Universidad Nacional Autónoma de México

Facultad de Ciencias

Simulación numérica de flujos en ductos en procesadores gráficos de alto rendimiento

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
FÍSICO

PRESENTA:
CARLOS ECHEVERRÍA SERUR

DIRECTOR DE TESIS:
CARLOS MÁLAGA IGUIÑIZ

2011

Universidad Nacional Autónoma de México

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

A la memoria de mi padre.

Datos del Jurado

Datos del Alumno
Carlos Echeverría Seur
Facultad de Ciencias, UNAM
Física 40509250-3
+5215554567380

Datos del Tutor
Dr. Carlos Málaga Iguiñiz
Facultad de Ciencias, UNAM
Dep. Fluidos Complejos y Mecánica Estadística
+525556624970

Datos del Sinodal 1
Dr. Arturo Olvera Chávez
Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas, UNAM
Departamento de Matemáticas y Mecánica
+525556223560

Datos del Sinodal 2
Dr. Steven Czitrom Baus
Instituto de Ciencias del Mar y Limnología, UNAM
Dep. Sistemas Oceánicos y Costeros
+525556225792

Datos del Sinodal 3
Dr. Marcos Ley Koo
Facultad de Ciencias, UNAM
Departamento de Física
+525556224592

Datos del Sinodal 4
Dr. Francisco Madujano Sánchez
Facultad de Ciencias, UNAM
Dep. Fluidos Complejos y Mecánica Estadística
+525556224973

"It's more fun
to compute."
-Kraftwerk-

Resumen

El objetivo principal de esta tesis fue el de implementar un método numérico para simular el flujo tridimensional de un fluido haciendo uso del computo paralelo en tarjetas gráficas. El método numérico utilizado fue el método de lattice-Boltzmann D3Q15 y el código generado fue escrito en el lenguaje de programación en paralelo CUDA. El programa fue implementado haciendo uso de una tarjeta de video Nvidia GeForce GTX 260. Se hicieron simulaciones para validar el método con el flujo de un fluido generado por un gradiente de presiones a lo largo de un ducto cilíndrico, o flujo de Poiseuille, y con el flujo pulsátil a través de un ducto cilíndrico generado por un gradiente de presiones oscilante en el tiempo, o flujo de Womersley. En ambos casos se encontró que las soluciones numéricas aproximan las soluciones analíticas de dichos flujos. Una vez hecho esto se realizaron unas simulaciónes sin contraparte analítica, en particular la de un flujo oscilatorio a través de un ducto inmerso en un medio fluido. Se encontraron vórtices que viajan a la salida del ducto a largo del medio durante el transcurso del flujo, estos vórtices concuerdan cualitativamente con fenómenos observados en la realidad como puede ser visto en la generación de los llamadas "anillos de humo".

La ecuación de calor en dos dimensiones calculada con el método de Euler y el método de diferencias finitas al igual que el método de lattice Boltzmann D2Q9 fueron también implementados en los procesadores de alto rendimiento de la tarjeta de video para explorar las posibilidades y el desempeño brindados por el lenguaje de programación y sus distintos niveles de memorias. Se encontró que el uso de la memoria de texturas es el más eficiente para problemas con localidad en los datos, mientras que la memoria global y la memoria compartida alcanzan un mismo desempeño cuando se tratan de resolver dichos problemas. Es importante resaltar que un análisis del mejor tipo de memoria deberá ser hecho para cada método en específico. A lo largo de la tesis se puede encontrar una introducción al lenguaje de programación en paralelo CUDA con ejemplos detallados acerca de la utilización de las diferentes memorias.

Índice

1. Introducción **1**

2. Computo en Procesadores Gráficos **3**
2.1. GPU's como computadoras paralelas . . . 3
2.1.1. Arquitectura . . . 6
2.2. CUDA . . . 7
2.2.1. Paralelismo en los datos y estructura de programación . . 8
2.2.2. Memoria del dispositivo y transferencia de datos . . . 9
2.2.3. Kernels, Mallas, Bloques e Hilos . . . 10
2.2.4. Más sobre memorias . . . 13
2.3. Ejemplo: transferencia de calor . . . 15
2.3.1. Implementación de la ecuación de calor en CUDA . . . 16

3. Dinámica de Fluidos **23**
3.1. Fluidos . . . 23
3.2. Fuerzas actuando sobre un fluido . . . 25
3.3. Descripción de un flujo . . . 27
3.4. Conservación de masa: ecuación de continuidad . . . 29
3.5. Ecuaciones de Movimiento . . . 30
3.5.1. Flujos Newtonianos . . . 31
3.5.2. Ecuaciones de Navier-Stokes . . . 32
3.6. Dos soluciones a la ecuación de Navier-Stokes . . . 32
3.6.1. Solución de estado estacionario: flujo de Poiseuille . . . 32
3.6.2. Flujo pulsátil en un tubo: flujo de Womersley . . . 34

4. Método de Lattice Boltzmann **39**
4.1. La Ecuación de Boltzmann . . . 39
4.2. Discretización de la Ecuación de Boltzmann . . . 41
4.3. Deducción de la ecuación de lattice Boltzmann y su función de distribución en equilibrio . . . 42
4.3.1. Discretización del tiempo . . . 42
4.3.2. Cálculo de los momentos hidrodinámicos . . . 43
4.3.3. Aproximación de bajo numero de Mach . . . 43
4.3.4. Discretización del espacio fase . . . 44
4.3.5. Modelo 2-dimensional de 9 velocidades en una lattice cuadrada . . . 45
4.3.6. Modelo 3-dimensional de 15 velocidades en una lattice cuadrada . . . 46
4.4. Incompresibilidad . . . 48
4.5. Condiciones de Frontera . . . 51
4.5.1. Halfway Bounce-Back . . . 51
4.5.2. Esquema de extrapolación . . . 51
4.6. Simulaciones . . . 53

5. Validación y Resultados **56**
5.1. Flujo de Poiseuille 56
5.2. Flujo de Womersley 59
5.3. Ducto abierto 62
5.4. Flujo alrededor de una esfera 67

6. Conclusiones **70**

7. Apéndices **72**
7.1. Apéndice A. Instrucciónes de localización y copiado de datos . . 72
7.2. Apéndice B. Declaración de variables en los distintos tipos de memoria 76
7.3. Apéndice C. Utilización de la memoria de texturas 78
7.4. Apéndice D. Programas que resuelven la ecuacion de calor con memoria compartida y memoria de texturas 80
7.5. Apéndice E. Kernel Para resolver la ecuación de lattice Boltzmann con esquema D3Q15 85

Referencias **88**

1. Introducción

El uso de las unidades de procesamiento gráfico de las computadoras (GPU's, por sus siglas en inglés) como dispositivos capaces de hacer cálculos numéricos en paralelo ha ganado gran interés en los últimos años [11]. Muchas de las áreas del cómputo científico han encontrado mejorías en el desempeño de sus aplicaciones mediante el uso de esta técnica [16, 20], y debido a su gran abundancia y bajo costo (prácticamente cualquier computadora cuenta con un GPU) el uso favorable de esta tecnología puede traer grandes ventajas y posibilidades al cómputo científico con recursos limitados. El propósito de esta tesis consistió en entender el lenguaje de programación CUDA y el uso de las diferentes memorias dentro de una tarjeta gráfica para implementar un método numérico en el área de la dinámica de fluidos, capaz de correr en paralelo. En el mejor de los casos la mejora en el tiempo de ejecución fue mayor a un orden de magnitud respecto a un programa en serie.

La utilización de esta técnica en beneficio de la comunidad científica no había tenido mucho éxito debido a la dificultad de los lenguajes de programación existentes que se necesitaban aprender para manipular los GPUs hasta el año 2006. A partir de ese año, aparecen nuevos lenguajes inspirados en C que permiten programar los GPUs de forma análoga a como se programan los CPUs. Entre ellos se encuentra CUDA [20, 19, 18], que es una interfaz de programación al igual que una arquitectura especifica de GPUs que facilita la programación de las tarjetas de video. Este lenguaje libre desarrollado por NVIDIA para trabajar en sus tarjetas de video es muy similar al lenguaje de programación C y cuenta con unas extensiones que permiten manipular la tarjeta de video [19]. Actualmente, los GPUs están compuestos por cientos de microprocesadores capaces de manejar y calcular una cantidad elevada de datos de manera paralela, haciendo de la programación de GPUs con propósitos generales y mas específicamente con propósitos científicos, una alternativa de bajo costo dentro del cómputo en paralelo [3, 11]. Varios métodos numéricos son paralelizables, el método de lattice Boltzmann D3Q15 [6] fué el que se implementó para simular el flujo de un fluido en tres dimensiones debido a la simplicidad de su algoritmo y a la facilidad para incluir fronteras rígidas arbitrarias.

El método de lattice Boltzmann es un método en la dinámica de fluidos computacional [7] que permite aproximar soluciones a las ecuaciones de Navier-Stokes indirectamente a partir de la aproximación BGK de la ecuación de Boltzmann que surge en la teoría cinética de gases [25]. Dicho método se ha utilizado para simular fenómenos como la transferencia de calor, la turbulencia, la separación de fases al igual que muchos otros campos [11]. La discretización temporal y espacial de la ecuación de Boltzmann-BGK provee un método numérico paralelizable para aproximar la evolución de distribuciones de probabilidad de un gas de partículas que pueden ser relacionadas con cantidades macroscópicas como la presión y la velocidad de un fluido. Gracias a estas características y a la simplicidad del algoritmo, el método mostró ser un buen candidato para ser implementado en los GPUs.

Esta tesis está dividida en siete capítulos. Primero se presenta una intro-

ducción al cómputo en paralelo en tarjetas de video. Se presenta la arquitectura actual de las tarjetas de video y se muestra el avance que han tenido estos dispositivos en los últimos años. También se introducen las nociones principales que deben de entenderse para poder utilizar los GPUs con propósitos generales. El lenguaje de programación CUDA es presentado mostrando las estructuras generales que debe tener un código escrito en este lenguaje. Se explica en detalle los distintos tipos de memorias con las que cuentan los GPUs. De su entendimiento y el uso eficiente de ellas depende el rendimiento del método que se dese usar. La ecuación de calor en dos dimensiones servirá como ejemplo de la implementación de este lenguaje y nos permitirá aterrizar los conceptos que se presentan a lo largo de este capítulo.

En el siguiente capitulo el método de lattice Boltzmann se deriva de la ecuación de Boltzmann de la teoría cinética de gases. Ciertas restricciones sobre la función de distribución nos permiten recobrar las ecuaciones de Navier-Stokes en el límite macroscópico. Se discute la discretización espacial y temporal de la ecuación de Boltzmann y se presentaran dos esquemas, uno en dos dimensiones llamado D2Q9 y otro en tres dimensiones, D3Q15, del método de lattice Boltzmann. Se presentan también las condiciones de frontera para éste método, y se hace una breve discusión acerca de la propiedad de incompresibilidad en el método.

