecnológico impulsado por la comunidad científica. Por ejemplo, la última generación de computadores permite realizar simulaciones de fenómenos físicos en tiempo real debido a que tiene una capacidad de FLOPS. Los FLOPS (**F**loating **P**oint **O**perations per **S**econd) son usados como operaciones por segundo y sirven para medir la velocidad de procesamiento de los procesadores(CPUs) y tarjetas gráficas(GPUs).

Una de las alternativas para aumentar la velocidad de procesamiento, es de usar varios CPUs con la posibilidad de ejecutar procesos en paralelo permitiendo el incremento de capacidad computacional como los clúster o supercomputadores. Hace pocos años, este tipo de supercomputadores estaba restringido a centros de investigación, lo que requería de altas inversiones monetarias para su adquisición y mantenimiento. Un Clúster, es el uso compuesto de un conjunto de ordenadores conectados entre sí normalmente por una red local de alta velocidad y un sistema (software) base administrador de recursos; para distribuir la carga de trabajo.

Tradicionalmente, estos clúster utilizan varias Unidades Centrales de Procesamiento (CPU) con varios núcleos, pero recientemente se incorporó la posibilidad de realizar cálculos numéricos en procesadores gráficos (GPU). Los GPUs fueron creados inicialmente para los videojuegos y paulatinamente fue aceptada en la industria cinematográfica y de animaciones tridimensionales. En las últimas 2 décadas, las GPUs empezaron a ser usadas en el campo científico con el desarrollo y resolución de simulaciones de fenómenos físicos de todo tipo, como por ejemplo mareas, nieve, viento y electricidad.

1.1. Descripción del problema

En la Industria local existe la necesidad de fortalecer las capacidades tecnológicas para realizar simulaciones numéricas de fenómenos físicos complejos de interés del área de todas las ingenierías considerando las limitaciones. Lo cual el uso de GPUs se mueve atractivo, es necesario el uso de una plataforma de computación adecuada. En la industria marítima permitirá el estudio de fluidos y el análisis del comportamiento de estructuras flotantes considerando el medio marino donde operan. Como primer paso, el presente trabajo explorará el potencial de uso de las GPUS en simulaciones de fluidos.

1.2. Justificación del problema

En nuestra Región, se acostumbra que, debido a las limitaciones tecnológicas, los estudios de ingeniería de proyectos de gran escala sean realizados por consultoras extranjeras produciendo fuga de divisas, lo que aumenta su costo y no fortalece nuestro medio. Además es posible que se ignore por completo este tipo de estudios, y se pasa del diseño conceptual a la construcción, Aumentando el riesgo de accidentes y desperdicios de recursos.

1.3. Objetivos

1.3.1. Objetivo General

Simular un fluido newtoniano utilizando un método numérico Lagrangiano implementado en tarjetas gráficas GPU.

1.3.2. Objetivos Específicos

- Comprender el lenguaje de programación CUDA (***C**ompute **U**nified **D**evice **A**rchitecture)* usado para implementar cálculos numéricos en tarjetas gráficas.
- Comprender el método **S**moothed **P**article **H**ydrodynamics (SPH) a implementarse en Arquitectura Unificada de Dispositivos de Cómputo “CUDA”.
- Implementar el método SPH (**S**moothed **P**article **H**ydrodynamics) dentro del tutorial “Particles” de NVIDIA.

1.4. Marco teórico

1.4.1. Tarjetas Gráficas GPU, antecedentes.

En 1981, IBM desarrolla la primera tarjeta gráfica conocida como MDA (**M**onochrome **D**isplay **A**dapter) que contaba con una capacidad de memoria de 4Kb, y era utilizada para manejar monitores monocromáticos. Posteriormente, IBM desarrolló de forma progresiva varias tarjetas con diferentes avances tecnológicos como el de la resolución, el número de colores y hasta se alcanzaron 2MB de memoria de video en 1999. Ver figura 1.1

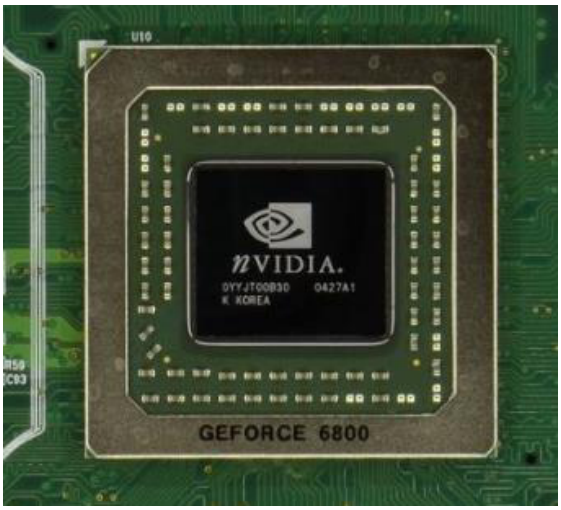


Figura 1.1 Procesador Gráfico Desarrollado por NVIDIA.
FUENTE: (NVIDIA, 2017)

En ese año, NVIDIA Corporation lanzó la tarjeta GeForce 256 que tenía incorporado una tarjeta de video, el cual permitía realizar cálculos numéricos y ejecutar algoritmos. La velocidad y potencia de procesamiento de la GPU se media en el número de pixeles o bien la tasa de dibujo de polígonos también medida en millones por segundo. La GeForce 256 era capaz de procesar 10 millones de polígonos por segundo, en la tabla 1-1 se muestra las primeras tarjetas gráficas de cada serie fabricadas por NVIDIA. A lo largo de los años, estas GPUS han evolucionado hasta alcanzar 2000 millones de polígonos por segundo.

GPU	FPS	Cores	Almacenamiento	Lanzamiento
GeForce 256	960 FLOPS	------	32 MB	1999
Quadro FX 330	1280 FLOPS	------	64 MB	2004
Tesla C870	430 GFLOPS	128	1.5 GB	2007
GeForce Mobile GTX 260	476 GFLOPS	192	896 MB	2008

Tabla 1-1 Características de las primeras Tarjetas Gráficas Según su Serie

Posteriormente INTEL & AMD ingresaron a la industria de ensamblaje de tarjetas GPU intentando captar una cuota del mercado. Debido a su arquitectura diferente, fue necesario el desarrollo de un lenguaje de programación para explotar el potencial de los GPUS, NVIDIA desarrolló CUDA.

1.4.2. Características de las tarjetas gráficas GPU

Las tarjetas gráficas GPU estan Conformadas por núcleos (cores), cada núcleo CUDA o Core CUDA es como un mini procesador que se encarga de realizar instrucciones en forma paralela. Un Hilo o threads es una secuencia de tareas encadenadas muy pequeñas que son ejecutadas por un sistema operativo. En la figura 1.2 se muestra la estructura de la GPU. Además, los núcleos son los encargados de administrar la informacion gráfica a traves de un conjunto de instrucciones basadas en operaciones aritméticas.

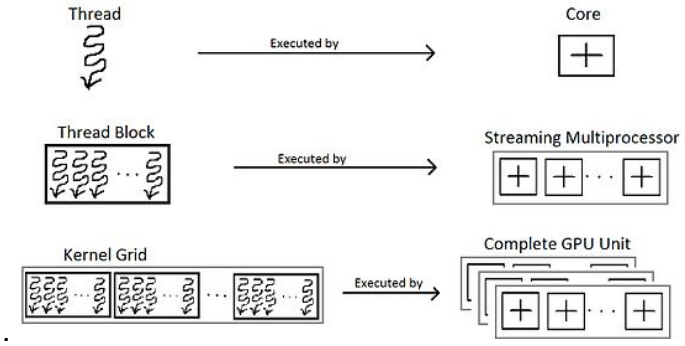


Figura 1.2 Estructura de la GPU.

FUENTE: (IBM, 2019)

Los nucleos de las GPUs paralelisan la información por medio de los hilos (threads) enviando ordenes simultaneas para que realicen los cálculos en un menor tiempo. la figura 1.3 muestra la comparación entre anchos de bandas de memorias (Memory Bandwidth) GPU y CPU medidas en GB/s (Giga Byte per Second).

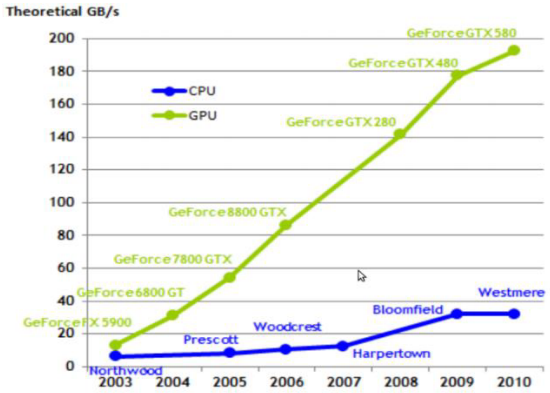


Figura 1.3 Comparación GPU vs CPU.

FUENTE: (Garcia, 2010)

Una ventaja que poseen las GPUs con respecto a las CPUs tradicionales es el precio. Es posible pagar $250 por un modelo basico GeForce GT740, la cual posee una velocidad de computo de 0,76 TFLOPS, 384 nucleos y hasta $4000 por la mas ávanzada NVIDIA TITAN V que posee una velocidad de procesamiento de 110 Teraflops y una memoria 3D de 12GB y 700 cores, ver figura 1.4.

Figura 1.4 Tarjeta Gráfica GEFORCE GT 740 (izquierda) y NVIDIA TITAN V (derecha) Desarrollada por NVIDIA Corporation.

FUENTE: (NVIDIA, 2019)

1.4.3. Programación Paralela

Programación paralela es una forma de computación en la cual realizan varios cálculos simultáneamente utilizando un conjunto de procesadores capaces de trabajar en sincronía. Divide un problema grande en varios problemas pequeños, que son posteriormente solucionados de manera independiente.

La programacion paralela nace como resultado de la necesidad de resolver problemas complejos que requieren gran capacidad de memoria y cantidad de iteraciones o cálculos. En la figura 2.5 se muestra las configuraciones de las CPU y GPU. Donde **ALU** es un contador digital que realiza operaciones aritméticas y lógicas. **CACHE** sirve para reducir el tiempo de acceso de memoria. La función de **CONTROL** es de buscar, decodificar y ejecutar las instrucciones en la memoria principal. Por ultimo **DRAM** carga todas las instrucciones que ejecuta el procesador.

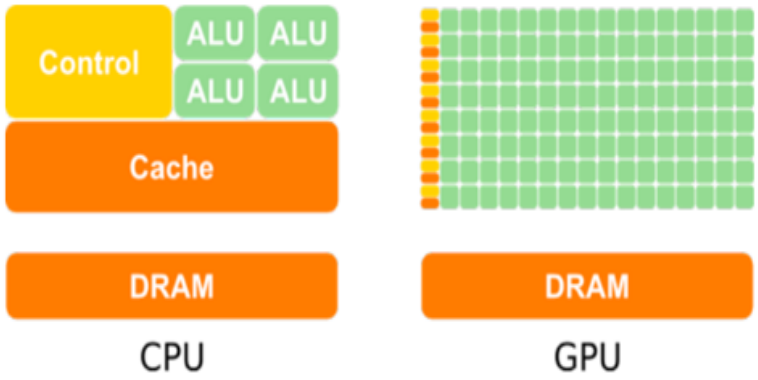


Figura 1.5 Configuración de CPU y GPU.
FUENTE: (Ye, 2015)

Las ventajas que posee el paralelismo son:

- Brinda a usuarios en general el beneficio de mayor velocidad de computo.
- Permite dividir un problema macro en partes independientes.
- Permite ejecutar problemas de mayor complejidad reduciendo la incertidumbre de la respuesta obtenidad.

Entre las desventajas que maneja el paralelismo tenemos:

- Mayor dificultad a la hora de escribir programas.

- Gran consumo de energía por las arquitecturas de múltiples núcleos.
- Inversion adicional en el sistema de enfriamiento de las GPU.

1.4.4. Lenguaje de programación de las tarjetas gráficas GPU

NVIDIA Corporation desarrolló la plataforma de cálculo paralelo CUDA (***C**ompute **U**nified **D**evice **A**rchitecture)*, que incluye un compilador NVCC y un conjunto de herramientas que permiten ejecutar instrucciones en las GPUs. Su programación está basada en lenguaje de programación C, C++ y mediante "wrappers" o adaptadores que permiten a los programadores usar lenguajes como Java, Fortan, OpenGL y Direct3D. En la figura 1.6 se muestra la diferencia del código fuente de la suma de 2 arreglos x[i] y y[i]. En el caso del lenguaje C el proceso es serial pero en CUDA se ejecuta de forma paralela donde cada nodo del CORE ejecuta de forma aleatoria un elemento i del arreglo.

