TESIS CARRERA DE DOCTORADO EN CIENCIAS DE LA INGENIERÍA

MODELOS DE AUTÓMATAS CELULARES SOBRE UNIDADES DE PROCESAMIENTO GRÁFICO DE ALTA PERFORMANCE

Mg. Ing. Pablo R. Rinaldi
Doctorando

Dr. Enzo Alberto Dari
Director

Dr. Marcelo Javier Vénere
Codirector

Marzo de 2011

Grupo MECOM, Centro Atómico Bariloche, Argentina
Instituto PLADEMA, UNCPBA Tandil, Argentina

Instituto Balseiro
Universidad Nacional de Cuyo
Comisión Nacional de Energía Atómica
Argentina

a mi bebé

Índice de abreviaturas

AC	Autómatas celulares
ALU	Unidad aritmético lógica
AO	Ordenamiento de aplicación
BLAS	Subrutinas básicas de álgebra lineal
BGK	Regla de colisión de Bhatnagar, Gross y Krook
CFD	Fluidodinámica computacional
CPU	Unidad central de procesamiento
CUDA	*Compute Unified Device Architecture*
DA	Arreglos dinámicos
DMA	Acceso directo a memoria
FE	Elementos finitos
FFT	Transformada rápida de Fourier
GPGPU	Computación de propósito general en unidades de procesamiento gráfico
GPU	Unidad de procesamiento gráfico
HPC	Computación de alta performance
LBM	Método de lattice Boltzmann
LGA	Autómata de gas reticular
LUPS	Celdas de grilla actualizadas por segundo
MDT	Modelo digital del terreno
MLUPS	Millones de celdas de grilla actualizadas por segundo
MPI	Interfaz de pasaje de mensajes
PETSc	*Portable extensible toolkit for scientific computation*
PC	Computadora personal
RAM	Memoria de acceso aleatorio
Re	Número de Reynolds
SDK	Conjunto de herramientas de desarrollo de software
SIMD	Instrucción simple para múltiples datos
SIMT	Instrucción simple para múltiples hilos de ejecución
SLI	*Scalable link interface*
VPB	Condiciones de contorno de presión y velocidad
Wo	Número de Womersley

Índice de contenidos

Resumen

Se desarrolló un enfoque novedoso de Fluidodinámica Computacional utilizando Unidades de Procesamiento Gráfico (GPU) como estrategia alternativa a los esquemas clásicos. Al tratarse de un nuevo paradigma paralelo de software-hardware, muchas soluciones clásicas no son aplicables y los algoritmos deben replantearse completamente.

Se propusieron estrategias de implementación de simuladores basados en autómatas celulares (AC) utilizando la tecnología *Compute Unified Device Architecture* (CUDA). Los recursos de la GPU se estudiaron profundamente analizando los diferentes tipos de memorias, evaluando esquemas de almacenamiento de datos y patrones de acceso. Entre otras optimizaciones se buscó la máxima performance con distintos esquemas de división del dominio y configuraciones de ejecución.

Los tiempos de cálculo son comparables a los de equipos mucho más costosos, como clusters de servidores, lográndose aceleraciones de hasta dos órdenes de magnitud respecto a códigos equivalentes para CPU. La validación se realizó con escenarios en dos y tres dimensiones mostrando muy buena concordancia con otras simulaciones y mediciones experimentales. También se realizó la búsqueda de parámetros típicos en flujos oscilatorios 3D mostrando que los AC sobre GPU poseen un gran potencial en ingeniería como simuladores de alta performance a muy bajo costo.

Palabras Clave: Autómatas Celulares, GPGPU, HPC, Métodos de Lattice Boltzmann, CUDA.

Cellular Automata models on high performance graphic processing units

Abstract

A new approach for Computational Fluid Dynamics was developed using Graphic Processing Units (GPU) as an alternative to classical models. Being a new software-hardware parallel paradigm, most typical solutions are no longer useful, and algorithms must be completely rethought.

Implementation strategies for cellular-automata based simulators were proposed using *Compute Unified Device Architecture* (CUDA) technologies. GPU resources were deeply investigated analyzing memory access patterns and data layouts. Maximum performance was sought with several domain division strategies and execution configurations between other optimizations.

Calculation times are similar to those reached in high performance equipments or server clusters, with up two orders of magnitude speedup over similar desktop CPU implementations. The model was validated on two and three dimensional scenarios showing good agreement with other simulations and experimental data. An exhaustive parameter sensitivity analysis of a 3D oscillatory case is presented, showing the potential of this technology for low-cost high-performance engineering simulations.

Keywords: Cellular Automata, GPGPU, HPC, Lattice Boltzmann Methods, CUDA.

Capítulo 1: Introducción

1.1. Autómatas celulares

En los cincuenta, Ulam y Von Neumann [01] concibieron la idea de una ingeniosa herramienta matemática denominada Autómata Celular (AC). Mostraron que ciertos fenómenos complejos se pueden simular con celdas discretas dispuestas en arreglos regulares, que interactúan de acuerdo a reglas simples. Los simuladores desarrollados a partir de este concepto consisten en un conjunto de células o celdas independientes con un estado asociado. Las células evolucionan cambiando su estado a cada paso del tiempo. El próximo estado de cada celda se determina en base a su estado actual y el de las celdas en una determinada vecindad local. Las reglas de interacción que generalmente se aplican a los vecinos inmediatos, pueden o no guardar una semejanza con las leyes físicas que gobiernen el fenómeno.

El clásico ejemplo de AC es el propuesto por John Conway denominado "Juego de la Vida" o (*"Game of Life"*) [02]. Este autómata fue sugerido por Stanislaw Ulam en la década de 1940 y formalizado por primera vez por el matemático John von Neumann, quien trató de encontrar una máquina hipotética que construyera copias de sí misma con el objetivo de emular procesos de reproducción biológica. Conway simplificó estas ideas 3 décadas después dando lugar a un nuevo campo de investigación: los autómatas celulares. El modelo formulado es interesante porque pueden emerger patrones complejos de reglas simples. De hecho, muchas aplicaciones han demostrado que un AC es una muy buena aproximación para recrear fenómenos físicos complejos.

Otro ejemplo de AC unidimensionales simple es el presentado por Wolfram [03] cuya evolución temporal genera gráficos similares a patrones de pigmentación hallados en caracoles marinos. Esto representa, según Wolfram, ejemplos de la complejidad de algunos fenómenos naturales que en realidad son generados por procesos con reglas básicas extremadamente simples. En la figura 1.1 se muestra un patrón altamente complejo encontrado en caparazones de moluscos y una imagen muy similar generada a partir de la evolución de un AC unidimensional simple.

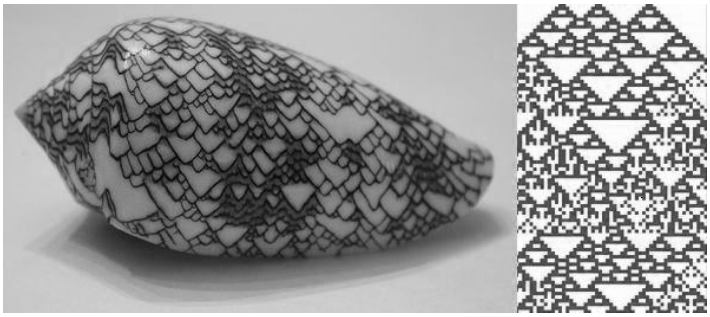


Figura 1.1: Caracol marino con pigmentación similar a la generada por un autómata celular.

En la última década, los AC han sido utilizados con éxito para el movimiento de flujos superficiales como el simulador de escurrimiento de lava volcánica SCIARA [04] o el modelo de escurrimiento por inventario hidrológico AQUA [05] [06] que simula eventos reales de tormenta. Estos modelos son aplicables a dominios de gran tamaño debido a la baja complejidad computacional de sus cálculos.

Particularmente, para algunos AC se encontró que los promedios estadísticos tienden a la solución de ecuaciones diferenciales parciales de campos fluidos, típicamente la ecuación de Navier-Stokes [07]. Este es el caso del Método de Lattice Boltzmann (LBM) que reproduce el movimiento microscópico de partículas de fluido mediante reglas locales lineales en una grilla de celdas, de modo que las propiedades macroscópicas responden a las ecuaciones de Navier-Stokes [08].

1.2. El Método de Lattice Boltzmann

El origen de LBM es el autómata de Lattice Gas (LGA) que es el primer AC propuesto para simular fluidos. El campo del LGA comenzó con la publicación de Frisch, Hasslacher y Pomeau [09], quienes mostraron que un tipo de "juego de billar" simplificado, representado por la propagación y colisión de partículas del fluido, llevaba a las ecuaciones de Navier-Stokes en un adecuado límite macroscópico. En general, un LGA consiste en una grilla regular con partículas residiendo en los nodos que la componen. Estas partículas se mueven una unidad de celda en la dirección de sus velocidades. Dos o más partículas arribando en el mismo sitio pueden colisionar.

Un ejemplo de LGA es el conocido autómata de Hardy-Pomeau-de Pazzis (HPP) [10] [11] de sólo 4 estados (Fig. 1.2). El HPP fue el primer modelo completamente discreto para la simulación de un fluido, siendo demostrado que simulaba las ecuaciones

de flujo asegurando las leyes de conservación. Sin embargo, las ecuaciones del flujo resultante eran anisotrópicas.

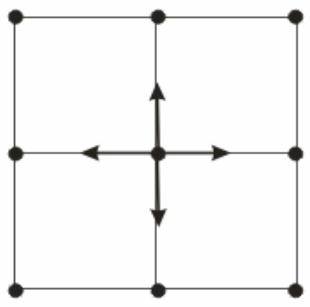


Figura 1.2: LGA: Celda rectangular del autómata HPP.

Diez años después, Frish [09] encontró que la simetría de la grilla juega un rol importante en la recuperación de las ecuaciones de Navier-Stokes. Basándose en una grilla hexagonal de 6 velocidades (modelo FHP) se obtuvo el primer LGA que recreaba correctamente las ecuaciones de Navier-Stokes (Fig. 1.3).

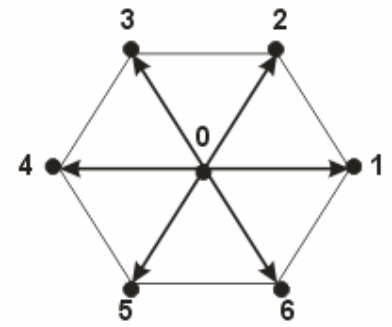


Figura 1.3: LGA: Celda hexagonal del autómata FHP.

A diferencia de los métodos clásicos que resuelven las ecuaciones macroscópicas discretizadas de Navier-Stokes, el concepto fundamental del Método de Lattice Boltzmann es construir modelos cinéticos simplificados que incorporen la física esencial de procesos microscópicos o mesoscópicos, de manera que las variables macroscópicas promediadas obedezcan las ecuaciones de conservación deseadas [12]. Al desarrollarse una versión simplificada de la ecuación cinética se evita resolver ecuaciones cinéticas complicadas tales como la ecuación completa de Boltzmann y hacer el seguimiento de cada partícula como se realiza en las simulaciones de dinámicas moleculares.

En el área computacional existen diferentes formas del modelo de Lattice Boltzmann. La elección sobre el tipo de modelo que debe usarse dependerá de la precisión y estabilidad esperada, la complejidad de su implementación y el costo computacional entre otros aspectos. Los autómatas de LBM han sido utilizados con éxito para la simulación de diferentes fenómenos complejos como fluidos en canales [13] [14], flujo en medios porosos [15] y en diversos tipos de flujos sanguíneos como sistólicos [16], flujo en válvulas cardíacas mecánicas [20] y en aneurismas seculares [21] entre otras.

1.3. Entornos paralelos

La naturaleza discreta de los simuladores basados en AC permite que sean ejecutados eficientemente en computadoras por lo que una gran variedad de modelos han sido implementados con esta técnica. En la mayoría de los casos, al tratarse de modelos explícitos sobre grillas regulares como lo es LBM, el código es fácilmente paralelizable debido a la independencia de los cálculos en cada celda. Sin embargo, los pasos de tiempo pequeños y el gran volumen de información intercambiado constituyen el cuello de botella principal a la hora de realizar implementaciones en paralelo utilizando métodos de división del dominio en combinación con pasaje de mensajes [17].

En el caso de grandes clusters o supercomputadoras, las arquitecturas están generalmente optimizadas para la comunicación entre servidores. El trabajo de Carter, Oliver y Shalf [18] analiza la performance de 3 códigos científicos en arquitecturas paralelas vectoriales y plataformas superescalares, mostrando que las aplicaciones que tienen un alto grado de paralelismo de datos (como los basados en el Método de Lattice Boltzmann) pueden beneficiarse en gran manera de esta clase de arquitecturas multiprocesador. Sin embargo, también señalan que los diseñadores de microprocesadores deberían invertir en eliminar los cuellos de botella en las aplicaciones científicas, mejorando el ancho de banda y la administración de memoria, ya que en el mejor de los casos no se llega al 35 por ciento de la performance de los procesadores.

En el caso de equipos más pequeños como PCs o máquinas de escritorio el panorama es mucho menos alentador ya que desde hace varios años la performance real

de los microprocesadores no sigue el crecimiento histórico dado por la ley de Moore. Al no existir otra manera de aumentar la performance de los procesadores, los principales fabricantes (*e.g.* INTEL y AMD) han apostado a colocar más núcleos por chip. Pero la forma en que se deben utilizar esos procesadores en paralelo no es un problema simple, como tampoco lo es la técnica de programación óptima para su aprovechamiento. Un estudio del año 2008 realizado en Sandia National Laboratories simuló las futuras computadoras de alta-performance basadas en 8 núcleos, 16 núcleos y 32 núcleos, que es lo que los fabricantes pronostican como el futuro de la industria. Los resultados son descorazonantes. Debido a lo limitado del ancho de banda de memoria y la poca adaptabilidad de los esquemas de manejo de memoria a esquemas multi-núcleo, la evolución de estas máquinas tiende a estancarse o incluso declinar a medida que la cantidad de procesadores aumenta. La performance de estos equipos es especialmente mala para aplicaciones informáticas del tipo *data-intensive*. [19].

Una arquitectura computacional más apropiada para este tipo de modelos es la de memoria compartida, que típicamente consiste en un bloque de memoria de acceso aleatorio RAM compartida por diferentes procesos, o *threads,* corriendo en paralelo. La comunicación entre los procesos se realiza leyendo y escribiendo en una memoria común de alta velocidad de acceso. Un sistema de memoria compartida es relativamente fácil de programar dado que todos los procesos comparten una misma vista de los datos y la velocidad de comunicación entre procesos está limitada solamente por el tiempo de acceso a esa memoria. El problema con este esquema es que debe haber controles de concurrencia y sincronización.

Esta clase de tecnología se encuentra disponible hoy en día en las CPU como también en el hardware destinado al procesamiento gráfico en computadoras personales normales, con la diferencia que estas últimas cuentan con cientos de núcleos y la memoria compartida se implementa de manera más sencilla al tratarse de arquitecturas SIMD (Instrucción Simple para Múltiples Datos).

1.4. GPU Computing

La Unidad de Procesamiento Gráfico (GPU) es el chip de las placas gráficas que efectúa las operaciones requeridas para renderizar píxeles en la pantalla. Las GPUs modernas de uso doméstico como la de la figura 1.4 están optimizadas para ejecutar una

instrucción simple sobre cada elemento de un extenso conjunto simultáneamente (*i.e.* SIMD).

Figura 1.4: Tarjeta gráfica XFX con chip NVIDIA GeForce 8800 GTS.

La fuerza impulsora del desarrollo de las placas gráficas ha sido el negocio millonario de la industria de los videojuegos. De hecho, la performance de las GPUs ha excedido la ley de Moore por más de 2.4 veces al año [22]. La causa de esto es la ausencia de circuitos de control, que es lo que ocupa mayor espacio en una Unidad Central de Procesamiento tradicional (CPU). En contraste, las GPU dedican casi la totalidad del espacio de chip a Unidades Aritmético Lógicas (ALU), incrementando enormemente el poder de procesamiento, pero penalizando severamente las elecciones de bifurcación incorrectas. Esto es cuando *threads* paralelos toman diferentes caminos en sentencias condicionales del tipo *if – else*.

Para el uso tradicional en computación gráfica esto no es un inconveniente, pero cuando se intenta utilizar la placa gráfica como un procesador de propósito general, es un problema complicado que debe ser tenido en cuenta.

El esquema de la figura 1.5 muestra las cantidades de transistores dedicados a diferentes tareas en la CPU comparado con la GPU.

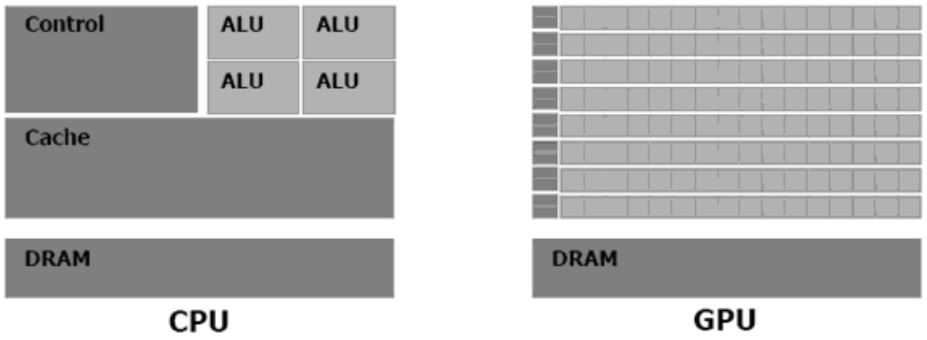


Figura 1.5: Comparación del uso de transistores entre CPU y GPU [23].

Las GPUs actuales están muy bien adaptadas al tipo de cálculo requerido para la simulación con autómatas celulares. Los modelos de AC se ejecutan generalmente sobre una grilla de celdas mientras que las placas gráficas programables están optimizadas para ejecutar cálculos sobre los píxeles de un monitor, los cuales pueden considerarse como una grilla de celdas. Las GPUs alcanzan altos niveles de performance a través de la paralelización, son capaces de procesar múltiples vértices y píxeles simultáneamente. Este alto nivel de paralelismo permite que las simulaciones corran significativamente más rápido sobre la GPU que sobre la clásica CPU. Otra característica de las placas gráficas es que están optimizadas para realizar múltiples lecturas de texturas por ciclo, lo cual se puede aprovechar almacenando las grillas de simulación como texturas.

Una primera demo de NVIDIA [24] mostraba mapeo de texturas implementando el Juego de la Vida en la GPU. Desde entonces se ha estudiado la posibilidad de implementar ACs más complejos sobre hardware de gráficos. Existen algunas implementaciones particulares de modelos de Lattice Boltzmann que simulan el comportamiento de gases en tiempo real utilizando texturas para representar la red sobre la GPU [25]. Otros trabajos como el de Harris, Coomble, Scheuermann y Lastra [26] utilizan AC de punto flotante para crear motores físicos de evaporación, condensación y formación de nubes sobre placas gráficas, que han conseguido aceleraciones de hasta 25 veces por sobre una implementación similar en CPU. Otras implementaciones del Método de Lattice Boltzmann para la simulación de fluidos sobre GPUs, como la de Li, Wei y Kaufman [27] alcanzan aceleraciones de un orden de magnitud con respecto a una misma versión para CPU.

Estos y otros trabajos, que son el punto de partida del denominado "*GPGPU-Computing",* o Computación de propósito general sobre unidades de procesamiento gráfico, fueron implementados mediante lenguajes de programación de gráficos como OpenGL o DirectX, mucho más reducidos que un lenguaje de programación general en cuanto a los tipos de estructuras de datos, operaciones de memoria y estructuras de control. La técnica consiste básicamente en traducir el procesamiento de datos como deformación de texturas gráficas. Si bien estos trabajos muestran que las GPUs son una poderosa herramienta para acelerar simulaciones de ACs relativamente simples, modelos más complejos con una traducción a textura menos lineal requieren diferentes estrategias para su implementación eficiente en hardware de gráficos.

En febrero de 2007 la empresa NVIDIA publicó el primer SDK CUDA, la primera plataforma específica para computación sobre GPU, compuesta por un compilador y un conjunto de herramientas de desarrollo que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos que corran sobre GPUs. Las aplicaciones CUDA funcionan en todas las GPUs NVIDIA de la serie G8X en adelante, incluyendo GeForce, Quadro y la línea Tesla, que fueron desarrolladas en conjunto con el SDK.

Como se verá, el presente trabajo se presenta siguiendo la cronología del desarrollo realizado. La tecnología del GPU Computing y en particular CUDA, así como la bibliografía especializada en este tema, sufrieron una evolución muy abrupta en los últimos 5 años en coincidencia al desarrollo de esta tesis. Es por esto que la versión de la plataforma y las placas gráficas utilizadas no son la misma en cada uno de los capítulos e incluso algunas decisiones de implementación variaron a lo largo del tiempo por la evolución del lenguaje de programación y la arquitectura de las placas gráficas.

Capítulo 2: Plataforma y modelo CUDA

2.1. Arquitectura de los procesadores

En una CPU tradicional existe una mayor dedicación de espacio o cantidad de transistores al control que al procesamiento. Además existe un esquema jerárquico de memoria compuesto por una memoria principal y varias memorias caché divididas en niveles que, a medida que nos acercamos al procesador, son más pequeñas (menos bytes) y rápidas (se accede en menos ciclos de reloj). El procesador accede a la memoria a través de las *cachés* (figura 2.1).

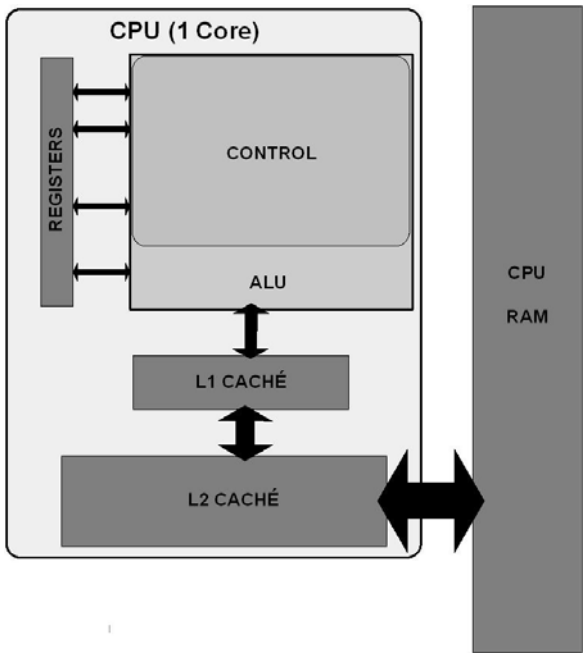


Figura 2.1: Esquema simplificado de memoria de una CPU tradicional con un solo núcleo.

En una GPU con tecnología previa a CUDA existen múltiples procesadores pequeños (del orden de cien) con muy poca dedicación al control y gran poder de procesamiento (figura 2.2), una sola memoria principal (de entre 256 megas y 2 gigas), y el flujo de los datos es en un solo sentido. Es decir, los datos se leen de un área y se escriben en otra (no es posible realizar la operación A[i] = A[i+1] por ejemplo). Los datos se copian de la memoria principal del procesador (CPU RAM) a la memoria principal de la placa y viceversa.

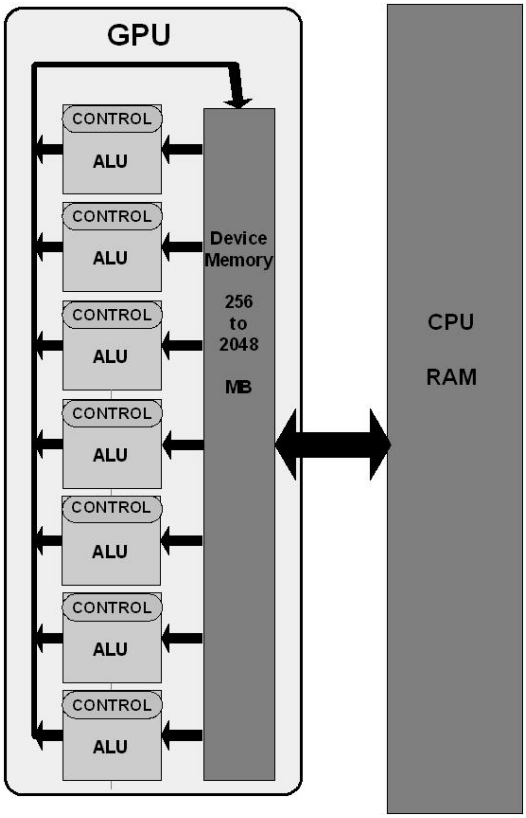


Figura 2.2: Esquema simplificado de una GPU tradicional con varios núcleos.

En una GPU NVIDIA los procesadores se agrupan de a 8 en Multiprocesadores con bancos de registros y memoria compartida de acceso rápido (figura 2.3). Como en todas las placas gráficas, también existe una memoria principal, la cual se comunica con la memoria RAM del sistema. Los *threads* que corren dentro de los procesadores pueden acceder directamente a cualquiera de los niveles de memoria en ambos sentidos, y a su vez son los encargados de administrar estos espacios. Además, la GPU cuenta con un *scheduler* o *thread excecution manager* que administra la ejecución de los mismos.

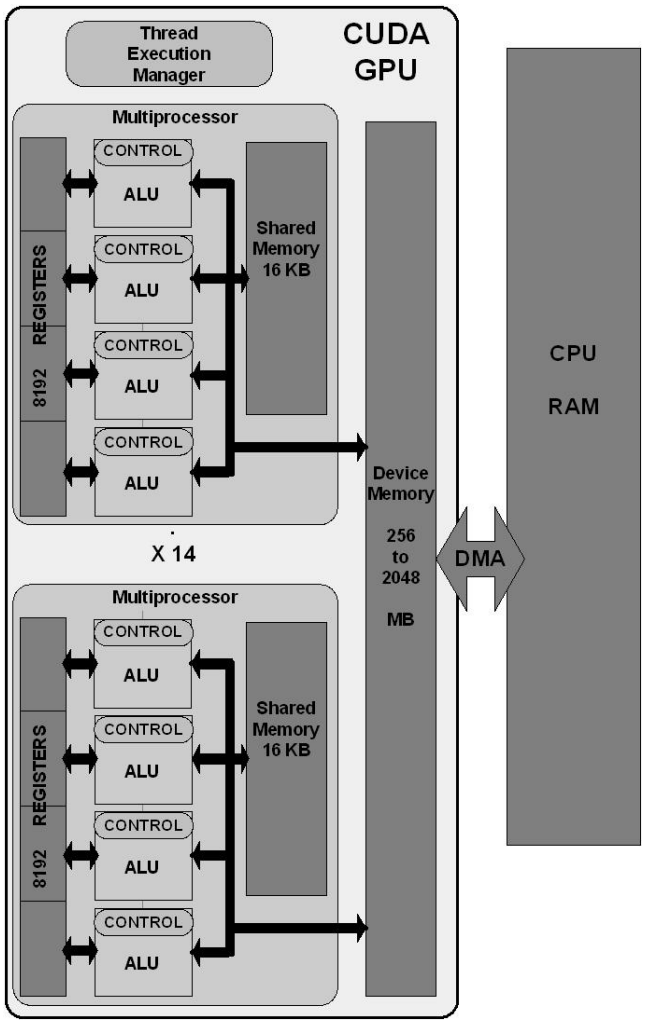


Figura 2.3: Esquema simplificado de una GPU NVIDIA con tecnología CUDA..