En el capítulo 4 se presentan las ecuaciones de movimiento para un flujo incompresible y las soluciones a los flujos de Poiseuille y Womersley con las que se validó el método implementado. En el siguiente capítulo se presentan las predicciones obtenidas por el método para estos flujos al igual que otras simulaciones de flujos tridimensionales. Estas simulaciones están inspiradas en flujos oscilantes a la salida de ductos presentes en el Sistema de Bombeo por Energía de Oleaje (SIBEO).

En el capítulo 6 se presentan las conclusiones de este trabajo. Por último, el capítulo 7 consta de un compendio de apéndices con detalles acerca del lenguaje de programación y las subrutinas más importantes que se utilizaron para hacer las simulaciones.

2. Computo en Procesadores Gráficos

Los microprocesadores basados en una única unidad central de procesamiento (CPU's) se encontraron con un rápido incremento en su desempeño por más de dos décadas. Este incremento disminuyó su aceleración alrededor del 2003 debido a problemas relacionados con el consumo de energía y la disipación de calor. Estos problemas representan una limitación en la frecuencia del reloj interno del procesador y las operaciones que pueden ser llevadas acabo durante un ciclo dentro de un CPU. Las compañías que desarrollan procesadores han cambiado a modelos en los cuales existen múltiples unidades de procesamiento, conocidas como núcleos, dentro de un mismo "chip" para incrementar el poder de procesamiento [16].

Típicamente, las aplicaciones de software en la actualidad están escritas como programas seriales. Debido a que los programas escritos de esta manera sólo pueden ser ejecutados por uno de los núcleos de procesamiento, los desarrolladores de software están actualmente adaptándose a las nuevas arquitecturas de múltiples núcleos para mejorar el desempeño de sus programas. En principio, el software que continuará disfrutando de mejorías en su desempeño con cada generación de microprocesadores será aquel que pueda trabajar en paralelo. En este tipo de software una tarea común es dividida en los llamados hilos de ejecución que cooperan para completar la tarea de manera más rápida. Esta nueva incentiva para el desarrollo de la programación en paralelo ha sido referida como la revolución de la concurrencia [21].

2.1. GPU's como computadoras paralelas

Desde el año 2003 la industria de los semiconductores se ha dividido en dos trayectorias para diseñar microprocesadores. La trayectoria de los "multicore" o multinúcleos busca mantener la velocidad de ejecución de programas seriales con la posibilidad de tener varios programas corriendo al mismo tiempo utilizando los diferentes núcleos de procesamiento. Los "multicores" comenzaron como procesadores de dos núcleos, con el número de núcleos aproximadamente duplicándose en cada generación de procesadores. Un ejemplo actual es el microprocesador Intel Core i7 que tiene cuatro núcleos de procesamiento, cada uno es un procesador capaz de implementar un conjunto completo de instrucciones complejas. El microprocesador está diseñado para maximizar la velocidad de ejecución de programas seriales.

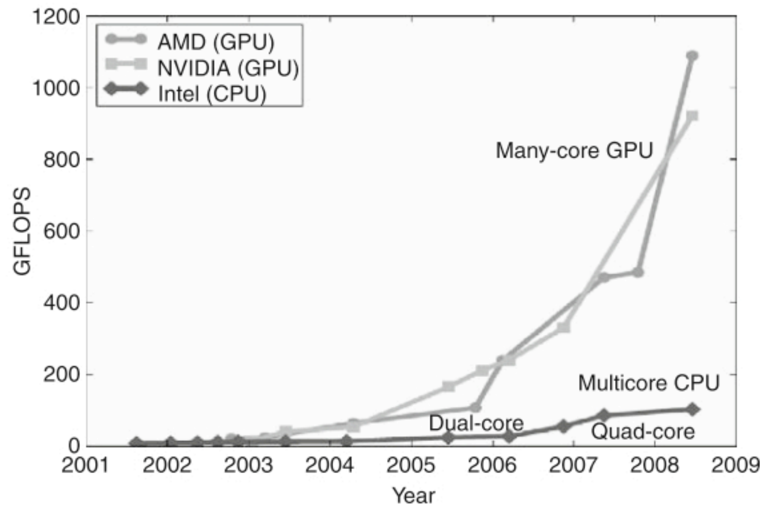


Figura 1: Brecha entre el desempeño de CPUs y GPUs. (Tomada de [16])

En contraste, la trayectoria de los "manycore" o varios-núcleos (GPU's) se enfoca principalmente en la velocidad de programas escritos en paralelo. Los "manycores" comenzaron como un gran número de núcleos más pequeños, y, una vez más, el número de núcleos se duplicó con cada generación. Un ejemplo relativamente actual es la unidad de procesamiento gráfica NVIDIA GeForce GTX 260 con 192 núcleos, cada uno es un pequeño procesador capaz de procesar una instrucción simple y comparte su control y sus instrucciones con otros siete núcleos. Los GPUs han liderado la carrera de desempeño por costo de operaciones flotantes por segundo entre las opciones de cómputo caseras desde el año 2003. Este fenómeno esta ilustrado en la Figura 1 en donde el eje y son GFLOPS (Giga FLoating point Operations Per Second). Mientras el aumento en el desempeño de procesadores "multicore" se ha frenado significativamente, los GPU's han continuado superándose con cada generación. En el año 2009, la razón entre la capacidad de cálculo entre los "multicore" y los "manycore" fué de 1 a 10 (estas no son necesariamente velocidades alcanzables, sino que son meramente velocidades que son potencialmente alcanzables en dichos procesadores).

Esta gran brecha en el desempeño de ejecuciones en paralelo han motivado a muchos desarrolladores de software a migrar las partes de su software que requieren cómputo intensivo para ser ejecutados en los GPUs. Estas "partes de cómputo intensivo" en los programas son el blanco de la programación en paralelo, es decir, cuando hay más trabajo que realizar, hay más oportunidad de dividir el código entre trabajadores paralelos cooperativos. La razón por la que existe esta brecha en el desempeño proviene de las diferencias en las filosofías fundamentales del diseño entre los dos tipos de procesadores. La diferencias en los diseños entre el CPU y el GPU están esquematizadas en la Figura 2.

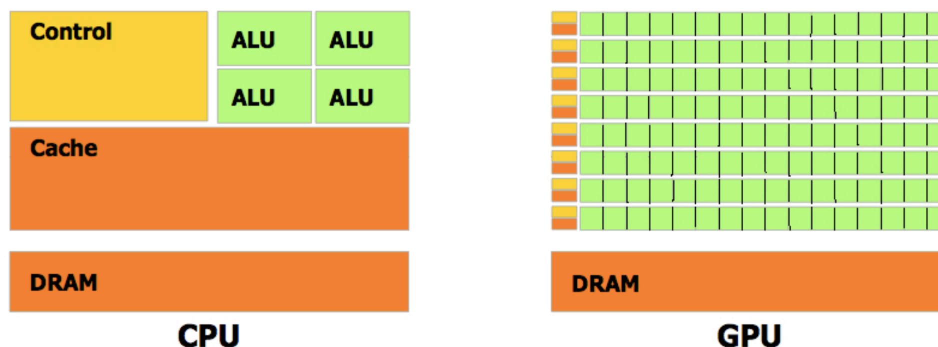


Figura 2: Diferencias en el diseño de CPUs y GPUs. (Tomada de [16])

El diseño de un CPU esta optimizado para el desempeño de código serial, mientras que el diseño de los GPUs se enfoca en el desempeño del código paralelo. En un CPU de cuatro núcleos tenemos cuatro unidades aritméticas lógicas (ALUs) que pueden hacer una secuencia de tareas complicadas sobre una gran cantidad de datos. Por otro lado, los GPUs están diseñados para tener cientos de ALUs que pueden hacer tareas sencillas sobre una cantidad reducida de datos. Esto aumenta la razón de operaciones aritméticas por movimientos en memoria, haciendo que las tarjetas gráficas tengan el desempeño mostrado en la Figura 1.

El rápido avance en el desarrollo de los GPUs está motivado por la creciente demanda en animación por computadora que requieren las industrias de las películas, los videojuegos, y el diseño gráfico. Estas industrias ejercen una tremenda presión económica por tener la habilidad de realizar un número masivo de cálculos de punto flotante por cuadro de video para el despliegue gráfico. Esta demanda motiva a los productores de GPUs a buscar maneras de maximizar el área de chips dedicados a cálculos de punto flotante, ya que para representar un cuadro de video es necesario especificar las cantidades de colores (en punto flotante) para cada pixel del cuadro.

Es necesario enfatizar que los GPUs están diseñados como máquinas de cómputo numérico que no tendrán un buen desempeño en tareas en las que el CPU esta diseñado para tener un buen desempeño. Por lo tanto, uno debe esperar que la mayoría de los programas utilicen ambos el CPU y el GPU, ejecutando partes seriales en el CPU y partes con una alta intensidad numérica en los GPUs. Esta es la principal razón por la cual el modelo de programación CUDA (Compute Unified Device Architecture), introducido por NVIDIA en el 2007, al igual que el lenguaje de programación OpenCL estén diseñados para implementar una ejecución conjunta GPU/CPU en los programas escritos en estos lenguajes.

2.1.1. Arquitectura

Motivado por la insaciable demanda en el mercado de gráficos en 3D en alta definición, el GPU ha evolucionado en un procesador altamente paralelo. La Figura 3 muestra la arquitectura de un GPU moderno capaz de implementar CUDA. Esta organizado en un arreglo de procesadores múltiples (single multiprocessors o SM's por sus siglas en inglés) capaces de ejecutar la misma subrutina en cada uno de sus procesadores (streamming processors o SPs por sus siglas en inglés) al mismo tiempo. Cada SP ejecuta la misma subrutina pero sobre un grupo diferente de datos, a este camino de ejecución se le conoce como "hilo". Siempre que el método lo permita, esto lleva a un trabajo colectivo entre los SPs para llevar a cabo una tarea en menor tiempo.

En la Figura 3, dos SMs conforman un bloque. El numero de SMs en los bloques puede variar de una generación de GPUs a la siguiente. También, cada SM en la Figura 3 tiene 8 SPs que comparten lógicas de control y cache de instrucciones. Cada GPU tiene memoria DRAM de hasta 6 gigabytes de tipo GDDR (graphics double data rate), conocida como "memoria global". Esta memoria global se usa comúnmente para almacenar los datos que representan la imagen que despliega el monitor. Para aplicaciones gráficas, es utilizada para tener imágenes de video, e información de texturas para su representación en 3D, pero para el cómputo científico funciona como una memoria con un alto ancho de banda que se encuentra sobre el GPU. Existen también otros tipos de memoria además de la memoria global, que son llamadas la "memoria compartida" (shared memory), la "memoria constante" (constant memory) y la "memoria de texturas" (texture memory), de ellas se hablará más adelante.