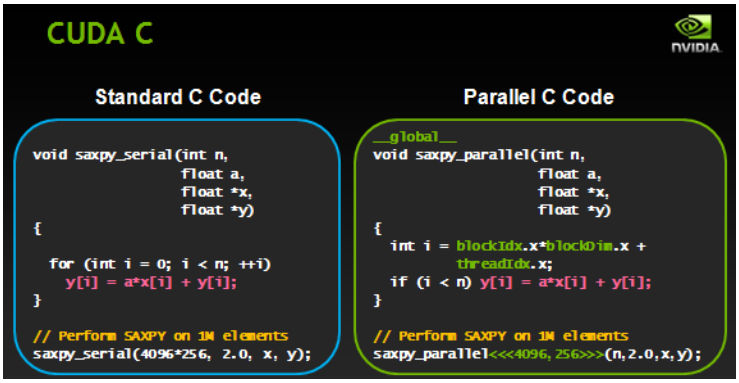


Figura 1.6 Programación en C y CUDA.

FUENTE: (NVIDIA, 2014)

La función define los valores de entrada *int* n, *int* i, float a, donde n e i son valores enteros y a es un numero decimal. Además se definen los arreglos float x[i], float y[i], la suma de los arreglos se realiza siempre y cuando se cumpla la condición de memoria, la cual comienza de cero hasta menor que n en ambos casos.

Entre las ventajas que tiene el uso del lenguaje CUDA tenemos las siguientes:

- Memorias compartidas.
- Lecturas más rápidas de y hacia la GPU.
- Lecturas dispersas (se puede consultar cualquier posición de memoria).

- Soporte para enteros y operadores a nivel de bit.

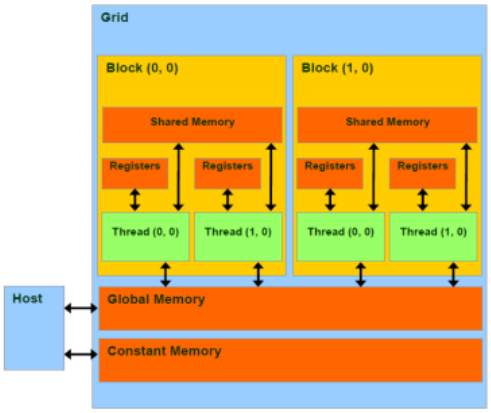


Figura 1.7 Configuración de Memoria en CUDA en una GPU (Cámara, Represa, & Sánchez, 2016)

El modelo de programación CUDA asume un sistema compuesto por un ***host*** (CPU) y un ***device*** (GPU), y cada uno de ellos con sus propios espacios de memoria separados. La única zona de memoria accesible desde el ***host*** es la memoria **global**. Además, tanto la reserva o liberación de memoria global como la transferencia de datos entre el host y el device debe hacerse de forma explícita desde el host mediante funciones específicas de CUDA llamadas kernel. En la figura 1.7 se observa la Jerarquía de la memoria. En la tabla 1-2, se muestra los diferentes tipos de especificadores para funciones usadas como kernels CUDA.

Especificador	Llamada desde:	Ejecutada en:
__device__ float DeviceFunc()	device	device
__global__ void kernelFunc()	host	device
__host__ float HostFunc()	host	host

Tabla 1-2 Especificadores para funciones

Las funciones *__device__* y *__host__* tienen como particularidad el devolver datos al regresar desde la llamada (a través de la sentencia return), el especificador *__host__* se emplea para las funciones de llamada y ejecutadas en el CPU. Por otro lado una función que se llama desde la CPU (host) y ejecutada en la GPU (device) se declara como *__global__*.

Retomando el ejemplo en la figura 1.5 tenemos que en el código CUDA describe la función kernel como *__global__*. Es decir se emplea una sintaxis para los índices e hilos de la GPU “int i = blockDim.x * blockIdx.x + threadIdx.x; “. Por ultimo ejecuta el kernel <<4096, 256>> (n, 2.0, x, y), la cual ejecuta 4096 bloques x 256 thread/block con los valores iniciales n, 2, x, y.

1.5. Potenciales aplicaciones de las tarjetas Gráficas GPU.

Las tarjetas gráficas permiten el desarrollo de simulaciones en tiempo real, que son usadas por la industria para comprender fenómenos físicos, que carecen de una solución analítica. En la tabla 1-3 se muestra las aplicaciones más relevantes que han sido implementados usando la plataforma CUDA de NVIDIA hasta la actualidad.

Industria	Aplicaciones
Animación y modelaje	Juegos, Industria cinematográfica.
Astronomía y Astrofísica	Explosión de supernovas.
Bioinformática	Salud y ciencias de la vida.
Clima y Modelado del océano	Fines Académicos y Gubernamentales.
Aeroespacial	CFD, turbulencia en fluidos.

Tabla 1-3 Aplicaciones de las Tarjetas Graficas (NVIDIA, 2018)

En el Ecuador, su uso no estaría limitado a la industria naval sino podría implementarse para la evaluación de fenómenos físicos como erupciones volcánicas o fallas estructurales de represas hidroeléctricas, ver figura 1.8. Esto permitirá preparar planes de mitigación en los pueblos afectados reduciendo el impacto económico el país.

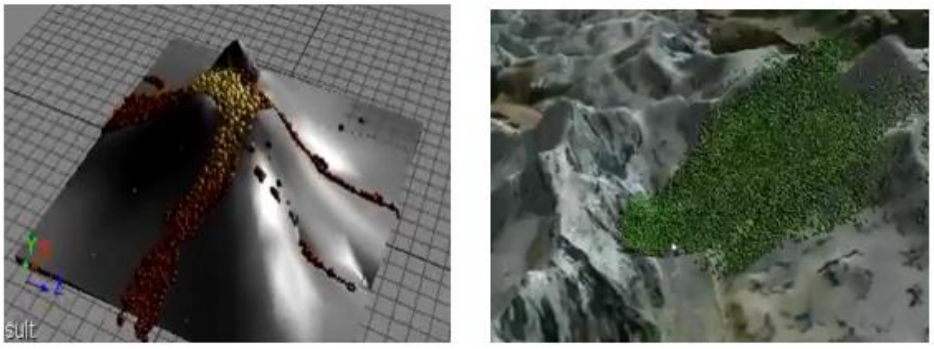


Figura 1.8 Simulación de Erupción volcánica y Ruptura de presa.
Fuente: (Fang, 2012)

En nuestra industria naval, se podría utilizar para evaluar el comportamiento dinámico de embarcaciones navegando en los diferentes estados de mar del Ecuador o analizar estructuras costeras, ver figura 1.8.

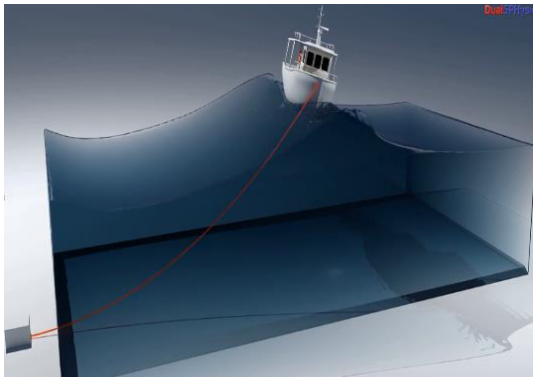


Figura 1.9 Simulación de Amarre.
Fuente: (DualSPHysics, 2015)

CAPÍTULO 2

2. METODOLOGÍA

2.1. Descripción Lagrangiana de Fluidos

La característica principal de esta descripción es que divide el fluido físico en partículas de masa constante y se sigue a cada una de ellas en el tiempo, de acuerdo a las ecuaciones gobernantes que son aproximadas usando un kernel, como se muestra en la figura 2.1. Luego de distribuir de forma uniforme las partículas en el dominio físico se aproximan las fuerzas de interacción entre partículas asumiendo que el volumen de cada partícula es Δx^3. La separación promedio entre partículas es el diámetro Δx.

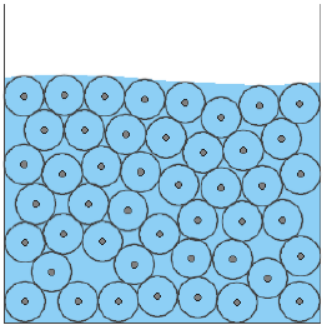


Figura 2.1 Representación de un Fluido Lagrangiano.
FUENTE: Kelager, 2006, figura 4.1, pág. 14

Luego, se calcula la aceleración de cada una de las partículas a partir de las ecuaciones gobernantes (2.1), (2.2) y (2.3). Posteriormente, se calcula la velocidad y la posición. En resumen, el movimiento de una partícula se muestra en la figura 2.2.

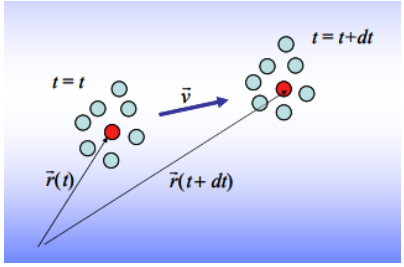


Figura 2.2 Movimiento de una Partícula en un Fluido Lagrangiano.
FUENTE: (Tamburrino, 2008)

$$\rho\left(\frac{\partial u_i}{\partial_t} + u \cdot \nabla u_i\right) = -\frac{\partial_p}{\partial x_i} + \upsilon \Delta u_i + f_\varepsilon^i \quad (2.1)$$

$$\nabla \cdot u = 0 \quad (2.2)$$

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = 0 \quad (2.3)$$

Donde υ es la viscosidad cinemática del fluido, ρ la densidad del fluido, f son las fuerzas externas que actúan sobre el fluido por ejemplo a gravedad.

2.2. Método SPH

Este método fue propuesto inicialmente por (Lucy, Monaghan & Gingold, 1977) para el estudio de astrofísica. Es un método Lagrangiano, que se puede ajustar fácilmente para la resolución de variables como la densidad o superficies libres.
El método **S**moothed **P**articles **H**ydrodynamics (SPH) divide el fluido en un conjunto de elementos discretos, a los cuales se refiere como partículas y permite resolver las ecuaciones de Navier-Stokes para el estudio de flujos de fluidos.

SPH es básicamente un método de interpolación, que usa una función kernel que se aproxima a una función delta dirac cuando la separación entre partículas tiende a cero. En principio, permite aproximar cualquier función $A(R)$ con una función de aproximación denominada kernel, como se muestra en (2.4).

$$A(R) = \int_\Omega A(r')W\left(r_i - r_j, h\right)dr' \quad (2.4)$$

Donde $\vec{r}$ es el vector posición, W es la función de aproximación o kernel, h es la longitud del dominio Ω aproximado, el cual debe ser mayor que la separación inicial de las partículas. Las partículas fuera del dominio Ω, es decir $\left(r_i - r_j > h\right)$ no son consideradas de $A(R)$ en el cálculo.

Las Ecuaciones gobernantes (2.1, 2.2, 2,3) son reescritas usando (2.4) y se obtiene un sistema de ecuaciones algebraicas que permite aproximar la velocidad y densidad de todas las partículas. Inicialmente cada partícula i tiene una posición $\vec{r_i}$, una velocidad

$\vec{v}_i$ y una densidad ρ_i. La partícula i interactúa con el conjunto de n partículas j dentro del radio h con respecto a i, ver figura 2.3. Para calcular la densidad de cada partícula se utiliza la ecuación (2.5).

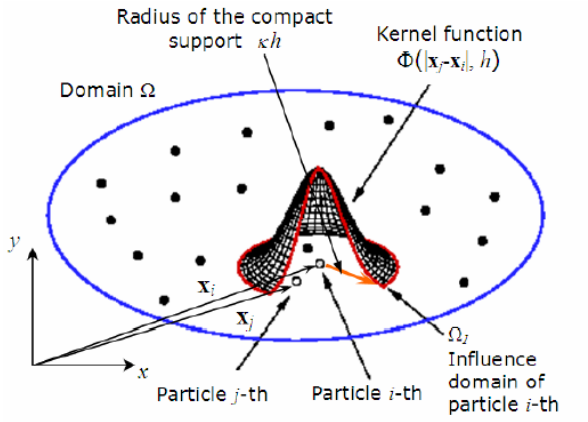


Figura 2.3 Representación del Método SPH.
FUENTE: Maneti, 2009, figura 1, pág. 24

$$\rho_i = \rho(r_i) \tag{2.5}$$

$$\rho_i = \sum_{j=1}^{N} m_j W\left(|r_i - r_j|, h\right)$$

$$\rho_i = \sum_{j=1}^{N} m_j W_{ij}$$

Donde m_j es la masa de las partículas vecinas, $(r_i - r_j)$ es la distancia entre partículas, h es el radio de la función y W es la función kernel evaluado en $|(r_i - r_j), h|$. En la figura 2.4 se muestra el diagrama de flujo propuesto para el método SPH desarrollado por (Huang & Dai, 2013).