Las placas gráficas utilizadas en este trabajo tienen distintas GPUs NVIDIA, todas ellas compatibles con el lenguaje de programación CUDA. Se utilizaron placas de la serie G80 (GeForce 8600 GT y 8800 GT) y de la serie GTX 200 (GTX 260). Estas series de placas gráficas de la empresa NVIDIA fueron desarrolladas en conjunto con el modelo de programación CUDA [28] de la misma marca. La G80 es la primera línea de GPUs compatibles con esa tecnología.

Las GPUs CUDA están compuestas por un conjunto de multiprocesadores. Cada uno de los procesadores dentro de los multiprocesadores ejecuta la misma instrucción en cada ciclo de reloj, pero sobre diferentes datos simultáneamente en lo que se define como arquitectura SIMT (*single instruction multiple thread*). Esta arquitectura, a diferencia de la SIMD (*single instruction multiple data*) permite que *threads* paralelos

puedan seguir diferentes caminos en el caso de existir instrucciones condicionales. Cada uno de estos multiprocesadores tiene sus propias memorias de los siguientes cuatro tipos (figura 2.4):

1. Registros de 32-bits por procesador.
2. Memoria caché de datos paralela o memoria compartida que se comparte entre todos los procesadores.
3. Una caché constante de lectura solamente compartida por todos los procesadores, que acelera las lecturas de la memoria de constantes, que se implementa como un área dentro de la memoria del dispositivo de lectura.
4. Una caché de textura de lectura solamente, compartida por todos los procesadores que sirve para acelerar las lecturas del espacio de memoria de textura.

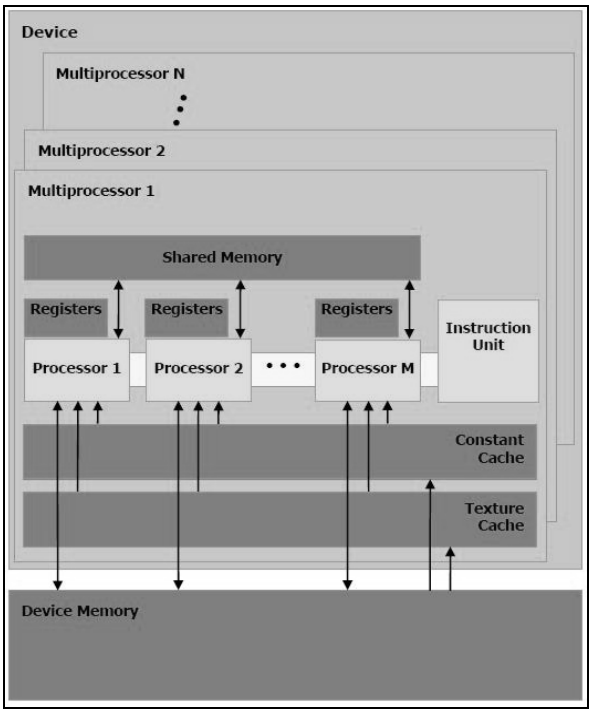


Figura 2.4: Conjunto de multiprocesadores SIMD con memoria compartida en una GPU NVIDIA. [23].

Los espacios de memoria local y global se implementan como regiones de lectura-escritura de la memoria del dispositivo y no se guardan en caché. Cada multiprocesador accede al caché de textura vía una unidad de textura que implementa

los distintos modos de acceso y filtros de datos. En la tabla 2.1 se detallan las especificaciones principales de las GPUs utilizadas en esta tesis.

NVIDIA GeForce	**8600 GT**	**8800 GT**	**GTX 260**
Cantidad de Multiprocesadores	4	14	24
Cantidad de procesadores de *shaders*	32	112	192
Tamaño de Memoria global	256 MB	512 MB	896 MB
Tamaño del bus de memoria	128 bit	256 bit	448 bits
Ancho de banda de memoria	22.4 GB/s	57.6 GB/s	111,9 GB/s
Performance pico estimada	114.2 Gflops	504 Gflops	715 Gflops
Reloj del núcleo	540 Mhz	600 Mhz	576 Mhz
Reloj del procesador	1180 Mhz	1500 Mhz	1242 Mhz
Reloj de la memoria	700 Mhz	900 Mhz	999 Mhz
Compatibilidad computacional CUDA	1.1	1.1	1.3

Tabla 2.1: Características de las GPUs NVIDIA utilizados.

2.2. Modelo de programación CUDA

La tecnología CUDA de NVIDA es una nueva arquitectura diseñada para permitir a la placa gráfica resolver problemas computacionales complejos. CUDA facilita el acceso de aplicaciones con alto costo computacional al poder de procesamiento de las GPU a través de una nueva interfaz de programación. La utilización del lenguaje C estándar simplifica en gran medida el desarrollo de software. El conjunto de herramientas de CUDA es una solución completa para programar placas gráficas compatibles. El *toolkit* incluye bibliotecas estándar FFT (*Fast Fourier Transform*) y BLAS (*Basic Linear Algebra Subroutines*), un compilador de C para la GPU y un driver específico. La tecnología CUDA es compatible con los sistemas operativos Linux y Windows® (XP®,Vista® y 7®) y MacOS.

2.2.1. Interfaz para la programación de aplicaciones (API)

La GPU, denominada *device* se ve como un dispositivo computacional capaz de ejecutar un gran número de hilos de ejecución en paralelo (*threads*). Opera como un coprocesador para la CPU principal denominada *host*. Las porciones de aplicaciones de cálculo intensivo corriendo en la CPU principal se descargan al dispositivo llamando a funciones denominadas *kernel,* que se ejecuta en la placa en forma de *threads* múltiples paralelos. Tanto el *host* como la GPU mantienen su propia memoria RAM, llamadas *host memory* y *device memory*, respectivamente. Se pueden copiar datos de una RAM a la otra a través de métodos optimizados que utilizan el acceso directo a memoria (DMA) de alta performance de la propia placa.

2.2.2. Bloques de *threads*

Un bloque de *threads* (*thread block*) es un grupo de hilos de ejecución que pueden cooperar entre sí compartiendo eficientemente datos a través de la memoria compartida de acceso rápido y sincronizando sus ejecuciones, lo cual permite coordinar el acceso a los datos especificando puntos de sincronización en el *kernel*. Cada *thread* se identifica por su *thread ID*, que es el número dentro del bloque. Una aplicación puede especificar un bloque como un arreglo tridimensional utilizando un índice de 3 componentes. El esquema del bloque se especifica en la llamada a función del dispositivo mediante una variable que contiene los enteros que definen las cantidades de *threads* en x, y, z.

Internamente, la variable global de la función *blockDim* contiene las dimensiones del bloque. La variable interna global *threadIdx* (*thread index*) contiene el índice del *thread* dentro del bloque. Para explotar el potencial del hardware eficientemente un bloque debe contener por lo menos 64 *threads* y no más de 512 [29].

2.2.3. Grillas de bloques de *threads*

Existe un número máximo de *threads* por bloque limitado por el hardware, el cual incluso puede ser menor según la cantidad de memoria local y compartida utilizada por cada uno. Sin embargo, bloques que ejecuten el mismo *kernel* pueden ser agrupados en una grilla de bloques, con lo que el número de *threads* que pueden ser lanzados en un solo llamado es mucho mayor (figura 2.5). Esto tiene como contrapartida una menor

cooperación entre *threads*, ya que los *threads* de diferentes bloques no pueden comunicarse de forma segura o sincronizarse entre sí. Cada bloque se identifica por su *block ID* y la estructura de una grilla se especifica en el llamado a la función del mismo modo que la dimensión de los bloques. Los diferentes bloques de una grilla pueden correr en paralelo y para aprovechar el hardware eficientemente se deberían utilizar al menos 16 bloques por grilla [29].

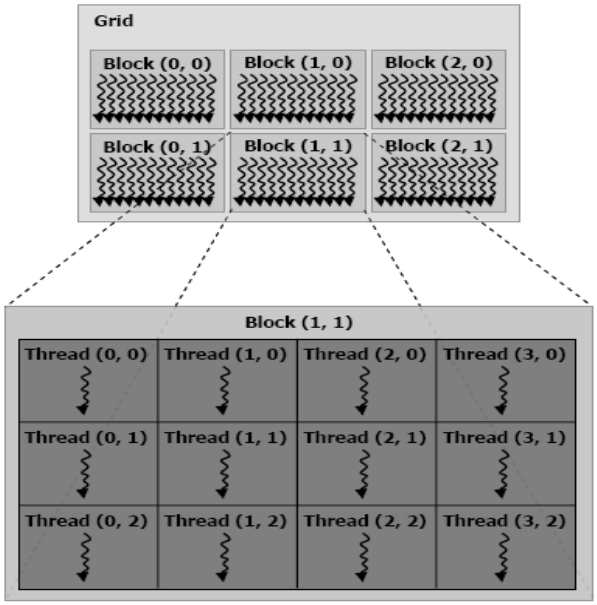


Figura 2.5: Bloques de threads organizados en una grilla de bloques [23].

2.2.4. Extensión del lenguaje C

Con la aparición de las GPUs multi-núcleo los procesadores principales son ahora sistemas paralelos. Más aún, el paralelismo crece con la ley de Moore. El desafío es desarrollar aplicaciones de software paralelas que se adapten de manera transparente a la cantidad de núcleos de la manera en que las aplicaciones gráficas 3D se adaptan a las diferentes placas de videos con gran diversidad de núcleos. El modelo de programación paralela CUDA fue diseñado para cumplir este desafío al mismo tiempo que mantenga una curva de aprendizaje baja para los programadores familiarizados con lenguajes de programación estándar como C.

El lenguaje CUDA es una extensión del lenguaje C que permite al programador definir funciones C, denominadas *kernels*, las cuales al ser llamadas se ejecutan

paralelizadas n veces en n *threads* diferentes. Esto contrasta con una función C clásica, que ejecuta en uno solo *thread* cada vez que es llamada.

Un *kernel* se define utilizando el especificador de declaración *__global__* y especificando la cantidad de *threads* CUDA que lo ejecutan mediante la configuración de ejecución <<<…>>>. Cada *thread* que ejecuta el *kernel* tiene un identificador único que es accesible mediante la variable interna *threadIdx*. Como ejemplo, se muestra un código que calcula la función (2.1) de manera secuencial en código C (figura 2.6) y de manera paralela en CUDA (figura 2.7).

$$y[i]=\alpha x[i]+y[i]; \quad i=0:N \tag{2.1}$$

```
void f_serial (float alpha, float *x, float *y){
      for(int i=0; i<N; i++)
            y[i] = alpha*x[i] + y[i];
}
// Invocación de la función:
f_serial(alpha, x, y);
```

Figura 2.6: Código C secuencial y llamado de la función.

```
__global__ void f_parallel (float alpha, float *x, float *y){
int i = blockIdx.x*blockDim.x + threadIdx.x;
      y[i] = alpha*x[i] + y[i];
}
// Invocación del kernel f_parallel (256 threads por bloque)
int nblocks = (N+255) / 256; //Calculo la cantidad de bloques
f_parallel<<<nblocks, 256>>>(alpha, x, y);
```

Figura 2.7: Código CUDA paralelo y llamado de la función con 256 threads por bloque.

El núcleo CUDA encierra tres abstracciones clave: la jerarquía de grupos de *threads*, la memoria compartida y las barreras de sincronización (que el programador las ve como simples extensiones del lenguaje). Estas abstracciones proveen paralelismo de datos fino a nivel de *threads* y paralelismo a nivel de tarea mediante bloques. Los problemas pueden dividirse en sub-problemas que se resuelven independientemente en paralelo en bloques de *threads* y cada sub-problema se resuelve también en paralelo de manera cooperativa entre los *threads* asignados. La escalabilidad es directa ya que el mismo código compilado puede correr en GPUs con diferente número de núcleos como lo muestra la figura 2.8, beneficiándose del administrador o *thread scheduler*.

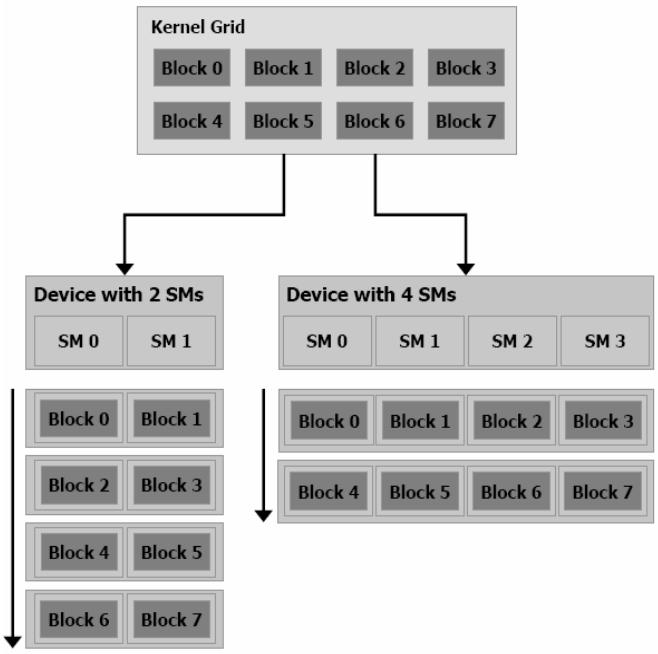


Figura 2.8: Escalabilidad de un kernel CUDA en distintos dispositivos [23].

2.2.5. Tipos de funciones

Los calificadores de funciones o *Function Type Qualifiers* se utilizan para definir el tipo de función, a saber:

- ***device*** es una función que se ejecuta en la placa gráfica y se puede llamar sólo desde la placa.
- ***global*** es una función *kernel*. Se ejecuta en el dispositivo y se puede llamar sólo desde un programa que corre en la CPU o *host*. Cualquier llamada a una función *global* debe especificar la configuración de ejecución (dimensiones de la grilla de bloques y de los bloques).
- ***host*** es una función que se ejecuta en la CPU y que sólo se puede invocar desde código que corre en la CPU, del mismo modo que una función C clásica.

2.2.6. Tipos de variables

- ***device*** es una variable que reside en el espacio global de memoria de la placa. Es accesible desde todos los *threads* de la grilla (con una latencia aproximada de 200-300 ciclos de reloj) y desde el *host* a través de las bibliotecas.

- ***shared*** es una variable que se almacena en el espacio compartido de memoria de un bloque y es accesible únicamente desde todos los *threads* del bloque (pero con una latencia aproximada de sólo 2 ciclos de reloj).

2.2.7. Administración de la memoria

Existen funciones específicas para la administración de la memoria del dispositivo:

- ***cudaMalloc(…)*** ubica una cantidad especificada de bytes de memoria consecutiva y retorna un puntero a ese espacio que puede utilizarse para cualquier tipo de variable.
- ***cudaMemcpy(…)*** copia bytes de un área de memoria a otra. Pudiendo copiar desde la memoria principal de la CPU a la memoria de la placa, a la inversa y entre diferentes espacios de la memoria de la GPU.
- Ambas funciones sólo pueden ser llamadas desde código que corre en la CPU.

2.3. Sincronización

La función *syncthreads()* define un punto de sincronización para todos los *threads* de un bloque. Una vez que todos los *threads* alcanzan este punto, la ejecución continúa normalmente. No existen funciones para sincronizar entre diferentes bloques. Esta función sólo puede utilizarse dentro de funciones del tipo *device*.

Capítulo 3: Modelo AQUA de escurrimiento superficial

En este capítulo se presenta una implementación en CUDA de un Autómata Celular para simulación de escurrimientos superficiales de fluidos, analizando diferentes estrategias de paralelización y aprovechando los distintos tipos de memoria de la placa gráfica. El modelo en cuestión, denominado AQUA, se trata de un algoritmo empírico que simula el movimiento superficial dirigido por las diferencias de alturas de terreno.

La GPU utilizada en este desarrollo es una GeForce 8600 GT con compatibilidad computacional CUDA 1.1 y la versión del SDK es la 1.1.

3.1. Evolución de AQUA

La serie de modelos AQUA han demostrado gran capacidad para simular eventos reales de tormenta para amplias regiones de llanura con bajo tiempo de simulación debido a la escasa complejidad computacional de sus cálculos y aprovechando el alto nivel de detalle de los Modelos Digitales del Terreno (MDT) generados en base a imágenes satelitales.

El primer modelo AQUA es un algoritmo de autómatas celulares desarrollado para la simulación de escurrimiento superficial [30]. Esta versión ya presentaba grandes ventajas con respecto a los modelos clásicos de hidrología en cuanto a precisión y nivel de detalle. En la segunda versión [05] [06] el modelo fue reformulado y calibrado con datos provenientes de inundaciones reales. AQUA utiliza una grilla regular como representación de la geometría del terreno. La información sobre la topografía se encuentra discretizada en celdas de igual tamaño que contienen un valor de altura del terreno y cota de agua en cada tiempo discreto. El modelo básicamente toma conjuntos de 9 celdas siguiendo la vecindad de Moore de rango 1 [31] y redistribuye el agua contenida buscando la cota más baja.

En este capítulo se describirá la implementación en GPU de la última versión del algoritmo, denominada AQUA-GRAPH [32], la cual introduce parámetros de arcos entre celdas. Posteriores evoluciones de este modelo como GRAPHFLOW [33] utilizan una estructura de datos grafo y permiten trabajar con células de diferentes tamaños para reducir el costo computacional en base a diversos tipos de discretizaciones temporales y espaciales.

3.2. Modelo de escurrimiento AQUA-GRAPH

En AQUA-GRAPH cada celda de una grilla regular, identificada como *n(x,y)*, representa una porción del terreno y contiene la información de la cota promedio *h(x,y)*, el nivel de agua *w(x,y)*, y fuentes-sumideros [32] (figura 3.1).

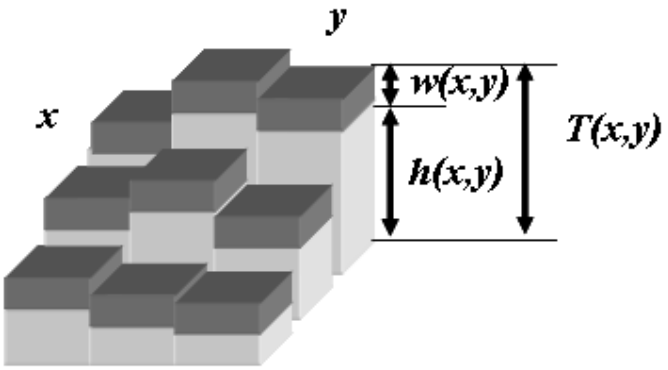


Figura 3.1: Representación interna del modelo.

T(x,y,t) define la altura total de la celda ubicada en la coordenada *(x, y)* en un paso de tiempo determinado, y es la suma de la cota de terreno *h(x,y)* más el nivel de agua en ese paso de tiempo *w(x,y,t)*:

$$T(x,y,t) = h(x,y) + w(x,y,t) \tag{3.1}$$

A cada paso, el agua se transfiere entre celdas vecinas siguiendo una ley hidráulica discretizada. Los pasos del algoritmo de escurrimiento son los siguientes:

1. Para cada celda determinar el entorno de recepción *R(x,y,t)* comprendido por las celdas vecinas con altura total *T(x,y,t)* inferior a la celda considerada (eq. 3.2 – fig. 3.2):

$$R(x,y,t) = \{x(u,v) / x-1 \le u \le x+1,\ y-1 \le v \le y+1 \wedge T(u,v,t) < T(x,y,t)\} \tag{3.2}$$

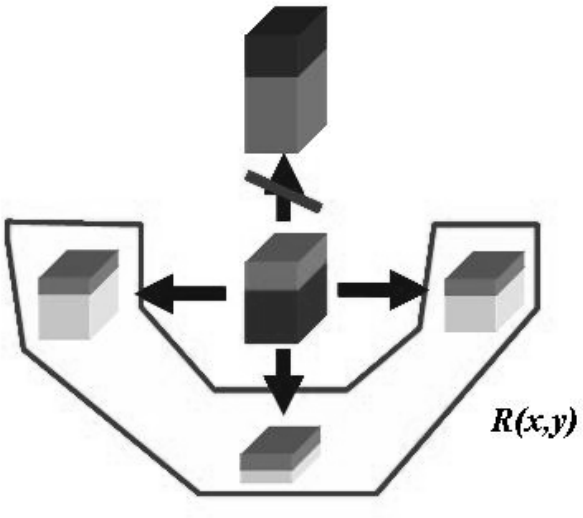


Figura 3.2: Entorno de recepción de una celda (centro).

2. Calcular el nivel que alcanzará el agua en la celda y su entorno si todo el líquido escurriera a su mínima posición, llamada altura de equilibrio *Teq(x,y,t)* (ver la figura 3.3).

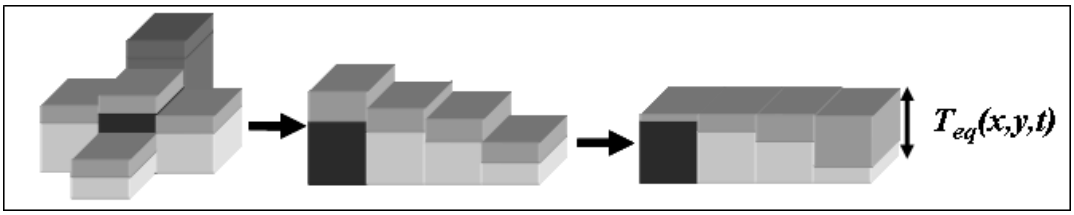


Figura 3.3: Cálculo de Teq(x,y,t) para la celda n(x,y) (oscura) y su entorno R(x,y,t) (gris). El nodo más alto (gris oscuro) pertenece a la vecindad pero no al entorno.

3. Determinar la cantidad de agua que saldrá de la celda *Δw(x,y,t)* controlando que la diferencia entre la altura de equilibrio y la total no sea mayor al total de agua en la celda. Si es así solo puede transferirse el nivel de agua disponible tal como lo describe la siguiente ecuación:

$$\Delta w(x,y,t) = \min\left[T(x,y,t) - T_{eq}(x,y,t),\, w(x,y,t) \right] \qquad (3.3)$$

4. Distribuir *Δw(x,y,t)* entre las celdas del entorno *R(x,y,t)* pesando con la raíz cuadrada de las diferencias de altura siguiendo la ley de pérdida de carga de sistemas hidráulicos [34]:

$$\Delta w(x,y,u,v,t) = \Delta w(x,y,t) \frac{\sqrt{T(x,y,t) - T(u,v,t)}}{\sum_{(j,k) \in R(x,y,t)} \sqrt{T(x,y,t) - T(j,k,t)}} \quad (3.4)$$

5. Transferir a cada celda *n(u,v)* en *R(x,y,t)* el nivel de agua correspondiente *Δw(x,y,u,v)* multiplicado por un coeficiente de relajación *α*. Este parámetro regula la transferencia de flujo entre las celdas en cada iteración y permite definir la relación entre esta última y el tiempo real del problema físico que se intenta resolver:

$$w(u,v,t+1) = w(u,v,t) + \alpha \Delta w(x,y,u,v,t) \quad (3.5)$$

6. Una vez que se ejecutaron todas las celdas de la grilla, actualizar el paso del tiempo haciendo:

$$w(x,y,t) = w(x,y,t+1) \quad (3.6)$$

En el Apéndice 1 se da un ejemplo simple de autómata celular donde se muestra como se introduce, de manera similar a la ecuación 3.5 un parámetro de relajación para fijar la escala de tiempo de una iteración del modelo.

3.3. Implementación paralela en CUDA

A continuación se detallan los pasos seguidos para la implementación del código AQUA utilizando el lenguaje CUDA y las diferentes estrategias de paralelización investigadas.

3.3.1. Representación de los datos

Los datos de altura de terreno y nivel de agua se almacenan en un arreglo denominado *hTerreno* de tipo *host* de variables *float2* (que contienen dos campos de punto flotante denominados *x* e *y*). Luego se declaran dos arreglos *float2* del mismo tamaño, *dTerreno* y *dAux,* con el modificador *device* para que se alojen en la memoria de la placa gráfica. El primero se usa para copiar los datos de *hTerreno* y el segundo como estructura auxiliar. Ambos arreglos se inicializan con los datos de entrada. Dentro de los *kernels* no se declaran grandes estructuras sino variables temporales auxiliares, las cuales se alojan en los registros sin necesidad de declararlas con modificadores.

3.3.2. Kernels

La ejecución se divide en 2 pasos principales consecutivos implementados como dos *kernels*. En el primero, denominado *Flow(),* cada celda ejecuta los cinco primeros pasos del algoritmo presentado en la sección 3.2, y se almacenan los resultados parciales en el arreglo auxiliar *dAux*. Para evitar problemas de concurrencia, se acumula en uno de los campos del arreglo auxiliar los incrementos de agua en las celdas vecinas y en el otro campo los decrementos de la celda que se analiza. Por ejemplo, una celda ubicada en la posición *"i"* del arreglo tiene a *dTerreno[i].x* como altura de terreno y *dTerreno[i].y* como cota de agua en un determinado tiempo *t*. De esta manera, si esta celda transfiere un nivel de agua *w* a la celda ubicada en *"j"*, el incremento en la altura de agua de la celda vecina se almacena en *dAgua[j].y* y el decremento en la celda origen se almacena en *dAgua[i].x*. Este primer paso es el que puede generar problemas de concurrencia a la hora de paralelizar, porque la ejecución sobre una celda debe leer y actualizar varias celdas vecinas.

En el segundo *kernel*, denominado *Actualize()* se ejecuta para cada celda el último paso del algoritmo presentado, el cual consiste en aplicar al nivel de agua actual de cada celda tanto el incremento como el decremento almacenado en el arreglo auxiliar. Este paso es totalmente paralelizable ya que cada celda se actualiza a sí misma sin interactuar con las vecinas.

3.3.3. Threads y Bloques

En la implementación inicial, con grillas de hasta 256 × 256 celdas, todas las simulaciones se realizaron con un solo bloque de *threads*. Cada *thread* recorre una única fila de la grilla de principio a fin ejecutando el algoritmo. La fila sobre la que ejecuta cada *thread* se determina por la variable interna *threadIdx.y.* Los *threads* se sincronizan al comenzar y finalizar el recorrido. El inconveniente de esta paralelización es que *threads* que comparten alguna celda vecina actualizan al mismo tiempo celdas de la grilla auxiliar (figura 3.4).

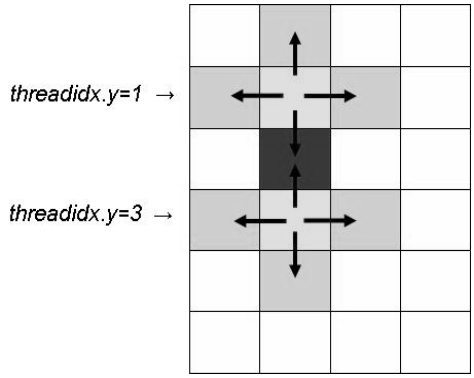


Figura 3.4: Ambos threads actualizan la celda 1,3 (oscura) al mismo tiempo.

La mayoría de los simuladores que han sido implementados exitosamente sobre GPUs [35], [29], [36] son modelos que para cada celda leen el estado de las vecinas, computan el nuevo estado de la celda y lo actualizan en memoria auxiliar. Estos simuladores no tienen problemas de concurrencia al ser paralelizados, independientemente de la tecnología aplicada.

En el trabajo de Podlozhnyuk [37] se implementan índices desfasados sobre memoria compartida que evitar errores de concurrencia cuando varios *threads* escriben el mismo dato. En nuestro caso se optó por una solución similar desfasando los índices en función de la fila que recorren y tratando a cada línea como un arreglo circular. Los *threads* empiezan a recorrer la fila desde un punto intermedio y al llegar al final saltan al inicio para completar el recorrido.

