

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
CIUDAD - CHILE



“IMPLEMENTACIÓN EFICIENTE EN GPU DE
MÉTODO DE LATTICE BOLTZMANN,
APLICANDO HERRAMIENTAS DE
COOPERATIVE GROUPS”

MONSERRAT FIGUEROA LAGOS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Claudio Torres
Profesor Correferente: Alvaro Salinas

Enero - 2022

AGRADECIMIENTOS

Mis más sinceros agradecimientos a las personas que me han apoyado y ayudado a lo largo de este tiempo, gracias al profesor Torres quien me guió a lo largo de esta memoria, a mis amigas, amigos y familia, mil gracias por tenerme paciencia.

RESUMEN

Resumen—

El trabajo realizado presenta un algoritmo para la implementación de el método de Lattice Boltzmann con enfoque push en un solo Kernel de CUDA haciendo uso de las herramientas de grupos cooperativos presentados por NVIDIA. Este algoritmo soluciona la problemática de sincronización de los threads entre los pasos de streaming y collision de manera eficiente sin generar condiciones de carrera ni bloqueos.

La solución propuesta muestra un rendimiento comparable a soluciones ya desarrolladas anteriormente, mejorando incluso el tiempo de ejecución promedio del programa en nuevas arquitecturas de GPUs NVIDIA.

Palabras Clave— CUDA; LBM; Cooperative Groups

ABSTRACT

Abstract— The following work presents an algorithm for the implementation of the Lattice Boltzmann method with a push approach in a single CUDA Kernel using the Cooperative Groups library developed by NVIDIA. This algorithm efficiently solves the thread synchronization problem between the streaming and collision steps without generating race conditions or deadlock.

The proposed solution shows a performance comparable to previously developed solutions, even improving the average execution time of the program on new NVIDIA GPU architectures. .

Keywords— CUDA; LBM; Cooperative Groups

GLOSARIO

CUDA: Compute Unified Device Architecture.

GPU: Graphics Processing Unit.

CPU: Central Processing Unit.

LBM: Lattice Boltzmann Method.

SWE: Shallow Water Equations.

ÍNDICE DE CONTENIDOS

RESUMEN	II
ABSTRACT	II
GLOSARIO	III
ÍNDICE DE FIGURAS	VI
ÍNDICE DE TABLAS	VII
INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	2
1.1 Objetivos	3
1.1.1 Objetivo general	3
1.1.2 Objetivos Específicos	3
CAPÍTULO 2: Marco Conceptual	4
2.1 Shallow Water Equations	4
2.2 Metodo de lattice Boltzmann	5
2.2.1 Esquema pull y push	8
2.3 Condiciones de Borde	9
2.3.1 Condiciones de Borde Abiertas	9
2.3.2 Condiciones de tipo Bounce-Back	10
2.4 CUDA	10
2.4.1 Cooperative Groups	12
2.5 Estado del Arte	14
2.5.1 Método de Lattice Boltzmann	14
2.5.2 Sincronización de threads	15
CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	17
3.1 Algoritmo base	17
3.2 Organización de datos	19
3.3 Sincronización	19
3.3.1 Sincronización Global utilizando grid.sync()	19
3.3.2 Sincronización por sección	21
3.3.3 Matriz de coordinación	24
3.4 Efecto Coriolis	32
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	34
4.1 Especificaciones de ambiente de pruebas	36
4.2 Resultado Numérico	37

4.3 Experimentos	39
4.3.1 Comparación de diferentes desarrollos con matriz de coordinación .	39
4.3.2 Comparación de las diferentes propuestas de solución	42
4.3.3 Comparación al utilizar distintos tamaños de grupos cooperativos .	43
4.4 Comparación de propuesta seleccionada con trabajos anteriores	47
4.5 Efecto Coriolis	51
CAPÍTULO 5: CONCLUSIONES	54
5.1 Trabajo Futuro	55
ANEXOS	56
REFERENCIAS BIBLIOGRÁFICAS	57

ÍNDICE DE FIGURAS

1	Modelo D2Q9	6
2	Comparación de streaming entre esquema pull y push	8
3	Streaming en nodo de borde abierto	9
4	Condición de borde de tipo Bounce-Back	10
5	Ejecución de programa bajo modelo de programación de CUDA. Fuente: CUDA C Programming Guide v11.2.	11
6	Asignación de ejecución de bloques en diferentes SM. A mayor cantidad de multiprocesadores, mayor es el paralelismo alcanzado. Fuente: CUDA C Programming Guide v11.2	12
7	Niveles de sincronización de grupos de distinto tamaño. Fuente: Nvidia Developer Technical Blog [1]	13
8	Ejemplo de grilla de 6×6 dividida en secciones iguales de 3×3	21
9	Modelo de concatenación de sub-filas por sección: Representación de la grilla en dos dimensiones con su indexación correspondiente en el arreglo de una dimensión.	22
10	Modelo de separación de las filas de cada sección: Representación de la grilla en dos dimensiones con su indexación correspondiente en el arreglo de una dimensión.	23
11	Streaming de componentes de función de distribución en modelo de concatenación de sub filas	23
12	Streaming de componentes de función de distribución en modelo de cambio de lectura	24
13	Matriz de coordinación para grilla dividida en secciones de dimensión 4×4	26
14	Grupos solapados	28
15	Grupos no solapados	30
16	Estado inicial de caso de prueba	35
18	Error en el nivel del agua calculado vs numero de Iteración	37

17	Comparación Salinas-Propuesta	38
19	Comparación de tiempo de ejecución de las diferentes configuraciones para la matriz de coordinación utilizando grupos solapados	40
20	Comparación de tiempo de ejecución de las diferentes configuraciones para la matriz de coordinación utilizando grupos separados	41
21	Comparación de tiempos de ejecución de diferentes las propuestas de solución	43
22	Comparación de rendimiento según tamaño de grupo para propuesta de grupos solapados	45
23	Comparación de rendimiento según tamaño de grupo para propuesta de grupos separados	46
24	Comparación de performance entre las soluciones desarrolladas en GPU Nvidia RTX 2060	48
25	Comparación de performance entre las soluciones desarrolladas en GPU Tesla K80	49
26	Comparación de performance entre las soluciones desarrolladas en GPU Tesla T4	50
27	Comparación de efecto Coriolis en la Simulación	52

ÍNDICE DE TABLAS

1	Parametros de ejecución	35
2	Especificaciones del ambiente de ejecución de experimentos	36
3	Especificaciones del ambiente de ejecución	36
4	Especificaciones del ambiente de ejecución	37
5	Resultados de pruebas de versiones del modelo con grupos solapados . . .	40
6	Resultados de pruebas de versiones del modelo con grupos separados . .	41
7	Resultados de pruebas de comparación de algoritmos	42

8	Resultados de pruebas utilizando diferentes tamaños de grupos para algoritmo con grupos Solapados	44
9	Resultados de pruebas utilizando diferentes tamaños de grupos para algoritmo con grupos separado	45
10	Resultados de pruebas comparativas realizadas en tarjeta Nvidia RTX 2060	47
11	Resultados de pruebas comparativas realizadas en tarjeta Nvidia Tesla K80	48
12	Resultados de pruebas comparativas realizadas en tarjeta Nvidia Tesla T4	49
13	Tiempos de inicializacion de datos	51
14	Parametros de ejecución	51

INTRODUCCIÓN

La simulación computacional se ha convertido en una herramienta indispensable en un gran número de disciplinas, tales como, astrofísica, física de partículas, ciencia del clima, mecánica de fluidos, entre muchos otros, y continúa creciendo, ya que los modelos computacionales pueden proveer información que es difícil de obtener a través de experimentos físicos. Muchas simulaciones computacionales presentan un gran grado de paralelismo a nivel de datos y al ser computacionalmente intensivas se convierten en procesos prometedores para la ejecución a nivel de GPU.

En el contexto de esta memoria se aborda una simulación del área de dinámica de fluidos enfocada específicamente a aguas en océanos abiertos, cuyo uso puede extrapolarse a simulación de tsunamis, un área de estudio de gran importancia debido a las características geográficas de Chile además de los efectos del cambio climático y su efecto sobre el mar. Específicamente se presenta una implementación del método de Lattice Boltzmann que es un método numérico utilizado para la resolución de un sistema de ecuaciones diferenciales conocidas como Shallow Water Equations, el cual describe la evolución de la superficie de un fluido en el tiempo y el espacio.

Este método presenta características que lo vuelven un perfecto candidato para ser implementado con programación paralela, por lo que en esta memoria se desarrolló en CUDA un kernel que ejecuta los pasos del algoritmo, utilizando un enfoque distinto al momento de programar, diferenciándolo de implementaciones ya existentes, buscando sacar provecho Cooperative Groups una librería nueva de CUDA enfocada en la sincronización de threads.

El siguiente trabajo presenta el proceso de desarrollo, mostrando las diferentes problemáticas y soluciones que fueron presentándose a lo largo de la implementación para finalizar con diferentes pruebas, comparando la solución implementada con otros desarrollos realizados anteriormente.

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

LBM es un método que de manera general consiste fundamentalmente en discretizar un campo de flujo, simulando el comportamiento estadístico de una colección de partículas ficticias sin tener que recurrir al cálculo de ecuaciones diferenciales explícitamente. Debido a la localidad de las interacciones de las partículas, este algoritmo permite ser implementado tomando ventaja de las tecnologías de computación paralela, por lo que existen múltiples implementaciones de este método utilizando diferentes herramientas de computación paralela, como lo es CUDA.

Al realizar una aplicación utilizando CUDA existen muchos factores que pueden afectar el desempeño y que representan un desafío al momento de programar, como el manejo de memoria, condiciones de carrera, configuraciones en la ejecución, control de flujo, optimización en instrucciones aritméticas y de acceso de memoria. Aunque hay múltiples trabajos dedicados a abordar este tipo de problemáticas de diferentes formas, la tecnología de las tarjetas gráficas avanza rápidamente, lo que conlleva a la actualización constante por parte de la librería de CUDA y la aparición de nuevas herramientas que permiten ampliar la forma de programar y abordar ciertos problemas.

Si bien gran parte de la computación de LBM es enfocada de manera local en el propio nodo, también es necesaria la interacción entre nodos vecinos en un paso denominado *streaming*, lo que implica escritura de información por parte de un nodo, que será necesitada por otro para continuar con el resto del algoritmo, por lo cual el aspecto de la sincronización entre threads es importante para evitar condiciones de carrera.

Para evitar las condiciones de carrera se han desarrollado múltiples versiones de LBM en CUDA utilizando diferentes enfoques. Sin embargo, hasta el momento no se han publicado desarrollos que aprovechen Cooperative Groups, una librería por parte de Nvidia que presenta diferentes funcionalidades enfocadas en la sincronización de threads, las cuales podrían impactar positivamente en el rendimiento del programa desarrollado.

Ya que el contexto del desarrollo de un programa que ejecute LBM es la simulación de tsunamis, es importante considerar las dimensiones y condiciones físicas que se tienen que considerar en la simulación, por lo cual también en el desarrollo del kernel del algoritmo se tiene que tener en cuenta la fuerza debido al efecto Coriolis que impacta en el movimiento del mar.

En resumen, en esta memoria se abordará 2 problemas:

1. Análisis del uso de *Cooperative Groups* en la implementación del método Lattice Boltzmann aplicado a dinámica de fluidos.

2. Incluir el término de fuerza de Coriolis en la simulación computacional, requerido para simulaciones numéricas de océanos que cubran gran parte del territorio.

1.1. Objetivos

Los objetivos de esta memoria son los siguientes

1.1.1. Objetivo general

Realizar una nueva implementación del Método de Lattice Boltzmann en el marco simulación de tsunamis en CUDA utilizando Cooperative Groups, considerando una integración eficiente de las condiciones de borde y la fuerza de Coriolis en la formulación.

1.1.2. Objetivos Específicos

1. Implementar en CUDA de cada una de las variaciones del algoritmo aplicando la funcionalidad de Cooperative Groups, considerando además el efecto de la fuerza de Coriolis.
2. Ejecutar un conjunto de pruebas que permita comparar el desempeño de las distintas variaciones de la implementación.
3. Comparar con implementación existente en trabajo anterior.
4. Concluir sobre el efecto del uso de nuevas funcionalidades de CUDA en el desempeño de ejecución.

CAPÍTULO 2

Marco Conceptual

2.1. Shallow Water Equations

Las ecuaciones de aguas tranquilas, en inglés shallow water equations (SWE), son un sistema de ecuaciones diferenciales parciales obtenidas al integrar verticalmente las ecuaciones de conservación de masa y momento de Navier-Stokes [25]. Este sistema de ecuaciones diferenciales parciales describe la evolución de la superficie de un fluido, asumiendo que la escala horizontal del dominio es mucho más grande que la escala vertical. Las ecuaciones son expresadas de la siguiente manera

$$\frac{\partial h}{\partial t} + \frac{\partial(hu)}{\partial x} + \frac{\partial(hv)}{\partial y} = 0 \quad (1)$$

$$\frac{\partial(hu)}{\partial t} + \frac{\partial}{\partial x} \left(hu^2 + \frac{1}{2}gh^2 \right) + \frac{\partial(huv)}{\partial y} = -gh \frac{\partial b}{\partial x} + \mathbf{F}_1 \quad (2)$$

$$\frac{\partial(hv)}{\partial t} + \frac{\partial(huv)}{\partial x} + \frac{\partial}{\partial y} \left(hv^2 + \frac{1}{2}gh^2 \right) = -gh \frac{\partial b}{\partial y} + \mathbf{F}_2 \quad (3)$$

donde $h(x, y, t)$ es la altura del fluido, $u(x, y, t)$ y $v(x, y, t)$ corresponden a las componentes x e y de la velocidad respectivamente, $b(x, y)$ representa la elevación de la batimetría, g es la constante de aceleración de gravedad y $\mathbf{F}_i$ son las componentes del resultado de todas las fuerzas que afecta el sistema, tales como la fuerza ejercida por el viento, el roce con el fondo o el efecto Coriolis, este último termino, será considerado en la implementación en esta memoria y viene dado por la siguiente ecuación [31]:

$$F_i = \begin{cases} C_c h v, & i = x \\ -C_c h u, & i = y, \end{cases} \quad (4)$$

donde C es el coeficiente de Coriolis y viene dado por $2\omega \sin \phi$ donde ω es la velocidad angular de la tierra y ϕ es la latitud.

Para la resolución de estas ecuaciones es necesario las condiciones iniciales de velocidad $u(x, y, 0) = u_0(x, y)$ y nivel del agua $h(x, y, 0) = h(x, y)$, así también es necesario determinar las condiciones de borde que serán detalladas mas adelante.

2.2. Metodo de lattice Boltzmann

El método de Lattice Boltzmann (LBM) es un modelo cinético a escala mesoscópica, basado en los métodos de Lattice Gas Automata, que se utiliza para simulaciones de flujos de fluidos además de ser considerado como una buena manera de obtener la resolución de las ecuaciones de Shallow Water de manera numérica [8]. La idea fundamental es que los fluidos pueden ser imaginados como grandes cantidades de pequeñas partículas moviéndose aleatoriamente. Usualmente, es discretizado en una grilla cartesiana en el cual cada nodo representa un conjunto de partículas que se distribuyen hacia distintas direcciones.

El intercambio de momento y energía es obtenido por el proceso de dispersión de las partículas y las colisiones entre ellas. LBM simplifica la idea original de la dinámica de gases al reducir el número de partículas y confinarlas a los nodos de una grilla.

El método de Lattice Boltzmann ha presentado mayor aceptación en comparación con otros métodos, ya que los modelos son más estables numéricamente [30], además de incluir implícitamente los criterios de convergencia y se adecuan fácilmente a las condiciones de frontera que son complejas o que varían con el tiempo. Este método ha sido utilizado ampliamente en campo de simulación de fluidos.

Existen múltiples modelos de lattice para seleccionar, uno de los mas comunes y que ha presentado buenos resultados es el modelo D2Q9 [27], que representa una grilla de dos dimensiones tal como se aprecia en la Figura 1 . Como se puede ver este modelo las partículas están restringidas a moverse en 9 direcciones, estas velocidades son referidas como las velocidades microscópicas. La magnitud de las velocidades $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ y $\mathbf{e}_4$ es igual a lu/ts , mientras que la de $\mathbf{e}_5, \mathbf{e}_6, \mathbf{e}_7$ y $\mathbf{e}_8$ corresponde a $\sqrt{2}lu/ts$. Así mismo, es posible apreciar que los vectores $\mathbf{e}_a$ están definidos de la siguiente manera:

$$\mathbf{e}_\alpha = \begin{cases} (0, 0) & \alpha = 0 \\ \left(\cos \frac{(\alpha-1)\pi}{4}, \sin \frac{(\alpha-1)\pi}{4} \right) & \alpha = 1, 2, 3, 4 \\ \left(\sqrt{2} \cos \frac{(\alpha-1)\pi}{4}, \sqrt{2} \sin \frac{(\alpha-1)\pi}{4} \right) & \alpha = 5, 6, 7, 8 \end{cases} \quad (5)$$

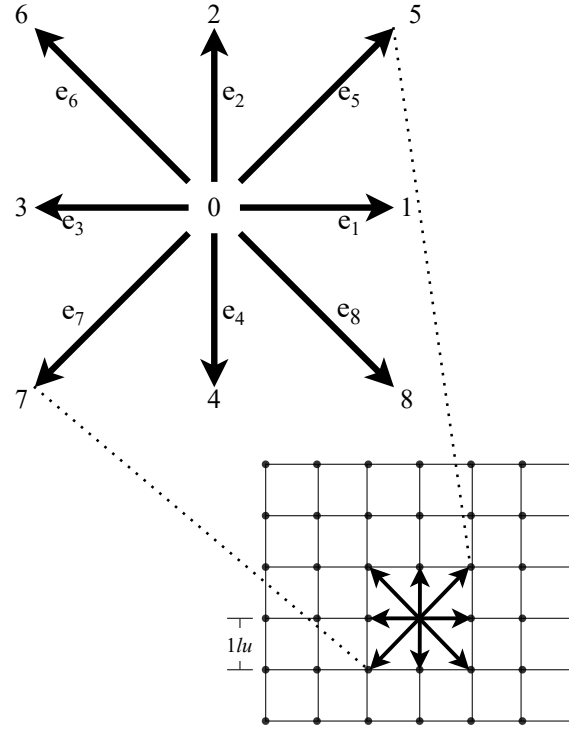


Figura 1: Modelo D2Q9

De manera similar, a cada uno de los 9 conjuntos de partículas en un nodo se le asocia una función de probabilidad de distribución f_i , dicha función equivale a pensar en un histograma que representa la frecuencia de ocurrencia con la que las partículas de un determinado nodo toma una de las 9 velocidades posibles.