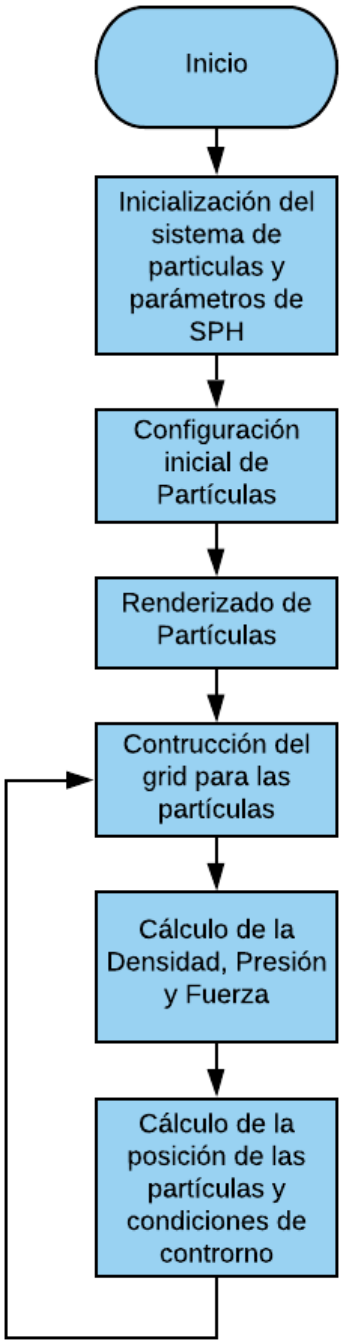


Figura 2.4 Diagrama de Flujo del método SPH

2.2.1. Función Kernel $W(r_i - r_j, h)$

Es una función que permite interpolar los valores de cualquier propiedad del fluido en función del valor de las partículas del entorno. En la literatura, se han propuesto varias

funciones kernel entre ellas están: Cuadrática, Spline cubico, Gaussiana, Quintico (Garcia C. , 2012). Como se muestra en la tabla se da una breve descripción.

Kernel	$\boldsymbol{\alpha_D}$ $(\boldsymbol{2D})$	$\boldsymbol{\alpha_D}$ $(\boldsymbol{3D})$	**W(r,h)**	**Rango**
Gaussiana	$\frac{1}{(\pi h^2)}$	$\frac{1}{(\pi^{3/2} h^3)}$	$\alpha_D . e^{-q^2}$	$0 \leq q \leq 2$
Cuadrática	$\frac{2}{(\pi h^2)}$	$\frac{5}{(\pi h^3)}$	$\alpha_D \left[\frac{3}{16} q^2 - \frac{3}{4} q\right]$	$0 \leq q \leq 2$
Spline Cubico	$\frac{10}{(7\pi h^2)}$	$\frac{1}{(\pi h^3)}$	$\alpha_D \begin{cases} 1 - \frac{3}{2} q^2 + \frac{3}{4} q^3 \\ \frac{1}{4}(2-q)^3 \\ 0 \end{cases}$	$0 \leq q \leq 1$ $1 \leq q \leq 2$ $q \geq 2$
Quintic	$\frac{7}{(4\pi h^2)}$	$\frac{7}{(8\pi h^3)}$	$\alpha_D \left(1 - \frac{q}{2}\right)^4 (2q+1)$	$0 \leq q \leq 2$
Lucy	$\frac{5}{(\pi h^2)}$	$\frac{105}{16\pi h^3}$	$\alpha_D \begin{cases} (1+3q)(1-q)^3 \\ 0 \end{cases}$	$q \leq 1$ $q \geq 1$

Tabla 2-1. Tipos y Características de Funciones Kernel.

Donde $q = \frac{r}{h}$

r: $|r_i - r_j|$ = Es la distancia entre las partículas i y j.

$\alpha_D =$ Es un Factor dimensional, si es 2D abarca un circulo y 3D una esfera.

Además, las funciones deben cumplir las siguientes condiciones:

- Positiva dentro del Dominio, es decir $W(r - r', h) \geq 0$
- Debe ser normalizada $\int W(r - r', h) dr' = 1$
- Soporte Compacto fuera $W(r - r', h) = 0$ del dominio.
- Comportamiento monotónicamente decreciente de $W(r - r', h)$
- Comportamiento de la función delta: $\lim_{h \to \infty} W(r - r', h) dr' = \delta(r - r')$

Para el desarrollo del presente proyecto se usó la función Kernel polinómica de grado 4, la cual su derivada es más accesible, la función kernel Quartic de (Lucy, 1977) cumple esta condición, mostrada en (2.6).

$$W(r,h) = \alpha_d \begin{Bmatrix} (1+3q)(1-q)^3 & | q \leq 1 \\ 0 & | q > 1 \end{Bmatrix} \tag{2.6}$$

El gradiente de la funcion kernel es su primera derivada, como se muestra en (2.7).

$$\nabla_i W_{ij} = \alpha_d \begin{Bmatrix} \frac{\partial W_{ij}}{\partial X_i} \\ \frac{\partial W_{ij}}{\partial Y_i} \\ \frac{\partial W_{ij}}{\partial Z_i} \end{Bmatrix} = \alpha_d \frac{\partial W_{ij}}{\partial q} \nabla_i q = \frac{\partial W_{ij}}{\partial q} * \frac{X_i - X_j}{|X_i - X_j| h}$$

$$\frac{\partial W_{ij}}{\partial q} = \alpha_d \begin{Bmatrix} 3(1-q)^3 - 3(1+3q)(1-q)^2 & ; q \leq 1 \\ 0 & ; q > 1 \end{Bmatrix} \tag{2.7}$$

La figura 2.5 muestra la función kernel y su primera derivada

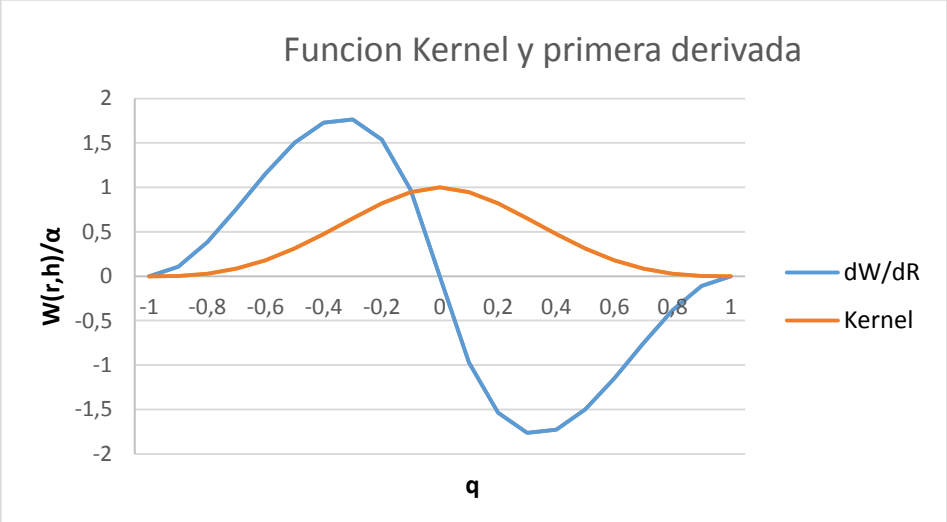


Figura 2.5 Función Kernel de (Lucy, 1977).

2.2.2. Ecuación de Estado

Mediante la formulación de la ecuación de estado de los gases (Nuli & Kulkarni, 2012) se obtiene el valor de las presiones de cada una de las partículas en función de su densidad aproximada usando la ecuación (2.5). La formulación consiste en:

$$P_i = C_s^2(\rho_i - \rho_0) \tag{2.8}$$

Donde tenemos:

P_i : Presión ejercida sobre la Partícula i

C_s : Velocidad artificial del sonido 100 $\frac{m}{s}$

ρ_i: Densidad de la partícula i.

ρ_0: Densidad de referencia del agua 1000 $\frac{kg}{m^3}$

2.3. Tutorial de Partículas “Particles”

Este tutorial fue desarrollado por Green (2010) y distribuido por la empresa NVIDIA, incluido en el CUDA Tool Kit como uno de los ejemplos en la carpeta de simulaciones. El código fuente permite modelar la interacción entre partículas dentro de un cubo unitario, y calcula las fuerzas de contacto entre partículas considerando que están conectadas con un resorte con amortiguamiento y muestra el resultado vía interfaz gráfica en tiempo real.

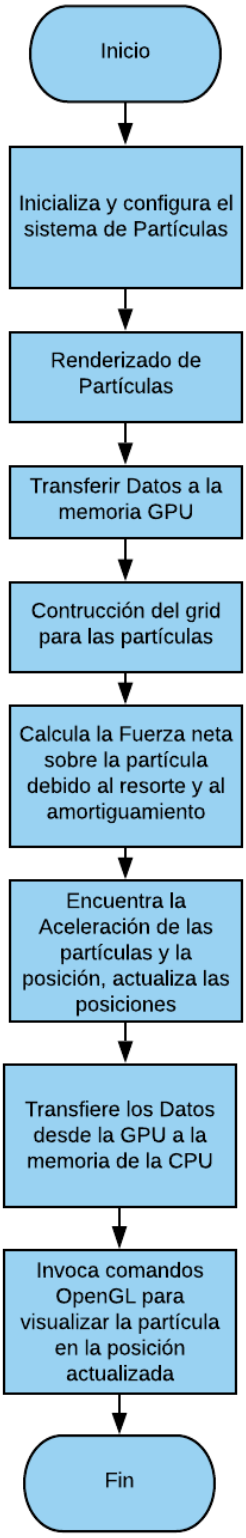


Figura 2.6 Diagrama de Flujo del Tutorial Particles

En la figura 2.6 se presenta el diagrama de flujo del tutorial particles y se lo detalla continuación:

1. Inicializa la simulación y configura el sistema de partículas.
2. Realiza el renderizado de las partículas.
3. Almacena los arreglos velocidad y posición en la memoria de la GPU.
4. Desarrolla el enmallado para las partículas, aplicando el método “sorting”.
5. Calcula la fuerza neta sobre las partículas debido al spring y damping.
6. Encuentra la aceleración, posición de las partículas y las almacena en la GPU.
7. Transfiere los valores de la GPU a la memoria de la CPU.
8. Empleando comandos OpenGL muestra las partículas actualizadas.

El tutorial tiene 7 archivos importantes donde se declaran funciones, parámetros, subrutinas, y un programa principal que realiza la llamada y ejecución del programa. En la tabla 2-2 se identifica cada uno y se describe la función en la simulación.

Elemento	Extensión	Descripción
particles	cpp	Ejecuta la simulación llamando a las demás subrutinas.
particles_kernel	cuh	Define los parámetros de simulación
particles_kernel_impl	cuh	Define los Kernel del sistema de partículas
particleSystem	cpp	Almacena los arreglos posición y velocidad de la simulación en la GPU
particleSystem	cuh	Define los métodos de cálculo para la simulación
particleSystem	h	Asigna parámetros a la CPU, GPU y el enmallado
particleSystem_cuda	cu	Contiene las funciones kernel que se ejecutan desde “particleSystem.cpp”

Tabla 2-2 Principales Códigos del Tutorial.

Las extensiones de cada archivo CUDA nos indica:

- **h, cpp, c, hpp, inc**: Son archivos que no contienen el código CUDA C (p. ej., __ device __ y otras palabras clave, llamadas de kernel, etc.), no realizan llamadas en tiempo de ejecución cuda (funciones cuda).
- **cu**: Son archivos fuente de CUDA C. Estos archivos se pasan al compilador NVCC para compilarse en objetos vinculables (host / dispositivo).

- **cuh, cuinc**: Son archivos que se incluyen en archivos .cu. Estos archivos pueden tener palabras clave CUDA C y / o llamar a funciones de tiempo de ejecución CUDA.

En resumen, el algoritmo implementado se basa en 3 pasos principales:

- La integración.
- Construcción de red de datos.
- El proceso de colisión.