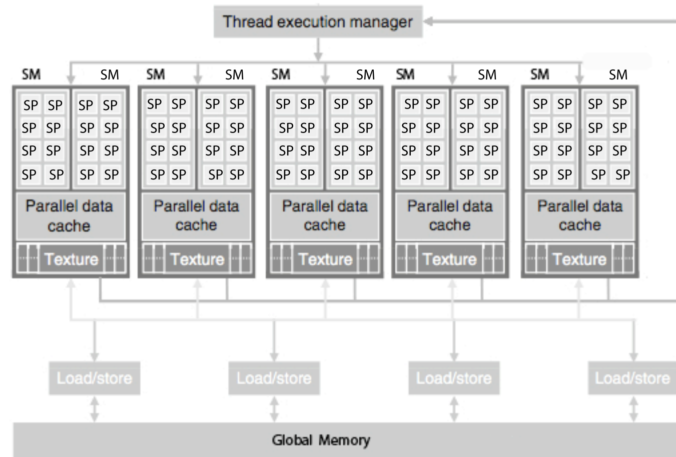


Figura 3: Arquitectura de un GPU capaz de implementar CUDA. (Tomada de [16]).

En particular, la tarjeta de video GTX 260 tiene 192 SPs (24 SMs, cada uno con 8 SPs). Cada SP tiene una unidad de suma-multiplicación MAD por sus siglas en inglés. Además de unidades especiales de cálculo que realizan funciones de punto flotante como la raíz cuadrada (SQRT) y cálculo de funciones trascendentes. Típicamente un GPU puede procesar del orden de 5000-12,000 hilos por paso de tiempo. El nivel de paralelismo que pueden lograr los GPUs está incrementándose rápidamente, es decir, cada nueva generación de GPUs está incrementando el número de SM's y SP's que contiene. Es muy importante tratar de buscar estos altos niveles de paralelismo cuando se desarrolla softwae en paralelo para ser implementado en los GPUs.

2.2. CUDA

Muchos lenguajes de programación en paralelo han sido propuestos en las últimas décadas. Los lenguajes que son más comúnmente utilizados son el "Message Passing Interface" (MPI por sus siglas en inglés) para cómputo escalable en clusters, y OpenMP para sistemas de multiprocesadores con memoria compartida. MPI es un modelo en el cual los nodos computacionales de un cluster no comparten memoria; toda la interacción y el compartir información debe ser hecho a través de mensajes pasados explícitamente. MPI ha logrado ser útil en el cómputo científico; programas escritos en MPI han corrido satisfactoriamente en clusters con más de 100,000 nodos [20]. Sin embargo, el esfuerzo requerido para portar un programa hacia MPI puede ser extremadamente alto debido a la falta de memoria compartida entre los nodos computacionales. Los GPU's, por

otro lado, proveen memoria compartida para ejecución en paralelo para sobrepasar esta dificultad. En torno a la comunicación $CPU \leftrightarrow GPU$, actualmente CUDA provee una capacidad de copiado muy limitada. Los programadores necesitan manipular las transferencias de datos entre el GPU y el CPU. Muchos aspectos de CUDA son similares a ambos, MPI y OpenMP en el sentido que el programador maneja las construcciones del código paralelo, aunque los compiladores de OpenMP hacen más trabajo en la automatización en administrar la ejecución en paralelo.

Recientemente, muchas de las compañías mas influyentes en la industria del cómputo, como Apple, Intel, AMD/ATI, y NVIDIA, conjuntamente han desarrollado un modelo de programación estandarizado llamado OpenCL. Muy similar a CUDA, el modelo de programación de OpenCL define extensiones del lenguaje y API's (Aplication Programming Interface) para permitir a los programadores manejar el paralelismo y la entrega de datos a procesadores masivamente paralelos. Por un lado en OpenCL se pueden desarrollar programas que corran en cualquier GPU o en cualquier procesador "multicore", sin embargo el lenguaje de programación es un poco más complicado que CUDA, y en las tarjetas NVIDIA los programas escritos en CUDA son más rápidos.

2.2.1. Paralelismo en los datos y estructura de programación

En el lenguaje de CUDA, el sistema de computo consiste de un "host" u ordenador principal, que es un CPU tradicional, y uno o mas "devices" o dispositivos, que son los GPUs. Actualmente existen varios métodos numéricos que presentan una gran cantidad de paralelismo en los datos, una propiedad que permite realizar las mismas operaciones aritméticas sobre distintos datos de manera simultanea. Es decir, pueden hacer uso del estilo de programación en paralelo llamado SPMD o "single-program multiple-data", en el cual se puede aplicar el mismo conjunto de operaciones sobre distintos datos sin que el resultado de uno dependa del resultado de otro. Esta independencia en los datos es la base de el computo paralelo en las tarjetas de video. Un programa escrito en CUDA consiste de una o mas fases que son ejecutadas sobre el host (CPU) y otras fases que son ejecutadas sobre el device (GPU). Las fases que exhiben poco o ningún paralelismo en los datos son implementadas en el CPU, mientras que las fases que exhiben una gran cantidad de paralelismo en los datos son implementadas en el GPU. El código del host es codigo escrito en el lenguaje de programación C; esta compilado por el compilador de C estándar y corre como cualquier otro proceso del CPU. El código del device es escrito también utilizando C extendido con instrucciones que permiten identificar las funciones paralelas que serán ejecutadas en el GPU y sus variables asociadas. A la subrutina que será ejecutada por todos los hilos durante una fase paralela del programa se le llama kernel.

La ejecución de un programa típico de CUDA se ilustra en la Figura 4. La ejecución comienza con actividad en el CPU. Cuando se invoca a un kernel, la ejecución se transfiere al GPU, en donde un gran numero de hilos son generados para trabajar con los datos de manera paralela. Todos los hilos que son generados por el kernel durante el llamado son colectivamente llamados "grids" o mallas y

están agrupados en bloques de hilos, un SM se encargará de uno o más bloques. En la Figura 4 se muestra la ejecución de dos mallas de hilos. Cuando todos los hilos de un kernel completan su ejecución, la malla correspondiente termina, y la ejecución se transfiere nuevamente al host hasta que otro kernel sea llamado.

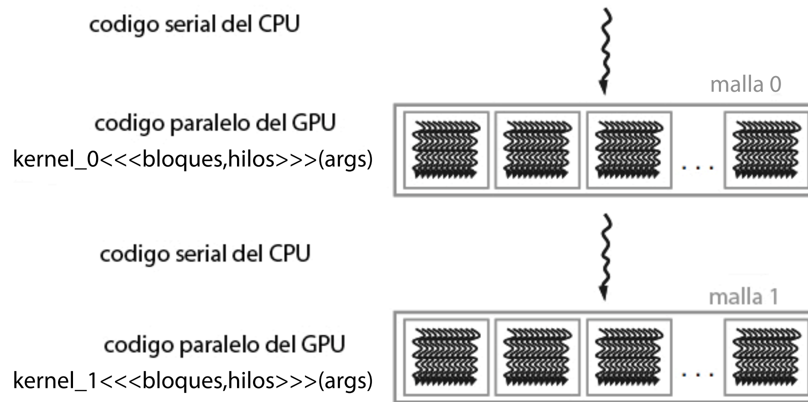


Figura 4: Ejecución de un programa escrito en CUDA. (Tomada de [16])

2.2.2. Memoria del dispositivo y transferencia de datos

En la arquitectura de los GPUs, el "host" y el "device" tienen espacios de memoria separados. Para poder ejecutar un kernel en el device, el programador necesita localizar memoria en el device, y transferir los datos pertinentes de la memoria del host a la memoria localizada del device. De manera similar, después de ejecutar un kernel de manera única o iterativa, el programador necesita transferir los datos resultantes de la memoria del device a la memoria del host, o bien puede ejecutar otro kernel sobre los datos generados por el primer kernel y hacer la copia una vez que el usuario requiera trabajar en el CPU con los datos procesados en el GPU. En la Figura 5 se muestra la estructura básica de un programa que hace uso del GPU. En las etapas 1 y 2 del programa se localiza la memoria del GPU que guardará copias de los datos del CPU y se copian dichos datos a la memoria del GPU. En la etapa 3 se llama a la ejecución en paralelo del kernel, a partir de aquí es donde el GPU trabajará con los datos que han sido enviados desde el CPU. Después, en la etapa 4, se ejecutan las operaciones dictadas por el kernel que típicamente terminan con la escritura de los resultados en la memoria global del GPU. En la etapa 5 se puede reiniciar el mismo kernel sobre los datos resultantes del primer llamado o se puede ejecutar otro kernel antes de regresar los resultados al CPU. Por último, en el paso 6 se copia el resultado de los cálculos a la memoria del CPU. Aquí es importante mencionar que es el CPU el que dirige todos estos procesos, es el CPU el que manda ejecutar los kernels y el que puede hacer copias dentro de la memoria global del GPU. Un GPU no puede trabajar independientemente del CPU. También hay que notar que el proceso que más tiempo lleva, es la transferencia de datos

entre la memoria RAM del CPU y la memoria global del GPU. Uno debe procurar programar haciendo la mínima transferencia posible de datos entre estas memorias.

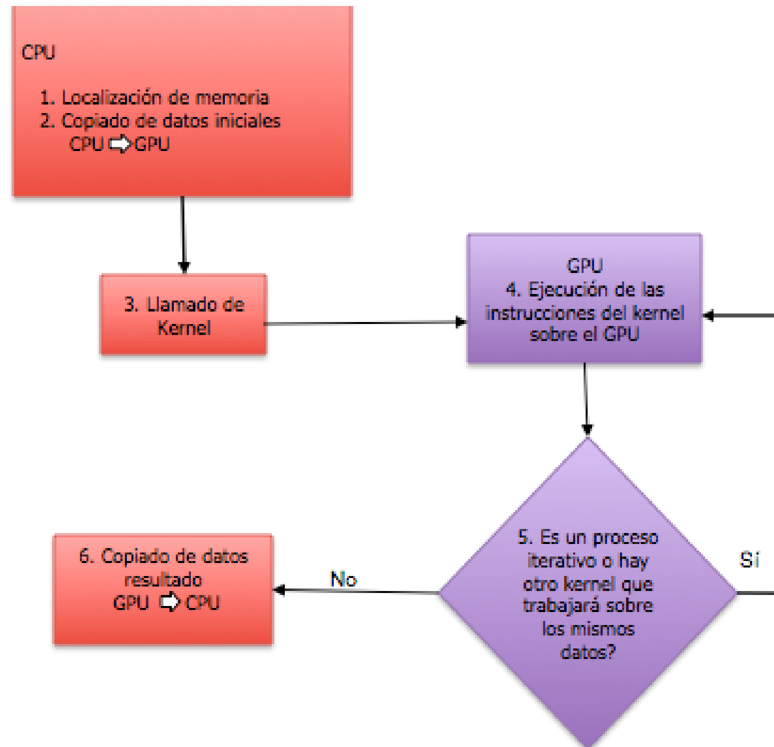


Figura 5: Estructura general de un programa.

La localización de memoria, la transferencia de datos entre las memorias del CPU y el GPU así como la copia de datos dentro del GPU son instrucciones que están fuera de la subrutina que consideramos el kernel. En ese sentido, estas instrucciones son parte del código que ejecuta el CPU. Estas instrucciones asociadas a la manipulación de datos y su descripción se encuentran en el Apéndice A de esta tesis.