Algorithm 1 LBM

- 1: Calculate initial values for f_{eq} and S
 - 2: **for** $t = 0$ to N **do**
 - 3: **for** Every node in lattice **do**
 - 4: Collision Step
 - 5: Streaming Step
 - 6: Boundary Condition Step
 - 7: Recalculate Macroscopic variables
 - 8: **end for**
 - 9: **end for**
-

Esta función se actualiza en cada iteración del algoritmo 1 que consiste esencialmente en la realización de dos pasos llamados *streaming* y *collision*. El primero de ellos corresponde a la etapa que representa el movimiento de las partículas, desplazando las partículas de un nodo a otro de acuerdo a los valores de la función de distribución para cada una de las direcciones posibles. Por otra parte, la etapa de *collision* representa

la interacción entre las partículas, pudiendo cambiar sus velocidades direccionales al llegar al siguiente nodo.

LBM puede modelarse en una ecuación, utilizando la aproximación BGK (Bhatnagar-Gross-Krook) [7] para la implementación de la colisión, originando la siguiente ecuación de Lattice Boltzmann:

$$f_\alpha(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t) = f_\alpha(\mathbf{x}, t) - \underbrace{\frac{1}{\tau} (f_\alpha(\mathbf{x}, t) - f_\alpha^{(eq)}(\mathbf{x}, t) + S_\alpha(\mathbf{x}, t))}_{\text{collision}} \quad (6)$$

streaming

donde τ es el denominado tiempo de relajación y $f_a^{(eq)}$ corresponde a la función de equilibrio y S_α corresponde al término fuente que se define por la siguiente ecuación

$$S_\alpha = \begin{cases} 0, & \alpha = 0 \\ \frac{\Delta t}{6e^2} \mathbf{e}_{\alpha i} \mathbf{F}_i(\mathbf{x}, t) - \omega_\alpha \frac{g \bar{h}(\mathbf{x}, t)}{e^2} [b(\mathbf{x} + \mathbf{e}_\alpha \Delta t) - b(\mathbf{x})], & \alpha = 1, \dots, 8 \end{cases} \quad (7)$$

Donde $\mathbf{F}_i(\mathbf{x}, t)$ es la componente i del término de fuerza que ejerce sobre el sistema, $b(\mathbf{x})$ es la batimetría del nodo, g es el valor de la aceleración gravitacional, ω_α corresponde a los valores de peso para que se satisfaga el principio de bien balanceado que en este caso corresponden a $\omega_\alpha = 1/3$ para $\alpha = 1, 2, 3, 4$ y $\omega_\alpha = 1/12$ para $\alpha = 5, 6, 7, 8$ y $\bar{h}(\mathbf{x}, t)$ se define de la siguiente manera:

$$\bar{h}(\mathbf{x}, t) = \frac{1}{2} [h(\mathbf{x} + \mathbf{e}_a \Delta t, t + \Delta t)] \quad (8)$$

h corresponde a la variable macroscópica que representa a la profundidad del fluido y puede ser recuperada de la siguiente manera:

$$h = \sum_{\alpha=0}^8 f_\alpha \quad (9)$$

Por otro lado, la variable macroscópica que corresponde al vector de velocidad $\mathbf{u}$, puede ser recuperado por

$$\mathbf{u} = \frac{1}{h} \sum_{\alpha=0}^8 \mathbf{e}_\alpha f_\alpha \quad (10)$$

Por último, la ecuación de equilibrio mencionada anteriormente, fue propuesta por Zhuo [31] para la resolución de SWE y es definida de la siguiente manera

$$f_{\alpha}^{(eq)} = \begin{cases} h - \frac{5gh^2}{6e^2} - \frac{2h}{3e^2} \mathbf{u}_i \mathbf{u}_i, & \alpha = 0, \\ \frac{gh^2}{6e^2} + \frac{h}{3e^2} \mathbf{e}_{\alpha i} \mathbf{u}_i + \frac{h}{2e^4} \mathbf{e}_{\alpha i} \mathbf{e}_{\alpha j} \mathbf{u}_i \mathbf{u}_j - \frac{h}{6e^2} \mathbf{u}_i \mathbf{u}_i, & \alpha = 1, 2, 3, 4 \\ \frac{gh^2}{24e^2} + \frac{h}{12e^2} \mathbf{e}_{\alpha i} \mathbf{u}_i + \frac{h}{8e^4} \mathbf{e}_{\alpha i} \mathbf{e}_{\alpha j} \mathbf{u}_i \mathbf{u}_j - \frac{h}{24e^2} \mathbf{u}_i \mathbf{u}_i, & \alpha = 5, 6, 7, 8 \end{cases} \quad (11)$$

2.2.1. Esquema pull y push

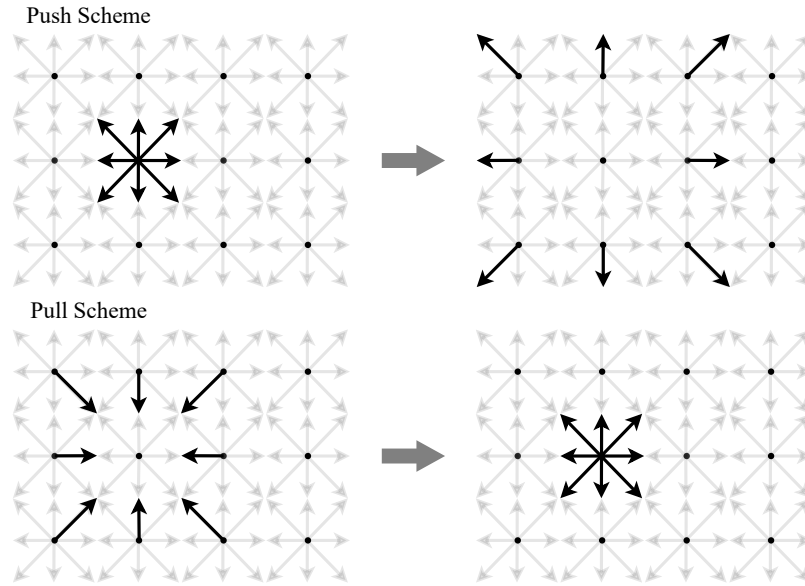


Figura 2: Comparación de streaming entre esquema pull y push

Debido a la atención que ha recibido LBM a lo largo de los años y debido a su característica algorítmica que abre un gran campo para la optimización en diferentes ámbitos, por ejemplo existen diferentes formas de implementar el método, ya sea disminuyendo la cantidad de pasos como ahorrar en el uso de memoria [26]

En esta memoria se hace énfasis en el paso de *streaming* donde existen diferentes técnicas para implementarlo. Este paso usualmente es independiente de la estructura de datos que representa el lattice, sin embargo, dependiendo de como los nodos están guardados en memoria, la complejidad en que se determinan los vecinos puede variar. Los dos esquemas de implementación más conocidos y al ser intuitivos de implementar son los esquemas *push* y *pull*. En el método *push* la colisión ocurre antes que la propagación y este paso se enfoca en la escritura sobre los nodos vecinos, tal como se puede ver en la Figura 2, mientras que en el método *pull* se voltean ambos pasos principales del algoritmo y la propagación ocurre antes de la colisión en donde el nodo en lugar de

escribir sobre el resto lee la información de sus vecinos para realizar los cálculos locales. Ambos métodos tienen igual número de accesos a memoria e implementación similar que requiere dos grid, una de entrada y otra de salida, en el algoritmo.

2.3. Condiciones de Borde

Cuando un nodo se encuentra en el límite del dominio, después de ejecutarse el paso de *streaming* se dejan varias componentes de la función de distribución sin valor actualizado dado a la ausencia de algunos nodos vecinos. Por esa razón es necesario encontrar los valores faltantes utilizando las condiciones de borde necesarias. En esta memoria se utilizan las siguientes:

2.3.1. Condiciones de Borde Abiertas

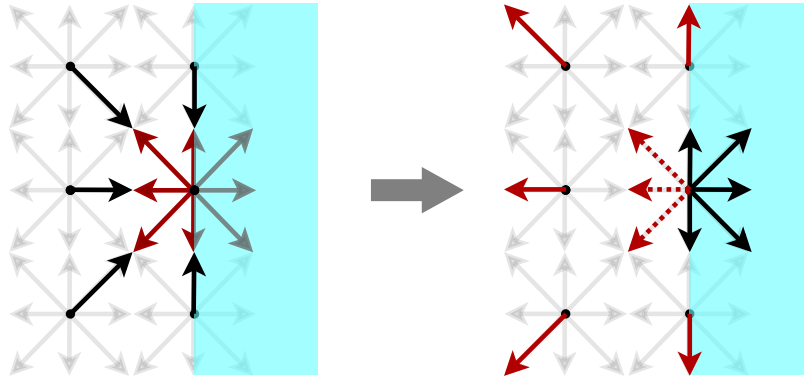


Figura 3: Streaming en nodo de borde abierto

Las condiciones de borde abiertas son de gran importancia en las simulaciones de flujos de aguas en un contexto de océano abierto. Este tipo de borde representa la continuidad del flujo fuera del dominio sin que las olas sean detenidas o choquen con algún objeto sólido, sin embargo, al no tener todos los nodos vecinos como parte del dominio, posterior al paso del *streaming* al nodo borde en cuestión le quedarían faltando componentes de la función de distribución, tal como se ejemplifica en la Figura 3.

Si bien no existe un consenso de como enfrentar este tipo de borde, en esta memoria se utiliza el método implementado en [22] que cumple con la condición de estar bien balanceado conocida como condición de borde “*do-nothing*” (hacer nada), que implica en solo agregar el término de colisión a las componentes anteriores del nodo. Esto

quedaría expresado de la siguiente forma

$$\hat{f}_\alpha(\mathbf{x}, t + \Delta t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau} (f_\alpha(\mathbf{x}, t) - f_\alpha^{(eq)}(\mathbf{x}, t)) \quad (12)$$

Donde $\hat{f}_\alpha$ son las nuevas funciones calculadas de las componentes faltantes las cuales tenían anteriormente como valor la función f_α

2.3.2. Condiciones de tipo Bounce-Back

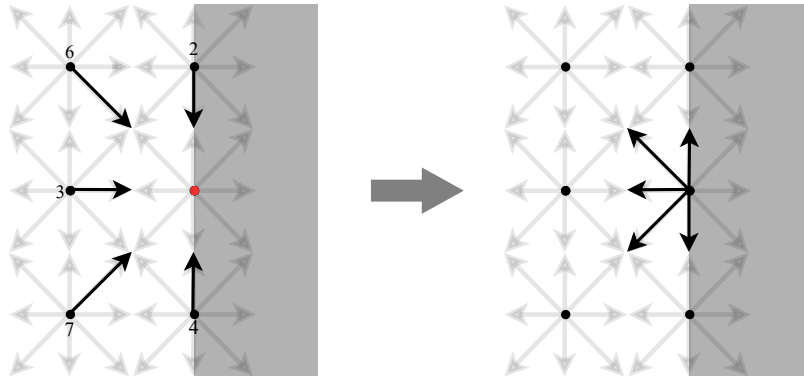


Figura 4: Condición de borde de tipo Bounce-Back

Las condiciones de borde de tipo *Bounce-Back* representan la presencia de un muro u objeto sólido que impide el paso de fluido a través de él. Esto supone una velocidad nula en la frontera del dominio, por lo que su implementación se reduce a reflejar las componentes de la función de distribución en dirección opuesta, como es posible apreciar en la Figura 4.

Siguiendo el ejemplo anterior, considerando que f_β^α , es la función de distribución de la componente β propagada por nodo vecino ubicado en dirección α , las componentes reflejadas se obtienen de la siguiente manera:

$$\hat{f}_3(\mathbf{x}, t + \Delta t) = f_1^3 \quad \hat{f}_6(\mathbf{x}, t + \Delta t) = f_8^6 \quad \hat{f}_7(\mathbf{x}, t + \Delta t) = f_5^7$$

2.4. CUDA

CUDA es una plataforma de computación y modelo de programación paralela introducido por NVIDIA [10] en noviembre del 2006 que permite aprovechar al máximo

sus unidades de procesamiento gráfico (GPU), permitiendo resolver múltiples modelos computacionales de gran complejidad de manera más eficiente que en CPU.

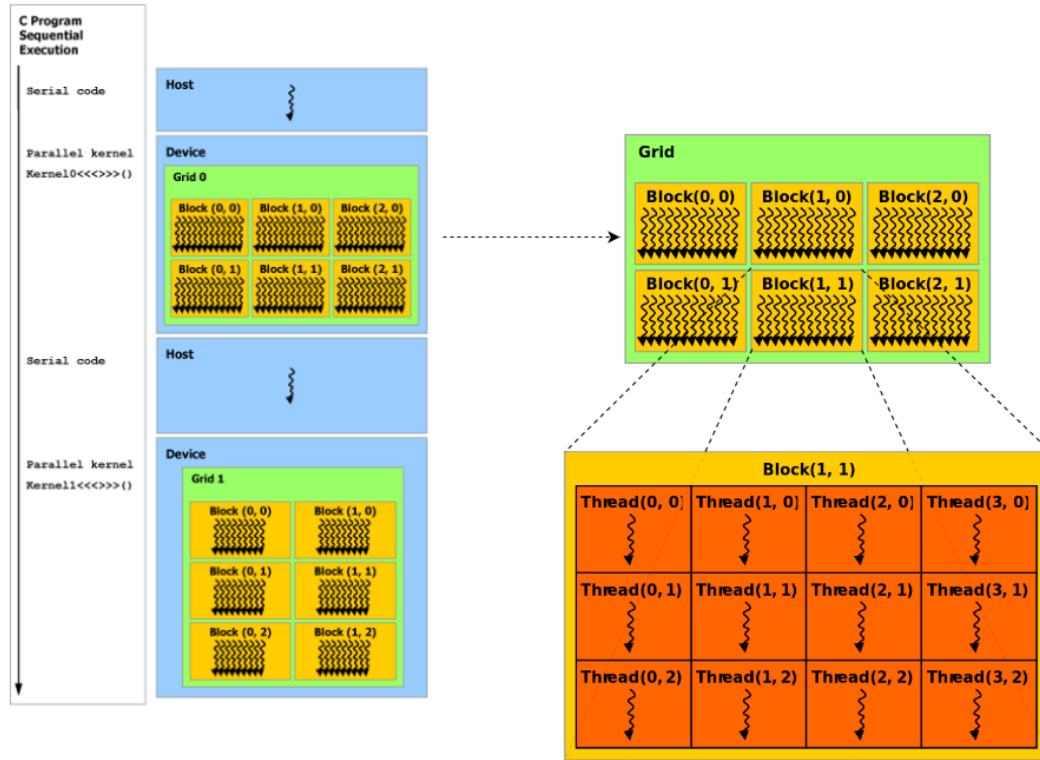


Figura 5: Ejecución de programa bajo modelo de programación de CUDA.

Fuente: CUDA C Programming Guide v11.2.

CUDA ofrece un ambiente de software que permite a los desarrolladores utilizar C++ como lenguaje de alto nivel, este ambiente permite al programador definir funciones de C++ llamadas kernels, que al ser invocadas son ejecutadas N veces en N diferentes hebras o threads de CUDA, al contrario de las funciones comunes de C++, estas hebras son ejecutadas en la GPU que actúa como coprocesador del resto programa de C++ que se ejecuta de manera serial en la CPU.

Las hebras son agrupadas en una unidad llamada bloque, mientras que el conjunto de bloques es agrupado en una unidad llamada grid, tal como se puede ver graficado en la Figura 5. Tanto los bloques como la grid pueden tener 1,2 o 3 dimensiones, aunque es común utilizar una sola dimensión adaptando la estructura de los datos de entrada del programa de manera conveniente. Esta descomposición ayuda a los *threads* a organizarse para resolver subproblemas al mismo tiempo que promueve la escalabilidad, ya que cada bloque puede ser asignado para ser ejecutado por cualquier multiprocesador de streaming (SM) disponible en la GPU. Cada bloque solo puede ser procesados por un único SM, aunque este último puede tener asignado múltiples bloques dependiendo de

su capacidad.



Figura 6: Asignación de ejecución de bloques en diferentes SM. A mayor cantidad de multiprocesadores, mayor es el paralelismo alcanzado. Fuente: CUDA C Programming Guide v11.2

Los threads de CUDA pueden acceder a los datos a través de múltiples espacios de memoria durante la ejecución del kernel. Cada thread tiene un espacio de memoria local propio que puede ser accedido solo por este, por otro lado, los threads pueden compartir información a través de la memoria global o compartida, la diferencia entre ambas es que la última presenta mayor velocidad de para acceder pero tiene la restricción que solo puede ser accedida por threads de un mismo bloque mientras que a la memoria global pueden acceder todos los threads de la grid, pero el acceso requiere más tiempo.

2.4.1. Cooperative Groups

Anteriormente, el modelo de programación de CUDA ha previsto solo una funcionalidad para sincronizar un conjunto de hebras cooperativas, la función `syncthreads()` que provee una “barrera” para todos los threads que pertenecen a un bloque. Sin embargo, se ha visto que desarrolladores de CUDA necesitan definir sincronizaciones de grupos de diferentes tamaños de un bloque, para alcanzar un mejor rendimiento, mejorar diseño del código y crear código reutilizable para la sincronización de grupos de threads cooperativos.