Esta simple solución es factible gracias a que los *threads* son estrictamente paralelos en una arquitectura SIMD y que el dominio se definió como una grilla regular. En dominios más complejos definidos por estructuras de tipo grafo donde AQUA-GRAPH sigue siendo válido, la división de tareas de actualización debería hacerse dinámicamente por métodos del estilo de coloreo de grafo.

El corrimiento de índices se muestra en la figura 3.5. También deben incluirse sincronizaciones dentro de los ciclos de cada *thread,* de manera que los índices sigan conservando la separación a lo largo del las filas. En la figura 3.6 se muestra el código del *kernel* que maneja los índices.

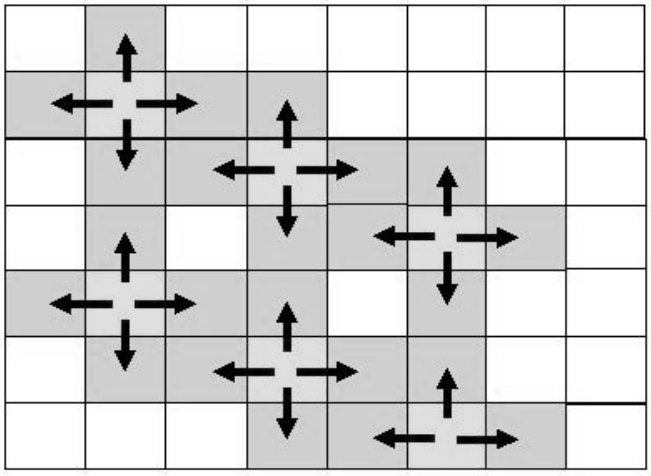


Figura 3.5: Cada thread comienza 3 celdas más adelante que el anterior para no producir errores de concurrencia.

```
__global__ void Flow(float2* dTerreno, float2* dAux, int DIMX, int
DIMY){
      int idy = threadIdx.y;
      int desp = ((idy)*2)%6; //desplazamiento del inicio en c/thread
      __syncthreads();
      for(int p=0; p<DIMX; p++){ //recorre toda la dimension en x
            int indice = p + desp;
            idice = indice%DIMX;
            __syncthreads();
            …
      }
}
```

Figura 3.6: Código con el desplazamiento de índices dentro del kernel.

Para las GPUs utilizadas en este trabajo, el esquema de cálculo descrito sólo puede ser aplicado a grillas que no excedan las 256 celdas por lado, lo cual constituye el límite de *threads* que se pueden lanzar en simultáneo dentro de un bloque. Para grillas mayores la ejecución debe dividirse en bloques cuadrados de igual tamaño donde se ejecuta el mismo código *kernel* pero en diferentes bloques de *threads*. En este caso surge un inconveniente en las filas de los límites entre bloques, donde *threads* de un bloque deben actualizar celdas que están siendo analizadas por *threads* de otro bloque. Si bien este problema no es común a otros AC, existen generalmente problemas similares al unir porciones de datos procesados en paralelo. La solución adoptada es similar a la del método *LBExchange()* utilizado por Tölke [29].

Para evitar problemas de concurrencia, cada bloque ejecuta sobre las filas internas de la grilla, y al terminar esta ejecución se corre otra función *kernel* de un solo

bloque que calcula el modelo en los bordes horizontales de las divisiones de la grilla. Esta función se denomina *FlowUnion()*. El esquema se muestra en la figura 3.7 donde las líneas rojas marcan la división en bloques. El algoritmo se ejecuta con el *kernel Flow()* para las celdas naranja y con *FlowUnion()* para las celdas verdes. Las flechas indican el punto de inicio y el sentido de ejecución de los *threads* de cada bloque.

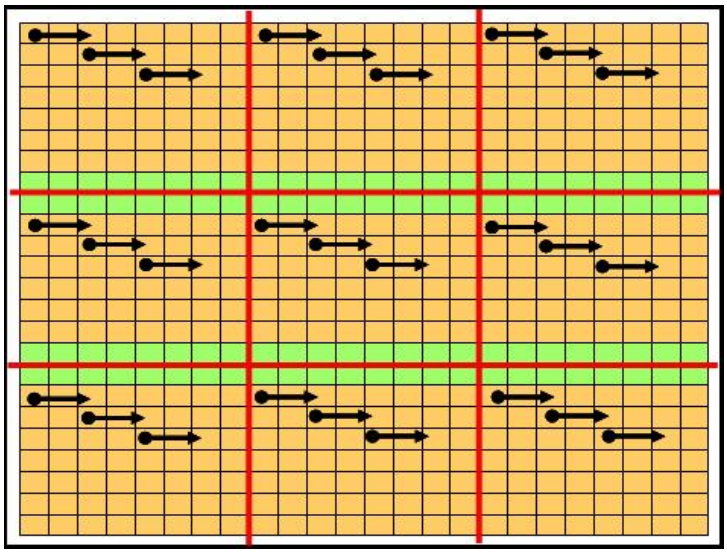


Figura 3.7: Configuración de bloques de threads para grandes tamaños de grillas.

3.4. Resultados

3.4.1. Implementación para comparación

Para realizar comparaciones de tiempo de respuesta con un cálculo equivalente sobre CPU se implementó el mismo algoritmo en lenguaje C++ con un solo *thread* de ejecución. Los datos de entrada son exactamente los mismos, pero sin necesidad de copiarlos a la memoria del dispositivo. Como no existen problemas del tipo *"race conditions"*, el algoritmo es más simple con solo dos recorridos de la grilla: el escurrimiento o *Flow()* donde se implementa hasta el paso 5 de la sección 3, y la actualización o *Actualize()* que implementa el último paso.

Tratándose de un modelo de escurrimiento superficial especialmente diseñado y calibrado para condiciones de escasa pendiente la validación del modelo se realizó con una simulación simple con parámetros ya definidos y comparando solamente los

resultados de ambas versiones, CPU y CUDA. Comparaciones de simulaciones de este modelo contra datos experimentales pueden encontrarse en la referencia [32].

Se compararon los resultados de ambos códigos en una simulación del escurrimiento superficial sobre un plano inclinado con un pulso fuente de agua en el contorno superior. Inicialmente todas las celdas tienen el mismo nivel de agua y un factor de relajación α de 0,005. Estas condiciones fuerzan a que el algoritmo se ejecute por completo para todas las celdas de la grilla durante toda la simulación. La pendiente del plano se eligió de manera que quede perpendicular a las líneas de ejecución de los *threads* para que el movimiento del agua genere interacción entre estos últimos. El gráfico de la figura 3.8 muestra la evolución de altura de agua en el punto central comparando las implementaciones del algoritmo en GPU y CPU. Las gráficas muestran gran concordancia entre los modelos. El error máximo es del orden de 10^{-4} al cabo de 4.000 iteraciones debido a errores de redondeo (el cálculo se hizo en simple precisión).

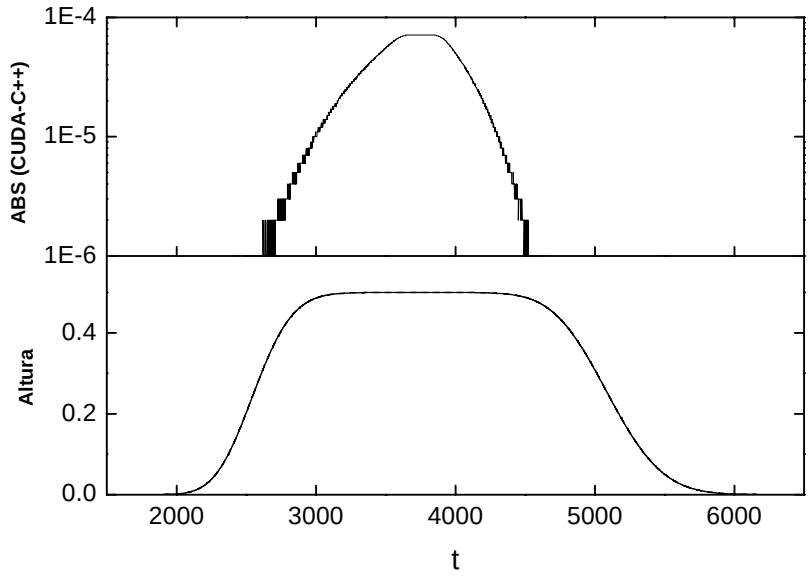


Figura 3.8: Evolución temporal de la altura de agua en la celda central (grilla de 512×512) de un plano inclinado con un pulso de agua en su borde superior. En el gráfico superior se muestra la diferencia entre los resultados obtenidos con CPU y GPU.

3.4.2. Performance

Se realizaron sucesivas simulaciones de 5000 pasos de tiempo cada una sobre escenarios cuadrados calculando tiempos de ejecución mediante el *timer* provisto por la biblioteca. Se tomaron en cuenta solamente los tiempos de ejecución para ambos ejemplos y no la creación de estructuras ni la copia de datos entre la memoria principal y la del dispositivo. Para cada tamaño de grilla se probaron diferentes configuraciones de ejecución de los *kernels* CUDA (bloques y *threads*).

En la tabla 3.1 se muestran el tiempo de ejecución promedio de una iteración para cada tamaño de grilla. La última columna indica la relación entre las implementaciones para CPU y para la placa gráfica. Los tiempos de CPU corresponden a una PC con procesador AMD Athlon 64 X2 Dual Core de 2,41 GHz y 4 GB. de memoria RAM. El sistema operativo es GNU/Linux Debian 5.0 Lenny y el código C++ fue compilado con GCC con las opciones de optimización por defecto.

Tamaño de la Grilla	**Versión C++**	**Implementación en CUDA**		**Speedup**
	Tiempo de CPU (ms.)	**Esquema de bloques y *threads***	**Tiempo de GPU (ms.)**	
128×128 16384	17.8	1 × 128	2.43	7.32
		4 × 64	2.25	**7.91**
256×256 65536	71.41	1 × 256	7.34	9.72
		16 × 64	6.76	**10.56**
512×512 262144	279.58	16 × 128	24.07	11.61
		64 × 64	23.65	**11.68**
1024×1024 1048576	961.62	16 × 256	113.62	8.45
		64 × 128	107.30	5.18
		256 × 64	107.09	8.97
		1024 × 32	72.61	**13.24**
		4096 × 16	75.80	12.68
		16384 × 8	111.19	8.64

Tabla 3.1: Comparación de tiempos de ejecución para las diferentes implementaciones según el tamaño de la grilla.

Si bien la ejecución en un solo bloque es más simple porque no requiere del paso de unión *UnionFlow()*, al dividir la ejecución de la misma grilla en bloques de *threads* la cantidad de celdas que recorre cada uno de los *threads* se reduce y aumenta el nivel

de paralelización. Es por esto que al aumentar la cantidad de bloques, los tiempos de ejecución disminuyen para un mismo tamaño de grilla.

En la figura 3.9, se grafica el tiempo que demora en promedio una iteración completa en función de la dimensión de la grilla para las dos versiones CPU y GPU. Para este último caso se grafican los tiempos de las configuraciones óptimas.

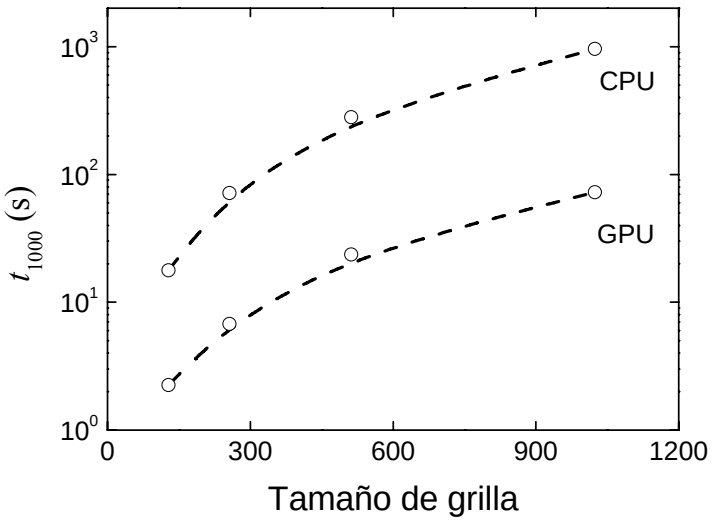


Figura 3.9: Comparación de tiempos de procesamiento (tiempo requerido para ejecutar 1000 pasos de tiempo).

La figura 3.10 muestra la cantidad de celdas calculadas en un microsegundo. Se puede observar como este valor permanece prácticamente constante para la implementación de CPU, mientras que para la versión GPU el nivel de *throughtput* aumenta notablemente con la paralelización.

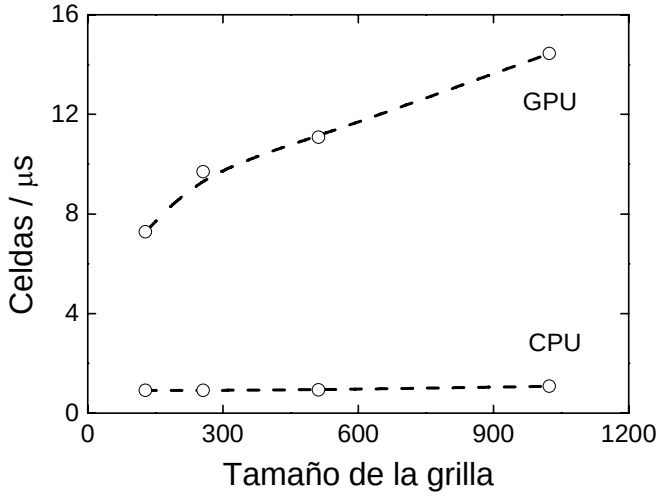


Figura 3.10: Comparación de throughtput.

3.4.3. Configuración ideal de bloques

Se ensayaron en la GPU seis configuraciones diferentes de bloques y *threads* utilizando la grilla más grande (1024 × 1024). Se dividió la grilla en 16 bloques cuadrados cada uno con 256 *threads* que ejecutan sobre 256 celdas cada uno. A partir de ahí se fue aumentando el nivel de paralelización, aumentando la cantidad de bloques y de esa manera la cantidad de *threads* totales ya que se divide el dominio en las dos dimensiones dejando menos celdas a recorrer por cada *thread*. En la última prueba se utilizaron 16384 bloques de 8 *threads* cada uno de los cuales ejecuta el código sobre 8 celdas. En la figura 3.11 pueden verse los tiempos de ejecución para 1000 iteraciones en función de la cantidad de *threads* utilizados. La configuración ideal es de 1024 bloques de 32 *threads* cada uno.

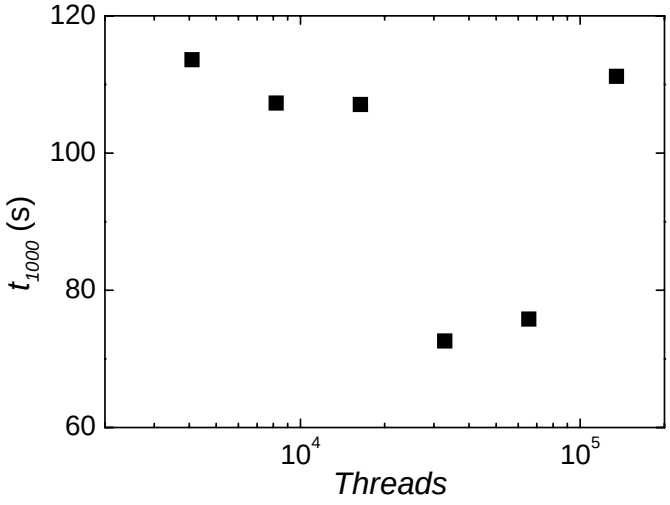


Figura 3.11: Comparación de tiempos de procesamiento de 1000 iteraciones de la GPU para diferentes configuraciones de threads.

3.4.4. Análisis de los resultados

Los resultados de performance son altamente alentadores ya que muestran una disminución en el tiempo de cálculo de un orden de magnitud con respecto a implementaciones similares para CPU, teniendo en cuenta que la GPU utilizada es de la primera generación de placas compatibles con la tecnología CUDA y destinada principalmente a PCs hogareñas. En cuanto al código para CPU, éste se muestra como referencia ya que no tiene ningún tipo de optimizaciones o paralelizaciones. Como el procesador es de doble núcleo, es posible que el tiempo de cálculo pueda reducirse llegando en un caso ideal a cerca de la mitad.

Si bien se evaluaron diferentes configuraciones de *threads* existen otras posibilidades por explorar como la optimización de ocupación de los recursos de la placa gráfica y el uso de las diferentes clases de memoria. Algunos autores recomiendan configuraciones óptimas de bloques de *threads,* pero no siempre funcionan eficientemente. Esto se debe fundamentalmente a particularidades del modelo AQUA-GRAPH y a las decisiones tomadas en la implementación presentada en este capítulo para solucionar los diferentes problemas de paralelización del código.

Por otro lado, el modelo AQUA es un claro ejemplo de que no se puede simplemente traducir el código C++ a lenguaje CUDA sino que se deben analizar

cuidadosamente los problemas de concurrencia que pueden surgir de la implementación en paralelo. En este caso se solucionó el problema desfasando índices y dividiendo el dominio logrando muy buenos resultados siendo que la GPU utilizada es de las más económicas de la primera generación CUDA.

Capítulo 4: Método de Lattice Boltzmann

4.1. Introducción

El autómata de gas reticular (LGA) y su último derivado, el método de Lattice Boltzmann, son ACs relativamente nuevos que sirven para simular fenómenos de transporte. En particular el método de Lattice Boltzmann (LBM) ha sido formulado como un modelo simple, eficiente y adecuado para la simulación de flujos de fluidos. A partir de cálculos de aritmética simple, el método genera soluciones de ecuaciones de flujos complejos, presentándose como una alternativa interesante a los esquemas numéricos tradicionales. Por ello, el método de Lattice Boltzmann se está convirtiendo en una poderosa herramienta de diseño en ingeniería.

A diferencia del modelo AQUA-GRAPH presentado en el capítulo 3 donde el estado de cada celda está determinado por una variable escalar que representa la altura física del fluido, en LBM este estado se define mediante un conjunto de variables escalares. Estas variables representan las porciones de partículas de fluido que se encuentran en la celda en ese momento y se mueven en determinadas direcciones discretas hacia celdas vecinas. Estas variables no son valores observables del fenómeno físico sino que sirven para calcularlos.

Las simulaciones en LBM pueden ser en grillas de una, dos, o tres dimensiones, donde cada nodo de la grilla representa un punto del espacio que contiene sólido o fluido. El flujo se simula como partículas de fluido que se propagan a través de la grilla en iteraciones discretas y colisionan unos con otros en los centros de las celdas de la grilla. La formulación algorítmica del método se divide básicamente en dos pasos consecutivos, denominados colisión y advección. El cálculo de estos dos pasos para cada una de las celdas de la grilla constituye una iteración del modelo y se puede tomar como inicio cualquiera de los dos [38]. Existen numerosas discretizaciones de grillas de LBM que se denominan generalmente de la forma $dXQY$ donde X indica el espacio dimensional e Y las posibles direcciones discretas de movimiento entre las que se encuentra la dirección nula o central.

Desde el punto de vista computacional, como las colisiones están restringidas a los nodos locales de la grilla, los cálculos de la colisión sólo dependen de datos

provenientes de celdas vecinas. Esta localidad espacial del acceso a datos convierte a LBM en un excelente candidato para la paralelización. Sin embargo, fácilmente paralelizable no es lo mismo que trivialmente paralelizable o "*embarrassingly parallel*". Por ejemplo, no es trivial el método ordinario para paralelizar por medio de la división de dominio, ya que en LBM implica la transferencia de porciones de fluido en las interfaces de división a través de las iteraciones.

4.2. Regla BGK para simulación de fluidos

Una importante simplificación de LBM fue hecha por Higuera y Jiménez [39] que linealizaron el operador de colisión, asumiendo que la distribución está cercana al equilibrio local. El mismo año fue propuesta por Higuera et al. [40] una regla de colisión mejorada, con estabilidad lineal. Sin embargo, la aproximación del operador de colisión presentada por Bhatnagar, Gross y Krook denominada BGK [41] que utiliza un término de relajación en dirección del equilibrio local es la más popular para simular Navier-Stokes (NS) [42], [43], [12].

La ecuación de Lattice Boltzmann para el modelo BGK, sobre una grilla de dos o tres dimensiones y Q direcciones se escribe generalmente como [42], [43]:

$$f_i\left(x+\delta t e_i, t+\delta t\right) - f_i\left(x,t\right) = \frac{1}{\tau}\left[f_i^{(eq)}\left(x,t\right) - f_i\left(x,t\right)\right],\ i = 0,1,...,Q-1 \quad (4.1)$$

donde $f_i(x,t)$ es la función de distribución de densidad a lo largo de la dirección e_i en la celda ubicada en x en la iteración t. Ambas escalas, espacial y temporal, se fijan como $|\delta x| = |\delta t| = 1$ en unidades de la grilla.

La regla de iteración básica del autómata LBM basado en esta ecuación consiste en dos pasos básicos denominados advección y colisión que se combinan mediante una relajación lineal. De ese modo, la viscosidad cinemática del fluido simulado puede ser controlada por el parámetro de relajación τ, de acuerdo a:

$$\nu = \left[\frac{\left(2\tau - 1\right)}{6}\right] e^2 \delta t \quad (4.2)$$

donde la velocidad de las partículas está dada por el vector $e_i = \delta x/\delta t$.

Para la versión de dos dimensiones implementada en este trabajo denominada d2Q9, los valores de las velocidades de las partículas se definen de la siguiente manera:

$$
\begin{aligned}
e_i &= (0.0) && i = 0 \\
e_i &= \left(\cos\left(\frac{\pi(i-1)}{2} \right),\ \sin\left(\frac{\pi(i-1)}{2} \right) \right) && i = 1,2,3,4 \\
e_i &= \sqrt{2} \left(\cos\left(\frac{\pi\left(i-4-\frac{1}{2}\right)}{2} \right),\ \sin\left(\frac{\pi\left(i-4-\frac{1}{2}\right)}{2} \right) \right) && i = 5,6,7,8
\end{aligned}
\tag{4.3}
$$

Entonces, los valores resultantes de los vectores direccionales $e_i = (Vx, Vy)$ son:

e_i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Vx	0	1	0	-1	0	1	-1	-1	1
Vy	0	0	1	0	-1	1	1	-1	-1

Tabla 4.1.Valores de los vectores direccionales e_i en el modelo d2Q9.

La figura 4.1 muestra las direcciones de las velocidades en una celda del modelo d2Q9.

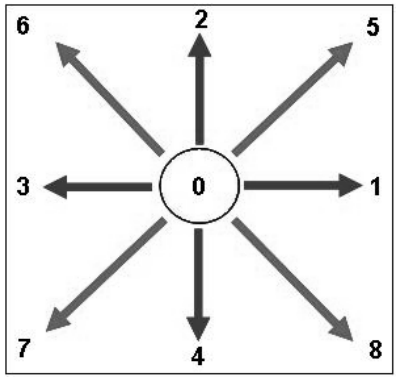


Fig.4. 1. LBM: Funciones de distribución en el modelo d2Q9.

Dentro de los modelos de tres dimensiones de LBM, existen numerosas variantes de acuerdo al número de direcciones utilizadas para discretizar una celda. Desde versiones de 27 direcciones hasta modelos minimalistas que logran simular procesos complejos con sólo 13 direcciones como la versión de d'Humières, Bouzidi y Lallemand [44], también los hay de 15, 19, 23, etc. Pero sin duda, el más ampliamente difundido es el d3Q19 (figura 4.2) propuesto por Qian et al [43] por presentar el mejor equilibrio entre precisión y memoria utilizada. En el trabajo de Keating et al. [45], se

comparan los resultados de simulaciones numéricas con 15, 19 y 27 direcciones encontrando pequeñas diferencias entre los resultados especialmente para altos números de Reynolds.

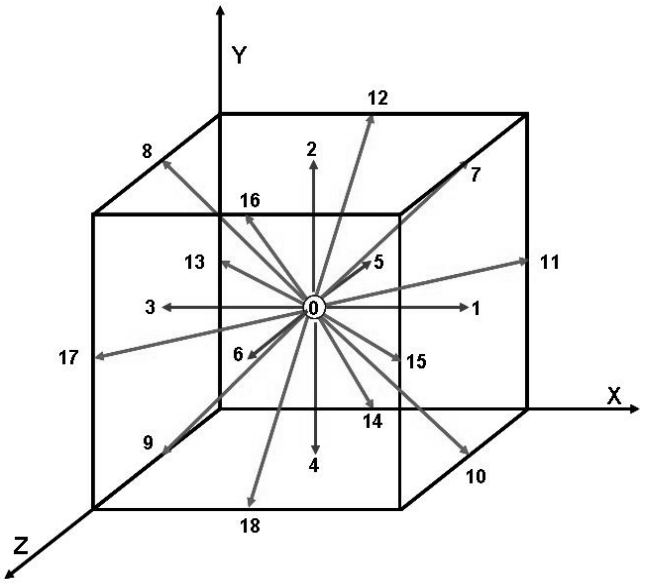


Fig.4.2: LBM: Funciones de distribución en el modelo d3Q19.

Para el modelo d3Q19, la ecuación de Lattice Boltzmann sigue siendo la 4.1 pero en este caso Q = 19. Y las direcciones discretas están dadas por:

e_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Vx	0	1	0	-1	0	0	0	1	-1	-1	1	1	0	-1	0	1	0	-1	0
Vy	0	0	1	0	-1	0	0	1	1	-1	-1	0	1	0	-1	0	1	0	-1
Vz	0	0	0	0	0	-1	1	0	0	0	0	-1	-1	-1	-1	1	1	1	1

Tabla 4.2: Valores de los vectores direccionales e_i en el modelo d3Q19.

4.3. Paso de advección

En este trabajo se utilizó un esquema "*pull*", donde el paso de advección se realiza previo a la colisión [38]. Este paso consiste en el cálculo para cada una de las celdas del avance de las partículas provenientes de las celdas vecinas a lo largo de sus direcciones de movimiento. El intercambio de partículas se realiza según la siguiente regla:

$$f^a(\vec{e},\vec{x},t) = f(\vec{e},\vec{x}-\vec{e}\,\delta t,t-\delta t) \tag{4.4}$$

donde f^a es la función de distribución advectada. La figura 4.3 muestra el ejemplo de advección para la versión d2Q9.

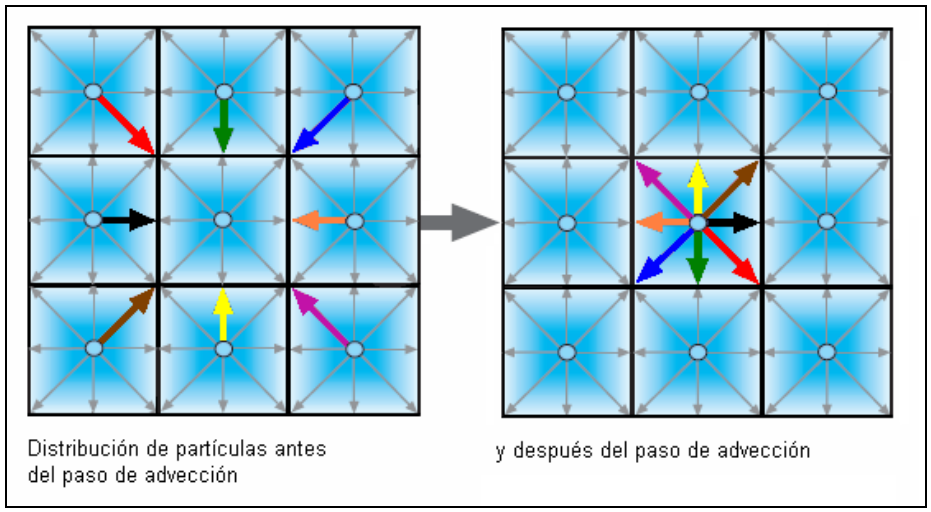


Figura 4.3: Paso de advección para la celda central.