2.2.3. Kernels, Mallas, Bloques e Hilos

Hablemos ahora de manera más específica acerca de las funciones kernel y de los efectos de llamar a estos kernels. En CUDA, una kernel especifica el código que será ejecutado por todos los hilos durante una fase paralela del programa. La Figura 6 muestra una función kernel típica. La sintaxis es C con algunas extensiones notables. Primero, hay una palabra exclusiva de CUDA

"__*global*__" frente a la declaración de la función $kernel()$. Esta palabra indica que la función es un kernel y que puede ser llamada desde una subrutina típica del host para generar una malla de hilos en el dispositivo.

```
27  ...
28  __global__ void kernel(float* Md, float* Nd, float* Pd, int Dim)
29  {
30      // Indices 2D
31      int tx=threadIdx.x;
32      int ty=threadIdx.y;
33      ...
34      método numérico
35      ...
36  }
37  ...
```

Figura 6: Ejemplo de la estructura de un kernel típico.

En general, CUDA extiende las declaraciones de funciones de C con palabras clave que funcionan como identificadores. La palabra __*global*__ indica que la función que esta siendo declarada es una función kernel de CUDA. La función será ejecutada en el dispositivo y solamente puede ser llamada desde el CPU para general una malla de hilos en el GPU.

Otras extensiones al lenguaje C que se muestran en la Figura 6 son las palabras clave $threadIdx.x$ y $threadIdx.y$ que se refieren a los índices de cada hilo. Ya que todos los hilos ejecutan el mismo código, estas palabras permiten distinguir a los hilos unos de otros y dirigirse hacia las partes de los datos particulares para las cuales están diseñados para trabajar. Estas palabras son variables predefinidas que permiten a todo hilo accesar a sus coordenadas específicas dentro de una malla. Hilos distintos verán valores distintos en sus variables $threadIdx.x$ y $threadIdx.y$. Estas coordenadas reflejan una organización multidimensional para los hilos. Cuando un kenel es llamado, o invocado, éste es ejecutado como una malla de hilos paralelos. Los hilos en una malla están organizados en una jerarquía de dos niveles como se muestra en la Figura 7. Por simplicidad, se muestran solamente un pequeño número de hilos. En el nivel más alto de la jerarquía, cada malla consiste de uno o más bloques de hilos. Todos los bloques dentro de una malla tienen el mismo número de hilos. En la Figura 7, el llamado al Kernel 1 crea la Malla 1. Cada malla de hilos está compuesta típicamente por miles o millones de hilos. La creación de suficientes hilos para utilizar al máximo las capacidades del hardware generalmente necesita una gran cantidad de datos. La Malla 1 está organizada como un arreglo de 2×2 de 4 bloques. Cada bloque tiene una coordenada bidimensional única dada por las palabras clave $blockIdx.x$ y $blockIdx.y$. Todos los bloques deben tener el mismo número de hilos organizados de la misma manera. Cada bloque, a su vez, es organizado como un arreglo tridimensional de hilos con un tamaño total de hasta 1024 hilos dependiendo de la tarjeta de video. Las coordenadas de los hilos en un bloque están unívocamente definidas por tres índices: $threadIdx.x$, $thradIdx.y$,

y $thradIdx.z$. No es necesario utilizar las tres dimensiones de los bloques para todos los programas. En la Figura 7 cada bloque es organizado en un arreglo tridimensional de hilos con dimensiones $4 \times 4 \times 2$. Esto le da a la Malla 1 un tamaño total de $4 \times 16 = 64$ hilos.

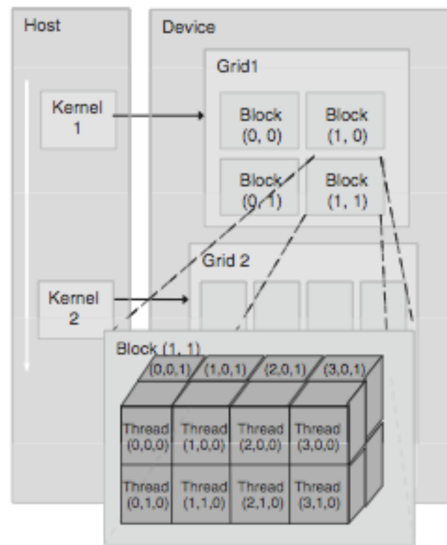


Figura 7: Organización de hilos en CUDA. (Tomada de [16]).

Cuando el código del host invoca a un kernel, éste fija las dimensiones de la malla y de los bloques a través de unos parámetros conocidos como "configuración de ejecución". Esto se hace a través de dos estructuras, la primera es para describir la configuración de bloques, mientras que la segunda define la configuración de la malla. En el ejemplo de la Figura 8 tenemos solamente un bloque (1×1) en cada malla. La ultima línea de código invoca al kernel. La sintaxis especial entre el nombre del kernel y los parámetros tradicionales de la función es una extensión de C que provee las dimensiones de la malla en términos del número de bloques y la dimensión de los bloques en términos de hilos.

```
41  // Se define la configuración de ejecución
42  dim3 dimBlock(Dim, Dim);
43  dem3 dimGrid(1,1);
44
45  //Llamado al kernel y generación de hilos
46  kernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Dim);
```

Figura 8: Parte de un código que hace una llamada a un kernel

2.2.4. Más sobre memorias

Hemos visto la estructura general que se necesita para escribir un kernel en CUDA y que sea ejecutada por un número masivo de hilos. Los datos que deben ser procesados por estos hilos son transferidos desde la memoria del host hacia la memoria global del dispositivo. Los hilos pueden accesar a su porción de los datos de la memoria global utilizando sus identificadores de bloque y de hilo. Sin embargo, la lectura de la memoria global no es la manera más rápida de accesar a los datos. El bajo desempeño se debe al hecho que la memoria global, que es típicamente implementada con memoria de acceso aleatorio dinámico (DRAM por sus siglas en inglés) tiende a tener largos tiempos de acceso (cientos de ciclos de reloj) y un ancho de banda limitado. Aunque el tener varios hilos disponibles para ser ejecutados puede tolerar estos tiempos de acceso, es fácil encontrarse con situaciones en las cuales la congestión del tráfico en los caminos de acceso a la memoria global permita que solamente pocos hilos progresen, causando que algunos de los multiprocesadores SMs estén ociosos. Para aligerar esta congestión, existen otros métodos de acceso a la memoria que pueden reducir el número de veces que esta memoria global es accesada.

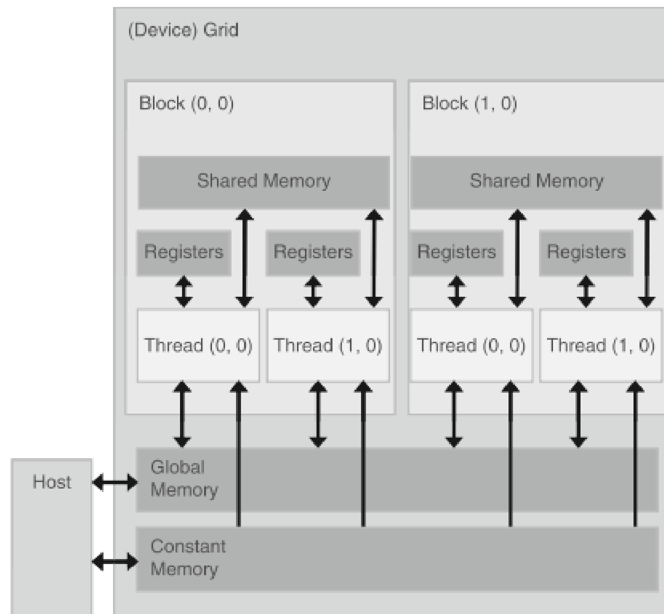


Figura 9: Esquema de memorias de un GPU. (Tomada de [16]).

Los GPUs tienen varios tipos de memorias que pueden ser utilizadas por los programadores para lograr velocidades altas de ejecución de los kernels en función del método que se quiera implementar. La Figura 9 muestra estas memorias. En la parte más baja de la imagen podemos ver la memoria global y

la memoria constante. Estos tipos de memoria pueden ser escritos y leídos por el host al hacer uso de las funciones del API mostradas en el Apéndice A. La memoria constante es una memoria de corto tiempo de acceso, alto ancho de banda, y solamente de lectura cuando varios hilos accesan la misma dirección simultáneamente. Los registros y la memoria compartida en la Figura 9 son memorias que se encuentran físicamente sobre el SM. Las variables que residen en estos tipos de memoria pueden ser accesados a una velocidad muy alta y de manera altamente paralela. Los registros están localizados para cada hilo individualmente; cada hilo puede accesar solamente sus propios registros. Un kernel típicamente usa registros para guardar variables accesadas frecuentemente y que son definidas por cada hilo. La memoria compartida se localiza para los bloques de hilos; todos los hilos dentro de un bloque pueden accesar a datos que se han localizado en esta memoria. La memoria compartida es una manera eficiente de cooperar entre hilos al compartir sus datos iniciales y los resultados parciales de su trabajo. Al declarar una variable en alguno de los tipos de memoria, el programador dicta la visibilidad y la velocidad de acceso a esta variable. En el Apéndice B se puede encontrar cómo declarar los distintos tipos de variables para que residan en las diferentes memorias.

Existe un último tipo de memoria de lectura, llamada "memoria de textura", que puede mejorar el desempeño y reducir el tráfico de memoria cuando las lecturas tienen ciertos patrones de acceso. Aunque la memoria de textura fue diseñada para programas gráficos tradicionales, esta memoria también puede ser utilizada de manera eficiente para algunos programas con propósitos generales. Al igual que la memoria compartida, la memoria de texturas también está "chacheada" sobre el SM, de tal manera que en algunas situaciones logrará un mejor desempeño en comparación a los accesos a la memoria global fuera del SM. Específicamente, los caches de texturas están diseñados para aplicaciones gráficas en donde los patrones de acceso a la memoria exhiben una alta "localidad espacial". En un programa de cómputo general, esto implica que un hilo es probable que lea de una dirección "cercana" a la dirección que los hilos vecinos leen, como se muestra en la Figura 10.

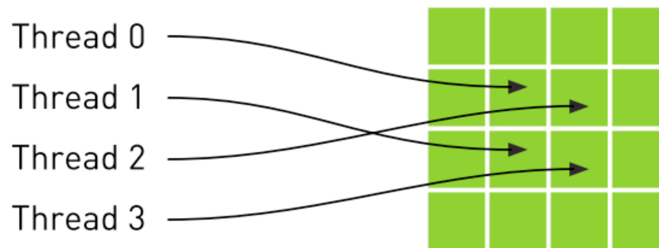


Figura 10: Mapeo de hilos a una región bidimensional de memoria. (Tomada de [20])

Cuando un hilo lee de la memoria global, y los datos a los que necesita acce-

sar no se encuentran unos a lado de otros, se necesitan hacer varios accesos a la memoria. Por ejemplo, en la Figura 10, las cuatro direcciones que se muestran no son consecutivas, así que no estarían guardadas conjuntamente en la memoria global. La memoria de texturas, sin embargo, esta diseñada para acelerar este tipo de patrones de acceso. Esta memoria almacena los datos que requiere de manera contigua de tal manera que solamente se necesitará hacer un acceso a esta memoria. De esta manera, se verá un incremento en el desempeño al utilizar memoria de texturas en vez de memoria global. De hecho, estos patrones de acceso son muy comunes en el computo científico, por ejemplo en el método de diferencias finitas. Así, la memoria de texturas es una manera de guardar datos que en la memoria global se encuentran lejanos unos de otros. De hecho la memoria de texturas está optimizada para trabajar en dos dimensiones, pero se pueden crear texturas de una, dos y hasta tres dimensiones. Para utilizar este tipo de memoria no existen palabras claves que funcionen como calificadores de las variables como en los otros tipos de memoria, sino que existe otra manera para definir variables de tipo textura. Dicha manera es más complicada, y utiliza distintos elementos que nos provee CUDA, como "cudaArrays", "channel descriptors", y distintas estructuras que funcionan como arreglos de parámetros para el uso de la memoria de textura. En el Apéndice C se encuentra una explicación detallada de dicha manera y sus instrucciones.