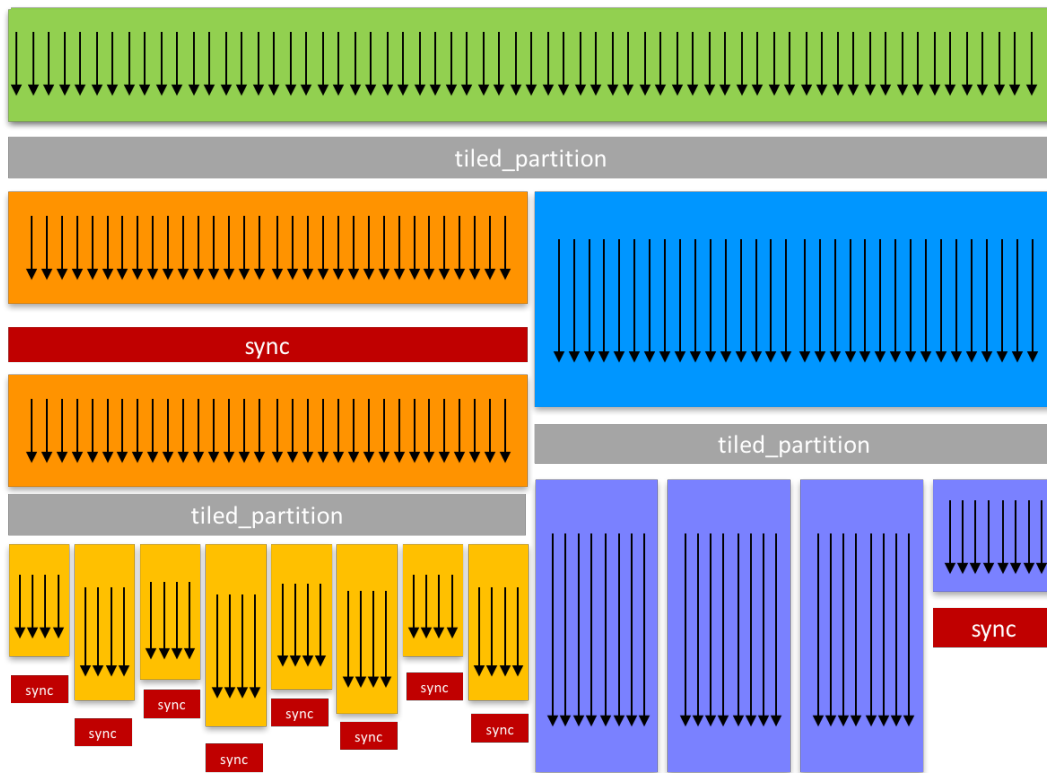


Figura 7: Niveles de sincronización de grupos de distinto tamaño.
Fuente: Nvidia Developer Technical Blog [1]

Cooperative Groups es una extensión del modelo de programación de CUDA introducido en CUDA 9 en 2017 [13], su propósito principal es poder organizar en distinto nivel grupos de threads sincronizados entre ellos, lo que ayudaría a desarrollar mejores y más eficientes descomposiciones paralelas.

Actualmente, los niveles de organización provistos por esta extensión son los siguientes [2]:

- Grupo de bloque: Cooperative groups explicita como un nuevo tipo de datos el concepto de bloque que ha existido anteriormente, permitiendo la sincronización de manera similar a lo que hacía syncthreads
- Grupo de Grid: Este grupo representa todos los threads ejecutados por una sola grid
- Grupo Multi-Grid: Representa a todos los threads ejecutados en todos los dispositivos ejecutados en un kernel cooperativo multi-device.
- Thread Block Tile: Es un conjunto de threads que resultan de la partición exacta de un bloque, por lo tanto un tile no puede pertenecer a más de un bloque.

Actualmente, en las versiones estables de CUDA los grupos de este tipo pueden ser de tamaño inferior a 32.

2.5. Estado del Arte

2.5.1. Método de Lattice Boltzmann

LBM se posicionó como una alternativa a los métodos tradicionales de dinámica de fluidos tales como el método de diferencias finitas o el método de los elementos finitos [29] para realizar simulaciones computacionales que se relacionen con la resolución de ecuaciones diferenciales. Este método se dedica a resolver la ecuación de Lattice Boltzmann en lugar de resolver directamente las ecuaciones de Navier Stokes. Ha recibido atención en las últimas décadas debido a sus ventajas computacionales, ya que realiza operaciones locales, lo que permite aprovechar arquitecturas computacionales paralelas para realizar las simulaciones.

Fue propuesto por primera vez los autores Guy McNamara y Gianluigi Zanetti [20] en 1988 donde se basaron en el modelo de Lattice Gas automata, posteriormente en 1989 Higuera y Jimenez [15] aplicaron la ecuación de Lattice Boltzmann en problemas de hidrodinámica, sin embargo la resolución de las ecuaciones de Shallow Water utilizando LBM fue visto después por Salmon en 1999 [23], quien fue de los primeros en insertar el modelo D2Q9, y posteriormente por Dellar en 2002 [9], mientras que en 2004 Jian Guo Zhou [31] publicó un trabajo exhaustivo del método, siendo éste último la base para la implementación de LBM en esta memoria, tanto en la ecuación de equilibrio utilizada como en los términos de fuerza de Coriolis utilizada. Así mismo, en esta memoria también se toma como base la adaptación realizada por Salinas que se basó también en el trabajo de Zhou, la cual se detalla más adelante.

Paralelamente, desde el principio de los 2000 se ha estudiado la implementación de LBM en GPU en el trabajo de Li et al [18]. Desde entonces múltiples trabajos han propuesto técnicas de optimización para GPU tales como Mawson y Revell [19], Habich et al. [12] y Herschlag et al. [14] que han enfocado sus trabajos en el diseño de la memoria debido a que el rendimiento viene afectado principalmente por el ancho de banda de esta, sin embargo el lanzamiento de CUDA por NVIDIA facilitó el desarrollo de los algoritmos en GPU, convirtiéndose en una de las herramientas más utilizadas para las recientes implementaciones de LBM en GPU, trabajos importantes a mencionar son Tolke que en 2010 [24] que detalla diferentes propiedades del modelo de programación en CUDA tales como la configuración de la grid y la distribución de trabajo en los distintos kernel, por otro lado, Bernaschi [6] presenta un trabajo con dominios irregulares analizando el manejo de memoria e implementando accesos coalescentes.

La implementación de LBM de esta memoria está principalmente basada en los dos

trabajos realizados por Salinas, el primero en 2018 [22] propone una variante del método de Lattice Boltzmann para soportar también condiciones de borde abiertas y al mismo tiempo ser bien balanceado y en el segundo trabajo en 2019 [21] presenta un exhaustivo análisis de rendimiento en la implementación de su adaptación de LBM integrando diferentes condiciones de borde, implementación que se realizó utilizando un solo kernel con enfoque *pull*, donde consideran diferentes ámbitos de optimización como la asignación de la clase de un nodo o el manejo de condiciones.

2.5.2. Sincronización de threads

Uno de los problemas que existen al programar en CUDA u otros frameworks de computación paralela son los bugs de concurrencia, como los *deadlocks* y las condiciones de carrera. Este último problema ocurre cuando más de thread accede a una dirección de memoria en común, lo que implica un comportamiento incorrecto e indeterminado difícil de detectar, lo que ha llevado a la implementación de diferentes herramientas y métodos de análisis de código para detectar bugs como GPUVerify [5] o PUG [17]. Para evitar ese problema es importante una barrera de sincronización entre los threads y permitiendo separar y ordenar los accesos a memoria.

Anteriormente a la existencia de Cooperative Groups en CUDA la principal funcionalidad para la sincronización de threads era `syncthreads()` que está limitada a la sincronización por bloque y `threadfence()` [3] que permite sincronizar a base de la escritura en memoria bloqueando el thread que escribe hasta que los demás threads de la grid puedan leer la información escrita por este, aunque esta última no es muy usada a diferencia de `syncthreads()` debido al *overhead* que provoca afectando negativamente el rendimiento del programa[11]. Así mismo, también es común utilizar operaciones atómicas para la comunicación entre threads y evitar las condiciones de carrera. Un trabajo a destacar es [28] donde implementa dos maneras de barrera de sincronización entre bloques utilizando operaciones atómicas y accesos a memoria global, en este trabajo se obtiene que el uso de una barrera de sincronización dentro del kernel presenta mejor rendimiento que la sincronización realizada por la CPU al terminar un kernel y llamar a la otra parte.

Ya que Cooperative Groups es una tecnología relativamente reciente, no existen muchos trabajos que midan su rendimiento de manera satisfactoria, uno de ellos es el trabajo de Hu et al. [16] en que Presentan Altis una suite para medir *benchmarks* enfocada en la computación GPGPU, con su suite midieron el algoritmo SRAD para procesamiento de imágenes en CUDA probando imágenes pequeñas con sincronización de un grupo grid de Cooperative Groups, sin embargo, no pudieron obtener resultados consistentes para poder concluir si la incorporación de Cooperative Groups afectaba negativamente o no. Otro trabajo a mencionar, realizado por Aslam et al. en 2020 [4] presentan diferentes implementaciones del método de Jacobi para la resolución de sistema de ecuaciones, para lo cual también prueban con Cooperative Groups y operaciones atómicas para la

sincronización y comunicación de threads, el trabajo muestra que la coordinación a nivel de grid no afecta en negativamente el rendimiento, sin embargo, las implementaciones con operaciones atómicas sí se mostraron con menor rendimiento que las otras.

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

Como se mencionó anteriormente, la propuesta de solución de esta memoria está basada en los trabajos realizados por Salinas nombrados en el capítulo anterior. A partir de [22] se rescata la adaptación de LBM para la resolución de SWE, contemplando la misma manera de abordar las condiciones de borde propuesta en el trabajo. Así mismo, se decide utilizar las consideraciones de optimización presentadas en [21], donde se puede ver que la propuesta que obtiene mejor rendimiento es aquella que clasifica los tipos de nodo utilizando valores binarios.

Sin embargo, ya que se busca evaluar el uso de Cooperative Groups al implementar LBM en CUDA, la implementación propuesta en los trabajos mencionados donde se utiliza el enfoque pull con un solo kernel, no resulta útil, ya que no requiere sincronización entre grupos de threads además de la sincronización global que existe al salir del kernel, por lo tanto en este trabajo se optará por utilizar el enfoque push convencional de LBM, que requiere sincronización de los threads posterior al paso de streaming. Además, este último enfoque presenta un rendimiento y complejidad similar al método push, por lo que se puede comparar el impacto en el uso de Cooperative Groups.

3.1. Algoritmo base

En la propuesta de solución se busca implementar un time step del método de Lattice Boltzmann en un solo kernel utilizando el enfoque push, cuyo algoritmo general se muestra en 2, donde entre las líneas 6 y 16 se ejecuta los pasos de Collision y Streaming y entre las líneas 18 y 29 se ejecutan los pasos donde se calculan las condiciones de borde y las variables macroscópicas. En el algoritmo se puede apreciar la problemática que afronta el enfoque push, la condición de carrera que se genera tras el paso del streaming, ya que si el thread que lee un nodo, sigue con los cálculos posteriores, es posible que lea los valores en f_2 antes que sus vecinos hayan escrito en este, por lo que es completamente necesario esperar a que los demás escriban los valores de distribución f nuevos para que pueda continuar con los cálculos posteriores y para eso debe existir un punto de sincronización entre los threads que está representado en la línea 17.

En trabajos anteriores se ha visto que es común dividir este proceso en dos kernel distintos sincronizando globalmente los threads entre las llamadas a estos, lo que genera un impacto en el rendimiento del programa, en cambio, en esta memoria se busca tomar ventaja de las funcionalidades de sincronización de la librería Cooperative Groups de CUDA para resolver este problema. Sin embargo, ya que la librería es relativamente nueva y sigue teniendo varias limitaciones, se decidió probar diferentes métodos de sincronización en la implementación del kernel.

Algorithm 2 Push Scheme Kernel

```

1: Time Step( $f_2, f_1, h, b, u, v, \mathbf{e}, e, \tau, g, L_x, L_y, \mathbf{F}$ )
2:  $i = \text{Thread ID}$ 
3: if  $i < L_x \times L_y$  then
4:    $x = i \% L_x$ 
5:    $y = i \% L_y$ 
6:   for  $\alpha = 0, 1, \dots, 8$  do
7:      $f_{eq}[\alpha] = EDF(h[x][y], v[x][y], u[x][y], g, \alpha, \mathbf{e}, e)$ 
8:     if  $\alpha$  needs open boundary condition then then
9:        $f_2[x][y][\alpha] = (1 - 1/\tau)f_1[x][y][\alpha] + f_{eq}[\alpha]/\tau$ 
10:    else
11:       $x_n = \text{neighborX}(x, \alpha)$ 
12:       $y_n = \text{neighborY}(y, \alpha)$ 
13:       $S = \text{SourceTerm}(\alpha, h, b, \mathbf{e}, e, \mathbf{F})$ 
14:       $f_2[x_n][y_n][\alpha] = (1 - 1/\tau)f_1[x][y][\alpha] + f_{eq}[\alpha]/\tau + S$ 
15:    end if
16:  end for
17:  Sync()
18:  for  $\alpha = 0, 1, \dots, 8$  do
19:    if  $\alpha$  needs Bounce Back condition then then
20:      if  $\alpha \in \{1, 2, 5, 6\}$  then
21:         $f_2[x_n][y_n][\alpha] = f_2[x_n][y_n][\alpha + 2]$ 
22:      else
23:         $f_2[x_n][y_n][\alpha] = f_2[x_n][y_n][\alpha - 2]$ 
24:      end if
25:    end if
26:  end for
27:   $h[x][y] = \text{CalculateH}(f_2[x][y])$ 
28:   $v[x][y] = \text{CalculateV}(\mathbf{e}, f_2[x][y], h[x][y])$ 
29:   $u[x][y] = \text{CalculateU}(\mathbf{e}, f_2[x][y], h[x][y])$ 
30: end if

```

3.2. Organización de datos

Una de las primeras consideraciones que se tienen al programar con CUDA es la forma de organizarlos datos (en este caso la información relacionada con los nodos), ya que una cualidad que impacta directamente con el rendimiento del programa es que los threads accedan a memoria global de forma coalescente. Este término se refiere a combinar múltiples accesos a memoria en una sola transacción o en las menos posibles, permitiendo así el uso óptimo de la banda ancha de la memoria global. Esto ocurre cuando en dispositivos con un *compute capability* mayor que 2.0. los threads de un *warp* (grupo de 32 threads) acceden a 128 bytes de memoria consecutiva, esto significa que 32 flotantes de precisión simple siempre pueden ser accedidos en una sola transacción, mientras que 32 flotantes de precisión doble pueden ser accedidos en dos transacciones. Para lograr esto, el acceso debe ser alineado y secuencial.

LBM es un algoritmo que usa un *data layout* o grilla de dos dimensiones y el manejo de memoria es un problema importante para optimizar. En este caso, los datos de la grilla serán almacenados como arreglos en memoria global de una dimensión concatenando las filas de la grilla, debido a que permiten un acceso más rápido por parte de los threads a diferencia de los arreglos en memoria de dos dimensiones o más, por lo tanto como mínimo, para implementar el algoritmo para LBM se utilizarán diferentes arreglos de una dimensión que almacenen los valores del nivel del agua h , la batimetría b , las velocidades u_x y u_y y la función de distribución f_α de cada nodo.

Ya que la función de distribución f_α presenta múltiples componentes, existen diferentes maneras para almacenarlos en memoria, las más comunes son conocidas como estructura de arreglos (SoA) y arreglos de estructuras (AoS). En el trabajo anterior queda demostrado que utilizando la organización tipo estructura de arreglos (SoA) permite el acceso coalescente al leer cada componente de la función

3.3. Sincronización

Como se mencionó previamente, la principal problemática en la propuesta de solución de esta memoria es la sincronización de los thread. Las opciones implementadas serán detalladas a continuación:

3.3.1. Sincronización Global utilizando `grid.sync()`

La forma más sencilla de desarrollar una sincronización global de los threads de CUDA en un punto intermedio es utilizando el grupo cooperativo que representa la grid en que se ejecuta al kernel y hacer uso de la funcionalidad proporcionada por la librería `cooperative groups` `grid.sync()` que sincroniza todos los thread pertenecientes a la

grid en cualquier punto de ejecución dentro del kernel, por ende es posible implementar el enfoque push sin la necesidad de separar el método en dos kernel distintos de manera sencilla. Entonces la implementación de esta forma de sincronización se puede ver en el algoritmo 3

Algorithm 3 Push Scheme Kernel Grid Sync

```

1: Time Step( $f_2, f_1, h, b, u, v, \mathbf{e}, e, \tau, g, L_x, L_y, \mathbf{F}$ )
2:  $i = \text{Thread ID}$ 
3: group = this_grid()
4: if  $i < L_x \times L_y$  then
5:    $x = i \% L_x$ 
6:    $y = i \% L_y$ 
7:   for  $\alpha = 0, 1, \dots, 8$  do
8:      $f_{eq}[\alpha] = EDF(h[x][y], v[x][y], u[x][y], g, \alpha, \mathbf{e}, e)$ 
9:     if  $\alpha$  needs open boundary condition then then
10:       $f_2[x][y][\alpha] = (1 - 1/\tau)f_1[x][y][\alpha] + f_{eq}[\alpha]/\tau$ 
11:    else
12:       $x_n = \text{neighborX}(x, \alpha)$ 
13:       $y_n = \text{neighborY}(y, \alpha)$ 
14:       $S = \text{SourceTerm}(\alpha, h, b, \mathbf{e}, e, \mathbf{F})$ 
15:       $f_2[x_n][y_n][\alpha] = (1 - 1/\tau)f_1[x][y][\alpha] + f_{eq}[\alpha]/\tau + S$ 
16:    end if
17:  end for
18:  group.sync()
19:  for  $\alpha = 0, 1, \dots, 8$  do
20:    if  $\alpha$  needs Bounce Back condition then then
21:      if  $\alpha \in \{1, 2, 5, 6\}$  then
22:         $f_2[x_n][y_n][\alpha] = f_2[x_n][y_n][\alpha + 2]$ 
23:      else
24:         $f_2[x_n][y_n][\alpha] = f_2[x_n][y_n][\alpha - 2]$ 
25:      end if
26:    end if
27:  end for
28:   $h[x][y] = \text{CalculateH}(f_2[x][y])$ 
29:   $v[x][y] = \text{CalculateV}(\mathbf{e}, f_2[x][y], h[x][y])$ 
30:   $u[x][y] = \text{CalculateU}(\mathbf{e}, f_2[x][y], h[x][y])$ 
31: end if

```

Sin embargo, se presenta una gran desventaja, esta funcionalidad solo puede ser ejecutada en un kernel que es lanzado por la función `cudaLaunchCooperativeKernel()`, esta funcionalidad limita la cantidad de threads a ejecutar a la cantidad máxima de threads que pueden estar activos al mismo tiempo en una SM multiplicada por la cantidad de SM disponibles, esta cantidad puede variar dependiendo del tamaño del bloque y la cantidad de registros del kernel a ejecutar.

3.3.2. Sincronización por sección

Una de las características de LBM es la localidad de este algoritmo permitiendo que los cálculos de un nodo sean independientes del resto lo que favorece su computación paralela, por eso una de las desventajas que presenta la sincronización global mostrada anteriormente es la perdida de independencia en la ejecución del algoritmo a cada nodo, ya que un nodo solo tiene la necesidad de coordinarse con los 8 vecinos que están a su alrededor, sin embargo con este tipo de sincronización, el thread que procesa ese nodo está obligado a esperar que el todo resto de la grid termine su ejecución, incluyendo threads de nodos que se encuentran muy lejos.