4.4. Paso de colisión

El lado derecho de la ecuación 4.1 representa el término de colisión donde τ es la relajación temporal simple que controla el rango de aproximación al equilibrio. La función de distribución de equilibrio $f_i^{(eq)}(x,t)$ depende solamente de la velocidad y densidad local y tiene la siguiente forma [42]

$$f_i^{(eq)} = w_i\rho\left[1+3(e_i\cdot u)+\frac{9}{2}(e_i\cdot u)^2-\frac{3}{2}u\cdot u\right] \tag{4.5}$$

Donde w_i es un peso asociado con cada una de las velocidades de la grilla e_i, los cuales derivan de la distribución de Maxwell-Boltzmann [46]. Para el modelo d2Q9 los pesos son:

$$w_0 = \frac{4}{9},\quad w_i = \frac{1}{9},\ i = 1:4;\qquad w_i = \frac{1}{36},\ i = 5:8; \tag{4.6}$$

y para el modelo d3Q19 los pesos son:

$$w_0 = \frac{1}{9},\quad w_i = \frac{1}{18},\ i = 1:6;\qquad w_i = \frac{1}{36},\ i = 7:18; \tag{4.7}$$

La densidad de cada celda y la velocidad macroscópica del fluido se definen en términos de la función de distribución de la siguiente manera:

$$\sum_{i=0}^{Q} f_i = \rho \tag{4.8}$$

$$\sum_{i=0}^{Q} f_i e_i = \rho u \tag{4.9}$$

Este modelo tiene validez para flujos subsónicos (o sea cuando u es menor que la velocidad del sonido). Para los modelos d2Q9 y d3Q19 la velocidad del sonido es $c_s = e/\sqrt{3}$ [46], siendo este valor válido para estas grillas y difiriendo para otras geometrías. Este valor en general es reconocido como la "pseudo-velocidad del sonido" de la grilla y depende de los pesos w_i de las ecuaciones (4.7) y (4.8) respectivamente [47]. La presión es proporcional a la densidad y se calcula como:

$$p = c_s^2 \rho \tag{4.10}$$

4.5. Condiciones de contorno

Una de las cuestiones más complejas de la hora de realizar simulaciones correctas con LBM es lo concerniente a la definición de las condiciones de contorno. Ya que celdas situadas en el contorno de la grilla no tienen celdas vecinas de donde obtener las distribuciones en el paso de advección. En esos casos, las distribuciones se calculan en base a las existentes y a la condición física que se quiere dar a ese borde.

En las implementaciones que se presentan en los capítulos siguientes se utilizaron condiciones de no deslizamiento en las paredes, lo que significa velocidad neta cero en todas las direcciones. Para su implementación se siguió el esquema básico *on-grid bounce-back* [47] donde el límite que divide el sólido del fluido se ubica a mitad de la celda. En la figura 4.4 se muestra el esquema de esta situación para el modelo d2Q19.

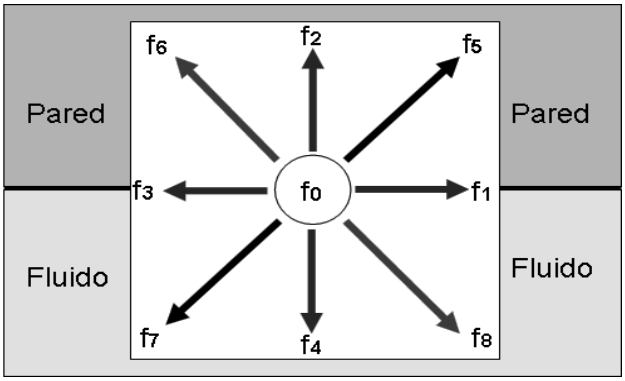


Figura 4.4: Condiciones de contorno on-grid bounce back.

En este caso, las distribuciones que deben calcularse son f_4, f_7 y f_8. La opción más sencilla fue propuesta por Ziegler [48], que consiste en revertir directamente hacia el fluido los valores de las distribuciones entrantes (f_2, f_5 y f_6 en el ejemplo). Sin embargo, con este esquema sólo se consigue una precisión de primer orden. Por esto se implementó la versión modificada propuesta por Zou y He [49] donde se llega a precisión de segundo orden. En el ejemplo, la velocidad debe ser cero tanto en x como en y para lo cual, por la ecuación (4.9) se debe cumplir:

$$\rho u_x = f_1 + f_5 + f_8 - f_3 - f_6 - f_7 = 0 \tag{4.11}$$

$$\rho u_y = f_2 + f_5 + f_6 - f_4 - f_7 - f_8 = 0 \tag{4.12}$$

La ecuación (4.8) completa el sistema para obtener los valores de las distribuciones faltantes de la figura 4.4. Entonces las fs desconocidas para un nodo en la pared superior del modelo d2Q19 se definen de la siguiente forma:

$$f_4 = f_2, \qquad f_7 = f_5 + \frac{1}{2}(f_1 - f_3), \qquad f_8 = f_6 - \frac{1}{2}(f_1 - f_3) \tag{4.13}$$

Otros tipos de condiciones de contorno utilizadas implican definir alguna condición de velocidad o presión impuesta sobre el límite del dominio que se utiliza como condición de entrada-salida. Por ejemplo, para simular el flujo a través de un canal en dos dimensiones se aplicó el esquema *Velocity-Pressure Boundary* (VPB) utilizado por Zou y He [49] donde se define la velocidad a la entrada del dominio y la presión a la salida. Se hizo una modificación para forzar un perfil parabólico en la entrada. Para que el flujo se mueva en la dirección positiva de x, en la entrada del canal

se calcula primero la densidad en función de las distribuciones y la velocidad impuesta de la siguiente manera:

$$\rho = \frac{f_0 + f_3 + f_7 + 2(f_5 + f_4 + f_6)}{1 - u_x} \tag{4.14}$$

Luego se obtienen las distribuciones faltantes en función de las existentes, la velocidad impuesta, y las ecuaciones (4.8) y (4.14):

$$f_1 = f_5 + \frac{2}{3}\rho u_x \tag{4.15}$$

$$f_2 = f_6 + \frac{1}{6}\rho u_x - \frac{1}{2}(f_3 - f_7) \tag{4.16}$$

$$f_8 = f_4 + \frac{1}{6}\rho u_x + \frac{1}{2}(f_3 - f_7) \tag{4.17}$$

A la salida del canal se fija la presión, que como es proporcional a la densidad equivale a fijar ρ constante. Además, se asume que la componente en la dirección y de la velocidad (transversal al canal) es 0. Entonces, para una salida en la dirección positiva de las x queda se obtiene u_x en función de ρ con la ecuación (4.14):

$$\rho = \frac{f_0 + f_3 + f_7 + 2(f_5 + f_4 + f_6)}{1 - u_x} \tag{4.18}$$

$$f_5 = f_1 - \frac{2}{3}\rho u_x \tag{4.19}$$

$$f_4 = f_8 - \frac{1}{6}\rho u_x - \frac{1}{2}(f_3 - f_7) \tag{4.20}$$

$$f_6 = f_2 - \frac{1}{6}\rho u_x + \frac{1}{2}(f_3 - f_7) \tag{4.21}$$

De manera análoga se aplicaron estas condiciones al esquema de tres dimensiones d3Q19. Originalmente presentadas por Maier, Bernard y Grunau [50], extendidas a segundo orden en dos dimensiones por Zou y He [49] y luego generalizadas al modelo específico d3Q19 en el trabajo de Hecht y Harting [51], este esquema permite combinar condiciones de contorno del tipo "*on grid bounce back*" con condiciones externas impuestas de presión y/o velocidad.

Por ejemplo, en la superficie superior central de una grilla cúbica, las distribuciones faltantes son f_4, f_9, f_{10}, f_{14}, y f_{18} (ver figura 4.2), y si se pretende imponer una condición de velocidad u_x paralela a esta superficie, sabemos que:

$$\rho = f_0 + f_1 + f_3 + f_6 + f_5 + f_{15} + f_{11} + f_{17} + f_{13} + 2\left(f_2 + f_7 + f_8 + f_{16} + f_{12}\right) \tag{4.22}$$

Entonces, las f_i faltantes se calculan de la siguiente manera:

$$f_4 = f_2 \tag{4.23}$$

$$f_9 = f_7 - \frac{1}{2}\rho u_x + \frac{1}{2}\left[\left(f_1 + f_{15} + f_{11}\right) - \left(f_3 + f_{17} + f_{13}\right)\right] \tag{4.24}$$

$$f_{10} = f_8 + \frac{1}{2}\rho u_x - \frac{1}{2}\left[\left(f_1 + f_{15} + f_{11}\right) - \left(f_3 + f_{17} + f_{13}\right)\right] \tag{4.25}$$

$$f_{14} = f_{16} + \frac{1}{2}\left[\left(f_{17} + f_6 + f_{15}\right) - \left(f_{13} + f_{11} + f_5\right)\right] \tag{4.26}$$

$$f_{18} = f_{12} - \frac{1}{2}\left[\left(f_2 + f_7 + f_8\right) - \left(f_4 + f_{10} + f_9\right)\right] \tag{4.27}$$

4.6. Parámetros del modelo

Un punto clave en implementaciones LBM para la simulación de fluidos es la elección cuidadosa de los parámetros numéricos, especialmente en problemas dependientes del tiempo, para poder reproducir buenas aproximaciones al fenómeno físico y predecir correctamente los diferentes regímenes de flujo.

Trabajos que caracterizan hasta cierto punto las variables principales de LBM pueden encontrase en las referencias [52], [53] y [54]. Particularmente, en el trabajo de Artoli [52] se desarrolla un estudio completo del impacto de los parámetros numéricos y su influencia en la solución final. Mostrando que si se pone a punto correctamente, el método de Lattice Boltzmann es capaz de simular problemas físicos correctamente.

Estos estudios, coinciden en que para una misma discretización espacial dada y un mismo número de Reynolds se consigue mayor precisión manteniendo lo más pequeña posible la velocidad máxima y reduciendo la viscosidad mediante el parámetro de relajación τ. Un ejemplo de estas relaciones paramétricas puede verse en el apéndice II.

Capítulo 5: Lattice Boltzmann 2D en CUDA

A continuación se presenta una implementación en CUDA del modelo de Lattice Boltzmann para la simulación de flujo en dos dimensiones. En particular, se analiza la utilización de LBM en CUDA para la simulación del flujo en un canal con una expansión súbita.

El código desarrollado sigue el esquema del presentado en el capítulo 3, que consta de una aplicación principal en lenguaje C que crea e inicializa los datos de entrada y realiza invocaciones a dos versiones equivalentes del algoritmo LBM. Una versión de ejecución paralela en CUDA que se denomina *kernel* y otra secuencial en C utilizada para comparaciones de tiempos de ejecución. En este caso se contó con una placa GeForce 8800 GT de la misma generación que la 8600 pero de mejor performance mientras que la versión del SDK es la 2.0.

5.1. Estructuras de datos

Para la representación interna de los datos se utilizaron 18 arreglos de punto flotante para almacenar la información de las 9 funciones de distribución f_i en dos pasos de tiempo (t y $t+1$). Estos arreglos unidimensionales son creados dentro de un programa C y luego se copian en la memoria del dispositivo antes de realizar el llamado al *kernel*. Con estos arreglos se representa una grilla rectangular de relación 1 a 3 entre altura y ancho respectivamente. El tamaño definido para la primera simulación es de 32×96 celdas, ya que por cuestiones de performance es recomendable utilizar múltiplos de 16 en las estructuras principales [55]. Al utilizar las condiciones de contorno descritas en el capítulo anterior denominadas *on-grid bounce-back,* las paredes del canal quedan ubicadas en el centro de la celda, con lo cual un canal recto con dos paredes tiene como altura total una celda menos que las utilizadas para su representación interna (figura 5.1).

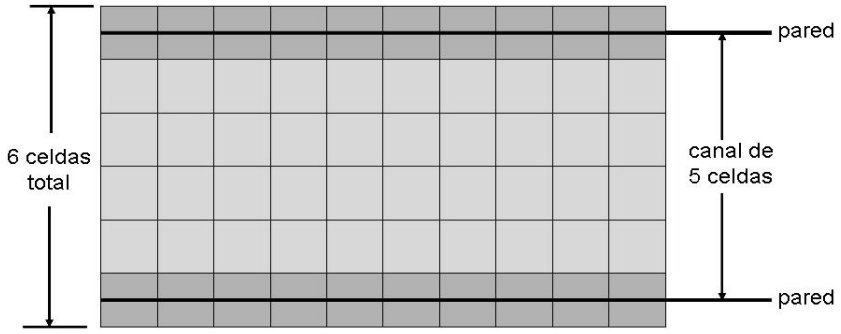


Figura 5.1: Ejemplo de canal de 5 celdas representado con 6 utilizando OGBB.

Para definir la geometría, se utiliza un arreglo de enteros, de las mismas dimensiones que las grillas de *fs* donde para cada celda se especifica su tipo con un valor entero permitiendo cambiar geometrías sin variar el resto del código. Para el problema particular de un canal con una expansión súbita en el primer tercio, se utilizaron 12 tipos de celdas diferentes:

1. Flujo: Celdas internas al canal, advección normal.
2. *Inlet*: Celdas de la pared izquierda en la entrada del canal donde se aplica la condición de flujo de entrada con velocidad forzada.
3. *Outlet*: Celdas de la pared derecha donde se aplica las condiciones de salida de presión constante.
4. Borde Superior: Bounce back hacia abajo con velocidad cero en x.
5. Borde inferior: Bounce back hacia arriba con velocidad cero en x.
6. Borde izquierdo: Bounce back hacia la derecha con velocidad cero en y.
7. Borde superior izquierdo del canal: condiciones de entrada y bounce back combinadas, la velocidad en x es cero.
8. Borde inferior izquierdo del canal: condiciones de entrada y bounce back (equivalente a 7).
9. Borde superior derecho: condiciones de salida y bounce back (velocidad cero en x).
10. Borde inferior derecho: condiciones de salida y bounce back.
11. Vértice de la expansión: Bounce back hacia arriba y hacia la derecha (velocidad cero en x y en y).
12. No se opera, celdas fuera del dominio.

5.2. Algoritmo

El método de Lattice Boltzmann para el modelo BGK de 2 dimensiones y nueve velocidades presentado en el capítulo 4 consiste básicamente en dos ciclos anidados a lo largo de las dos dimensiones de la grilla. Dentro de esos ciclos se calcula para cada celda el paso de advección, se aplican las condiciones de contorno particulares a esa celda y, por último, se calcula la regla de colisión y relajación. Al concentrar todos los pasos del algoritmo como uno solo, se reduce substancialmente la cantidad de datos transferidos desde y hacia la memoria principal. En la implementación CUDA, estos ciclos anidados son reemplazados por múltiples *threads* paralelos que ejecutan el código de una sola celda llegando al máximo nivel de paralelización posible.

Por otro lado, el uso de dos copias de los datos, una para el tiempo t y otra para $t+1$, facilita la paralelización eliminando la dependencia de datos. Los datos se leen de una copia y se escriben en otra durante un paso del algoritmo completo, análogamente a como se realiza en el modelo AQUA presentado en el capítulo 3. Sin embargo, a diferencia del modelo AQUA, en LBM no existen problemas de concurrencia ya que cada celda lee y actualiza valores diferentes a las demás. De esta manera, no es necesario utilizar sincronizaciones ni esquemas de dos pasadas, al menos en lo que respecta al control de accesos concurrentes.

5.3. Optimizaciones

Con lo mencionado hasta este punto ya se logra una implementación válida de LBM en CUDA con una mejora de performance similar a la presentada en el capítulo 3, entre 10 y 20 veces mayor a una implementación equivalente en C++. Pero para sacar el mayor provecho posible del hardware de la GPU es necesario profundizar en la estructura interna de la placa gráfica y el nuevo paradigma de programación, identificando las operaciones más costosas, tanto de cálculo como de acceso a memoria, para buscar alternativas más eficientes.

5.3.1. Acceso a memoria global

Los *threads* de CUDA pueden acceder a datos ubicados en diferentes espacios de memoria durante su ejecución, cada *thread* tiene su memoria local privada (registros),

cada bloque de *threads* tiene su memoria compartida (o *shared*) visible a todos los *threads* del bloque y, finalmente, todos los *threads* pueden acceder a la misma memoria global o *device memory*. La principal diferencia desde el punto de vista de performance es que demora aproximadamente 4 ciclos de reloj acceder a un dato en memoria compartida, mientras que la latencia de leer un dato de memoria global es de entre 400 y 600 ciclos. Como los datos se almacenan originalmente en la memoria global, la estrategia de acceso tiene un impacto fundamental en el tiempo de ejecución total del algoritmo. A continuación se detallan las estrategias principales utilizadas en este trabajo para reducir el impacto del costo de acceso a memoria global.

El *scheduler* de *threads* puede esconder gran parte de la demora de acceso a memoria global. Entonces, si hay suficientes instrucciones aritméticas independientes que puedan realizarse durante ese lapso, en el llamado al *kernel* se pueden lanzar varios bloques de *threads* por cada multiprocesador para que, cuando un bloque queda en espera de datos, otro tome su lugar en el multiprocesador.

Para el código LBM implementado en este trabajo, la actualización de una celda se realizó de manera tal de minimizar los accesos a memoria global, copiando estos valores a memoria compartida al inicio del cada paso para su posterior procesamiento, y al finalizar los que han sido modificados se almacenan en memoria global.

En una arquitectura clásica de CPU con memoria caché es más eficiente de recorrer una matriz almacenada por filas, ya que al leer el primer dato a memoria los subsiguientes son copiados a la memoria caché, acelerando las lecturas siguientes como lo muestra el esquema de la figura 5.2

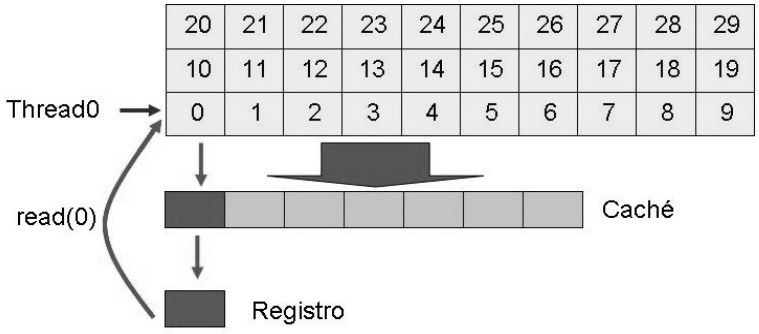


Figura 5.2: Esquema de recorrido por filas de una grilla en una arquitectura con memoria caché.

En una GPU CUDA no existe memoria caché, pero sí hay accesos agrupados a memoria compartida. Los *threads* paralelos deben recorrer la estructura por columnas para beneficiarse de esta característica como se muestra en el esquema de la figura 5.3.

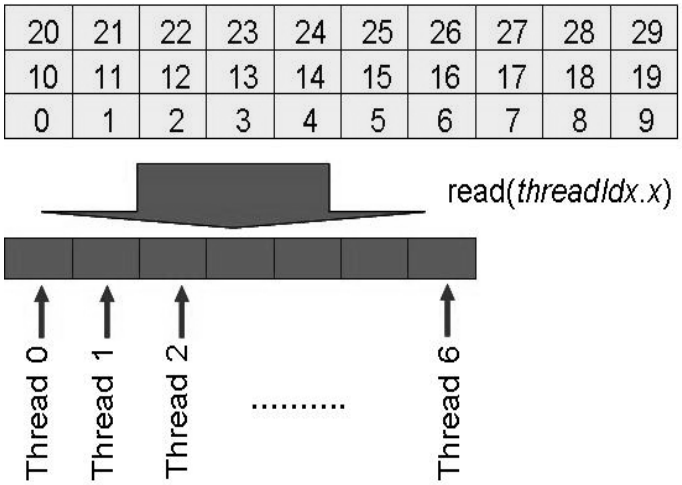


Figura 5.3: Esquema de accesos alineados a una grilla en una arquitectura de GPU CUDA.

Al asignar celdas consecutivas en la dirección *x* a un bloque de *threads* paralelos, los accesos a memoria de éstos se realizan a elementos almacenados consecutivamente en memoria global. Es decir, el *thread x* lee la dirección *base+x*, el *thread x+1* lee la dirección *base+x+1* y así sucesivamente. Para las placas con compatibilidad 1.1 como la utilizada, si la dirección *base* es múltiplo del tamaño de dato y todos los *threads* leen un dato, los accesos a memoria principal de la placa gráfica de cada grupo de 16 *threads* (denominado *half warp* en CUDA) se deben realizar como uno solo y no como una secuencia de 16 accesos [55]. Esto es lo que se denomina accesos agrupados a memoria o "*coallesced memory access"*.

Como el costo de lectura en memoria global del dispositivo es inferior al de escritura, es preferible maximizar la agrupación de accesos a memoria para escrituras. Esto se logra al aplicar el esquema *"pull",* donde se realiza la lectura en el paso de advección que es el que genera los posibles accesos a memoria no-alineados ya que cada *thread* debe acceder a datos almacenados en celdas vecinas a la que procesa. En el trabajo de Tölke [29] se utiliza un esquema de dos pasadas para forzar escrituras y lecturas agrupadas en todos los casos, pero a costa de dos llamadas *kernels* y dos

recorridos de la estructura por cada paso de tiempo. Esta solución fue implementada pero se descartó por tener menor performance global.

Los pasos del algoritmo LBM en CUDA resultantes son los siguientes:

1. Sincronizar *threads*.
2. Leer las funciones de distribución de las celdas adyacentes en memoria global, i.e. $f_i(x-e\delta t,\ t-\delta t)$ a memoria compartida. Esto se hace de manera agrupada solamente para las distribuciones centro, norte y sur (0, 2 y 4 en la figura 4.1).
3. Aplicar las condiciones de contorno para obtener las *fi* faltantes.
4. Calcular ρ, u y $f_i^{(eq)}$.
5. Sincronizar *threads*.
6. Escribir los valores actualizados en la celda actual, i.e. $f_i(x,t)$ a memoria global. Estos accesos son alineados en todos los casos.

5.3.2. Otras optimizaciones

Se aplicaron otras optimizaciones al código, como la minimización de parámetros transferidos declarándolos como constantes internas, lo cual reduce la cantidad de accesos a memoria global y el uso de registros. Algunos bucles internos, como los que recorren las 9 distribuciones de una misma celda, se escriben de manera secuencial, ya que se conoce de antemano los límites, eliminando de esta manera el espacio de memoria utilizado por los índices y los cálculos de límite e incremento.

También se eliminaron todas las bifurcaciones posibles ya que en un código CUDA, los *threads* paralelos de cada grupo de 16 recorren de manera secuencial las diferentes bifurcaciones del código donde entra alguno de los 16 *threads,* lo cual aumenta significativamente la cantidad de operaciones aritméticas. De todos modos, algunas de estas bifurcaciones no se pueden evitar (en las condiciones de contorno), pero en LBM se conoce cuando ocurren. Para el problema puntual de la expansión súbita, el grupo de *threads* con más condiciones de contorno diferentes es el que contiene el vértice de la expansión, con 3 tipos de celdas diferentes (borde inferior, vértice y fluido).

5.4. Simulación de expansión súbita

La validación del modelo LBM CUDA d2Q9 se hizo mediante la simulación del flujo en un canal de dos dimensiones con una expansión súbita en el primer tercio como el presentado la figura 5.4.

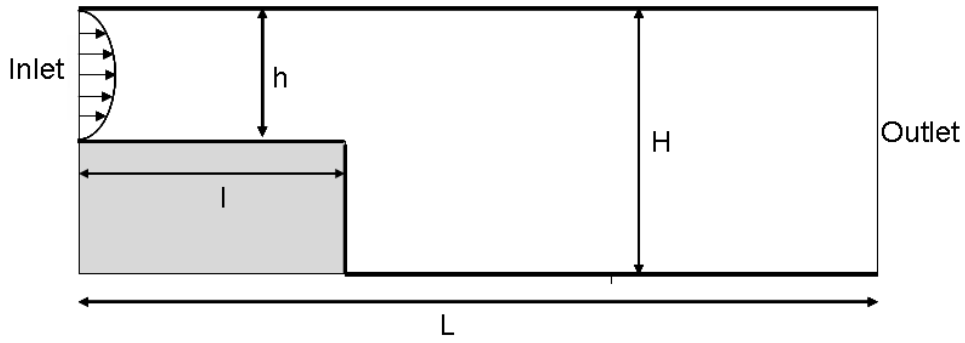


Figura 5.4: Canal con expansión súbita en el primer tercio.

La geometría elegida fue la siguiente: La longitud total del canal (L) es de 3, el ancho del canal es de 0,5 en la entrada (h) y de 1 en la salida (H), con lo cual la expansión ubicada a distancia 1 de la entrada (l) es de 0,5. El flujo de entrada se define como un perfil parabólico con velocidad adimensional máxima central u_x de 0,3, y las condiciones en la salida son de presión constante y velocidad vertical u_y = 0. Las propiedades adimensionales del fluido simulado son ρ = 1,0 y μ = 0,003, con lo cual la viscosidad cinemática adimensional es ν = 0,003. Entonces, si suponemos una profundidad (z) muy grande con respecto al alto del canal de entrada, el diámetro hidráulico de la entrada se calcula como [56]:

$$Dh = \frac{4A}{P} = \frac{4hz}{2h+2z} \cong \frac{4hz}{2z} = 2h \tag{5.1}$$

y el número de Reynolds para ese diámetro hidráulico es:

$$\mathrm{Re}_D = \frac{U_{\max} Dh}{\nu} = 100 \tag{5.2}$$

Por lo tanto la relación de escala final máxima posible para las dimensiones iniciales elegidas en el modelo de LBM es δ = 1/30, lo cual corresponde a un canal de entrada de 15 celdas. Si tomáramos un canal de entrada de 16 celdas (relación 1/32),

serían necesarias 33 celdas para representar el canal de salida, por lo comentado en la sección 5.1 acerca de las condiciones de contorno.

5.4.1. Comparación con método de elementos finitos

Para contrastar los valores obtenidos se realizó una simulación con un programa que resuelve las ecuaciones de Navier-Stokes con un modelo de elementos finitos *equal-order* con igual interpolación para velocidad y presión, y estabilización *sub-grid scale* (SGS). Este programa está linealizado por atraso del término convectivo, por lo tanto resuelve un estacionario mediante un pseudo-transitorio. Para que las simulaciones fueran lo más equivalentes posibles se utilizó una malla regular de 1821 nodos, con la diferencia de que estos nodos forman elementos triangulares. Las condiciones de contorno son las mismas y se fijaron sobre 137 elementos.

Se simuló un transitorio de 100 segundos del problema físico en ambos modelos (FE y LBM) partiendo del estado estacionario de velocidad cero en todas las celdas. Por tratarse de diferentes enfoques, LBM requirió 3000 iteraciones mientras que sólo se ejecutaron 100 pasos del modelo FE.

5.4.1.1. Campos de velocidades

A continuación se muestran diferentes visualizaciones del campo de velocidades para cada uno de los modelos al finalizar las simulaciones. Estas vistas representan un análisis cualitativo ya que los valores absolutos son levemente diferentes.

En la primera imagen (figura 5.5) se aprecia claramente la concordancia de ambos modelos en cuanto a los perfiles de velocidades u_x. La velocidad máxima del centro se reduce al pasar la expansión súbita (va de amarillo a rojo) y se dirige hacia el centro del canal.