En resumen, CUDA provee registros, memoria compartida, memoria constante y memoria de texturas que pueden ser accesados a una mayor velocidad y de manera más eficiente que la memoria global. La utilización de estas memorias eficazmente muy probablemente requiera de una modificación en el algoritmo utilizado. Es también importante tener en cuenta los tamaños limitantes de estos tipos de memorias. Una vez que sus capacidades han sido excedidas, estas memorias se convierten en factores limitantes de la mejoría en el desempeño de cualquier programa. Es importante resaltar que la memoria que brindará un mejor desempeño está en función del método numérico que quiera ser implementado; distintos métodos encontrarán un mejor desempeño utilizando distintas memorias.

2.3. Ejemplo: transferencia de calor

Para mostrar muchos de los conceptos que se han presentado y tratado en este capítulo, se expondrá un problema sencillo que será resuelto utilizando los GPUs. El problema físico que se expondrá es el de la transferencia de calor en dos dimensiones. La ecuación a resolver es la siguiente:

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \tag{1}$$

$$0 < x < l_x$$

$$0 < y < L_y$$

$$t \geq 0$$

en donde $T = T(x, y, t)$ representa el campo de temperaturas y es la variable dependiente, y α es el coeficiente de difusividad térmica. Para resolver esta ecuación, es necesario especificar las condiciones de frontera en $x = 0, L_x$ y en $y = 0, L_y$ y las condiciones iniciales en $t = 0$. Las condiciones de frontera utilizadas son:

$$\nabla T(0, y, t) \cdot \mathbf{n} = 0, \;\; \nabla T(1, y, t) \cdot \mathbf{n} = 0$$

$$\nabla T(x, 0, t) \cdot \mathbf{n} = 0, \;\; \nabla T(x, 1, t) \cdot \mathbf{n} = 0$$

$$T(x, y, 0) = T_0(x, y)$$

En donde $\mathbf{n}$ es el vector normal a la frontera, esto corresponde a condiciones de frontera adiabáticas, que dan lugar a que el flujo de calor en las fronteras sea igual a cero. También hemos escogido a $T_0(x, y) = 0$ para todo el dominio excepto por el punto medio $(x/2, y/2)$ en donde habrá una fuente de calor a temperatura constante.

Escogiendo una malla regular en donde $\triangle x$ y $\triangle y$ son la distancia local entre dos puntos en el espacio con $\triangle x = \triangle y$, y $\triangle t$, el paso temporal. La fórmula específica para encontrar la temperatura en los distintos pasos de tiempo está dada por,

$$T_{i,j,n+1} = T_{i,j,n} + \triangle t \cdot \alpha \cdot \left[\frac{(T_{i+1,j,n} - 2T_{i,j,n} + T_{i-1,j,n})}{\triangle x^2} + \frac{(T_{i,j+1,n} - 2T_{i,j,n} + T_{i,j-1,n})}{\triangle y^2} \right]. \tag{2}$$

en donde los subíndices i y j representan las coordenadas espaciales y el subíndice n representa un paso en el tiempo. Esta es la forma discreta de la ecuación de calor en dos dimensiones con constante de difusividad térmica α.

2.3.1. Implementación de la ecuación de calor en CUDA

Para poder resolver este problema es necesario notar que la ecuación (2) nos permite calcular el campo de temperaturas para cada punto en un intervalo de tiempo fijo de manera independiente a todos los otros puntos. Para hacer esto se pueden construir dos arreglos que contendrán el campo de temperaturas, el primero *t_data_old* será nuestro campo de temperaturas al tiempo t y el segundo *t_data* será nuestro campo de temperaturas después de aplicar una iteración de la ecuación (2), es decir será nuestro campo de temperaturas al tiempo $t + 1$. Podemos darnos cuenta de que para calcular el valor del campo de temperaturas para un tiempo dado únicamente necesitamos de los valores de la temperatura de ciertos nodos vecinos en un mismo tiempo anterior. Es decir, aquí se exhibe un paralelismo en los datos; cada punto en el espacio representado por un nodo computacional puede calcular su valor para un tiempo

posterior independientemente del resultado de sus vecinos. Esto nos permite resolver el problema de manera paralela. Así, podemos seccionar el dominio físico de nuestra ecuación en bloques computacionales con dimensiones convenientes. Cada uno de estos bloques resolverá la ecuación (2) al mismo tiempo por los distintos SMs de la tarjeta de video. El bloque 1 será enviado a un SM en donde cada nodo será calculado por un SP, mientras que el bloque 2 será enviado a otro SM en el mismo GPU. Esto se muestra esquematizado en la Figura 11.

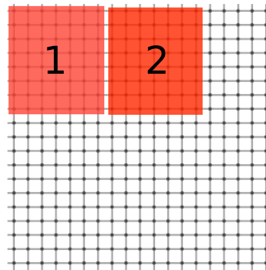


Figura 11: Esquema de seccionamiento del dominio físico y computacional.

En las Figuras 12 y 13 se muestra el cuerpo principal de nuestro programa, la función $main()$. En esta parte del código se refleja la estructura general de cualquier programa que utilice CUDA. Podemos ver que a lo largo del cuerpo de la función $main()$ se implementan los pasos esquematizados en la Figura 5 de la sección 2.2.2 de este capítulo. Primero se le dan valores a variables que definirán a nuestra malla en términos de $\triangle x, \triangle y$, y $\triangle t$ al igual que el valor de nuestra constante de difusividad térmica en la variable $kcond$, el número de puntos de la malla está representado por ni y nj, el número de nodos en la dirección x y y respectivamente. En la línea 57 esta implementada la localización de memoria para nuestra variable $t0$ que representa el campo de temperaturas en todo el dominio, es decir, es nuestra variable dependiente. Esta variable está localizada en un espacio de memoria en el CPU y se utiliza la sintaxis clásica de C para localizarla. Inmediatamente después, se encuentra la localización de memoria global en el GPU. Estas dos variables localizadas en el GPU representarán también el campo de temperaturas; t_data_old es el campo de temperaturas en un determinado paso de tiempo, mientras que t_data representa el campo en un intervalo de tiempo siguiente, después de haber implementado un paso de la ecuación (2). Para este paso se utiliza la instrucción de CUDA para la localización de memoria explicada en el Apéndice A de esta tesis. Después de inicializar nuestro campo de temperaturas en el CPU se encuentra la transferencia de datos desde el CPU hacia el GPU con la función $cudaMemcpy2D()$, una función que pertenece a las extensiones de CUDA del lenguaje C. Después se encuentra el bucle iterativo que llamara un número de veces finito a nuestra función $ec_calor()$ en la cual se encuentra el llamado al kernel que resolverán la ecuación en el GPU de manera paralela. Una vez terminado el ciclo iterativo se encuentra el llamado a la función $Imprimir()$ la cual copiará los resultados

desde el GPU de regreso hacia el CPU para poder ser manipulados. Como podemos ver, en el cuerpo principal de nuestro programa se encuentra reflejada la estructura de localización de memoria y copiado de datos que es necesaria para implementar un código en los procesadores gráficos.

```
42  int main(void)
43  {
44      int i;
45      int totpoints;
46
47      dt = 0.01f;
48      dx = 0.1f;
49      dy = 0.1f;
50      kcond = 0.01f;
51      ni=640;
52      nj=640;
53      totpoints = ni*nj;
54
55      printf ("ni = %d\n", ni);
56      printf ("nj = %d\n", nj);
57      printf ("Numero de puntos = %d\n", totpoints);
58
59      // Asigna la memoria en el CPU (host)
60      t0 = (float *)malloc(ni*nj*sizeof(float));
61
62      // Asigna la memoria en el GPU (device)
63      cudaMallocPitch((void **)&t_data, &pitch, sizeof(float)*ni, nj);
64      cudaMallocPitch((void **)&t_data_old, &pitch, sizeof(float)*ni, nj);
65
66      // Valores iniciales del campo t
67      for (i=0; i<totpoints; i++) {
68      t0[i] = 0.f;
69      }
70      t0[totpoints/2 + ni/2] = 100.f;
71
72      // Copia valores iniciales al GPU
73      cudaMemcpy2D((void *)t_data, pitch, (void*) t0,
...     sizeof(float)*ni,sizeof(float)*ni, nj,cudaMemcpyHostToDevice);
74
75      paso = 0;
76
77      for (i=1;i<=10000;i++){
78      paso = paso + 1;
```

Figura 12: Función Main Parte 1

```
79        ec_calor();
80        if (paso%1000 == 0) printf ("Iteracion: %d\n", paso);
81        }
82
83        Imprimir();
84
85        return 0;
86    }
```

Figura 13: Función Main Parte 2

A continuación, en la Figura 14 se muestra el cuerpo de la función $ec_calor()$. En esta función se encuentra una copia de memoria $GPU \rightarrow GPU$ entre las variables t_data y t_data_old para actualizar el campo de temperaturas entre un paso de tiempo y el siguiente. Después de esta transferencia de información, se encuentra el punto en el que definiremos el número de hilos que serán generados al invocar nuestro kernel. El número de hilos que ejecutarán el programa esta dictado por la definición de dos variables: $grid$ y $block$, en donde $grid$ define las dimensiones de la malla de bloques y $block$ define las dimensiones del bloque de hilos. Estas variables son utilizadas a continuación al invocar al kernel $ec_calor_kernel()$. Podemos ver que antes de los parámetros de la función, se encuentra el numero de hilos que llevarán a cabo los cálculos que estén en el código del kernel. Esto es hecho al definir el tamaño de la malla y de los bloques con las variables $grid$ y $block$. El llamado a este kernel será hecho cada paso de tiempo ya que dicho llamado se encuentra dentro de la función $ec_calor()$ y cada vez que este kernel sea llamado se generarán el mismo número de hilos para realizar los cálculos en paralelo.

```
94    void ec_calor(void)
95    {
96        // Copiado de t_data a t_array y "Bind" de t_array a la textura
97       cudaMemcpy2D((void *)t_data_old, pitch, (void *)t_data,sizeof(float)*ni,
98                                sizeof(float)*ni, nj, cudaMemcpyDeviceToDevice);
99
100       // Definición de las dimensiones de la Malla de hilos
101       dim3 grid = dim3(ni/16, nj/16);
102       dim3 block = dim3(16, 16);
103
104       // Llamado al kernel
105       ec_calor_kernel<<<grid, block>>>(ni,nj,pitch,kcond,dt,dx,dy,t_data, t_data_old);
106
107   }
```