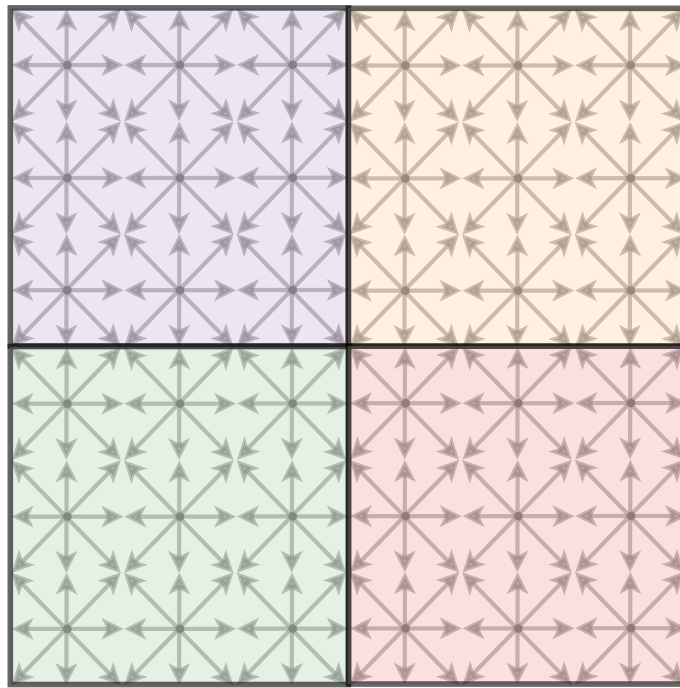


Figura 8: Ejemplo de grilla de 6×6 dividida en secciones iguales de 3×3

Para evitar la necesidad de una sincronización global, se ideó una forma de dividir la grilla en secciones iguales, tal como se puede ver en la Figura 8, en donde se ejecute una sincronización local solamente entre los threads de cada sección, utilizando Cooperative Groups. De esta forma, los threads no tienen la necesidad de sincronizarse con los threads de secciones lejanas a este.

Para el uso de Cooperative groups solo pueden ser agrupados threads que se encuentren seguidos en el mismo bloque (exceptuando el grupo tipo grid que contiene a todos los bloques) por lo cual es fundamental organizar el orden en que los datos de los nodos son representados o leídos en memoria, ya que cada thread de un grupo debería leer a los nodos miembros de la sección correspondiente. Con la división de la grilla, seguir

con la forma de organizar y leer los datos de la manera convencional que se implementó anteriormente imposibilita que los threads de cada sección pertenezcan al mismo grupo, por consecuencia se pensó en dos modelos distintos que solucionaran este problema, el primero cambiando la organización de como los datos son estructurados en memoria y la otra cambiando el orden en que los threads leen los datos sin tener que cambiar la estructura de estos.

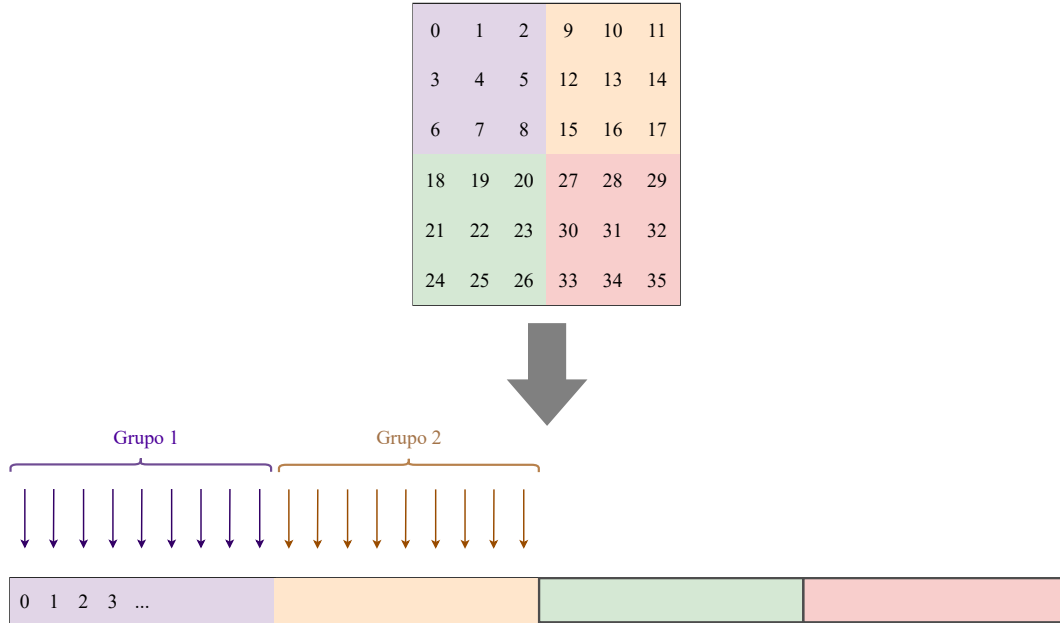


Figura 9: Modelo de concatenación de sub-filas por sección: Representación de la grilla en dos dimensiones con su indexación correspondiente en el arreglo de una dimensión.

Conservar la lectura ordenada La primera solución a este problema es cambiar el orden en que se organiza los valores de la información de nodos en los arreglos principales. Para eso se deben ordenar los datos de una sección de tal manera que estos queden posicionados consecutivamente en el arreglo para que puedan ser leídas directamente por el grupo de threads correspondiente, es decir las sub-filas correspondientes a una sección quedarían concatenadas juntas y posteriormente les seguirían las sub-filas concatenadas de la sección siguiente tal como lo ilustra la Figura 9, de esta manera el identificador de los threads coincide con la indexación del arreglo de entrada. Esta forma conlleva una preparación de datos distinta en el programa principal ejecutado en la CPU, ya que los datos se irían almacenando en los arreglos de manera cruzada, pero simplifica la lectura dentro del kernel al tener la característica que cada thread acceda de manera sencilla y coalescente sobre los arreglos cuando se trata del nodo propio.

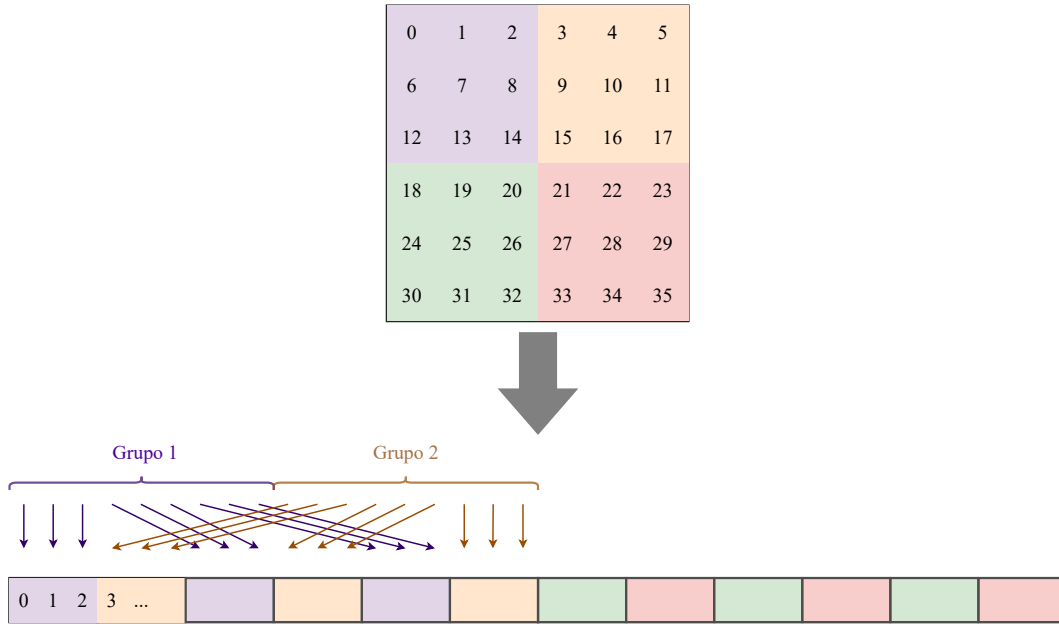


Figura 10: Modelo de separación de las filas de cada sección: Representación de la grilla en dos dimensiones con su indexación correspondiente en el arreglo de una dimensión.

Conservar los datos ordenados Otra forma de solucionar la agrupación de los threads para su sincronización evita impactar en la organización de los nodos en el arreglo. Se cambia la manera en que se acceden a estos y se seguiría organizando los datos de los nodos de manera convencional concatenando las filas de la grilla en su arreglo respectivo, tal como se explicó al comienzo del capítulo, pero los thread no accederían a direcciones continuas del arreglo sino que se separarían por cada cambio de sub-fila de la sección, esto es ilustrado en la Figura 10. Por consecuencia, dependiendo de las dimensiones de la sección puede afectar o no la coalescencia, ya que si el largo de una subfila es menor a 32, ya no se cumpliría con accesos alineados y secuenciales al momento de leer el nodo asignado, afectando el rendimiento en la ejecución.

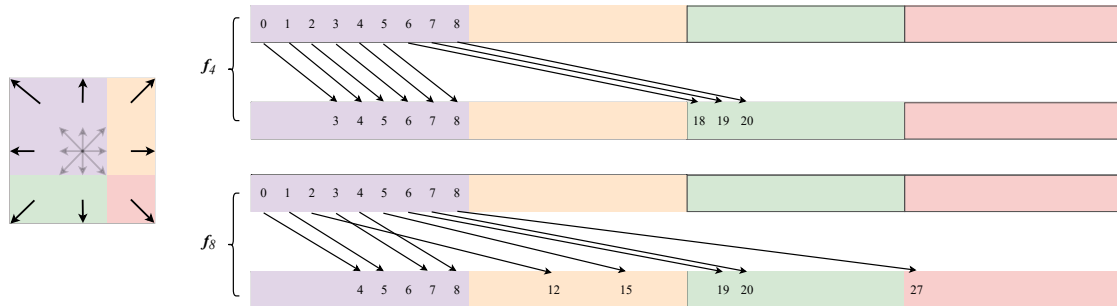


Figura 11: Streaming de componentes de función de distribución en modelo de concatenación de sub-filas

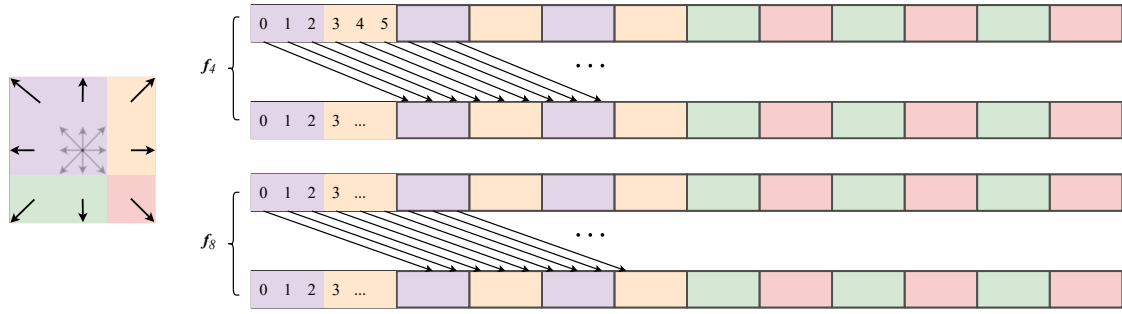


Figura 12: Streaming de componentes de función de distribución en modelo de cambio de lectura

Aunque en un principio, la primera solución pareciera la más adecuada, esta presenta una desventaja que radica en el paso de streaming del algoritmo, ya que el enfoque push implica la escritura sobre los nodos vecinos, si bien facilita la lectura del nodo actual, esta forma de orden de datos dificulta el cálculo de la ubicación y escritura en los vecinos de los nodos que se encuentran en los límites de la sección, lo que provoca que se pierda la coalescencia por completo, lo cual significaría un impacto en el rendimiento del programa. La Figura 11 representa este problema, el ejemplo muestra como sería el streaming de las componentes 4 y 8 desde los nodos de una sección a sus vecinos, los nodos que no limitan con otras secciones no presentan gran complejidad para escribir, pero se puede observar que la situación cambia cuando se trata de un nodo que limita con otra sección, sobre todo el nodo que se encuentra en una esquina, el cual tiene que escribir en cuatro secciones distintas. A diferencia de la segunda solución, donde el mismo escenario de streaming mostrado en la Figura 12 muestra una escritura sobre el arreglo notoriamente más ordenada.

Hasta este punto, es posible desarrollar el kernel de manera que la mayoría de los threads puedan sincronizarse sin problemas solo dependiendo del propio grupo, aunque aún es necesario crear una sincronización especial para los límites de cada sección, ya que estos requieren que threads de nodos de otras secciones escriban sobre ellos, una posibilidad es utilizar la sincronización global solo para estos casos y el resto de los threads solo necesitan la sincronización local, resultando el algoritmo 4 para ambos modelos, solo cambia la manera de calcular los valores de x e y . No obstante, esto no solucionaría del todo el problema de un principio que era evitar la sincronización global.

3.3.3. Matriz de coordinación

Si bien la idea de dividir por secciones busca la independencia de cada nodo con respecto a los demás nodos que se encuentran muy lejanos a este, simplemente dividir la grilla en secciones genera otro problema, por lo tanto, es necesario una forma de coordinación entre grupos distintos donde la sincronización solo involucre a los nodos de los límites

Algorithm 4 Push Scheme Kernel

```

1: Time Step( $f_2, f_1, h, b, u, v, \mathbf{e}, e, \tau, g, L_x, L_y, \mathbf{F}$ )
2:  $i = \text{Thread ID}$ 
3:  $\text{groupGrid} = \text{this\_grid}()$ 
4:  $\text{group} = \text{this.group}()$ 
5: if  $i < L_x \times L_y$  then
6:    $x = \text{calcX}(i)$ 
7:    $y = \text{calcY}(i)$ 
8:   for  $\alpha = 0, 1, \dots, 8$  do
9:      $f_{eq}[\alpha] = EDF(h[x][y], v[x][y], u[x][y], g, \alpha, \mathbf{e}, e)$ 
10:    if  $\alpha$  needs open boundary condition then then
11:       $f_2[x][y][\alpha] = (1 - 1/\tau)f_1[x][y][\alpha] + f_{eq}[\alpha]/\tau$ 
12:    else
13:       $x_n = \text{neighborX}(x, \alpha)$ 
14:       $y_n = \text{neighborY}(y, \alpha)$ 
15:       $S = \text{SourceTerm}(\alpha, h, b, \mathbf{e}, e, \mathbf{F})$ 
16:       $f_2[x_n][y_n][\alpha] = (1 - 1/\tau)f_1[x][y][\alpha] + f_{eq}[\alpha]/\tau + S$ 
17:    end if
18:  end for
19:   $\text{group.sync}()$ 
20:  if  $i$  is not a internal border then
21:    for  $\alpha = 0, 1, \dots, 8$  do
22:      if  $\alpha$  needs Bounce Back condition then then
23:        if  $\alpha \in \{1, 2, 5, 6\}$  then
24:           $f_2[x_n][y_n][\alpha] = f_2[x_n][y_n][\alpha + 2]$ 
25:        else
26:           $f_2[x_n][y_n][\alpha] = f_2[x_n][y_n][\alpha - 2]$ 
27:        end if
28:      end if
29:    end for
30:     $h[x][y] = \text{CalculateH}(f_2[x][y])$ 
31:     $v[x][y] = \text{CalculateV}(\mathbf{e}, f_2[x][y], h[x][y])$ 
32:     $u[x][y] = \text{CalculateU}(\mathbf{e}, f_2[x][y], h[x][y])$ 
33:  else
34:     $\text{groupGrid.sync}()$ 
35:    Go to line 21
36:  end if
37: end if

```

con sus respectivos vecinos implicados, sin que sea necesario una sincronización global.

Como se debe idear una manera de sincronización entre dos o más nodos de los bordes internos, los cuales no necesariamente se encuentran seguidos en el arreglo en memoria ni son leídos por threads contiguos, para sincronizarlos no es posible utilizar algún nivel de agrupamiento de Cooperative Groups además del grupo que representa a la grid (lo cual sería global), por lo que se debe recurrir a un método de comunicación entre ellos. Basándose en trabajos anteriores, es común utilizar operaciones atómicas para la coordinación de threads, que permite que los threads se turnen para acceder a una dirección de memoria para realizar algún cambio en el valor contenido en esta y evitando así las condiciones de carrera.

Para hacer uso de las operaciones atómicas es necesario definir la dirección de memoria que deben compartir los threads vecinos involucrados en la sincronización. Como son muchos los nodos involucrados, no es posible simplemente designar una sola variable ya que todos los threads de todas las secciones accederían a esa variable quitando el paralelismo al programa. Por lo que se optó por usar un arreglo que tenga la suficiente cantidad de elementos para que sean únicamente accedidos por el conjunto de vecinos respectivo.

Para eso, la solución desarrollada en esta memoria es una nueva estructura que será denominada como una **Matriz de Coordinación** que permitiría determinar al thread si los demás threads vecinos necesarios para que continúe con su ejecución hayan terminado el paso de streaming y hayan escrito los valores requeridos para la computación de los siguientes pasos del método.

1	1	1	2	1					
1	1	1	2	1	...				
1	1	1	2	1					
2	2	2	4	2					
1	1	1	2	1					
	⋮				⋮				

Figura 13: Matriz de coordinación para grilla dividida en secciones de dimensión 4×4

La matriz ilustrada en la Figura 13 determina la cantidad de threads involucrados en la sincronización, como se puede ver las casillas que tienen valor 1 representan los threads que trabajan con los nodos que se encuentran dentro de la sección lo que implica que es solo ese thread involucrado en la sincronización y no es necesario esperar, mientras que las casillas con otros valores representan los nodos que requieren coordinarse con otras secciones, este límite es compartido por los threads involucrados en la sincronización. Si bien, un nodo puede tener más de dos vecinos de otra sección, lo que implicaría más de dos threads a sincronizarse, lo que difiere de la matriz ilustrada, sin embargo, puesto que el primer nivel de sincronización es por grupo, posterior al paso de streaming, solo se requiere un thread representante del grupo tenga que coordinarse con otro thread representante de la sección vecina. Para elegir los vecinos que se sincronizan simplemente se optó por coordinar los vecinos de las secciones colindantes que se encuentren más cercanos entre sí.

Para hacer uso de la matriz de coordinación descrita, los threads involucrados acceden a la casilla de la matriz que les corresponde utilizando operaciones atómicas y restan 1 al valor de esta, lo que posteriormente significa que cuando la casilla llegue a 0 todos los threads involucrados ya habrían pasado ese punto de la ejecución del kernel y conociendo ese valor es posible para un thread del límite de la sección determinar si los vecinos la otra sección ya terminaron de escribir los valores de la función de distribución correspondiente y a la vez ser independiente de lo que ocurre con otras secciones más lejanas.