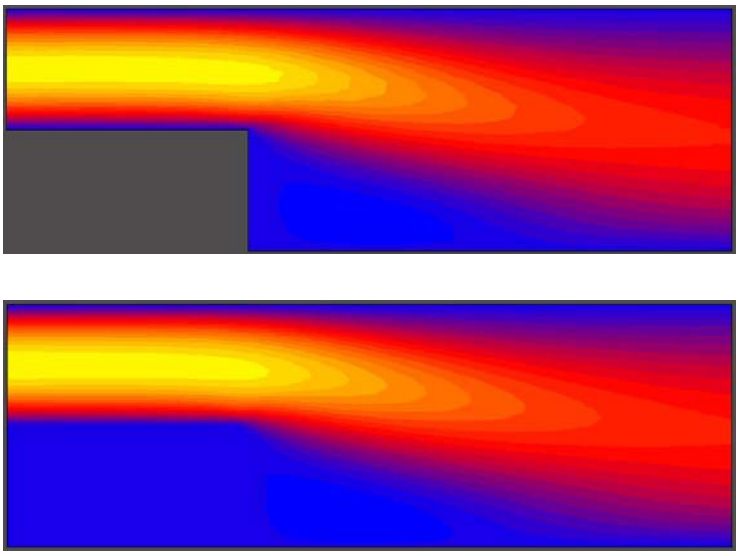


Figura 5.5: Comparación de la velocidad en x (u_x) por gráfico de zonas de velocidad. La versión de Elementos Finitos es la imagen superior, la de Lattice Boltzmann en GPU la inferior.

En la figura 5.6 pude verse como la velocidad en *y* (u_y) es prácticamente nula en la entrada del canal y sobre la salida en ambos modelos (color naranja). El flujo alcanza un máximo positivo después de pasar por la expansión correspondiente al vórtice (amarillo) y un máximo negativo en el centro del canal (color azul). Se observa concordancia entre ambos modelos.

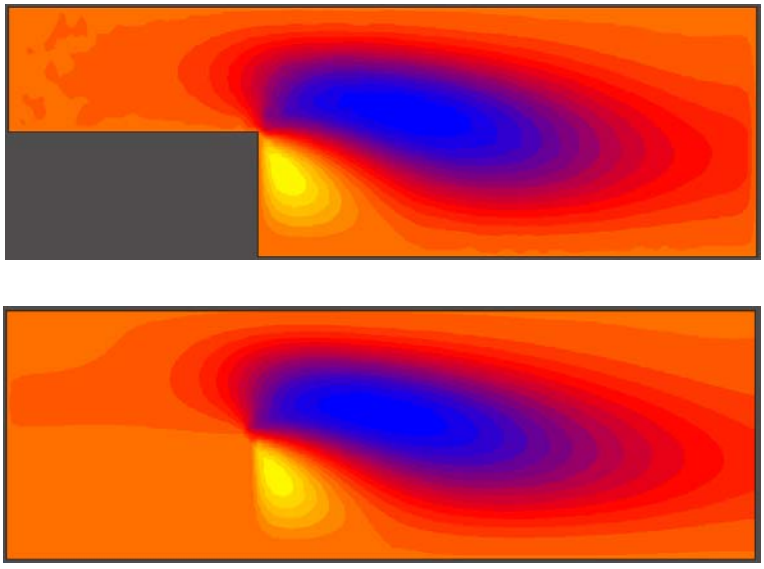


Figura 5.6: Comparación de la velocidad en y (u_y) por gráfico de zonas de velocidad. La versión de Elementos Finitos es la imagen superior, la de Lattice Boltzmann en GPU la inferior.

En la última comparación (figura 5.7) se muestra el campo de velocidades en ambos casos. Las líneas de corriente (azules) describen patrones similares y en ambos casos se observa el vórtice que se genera en la esquina inferior del la expansión.

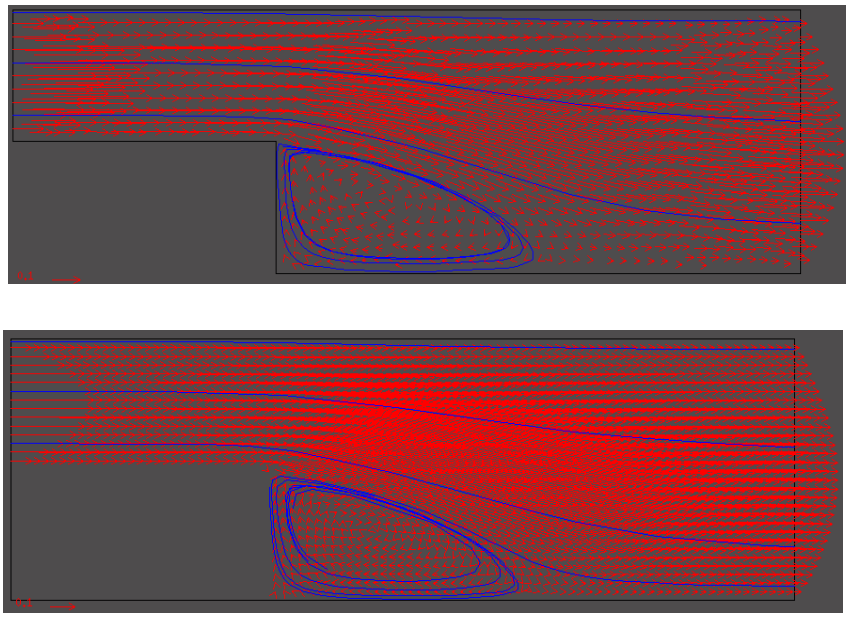


Figura 5.7: Comparación de los campos de velocidades y líneas de corriente. FE arriba, LBM-GPU debajo.

5.4.1.2. Perfiles de velocidades

Para evaluar cuantitativamente los resultados, se midieron los perfiles de velocidad en diferentes posiciones a lo largo del canal. A continuación se presentan los gráficos correspondientes a estos perfiles de velocidad u_x normalizados superpuestos.

En la figura 5.8 se observa el perfil de velocidad inmediatamente después de la expansión (donde conserva el perfil impuesto) y en el centro del canal donde la velocidad se vuelve levemente negativa entre y=0 e y=0,2. Este retroceso del fluido produce el vórtice señalado anteriormente.

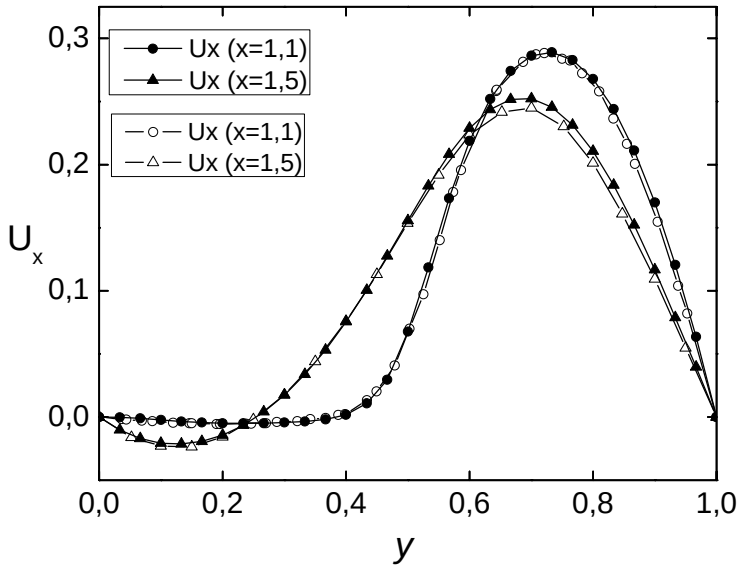


Figura 5.8: Perfil de velocidad u_x normalizado a lo largo de un corte vertical del canal inmediatamente después de la expansión (x = 1,1, círculos) y en el centro del canal (x = 1,5, triángulos). Comparación entre ambas simulaciones. LBM (símbolos llenos), FE (símbolos vacíos).

5.5. Resultados de performance

La unidad más utilizada en la medición de performance de modelos LBM es el número de celdas actualizadas en un segundo o LUPS (*Lattice-Site Updates per Second*) [57]. En la tabla 5.1 se muestran los diferentes resultados de performance obtenidos en este trabajo. Se detallan, el tamaño total de la grilla, el tiempo promedio de una iteración para toda la grilla, y los correspondientes MLUPS aproximados para la versión en C++ y para la versión CUDA con la mejor configuración de bloques y *threads* testeada en cada caso. La última columna muestra la aceleración del cálculo obtenida utilizando la tecnología de GPU. La versión para CPU es estrictamente secuencial como la implementada en el capítulo tres y sirve de punto de partida para evaluar la performance del código CUDA.

Tamaño de la Grilla	Versión C++		Implementación en CUDA			Speedup
	Tiempo de CPU (ms.)	MLUPS CPU	Bloques y threads	Tiempo de GPU (ms.)	MLUPS GPU	
96×32 3072	0.59	5.02	1 × 32 96	0.041	75	14.4
192×64 12288	2.1	5.85	3 × 64 64	0.085	144	24.7
384×128 49152	13.47	3.65	6 × 128 64	0.310	158	43.45
768×256 196608	54.87	3.58	6 × 256 128	1.29	152	42.53
1536×512 786432	215.52	3.64	6 × 512 256	18.23	43	11.82

Tabla 5.1. Tiempos de ejecución para el modelo de Lattice Boltzmann 2D.

En la figura 5.9 se grafican los tiempos de ejecución para cada modelo en función del número de celdas y la línea de límite para simulaciones en tiempo real.

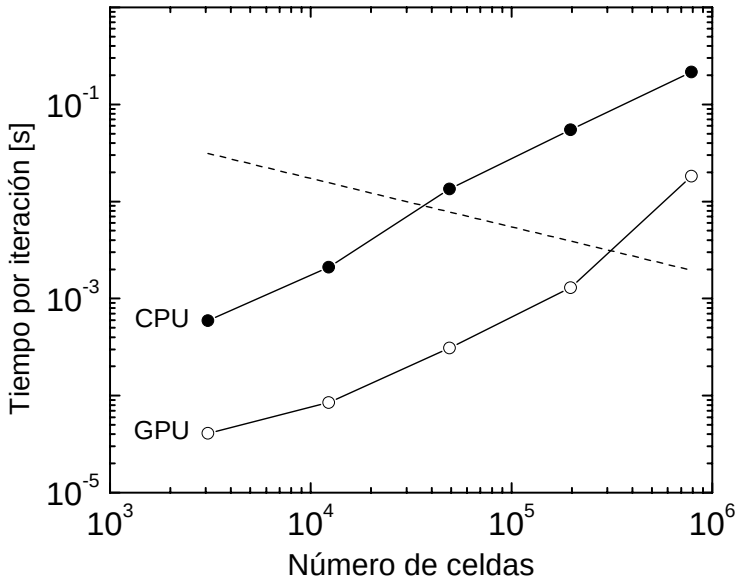


Figura 5.9: Comparación de los tiempos de ejecución en función del número de celdas. La línea punteada es el tiempo real.

El modelo LBM en CUDA y el de elementos finitos fueron también ejecutados utilizando el comando *time* para calcular el tiempo total insumido por la aplicación en una corrida. Lo que se pretende mostrar es que el tiempo total que nos insume calcular una solución de un problema determinado utilizando esta nueva tecnología es viable desde el punto de vista del usuario final, y que está dentro de los márgenes actuales de una simulación CFD. El código LBM presentado fue desarrollado y optimizado para la simulación en 2D, mientras que el solver de FE es una compleja herramienta de simulación con áreas de aplicación mucho más amplias.

	FE – CPU	**LBM – CPU**	**LBM – GPU**
Tiempo Total [s]	42,39	1,77	0,123

Tabla 5.2: Tiempos totales de las simulaciones en los diferentes modelos.

Estos tiempos de ejecución corresponden a máquinas PC de escritorio con procesador Intel Core2 Quad de 2.4 GHz y 8 GB de memoria RAM con sistema operativo GNU/Linux Debian Lenny.

5.6. Implementación en PETSc

La tendencia de los procesadores actuales (CPUs) es aumentar el número de núcleos por procesador en lugar de su velocidad de cálculo, ya que ésta ha llegado prácticamente a su límite [19]. Es así que la capacidad de procesamiento teórica de estos procesadores multi-núcleo generalmente se expresa como la capacidad individual de cada uno de los procesadores multiplicada por la cantidad de núcleos. Pero cuando corremos un código C++ secuencial con un solo hilo de ejecución, como es el utilizado para comparación en este trabajo, se utiliza solamente uno de los núcleos.

Para intentar una comparación de performance más correcta, se implementó una versión paralela para CPU que aproveche completamente el potencial de las arquitecturas de CPU multi-núcleo. Para ello se utilizó una plataforma conocida originalmente para procesamiento distribuido en varias terminales basado en el concepto de pasaje de mensajes, denominada *Parallel Extensible Toolkit for Scientific Computations* (PETSc) [58], desarrollada por el Mathematics and Computer Science Division del Argonne National Laboratory. Este paquete, escrito en C++ bajo el

paradigma orientado a objetos y basado en las librerías *Message Passing Interface* (MPI) abstrae los problemas de comunicación entre procesos utilizando objetos distribuidos de alto nivel. El entorno de desarrollo permite la creación de aplicaciones que funcionen en forma distribuida utilizando MPI sin necesidad de especificar la comunicación entre los procesos.

5.6.1. Representación de los datos

Como se trata de una grilla regular, para la representación de los datos se usaron arreglos dinámicos (*DA*). Estos arreglos se utilizan en conjunto con vectores PETSc para representar grillas regulares que requieren comunicación de datos no locales para realizar cálculos, como en el caso modelos de diferencias finitas o autómatas celulares. Los *DA* se encargan de la comunicación y de definir cómo se distribuye la información entre los procesos, pero no de almacenar los datos. Los valores de las f_i se almacenan en Vectores.

Las grillas que representan las distribuciones de partículas f_s en t y $t+1$ denominadas *Old* y *New* son de 3 dimensiones, las 2 dimensiones espaciales de la grilla y una tercera para las 9 f_i. Las matrices que almacenan los valores de ρ, u, w y la geometría son grillas de dos dimensiones. Ambos arreglos se crean con el modo *STENCIL_STAR,* ya que el dominio se va a partir sólo verticalmente. Todos los vectores almacenan variables de tipo *PetscScalar*. En el caso del *tipo* representa valores enteros.

Todas las estructuras de datos tienen una versión global, con el tamaño total del dominio y cada proceso tiene una copia local con la porción del dominio que le corresponde procesar. A su vez, las copias locales tienen celdas fantasma o "*ghost*" que representan la interfaz con el proceso vecino y es donde se almacenan datos correspondientes a porciones del domino que son ejecutadas por otros procesos. Estos valores son modificados en el paso de advección de LBM.

5.6.2. Inicialización

Las condiciones de contorno se implementan del mismo modo que la versión CUDA. La definición de la geometría se hace de manera secuencial ya que el costo computacional es mínimo comparado con la simulación y el código puede hacerse independiente de la forma en que se divide el dominio.

La función encargada de crear la geometría se llama solamente desde el proceso de rango 0 (o proceso principal). El método crea un arreglo de una dimensión de *PetscScalar* en el que se cargan los diferentes tipos de celdas, siguiendo el ordenamiento por filas clásico de una aplicación secuencial C++. Para cargar los valores del arreglo al vector global de la geometría se utiliza la función *VecSetValues()*, pero como el ordenamiento de los datos en el vector distribuido es diferente al ordenamiento natural se debe crear un objeto *Application_Ordering* (*AO*). Entonces, para cargar el vector se le da al vector de índices el ordenamiento de PETSc mediante el *AO* creado. Luego, para actualizar la información local almacenada en el vector global, todos los procesos ejecutan los llamados a las funciones de la figura 5.10.

```
VecAssemblyBegin(gt);
VecAssemblyEnd(gt);
DAGlobalToLocalBegin(dat,gt,INSERT_VALUES,lt);
DAGlobalToLocalEnd(dat,gt,INSERT_VALUES,lt);
```

Figura 5.10: Código de actualización del vector local.

El otro paso de la inicialización consiste en definir el estado inicial de las f_i. Como se pretende iniciar la simulación desde un estado estacionario, se realiza el cálculo de las f_i^{eq} en cada una de las celdas partiendo de una velocidad $u_o = (0,0)$ y de una densidad $\rho = 1$. Este paso se puede ejecutar en paralelo ya que consiste solamente del cálculo de la regla de colisión.

5.6.3. Procesamiento distribuido

El algoritmo principal del modelo se ejecuta de manera distribuida sobre los vectores locales a cada proceso, tanto de f_i como de ρ, w y *tipo*. La función que realiza el cálculo *executeBGK()* se llama alternativamente con un vector de f_i como origen y el otro como destino. Antes de invocar *executeBGK* se deben copiar los valores de los vectores globales a los locales con el comando de 2 pasos *DAGlobalToLocal*. Este comando actualiza los valores de las celdas *ghost* de los vectores locales con las modificaciones realizadas por los demás procesos. Después del llamado a *executeBGK*, se invoca a la función inversa *DALocalToGlobal* para volcar los datos actualizados locales a los vectores globales. Esta función se realiza en un solo paso ya que la actualización es totalmente local. Cada proceso actualiza la parte del vector global que contiene. Por lo tanto no requiere de comunicación entre procesos.

5.6.4. Algoritmo principal

La función *executeBGK()* recibe como parámetros los vectores locales, los DAs y parámetros del modelo (τ, e, w). Para acceder a estos vectores en forma de arreglos bidimensionales, se utiliza el comando *DAVecGetArray*.

Inicialmente se obtienen los índices iniciales y los tamaños de los vectores locales mediante *DAGetCorners*. Como las grillas son todas equivalentes y los DAs se crean especificando la división entre procesos (no se utilizó *PETSC_DECIDE*), los límites en x e y son los mismos para todos los arreglos. En el caso de las distribuciones f_i el índice en z va siempre de 0 a 9, al igual que el índice de los arreglos de las direcciones e_i y el de los pesos w de la regla de colisión BGK. Los índices corresponden a la parte estrictamente local del vector, sin los valores de frontera o *ghosts*. Los arreglos se recorren de la siguiente manera:

```
for (y=ys; y<ys+ym; y++){
      for(x=xs; x<xs+xm; x++){
          //Codigo para cada celda
      }
}
```

Figura 5.11: Código de recorrido de las estructuras.

El primer cálculo es el paso de advección, que accede a las celdas vecinas (celdas *ghosts* cuando se procesa en el límite local de la grilla). En el caso de los límites de la grilla real los valores de *up* o *dn* son puestos en cero y se copia el valor de la misma celda. Este valor luego se recalcula al aplicar las condiciones de contorno correspondientes. Fuera del ciclo para cada celda y una vez calculadas todas las distribuciones, se liberan los arreglos con el comando *DAVecRestoreArray*.

5.6.5. Performance

Las pruebas se centraron en la posibilidad de utilizar un entorno PETSC para distribuir aplicaciones dentro de una sola PC con procesadores de múltiples núcleos. En este caso particular se utilizó un procesador AMD Athlon 64 X2 Dual Core de 2,41 GHz. En la tabla 5.3 se muestran los tiempos totales tomados con el comando *time* para un ciclo del modelo, según el tamaño de la grilla y la cantidad de procesos creados.

Tamaño de la Grilla	Versión C++ (ms.)	Implementación en PETSc (ms.)			
		1 Proceso	2 Procesos	3 Procesos	4 Procesos
96×32 3072	0.65	**4.63**	6.48	7.24	7.86
192×64 12288	3.42	17.92	21.61	20.16	**16.76**
384×128 49152	15.54	68.67	66.54	57.04	**51.90**
768×256 196608	61.46	287.29	233.83	210.49	**197.54**
1536×512 786432	222.21	1161.20	1149.34	**785.17**	836.45
1024×3072 3145728	1010.31	4768.77	6216.61	**3052.07**	4132.87

Tabla 5.3: Tiempos de ejecución para las diferentes configuraciones.

Los resultados muestran que al aumentar el tamaño de la grilla se obtienen tiempos de simulación menores en las versiones distribuidas en comparación con versiones PETSc de un solo proceso, salvo para pequeñas grillas donde la relación lado/área es grande y el costo de la comunicación es mayor que la mejora introducida por la distribución. En el mejor caso, los tiempos se reducen al 65% tratándose de una CPU con 2 núcleos. Al comparar los tiempos de ejecución con una implementación C++, vemos que la diferencia de tiempos de ejecución de la versión en PETSc es de aproximadamente 4 veces más lenta por el *overhead* generado al tratarse de una arquitectura de alto nivel.

Tecnologías de más bajo nivel como SMP podrían reducir los tiempos de la versión para CPU al tener un menor *overhead* y perimitir esquemas de memoria compartida similares a los utilizados en la implementación CUDA.

Los resultados también dependen de la geometría del canal simulado ya que la performance del código distribuido mejoraría para geometrías donde se minimicen las zonas de comunicación entre procesos, por ejemplo un canal donde el largo sea mucho mayor al ancho.

Capítulo 6: Lattice Boltzmann 3d en CUDA

En este capítulo, se presenta una implementación en CUDA del modelo de Lattice Boltzmann en tres dimensiones, en particular la versión d3Q19 con regla de colisión BGK. El problema simulado para la validación del esquema numérico es el flujo en una cavidad cúbica. El esquema del código es similar a los presentados en los capítulos 3 y 5, con una aplicación principal en lenguaje C, que crea e inicializa los datos de entrada y realiza invocaciones a sendas versiones CUDA y C del algoritmo. La GPU utilizada para esta implementación es una GeForce GTX 260 de la serie G200 con capacidad computacional 1.3 en conjunto con las versiones 2.3 y 3.1 de las herramientas de desarrollo.

6.1. Estructuras de datos

En 3D, al aumentar notablemente el volumen de datos a transferir con respecto a los modelos de dos dimensiones, los esquemas de acceso a memoria y la representación interna de los datos cobran aún más importancia en el diseño de un algoritmo eficiente. Existen estrategias básicas, como la utilización de una sola grilla con funciones macros de direccionamiento que han sido utilizadas exitosamente en entornos distribuidos de CPUs y son aplicables a GPU. Pero, dada la particular arquitectura de las placas gráficas, hay otros aspectos como el acceso a memoria, la secuencia de ejecución o la manera en que se recorren las estructuras, que deben implementarse de manera diferente para obtener una mejor performance.

La primera etapa de la implementación consiste en definir la representación interna que se dará a los datos (la forma en que se almacenan las distribuciones de partículas) ya que esto tiene un impacto substancial en la performance [38]. Para la versión 3D se utilizó un solo arreglo unidimensional y definiciones Macro de acceso con el objeto de poder modificar fácilmente la disposición de los datos y evaluar diferentes alternativas. La macro en cuestión utiliza 3 índices dimensionales *(x, y, z)* para referenciar la celda de la matriz tridimensional, un índice "*q*" que indica la distribución particular (de 0 a 18) y el parámetro "*t*" que sirve para alternar entre los dos pasos de tiempo. También recibe como parámetro las dimensiones totales de la grilla y la cantidad total de direcciones *Q*. La figura 6.1 muestra el código de la macro para la

disposición de almacenamiento utilizada. Con este esquema, los valores de una misma dirección e_i para todas las celdas del dominio quedan en posiciones consecutivas de memoria, moviéndose en la dirección x de la grilla tridimensional representada.

```
#define dirQ(x,y,z,q,t,DIMX,DIMY,DIMZ,Q)
   (x+(y)*DIMX+(z)*DIMX*DIMY+(q)*DIMX*DIMY*DIMZ+(k)*Q*DIMX*DIMY*DIMZ))
```

Figura 6.1: Código de direccionamiento para acceso a las estructuras.

6.2. Algoritmo

Análogamente a 2D, el algoritmo se implementa siguiendo un esquema "*pull*" donde para cada celda, primero se obtienen las f_i "trayendo" los valores desde celdas vecinas, se aplican las condiciones de contorno, se calcula el paso de colisión y por último el de relajación, minimizando así la cantidad de datos transferidos desde y hacia la memoria principal.

6.3. Accesos a memoria agrupados

En las GPU con compatibilidad computacional CUDA 1.2 o superior como la utilizada, los requisitos para que los accesos a memoria global del dispositivo sean alineados son mucho menores. Solamente debe cumplirse que los 16 *threads* paralelos accedan a datos que se encuentran en un espacio de memoria continuo de un tamaño dado por el tipo de dato accedido. La figura 6.2 muestra 2 casos donde los accesos son agrupados en uno solo y que no lo serían en placas de versiones anteriores. Si el espacio de memoria accedido no coincide con un segmento de memoria alineado los 16 accesos se agrupan como 2 como lo muestra la figura 6.3.

Utilizando este esquema, en la peor circunstancia, podría darse el caso en que los accesos se agrupen de a dos en lugar de a uno, pero nunca como los 16 accesos originales. Por lo tanto no es necesario realizar dos pasadas para implementar un LBM eficientemente. En implementaciones recientes como la de Kuznik, Obrecht, Rusaouën y Roux [59] se sigue la idea de Tölke [29] de dos pasadas para forzar lecturas y escrituras alineadas, pero con las placas de la generación 1.2 en adelante no es aconsejable ya que la cantidad de accesos no agrupados se reduce notablemente.

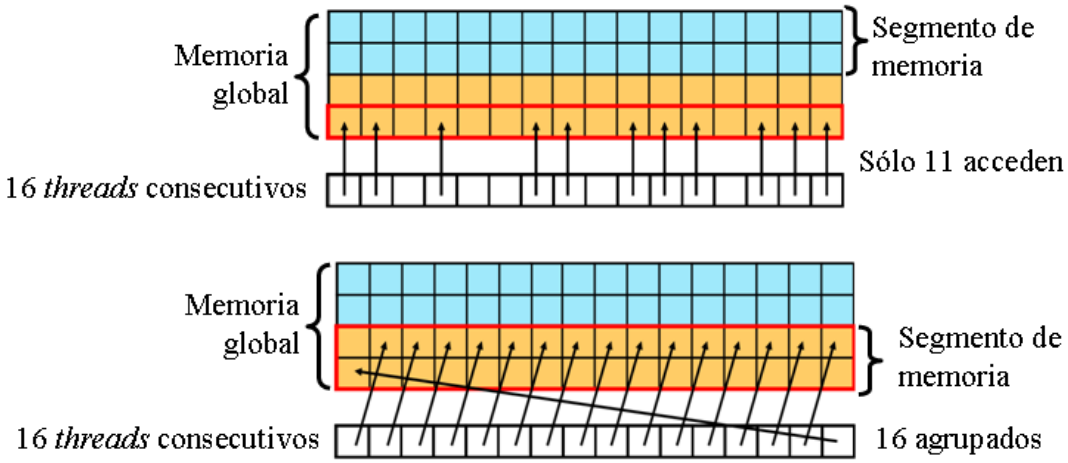


Figura 6.2: Ejemplos de 16 threads que acceden a memoria y que se realizan como uno solo.

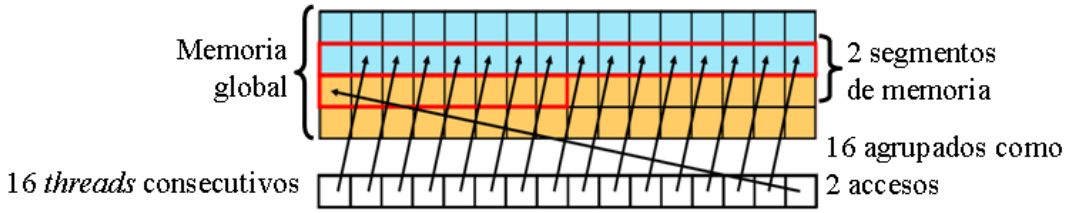


Figura 6.3: Ejemplo de16 accesos que se agrupan en 2 accesos a memoria.