Figura 14: Subrutina para invocar el kernel que resuelve la ecuacción de calor con memoria global

Por último, en la Figura 15 se muestra el cuerpo de nuestro kernel que resolverá la ecuación de calor. Estas instrucciones serán implementadas por cada hilo sobre distintos datos, es decir, cada punto de nuestro dominio físico será calculado por un núcleo del GPU. Lo primero que podemos notar

de nuestra función kernel a diferencia de cualquier función tradicional escrita en C es la definición de los índices i, y j en término de las variables predefinidas $threadIdx.x$, $blockIdx.x$, etc. Debido a que estamos utilizando la memoria global del GPU, nuestro campo bidimensional se encuentra acomodado en un arreglo unidimensional de datos. Cada entrada del arreglo tendrá el valor del campo de temperaturas en un punto definido. La combinación específica de las variables $i = blockIdx.x \times blockDim.x + threadIdx.x$ y $j = blockIdx.y \times blockDim.y + threadIdx.y$ permiten identificar a cada hilo con la entrada correspondiente del arreglo que representa el campo de temperaturas. Así, cada hilo podrá realizar los cálculos de un nodo independientemente de los otros hilos. Después, creando los índices únicos a cada hilo $i2d, i2d2, i2d3, etc.$ podemos accesar a los distintos valores de la temperatura de los nodos vecinos para implementar la discretización en diferencias finitas de la ecuación de calor.

```
109  __global__ void ec_calor_kernel (int ni,int nj,int pitch, float kcond, float dt,
110                                   float dx, float dy, float *t_data, float *t_data_old)
111  {
112      int i, j, i2d, i2d2, i2d3, i2d4, i2d5;
113      float told,tnow,tip1,tim1,tjp1,tjm1;
114
115      i = blockIdx.x*BlockDim.x + threadIdx.x;
116      j = blockIdx.y*BlockDim.y + threadIdx.y;
117
118      i2d = i + j*pitch/sizeof(float);
119      i2d2= (i+1) + (j)*pitch/sizeof(float);
120      i2d3= (i-1) + (j)*pitch/sizeof(float);
121      i2d4= (i) + (j+1)*pitch/sizeof(float);
122      i2d5= (i) + (j-1)*pitch/sizeof(float);
123
124      if (i ==ni-1) i2d2= ni-1 + (j)*pitch/sizeof(float);
125      if (i == 0) i2d3= 0 + (j)*pitch/sizeof(float);
126      if (j ==nj-1) i2d4= i + (nj-1)*pitch/sizeof(float);
127      if (j == 0) i2d5= i + (0)*pitch/sizeof(float);
128
129      told = t_data_old[i2d];
130      tip1 = t_data_old[i2d2];
131      tim1 = t_data_old[i2d3];
132      tjp1 = t_data_old[i2d4];
133      tjm1 = t_data_old[i2d5];
134
135      tnow = told + dt*kcond*((tip1-2.0f*told+tim1)/(dx*dx)
136                            + (tjp1-2.0f*told+tjm1)/(dy*dy));
137      t_data[i2d] = tnow;
138  }
```

Figura 15: Kernel para resolver la ecuación de calor con memoria global.

Una vez hecho esto, nuestro programa está completo. Podemos utilizar los datos generados por este programa para graficar el campo de temperaturas en distintos pasos de tiempo. Unas imágenes se presentan en la Figura 16. La primera imagen muestra el dominio a temperatura inicial constante con una fuente de calor puntual en el centro. En las siguientes imágenes el campo de temperaturas comienza a difundirse hasta lograr que todo el dominio altere su temperatura inicial.

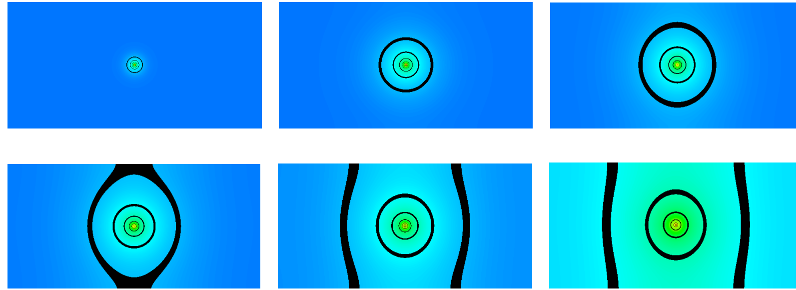


Figura 16: Evolución temporal de la ecuación de Calor en 2D. $L_x = 640$ nodos y $L_y = 320$ nodos. En ésta figura los colores representan distintos valores de la temperatura en donde el azul marino es la temperatura igual a 0 y el rojo es la temperatura igual a 100.

En el Apéndice D se presentan los códigos para resolver la ecuación de calor con memoria compartida y con memoria de texturas. Estos códigos se utilizaron para comparar la velocidad de ejecución entre estos tres distintos tipos de memoria. Se midió el tiempo que toma a cada programa completar un total de 10,000 iteraciones variando el tamaño del sistema. Todos los resultados presentados a continuación fueron generados con una tarjeta NVIDIA GeForce GTX 260 con 192 SPs. Los resultados se muestran en la Figura 17.

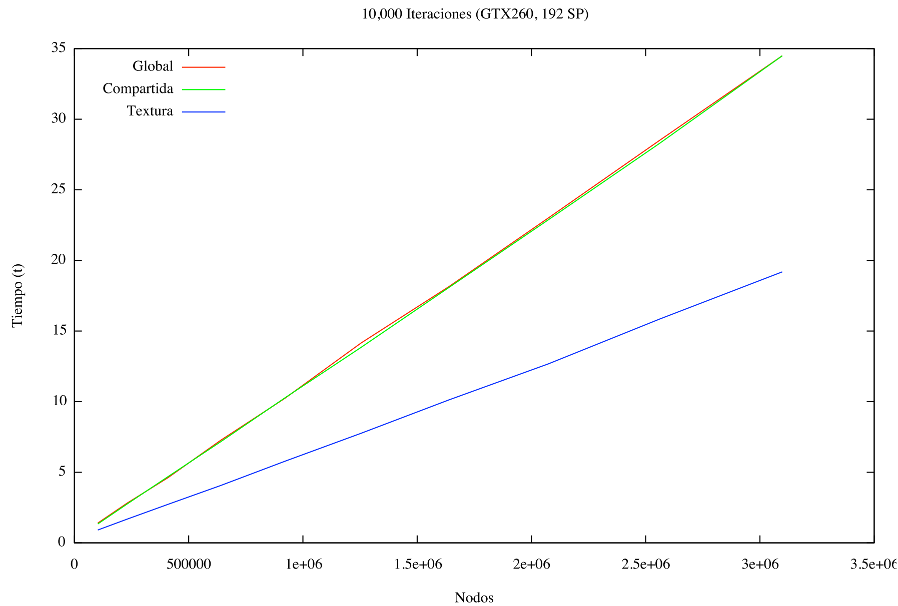


Figura 17: Desempeños de memoria con la ecuación de calor.

Los resultados muestran que para éste método en particular, la memoria de textura alcanza un mejor desempeño que las otras dos memorias. Se puede observar que logra alcanzar las 10,000 iteraciones de la ecuación de calor en la mitad del tiempo que le toma a la memoria global o a la memoria compartida.

En este capítulo se presentaron los conceptos básicos para entender la implementación de distintos algoritmos para su ejecución sobre procesadores gráficos de alto rendimiento. Sin embargo, la información presentada aqui debe de funcionar únicamente como una introducción a este tipo de programación, específicamente con el lenguaje CUDA. También es importante mencionar que existen muchas maneras de otimizar un código escrito en CUDA, ya sea mediante el tipo de memoria que se este utilizando como los tamaños de los bloques e hilos que sean optimos para cada programa en específico.

El interés de esta tesis por aprender a usar las tarjetas de video radica en la simulación de flujos. En el siguiente capítulo se presentan las ecuaciones de movimiento de los flujos incompresibles y un par de soluciones analíticas que nos servirán para validar el método implementado sobre estas tarjetas.

3. Dinámica de Fluidos

El interés de esta tesis por entender el lenguaje de programación en paralelo CUDA surge de intentar modelar fenómenos físicos dentro del campo de la dinámica de fluidos. Debido a la complejidad de las ecuaciones de movimiento, el uso de simulaciones en base a distintos métodos numéricos ha acompañado a la mecánica de fluidos desde los años 1950's [7]. A continuación se presenta una colección de nociones y conceptos básicos que son utilizados en la teoría de los fluidos al igual que las ecuaciones que gobiernan su comportamiento [2]. Al final de este capítulo se presentan el flujo de Poiseuille y el flujo de Womersley[23], que servirán más adelante para comprobar la validez del método numérico que se implementó en las tarjetas gráficas.

3.1. Fluidos

El término "fluido" utilizado como sustantivo o adjetivo se refiere a un estado de la materia, sin hacer referencia a alguna sustancia en particular. Desde temprano, se aprende que la materia, desde un punto de vista macroscópico, existe en tres distintos estados: sólido, liquido y gaseoso. El termino "fluido" se refiere a la segunda y tercera de estas clasificaciones colectivamente. A menudo se utiliza la palabra "flujo" para distinguir a un fluido de un sólido, sin embargo, para llegar a una distinción menos ambigua es necesario tomar en cuenta consideraciones mecánicas de estos cuerpos. Estas propiedades mecánicas están relacionadas con la manera en la cual un cuerpo responde a un intento de deformarlo. Desde este punto de vista, distinguimos a los fluidos de los sólidos debido a la manera en que éstos responden a fuerzas deformantes.

Cuando una fuerza deformante, relacionada a lo que llamamos esfuerzos, es aplicada a un cuerpo material, se encuentra que existen generalmente cuatro tipos de respuestas mecánicas, como se ilustra esquemáticamente en la Figura 18. En el primer tipo, el cuerpo no se deforma en lo mas mínimo, sin importar la magnitud de la fuerza aplicada. A cuerpos de este tipo se les refiere como "cuerpos rígidos" (una idealización). En el segundo y tercer tipos de cuerpos, se presenta una deformación bajo la acción de las fuerzas deformantes; la deformación induce tensiones internas capaces de balancear las fuerzas deformantes y detener la deformación. Si la fuerza es retirada y el cuerpo regresa a su geometría inicial al cuerpo se le conoce como "cuerpo elástico", mientras que si el cuerpo mantiene su estado de deformación es conocido como "cuerpo plástico". Finalmente, en el cuarto tipo, el cuerpo se deforma bajo la acción de la fuerza y continua deformándose cuando la fuerza es retirada. Los cuerpos que caen dentro de esta respuesta mecánica son conocidos como cuerpos fluidos.