Se desarrollaron dos formas de manejar los threads y los grupos. La primera es solapando los grupos y la segunda con los grupos separados, que serán explicados a continuación

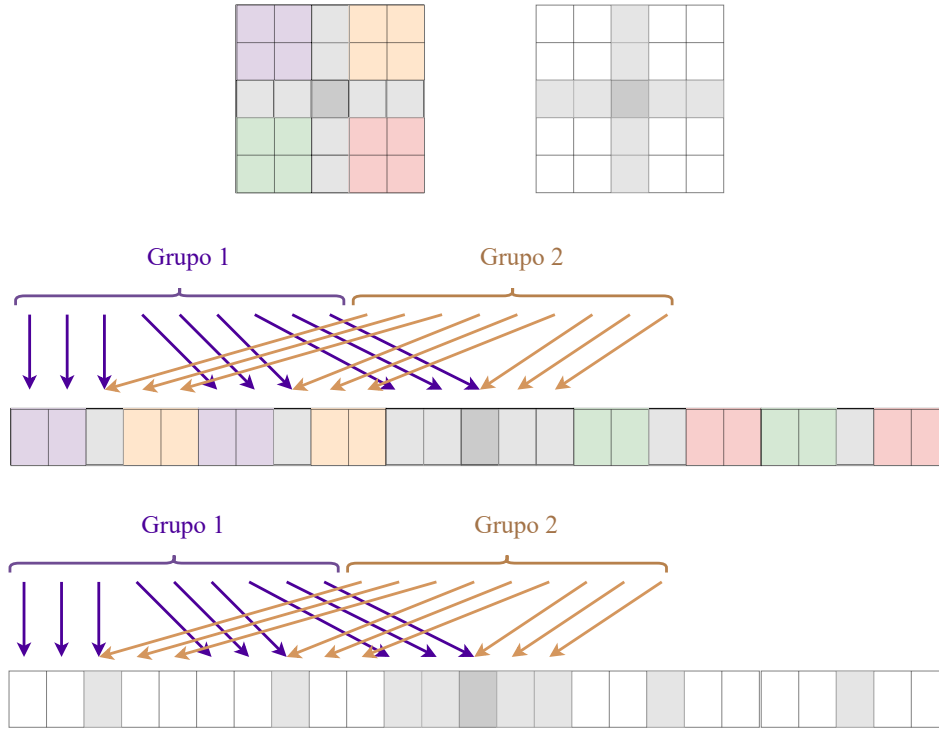


Figura 14: Grupos solapados

Grupos Solapados El concepto de grupos solapados significa que el límite entre secciones, al igual que en la matriz de coordinación, es compartido por más de un thread permitiendo que se acceda de la misma manera tanto a la información del nodo como a la matriz de coordinación, es decir, grupos de threads tienen nodos en común, tal como se muestra en la Figura 14. Por lo tanto, al comienzo de la ejecución del kernel, en los nodos límites más de dos threads ejecutan el paso del streaming lo cual no tendría impacto en la consistencia de los datos, dado que los valores con el que trabajan sería el mismo, pero sí impacta en el uso de recursos y aumenta el acceso a memoria, ya que habría más threads que nodos de la grilla trabajando.

Al llegar al punto posterior del streaming, al sincronizarse usando la matriz de coordinación, el último thread que ejecuta la operación atómica y asegura que el valor de la casilla haya quedado en 0, es el único thread que continúa con la ejecución del kernel con los pasos posteriores al streaming. Esto se ve plasmado en el algoritmo 5

Algorithm 5 Push Scheme Kernel

```

1: Time Step( $f_2, f_1, h, b, u, v, \mathbf{e}, e, \tau, g, L_x, L_y, \mathbf{F}, \mathbf{M}$ )
2:  $i$  = Thread ID
3: group = this.group()
4: if  $i < L_x \times L_y$  then
5:    $x = \text{calcX}(i)$ 
6:    $y = \text{calcY}(i)$ 
7:   for  $\alpha = 0, 1, \dots, 8$  do
8:      $f_{eq}[\alpha] = EDF(h[x][y], v[x][y], u[x][y], g, \alpha, \mathbf{e}, e)$ 
9:     if  $\alpha$  needs open boundary condition then then
10:       $f_2[x][y][\alpha] = (1 - 1/\tau)f_1[x][y][\alpha] + f_{eq}[\alpha]/\tau$ 
11:    else
12:       $x_n = \text{neighborX}(x, \alpha)$ 
13:       $y_n = \text{neighborY}(y, \alpha)$ 
14:       $S = \text{SourceTerm}(\alpha, h, b, \mathbf{e}, e, \mathbf{F})$ 
15:       $f_2[x_n][y_n][\alpha] = (1 - 1/\tau)f_1[x][y][\alpha] + f_{eq}[\alpha]/\tau + S$ 
16:    end if
17:  end for
18:  group.sync()
19:  turn = atomicSub( $\mathbf{M}[x][y], 1$ )
20:  if turn = 1 then
21:    for  $\alpha = 0, 1, \dots, 8$  do
22:      if  $\alpha$  needs Bounce Back condition then then
23:        if  $\alpha \in \{1, 2, 5, 6\}$  then
24:           $f_2[x_n][y_n][\alpha] = f_2[x_n][y_n][\alpha + 2]$ 
25:        else
26:           $f_2[x_n][y_n][\alpha] = f_2[x_n][y_n][\alpha - 2]$ 
27:        end if
28:      end if
29:    end for
30:     $h[x][y] = \text{CalculateH}(f_2[x][y])$ 
31:     $v[x][y] = \text{CalculateV}(\mathbf{e}, f_2[x][y], h[x][y])$ 
32:     $u[x][y] = \text{CalculateU}(\mathbf{e}, f_2[x][y], h[x][y])$ 
33:  end if
34: end if

```

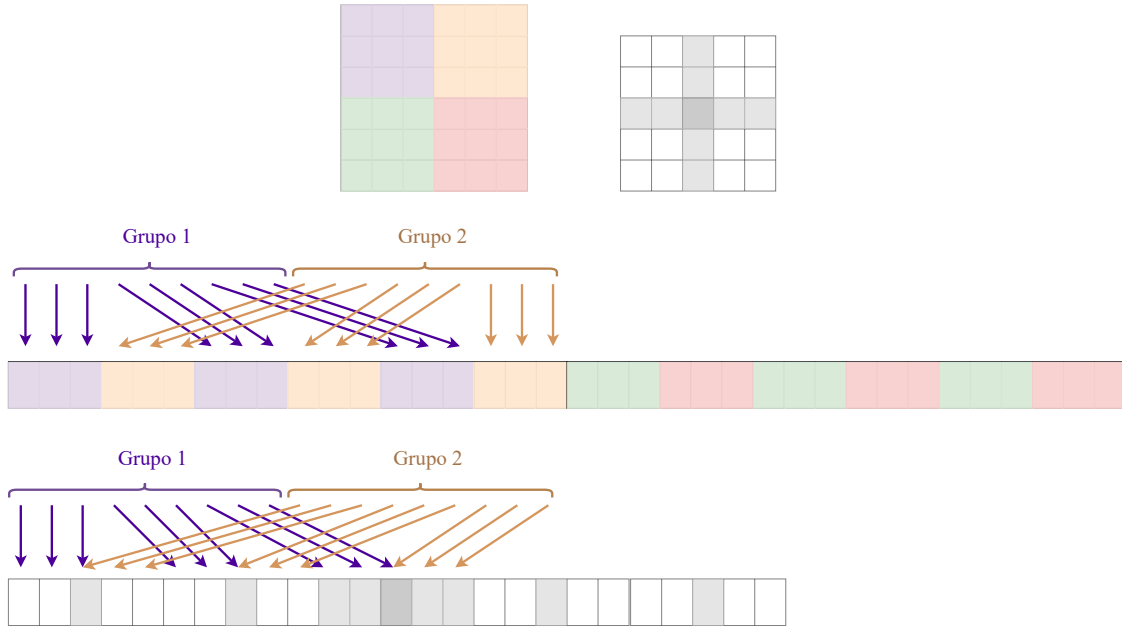


Figura 15: Grupos no solapados

Grupos Separados Por otro lado, ya que solapar los grupos trae consigo un gran gasto de recursos, sobre todo si el tamaño de los grupos son pequeños, lo que traería más bordes, por lo tanto, más threads trabajando sobre ello, se implementó la división sin solapar los grupos lo que implica que un nodo es accedido solo por un thread, sin embargo, la dinámica de acceso de la matriz de coordinación continua siendo la misma, es decir los bordes de las secciones son compartidos por más de un thread.

A diferencia del modelo solapado, posterior a la sincronización que se realiza con la matriz, no es posible que el último thread en realizar la operación atómica sea el único que siga con la ejecución, ya que esta vez los demás threads involucrados en la sincronización también deben completar los cálculos sobre el nodo que les corresponde desde el comienzo de la ejecución del kernel. De modo que en este caso todos los threads involucrados deben asegurarse que el valor de la casilla de la matriz de coordinación haya quedado en 0. Para eso, los threads que realizaron primero la operación atómica deben volver a consultar asegurándose que se encuentre en 0, ya que esa es la condición principal para poder avanzar a los siguientes pasos. El algoritmo de este proceso es mostrado en 6.

Mientras que esta solución se utilizan la misma cantidad de threads como de nodos en el dominio lo que ahorra recursos con respecto a la solución anterior, se tienen más accesos a memoria global, lo que impacta en el rendimiento de la ejecución, ya que tienen que consultar el valor en la casilla de la matriz de coordinación múltiples veces hasta que esta haya quedado en 0, además de presentar la desventaja que los grupos tienen una dependencia más profunda con sus grupos vecinos que los grupos solapados.

Algorithm 6 Push Scheme Kernel

```

1: Time Step( $f_2, f_1, h, b, u, v, \mathbf{e}, e, \tau, g, L_x, L_y, \mathbf{F}, \mathbf{M}$ )
2:  $i = \text{Thread ID}$ 
3: group = this.group()
4: if  $i < L_x \times L_y$  then
5:    $x = \text{calcX}(i)$ 
6:    $y = \text{calcY}(i)$ 
7:    $x_{matrix} = \text{calcX}(i, x)$ 
8:    $y_{matrix} = \text{calcY}(i, y)$ 
9:   for  $\alpha = 0, 1, \dots, 8$  do
10:     $f_{eq}[\alpha] = EDF(h[x][y], v[x][y], u[x][y], g, \alpha, \mathbf{e}, e)$ 
11:    if  $\alpha$  needs open boundary condition then then
12:       $f_2[x][y][\alpha] = (1 - 1/\tau)f_1[x][y][\alpha] + f_{eq}[\alpha]/\tau$ 
13:    else
14:       $x_n = \text{neighborX}(x, \alpha)$ 
15:       $y_n = \text{neighborY}(y, \alpha)$ 
16:       $S = \text{SourceTerm}(\alpha, h, b, \mathbf{e}, e, \mathbf{F})$ 
17:       $f_2[x_n][y_n][\alpha] = (1 - 1/\tau)f_1[x][y][\alpha] + f_{eq}[\alpha]/\tau + S$ 
18:    end if
19:  end for
20:  group.sync()
21:  turn = atomicSub( $\mathbf{M}[x_{matrix}][y_{matrix}], 1$ )
22:  if  $turn \leq 1$  then
23:    for  $\alpha = 0, 1, \dots, 8$  do
24:      if  $\alpha$  needs Bounce Back condition then then
25:        if  $\alpha \in \{1, 2, 5, 6\}$  then
26:           $f_2[x_n][y_n][\alpha] = f_2[x_n][y_n][\alpha + 2]$ 
27:        else
28:           $f_2[x_n][y_n][\alpha] = f_2[x_n][y_n][\alpha - 2]$ 
29:        end if
30:      end if
31:    end for
32:     $h[x][y] = \text{CalculateH}(f_2[x][y])$ 
33:     $v[x][y] = \text{CalculateV}(\mathbf{e}, f_2[x][y], h[x][y])$ 
34:     $u[x][y] = \text{CalculateU}(\mathbf{e}, f_2[x][y], h[x][y])$ 
35:  else
36:    while  $turn \neq 0$  do
37:       $turn = \mathbf{M}[x_{matrix}][y_{matrix}]$ 
38:    end while
39:    Go to line 23
40:  end if
41: end if

```

Por parte de los grupos solapados, un thread que pertenece al límite de la sección, si es que no sale elegido como ejecutor de los demás pasos del algoritmo, este termina su ejecución dentro del kernel, mientras que en este modelo los threads quedan esperando que sus grupos vecinos terminen. Debido a que los recursos de ejecución dentro de la GPU son limitados y son sujetos a muchos factores como es el tamaño del bloque, registros utilizados en el kernel o memoria compartida utilizada, existe una cantidad de warps limitada que un SM permite ejecutar al mismo tiempo, esto crea la posibilidad que el algoritmo utilizando el modelo de grupos separados provoque que grupo quede bloqueado esperando que sus grupos vecinos que están encolados esperando su ejecución sean ejecutados también.

Consideraciones en la implementación Ya que en esta memoria se busca obtener una versión optimizada del algoritmo de Lattice Boltzmann se contemplaron tres opciones diferentes para implementar el manejo de la matriz de coordinación, las cuales se detallan a continuación;

1. **Reinicio de la matriz:** Debido a que al terminar la ejecución del kernel principal los valores de la matriz de coordinación están en 0 para la siguiente iteración es necesario el reinicio de esta, lo cual se realiza copiando nuevamente a la matriz de entrada del kernel los valores desde un arreglo fijo utilizando la función `cudaMemcpy()` posterior a la ejecución de este.
2. **Acceso a memoria de la matriz:** Solo los threads de los límites de la sección acceden a la matriz, si bien al implementar un kernel también se busca disminuir la cantidad de operaciones condicionales para evitar la divergencia, diferenciando entre un thread de borde que si o sí tiene que operar sobre la matriz y un thread de un nodo interno cuya coordinación queda determinada por la sincronización del grupo, se pueden evitar realizar accesos a memoria global innecesarios lo cual podría mejorar la performance del kernel.
3. **No Reiniciar la matriz:** Para evitar el reinicio de la matriz de coordinación por cada time step ejecutado, el proceso de coordinación en lugar de restar sobre la matriz de coordinación, cada thread sumaría sobre la matriz, lo cual daría como resultado que al final de cada ejecución del kernel, los valores de esta sean un múltiplo de 1, 2 o 4 que dependen de cada iteración, lo cual permitiría determinar el tipo de nodo y así hacer el cálculo necesario para la coordinación.

3.4. Efecto Coriolis

Para agregar el término del efecto Coriolis, simplemente se agregó en el cálculo del término fuente los términos de fuerza de coriolis dados por la ecuación 4. Ya que para

el cálculo de este término se requieren los valores del coeficiente de Coriolis por cada nodo estos son necesarios en los parámetros del algoritmo.

Al igual que los términos de batimetría o altura, las latitudes de los nodos son fijas y son parte de los datos de entrada del programa implementado. Ya que la velocidad angular de la tierra también es constante, para evitar los cálculos aritméticos redundantes se decidió precalcular el término de Coriolis presente en la ecuación 4, estos valores son almacenados en un arreglo con la misma estructura que los arreglos de altura y batimetría el cual también es usado como parámetro de entrada en el kernel desarrollado.

Por lo tanto, en el kernel solo se calcula las componentes de la fuerza de Coriolis utilizando la matriz de entrada y las velocidades respectivas de cada nodo y finalmente es posible calcular el término fuente.

CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

Como validación de la solución propuesta se desarrollaron diferentes programas en C++ con un kernel principal para cada uno de los algoritmos descritos en el capítulo anterior, los cuales fueron puestos a prueba en diferentes experimentos para determinar la opción más adecuada que pueda competir con las soluciones ya existentes. Además del kernel principal, en el resto del programa se implementa los pasos auxiliares tales como lectura de datos, inicialización de arreglos, y asignaciones de memoria necesarios para el funcionamiento del kernel.

Los experimentos realizados consisten principalmente en ejecutar múltiples iteraciones y obtener el tiempo promedio de ejecución de una iteración del algoritmo de Lattice Boltzmann variando el tamaño del dominio utilizado. Se separaron los experimentos en diferentes secciones, de esta manera fue posible comparar el comportamiento de los programas y sus resultados dependiendo de ciertas variables específicas. Las secciones principales son:

1. Prueba de diferentes tipos de implementación para matriz de coordinación
2. Comparación de los diferentes algoritmos propuestos
3. Prueba de diferentes tamaño de grupos cooperativos

Posteriormente a los experimentos se escogió la mejor solución y se realizaron pruebas comparativas entre la implementación escogida, la solución propuesta por salinas considerando el enfoque pull y la solución con enfoque push con dos kernel separados como método de coordinación.