6.4. Uso de memoria compartida

La estructura del algoritmo y la secuencia de pasos utilizada son equivalentes a las presentadas en el capítulo 5. Primero se copian los datos de memoria principal a memoria compartida advectando para luego trabajar sobre esta memoria.

6.5. Configuración de ejecución

El dominio se divide en bloques de *threads* donde los índices "*y*" y "*z*" son constantes y a cada índice "*x*" le corresponde un *thread*. Por lo tanto un *thread* ejecuta el código de una sola celda y el nivel de paralelismo alcanzado es total (figura 6.4).

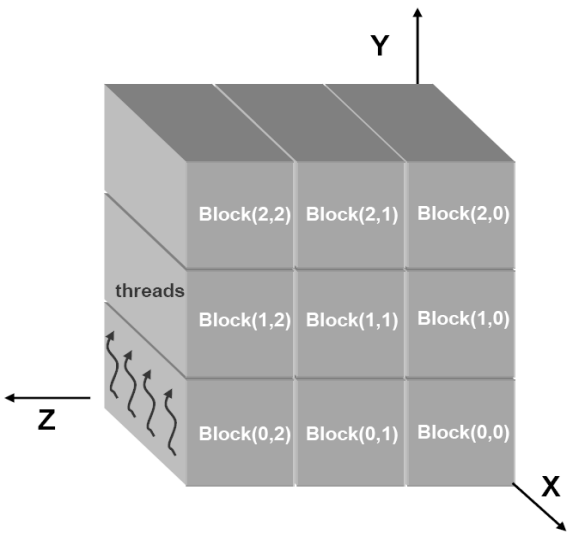


Figura 6.4: Esquema de división del dominio en bloques de threads.

Con esta configuración, el dominio no puede crecer en la dirección x más allá del límite de 512 *threads* por bloque. Sin embargo el límite se alcanza antes debido al tamaño total de memoria disponible que limita la estructura a menos de 180 celdas cúbicas. Para dominios con diferentes tamaños en x, y, z, es conveniente tomar x como el más pequeño ya que la limitación en cantidad de bloques es mayor.

6.6. Condiciones de contorno

Las condiciones de contorno implementadas son las descritas en el capítulo 4. Del mismo modo que en la versión de dos dimensiones, fue necesario definir diferentes tipos de celdas con su correspondiente cálculo de distribuciones restantes para poder simular el flujo en la cavidad cúbica. Los tipos de celdas son los siguientes:

1. Flujo celdas internas al cubo, advección normal.
2. Capa superior, velocidad constante u_{max}.
3. Pared trasera, $u_x = 0$, $u_y = 0$.
4. Pared derecha, $u_y = 0$, $u_z = 0$.
5. Pared frente, $u_x = 0$, $u_y = 0$.
6. Pared izquierda, $u_y = 0$, $u_z = 0$.
7. Piso, $u_x = 0$, $u_z = 0$.

8. Arista trasera derecha, $u_y = 0$.
9. Arista frontal derecha, $u_y = 0$.
10. Arista trasera izquierda, $u_y = 0$.
11. Arista frontal izquierda, $u_y = 0$.
12. Arista trasera inferior, $u_x = 0$.
13. Arista derecha inferior, $u_z = 0$.
14. Arista frontal inferior, $u_x = 0$.
15. Arista izquierda inferior, $u_z = 0$.
16. Vértice inferior trasero derecho (sin condiciones de velocidad).
17. Vértice inferior frontal derecho (sin condiciones de velocidad).
18. Vértice inferior frontal izquierdo (sin condiciones de velocidad).
19. Vértice inferior trasero izquierdo (sin condiciones de velocidad).

En el problema de la cavidad cúbica que se detalla más adelante en la sección 6.8, las condiciones de contorno están bien definidas desde el punto de vista físico pero no existe una forma clara de traducirlas a implementaciones de LBM. En particular, no está claro como definir las celdas de las aristas superiores del cubo. En el presente trabajo la condición de contorno en la capa superior con velocidad constante se implementó definiendo como capa superior toda la matriz de celdas superiores (con $x = DIMX-1$), sin considerar de manera diferente las aristas del cubo. Otra consideración particular a la implementación es en cuanto a las celdas ubicadas en las cuatro esquinas inferiores, en las que no se aplica ninguna condición de velocidad específica ya que estas celdas no tienen distribuciones de partículas que se dirijan hacia el fluido.

6.7. Otras optimizaciones

Además de reducir la cantidad de bifurcaciones del código es importante reducir el código interno de éstas, ya que se sigue de manera secuencial por los diferentes *threads*. Para el caso del paso de advección, se utiliza el mismo código para todos los *threads* y luego se aplican las condiciones particulares de presión o velocidad. Para las celdas ubicadas en los límites del cubo se utilizaron índices de desplazamiento que se ponen a cero si la celda pertenece a algún contorno. Para las bifurcaciones generadas por las condiciones de contorno de LBM no existe otra solución. Sin embargo, si se

analiza el dominio y la división de *threads* propuesta, en el peor de los casos, un grupo de 16 *threads* que procesan celdas vecinas consecutivas en *x* puede tener a lo sumo dos condiciones de contorno diferentes, y el código ejecutado en las diferentes condiciones de contorno es bastante simple.

La figura 6.5 muestra parte del código que inicializa estos índices y como se utilizan en el paso de advección.

```
west=1;
east=1;
…
if (x==0)//limite izquierdo
  west = 0;
if (x==DIMX-1) //limites derecho
  east=0;
…
f[b] =  F[dirQD(x,y,z,0,k,DIMX,DIMY,DIMZ,Q)];
f[b+1] =  F[dirQD(x-west,y,z,1,k,DIMX,DIMY,DIMZ,Q)];
…
f[b+18] =  F[dirQD(x,y+north,z-back,18,k,DIMX,DIMY,DIMZ,Q)];
```

Figura 6.5: Código para el tratamiento de los límites del dominio.

6.8. Simulación de cavidad cúbica

El problema estacionario de cavidad cúbica fue el elegido como caso de estudio. Por no existir solución analítica al problema, los resultados se comparan con otros encontrados en la literatura. El problema se describe como el escurrimiento estacionario de un fluido incompresible en una cavidad cúbica, compuesta de paredes impermeables rugosas en las que se impone un movimiento en la pared superior de la cavidad. Dicha pared se mueve con velocidad constante (*Umax*) en la dirección *x* mientras que las demás paredes permanecen inmóviles como lo muestra la figura 6.6.

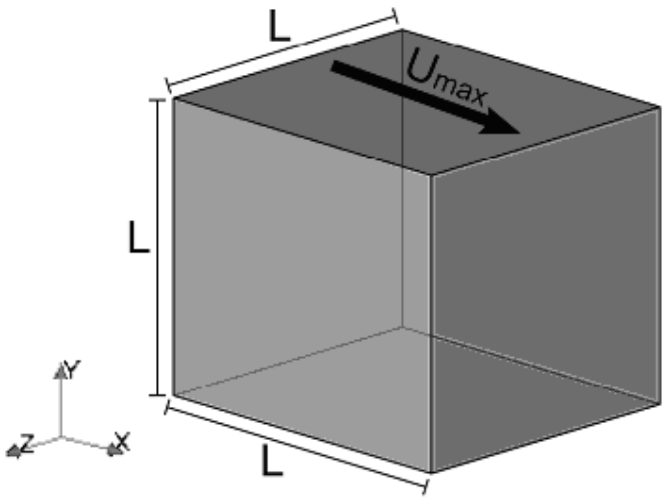


Figura 6.6: Esquema del problema de escurrimiento en cavidad cúbica.

El problema de la cavidad cúbica se caracteriza con el número Reynolds:

$$\mathrm{Re} = \frac{L U_{\max}}{\nu} \tag{6.1}$$

donde v es la viscosidad cinemática del fluido.

Se realizaron simulaciones numéricas de este problema para tres diferentes números de Reynolds 100, 400 y 1000 en grillas con dimensiones 16^3, 64^3 y 128^3 respectivamente con las condiciones de contorno descritas anteriormente. Las simulaciones fueron hechas partiendo del reposo en todas las celdas y densidad $\rho = 1$. Como criterio de convergencia al estado estacionario se usó la diferencia relativa media de los campos de velocidad entre un paso de tiempo y el siguiente. (eq. 6.2)

$$\sum_i \frac{\left\| U\left(x_i, t+\delta\right) - U(x_i, t) \right\|}{\left\| U(x_i, t+\delta \right\|} \leq 10^{-6} \tag{6.2}$$

En las figuras 6.7, 6.8 y 6.9 se grafican las componentes de velocidad U_y a lo largo de la dirección x que pasa por el centro geométrico de la cavidad cúbica y las componentes de velocidad U_x a lo largo de la dirección y que pasa por el mismo centro para los diferentes números de Reynolds. Las gráficas se comparan con los resultados de Yang, Yang, Chen y Hsu [60], las líneas corresponden a la simulación por LBM en CUDA y los puntos a los datos experimentales. Puede observarse que la concordancia es muy buena.

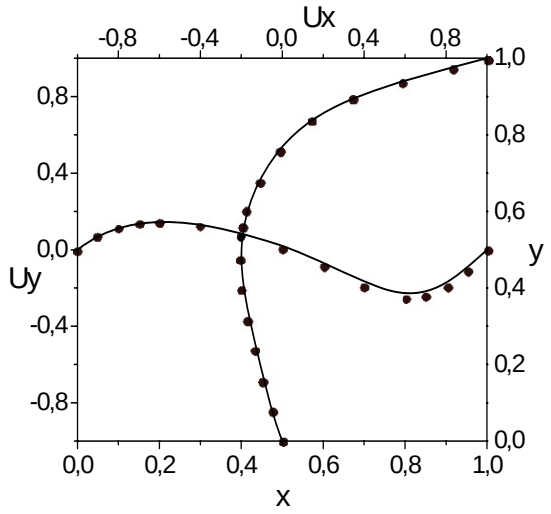


Figura 6.7: Componentes de velocidad Ux y Uy en el centro de la cavidad para Re=100.

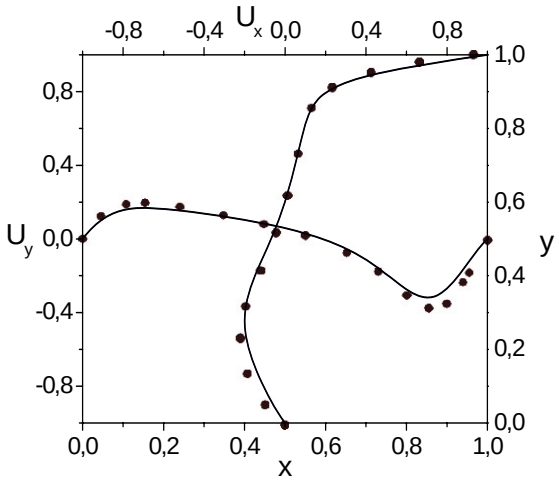


Figura 6.8: Componentes de velocidad Ux y Uy en el centro de la cavidad para Re=400.

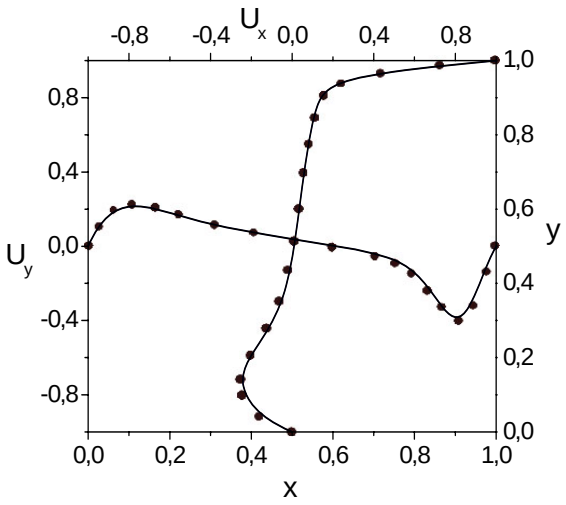


Figura 6.9: Componentes de velocidad Ux y Uy en el centro de la cavidad para Re=1000.

En las figuras 6.10, 6.11 y 6.12 se muestran imágenes 3D con las líneas de corriente que pasan por el eje central (*x=0,5; y=0,5*) calculadas a partir del campo de velocidades, donde puede observarse el vórtice central generado en cada caso y la naturaleza tridimensional del escurrimiento de fluido. Puede apreciarse como el vórtice central cambia de posición y forma en función del número de Reynolds.

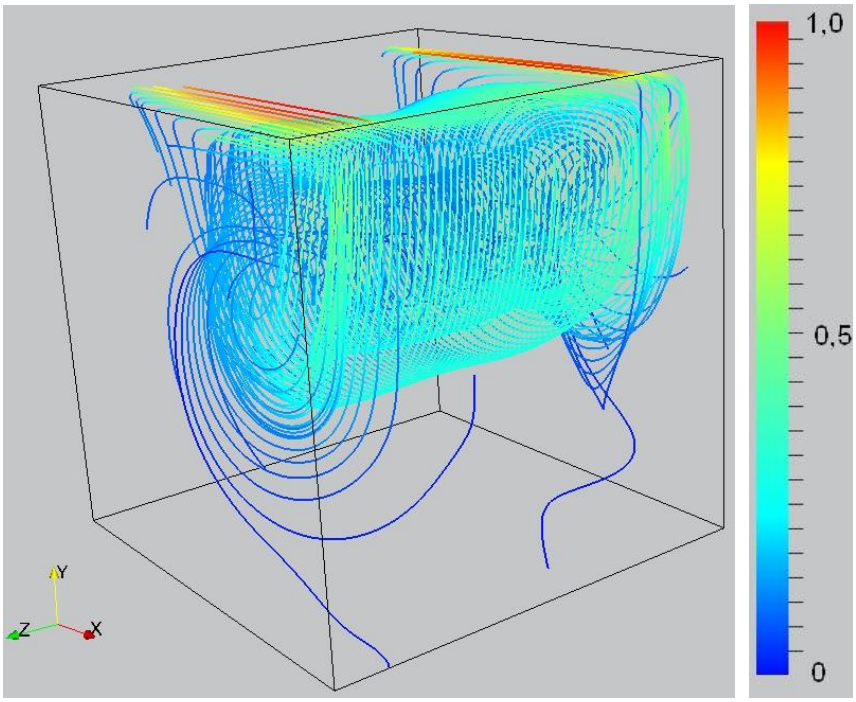


Figura 6.10: Líneas de corriente calculadas a partir del campo de velocidades para Re=100.

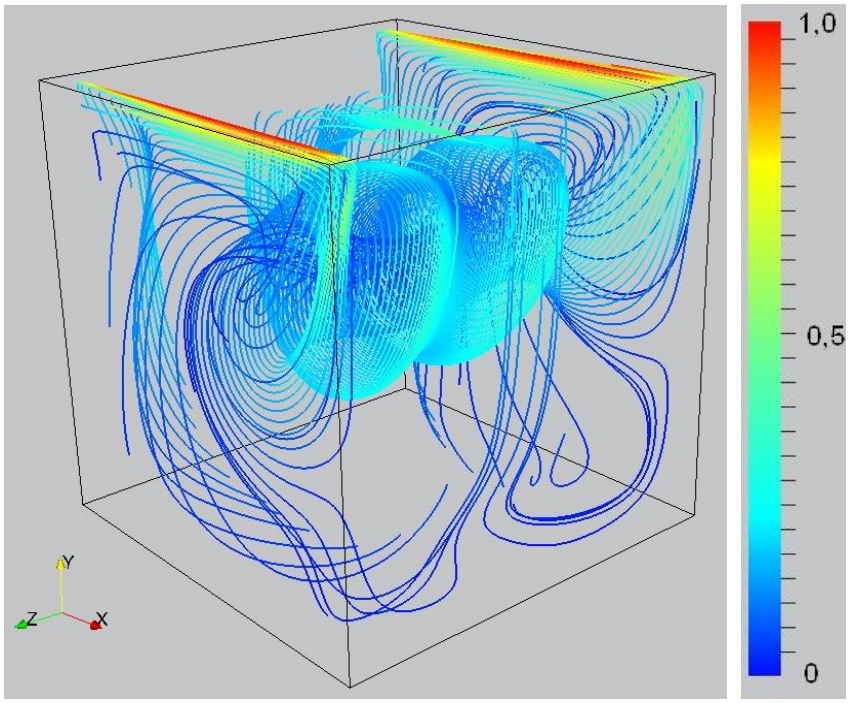


Figura 6.11: Líneas de corriente calculadas a partir del campo de velocidades para Re=400.

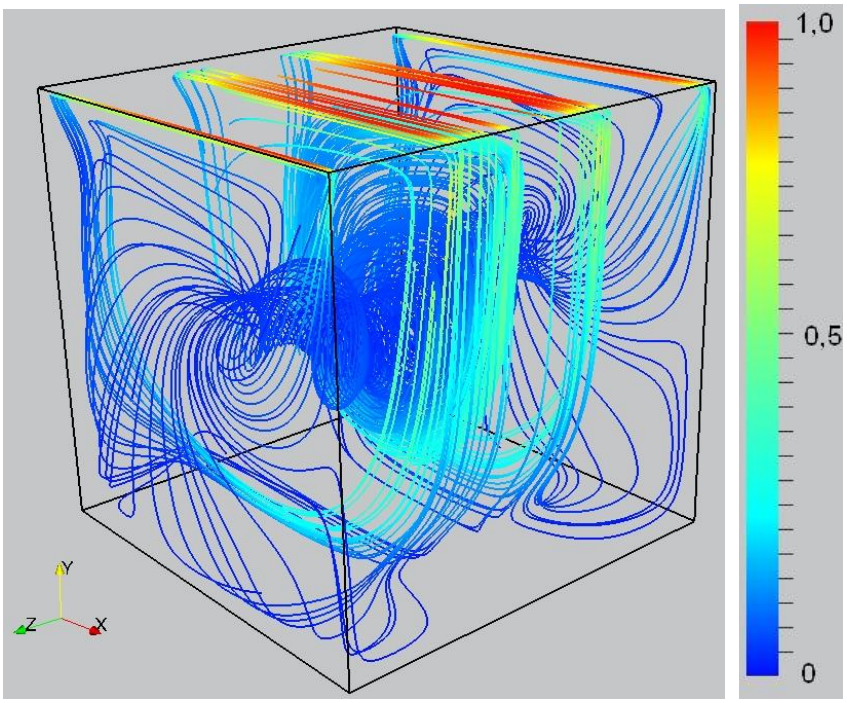


Figura 6.12: Líneas de corriente calculadas a partir del campo de velocidades para Re=1000.

En las gráficas siguientes (figuras 6.13, 6.14 y 6.15) se muestran las isosuperficies de módulo de velocidad en la mitad de la cavidad para los tres diferentes números de Reynolds. Aquí se vuelve a notar claramente como el vórtice central cambia de posición a medida que se aumenta el número de Reynolds. También se observa que, en el último caso (Reynolds = 1000), aparecen vórtices secundarios.

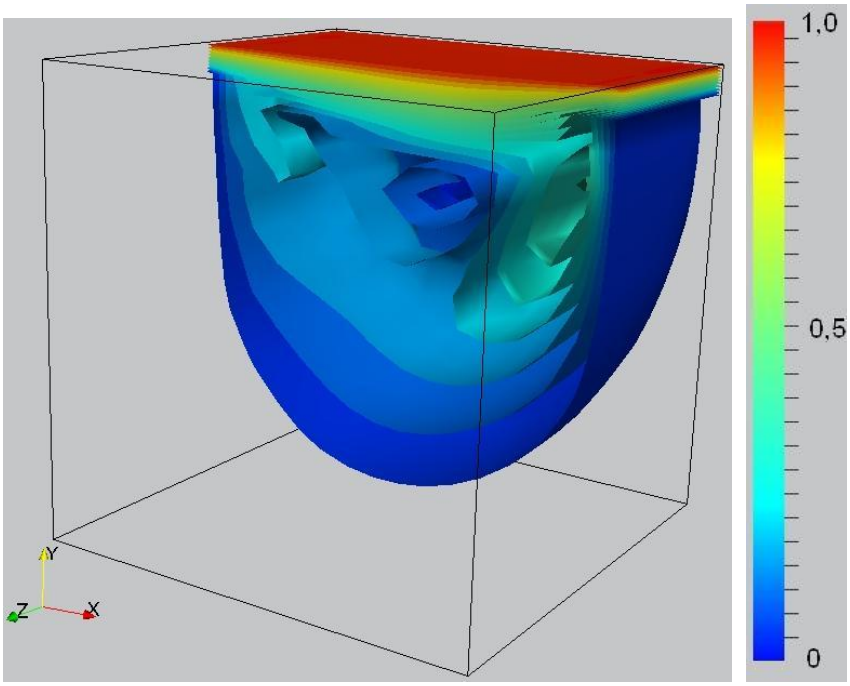


Figura 6.13: Isosuperficies de velocidad para Re=100.

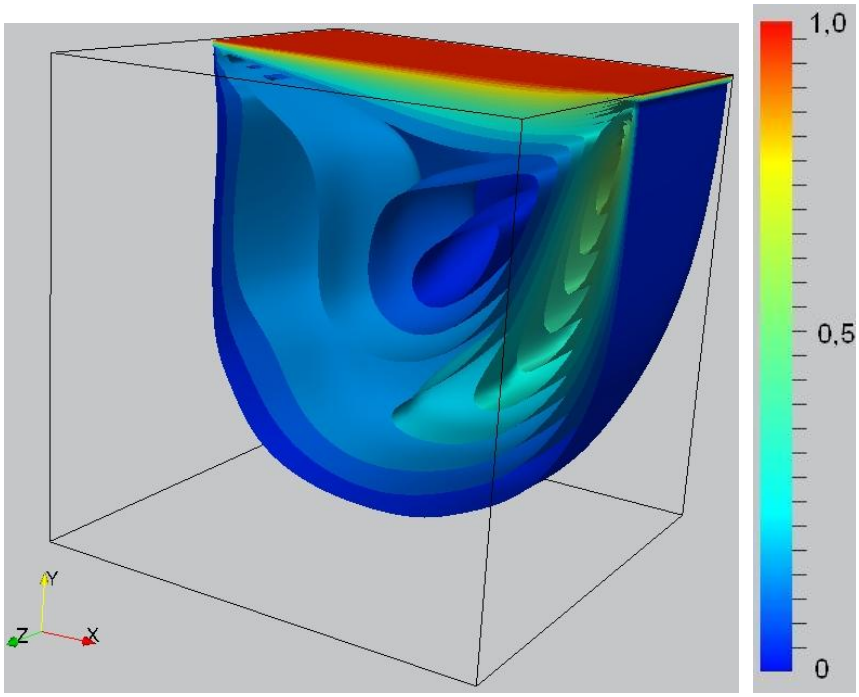


Figura 6.14: Isosuperficies de velocidad para Re=400.

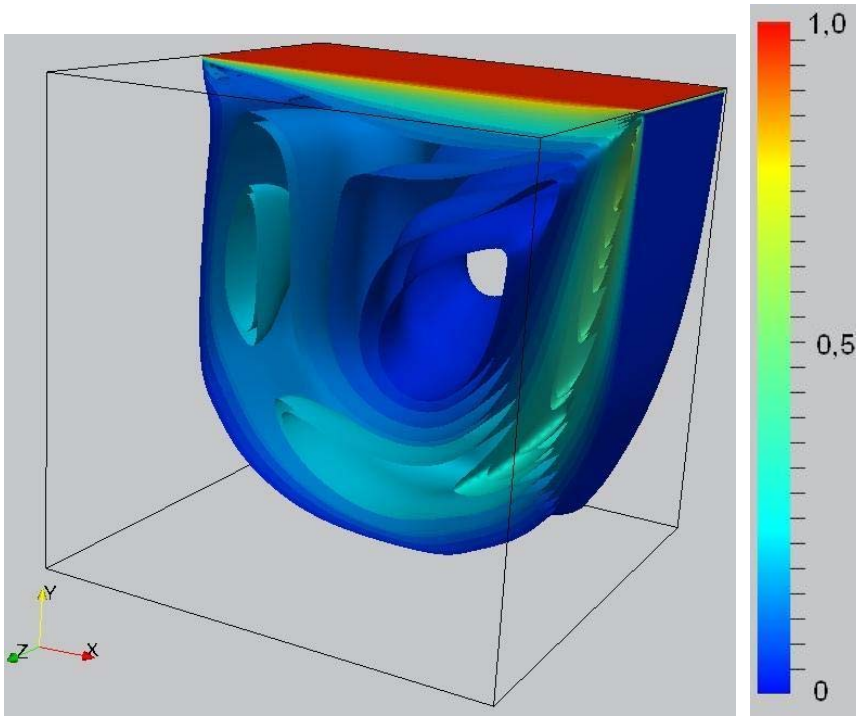


Figura 6.15: Isosuperficies de velocidad para Re=1000.

6.9. Performance

En la tabla 6.1 se muestra, en función del tamaño de grilla, la comparación de tiempo promedio de una iteración para toda la grilla y los correspondientes MLUPS aproximados para las versiones del modelo LBM en C y CUDA ejecutados sobre las correspondientes plataformas (CPU y GPU) y el *speedup* alcanzado. Puede verse que a partir de grillas de 64 celdas cúbicas se alcanzan *speedups* dos órdenes de magnitud.

Cabe señalar que la versión para CPU se implementó como un solo *thread* con tres ciclos anidados, uno para cada dirección y tanto el sentido de recorrido de la grilla como la macro de direccionamiento están optimizadas para este tipo de arquitecturas con memoria caché. La CPU utilizada tiene un procesador de doble núcleo AMD Athlon 64 X2 Dual Core de 2,41 GHz y 4 GB. de memoria RAM.

6.9.1. Ancho de banda

Para analizar el ancho de banda real alcanzado en las simulaciones se calcula la cantidad de datos transferidos desde y hacia memoria global por cada celda en cada iteración, medidos en Bytes. Este valor se multiplica por el número de LUPS máximo alcanzado y se compara con el ancho de banda teórico de la placa.

En el paso de advección se leen 19 *floats* (uno por cada dirección) más un *int* que define el tipo de celda. Luego de los cálculos se reescriben los 19 valores de punto flotante. Por lo tanto se transfieren 38×4 Bytes $+ 1 \times 2$ Bytes $= 154$ Bytes, a 259 MLUPS el ancho de banda alcanzado es de 37.1 GBytes/s, que es aproximadamente el 35% del ancho de banda pico de la placa. Este valor es muy alto para una simulación compleja como la presentada ya que para alcanzar el máximo teórico de la placa habría que implementar un código que solamente transfiera datos sin realizar ningún tipo de cálculos.

Aplicaciones que hacen este tipo de operaciones para obtener mediciones empíricas del ancho de banda de memoria en CUDA alcanzan valores cercanos al 90 %.

Tamaño de la Grilla	Versión C++		Implementación en CUDA			Speedup
	Tiempo de CPU (ms.)	MLUPS CPU	Bloques y threads	Tiempo de GPU (ms.)	MLUPS GPU	
32^3 30768	15.3	2.1	32 × 32 32	0.165	198	92
64^3 262144	124	2.1	64 × 64 64	1.066	245	116
96^3 884736	429	2.06	96 × 96 96	3.410	**259**	125
128^3 2097152	1092	1.92	128×128 128	9.629	217	113
144^3 2985984	1580	1.88	144×144 144	12.815	233	123
160^3 4096000	2164	1.89	160×160 160	16.512	248	**131**

Tabla 6.1: Tiempos de ejecución para el modelo de Lattice Boltzmann 3D.