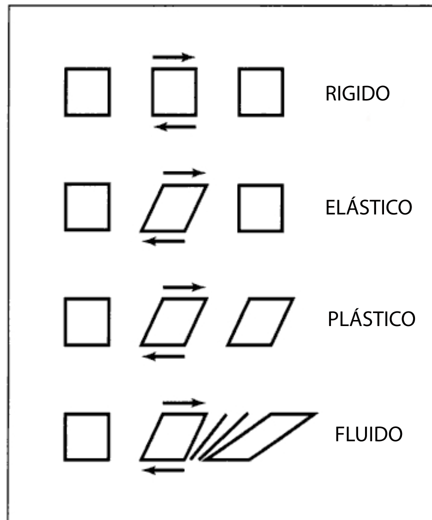


Figura 18: Clasificación mecánica de los cuerpos materiales. Un cuerpo inicialmente rectangular (izquierda) es sujeto a una fuerza cortante (centro), luego la fuerza es retirada (derecha).

La inhabilidad de los fluidos a mantenerse en equilibrio con una fuerza deformante, su inhabilidad a oponerse a esa fuerza y mantenerse en reposo, es una propiedad que tiene su origen en la estructura molecular de los fluidos, específicamente en el balance de fuerzas de atracción y repulsión entre las moléculas que forman el cuerpo. En un estado sólido, las fuerzas de atracción predominan hasta lograr que las moléculas vecinas no puedan separarse unas de otras, dándole al cuerpo su carácter sólido y su resistencia a la deformación. En un estado fluido, en contraste, el balance entre las fuerzas de atracción y repulsión es tan precario que las moléculas vecinas están casi libres de moverse relativamente entre ellas, dando al cuerpo su "fluidez" [23]. Además de esto, las vibraciones caóticas a las que están sometidas las moléculas logran que la escala microscópica sea completamente inadecuada para un estudio macroscópico de los fluidos. Este tipo de dificultades se resuelven en el estudio de la dinámica de fluidos al trabajar solamente en la escala macroscópica y tratando a un cuerpo fluido como una constitución no de moléculas sino de pequeñas piezas que son continuas unas con otras, sin ningún espacio vacío entre ellas. Estas piezas son comúnmente conocidas como "elementos fluidos". La visión de los fluidos como entes continuos y el concepto de "elementos fluidos" son esenciales en el estudio de la dinámica de fluidos ya que nos permiten utilizar funciones continuas para describir las propiedades de un flujo en la escala macroscópica. En esta descripción cada punto representa un elemento fluido, y las propiedades como la velocidad o la densidad están definidas en ese punto.

3.2. Fuerzas actuando sobre un fluido

Es posible hacer una distinción entre dos tipos de fuerzas que actúan sobre la materia continua. En el primer grupo están las fuerzas de largo alcance, como la gravedad, que decrecen lentamente con respecto al incremento en la distancia entre elementos interactuantes. Estas fuerzas son capaces de penetrar al interior de un fluido y actuar sobre todos los elementos fluidos. La gravedad es el ejemplo más importante pero existen otros tipos de fuerzas de este grupo como las fuerzas electromagnéticas y las fuerzas ficticias como la fuerza centrífuga, que aparecen en sistemas de referencia no inerciales. Una consecuencia de esta lenta variación de las fuerzas de cuerpo es que se pueden considerar constantes para toda la materia dentro de un pequeño elemento de volumen. Estas fuerzas son proporcionales a la distribución espacial de una cantidad física que determina la interacción, en el caso de la gravedad es la masa o la densidad, por lo tanto estas fuerzas son proporcionales al volumen del elemento de fluido. Así, éste grupo de fuerzas es conocido como fuerzas volumétricas o fuerzas de cuerpo.

En el segundo grupo se encuentran las fuerzas de corto alcance, que tienen un origen molecular, presentan una disminución extremadamente rápida con el incremento de la distancia entre elementos interactuantes, y son únicamente apreciables cuando esa distancia es del orden de la separación entre las moléculas del fluido. Son despreciables a menos que exista un contacto mecánico directo entre los elementos interactuantes. Si un elemento fluido es influenciado por fuerzas de corto alcance que sean generadas por materia fuera de este elemento, estas fuerzas de corto alcance pueden actuar solamente en una delgada capa adyacente a la frontera de elemento fluido, el espesor de esta capa es igual a la profundidad de la "penetración" de las fuerzas. El total de las fuerzas de corto alcance que actúan sobre el elemento esta entonces determinado por el área de la superficie del elemento. Considerando un elemento de superficie plano en el fluido, la fuerza de corto alcance local representa la fuerza total ejercida sobre el fluido de un lado de la superficie por el elemento que se encuentra en el otro lado. Si además se considera que la profundidad de penetración de estas fuerzas es pequeña comparada con las dimensiones del elemento de superficie, esta fuerza total ejercida a través del elemento será proporcional a su área δA y su valor para un tiempo t. Para un elemento en la posición $\mathbf{x}$ puede ser escrita como el vector

$$T(\mathbf{n}, \mathbf{x}, t)\delta A, \tag{3}$$

en donde $\mathbf{n}$ es la normal unitaria al elemento. La convención que es adoptada es que $\mathbf{T}$ es el esfuerzo ejercido por el fluido de el lado del elemento de superficie en el que $\mathbf{n}$ apunta, sobre el fluido en el cual $\mathbf{n}$ se aleja; así, una componente normal de $\mathbf{T}$ en el mismo sentido que $\mathbf{n}$ representa una tensión. La fuerza por unidad de área, $\mathbf{T}$ es llamada el "esfuerzo" local.

Mas adelante hablaremos de las ecuaciones que describen el movimiento de un fluido que esta sujeto a fuerzas de corto alcanze representadas por la ecuación (3), primero, determinaremos la dependencia de $\mathbf{T}$ sobre la dirección de la normal del elemento de superficie sobre el que actúa. Consideremos todas

las fuerzas que actúan sobre un fluido dentro de un elemento de volumen δV en la forma de un tetraedro como se muestra en la Figura 19.

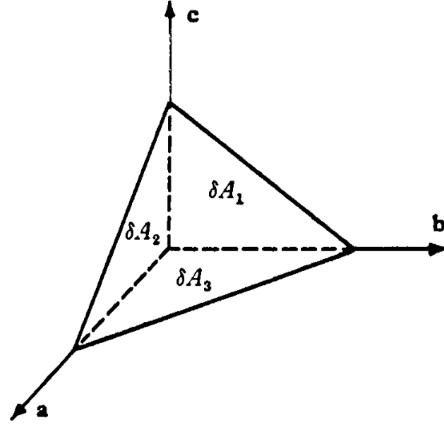


Figura 19: Un elemento de volumen en forma de tetrahedro con tres caras ortogonales

Las tres caras ortogonales tienen áreas δA_1, δA_2, y δA_3 y normales que apuntan hacia afuera $-\mathbf{a}$, $-\mathbf{b}$, $-\mathbf{c}$, y la cuarta cara inclinada tiene área δA y normal unitaria $\mathbf{n}$. Las fuerzas superficiales actúan sobre el fluido en el tetraedro a través de las cuatro superficies, y la suma la denotamos por

$$\mathbf{F_s} = \mathbf{T}(\mathbf{n})\delta A + \mathbf{T}(-\mathbf{a})\delta A_1 + \mathbf{T}(-\mathbf{b})\delta A_2 + \mathbf{T}(-\mathbf{c})\delta A_3;$$

La superficie δA está relacionada con las otras tres mediante una proyección, de modo que

$$\delta A_1 = \mathbf{a} \cdot \mathbf{n} \delta A$$

y relaciones similares se encuentran para δA_2 y δA_3. De este modo, la i-ésima componente de la suma de las fuerzas superficiales puede ser escrita como

$$F_{s_i} = [T_i(\mathbf{n}) - \{a_j T_i(\mathbf{a}) + b_j T_i(\mathbf{b}) + c_j T_i(\mathbf{c})\} n_j] \, \delta A \tag{4}$$

Ahora, la fuerza total volumétrica en el fluido dentro del tetraedro es proporcional al volumen δV, que es de un orden menor que δA en las dimensiones lineales del tetraedro. La masa del fluido dentro del tetraedro es también del orden de δV, y por lo tanto también lo es el producto de la masa y la aceleración del fluido en el teraedro. Así, si las dimensiones lineales del tetraedro se aproximan a cero sin cambiar de forma, se puede notar que los primeros dos términos de la ecuación que representa la segunda ley de Newton

$$masa \times aceleración = fuerzas\, volumétricas + fuerzas\, superficiales$$

se aproximan a cero como δV, mientras que el tercer término aparentemente se aproxima a cero como δA. En estas circunstancias la ecuación puede satisfacerse únicamente si el coeficiente de δA en la ecuación (4) tiende a cero, brindando en el límite,

$$T_i(\mathbf{n}) = \{a_j T_i(\mathbf{a}) + b_j T_i(\mathbf{b}) + c_j T_i(\mathbf{c})\} n_j \tag{5}$$

Los vectores $\mathbf{n}$ y $\mathbf{T}$ no dependen de ningún modo en la opción de ejes coordenados que se escoja, y la expresión dentro de los corchetes en la ecuación (5) debe representar la (i,j)-ésima componente de una cantidad que es también independiente de los ejes. Es decir, la cantidad en corchetes es una componente de un tensor de segundo orden, σ_{ij}, y

$$T_i(\mathbf{n}) = \sigma_{ij} n_j. \tag{6}$$

La componente σ_{ij} de este tensor es la i-ésima componente de la fuerza por unidad de área ejercida sobre un elemento de superficie plano normal a la dirección j , en una posición $\mathbf{x}$ en el fluido al tiempo t, y es conocido como el "tensor de esfuerzos". De este modo, sobre un elemento de superficie determinado por el vector normal $\mathbf{n}$ el esfuerzo se encuentra haciendo la proyección $\sigma \cdot \mathbf{n}$. Este esfuerzo puede descomponerse en la parte normal a la superficie y en otra parte tangente a la superficie. Son los esfuerzos tangentes los responsables de las deformaciones.

3.3. Descripción de un flujo

La hipótesis del continuo, introducida en la primera sección de este capítulo nos permite utilizar el concepto de "velocidad local" de un fluido. Lo mismo puede ser dicho de otros campos como el de presión. Existen dos maneras de describir un medio continuo. La primera, conocida como la descripción Euleriana es tal que las cantidades que describen al flujo están definidas como funciones de la posición en el espacio ($\mathbf{x}$) y el tiempo (t). El estado dinámico de un fluido esta dado por el campo de velocidades $\mathbf{u}(\mathbf{x}, t)$ y de presiones $p(\mathbf{x}, t)$ para todo punto en el espacio y el tiempo. Esta descripción puede ser pensada como una fotografía de la distribución espacial de la velocidad del fluido (y de otras cantidades como la densidad o la presión) para cualquier instante durante el movimiento.

La segunda manera, conocido como la descripción Lagrangiana, hace uso del hecho que, como en la mecánica de partículas, algunas de las cantidades físicas están referidas no solamente a ciertas posiciones en el espacio, pero también (y mas fundamentalmente) a pedazos de materia identificables. Las cantidades que describen al flujo son aquí funciones del tiempo y de la elección de los elementos materiales del fluido. Debido a que los elementos materiales de un fluido cambian de forma durante su movimiento, debemos identificar el elemento seleccionado de tal manera que su extensión no esté considerada. Un método conveniente es el de especificar el elemento por la posición de su centro de masa en un instante inicial, bajo el entendimiento que las dimensiones lineales iniciales

del elemento son tan pequeñas como para garantizar su pequeñez en instantes relevantes subsecuentes a pesar de las deformaciones y distorsiones del elemento. Así, los campos en la descripción Lagrangiana están referidos a la posición de los elementos materiales y el tiempo.