En los experimentos se midió el tiempo promedio de ejecución del kernel en 1000 iteraciones, utilizando el siguiente caso de prueba dado por la función 14 graficada en la Figura 16 que representa un plano de agua con una perturbación en el centro con una velocidad inicial igual a 0 y los 4 bordes son abiertos. Los valores utilizados para los parámetros Δx , Δy , Δt y τ son presentados en la Tabla 1 los cuales son constantes a lo largo de los experimentos.

$$Z = 3 \cdot e^{\frac{-(x^2+y^2)}{2^2}} + 20 \quad (13)$$

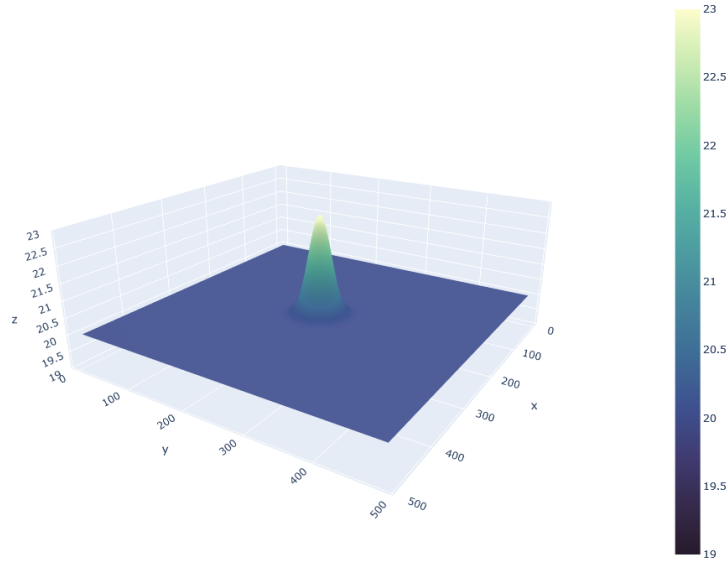


Figura 16: Estado inicial de caso de prueba

Tabla 1: Parametros de ejecución

$\Delta x = \Delta y$	2 [m]
Δt	0.05 [s]
τ	0.8
Nº iteraciones	1000

Los programas desarrollados obtienen la información de la grilla y valores de los parámetros del algoritmo a través de archivos de texto, los cuales presentan la siguiente estructura:

- Primera linea: L_x L_y D_x x_0 y_0
 - L_x : número de nodos a lo largo del eje x
 - L_y : numero de nodos a lo largo del eje y
 - D_x : distancia entre dos nodos laterales
 - x_0 : coordenada en metros del nodo de origen en el eje x
 - y_0 coordenada en metros del nodo de origen en el eje y
- Siguietes $L_x \times L_y$ líneas representan los datos de cada nodo: B W tipo

- B : Batimetria
- W : nivel del agua
- Tipo : tipo de nodo (0=seco, 1=costa, 2=mojado)

Donde los valores de B y W presentan un formato de punto flotante de precisión doble

4.1. Especificaciones de ambiente de pruebas

Los experimentos fueron ejecutados en un computador con las especificaciones mostradas en la Tabla 2. Para las pruebas comparativas de las soluciones, además, se sumaron dos maquinas para ejecutar los programas cuyas especificaciones se muestran en las Tablas 3 y 4. **Todos los experimentos realizados en las tres tarjetas fueron ejecutados utilizando flotantes de precisión doble en los cálculos respectivos del programa.**

Tabla 2: Especificaciones del ambiente de ejecución de experimentos

Modelo CPU	Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz
Sistema Operativo	Windows 10 Home
Modelo GPU	NVIDIA GeForce RTX 2060
Version de CUDA	11.5
Version de Driver	496.13
GPU Family	TU106-A
Compute Capability	7.5
Cantidad de SM	30

Tabla 3: Especificaciones del ambiente de ejecución

Modelo CPU	Intel(R) Xeon(R) CPU @ 2.20GHz
Sistema Operativo	Ubuntu 18
Modelo GPU	NVIDIA Tesla T4
Version de CUDA	11.5
Version de Driver	496.13
GPU Family	TU104
Compute Capability	7.5
Cantidad de SM	40

Tabla 4: Especificaciones del ambiente de ejecución

Modelo CPU	Intel(R) Xeon(R) CPU @ 2.20GHz
Sistema Operativo	Ubuntu 18
Modelo GPU	NVIDIA Tesla K80
Version de CUDA	11.2
Version de Driver	460.32.03
GPU Family	GK210
Compute Capability	3.7
Cantidad de SM	26

4.2. Resultado Numérico

Antes de realizar los experimentos se aseguró que los resultados obtenidos por los programas desarrollados fueran correctos por ende en esta sección se muestra el resultado general del algoritmo de Lattice Boltzmann implementado utilizando el enfoque push comparado con el resultado obtenido por la solución ya desarrollada por Salinas ¹. Los resultados mostrados fueron obtenidos en una prueba con un dominio de 976x976 nodos con los mismos datos especificados en la Tabla 1. Cabe destacar que los programas presentan el mismo código en cuanto al cálculo y solo se diferencian en la forma de sincronizar, aunque fueron todos probados, al tener los mismos resultados en esta sección solo se muestran los resultados obtenidos por la solución con el algoritmo 5 y el programa de referencia.

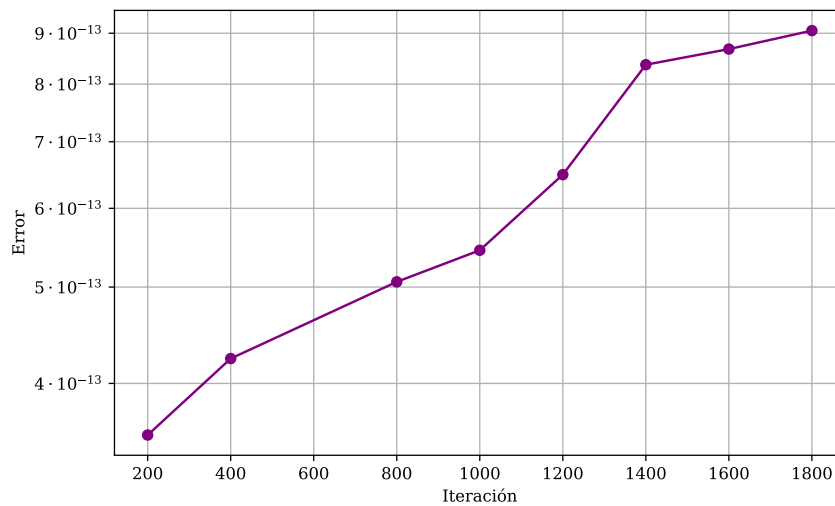


Figura 18: Error en el nivel del agua calculado vs numero de Iteración

¹Codigo disponible en <https://github.com/ASSalinasE/LBM-SWE-OBC-CUDA>

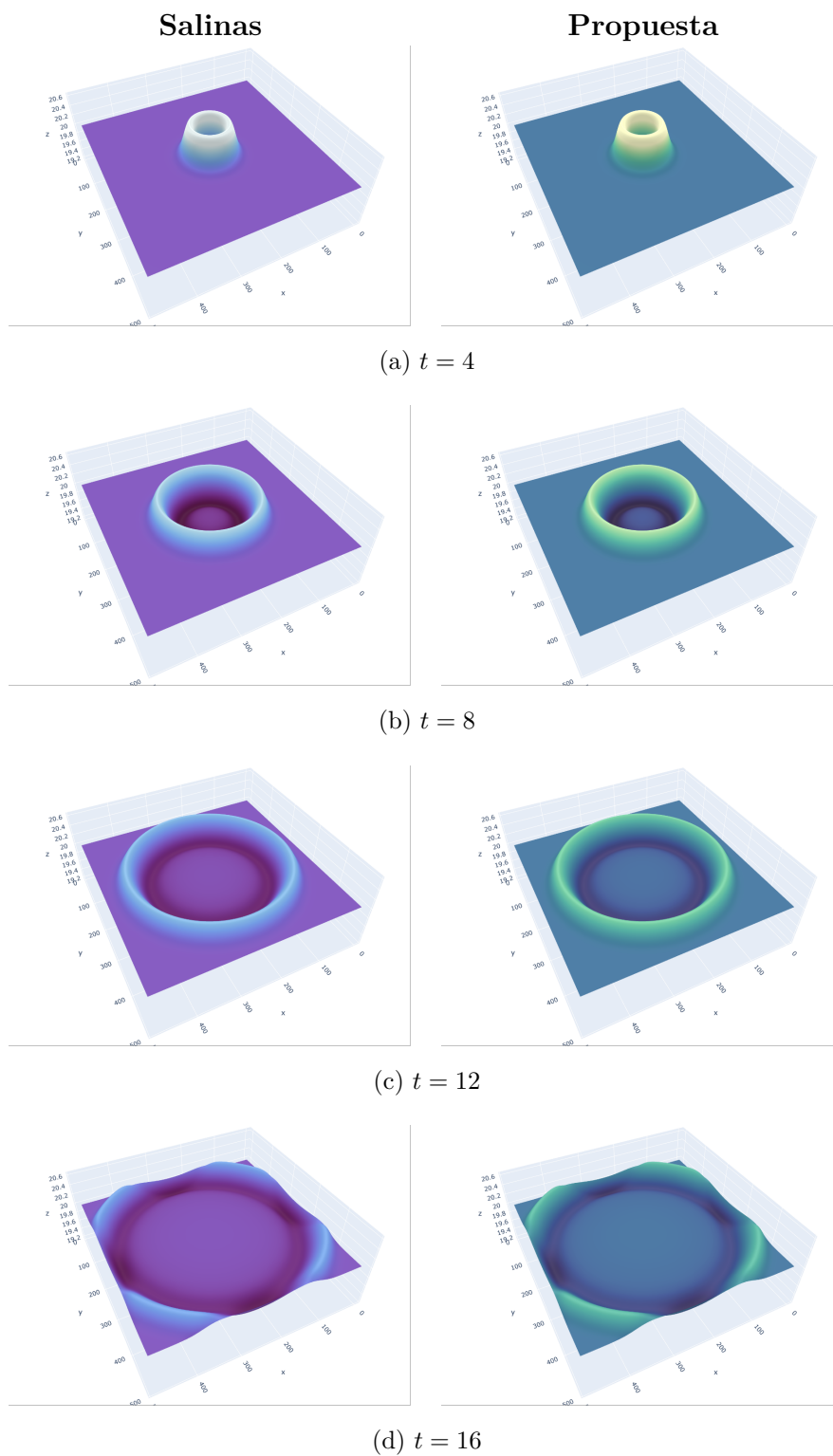


Figura 17: Comparación Salinas-Propuesta

Los resultados de las ejecuciones se pueden observar en la imagen 17 y además se calculo el error de las ejecuciones el cual es mostrado en la gráfica 18 y como se puede ver ambos programas presentan los mismos resultados numéricos de manera visible, con un error muy pequeño el cual puede deberse a diferencias algebraicas al momento de calcular diferentes valores y a la perdida de significancia que se provoca a lo largo de la ejecución del programa.

4.3. Experimentos

4.3.1. Comparación de diferentes desarrollos con matriz de coordinación

En esta instancia se realizaron pruebas de las tres versiones pensadas para el manejo de la matriz de coordinación detalladas en la sección 3.3.3 para cada uno de los modelos presentados (solapado y separado), con el propósito de escoger la que presente el mejor rendimiento de cada uno en cuanto a tiempo de ejecución. Las implementaciones escogidas serán las que se usen para realizar las pruebas en los experimentos posteriores.

El tiempo promedio de las iteraciones fue obtenido con la herramienta de perfilamiento de Nvidia nvprof que forma parte del Toolkit de CUDA. Solo se consideró el tiempo de las iteraciones del algoritmo ya que el resto del programa (lectura y preparación de los datos) es igual en todas las versiones. Para el cálculo total de tiempo de cada iteración se considera el tiempo total de ejecución del kernel, el tiempo de lanzamiento del kernel y el tiempo de copiado del arreglo para las versiones que reinician la matriz con la función `cudaMemcpy()`.

En este caso los grupos cooperativos utilizados son de tipo bloque utilizando la función `this_block` al inicializar el grupo. Para el caso de los grupos solapados el tamaño del bloque fue de 256 threads con grupos de dimensión de 16×16 . Sin embargo esta configuración no funcionó en el caso de modelo de grupos separados, ya que la ejecución del programa se quedó bloqueada en la ejecución del kernel de LBM y la ejecución no terminó incluso después de varios intentos, esto deja en evidencia la desventaja planteada en el capítulo anterior la cual implica que diferentes grupos se queden esperando la ejecución de los grupos vecinos ocupando recursos innecesariamente, por lo cual se cambio la configuración y los bloques utilizados fueron de 64 threads con grupos de dimensión 8×8 , los cuales funcionaron correctamente al realizar todas las pruebas.

Los resultados obtenidos se presentan a continuación en las tablas 5 y 6 para la versión de grupos solapados y separados respectivamente. Además se puede apreciar la comparación en el gráfico 19

Tabla 5: Resultados de pruebas de versiones del modelo con grupos solapados

Dimensiones \ Versión	C1 [ms]	C2 [ms]	C3 [ms]
256×256	0,3232	0,3630	0,3644
496×496	0,9415	0,9473	0,9426
736×736	2,8330	2,0119	1,8512
976×976	3,7308	3,6260	3,1050
1216×1216	5,9300	5,8042	4,7020
1456×1456	8,4690	8,3870	6,7157
1696×1696	11,3090	11,3087	9,0710
1936×1936	14,8200	14,7620	11,8290
2176×2176	18,4160	18,4210	14,8580

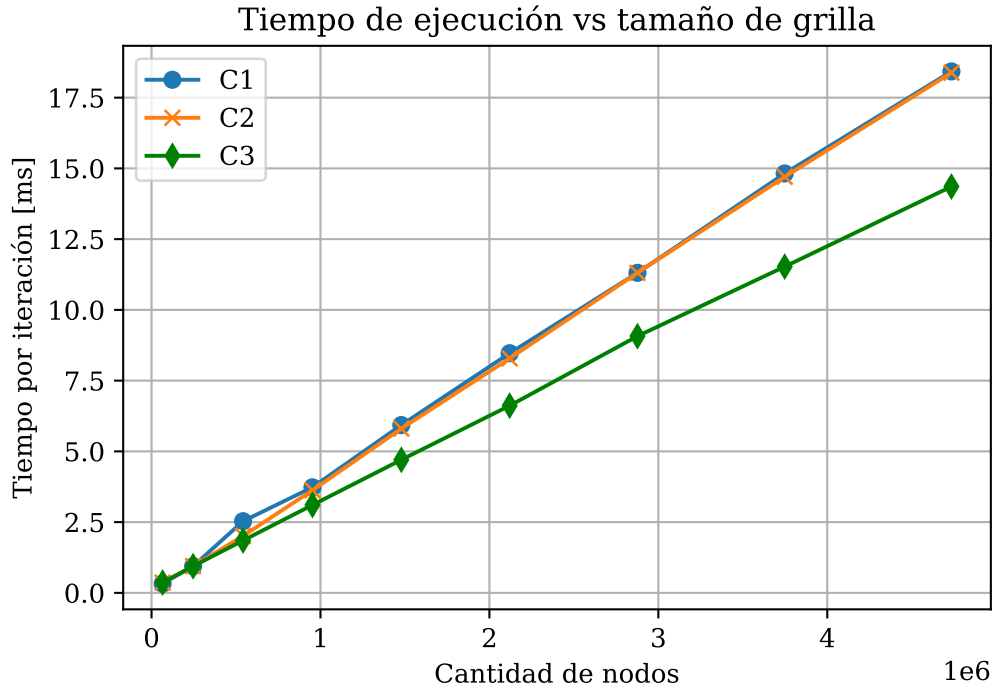


Figura 19: Comparación de tiempo de ejecución de las diferentes configuraciones para la matriz de coordinación utilizando grupos solapados

Tabla 6: Resultados de pruebas de versiones del modelo con grupos separados

Dimensiones \ Versión	C1 [ms]	C2[ms]	C3[ms]
256×256	0,4047	0,3770	0,3747
496×496	0,9711	0,9648	0,9466
736×736	2,7283	2,0840	1,9343
976×976	3,9830	3,7760	3,2866
1216×1216	6,1680	5,8790	5,0055
1456×1456	8,9000	8,4570	7,1953
1696×1696	12,2440	11,5650	9,8060
1936×1936	15,4950	14,9532	12,8067
2176×2176	19,7200	18,9288	16,3138

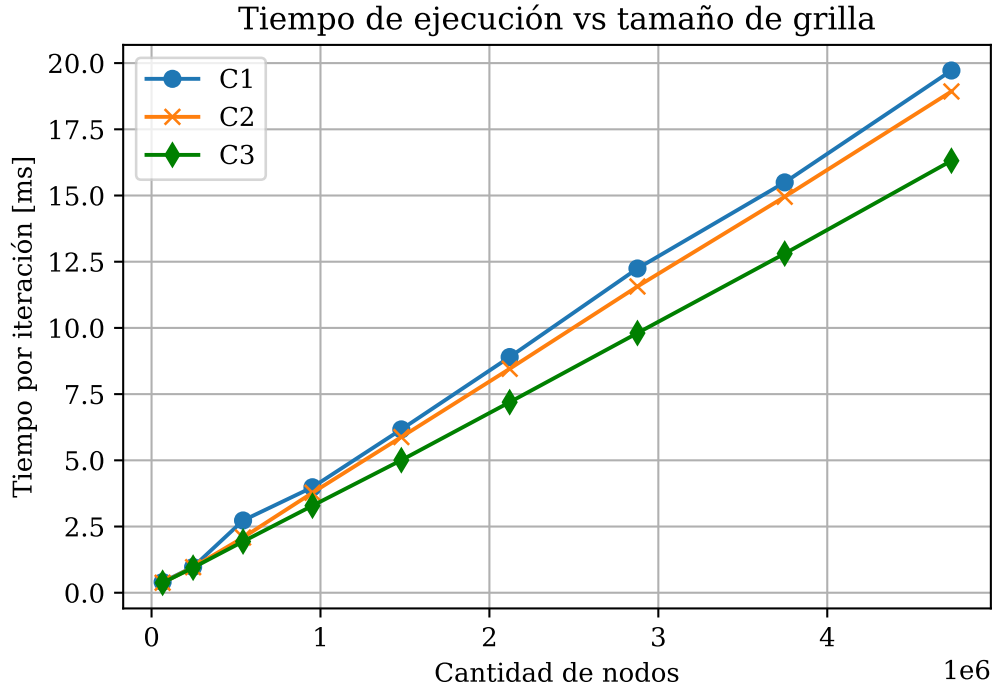


Figura 20: Comparación de tiempo de ejecución de las diferentes configuraciones para la matriz de coordinación utilizando grupos separados

Tal como se puede apreciar en los gráficos 14 y 20 en ambos modelos no existe gran influencia de la disminución del tiempo en el caso del uso del if a diferencia de la implementación que evita reiniciar la matriz de coordinación, que muestra una clara tendencia en la disminución de tiempo de ejecución a medida que el tamaño del dominio crece. Por lo cual para las siguientes pruebas se utilizará esta implementación en ambos modelos de agrupación.

4.3.2. Comparación de las diferentes propuestas de solución

En esta sección se realizaron diferentes pruebas a los diferentes algoritmos y modelos propuestos en el capítulo anterior, es decir los Algoritmos 3, 4, 5 y 6. Al igual que en la sección anterior el tiempo promedio de las iteraciones fue obtenido con la herramienta de perfilamiento de Nvidia nvprof y solo se consideró el tiempo de las iteraciones del algoritmo.