6.9.2. Performance máxima

Partiendo del caso de mayor performance de la tabla 6.1 se realizaron otro tipo de optimizaciones dejando de lado la flexibilidad y posibilidades de configuración. Se cambiaron por constantes la mayoría de los parámetros que no varían durante una simulación como el tamaño de la grilla, las condiciones de contorno y el parámetro de relajación τ. También se eliminó el cálculo intermedio de densidad y velocidades, y se aplicaron condiciones de contorno de orden simple. Con estas modificaciones el tiempo de una iteración promedio se redujo de 3.41 ms a 2.23 ms lo que se traduce en aproximadamente 400 MLUPS y un ancho de banda de memoria de 57 GBytes/s, más del 50% del máximo teórico de la placa.

6.10. Escalabilidad y limitaciones del hardware

El mayor problema en tres dimensiones es el tamaño de memoria necesario para almacenar toda la estructura. El cubo más grande simulado es de 180 celdas por lado y ocupa aproximadamente 900 MB de memoria. Esta es una limitante mayor dado que en la GPU no existe esquema de memoria virtual, y el tamaño de la memoria de la placa es generalmente entre 2 y 10 veces menor al de la memoria principal del CPU. Las alternativas existente son, utilizar varias placas gráficas conectadas entre si mediante la

tecnología SLI de NVIDIA (actualmente se permiten hasta 3), triplicando el máximo tamaño máximo. También puede utilizarse el esquema propuesto por Bailey, Myre, Walsh, Lilja y Saar [61] que utiliza una sola copia de la matriz de distribuciones y un esquema de dos pasos diferentes implementados como dos *kernels* diferentes que se invocan alternativamente. En uno se realiza la advección, el paso de colisión y se vuelven a advectar las partículas colisionadas a la misma posición de donde se leyeron. En el segundo kernel, se toman esas distribuciones que fueron almacenadas en posiciones intercambiadas pero dentro de la misma celda y se realiza solamente la colisión. Este esquema maximiza la ocupación de la memoria global de la placa gráfica en desmedro de la performance global de la aplicación pero sin llegar al órden de magnitud.

Capítulo 7: Simulación de cavidad cúbica con velocidad variable

7.1. Introducción

La notable mejora en performance alcanzada por el esquema LBM en GPU para la simulación 3D que reduce en la mayoría de los casos en dos órdenes de magnitud el tiempo de cálculo, constituye un gran avance que abre la puerta para abordar problemas complejos en una PC. Por ejemplo, cálculos de transitorios 3D o estudios sistemáticos de sensibilidad que requieren sucesivas simulaciones tridimensionales con pequeñas variaciones paramétricas.

Para evaluar esta potencialidad, se planteó un caso de flujo oscilatorio en una cavidad cúbica. Partiendo de la implementación presentada en el capítulo 6 y tomando el problema del flujo en una cavidad cúbica, se impuso una condición de velocidad variable en función del tiempo en la cara superior. Dado que los tiempos de simulación son muy cortos (menor a 5 segundos), se pudieron realizar numerosas pruebas en búsqueda de frecuencias que presenten amplitudes máximas en las oscilaciones internas de la cavidad.

Se utilizó una grilla de 32 celdas cúbicas con una velocidad máxima en la cara superior de 0,2 celdas/paso. El parámetro de relajación τ se fijó en 0,6, con lo cual se asegura un número de Reynolds máximo de 200. Incluso se pueden plantear situaciones similares con grillas de mayor tamaño en tiempos relativamente breves dado que esta implementación actualiza más de 200 millones de celdas LBM 3D por segundo (230 MLUPS promedio).

El caso de estudio es un escenario derivado del problema del flujo de Womersley [62], un problema clásico de análisis y benchmarking. El problema de Womersley consiste en el flujo de un fluido incompresible a través de un tubo cilíndrico con una presión de referencia de salida en el extremo derecho y una presión oscilatoria en el otro. (fig. 7.1).

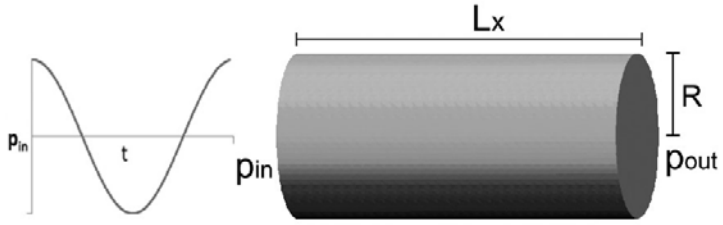


Figura 7.1: Escurrimiento de Womersley.

En nuestro caso consideramos como geometría la cavidad cúbica presentada en el capítulo anterior, imponiendo una condición de velocidad oscilatoria en la cara superior según la función:

$$U(t) = U_o \; sen \; \omega t \tag{7.1}$$

donde U_0 es la velocidad máxima y la frecuencia angular está determinada por:

$$\omega = \nu \left(\frac{Wo}{L} \right)^2 \tag{7.2}$$

donde *Wo* es el número de Womersley, *L* es la longitud de la arista de la cavidad y *v* es la viscosidad cinemática.

Se corrieron simulaciones variando el número de Womersley entre 0.3 y 1.6, y en cada caso se estudió la respuesta de la componente *x* de la velocidad u_x en un punto ubicado en el plano *z* central, en el punto medio del segmento diagonal entre el centro y la arista superior, esto es:

$$\frac{x}{L} = \frac{y}{L} = \frac{3}{4}; \;\; \frac{z}{L} = \frac{1}{2} \tag{7.3}$$

7.2. . Resultados

En la Fig. 7.2 se muestra la oscilación de u_x en el punto *test* en un periodo completo para diferentes *Wo*. En la gráfica se muestra también la oscilación externa *U* para comparación del desfasaje. Puede verse que tanto la amplitud como la fase varían con la frecuencia.

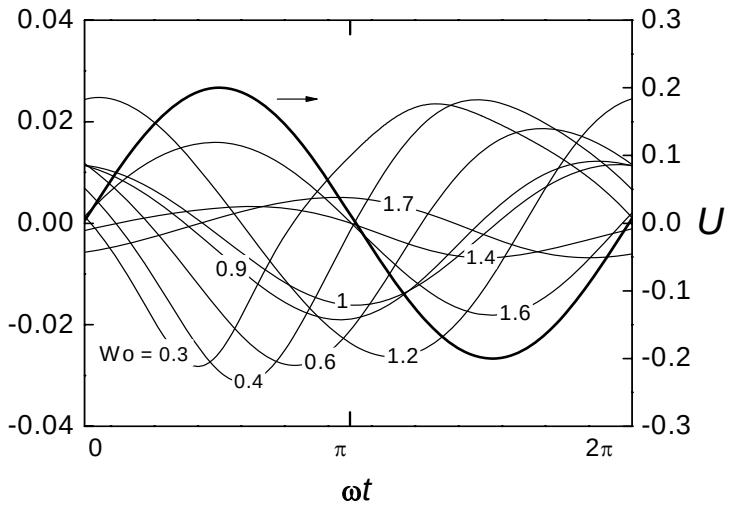


Figura 7.2: Velocidad Ux en función de la frecuencia. La curva más gruesa es la condición de contorno impuesta en la cara superior.

En la Fig. 7.3 se detallan las velocidades u_x máximas y mínimas, y la amplitud en el punto *test*. Puede verse que hay una máximo de amplitud en *Wo* = 1.2, pero también aparecen otros máximos relativos en *Wo* = 0.4 y *Wo* = 1.6.

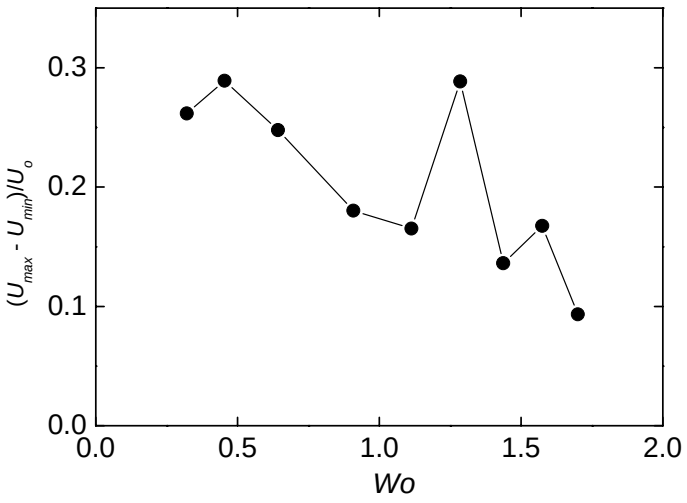
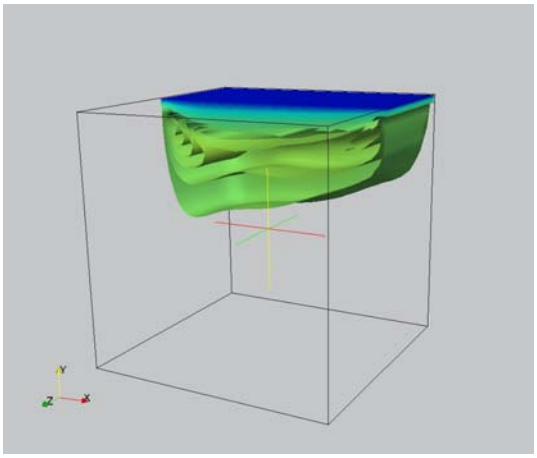
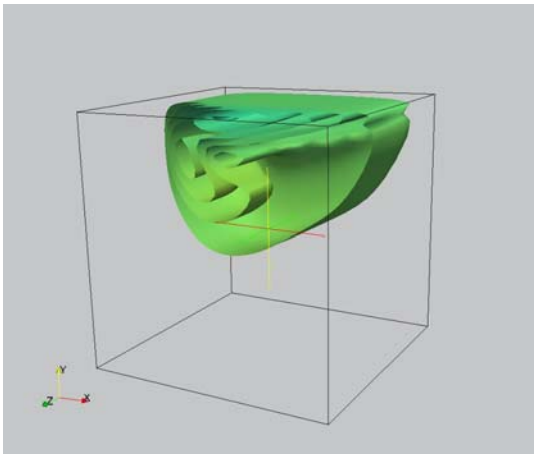


Figura 7.3: Amplitud de la oscilación de u_x en el punto test en función del número Womersley.

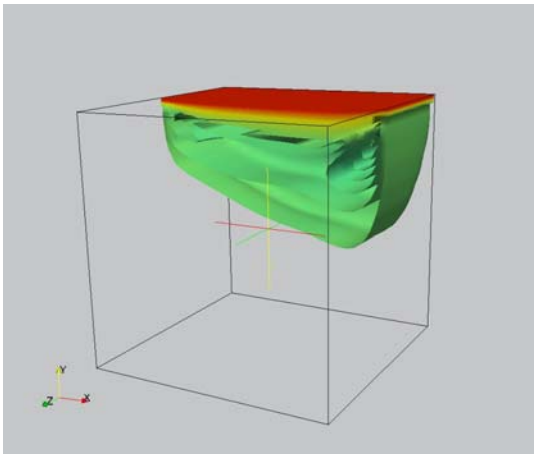
Las figuras 7.4, muestran las superficies de contorno de u_x en cinco instantes diferentes a lo largo de un periodo para un valor de Wo = 0.4, que como puede observarse en la figura 7.2 corresponde a un caso en que u_x oscila en contrafase con la condición de contorno U.



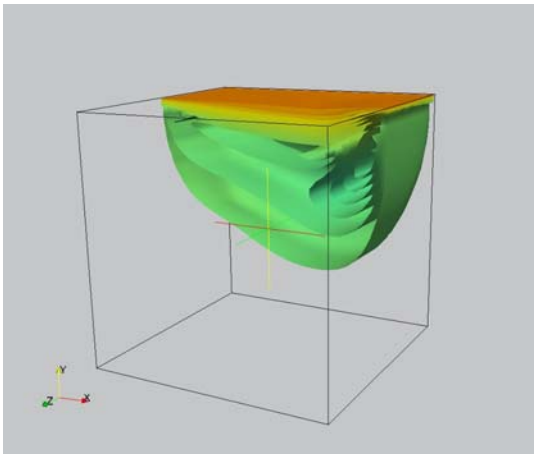
(a)



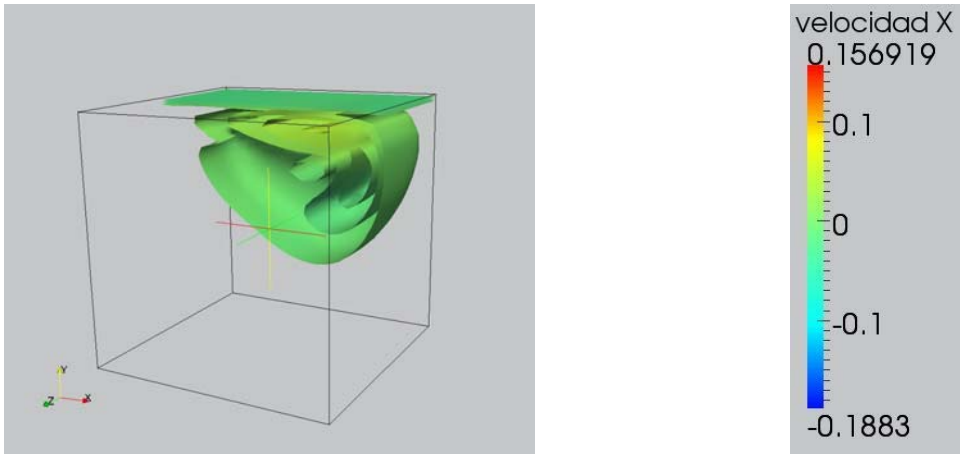
(b)



(c)



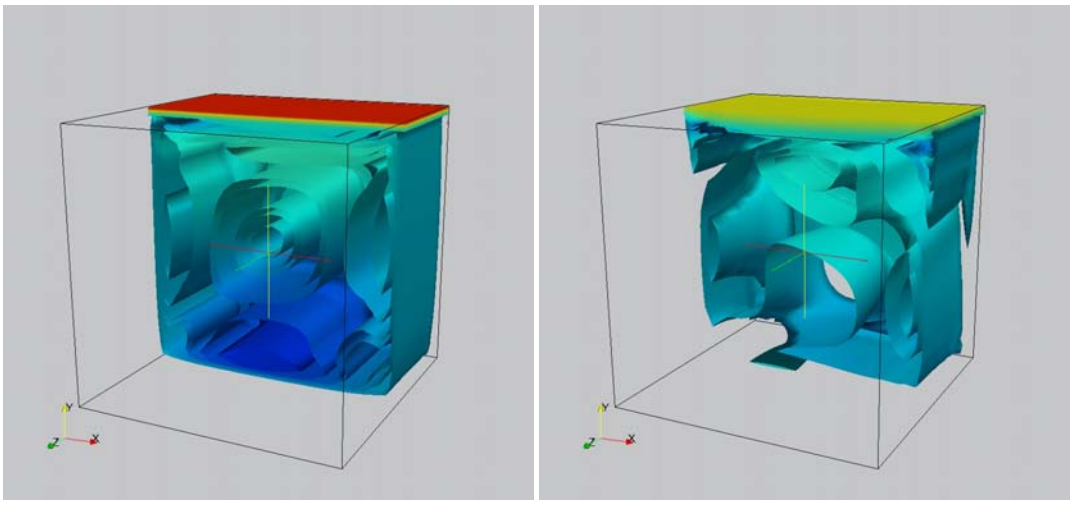
(d)



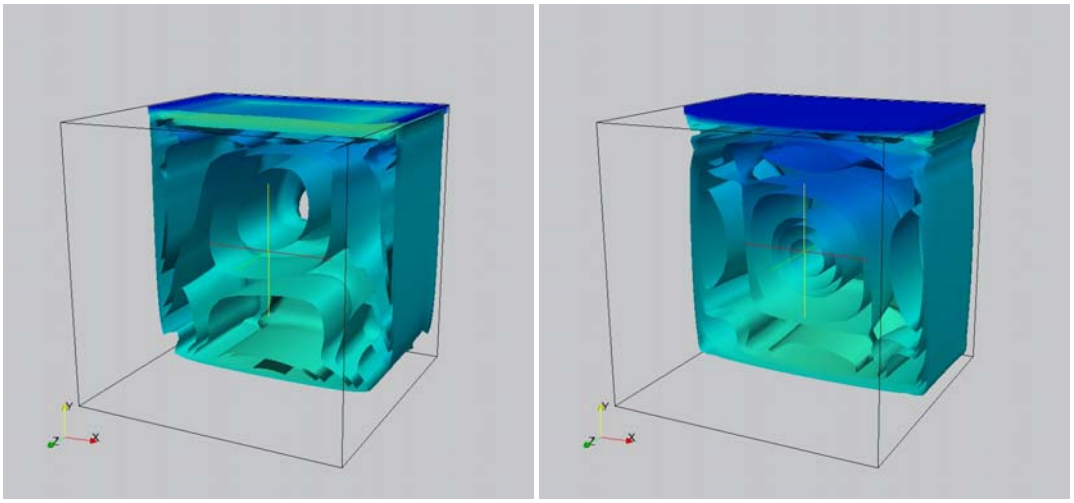
(e) *(f)*

Figuras 7.4: Isosuperficies de velocidad para Wo = 0.4.

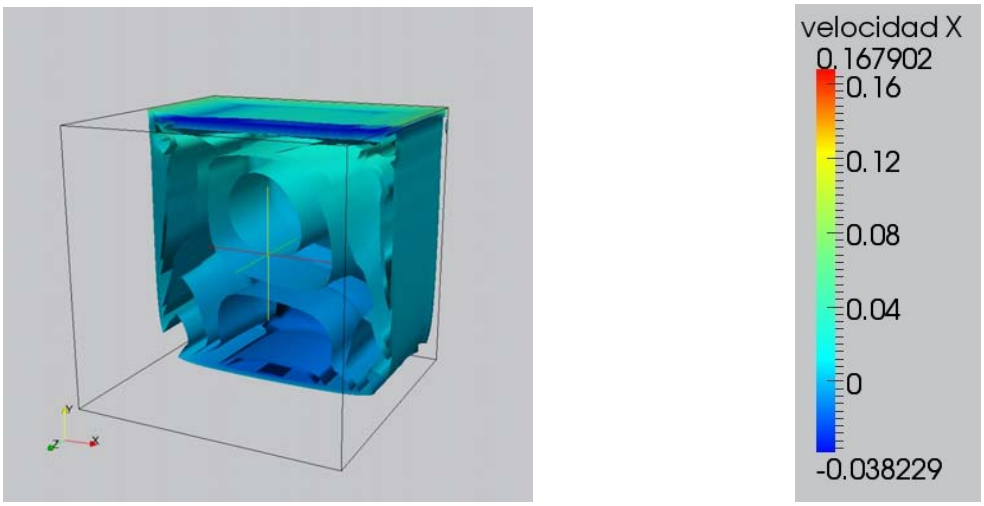
Las figuras 7.5 muestran las superficies de contorno de u_x a lo largo de un ciclo de oscilación para Wo = 1.6, que corresponde a una respuesta de U_x en fase con la condición de contorno.



(a) *(b)*



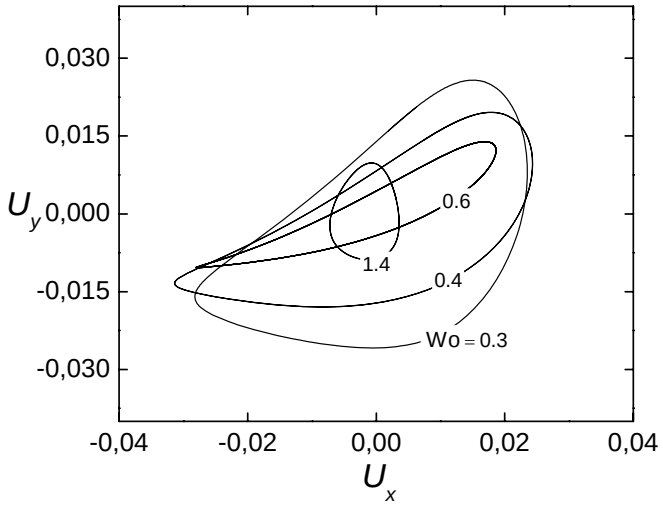
(c) *(d)*



(e) *(f)*

Figuras 7.5: Isosuperficies de velocidad para Wo = 1.6.

Combinando las componentes de la velocidad en un mismo punto se pueden obtener curvas en el espacio de las fases (u_x, u_y). En la figura 7.6 se pueden ver estas curvas para algunas de las frecuencias.

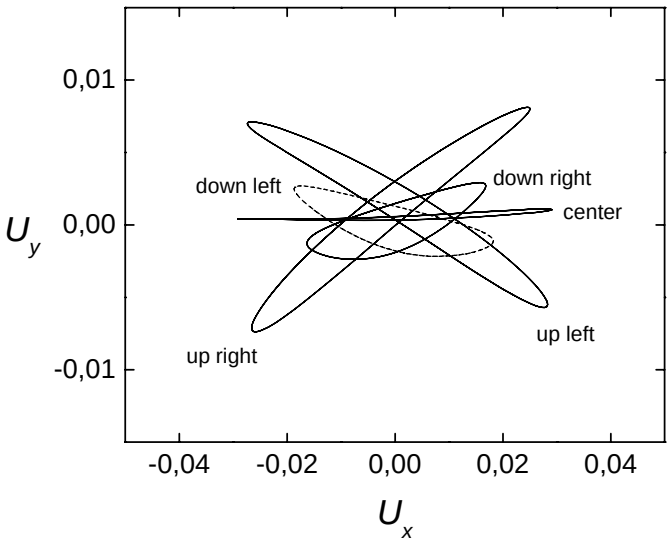


Figuras 7.6: Mapeo de las componentes de velocidad del punto test en el espacio de las fases.

Para analizar más en detalle la amplitud de oscilación máxima de *Wo* = 1.2, se calculó la velocidad en cinco puntos diferentes del cubo sobre el plano *z* central, el centro del cubo y los cuatro puntos simétricos al punto *test* original. Estos puntos son:

$$\begin{aligned}
center &: x = \frac{1}{2}L,\ y = \frac{1}{2}L,\ z = \frac{1}{2}L \\
up\,right &: x = \frac{3}{4}L,\ y = \frac{3}{4}L,\ z = \frac{1}{2}L \\
up\,left &: x = \frac{1}{4}L,\ y = \frac{3}{4}L,\ z = \frac{1}{2}L \\
down\,right &: x = \frac{3}{4}L,\ y = \frac{1}{4}L,\ z = \frac{1}{2}L \\
down\,left &: x = \frac{1}{4}L,\ y = \frac{1}{4}L,\ z = \frac{1}{2}L
\end{aligned} \tag{7.4}$$

Los ciclos en el espacio de las fases se muestran en la figura 7.7.



Figuras 7.7: Curvas en el espacio de las fases para 5 puntos de medición en frecuencia de resonancia con Wo = 1.2.

A su vez, en la Fig. 7.8 se graficaron los perfiles de velocidad máxima en cada dirección a lo largo de las líneas centrales (análoga a las figuras 6.7, 6.8 y 6.9 para el caso estacionario).

Estudios del tipo mostrado pueden realizarse en sólo algunas horas en geometrías variadas. Estos ejemplos muestran la enorme potencialidad para la ingeniería del modelo LBM implementado en GPU.

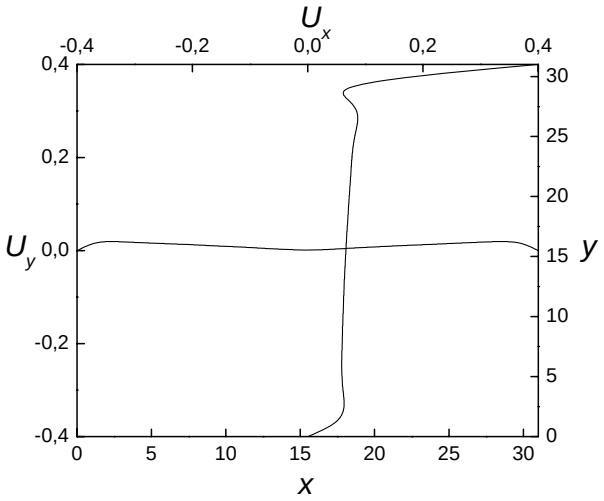


Figura 7.7: Perfiles centrales de U_x y U_y máximas Wo = 1.2.

Capítulo 8: Conclusiones

Se presentó un estudio extensivo y detallado de la tecnología de computación genérica sobre placas gráficas, denominada GPU Computing, más precisamente de la arquitectura propuesta por la empresa NVIDIA denominada CUDA. Se implementaron tres modelos de autómatas celulares diferentes, en dos y tres dimensiones buscando la mejor estrategia de paralelización y de utilización de los diferentes tipos de memoria disponibles en las GPUs.

En el primer caso estudiado, el AC de simulación de escurrimiento superficial AQUA se implementó sobre una de las primeras versiones CUDA, analizando principalmente las soluciones a los problemas de concurrencia presentados con la ejecución paralela. Las simulaciones se realizaron sobre una placa de la primera generación CUDA y se buscó optimizar la performance evaluando diferentes esquemas de división del dominio y configuraciones de ejecución. Los tiempos de ejecución resultantes reducen más de diez veces en comparación con implementaciones tradicionales.

En el segundo caso de estudio, el AC implementado fue el método de Lattice Boltzmann en su versión de dos dimensiones y nueve velocidades con regla de colisión BGK. Este modelo tiene mayor cantidad de información almacenada y transferida por celda, por lo que se realizó un estudio exhaustivo de los recursos de hardware disponibles en la GPU. En la implementación se utilizaron los diferentes tipos de memoria, analizando esquemas de almacenamiento de los datos y diferentes patrones de acceso a memoria global. Con estas y otras optimizaciones se obtuvieron aceleraciones de más de cuarenta veces en comparación con códigos equivalentes para CPU.

El tercer algoritmo desarrollado se trata de una versión de tres dimensiones del método de Lattice Boltzmann, que duplica la cantidad de variables por celda con respecto al de dos dimensiones, lo cual le da más relevancia aún al manejo de memoria así como a todo lo que atañe a las condiciones de contorno. Para este desarrollo se contó con una GPU más moderna y con la versión CUDA más actual que presenta algunas diferencias en los patrones ideales de acceso a memoria. Luego de varias optimizaciones se logró alcanzar más del 50% del ancho de banda de memoria teórico de la placa en simulaciones de fluidos con resultados validados y una aceleración de dos órdenes de magnitud.

Las implementaciones investigadas en este trabajo permitieron mejorar notablemente la performance alcanzada en trabajos previos. En particular, para los modelos basados en LBM en CPU, Mazzeo y Coveney [63] alcanzaron menos de 10 MLUPS en simulaciones con la versión d3Q15 sobre un procesador Power4 de 1,7 GHz. Otros estudios de performance sobre procesadores HPC como el de Wellein, Zeiser, Hager y Donath [38] lograron 64 MPLUS para un código LBM d3Q19 sobre una Cray X1. Implementaciones en paralelo como la de Pohl et al. [17] alcanzan 160 MLUPS corriendo simulaciones 3D sobre clusters de 64 núcleos de diferentes arquitecturas (Opteron, Xeon, Altix y Nec SX6). Todos éstos están por debajo de los 400 MLUPS de la implementación 3D de la presente tesis. Sólo en el trabajo de Mazzeo y Coveney [63] se sobrepasa este valor llegando a casi 4500 MLUPS, pero sobre un cluster de 1024 supercomputadoras HPCx IBM SP p690+ Power4 1.7 GHz [64], simulando una geometría compleja pero de baja cantidad de nodos intercomunicados entre procesadores.

Los valores de performance alcanzados dan una idea del potencial de cálculo de las GPU actuales y muestran que el GPU Computing es una técnica viable a la hora de acelerar procesos computacionales costosos y paralelizables como los Autómatas Celulares.