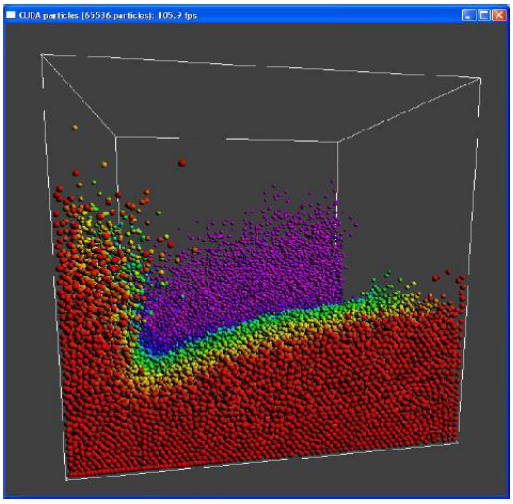


Figura 2.7 Captura de Pantalla de la Simulación Particles.
FUENTE: Green, 2013, figura 1, pág. 1

2.3.1. La Integración

Emplea el método de Euler[1] o de la recta tangente (Boyce & Diprima, 2000), es un procedimiento de integración numérica para la resolución de EDO (Ecuaciones Diferenciales Ordinarias) con valor inicial. El modelo matemático se constituye de la siguiente manera:

Supongamos que se tiene una ecuación diferencial de primer orden como en (2.9).

$$\frac{dy}{dt} = f(t, y) \tag{2.9}$$

[1] Llamado así en honor al matemático y físico Leonhard Euler (1707- 1783)

Y con la condición inicial

$$y(t_0) = y_0 \tag{2.10}$$

El método consiste en dividir los intervalos que van de t_0 a t_{n+1} en n subintervalos de ancho h como en (2.11). De manera que se obtiene un conjunto de puntos $t_0, t_1, t_2, \ldots\ldots, t_{n+1}$ del dominio $[t_0, t_{n+1}]$. Para todos estos puntos se cumple la condición que $t_i = t_0 + ih \quad 0 \leq i \leq n$ el valor de la solución exacta en $t = t_n$ es $\emptyset(t_n)$.

$$h = \frac{t_{n+1} - t_0}{n} \tag{2.11}$$

Dado que se conocen t_0 y y_0 también se conoce la pendiente de la recta tangente a la solución en $t = t_0$ y $\emptyset'(t_0) = f(t_0, y_0)$. Es posible construir una recta tangente a la solución en t_0 y obtener la solución aproximada y_1 de $\emptyset(t_1)$ al desplazarse a lo largo de la recta tangente desde t_0 a t_1 como se observa en la figura 2.8. Por lo tanto

$$y_1 = y_0 + \emptyset'(t_0)(t_1 - t_0) \tag{2.12}$$

$$y_1 = y_0 + f(t_0, y_0)(t_1 - t_0) \tag{2.13}$$

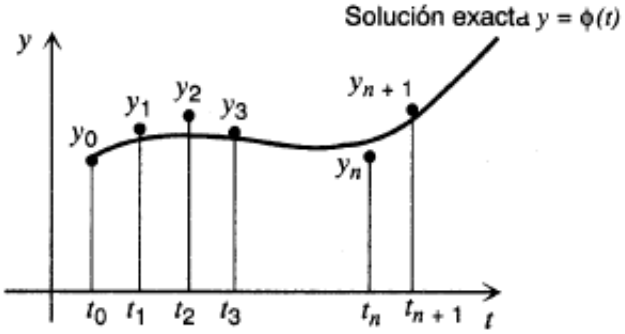


Figura 2.8 Método de Euler para la Resolución de EDO.
FUENTE: Boyce Di prima, 2000, figura 8.1.2, pág. 430

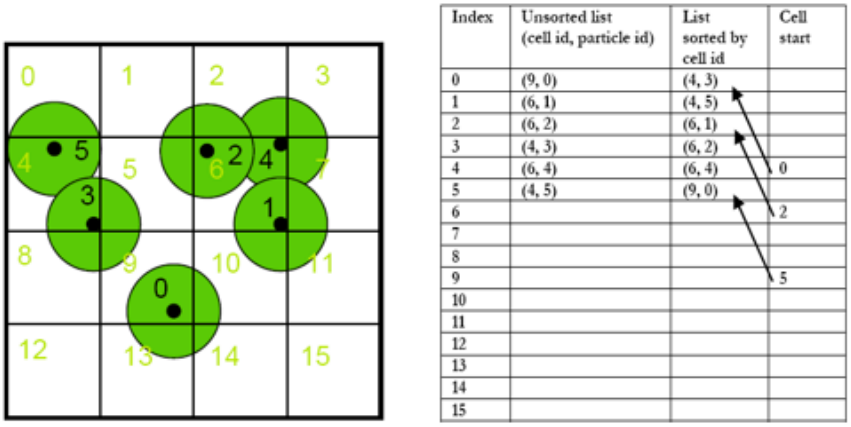
En el tutorial particles, el método de Euler realiza el cálculo de los atributos de las partículas (posición y velocidad) a medida que la partícula se desplaza en el espacio. Posteriormente, se actualiza la velocidad basándose en las fuerzas y la gravedad.

Finalmente, se actualiza la posición de las partículas basándose en los datos actualizados de la velocidad.

El vector de Las posiciones y velocidades de cada partícula son almacenadas en arreglos de tipo Float4. La posición se almacena y se presenta gráficamente mediante el uso del OpenGL.

2.3.2. Construcción de Red de Datos

Para todo el dominio el proceso de construcción de datos se emplea el método "Enmallado Uniforme", donde se subdivide en celdas o rejillas. Para simplificar el trabajo se implementa que las celdas tengan el mismo tamaño que el de la partícula ΔX.



Index	Unsorted list (cell id, particle id)	List sorted by cell id	Cell start
0	(9, 0)	(4, 3)	
1	(6, 1)	(4, 5)	
2	(6, 2)	(6, 1)	
3	(4, 3)	(6, 2)	
4	(6, 4)	(6, 4)	0
5	(4, 5)	(9, 0)	
6			2
7			
8			
9			5
10			
11			
12			
13			
14			
15			

Figura 2.9 Método de Clasificación de las Partículas en el Tutorial Particles.
FUENTE: Green, 2013, figura 2, pág. 6

La figura 2.9 muestra 6 partículas, cada uno tiene un índice que va desde el 0 hasta el 5. Preliminarmente se realiza un arreglo unsorted list (cell id, particle id) la cual muestra la celda en donde se encuentra cada partícula y su identificación. Por ejemplo la partícula 2 se encuentra en la celda 6.

En la generación de la cuadricula o enmallado se emplea el método "sorting" o de clasificación. El cual nos permite agrupar las partículas en celdas, simplemente clasificándolas por el índice de cada celda (grid). Para la partícula 2 se sigue el siguiente procedimiento:

1. Calcula el valor HASH de la partícula 2 basándose en el id de la celda que se encuentra, con una función KERNEL denominada "CalcHash", CalcHash almacenara el valor de 6.
2. El resultado del ítem 1 se almacenan en un Arreglo llamado "particleHash", el valor que almacenará particleHash será 6.
3. Clasifica la partícula 2 en función de su valor Hash, con esto se crea una nueva lista de id de la partícula 2 y el orden de la celda. Para este caso sería (6,2), quiere decir que en la celda 6 se encuentra la partícula 2
4. Por último, utilizando el KERNEL "findCellStart" calculamos el inicio de cada celda. Utilizando un hilo (thread) por cada partícula, lo compara con el índice de celda de la partícula actual con el índice de celda de la partícula anterior en la lista ordenada. Para el caso de la partícula 2 una vez realizado el findCellStart se ubicara en el índice 3.

2.3.3. Proceso de Colisión

Esta subrutina simula el fluido como un sistema de partículas conectadas entre sí con masas concentradas, resortes, y amortiguadores. Son calculadas usando (2.14). En la tabla 2-3 se muestran los valores iniciales del programa.

Parámetros	Valor
Damping (η)	0.02
timestep (Δt)	0.5
gravity constant(g)	0.0003
spring constant(κ)	0.5
Shear (k_t)	0.1
Dist (d)	0.031
attraction	0
ParticleRadius	0.016
Cell size	0.031

Tabla 2-3 Parámetros Iniciales del Tutorial

La fuerza debido al resorte es:

$$f_{i,s} = -k(d - |r_{ij}|)\frac{r_{ij}}{|r_{ij}|} \tag{2.14}$$

La fuerza debido al amortiguamiento es:

$$f_{i,d} = \eta v_{ij} \tag{2.15}$$

Donde, k es el coeficiente de resorte, d es la distancia promedio entre partículas y es aproximado como 2 veces su radio, r_{ij} es la distancia entre partículas y v_{ij} es la velocidad relativa de la partícula.

La fuerza debido a su velocidad tangencial es:

$$f_{i,t} = k_t v_{ij,t} \tag{2.16}$$

Donde velocidad tangencial es:

$$v_{ij,t} = v_{ij} - \left(v_{ij} \cdot \frac{r_{ij}}{|r_{ij}|}\right) \frac{r_{ij}}{|r_{ij}|} \tag{2.17}$$

Luego de calcular la contribución de cada una de las fuerzas que actúan sobre las partículas, la Fuerza se almacena en el arreglo newVel de tipo float4, el cual actualiza la velocidad y posición debido a la contribución de la fuerza. Ver figura 2.10

```
// write new velocity back to original unsorted location
uint originalIndex = gridParticleIndex[index];
newVel[originalIndex] = make_float4(vel + force, 0.0f);
vel += params.gravity * deltaTime;
vel *= params.globalDamping;

// new position = old position + velocity * deltaTime
pos += vel * deltaTime;
```

Figura 2.10 Actualización de Velocidades y Posiciones del Tutorial Particles.

2.4. Implementación de SPH en CUDA

El método SPH es similar a particles porque se basan en la interacción entre partículas para determinar su comportamiento, aunque se diferencian en cómo se modela la

interacción entre partículas. En SPH se aproximan las fuerzas de interacción considerando las ecuaciones gobernantes de los fluidos por lo que es necesario calcular los valores de densidad y presión para cada partícula. Fue necesario agregar al tutorial “Particles” 4 subrutinas para lograr la implementación del método. Como se muestra la figura 2.10. Las subrutinas encargadas de calcular la densidad son: Densidad entre partículas, Densidad Cel y Densidad_Upate. Por último, la subrutina encargada de implementar el método SPH se llama Collide_SPH.

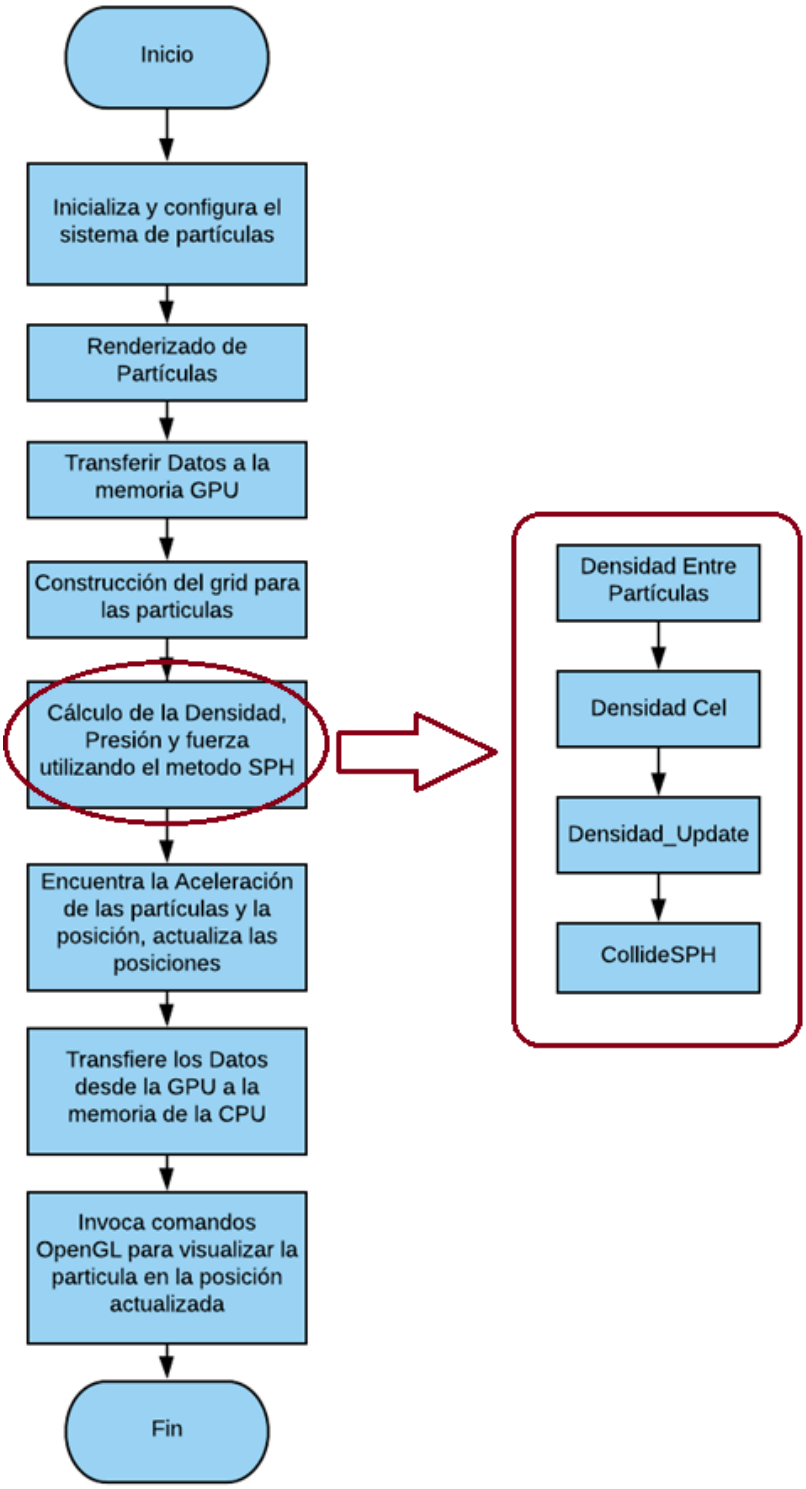


Figura 2.11 Implementación del Método SPH en CUDA.