Las pruebas fueron realizadas utilizando el caso de prueba detallado anteriormente. El tamaño de los dominios probados son de mucho menor tamaño que los dominios de las tras secciones debido a la limitación que presenta la función `cudaLaunchCooperativeKernel()` necesaria para ejecutar dentro del kernel la sincronización para el grupo cooperativo grid, la cual es utilizada en los Algoritmos 3 y 4 (con ambos modelos). El tamaño de los grupos cooperativos utilizados de igual manera presentan un tamaño mas pequeño y de tipo bloque de dimensión 8×8 a excepción del grupo cooperativo grid el cual es del tamaño de la grilla utilizada

Los resultados obtenidos pueden observarse en la Tabla 7 y el gráfico comparativo 21

Tabla 7: Resultados de pruebas de comparación de algoritmos

Algoritmo Dimensiones	Grid[μs]	M1[μs]	M2[μs]	Sol[μs]	Sep[μs]
16×16	89.019	90.789	86.575	68.576	77.317
40×40	101.228	108.775	91.510	88.129	83.645
64×64	103.373	118.575	106.998	120.036	95.104
88×88	125.383	129.215	126.996	178.357	112.001
112×112	146.394	155.789	160.456	200.375	135.010
120×120	155.375	164.366	169.400	190.541	147.158
136×136	182.696	188.967	190.362	220.502	168.362

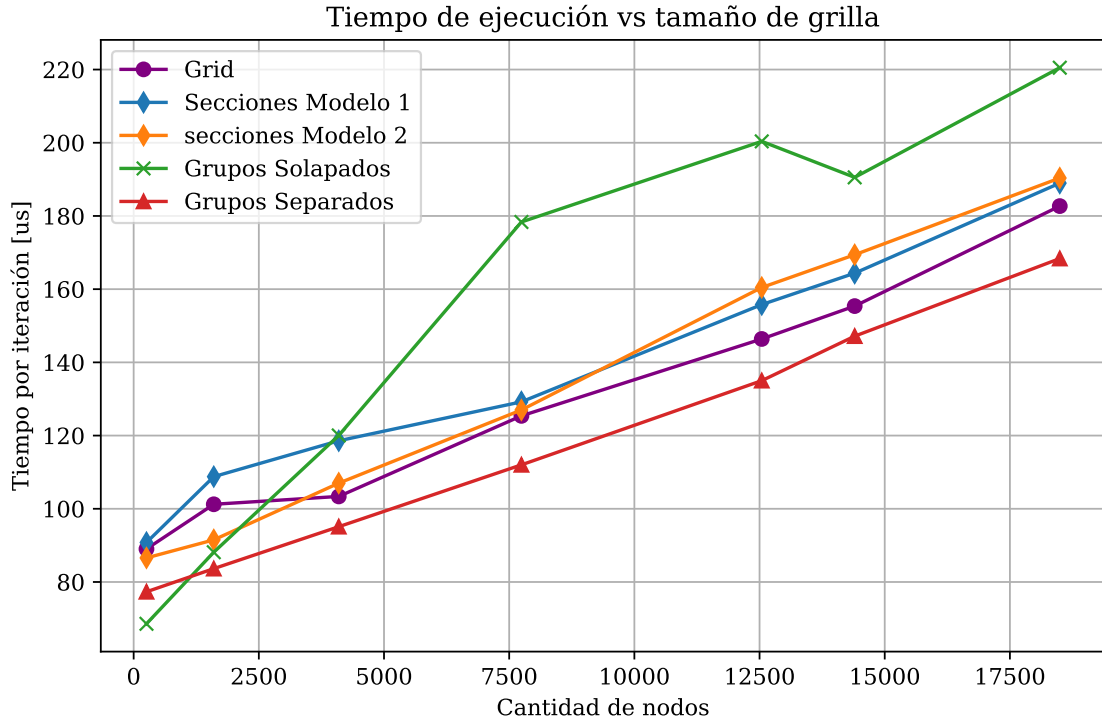


Figura 21: Comparación de tiempos de ejecución de diferentes las propuestas de solución

Al ser dominios mas pequeños los tiempos de ejecución son muy parecidos en todas las implementaciones exceptuando el de Grupos Solapados a diferencia del experimento anterior que presentaba mejor tiempo. De igual forma se puede ver que la solución con mejor desempeño es la del algoritmo de grupos Separados.

4.3.3. Comparación al utilizar distintos tamaños de grupos cooperativos

En esta sección se realizaron pruebas a los algoritmos 5 y 6 considerando las soluciones que presentaron mejor rendimiento en el primer experimento.

En este caso se busca comparar el efecto sobre el rendimiento del kernel al utilizar diferentes tamaños y tipos de grupo, los cuales pueden ser de tipo bloque y de tipo Tile. Para comparar equitativamente ambos programas se utilizaron diferentes tamaños de dominio para cada algoritmo los cuales se ajustaran a una división exacta de la grilla dependiendo del tamaño del grupo y que no existan Grupos Cooperativos con threads que queden fuera del dominio. Las características de los Grupos cooperativos utilizados es la siguiente:

- 16×16 tipo bloque
- 8×8 tipo bloque
- 4×4 tipo Tile con un bloque con 112 threads
- 4×4 tipo bloque

Al igual que los experimentos anteriores solo se midió el tiempo del kernel utilizando Nvidia nprof ya que el resto del programa es igual para ambos casos.

Los resultados obtenidos son mostrados en las Tablas 8 y 9 y gráficos 22 y 23 para los algoritmos de Grupos solapados y separados respectivamente.

Tabla 8: Resultados de pruebas utilizando diferentes tamaños de grupos para algoritmo con grupos Solapados

Grupos Dimensión	16x16	8x8	4x4 (tiles)	4x4 (bloques)
526×526	0.920647	1.042	1.512	2.587
736×736	1.717	2.001	2.98	5.06
946 ×946	2.814	3.268	4.922	8.41
1156 ×1156	4.167	4.822	7.298	12.597
1576×1576	7.703	8.952	13.668	23.583
1786 ×1786	9.902	11.383	17.657	30.431
1996 ×1996	12.369	14.323	22.122	38.092
2206×2206	15.114	17.625	27.003	46.577

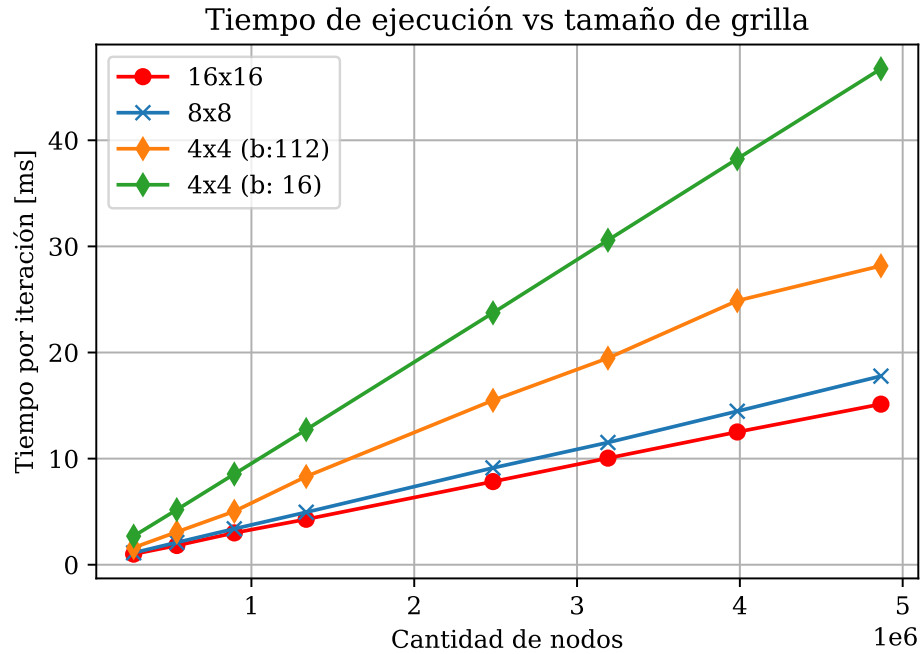


Figura 22: Comparación de rendimiento según tamaño de grupo para propuesta de grupos solapados

Tabla 9: Resultados de pruebas utilizando diferentes tamaños de grupos para algoritmo con grupos separado

Dimension \ Grupos	16x16	8x8	4x4 (tiles)	4x4 (bloques)
528 × 528	1.351	0.929558	1.25	1.734
736 × 736	2.5	1.721	2.268	3.261
944 × 944	-	2.824	5.431	5.391
1152 × 1152	-	4.198	8.77	8.282
1584 × 1584	-	8.002	15.002	24.002
1792 × 1792	-	10.367	-	70.003
2000 × 2000	-	13.556	-	-
2208 × 2208	-	16.523	-	-

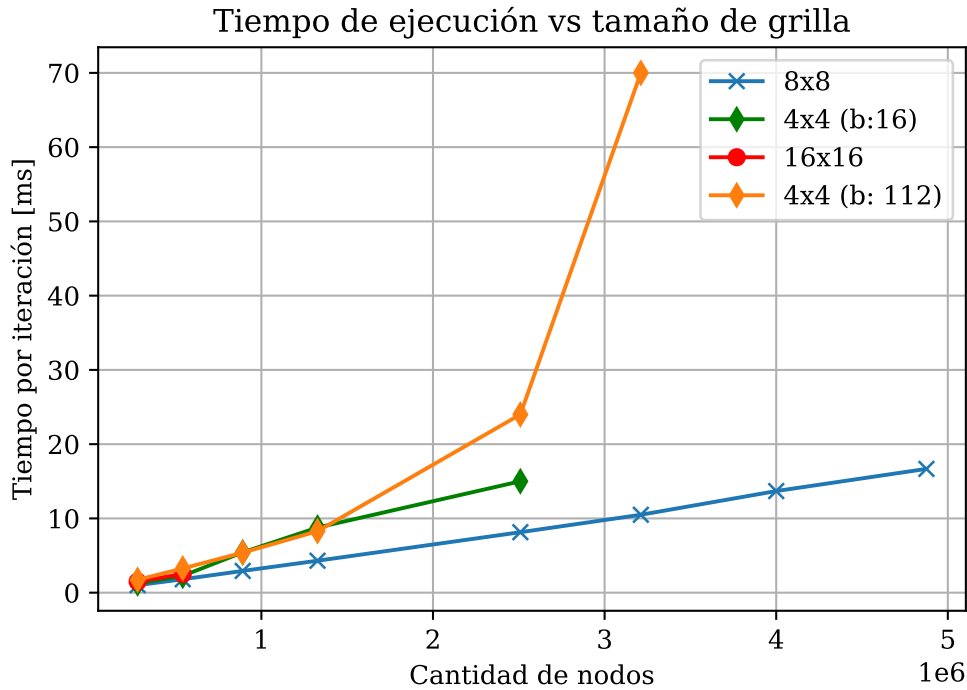


Figura 23: Comparación de rendimiento según tamaño de grupo para propuesta de grupos separados

En el caso de los grupos solapados, todas la pruebas pudieron completarse satisfactoriamente, sin embargo en el caso del programa que implementa el algoritmo de grupos separados se quedaba congelado a partir de cierta cantidad de nodos en la grilla dependiendo de la dimensión del grupo cooperativo utilizado. Este congelamiento comprueba la fuerte dependencia que existen entre grupos vecinos con este modelo, ya que significa que los grupos que se ejecutaban quedaron bloqueados esperando que los grupos vecinos también terminaran de ejecutarse, esto debido al uso de operaciones atómicas que bloqueaban recursos de la GPU ocasionando que threads no terminaran con la ejecución al no poder coordinarse con los otros threads de los nodos vecinos e impidiendo que nuevos bloques pudieran ejecutarse al no haber recursos disponibles. Como se pudo observar este bloqueo puede mitigarse disminuyendo el tamaño del bloque ejecutado, ya que entre mas pequeña la dimensión del bloque, mas bloques pueden ejecutarse al mismo tiempo, sin embargo en dominios muy grandes siempre existe la probabilidad de bloqueo.

4.4. Comparación de propuesta seleccionada con trabajos anteriores

En esta sección se realizan las comparaciones de rendimiento entre la solución escogida y las soluciones ya desarrolladas anteriormente. Se escogió como la mejor solución propuesta al algoritmo 5, debido a su rendimiento y constancia de funcionamiento, si bien utiliza más recursos, los grupos solapados no tuvieron ningún problema en la ejecución independiente de las dimensiones de los grupos o dominios a comparación con los otros algoritmos.

Las soluciones con las que se compara el algoritmo escogido fueron una implementación del enfoque push con dos kernel y el kernel con enfoque pull con las optimizaciones I4-B1 (uso de valores booleanos y uso de declaraciones if) las cuales fueron de las implementaciones con mejores resultados presentados en [21].

Al igual que los experimentos anteriores se evaluó el tiempo promedio del time step utilizando la herramienta Nvidia nprof, esta vez las pruebas se realizaron en tres máquinas distintas cuyos datos fueron especificados anteriormente. Las pruebas realizadas no se consideró el cálculo del efecto Coriolis ya que el programa desarrollado por Salinas no lo considera.

Los resultados obtenidos en las pruebas realizadas son los siguientes

Tabla 10: Resultados de pruebas comparativas realizadas en tarjeta Nvidia RTX 2060

Propuesta Dimensión	Salinas [ms]	2 Kernels [ms]	Memoria [ms]
256 × 256	0.304123	0.428132	0.3644
496 × 496	0.826156	0.988286	0.9426
736 × 736	1.958	2.04	1.8512
976 × 976	3.186	33.224	3.1050
1216 × 1216	5.058	5.091	4.7020
1456 × 1456	7.024	7.19	6.7157
1696 × 1696	9.542	9.799	9.0710
1936 × 1936	12.064	12.763	11.2290
2176 × 2176	15.496	16.044	14.4580

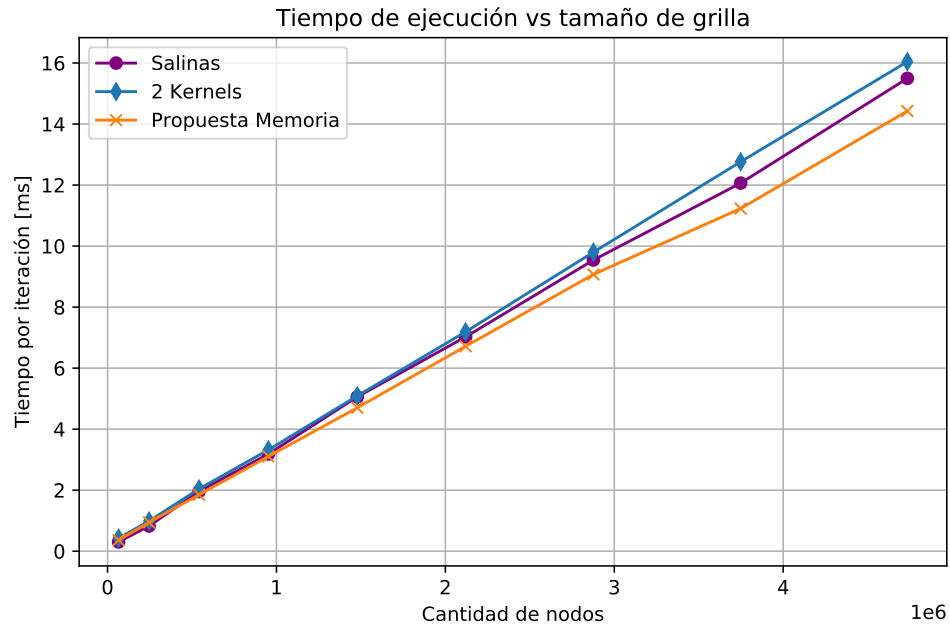


Figura 24: Comparación de performance entre las soluciones desarrolladas en GPU Nvidia RTX 2060

Tabla 11: Resultados de pruebas comparativas realizadas en tarjeta Nvidia Tesla K80

Propuesta Dimension	Salinas [ms]	2 Kernels [ms]	Memoria [ms]
256 × 256	0.2884	0.315	0.244314
496 × 496	0.983248	0.986	0.894376
736 × 736	2.004268	1.934	1.91898
976 × 976	3.40844	3.301	3.34726
1216 × 1216	5.28528	5.354	5.58432
1456 × 1456	7.48524	7.523	7.78204
1696 × 1696	9.98452	10.956	11.05314
1936 × 1936	13.10176	14.046	14.546
2176 × 2176	16.443	17.534	18.4268

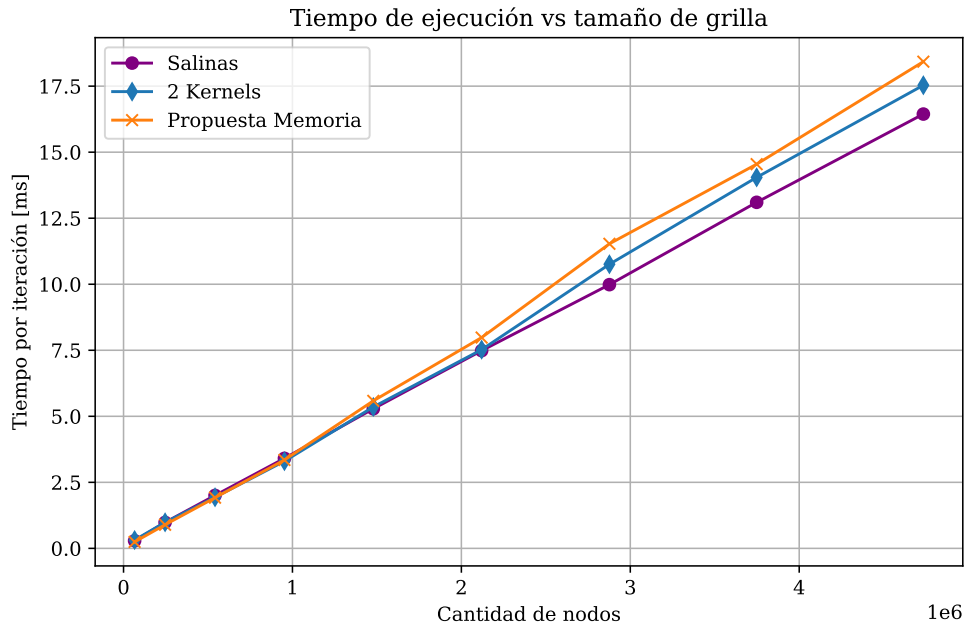


Figura 25: Comparación de performance entre las soluciones desarrolladas en GPU Tesla K80

Tabla 12: Resultados de pruebas comparativas realizadas en tarjeta Nvidia Tesla T4

Propuesta Dimension	Salinas [ms]	2 Kernels [ms]	Memoria [ms]
256 × 256	0.32270	0.4970	0.32522
496 × 496	0.82713	0.9850	0.78174
736 × 736	1.6329	1.6876	1.5240
976 × 976	2.8008	2.8809	2.5873
1216 × 1216	4.4230	4.4671	4.7551
1456 × 1456	6.1545	6.3966	5.5768
1696 × 1696	8.3958	8.887	7.4994
1936 × 1936	10.816	11.2417	9.7352
2176 × 2176	13.702	14.4027	12.190