Debe aclararse que tanto las placas gráficas como las CPU utilizados en la presente tesis corresponden a tecnologías computacionales de escritorio u hogareñas. Las GPU utilizadas tienen valores comerciales en Argentina que no sobrepasan los 200-300 dólares estadounidenses, son equipos de bajo costo disponibles hoy en día en cualquier laboratorio y originalmente diseñados para videojuegos o animaciones 3D. Por eso también se implementó una versión distribuida en PETSc para obtener tiempos de cálculo paralelo sobre este tipo de CPUs, pero los resultados no mejoraron el rendimiento de aplicaciones mono *thread*.

Diferentes autores dan configuraciones recomendadas u óptimas de bloques de threads sin embargo no siempre se comprobaron. Esto se debe fundamentalmente a particularidades de cada AC, de las decisiones de implementación y de la versión del *toolkit* utilizada. Un común denominador en los AC implementados es la baja complejidad computacional en relación a la comunicación de datos entre celda lo que hace que en todos los casos la performance esté acotada por el ancho de banda de

memoria. Esta condición fue explotada al máximo en las implementaciones buscando utilizar el máximo posible del ancho de banda teórico del hardware llegando al 35% mediante el uso de memoria compartida y accesos alineados.

La limitación principal se encontró en el tamaño de la memoria global (entre 4 y 10 veces menor a la del CPU), lo cual en la mayoría de los casos limita el uso de una GPU para realizar grandes simulaciones. De todos modos, existen equipos más grandes del tipo clusters como los TESLA de la empresa NVIDIA de hasta 100 placas TESLA de 6 GB de memoria cada una, lo cual mejora sustancialmente las posibilidades de una simple GPU.

Si bien la tecnología CUDA es de uso libre, su código es cerrado y es solamente utilizable sobre placas gráficas de la misma marca. Hoy día existen otras alternativas de código abierto como OpenCL, que no está tan desarrollado como CUDA pero que está evolucionando como Standard de GPU Computing y con mucha mayor compatibilidad de hardware (cualquier marca de GPU, arquitecturas multi-núcleo como los CPU actuales o incluso esquemas híbridos GPU-CPU). La empresa NVIDIA esta colaborando con el desarrollo de OpenCL y muchas de las ideas exitosas de CUDA están siendo volcadas a esta nueva plataforma.

Cuatro años atrás, al comienzo de este trabajo, el ambiente del HPC era en general muy reticente a esta nueva tecnología y se hablaba de que era una moda. Hoy en día es raro que laboratorios o centros de investigación que trabajan en HPC no posean algún grupo especializándose en GPU Computing. Es nuestra esperanza que los resultados de esta tesis sirvan para incentivar la investigación en este campo que tiene gran potencialidad en la ingeniería.

Apéndice I: Ejemplo de relación entre autómatas y las ecuaciones diferenciales

Los autómatas son esencialmente estados aritméticos que cambian siguiendo una regla de iteración. Por simplicidad supongamos un autómata finito representado por una sola variable, *x,* que sigue la regla:

$$x_n = \frac{x_{n-1}}{2} \tag{9.1}$$

La solución exacta de esta iteración es:

$$x_n = \frac{x_0}{2^n} \tag{9.2}$$

donde x_0 es el valor inicial.

La pregunta que podemos hacernos es ¿qué ecuación diferencial simula este autómata? Nótese que el lapso de tiempo trascurrido entre iteraciones no está determinado. Supongamos que por convención asignamos un valor fijo Δt al paso de tiempo entre iteraciones; entonces la ecuación del autómata se lee:

$$x(t + \Delta t) = \frac{x(t)}{2} \tag{9.3}$$

Desarrollando en Taylor a primer orden en Δt se obtiene:

$$\dot{x} = -\frac{1}{2\Delta t} x \tag{9.4}$$

Es decir que a primer orden el autómata planteado en la Ec. (9.1) aproxima un decaimiento exponencial:

$$x = x_o \; e^{-\lambda t} \tag{9.5}$$

con constante de decaimiento:

$$\lambda = \frac{1}{2\Delta t} \tag{9.6}$$

En una primera instancia entonces podría decirse que, a primer orden, se puede simular cualquier decaimiento exponencial con este autómata, asignando el paso de tiempo apropiado. Sin embargo, para que la simulación sea útil, deberíamos tener libertad de elegir el paso de tiempo Δt (por ejemplo para obtener un flujo de información dado para fines de control). Una manera de lograr esto es introduciendo un paso de relajación:

$$\begin{aligned} x_n &= \alpha\, x'_n + (1-\alpha)\, x_{n-1} \\ x'_n &= \frac{x_{n-1}}{2} \end{aligned} \tag{9.7}$$

Lo cual lleva al autómata:

$$x_n = \left(1-\frac{\alpha}{2}\right) x_{n-1} \tag{9.8}$$

el cual a primer orden aproxima la ecuación diferencial:

$$\dot{x} = -\left(1-\frac{\alpha}{2}\right)\frac{x}{\Delta t} \tag{9.9}$$

que es un decaimiento exponencial con constante de decaimiento:

$$\lambda = \left(1-\frac{\alpha}{2}\right)\frac{1}{\Delta t} \tag{9.10}$$

De esta manera, con el parámetro α se puede controlar la constante de decaimiento para cualquier Δt dentro del rango: $\frac{1}{2} < \lambda\Delta t < 1$.

El parámetro α del modelo AQUA cumple la misma función del parámetro de relajación del ejemplo mostrado. Análogamente, el parámetro τ del modelo de Lattice Boltzmann sirve para controlar la viscosidad del fluido simulado.

Apéndice II: Relación de parámetros del modelo LBM

En una simulación de fluidos por el método de Lattice Boltzmann las variables internas se definen en unidades de la grilla. La manera de relacionar estas variables con las variables físicas del problema real que se intenta simular es a través del número adimensional de Reynolds:

$$\mathrm{Re} = \frac{U\,L}{\nu} \qquad \frac{[m/s][m]}{[m^2/s]} = \frac{[\delta x/\delta t][\delta x]}{[\delta x^2/\delta t]} \tag{10.1}$$

donde U es la velocidad del fluido, L es la longitud característica del problema y ν es la viscosidad cinemática que, para el caso de LBM se calcula de la siguiente manera:

$$\nu = \left[\frac{(2\tau - 1)}{6}\right] e^2 \delta t \tag{10.2}$$

La ecuación 10.2 relaciona la viscosidad cinemática del fluido simulado con el parámetro de relajación τ, el cual debe ser siempre mayor a 0,5 para evitar viscosidades negativas. Como las partículas alojadas en una celda no pueden pasar más allá de la celda vecina en una iteración del modelo, la velocidad máxima está acotada por 0,5 en unidades de la grilla. En la práctica debe tomarse un margen de seguridad sobre estos límites para que las simulaciones no se vuelvan inestables. Los límites de estabilidad testeados para el modelo del capítulo 6 son:

$$\tau > 0,6 \qquad U < 0,4 \tag{10.3}$$

Con este valor de τ, la viscosidad cinemática mínima es de:

$$\nu = \left[\frac{(2 \bullet 0,6 - 1)}{6}\right] = 0.0\hat{3}\ \left[\frac{\delta x^2}{\delta t}\right] \tag{10.4}$$

Para el ejemplo de la cavidad cúbica del capítulo 6, si queremos simular un flujo a Reynolds 1000, la cantidad de celdas por lado mínima requerida se calcula de la siguiente manera:

$$1000 = \frac{0,4 \bullet L}{0.0\hat{3}} \qquad L \cong 84\,[\delta x] \tag{10.5}$$

Con lo que se necesitan al menos 84 celdas por lado para obtener una simulación correcta.

Agradecimientos

A las entidades financiadoras de este proyecto, la CNEA y la UNCPBA por el espacio y el equipamiento brindado tanto dentro del Centro Atómico Bariloche como en Tandil y principalmente el CONICET por mi beca de postgrado.

Al Insituto Balserio por darme tan valiosa formación académica, por la actitud casi paternal de los docentes de las materias que cursé que sin duda me hicieron todo más llevadero. Y un especial agradecimiento a Marcela Margutti que atendió siempre con buena onda y una sonrisa todo tipo de solicitudes y planteos con la mayor eficiencia.

A mi director, el Dr. Enzo Dari a quien considero un amigo, que me sirvió de guía en la senda de la programación paralela y el Linux Debian y también por los senderos del Cerro Catedral en bici.

A mi co-director, el Dr. Marcelo Vénere por su buena onda y por vincularme al grupo MECOM y a todo el Centro Atómico Bariloche.

A Alejandro Clausse por sus inagotables ganas de trabajar, responder, leer, corregir y sugerir tan fundamentales para mi trabajo.

A toda la gente que me ayudó a trabajar con el método de Lattice Boltzmann: Diego Dalponte, Federico Teruel, Gustavo Boroni, Pablo Blanco, Daniel Goldbert, Jain Parshant y con todo lo relacionado con plácas gráficas: Cristian García y Juan D'Amato.

A todo el PLADEMA, que resiste unido los embates de un entorno no tan amigable gracias a Rosana, Mariana, las facturas del Rafa, los asados del Boro y las tortas de Virginia. Aldo, Fer, Diego, Cristian, Juan, Mara, Lazito, Mario, Pablo, Fede y Guadcore que tiene la clave de root de la vida.

A toda la maravillosa gente de Bariloche, principalmente el grupo MECOM, del que me considero miembro vitalicio y en el que la pasé tan bien trabajando, con los cafecitos, las picadas de los viernes, las disquisiciones político-filosóficas y las charlas deportivas. Claudio, Mario, Fede, Mariano, Enzo, Paola, Daniela, Pablo, Matías, el Turco y algunos que se fueron buscando otros rumbos como Leo, Roberto, Marcus, Nathalie y Ciro. Siempre me siento en casa con ustedes.

A los pibes compañeros de fluidos: Juanpi, Martín y Facu que se solidarizaron desde un principio con nuestra situación (totalmente perdidos) y terminaron siendo grandes amigos con los que compartí estudio, asados, cervezas de la cruz y salidas en bici, también a Ana y al pequeño cabezón Agustín.

A Raúl Marino, mi guía en todo dentro del IB, un tipo genial si no fuera por su bigote que por suerte se afeitó, Ana, Ariadna y a esos mágicos vinitos mendocinos.

Al Pampa que fue de las primeras personas en ayudarme en mi estadía en Bariloche, explicando, saliendo en bici; con sus exquisitos corderos al asador y su hermosa familia Irina y Valentín.

Al grupito de 2008, Laura, Caludia (Mich y Baff), Mauri y Vani con los que escapábamos del encierro del pabellón bajo cualquier excusa o veíamos películas noventosas. A René, un groso. Cómo podés se tan grande y tan pequeño al mismo tiempo! y Nathalie que espero verlos juntos nuevamente pronto.

A los demás amigos del pabellón seis: Santiago, Reynaldo, Pablo L, Mario, Pablito y todos los demás menos el cubano que me rompió el mate.

A Pablo, Paula, Sol y Agus que me solucionaron muchas cosas y me integraron a su familia y a su grupo de amigos como si fuera uno más.

A Martín y Paula que eran la posta en el camino y fueron nuestro refugio tantas veces, a Diana que siempre me mandaba algo rico y a Helio por su confianza y por conseguirme el Cengel que estaba agotado.

A Diego, Marina y Ramón (el barilochense más groso que conozco), gracias a ustedes nunca estuve solo. Y a todos los que en algún momento fueron para Bariloche, papá y mamá que los quiero mucho, Eduardo y Mariana; Fony, Gerardo y Aneta; Ernesto y Elena; Alicia y Dori. A los marplanautas locos: Sebas, Chino, Ariel, Pablo, Raba y su chica, Luis, Gaby que fueron específicamente a visitarme y de paso hicieron snowboard. A Mati y Maru, genios. A la familia Dalponte: Mario, Esther, Facundo, Valeria y Martín. Y también los tandilenses Fer, Sole y Juana.

A mi más fiel compañero de todos los tiempos, que aguantó mis maltratos y las más agresivas condiciones climáticas, perdón verdolaga.

Y por último a esa personita maravillosa que amo tanto que no sólo soportó la soledad sino también mis llamadas enojado porque las cosas no iban bien. Por apoyarme siempre a full y hacer un sacrificio enorme para que yo logre mi objetivo. Mi compañera de la vida que ahora tiene a mi bebé creciendo en su interior. Gracias Julia, te amo!

Bibliografía

[01] Cooper N.G. (ed.). Los Alamos Science Number 15. Special Issue Stanislaw Ulam 1909-1984. Los Alamos National Laboratory, New Mexico. 1987.

[02] Gardner, M. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American* **223**, 120–123, 1970.

[03] Wolfram, S. *A New Kind of Science* [version electrónica]. Champaign, IL: Wolfram Media, 2002, 2010. [2007]. Disponible de: www.wolframscience.com. 1-57955-008-8.

[04] Crisci G.M., Rongo R., Di Gregorio S., Spataro W. The simulation model SCIARA: the 1991 and 2001 lava flows at Mount Etna. *Journal of Volcanology and Geothermal Research*, **3010**, 1-15, 2004.

[05] Rinaldi P., Dalponte D., Vénere M., Clausse A. Cellular automata algorithm for simulation of surface flows in large plains. *Simulation Modeling Practice and Theory*, **15**, 315-327, 2007.

[06] Dalponte, D.D., Rinaldi, P.R., Cazenave, G., Usunoff, E., Varni, M., Vives, L., Vénere, M.J., Clausse, A. A validated fast algorithm for simulation of flooding events in plains. *Hydrological Processes*, **21**, 115-1124, 2007.

[07] Wolfram S. Cellular automata fluids 1: Basic theory. *Journal of Statistical Physics*, **45(3)**, 475-526, 1986.

[08] Chen H., Chen S., Matthaeus, W.H., Recovery of the Navier-Stokes equations using a lattice Boltzmann method. *Physical Review*, **A 45**, R5339, 1991.

[09] Frisch, U., Hasslacher, B., Pomeau, Y. Lattice-Gas Automata for the Navier-Stokes Equation. *Phy. Rev. Lett.*, **56(14)**, 1505-1508, 1986.

[10] Hardy, J., de Pazzis, O., Pomeau, Y. Molecular Dynamics of a Lattice Gas: transport properties and time correlation functions. *Physical Review*, **A 13**, 1949, 1973.

[11] Hardy, J., de Pazzis, O., Pomeau, Y. Time Evolution of a Two- Dimensional Model System I: invariant states and time correlation functions. *J. Math. Phy.*, **14**, 1746, 1976.

[12] Chen S. and Doolen G. D. Lattice Boltzmann Methods for Fluid Flous. *Annu. Rev. Fluid Mech.*, **30**, 329, 1998.

[13] Hou, S., Zou, Q., Chen, S., Doolen, G., Cogley, A. Simulation of cavity flow by the lattice Boltzmann method. *Journal of Computational Physics*, **118(2)**, 329 – 347, 1995.

[14] Luo, L.S. Symmetry Breaking of Flow in 2D Symmetric Channels: Simulations by Lattice-Boltzmann Method. *International Journal of Modern Physics C*, **8(4)**, 859-867, 1997.

[15] Shirani, E., Jafari, S. Application of LBM in Simulation of Flow in Simple Micro-Geometries and Micro Porous Media. *The African Physical Review* **1(1)**, 1970-4097, 2007.

[16] Artoli, A.M., Abrahamyan, L., Hoekstra, A.G., Accuracy versus Performance in Lattice Boltzmann BGK Simulations of Systolic Flows. *Lecture Notes in Computer Science*, **3039**, 1611-3349, 2004.

[17] Pohl, T., Deserno, F., Thurey, N., Rude, U., Lammers, P., Wellein, G., et. al. Performance evaluation of parallel large-scale lattice Boltzmann applications on three supercomputing architectures. En: Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC '04, 2004, Washington DC, U.S.A.). IEE Computer Society; Washington DC, U.S.A., ACM/IEEE, 2004. p. 21.

[18] Carter, J., Oliker L., Shalf, J. Performance Evaluation of Scientific Applications on Modern Parallel Vector Systems. *Lecture Notes in Computer Science*, **4395**, 1611-3349, 2007.
[19] Moore, S.K. Multicore is bad news for supercomputers. *Spectrum, IEEE*, **45(11)**, 15, 2008.
[20] Pelliccioni, O., Cerrolaza, M., Herrera, M. Lattice Boltzmann dynamic simulation of a mechanical heart valve device. *Mathematics and Computers in Simulation*, **75(1-2)**, 1– 4, 2007.
[21] He, X., Duckwiler, G., Valentino, D.J. Lattice Boltzmann simulation of cerebral artery hemodynamics. *Computers & Fluids*, **38(4)**, 789–796, 2009.
[22] Moreland, K., Angel, E. The FFT on a GPU. En: SIGGRAPH/Eurographics Conference On Graphics Hardware, (6, 2003, San Diego, Califonia). Eurographics Association; San Diego, California, U.S.A. Doggett, M., Heidrich, W., Mark, W., Schilling, A., (eds.), 2003, pp. 112–119.
[23] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture – Programing Guide.* [version electrónica]. 3.1. Santa Clara, CA, U.S.A. NVIDIA Corporation, 2010, 2010. [2010]. Disponible de: http://developer.nvidia.com/object/cuda_3_1_downloads.html.
[24] NVIDIA. *NVIDIA OpenGL Game of Life Demo.* [en linea]. 2001. Disponible de: http://developer.download.nvidia.com/SDK/9.5/Samples/samples.html#GL_GameOfLife
[25] Wei X., Wei, L., Mueller K., Kaufman, A.E. The Lattice-Boltzman Method for Simulating Gaseous Phenomena. *IEE Transactions on Visualization And Computer Graphics*, **10**, 164-176, 2004.
[26] Harris, M.J., Coombe, G., Scheuermann, T., Lastra, A. Phisically-based visual simulation on graphics hardware. En: Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware, (5, 2002, Saarbrücken, Germany). Eurographycs Association; 2002, pp. 109-118.
[27] Li, W., Wei, X., Kaufman, A. Implementing Lattice Boltzmann Computation on Graphics Hardware. *The Visual Computer*, **19(7-8)**, 444-456, 2003.
[28] NVIDIA. *NVIDIA CUDA Home Page*. [en línea]. 2008]. Disponible en: http://www.nvidia.com/object/cuda_home_new.html
[29] Tölke, J. Implementation of a Lattice Boltzmann kernel using the Compute Unified Architecture Developer by nVIDIA. *Computing and Visualization in Science*, **13**, 29-39, 2008.
[30] Vénere, M.J., Clausse, A. A computational environment for water flow along floodplains. *International Journal on Computational Fluid Dynamics*, **16**, 327-330, 2002.
[31] Weisstein, Eric W. *MathWorld--A Wolfram Web Resource*. [en línea]. Wolfram Research, 2010, 2011. Moore Neighborhood. Disponible en: http://mathworld.wolfram.com/MooreNeighborhood.html.
[32] Dalponte, D.D., Rinaldi, P.R., Vénere, M.J., Clausse, A. Algoritmos de grafos y autómatas celulares: Aplicación al la simulación de escurrimientos. En: MECOM (VIII, 2005, Buenos Aires, Argentina). Mecánica Computacional, Vol. XXIV; Buenos Aires, Argentina: Larrateguy, A. (ed.), 2005. pp. 19.
[33] Rinaldi, P.R., Dalponte, D.D., Vénere, M.J., Clausse, A. Autómatas Celulares sobre Grafos de Nodos Irregulares: Aplicación a la Simulación de Escurrimientos

Superficiales en Zonas de Llanura. En: CACIC 2007, (13, 2007, Corrientes, Argentina). Anales. Corrientes, Argentina: Universidad Nacional del Nordeste, 2007, pp 1656-1667.
[34] Goldstein, S., (ed.). Modern Developments in Fluid Dynamics. Vol. I. Cap. VII: Flow in pipes and channels and along flat plates. New York: Dover Publications, Inc, 1965.
[35] Goodnight, N. *CUDA/OpenGL Fluid Simulation.* [Versión electrónica]. Santa Clara, CA, U.S.A. NVIDIA Corporation, 2007, 2007. Disponible de: http://new.math.uiuc.edu/MA198-2008/schaber2/fluidsGL.pdf
[36] Zhao, Y. Lattice Boltzmann based PDE Solver on the GPU. *The Visual Computer*, **24(5)**, 323-333, 2008.
[37] Podlozhnyuk, V. *Histogram calculation in CUDA*. [Versión electrónica]. Santa Clara, CA, U.S.A. NVIDIA Corporation, 2007, 2007. Disponible de: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram256/doc/histogram.pdf
[38] Wellein, G., Zeiser, T., Hager, G., Donath, S. On the single processor performance of simple lattice Boltzmann kernels, *Computers & Fluids*, **35**, 910-919, 2006.
[39] Higuera, F.J., Jiménez, J. Boltzmann approach to lattice gas simulations. *Europhysics Letters*, **9**, 663–668, 1989.
[40] Higuera, F.J., Succi, S., Benzi, R. Lattice gas dynamics with enhanced collisions. *Europhysics Letters*, **9**, 345–349, 1989.
[41] Bhatnagar, P., Gross, E. and Krook, M. A model for collisional processes in gases I: small amplitude processes in charged and neutral one-component system. *Physical Review*, 94, 511, 1954.
[42] Chen, S., Chen, H., Martinez, D. O., Matthaeus, W. H. Lattice Boltzmann model for simulation of magnetohydrodynamics. *Physical Review Letters,* **67**, 3776, 1991.
[43] Qian, Y., d'Humie`res, D., Lallemand, P. Recovery of Navier–Stokes equations using a lattice-gas Boltzmann method. *Europhysics Letters,* **17**, 479, 1992.
[44] d'Humières, D., Bouzidi, M., Lallemand, P. Thirteen-velocity three-dimensional lattice Boltzmann model. *Physical Review E*, **63(6)**, 066702-066729, 2001.
[45] Keating, B., Vahala, G., Yepez, J., Soe, M., Vahala, L. Entropic lattice boltzmann representations required to recover navier-stokes flows. *Physical Review E*, **75(3)**, 036712-036723, 2007.
[46] Geier, M.C. AB Initio Derivation of de Cascaded Lattice Boltzmann Automaton. (Doctorado en Ciencias Aplicadas). University of Freiburg – IMTEK, Facultad de Ciencias Aplicadas, 2006. 165p.
[47] Succi, S. The Lattice Boltzmann Equation for Fluid Dynamics and Beyond. Oxford UK: Oxford University Press, 2001.
[48] Ziegler, D. P. Boundary conditions for lattice Boltzmann simulations. *Journal of Statistical Physics,* **71(5)**, 1171-1177, 1993.
[49] Zou, Q. and He, X. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Physics of Fluids*, **9(6)**, 1591-1598, 1997.
[50] Maier, R. S., Bernard, R. S., Grunau, D. W. Boundary conditions for the lattice Boltzmann method. *Physics of Fluids*, **8**, 1788-1802, 1996.
[51] Hecht, M., Harting, J. Implementation of on-site velocity boundary conditions for D3Q19 lattice Boltzmann simulations. *Journal of Statistic Mechanics*, **2010**, 01018, 2010.
[52] Artoli, A.M., Hoekstra, A., Sloot, P. Optimizing lattice Boltzmann simulations for unsteady flows. *Computers & Fluids*, **35(2),** 227–240, 2006.

[53] Holdych, D. Truncation error analysis of lattice Boltzmann methods. *Journal of Computational Physics*, **193(2),** 595–619, 2004.
[54] Junk, M., Kehrwald, D. On the relation between lattice variables and physical quantities in lattice Boltzmann simulations. [en linea]. En: *Sitio del departamento de Matemática y Estadística de la Universidad Konstanz.K*. 2006.
[55] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture – NVIDIA CUDA C Best Practice Guide.* [version electrónica]. 3.1. Santa Clara, CA, U.S.A. NVIDIA Corporation, 2010, 2010. [2010]. Disponible de: http://developer.nvidia.com/object/cuda_3_1_downloads.html.
[56] White F. M. Mecánica de Fluidos. Edición 1 en español. México: McGraw-Hill, 1988.
[57] Lammers, P., Küster U. Recent Performance Results of the Lattice Boltzmann Method. En: Proceedings of the High Performance Computing Center Stuttgart, (2006, Stuttgartm Alemania). High Performance Computing on Vector Systems 2006. Part 2. Berlin, Heidelberg, Alemania: Springer Verlag, 2007. pp. 51-59.
[58] Balay, S., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., Curfman McInnes, L. et al. *PETSc user manual*. 3.1. 2009. Argone National Laboratory. Disponible de: http://www.mcs.anl.gov/petsc/petsc-2/documentation/index.html
[59] Kuznik, F., Obrecht, C., Rusaouën, G., Roux, J.J. LBM based flow simulation using GPU computing processor. *Computers & Mathematics with Applications*, **59(7)**, 2380-2392, 2009.
[60] Yang, J.Y., Yang, S.C., Chen, Y.N., Hsu, C.A., Implicit weighted eno schemes for the three-dimensional incompressible navier-stokes equations. *Journal of Computational Physics*, **146(1)**, 464–487, 1998.
[61] Bailey, P., Myre, J., Walsh, S.D.C., Lilja, D.J. Saar, M.O., Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors. En: 2009 International Conference on Parallel Processing, (38, 2009, Vienna, Austria). ICPP-2009; Vienna, Austria: IEEE Computer Society, 2009. pp 550-557.
[62] Womersley, J.R. Method for the calculation of velocity, rate of flow and viscous drag in arteries when the pressure gradient is known. *The Journal of Physiology*, **127,** 553–563, 1955.
[63] Mazzeo, M.D., Coveney, P.V. HemeLB: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries. *Computer Physics Communications*, **178(12)**, 894-914, 2008.
[64] Tendler, J.M., Dodson, S. Fields, S. Le, H. Sinharoy, B. *IBM, POWER4 System Microarchitecture*. [sitio official]. IBM Server Group, 2001, 2010. Disponile en: http://www.hpcx.ac.uk/.

Listado de publicaciones

Rinaldi P., Dalponte D., Vénere M., Clausse A. Cellular automata algorithm for simulation of surface flows in large plains. *Simulation Modeling Practice and Theory*, **15**, 315-327, 2007.

Dalponte, D.D., Rinaldi, P.R., Cazenave, G., Usunoff, E., Varni, M., Vives, L., Vénere, M.J., Clausse, A. A validated fast algorithm for simulation of flooding events in plains. *Hydrological Processes*, **21**, 115-1124, 2007.

Rinaldi, P.R., García Bauza, C., Clausse, A., Vénere, M.J. Paralelización de Modelos de Simulación de Autómatas Celulares sobre Placas Gráficas. En: ENIEF (XVII, 2008, San Luis, Argentina). Mecánica Computacional, Vol. XXVII; San Luis, Argentina: Cardona, A., Storti, M., Zuppa, C. (eds.), 2008. pp. 2943-2957.

Rinaldi, P.R., Dari, E., Dalponte, D.D., Vénere, M.J., Clausse, A. Métodos de Lattice Boltzmann sobre GPU para Simulación de Fluidos. Comparación con NS mediante Elementos Finitos. En: ENIEF (XVIII, 2009, Tandil, Argentina). Mecánica Computacional, Vol. XXVIII; Tandil, Argentina: García Bauza, C., Lotito, P., Parente, L., Vénere, M.(eds.), 2009. pp. 273-286.

Rinaldi, P.R., Dari, E., Vénere, M.J., Clausse, A. Uso de GPUs para la Simulación de Fluidos en 3D con el Método de Lattice Boltzmann. En: MECOM (IX, 2010, Buenos Aires, Argentina). Mecánica Computacional, Vol. XXIX; Buenos Aires, Argentina: Dvorkin, E., Goldschmit, M., Storti, M. (eds.), 2010. pp. 7095-7108.