2.4.1. Parámetros Iniciales

Se determina el número de particulas en cada eje considerando (2.19), distancia entre particulas (2.20), volumen de las particulas (2.21), masa inicial de las particulas.

$$Numpart_X = \sqrt[3]{Num\_Particles} \tag{2.19}$$

$$\Delta X = \frac{2.0}{(Numpart_X - 1)} \tag{2.20}$$

$$volmen\ de\ la\ particula = (\Delta X)^3 \tag{2.21}$$

Considerando (2.18), es necesario calcular el valor de la funcion Kernel; distancia relativa entre particulas (2.22), radio de la funcion suavizada (2.24) , calculo del valor alfa del kernel (2.25).

$$relpos = posB - posA \tag{2.22}$$

$$dist = length(relpos) \tag{2.23}$$

$$h = 2 * \Delta X \tag{2.24}$$

$$\alpha_D = \frac{105}{16\pi h^3} \tag{2.25}$$

Calculo de $\Delta\rho$: Es necesario evaluar el Kernel (Lucy) (2.27), para obtener la contribucion de la densidad (2.28).

$$q = \frac{|dist|}{h} \tag{2.26}$$

$$W = \alpha_D(1 + 3q)(1 - q)^3 \tag{2.27}$$

$$\Delta\rho_i = W * massB \tag{2.28}$$

2.4.2. Densidad entre partículas

Esta es la primera subrutina implementada para el cálculo de la contribución a la densidad de la particula i y de la particula j en el mismo dominio ver (2.5), en la figura 2.12 se detalla el fujograma. Concluye que el aporte Δp de la interacción entre la particula $i\ y\ j$ es :

$$\Delta\rho = m_j; W_{ij} \tag{2.18}$$

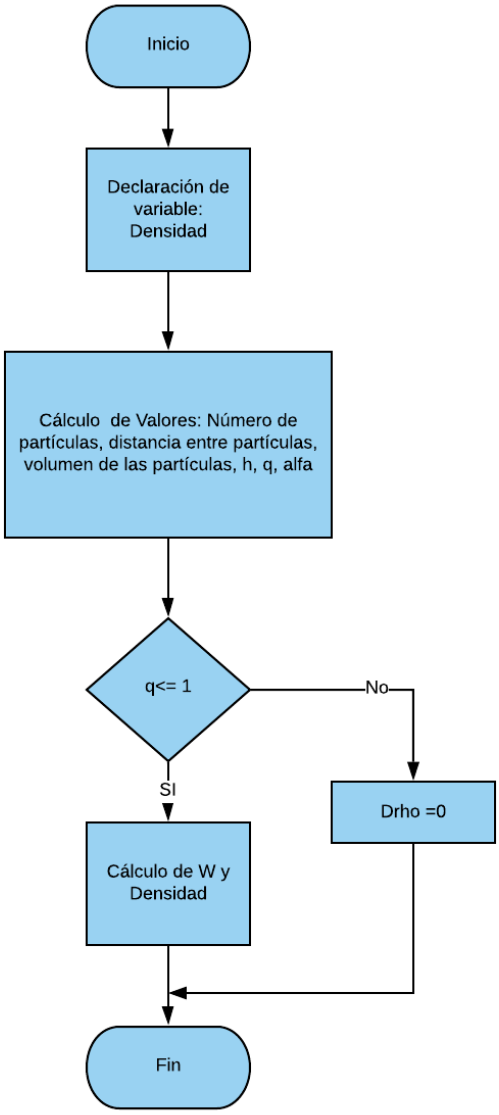


Figura 2.12 Flujograma de la Función Densidad entre Partículas.

En la figura 2.13 muestra la implementación de esta subrutina.

```
__device__
float Densidad_entre_particulas(float3 posA, float3 posB , float massA, float massB)       //se agrega las masas A,B
{

    float rho_ref = 1000;                                                                  //  Declaramos del valor inicial de la densidad
    float Densidad;                                                                        // Se declara la variable densidad
    uint Numparticulasx = int(powf(N, (1.0f / 3.0f)));                                     // Se declara el numero de particulas en cada eje
                                                                                           // Distancia el cubo donde estarn las particulas
    float3 relPos = posB - posA;                                                           // calcula la position relativa
    float dist = length(relPos);                                                           // Calculo de la distancia
    float distancia = 2.0 / (Numparticulasx - 1.0f);                                       // espaciamiento entre particulas
    float vol = powf(distancia, 3.0f);                                                     // volumen de cada particula

        float h = 2.0f * distancia;                                                        //  Declaramos el valor de h, que es el radio de separacion
        float alfa = 105.0f / (16.0f *Pi*(powf(h, 3.0f)));                                 //  Define la variable alfa para evaluar la funciona Kernel
        float R = abs(dist) / h;                                                           //  Evalua el valor de R

        if (R <= 1.0)
        {
            float W = alfa * (1.0f + (3.0f * R))*(powf((1.0f - R), 3.0f));                 //  Evalua la funcion Kernel Quartic(Lucy 1977) siempre y cuando  R <= 1
            Densidad += W * massB;                                                         //  densidad entre 2 particulas

        }

        return Densidad;
}
```

Figura 2.13 Implementación de la Subrutina Densidad entre Partículas.

2.4.3. Densidad Cell

Esta es la segunda subrutina utilizada para el cálculo de la densidad. Consiste en evaluar la particula de interes i con las particulas vecinas j dentro del enmallado de celdas vecinas, ver figura 2.14, para el calculo de la densidad.

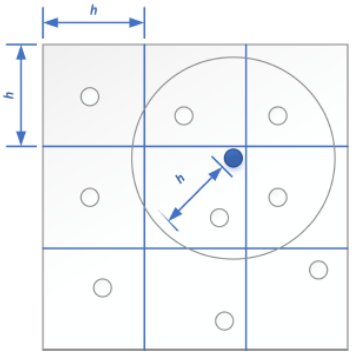


Figura 2.14 Enmallado del sistema (García Garino, 2013)

Esta subrutina se implementa de la siguiente manera:

- Recibe los valores posicion y masa, almacenados en las memorias HOST(CPU) y DEVICE(GPU).
- Se declara las variables iniciales: $\Delta\rho$, Indice Inicial e Indice Final de cada particula.

- Realiza las iteraciones sobre todas las particulas vecinas donde se encuentra la particula de interes i.
- Ejecuta la subrutina “Densidad entre particulas” para el calculo de la Densidad entre 2 particulas descrita en la sección anterior.

En la figura 2.15 se detalla el flujograma.

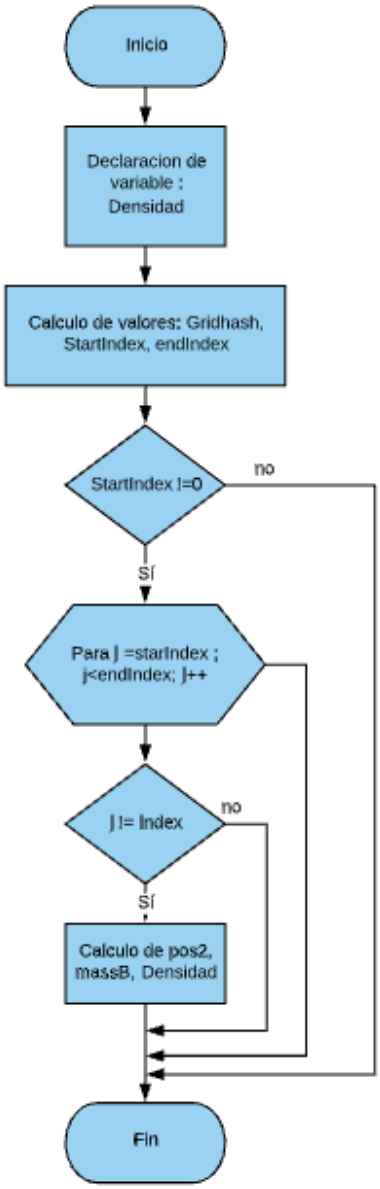


Figura 2.15 Flujograma de la Función Densidad cell.

En la figura 2.16 se muestra la implementacion de esta subrutina.

```
__device__
float Densidad_Cell(int3    gridPos,
                    uint    index,
                    float3  pos,
                    float   mass,                          // se agrega el valor masa
                    float4 *oldPos,
                    float  *masa_array,                    //se agrega el valor sortedmasa
                    uint   *cellStart,
                    uint   *cellEnd)

{
    uint gridHash = calcGridHash(gridPos);                 //Calcula el gridhash de cada particula
    // get start of bucket for this cell
    uint startIndex = FETCH(cellStart, gridHash);          //Calcula el Inidice de inicio
    float Densidad;                                        // Inicializa el valor de la Densidad
    if (startIndex != 0xffffffff)                          // Verifica que la celda no este vacia
    {

        uint endIndex = FETCH(cellEnd, gridHash);         // iterar sobre partículas en esta celda

        for (uint j = startIndex; j < endIndex; j++)
        {
            if (j != index)                                // chequea que la colision no sea con la misma particula
            {
                float3 pos2 = make_float3(FETCH(oldPos, j));               // SE calcula la nueva posicion
                float massB = FETCH(masa_array, j);                        // se agrega un valor de masab
                Densidad += Densidad_entre_particulas(pos, pos2,mass,massB);    // Calculo de la densidad entre 2 particlas en el enmallado
            }
        }

    }

    return Densidad;
}
```

Figura 2.16 Implementación de la Subrutina Densidad cell.

2.4.4. Densidad_Update

Esta es la tercera subrutina para el cálculo de la densidad. Y Consiste en evaluar la funcion Global de la particula de interes i con las particulas vecinas j considerando todo el dominio computacional x,y,z. Esta subrutina se ejecuta de la siguiente manera.

- Lee los datos almacenados de los arreglos masa y posición.
- Se declara el valor inicia de: Densidad, la cual se actualizó solamente si las particulas interactuan entre si.
- Se obtiene la direccion de la cuadricula (Grid) donde se encuentra cada particula.
- Examina las celdas vecinas para encontrar las particulas de interes y proceder con el cálculo.
- Almacena el arreglo densidad (densidad_array) para calculos futuros.

En la figura 2.17 se detalla el fujograma.

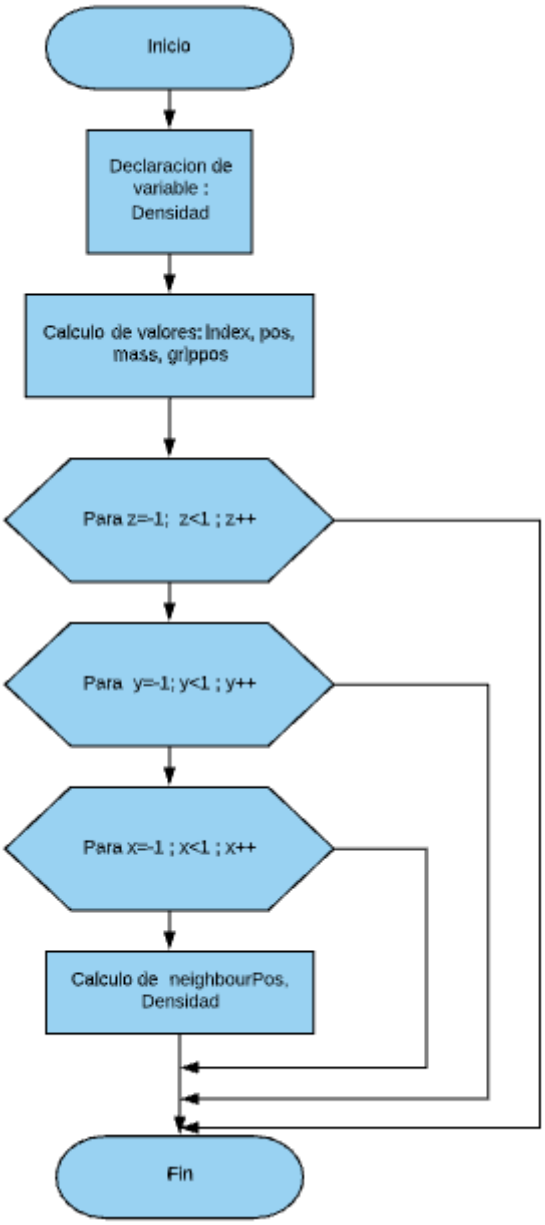


Figura 2.17 Flujograma de la Función Densidad_Update.