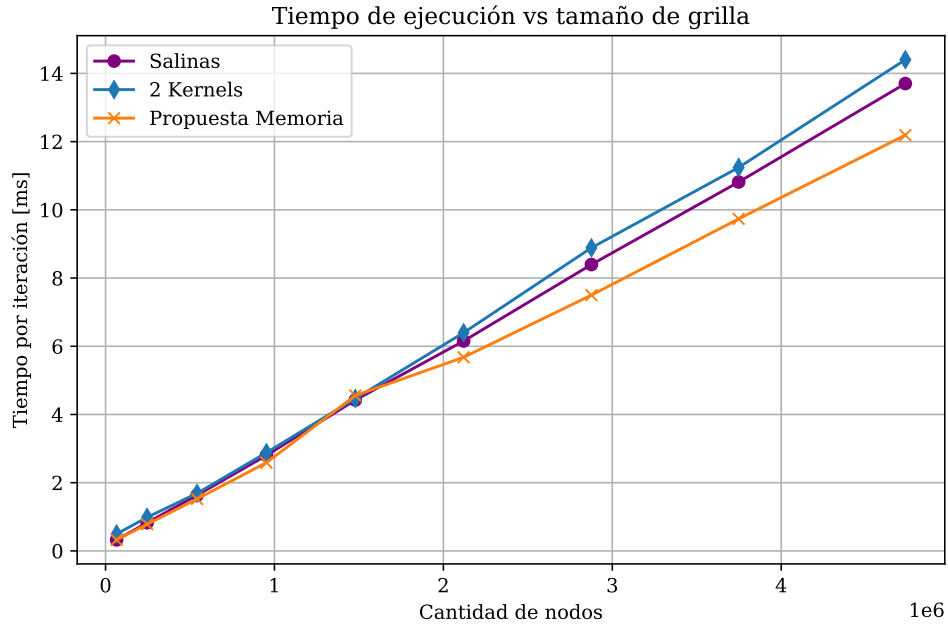


Figura 26: Comparación de performance entre las soluciones desarrolladas en GPU Tesla T4

En las arquitecturas mas nuevas como la Turing presente en las tarjeta RTX 2060 y Tesla T4 es posible ver mejor tiempo promedio de ejecución en el kernel que implementa el algoritmo propuesto en esta memoria a diferencia de la tarjeta Tesla K80 que tiene arquitectura Maxwell que es mas antigua que la libreria Cooperative Groups

Si bien el tiempo de ejecución del kernel es mejor, tambien es importante considerar el tiempo de inicializacion de la matriz de coordinacion por lo que tambien se midió el tiempo que demora en leer los datos del archivo de prueba considerando la inicializacion de la matriz y sin esta para comparar los tiempos, estos tiempos pueden veerse en la Tabla 13

Tabla 13: Tiempos de inicializacion de datos

Dimension \ Lectura	Con matriz [ms]	Sin matriz[ms]
256×256	21	15
496×496	50	40
736×736	120	108
976×976	212	188
1216×1216	360	308
1456×1456	486	475
1696×1696	619	593
1936×1936	779	749
2176×2176	1022	982

Como se puede ver en dominios de mayor tamaño la diferencia de tiempo es menor a que 100[ms] por lo que dependiendo de la cantidad de iteraciones el programa que utiliza el algoritmo con grupos solapados puede presentar mejor o peor tiempo total de ejecución.

4.5. Efecto Coriolis

En esta sección se muestra la comparación del resultado considerando el efecto Coriolis. Para esta prueba se utilizó un dominio cuadrado de dimensión 976×976 ubicado entre las latitudes $-36,03^\circ$ y $-36,55^\circ$, además se consideró un dominio cerrado para poder observar continuamente el movimiento del agua por un periodo mas largo de tiempo. El caso de prueba utilizado es similar en forma al caso utilizado en la pruebas de las secciones anteriores, diferenciándose en la altura de la figura. La ecuación utilizada es la siguiente:

$$Z = 20 \cdot e^{\frac{-(x^2+y^2)}{2^2}} + 200 \quad (14)$$

Al ser una figura que representa mayor tamaño, también se cambiaron los parámetros para el algoritmo los cuales se presentan en la Tabla 14

Tabla 14: Parametros de ejecución

$\Delta x = \Delta y$	60[m]
Δt	0.1 [s]
τ	0.8
N° iteraciones	40000

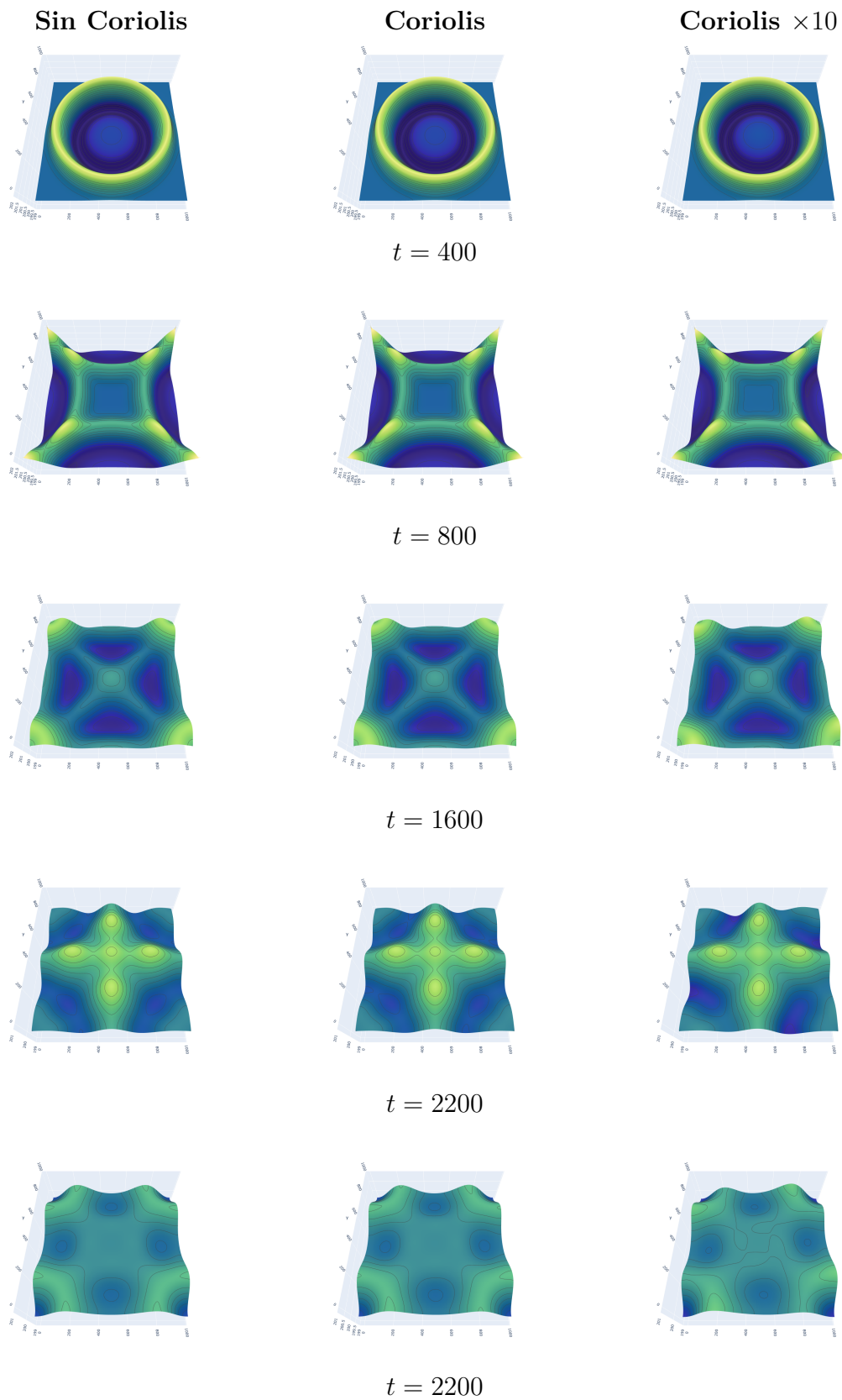


Figura 27: Comparación de efecto Coriolis en la Simulación

Los resultados obtenidos se pueden ver en la Figura 27 en la cual se muestra capturas de un tiempo específico. En la primera columna se muestra el caso donde $\omega = 0$, la segunda es un caso aproximado del la rotación real de la tierra donde se consideró $\omega = 0,0000727$ y la tercera columna considera $\omega = 0,000727$ para mostrar gráficamente de manera más clara el efecto Coriolis en el resultado de la simulación.

En la figura, se puede observar que los primeros instantes de tiempos los resultados de las tres versiones son muy similares, a partir de $t = 2200[s]$ ya se puede apreciar la diferencia entre las figuras de la misma fila, con una diferencia mas notoria en el resultado de la tercera columna.

CAPÍTULO 5

CONCLUSIONES

En el trabajo fue posible llevar a cabo el cumplimiento a los objetivos planteados en un principio, los cuales implicaban crear una implementación alternativa para el método de Lattice Boltzmann en CUDA la cual sea eficiente, que considere la librería Cooperative Groups y que pueda competir con soluciones anteriores, además de considerar el término del efecto Coriolis el cual es necesario para una mejor simulación en el contexto de tsunamis.

Se trabajaron diferentes alternativas para la sincronización del algoritmo, una problemática que afectaba al enfoque push. Por lo tanto, se probaron diferentes formas de sincronización aprovechando las funcionalidades que ofrece la librería de Cooperative Groups creando diferentes algoritmos de sincronización. Se aprovechó el uso de grupos para mejorar el paralelismo del algoritmo dividiendo la grilla en diferentes secciones, lo que terminó resultando en la creación de una nueva estructura de datos, denominada en esta memoria como Matriz de Coordinación.

Durante los experimentos con los diferentes algoritmos propuestos se pudo ver si bien la sincronización por grid es mucho más sencilla, la división de la grilla en secciones sí mejoró el tiempo de ejecución para el algoritmo de grupos separados, incluso en dominios pequeños. Sin embargo, para dominios más grandes el algoritmo de grupos solapados presenta notoriamente un tiempo de ejecución más bajo y un funcionamiento correcto consistente aun con dominios grandes y diferentes tamaños de grupos, por lo cual este algoritmo fue considerado como la mejor propuesta de solución en esta memoria, aunque presente mayor uso de recursos en comparación con las otras alternativas.

Se pudo observar que en arquitecturas más nuevas la solución escogida presentó mejores tiempos de ejecución que la implementación del método de Lattice Boltzmann con dos kernel por lo cual el método de sincronización propuesto en esta memoria muestra ventaja sobre la separación de funcionalidades en varios kernel para evitar condiciones de carrera, además de presentar mejor tiempo de ejecución en comparación con la implementación utilizando también un solo kernel con enfoque pull. Sin embargo, aun cuando el kernel presenta mejor desempeño hay que considerar el tiempo de inicialización de la matriz de coordinación, el cual puede ser amortiguado dependiendo de la cantidad de iteraciones que se realicen al método.

En conclusión, se pudo obtener un desarrollo de un kernel que puede competir en cuanto a eficiencia en dominios grandes con el trabajo de Salinas el cual ya presentaba un trabajo de optimización previa, solucionando en el camino un problema de coordinación global de threads, sin la necesidad de separar el kernel. Sin embargo, es importante considerar que exige más recursos para ejecutar y arquitecturas de GPU más nuevas. Por otro lado, se pudo visualizar el impacto en la simulación al considerar el efecto Coriolis y

como cambia sutilmente los resultados en cada iteración hasta hacerse notorios después de múltiples de estas, lo cual conlleva a un resultado más representativo de la realidad.

5.1. Trabajo Futuro

Si bien en este trabajo se trató de tomar múltiples consideraciones en el desarrollo del kernel, aún existen múltiples aspectos a mejorar en la programación o planteamientos nuevos que podrían ser útiles en un nuevo desarrollo del método de Lattice Boltzmann que busque ser eficiente, tales como:

- Considerar otros enfoques de Lattice Boltzmann, los enfoques push y pull no son los únicos enfoques para este método que existen, si bien son los más intuitivos se han descrito otras implementaciones como el algoritmo swap, shift o Lagrangiano, los cuales incluso necesitan menos memoria para ejecutar.
- Optimizar el código matemáticamente, la programación de los cálculos matemáticos de los valores de la función de equilibrio o termino fuente podrían simplificarse lo que beneficiaría en el tiempo de ejecución y evitar la pérdida de significancia.
- Optimizar la matriz de coordinación, en esta memoria se utilizó un arreglo que considera todos los nodos, incluso aquellos que no necesitan coordinarse, lo que conlleva en un desperdicio de memoria, por lo que se puede trabajar en otra forma de implementarla utilizando estructuras como una matriz dispersa.
- Considerar otras fuerzas para mejorar el resultado de la simulación, además de la fuerza del efecto Coriolis existen muchos otros factores que afectan en el movimiento del mar, como lo es la fuerza del viento o el roce del fondo marino.

ANEXOS

- Repositorio utilizado: <https://github.com/MonseFL/LBM-Cooperative-Groups>

REFERENCIAS BIBLIOGRÁFICAS

- [1] *Cooperative groups*, Septiembre 2017. <https://developer.nvidia.com/blog/cooperative-groups/>.
- [2] *Cuda C++ Programming Guide*, Dec 2020. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cooperative-groups>.
- [3] *Cuda C++ Programming Guide*, Dec 2020. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-fence-functions>.
- [4] Aslam, Maria, Omer Riaz, Shahzad Mumtaz y Ali Daniyal Asif: *Performance comparison of gpu-based jacobi solvers using cuda provided synchronization methods*. IEEE Access, 8:31792–31812, 2020.
- [5] Bardsley, Ethel y Alastair F. Donaldson: *Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels*. En Badger, Julia M. y Kristin Yvonne Rozier (editores): *NASA Formal Methods*, páginas 230–245, Cham, 2014. Springer International Publishing, ISBN 978-3-319-06200-6.
- [6] Bernaschi, Massimo, Massimiliano Fatica, Simone Melchionna, Sauro Succi y Efthimios Kaxiras: *A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries*. Concurrency and computation: practice and experience, 22(1):1–14, 2010.
- [7] Bhatnagar, P. L., E. P. Gross y M. Krook: *A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems*. Phys. Rev., 94:511–525, May 1954. <https://link.aps.org/doi/10.1103/PhysRev.94.511>.
- [8] Chen, Shiyi y Gary D. Doolen: *LATTICE BOLTZMANN METHOD FOR FLUID FLOWS*. Annual Review of Fluid Mechanics, 30(1):329–364, 1998. <https://doi.org/10.1146/annurev.fluid.30.1.329>.
- [9] Dellar, Paul J: *Lattice kinetic schemes for magnetohydrodynamics*. Journal of Computational Physics, 179(1):95–126, 2002.
- [10] Ebersole, Mark: *What Is CUDA?*, Nov 2012. <https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/>.
- [11] Feng, Wu chun y Shucaï Xiao: *To GPU synchronize or not GPU synchronize?* En *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, páginas 3801–3804, 2010.
- [12] Habich, Johannes, Christian Feichtinger, Harald Köstler, Georg Hager y Gerhard Wellein: *Performance engineering for the lattice Boltzmann method on GPGPUs*.

- Architectural requirements and performance results.* Computers & Fluids, 80:276–282, 2013.
- [13] Harris, Mark y Kyrylo Pereygin: *Cooperative Groups: Flexible CUDA Thread Programming*, Oct 2017. <https://developer.nvidia.com/blog/cooperative-groups/>.
- [14] Herschlag, Gregory, Seyong Lee, Jeffrey S Vetter y Amanda Randles: *GPU data access on complex geometries for D3Q19 lattice Boltzmann method.* En *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, páginas 825–834. IEEE, 2018.
- [15] Higuera, F J y Javier Jiménez: *Boltzmann approach to lattice gas simulations.* EPL (Europhysics Letters), 9(7):663, 1989.
- [16] Hu, Bodun y Christopher J Rossbach: *Altis: Modernizing GPGPU Benchmarks.* En *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, páginas 1–11. IEEE Computer Society, 2020.
- [17] Li, Guodong y Ganesh Gopalakrishnan: *Scalable SMT-based verification of GPU kernel functions.* En *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, páginas 187–196, 2010.
- [18] Li, Wei, Zhe Fan, Xiaoming Wei y Arie Kaufman: *GPU-based flow simulation with complex boundaries.* GPU gems, 2:747–764, 2003.
- [19] Mawson, Mark J y Alistair J Revell: *Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs.* Computer Physics Communications, 185(10):2566–2574, 2014.
- [20] McNamara, GR y Gianluigi Zanetti: *Use of the Boltzmann Equation to Simulate Lattice-Gas Automata.* Physical review letters, 61:2332–2335, Diciembre 1988.
- [21] Salinas, Álvaro, Claudio Torres y Orlando Ayala: *A fast and efficient integration of boundary conditions into a unified CUDA Kernel for a shallow water solver lattice Boltzmann Method.* Computer Physics Communications, 249:107009, 2020.
- [22] Salinas, Álvaro, Claudio E Torres y Orlando Ayala: *Well-balanced open boundary condition in a lattice Boltzmann model for shallow water with arbitrary bathymetry.* Computer Physics Communications, 230:89–98, 2018.
- [23] Salmon, Rick: *The lattice Boltzmann method as a basis for ocean circulation modeling.* Journal of Marine Research, 57(3):503–535, 1999.
- [24] Tölke, Jonas: *Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA.* Computing and Visualization in Science, 13(1):29, 2010.

- [25] Vreugdenhil, C. B.: *Equations*, páginas 15–46. Springer Netherlands, Dordrecht, 1994, ISBN 978-94-015-8354-1. https://doi.org/10.1007/978-94-015-8354-1_2.
- [26] Wittmann, Markus, Thomas Zeiser, Georg Hager y Gerhard Wellein: *Comparison of different propagation steps for lattice Boltzmann methods*. Computers & Mathematics with Applications, 65(6):924–935, 2013.
- [27] Wolf-Gladrow, Dieter A: *Lattice-gas cellular automata and lattice Boltzmann models: an introduction*. Springer, 2004.
- [28] Xiao, Shucai y Wu chun Feng: *Inter-block GPU communication via fast barrier synchronization*. En *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, páginas 1–12, 2010.
- [29] Yoshino, Masato, Y. Matsuda y C. Shao: *Comparison of Accuracy and Efficiency between the Lattice Boltzmann Method and the Finite Difference Method in Viscous/Thermal Fluid Flows*. International Journal of Computational Fluid Dynamics - INT J COMPUT FLUID DYNAMICS, 18:333–345, Mayo 2004.
- [30] Zhang, Junfeng: *Lattice Boltzmann method for microfluidics: models and applications*. Microfluidics and Nanofluidics, 10(1):1–28, 2011.
- [31] Zhou, Jian Guo: *Lattice Boltzmann methods for shallow water flows*, volumen 4. Springer, 2004.