En la figura 2.18 muestra la implementacion de esta subrutina.

```
__global__
void Densidad_updateD(  float *densidad_array,              //ouput : arreglo densidad se agrega
                        float4 *oldPos,                     // input: sorted positions
                        float  *masa_array,                  // input: masa array          // se agrega el parametro masa al global
                        uint  *gridParticleIndex,           // input: sorted particle indices
                        uint  *cellStart,
                        uint  *cellEnd,
                        uint  numParticles)

{
    uint index = __mul24(blockIdx.x, blockDim.x) + threadIdx.x; // Prepara la GPU para que procese el Indice

    if (index >= numParticles) return;
    float3 pos = make_float3(FETCH(oldPos, index));        //Lee los datos posicion de las particulas ordenadas
    float  mass = FETCH(masa_array, index);                // Lee los datos masa ordenados
    int3 gridPos = calcGridPos(pos);                       //obtiene la direccion del GRID
    float Densidad;                                        //inicializa la densidad
                                                           // examine neighbouring cells

                for (int z = -1; z <= 1; z++)              // realiza un mapeo en las 3 direcciones
                {
                    for (int y = -1; y <= 1; y++)
                    {
                        for (int x = -1; x <= 1; x++)
                        {
                            int3 neighbourPos = gridPos + make_int3(x, y, z);
                            Densidad += Densidad_Cell(neighbourPos, index, pos, mass, oldPos, masa_array, cellStart, cellEnd);      //Calculo de la densidad

                        }
                    }
                }
            // Escribe el nuevo valor de la densidad en un arreglo sin clasificar
            uint originalIndex = gridParticleIndex[index];
            densidad_array[originalIndex] = Densidad;
}
```

Figura 2.18 Implementación de la Subrutina Densidad_Update.

2.4.5. CollideSPH

En esta subrutina se calcula la fuerza entre dos particulas considerando la presión de cada una de ellas. Se modificó la funcion “CollideSpheres”, incluyendo los siguiente cambios:

- Se agregan nuevos valores a la funcion “CollideSpheres” se los muestra en la tabla 2-4.

	Descripción	Característica	tipo
masaA	Masa de la partícula A	Arreglo	float
masaB	Masa de la partícula B	Arreglo	float
rhoA	Densidad de la partícula A	Arreglo	float
rhoB	Densidad de la partícula B	Arreglo	float
deltatime	Incremento del tiempo	Escalar	float

Tabla 2-4 Valores iniciales de la función CollideSpheres

- Se declaran nuevas variables se detallan en la tabla 2-5

	Descripción	Característica	tipo
Force	Fuerza del sistema	Arreglo	float3
PressA	Presión de la partícula B	Arreglo	float
PressB	Presión de la partícula A	Arreglo	float
Grad_W	Gradiente del Kernel	Arreglo	float3
rho_ref	Densidad de referencia	Escalar	float
CS	Velocidad del sonido	Escalar	int

Tabla 2-5 Nuevos parámetros de CollideSpheres

- Se calcula los parametros iniciales con la misma formulacion de (2.18) hasta (2.26).
- Para implementar el método SPH se eliminan los cálculos que se muestran en la tabla 2-6.

	Descripción	Característica	tipo
tanVel	Velocidad tangencial	Arreglo	float3
relVel	Velocidad relativa	Arreglo	float3
Spring Force	Fuerza del resorte	Arreglo	float3
Damping Force	Fuerza de amortiguamiento	Arreglo	float3
Attraction Force	Fuerza de atracción	Arreglo	float3

Tabla 2-6 Valores Eliminados del Tutorial inicial

- Finalmente, se calculan los valores ∇W(Gradiente), presiones (PressA, PressB) y la fuerza, la formulacion a emplearse se detalla en (2.29), (2.30), (2.31), (2.32).

$$\nabla W = \left(\frac{\alpha_d * Norm}{h}\right) * \left(\left[\frac{3R}{|R|}\right] * (1-R)^3 - \left[\frac{3R}{|R|}\right](1+3R) * (1-R)^2\right) \quad (2.29)$$

$$PressA = Cs^2(Rho_A - Rho_{ref}) \quad (2.30)$$

$$PressB = Cs^2(Rho_B - Rho_{ref}) \quad (2.31)$$

$$fuerza = \frac{-rho_{ref} * \left[\frac{Press_A}{RhoA^2} + \frac{Press_B}{RhoB^2}\right] * Mass_B * \nabla W * dt}{Mass_A} \quad (2.32)$$

En la figura 2.19 se detalla el flujograma

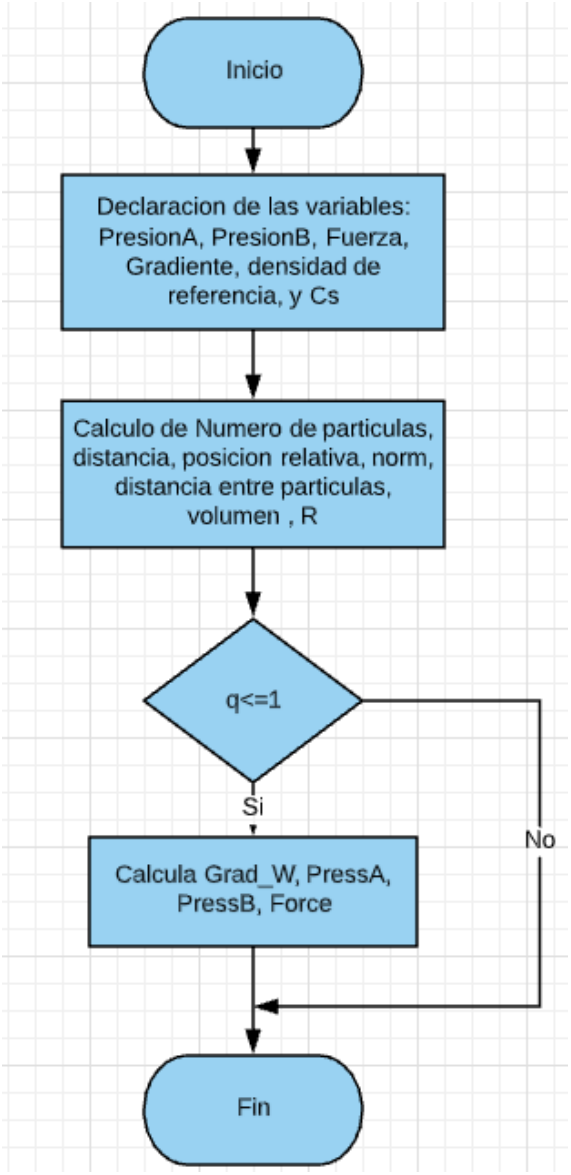


Figura 2.19 Flujograma de la función CollideSPH

En la figura 2.20 se muestra la subrutina implementada.

```
// collide two spheres using DEM method
__device__
float3 collideSph(float3 posA, float3 posB,
                  float3 velA, float3 velB,
                  float massA, float massB,
                  float RhoA, float RhoB, float deltaTime)                    //se agregan los valores de las densidad y de las masas

{
    // calculate relative position              ////////////////////////////////////////////////////////////
                                                //                                                        //
    float3 relPos = posB - posA;                //  Subrutina que realiza el calculo de la fuerza         //
    float dist = length(relPos);                //                      SPH                               //
    float3 norm = relPos / dist;                //                                                        //
    //float3 relVel = velB - velA;              //                                                        //
                                                ////////////////////////////////////////////////////////////
    float3 force = make_float3(0.0f);
    float3 grad_W= make_float3(0.0f);
    float rho_ref = 1000;                                                     //  Declaramos del valor inicial de la densidad
    uint Numparticulasx = int(powf(N, (1.0f / 3.0f)));                        // Se declara el numero de particulas en cada eje, se deja f
    float distanx = 2.0f;                                                     // Distancia el cubo donde estarn las particulas
    float distancia = distanx / (Numparticulasx - 1.0f);                      // espaciamiento entre particulas
    float vol = powf(distancia, 3.0f);                                        // volumen de cada particula

    float h = 2.0f * distancia;                                               //  Declaramos el valor de h, que es el radio de separacion
    float alfa = 105.0f / (16.0f *Pi*(powf(h, 3.0f)));                        //  Define la variable alfa para evaluar la funciona Kernel
    float R = abs(dist) / h;
    float PressA;
    float PressB;
    int cs = 100;                                                             // velocidad del sonido asumida
    if (R <= 1)
    {
        grad_W = (alfa*norm/h)*(((3*R)/(abs(R))*powf((1-R),3.0f))-((3*R/abs(R))*(1+3*R)*(powf((1-R),2))));      //  Primera derivada del Kernel

        PressA = powf(cs,2) * (RhoA-rho_ref);
        PressB = powf(cs,2)* (RhoB-rho_ref);

        force += ((-rho_ref)*(PressA / (powf(RhoA, 2)) + (PressB / (RhoB, 2)))*massB*grad_W*deltaTime)/(massA);

    }

    return force;
}
```

Figura 2.20 Implementación de la Subrutina para el Cálculo de la Presión.

CAPÍTULO 3

3. RESULTADOS Y ANALISIS

3.1. Equipo empleado en el desarrollo del proyecto

A continuación se detalla en la tabla3-1 las características del computador utilizado en el desarrollo del proyecto.

Elementos	**Detalles**
CPU	AMD-fx-8300 8GB RAM 3300 MHZ
Información del sistema	Windows 7 Professional, 64-bit
DirectX	11
GPU	
Información de la tarjeta Grafica **GeForce GT 740**	Núcleos CUDA 384 Potencia de computo 0,76 TFLOPS Memoria 1GB Arquitectura Kepler Interfaz de Memoria 128-bits

Tabla 3-1 Características del Computador Usado

3.2. Modificación del tutorial Particles

Para el desarrollo del método SPH (**S**moothed **P**article **H**ydrodynamics) se procedió a modificar el tutorial “particles”. Los cambios que se realizaron se detallan a continuación.

3.2.1. Distribución Inicial de Partículas.

Fue necesario modificar las distancias entre las partículas dentro de un cubo de 2 metros de largo especificada en el código fuente “particleSystem.cpp”, específicamente en la función “initGrid” se adiciona la siguiente ecuación (3.1)

$$\Delta_X = \frac{Dist_X}{(Numpart_X - 1)} \tag{3.1}$$

Donde:

$$Dist_X = Longuitud\ del\ cubo$$
$$Dist_X = Dist_Y = Dist_Z$$
$$\Delta_X = \Delta_Y = \Delta_Z$$
$$Numpart_X = \sqrt[3]{Num\_Particles} \quad (3.2)$$
$$Numpart_X = Numpart_Y = Numpart_Z$$

Los cálculos fueron desarrollados como se muestra en la tabla 3-2.

Variable	Descripción	Valor	Unidades
Num_Particles	Número de Partículas Totales	1000	partículas
NumpartX	Número de Partículas Eje X	10	partículas
DistX	Longitud del cubo	2	metros
g	Gravedad	9.8	m/s^2

Tabla 3-2 Datos ingresados para el calculo

El valor de la distancia obtenida con los datos ingresados es de:

$$Espaciamiento_X = 0{,}22\ metros$$

En la Figura 3.1 se muestran la implementación de los cambios que se realizaron.

```
void
ParticleSystem::initGrid(uint *size, float spacing, float jitter, uint numParticles)
{
    srand(1973);

    float xmax = 1.0f;          //Nueva configuracion del paralelepipedo colisionador de particulas en los 3 ejes
    float xmin = -1.0f;
    float ymax = 1.0f;
    float ymin = -1.0f;
    float zmax = 1.0f;
    float zmin = -1.0f;

    float distx = xmax - xmin;
    float disty = ymax - ymin;
    float distz = zmax - zmin;
    float arreglo = (m_numParticles *(distx / disty)*(distx / distz));

    uint Numpartx = (powf((arreglo), 1.0f / 3.0f)); //Numero de particulas por cada eje
    uint Numparty = (disty / distx)*Numpartx;
    uint Numpartz = (distz / distx)*Numpartx;

    numParticles = Numpartx * Numparty* Numpartz;
    float espaciamientox = distx / (Numpartx - 1);  //Distancia entre particulas iguales en el eje x
    float espaciamientoy = disty / (Numparty - 1);  //Distancia entre particulas iguales en el eje y
    float espaciamientoz = distz / (Numpartz - 1);  //Distancia entre particulas iguales en el eje z
```

Figura 3.1 Función initGrid modificada

3.2.2. Modificación de la Interfaz gráfica

Se cambia la tonalidad del fondo de pantalla modificando los parámetros de entrada de la función **glClearColor** en el código fuente particle.cpp", específicamente en la función "initGL". Esta función es la encargada de ejecutar la parte gráfica usando la interfaz OpenGL.

¿Qué es OpenGL?

OpenGL (**O**pen **G**raphics **L**ibrary) es una interfaz gráfica compuesta de una serie de librerías, que son utilizadas a través de lenguajes de programación como C++ en las cuales se puede construir formas geométricas, ubicar formas geométricas tridimensionales y aplicar textura a los objetos.

Función glClearColor

La función glClearColor es la encargada de darle la tonalidad a un objeto, la especificación es la siguiente:

Void glClearColor (GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*);

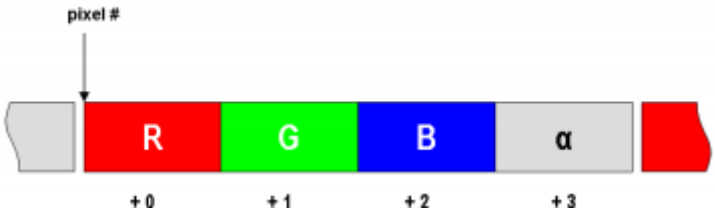


Figura 3.2 Configuración de Color de Cada Pixel (Cámara, Represa, & Sánchez, 2016)

Especifica los valores, rojo, verde, azul y alfa (**R**ed **G**reen **B**lue) para modificar los búferes de color, estos valores están sujetos a un rango de [0,1]. En el caso de la tonalidad de las partículas el código fuente encargado es "particleSystem.cpp", la función "colorRamp" coloca 7 tipos de colores: rojo, amarillo, verde, celeste, morado, azul. En el desarrollo de este proyecto se dejaron todas las partículas de color azul. En la figura 3.3 se muestra los cambios.

```
Void glClearColor (1.0, 1.0, 0.8, 1.0);
```

Figura 3.3 Datos modificados para obtener la tonalidad del proyecto

3.2.3. Opciones de menú principal.

La función "initMenus" situada en el código fuente particle.cpp", consta de 7 opciones para visualizar los diferentes cambios, en la tabla 3-3 se detallan las características del menú principal.

Opción	Función	Descripción
1	Reset	Reiniciar la simulación
2	Random	Agrega una distribución aleatoria de partículas
3	Add Sphere	Agrega una esfera con partículas
v	View mode	Mueve la cámara del sistema
m	move cursor	Mueve la esfera de colisión
p	toggle point rendering	Muestra las partículas como puntos
h	toggle sliders	Muestra los parámetros iniciales

Tabla 3-3 Menú Principal

Para fines del desarrollo del proyecto se deshabilita visualmente todo el menú principal, solo se deja la opción 1 (RESET).

3.2.4. Cubo de colisión.

Para modificar la dimensión del dominio de las partículas se debe ingresar en el código fuente "particle.cpp", la encargada de realizar este trabajo es la función "display". Dentro existen 2 funciones secundarias, La primera "glcolor3f" le da el color a las aristas del cubo, y la segunda "glutWireCube" le da las dimensiones al cubo de colisión en los 3 ejes. En la figura 3.4 se observan las líneas del código fuente que se modificaron.

```
glcolor3f(1.0, 1.0, 1.0);
glutWireCube(2.0);
```

Figura 3.4 Funciones del Cubo de Colisión

Por último en la figura 3.5 se observa la distribución de las partículas obtenidas con los cambios realizados.

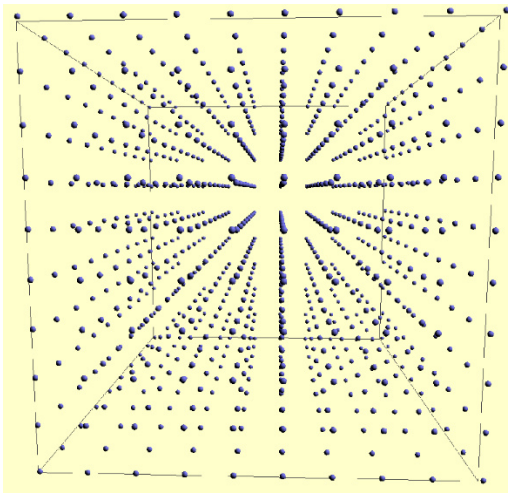


Figura 3.5 Resultado de las Modificaciones del Tutorial Particles.

Para obtener una mejor referencia en la figura 3.6 y 3.7 se realizó un modelado en Rhinoceros indicando el nivel de referencia y las dimensiones del cubo

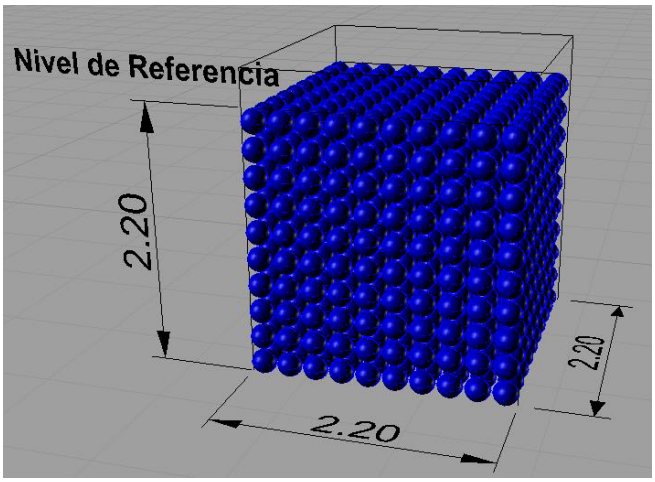


Figura 3.6 Modelado del Sistema

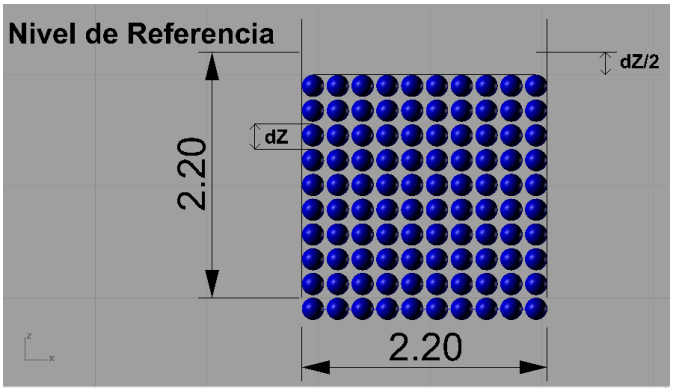


Figura 3.7 Vista Frontal del Sistema

3.3. Cálculo de la masa inicial del sistema

Como primer paso en esta simulación, se procede a calcular el valor inicial de la masa de cada partícula mediante la estimación de la presión hidrostática ecuación (3.3) con la ecuación de estado (2.8), similar a lo que se muestra en la figura 3.8.

$$P = \gamma \Delta Z \tag{3.3}$$

Igualando las ecuaciones (3.3) y (2.8) tenemos (3.4):

$$\gamma \Delta Z = C_s^2(\rho_i - \rho_0) \tag{3.4}$$

Donde:

γ : Peso específico del agua 9800 $\frac{N}{m^3}$

ρ_0 : Densidad de referencia del agua 1000 $\frac{kg}{m^3}$

C_s : Velocidad artificial del sonido 100 $\frac{m}{s}$

ΔZ : Variación de la profundidad en m.

Despejando de la ecuación (3.4) el valor de la densidad inicial ρ_i (3.5):

$$\rho_i = \frac{\gamma \Delta Z + C_s^2 \rho_0}{C_s^2} \tag{3.5}$$

El valor inicial de la masa será

$$m = \Delta x^3 * \rho_i \tag{3.6}$$

Donde:

Δx^3 : Volumen de la partícula

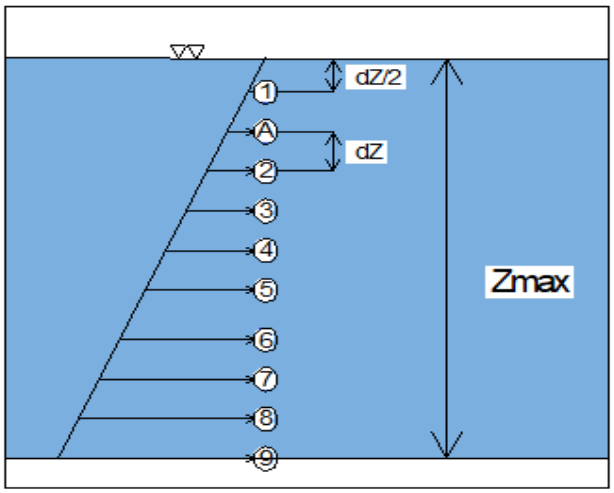


Figura 3.8 Distribución de presión Hidrostática de las Partículas.

En la figura 3.9 se muestra la subrutina implementada

```
uint Numpartx = (powf((arreglo), 1.0f / 3.0f)); //Numero de particulas por cada eje
uint Numparty = (disty / distx)*Numpartx;
uint Numpartz = (distz / distx)*Numpartx;

numParticles = Numpartx * Numparty* Numpartz;
float espaciamientox = distx / (Numpartx - 1);  //Distancia entre particulas iguales en el eje x
float espaciamientoy = disty / (Numparty - 1);  //Distancia entre particulas iguales en el eje y
float espaciamientoz = distz / (Numpartz - 1);  //Distancia entre particulas iguales en el eje z
```

```
int rho_ref = 1000;         //Densidad de Referencia
int gama = 9800;            //Peso especifico
int cs = 100;               // velocidad del sonido asumida
float vol = powf(espaciamientoy, 3);    // volumen de las particulas

    float press = -(gama* (m_hPos[i * 4 + 1] - ymax - 0.5*espaciamientoy));     // Calculo de la presion mediante P Hidrostatica
    float density =  (press + (powf(cs,2)*rho_ref))/(powf(cs,2));               //densidad a partir de la ecuacion de estado.
    float mass = density * vol;
    m_hMasa[i] = mass;                                                          // se agrega esta linea para el arreglo masa
```

Figura 3.9 Subrutina del cálculo de las masas iniciales

Una vez obtenido estos valores dentro del tutorial, se procede a almacenarlos en las memorias HOST (CPU) y DEVICE (GPU) para hacer uso en las otras subrutinas,

mediante un arreglo denominado “MASA”. En la tabla 3-5 se anexan los valores iniciales, y en la figura 3.10 tenemos los valores calculados del programa.

i	Z	Presión	Densidad	Masa
Nivel	m	N/m^2	Kg/m^3	kg
1	0.111	1088.889	1000.109	10.975
2	0.333	3266.667	1000.327	10.978
3	0.556	5444.444	1000.544	10.980
4	0.778	7622.222	1000.762	10.982
5	1.222	11977.778	1001.198	10.987
6	1.444	14155.556	1001.416	10.989
7	1.667	16333.333	1001.633	10.992
8	1.889	18511.111	1001.851	10.994
9	2.111	20688.889	1002.069	10.997

Tabla 3-4 Valores Iniciales Mediante Presión Hidrostática.

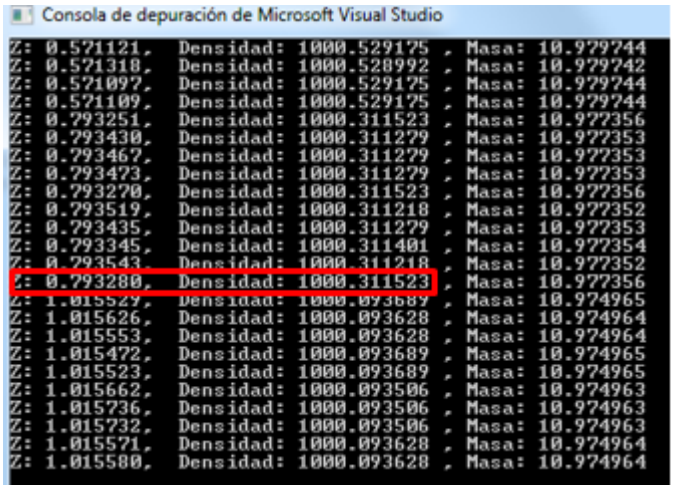


Figura 3.10 Valores Obtenidos del Programa Modificado.

En la tabla 3-5 se realiza la comparación de la partícula en ese nivel y su comparación con la predicción hidrostática. Luego de esto pasamos al cálculo de las densidades.

Profundidad [m]	Masa teórica [kg]	Masa Numérico [kg]	% ERROR
0.778	10.982	10.977	0.04

Tabla 3-5 Comparación de Valores Masa

3.4. Cálculo de las densidades

Una vez calculada la masa referencial del sistema para cada partícula, se procede a llevar estos datos a las tres subrutinas que son las encargadas de realizar este cálculo. El resultado obtenido de la partícula en el nivel 4 se los presenta en la figura 3.11.

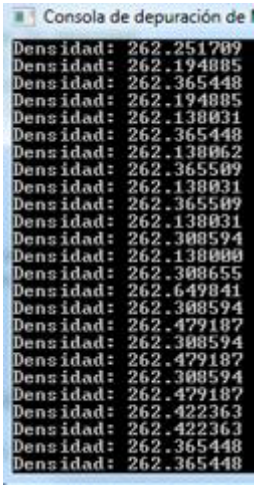


Figura 3.11 Resultados del valor de la Densidad

Como se observa en valor de la densidad es de 262.365 $^{kg}/_{m^3}$, esto significa que el programa solo está computando la contribución de la partícula del nivel 4. En la tabla 3-6 se muestra la comparación de resultados.

Profundidad [m]	Densidad teórica [kg]	Densidad Numérico [kg]	% ERROR
0.778	1000.762	262.365	73.76

Tabla 3-6 Comparación de Valores Densidad

Se almacenan estos valores en arreglos denominados ρ_A y ρ_B los cuales son las densidades entre las partículas A y B. Esto es en la subrutina Densidad_Update, ver figura 3.12.

```
    }
// Escribe el nuevo valor de la densidad en un arreglo sin clasificar
uint originalIndex = gridParticleIndex[index];
densidad_array[originalIndex] = Densidad;
```

Figura 3.12 Valores de Densidad Almacenados

3.5. Cálculo de las Presiones

Una vez obtenida la densidad de cada partícula en los 3 ejes se procede con el cálculo de la presión de la partícula en el nivel 4, la cual solo es la contribución de ella, ver tabla 3-8

Profundidad [m]	Presión teórica $\frac{N}{m^2}$	Presión Numérica $\frac{N}{m^2}$
0.778	7622.222	73763.51

Tabla 3-7 Comparación de Valores de Presión

CAPÍTULO 4

4. CONCLUSIONES Y RECOMENDACIONES

Después de haber implementado el método SPH (**S**moothed **P**article **H**ydrodynamics) n el tutorial “Particles” de CUDA se puede concluir lo siguiente:

CONCLUSIONES

1. El tutorial modificado simula la interacción de partículas de agua como si estuvieran conectadas en un sistema cuya fuerza es proporcional a las ecuaciones de Navier Stokes, calculando la densidad y presión.
2. Para la implementación del método SPH (**S**moothed **P**article **H**ydrodynamics) se requirió adicionar tres subrutinas al tutorial “Particles” y modificar de la subrutina de las colisiones (CollideSPH).
3. El método SPH es una herramienta poderosa que reduce la complejidad de las ecuaciones matemáticas de los fluidos.
4. Es posible utilizar las tarjetas gráficas GPU para la simulación de Fluidos y para cálculos numéricos.
5. Las simulaciones numéricas proporcionan una manera de estudiar fenómenos naturales o complejos sistemas.
6. El valor de la densidad obtenida por el programa fue de 262 $\frac{Kg}{m^3}$ esto es debido a la contribución de la partícula del nivel 4.
7. Es posible almacenar arreglos en las memorias CPU (Device) y GPU (Host) para realizar cálculos numéricos.
8. El método SPH podría ser usado en la industria naval en la simulación del comportamiento de estructuras flotantes en diferentes estados de mar en el Ecuador.

RECOMENDACIONES

Para investigaciones futuras se recomienda tener en cuenta lo siguiente:

1. Constatar que en el computador de trabajo estén implementadas las últimas versiones de todas las librerías CUDA para la ejecución del mismo.
2. Considerar las limitaciones de memoria RAM y del número de núcleos de la tarjeta gráfica para determinar el número máximo de partículas a utilizarse.
3. Aprender los aspectos básicos en el lenguaje de programación c++, para redactar el o los códigos fuente.
4. Corregir el cálculo de las densidades para obtener valores a simular de presión y por lo tanto de fuerzas entre partículas.
5. Verificar las unidades del tutorial “Particles” para mejorar comprensión de su implementación.
6. Verificar el sistema de unidades que utiliza el tutorial “Particles”.

BIBLIOGRAFÍA

Nuli, U., & Kulkarni, P. (octubre de 2012). *SPH BASED FLUID ANIMATION USING CUDA*. Obtenido de https://pdfs.semanticscholar.org/0fb8/5ea777c6e792e5ecaa49c9246f25295cff3c.pdf

Boyce, & Diprima. (2000). *Ecuaciones diferenciales y problemas con valores en la frontera.* Mexico: LIMUSA.

Cámara, J., Represa, C., & Sánchez, P. (Mayo de 2016). Obtenido de https://riubu.ubu.es/bitstream/handle/10259/3933/Programacion_en_CUDA.pdf;jsessionid=9D38EB7580498C9A0C99A8E564439C15?sequence=1

DualSPHysics. (29 de Enero de 2015). *Mooring simulation with DualSPHysics (SPH on GPU)*. Obtenido de https://www.youtube.com/watch?v=llGe341o-LE

Fang, C. (31 de MAyo de 2012). *SPH fluid dam break simulation in our digital earth*. Obtenido de https://www.youtube.com/watch?v=gZIT2PoUsGY

Garcia. (2010). *reacTIVision en CUDA: Aprovechando las capacidades de la computación*. Obtenido de https://www.semanticscholar.org/paper/ReacTIVision-en-CUDA-%3A-aprovechando-las-capacidades-Rodr%C3%ADguez/45c058ad8147061dbbe679cf5a2ccf8830b07557

García Garino, C. (19 de Noviembre de 2013). *IMPLEMENTACION PARALELA DE SPH USANDO*. Obtenido de https://webcache.googleusercontent.com/search?q=cache:uL4garBth6YJ:https://cimec.org.ar/ojs/index.php/mc/article/download/4401/4331+&cd=2&hl=es&ct=clnk&gl=ec

Garcia, C. (1 de Junio de 2012). *Smoothed particle hydrodynamics(SPH) : Heat Conduction*. Obtenido de https://www.cimat.mx/~neri/resources/SPH_Presentacion.pdf

Green, S. (Mayo de 2010). *Particle Simulation*. Obtenido de http://developer.download.nvidia.com/assets/cuda/files/particles.pdf

Huang, Y., & Dai, Z. (Marzo de 2013). *ResearchGate.* Obtenido de https://www.researchgate.net/publication/234124426_SPH-based_numerical_simulations_of_flow_slides_in_municipal_solid_waste_landfills

IBM. (19 de Julio de 2019). *CUDA thread model*. Obtenido de https://www.ibm.com/developerworks/library/l-gkryptdataencrypt1/index.html

Lucy, L. (Diciembre de 1977). *A numerical approach to the testing of the fission hypothesis*. Obtenido de http://adsabs.harvard.edu/abs/1977AJ.....82.1013L

Monaghan , J. J., & Gingold , R. A. (Noviembre de 1977). *Smoothed particle hydrodynamics - Theory and application to non-spherical stars*. Obtenido de http://adsabs.harvard.edu/abs/1977MNRAS.181..375G

NVIDIA. (30 de Julio de 2014). *Ejemplos de programación de GPGPU en CUDA*. Obtenido de https://compilefacil.wordpress.com/2014/07/30/ejemplos-de-programacion-de-gpgpu-en-cuda/

NVIDIA. (13 de 10 de 2017). Obtenido de https://hwbot.org/news/14976_the_gpu_flashback_archive_nvidia_geforce_6_and_the_geforce_6600_gt_card

NVIDIA. (2018). *NVIDIA*. Obtenido de https://www.nvidia.com/es-la/data-center/gpu-accelerated-applications/catalog/

NVIDIA. (2019). *NVIDIA*. Obtenido de https://www.nvidia.com/es-es/titan/titan-v/

Tamburrino, A. (8 de Agosto de 2008). *Mecánica de Fluidos 2008*. Obtenido de https://www.u-cursos.cl/ingenieria/2008/2/CI31A/1/material_docente/

Ye, I. (2015). *SHARCNET General Interest Seminar Series.* Obtenido de https://www.sharcnet.ca/help/images/d/db/GPU_Basics_2014.pdf

ANEXOS

********* Función CalcHash**********

```
// calculate grid hash value for each particle
__global__
            void calcHashD(uint   *gridParticleHash,  // output
                           uint   *gridParticleIndex, // output
                           float4 *pos,               // input: positions
                           uint    numParticles)
            {
                uint index = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
                if (index >= numParticles) return;
                volatile float4 p = pos[index];


                // get address in grid
                int3 gridPos = calcGridPos(make_float3(p.x, p.y, p.z));
                uint hash = calcGridHash(gridPos);


                // store grid hash and particle index
                gridParticleHash[index] = hash;
                gridParticleIndex[index] = index;
            }
```

********* Función CalcGridPos**********

```
// calculate grid hash value for each particle


    __device__ int3 calcGridPos(float3 p)
    {
        int3 gridPos;
        gridPos.x = floor((p.x - params.worldOrigin.x) / params.cellSize.x);
        gridPos.y = floor((p.y - params.worldOrigin.y) / params.cellSize.y);
        gridPos.z = floor((p.z - params.worldOrigin.z) / params.cellSize.z);
        return gridPos;
    }


    // calculate address in grid from position (clamping to edges)
```

```
__device__ uint calcGridHash(int3 gridPos)
{
    gridPos.x = gridPos.x & (params.gridSize.x-1);  // wrap grid, assumes size is
power of 2
    gridPos.y = gridPos.y & (params.gridSize.y-1);
    gridPos.z = gridPos.z & (params.gridSize.z-1);
    return __umul24(__umul24(gridPos.z, params.gridSize.y), params.gridSize.x) +
__umul24(gridPos.y, params.gridSize.x) + gridPos.x;
}
```

********* Función reorderdata**********

```
//rearrange particle data into sorted order, and find the start of each cell
       ' in the sorted hash array

        __global__
        void reorderDataAndFindCellStartD(uint   *cellStart,        // output:
        cell start index
                                          uint   *cellEnd,          // output:
        cell end index
                                          float4 *sortedPos,        // output:
        sorted positions
                                          float4 *sortedVel,        // output:
        sorted velocities
                                          uint   *gridParticleHash, // input:
        sorted grid hashes
                                          uint   *gridParticleIndex,// input:
        sorted particle indices
                                          float4 *oldPos,           // input:
        sorted position array
                                          float4 *oldVel,           // input:
        sorted velocity array
                                          uint    numParticles)
        {
            extern __shared__ uint sharedHash[];    // blockSize + 1 elements
            uint index = __umul24(blockIdx.x,blockDim.x) + threadIdx.x;


            uint hash;


            // handle case when no. of particles not multiple of block size
            if (index < numParticles)
            {
                hash = gridParticleHash[index];
```

```
    // Load hash data into shared memory so that we can look
    // at neighboring particle's hash value without loading
    // two hash values per thread
    sharedHash[threadIdx.x+1] = hash;


    if (index > 0 && threadIdx.x == 0)
    {
        // first thread in block must load neighbor particle hash
        sharedHash[0] = gridParticleHash[index-1];
    }
}
```

ANEXO A

********* Función original collideSpheres**********

```
__device__
float3 collideSpheres(float3 posA, float3 posB,
                      float3 velA, float3 velB,
                      float radiusA, float radiusB,
                      float attraction)
{
    // calculate relative position
    float3 relPos = posB - posA;


    float dist = length(relPos);
    float collideDist = radiusA + radiusB;


    float3 force = make_float3(0.0f);


    if (dist < collideDist)
    {
        float3 norm = relPos / dist;
        // relative velocity
        float3 relVel = velB - velA;
        // relative tangential velocity
        float3 tanVel = relVel - (dot(relVel, norm) * norm);


        // spring force
        force = -params.spring*(collideDist - dist) * norm;
        // dashpot (damping) force
        force += params.damping*relVel;
        // tangential shear force
        force += params.shear*tanVel;
        // attraction
        force += attraction*relPos;
    }


    return force;
}
```