Comúnmente se utiliza la descripción Euleriana para describir a los fluidos. La función $\mathbf{u}(\mathbf{x}, t)$ será entonces la variable dependiente en nuestro análisis y otras cantidades de flujo como la presión también serán miradas como funciones de $\mathbf{x}$ y t. De este modo, las llamadas "velocidades Eulerianas" son funciones del tiempo y de posiciones coordenadas dentro de un fluido. Por ejemplo en un sistema coordenado cilíndrico x, r, θ, Si denotamos a las componentes de la velocidad $\mathbf{u}$ por u, v, w, respectivamente, entonces

$$u = u(x, r, \theta), \quad v = v(x, r\theta), \quad w = w(x, r, \theta)$$

Debido a que en tiempos diferentes esta posición coordenada dentro del cuerpo es ocupada por elementos fluidos diferentes, las velocidades Eulerianas en un punto en un campo no representan las velocidades del mismo elemento para todo tiempo. Ellas representan las velocidades de distintos elementos que ocupan esta posición a través del tiempo.

Es evidente que en un flujo estacionario, es decir en un flujo en el que $\mathbf{u}$ no depende de t, un elemento fluido puede experimentar aceleraciones al moverse a un punto en el cual $\mathbf{u}$ tiene un valor distinto. En este caso, la derivada $\partial\mathbf{u}/\partial t$ no es la aceleración de un elemento en la posición $\mathbf{x}$ al tiempo t. La expresión correcta para la aceleración de un elemento material puede ser encontrada al notar que un elemento con posición $\mathbf{x}$ al tiempo t está en la posición $\mathbf{x} + \mathbf{u}\delta t$ al tiempo $t + \delta t$, y el cambio en su velocidad en el pequeño intervalo δt es

$$\mathbf{u}(\mathbf{x} + \mathbf{u}\delta t, t + \delta t) - \mathbf{u}(\mathbf{x}, t) = \delta t \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) + O(\delta t^2).$$

Así, la aceleración de un elemento fluido en $(\mathbf{x}, t)$ es

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u}. \tag{7}$$

Estas consideraciones pueden ser aplicadas a cualquier otra cantidad física puntual, digamos θ, que sea función de $\mathbf{x}$ y t; θ puede ser una cantidad escalar, como la densidad local o la temperatura de un fluido, o una cantidad vectorial, como la velocidad angular local del fluido. $\partial\theta/\partial t$ es la razón de cambio local debido a cambios temporales en la posición $\mathbf{x}$, y para encontrar la razón de cambio de θ para un elemento material debemos añadir la razón de cambio "convectiva" de $\mathbf{u} \cdot \nabla\theta$ causada por el transporte del elemento a una nueva posición. Es conveniente introducir la notación

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla, \tag{8}$$

de tal manera que la aceleración de un elemento fluido pueda ser escrita como $\frac{D\mathbf{u}}{Dt}$. Este operador aparece muy comúnmente en las ecuaciones que expresan leyes de conservación y se le llama derivada material o derivada Lagrangiana.

3.4. Conservación de masa: ecuación de continuidad

Para aplicar la ley de conservación de masa en un punto en un flujo, se puede considerar un volumen fijo del campo en forma de una caja cerrada alrededor del punto, para después encoger la caja hasta ese punto. Consideremos una superficie cerrada A con posición fija relativa a los ejes coordenados escogidos y que encierra un volumen V ocupado por un fluido. Si ρ es la densidad del fluido en la posición $\mathbf{x}$ al tiempo t, la masa del fluido encerrado por la superficie en cualquier instante es $\int_V \rho dV$ y la razón con la cual el fluido atraviesa la superficie es $\int_A \rho\mathbf{u}\cdot\mathbf{n}dA$, en donde δV y δA son elementos del volumen encerrado y de la superficie con normal $\mathbf{n}$ que lo encierra. En la ausencia de fuentes de masa, la masa total encerrada en el volumen es conservada. Tenemos que el cambio temporal de la masa dentro del volumen V es igual al flujo de masa por su frontera:

$$\frac{d}{dt}\int_V \rho dV = -\int_A \rho\mathbf{u}\cdot\mathbf{n}dA,$$

que al utilizar el teorema de Gauss y considerando que el volumen está fijo, esta igualdad puede ser escrita como

$$\int_V \left\{\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\mathbf{u})\right\} dV = 0. \tag{9}$$

Esta relación es válida para todas las opciones de volúmenes V que caen dentro del fluido, lo cual es posible únicamente si el integrando es idénticamente cero para todo el fluido. Así, para todo punto en el fluido

$$\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\mathbf{u}) = 0. \tag{10}$$

Esta ecuación es una ecuación fundamental en la dinámica de los fluidos. Un nombre común es la “ecuación de continuidad”.

Un flujo se denomina “incompresible” cuando la densidad de un elemento fluido no es afectada por cambios en la presión. La densidad de un fluido en un elemento de masa puede cambiar como consecuencia de la conducción molecular de calor hacia el elemento; sin embargo, circunstancias en las cuales la conducción de calor es despreciable son comunes, y el hecho que un fluido sea incompresible generalmente implica que la densidad de cada elemento de masa del fluido permanece constante. Así, para un fluido incompresible, la razón de cambio de ρ con respecto al movimiento es cero, esto es,

$$\frac{D\rho}{Dt} = 0. \tag{11}$$

La ecuación de contiuidad entonces toma la forma

$$\nabla \cdot \mathbf{u} = 0, \tag{12}$$

lo que nos dice que la divergencia del campo de velocidades es cero en todo punto del fluido.

3.5. Ecuaciones de Movimiento

La "ecuación de movimiento" para un fluido es, en su forma más fundamental, una relación que iguala la razón de cambio del momento de un elemento fluido y la suma de todas las fuerzas que actúan sobre él. Para un cuerpo fluido con volumen V encerrado por una superficie material S, el momento es $\int_V \mathbf{u}\rho dV$, y su cambio en el tiempo es

$$\frac{d}{dt}\int_V \rho\mathbf{u} dV + \int_A \rho\mathbf{u}(\mathbf{u}\cdot\mathbf{n})dA = \int_V \frac{D\mathbf{u}}{Dt}\rho dV,$$

que es simplemente la suma de los productos de la masa y la aceleración para todos los elementos dentro del volumen V.

Como hemos explicado anteriormente, en una porción de fluido actúan fuerzas volumétricas al igual que fuerzas superficiales. Si denotamos al vector resultante de las fuerzas volumétricas por unidad de masa como $\mathbf{F}$, de tal manera que la fuerza volumétrica total sobre la porción de fluido es

$$\int_V \mathbf{F}\rho dV.$$

La i-ésima componente de la fuerza superficial ejercida sobre un elemento de superficie de area δA y normal $\mathbf{n}$ puede ser representado como $\sigma_{ij}n_j\delta A$, donde σ_{ij} es el tensor de esfuerzos y la fuerza superficial total ejercida sobre la porción de fluido por la materia que la encierra es entonces

$$\int_A \sigma_{ij}n_j\delta A = \int_V \frac{\partial\sigma_{ij}}{\partial x_j}dV.$$

Entonces, el balance de momento para la porción de fluido escogida está expresada por

$$\int_V \frac{Du_i}{Dt}\rho dV = \int_V F_i\rho dV + \int_V \frac{\partial\sigma_{ij}}{\partial x_j}dV. \tag{13}$$

Esta relación integral se mantiene para todas las opciones del volumen V, lo cual es posible solo si

$$\rho\frac{Du_i}{Dt} = \rho F_i + \frac{\partial\sigma_{ij}}{\partial x_j}, \tag{14}$$

en todos los puntos del fluido. Esta ecuación diferencial que dicta la aceleración del fluido en términos de las fuerzas volumétricas locales y el tensor de esfuerzos es generalmente entendida como la "ecuación de movimiento". Esta ecuación no puede ser utilizada para determinar la distribución de la velocidad

de un fluido hasta que F_i y σ_{ij} sean explícitamente especificadas. La fuerza volumétrica que actúa sobre un fluido en muchos casos es debida al campo gravitacional de la tierra para el cual $\mathbf{F} = \mathbf{g}$. Debido a que el tensor de esfuerzos es una manifestación de las reacciones internas del fluido consigo mismo, la especificación de este tensor presenta un reto mas complicado.

3.5.1. Flujos Newtonianos

Un fluido que se encuentra en reposo generalmente se encuentra en un estado de compresión, y el tensor de esfuerzos se escribe como

$$\sigma_{ij} = -p\delta_{ij} \tag{15}$$

en donde p $(= -\frac{1}{3}\sigma_{ii})$ se le denomina la presión mecánica de un fluido y es en general una función de $\mathbf{x}$. Esto nos dice que en un fluido en reposo las fuerzas de contacto por unidad de área sobre un elemento de superficie plano dentro del fluido son fuerzas que actúan en la dirección normal al plano y tiene la misma magnitud para todas las direcciones de la normal $\mathbf{n}$ en cualquier punto. Esta propiedad es establecida como consecuencia de la suposición que para un fluido en reposo los esfuerzos tangenciales son cero. Sin embargo, estos resultados no son válidos par un fluido en movimiento; los esfuerzos tangenciales son distintos de cero.

Sin embargo, es útil el tener una cantidad escalar que caracterice un fluido en movimiento análoga a la presión estática de un fluido en el sentido que sea una medida de la intensidad local de los esfuerzos que producen cambios de volumen de fluido. Entonces es conveniente también entender al tensor de esfuerzos como la suma de una parte isotrópica $-p\delta_{ij}$, que tiene la misma forma del tensor de esfuerzos de un fluido en reposo, y una parte no-isotrópica, digamos d_{ij} que contribuye los esfuerzos tangenciales:

$$\sigma_{ij} = -p\delta_{ij} + d_{ij.} \tag{16}$$

La parte no-isotrópica se conoce como el tensor de esfuerzos viscoso o deviatórico y es debido únicamente a la existencia de movimiento en el fluido. La parte representada por el tensor de esfuerzos viscoso está relacionada al gradiente de velocidades por lo que comúnmente se conoce como relaciones constitutivas. El modelo de relación constitutiva para un fluido lineal e isótropo es conocido como la relación Newtoniana. Para fluidos Newtonianos tenemos que la expresión para el tensor de esfuerzos deviatórico es

$$d_{ij} = 2\mu(e_{ij} - \frac{1}{3}\triangle\delta_{ij}) \tag{17}$$

donde

$$e_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) y \triangle = e_{ii}. \tag{18}$$

Sustituyendo esta relación constitutiva en las ecuaciones de movimiento se encuentran las ecuaciones de Navier-Stokes.

3.5.2. Ecuaciones de Navier-Stokes

Con la expresión (17) para el tensor de esfuerzos deviatórico, el esfuerzo total se convierte en

$$\sigma_{ij} = -p\delta_{ij} + 2\mu(e_{ij} - \frac{1}{3}\triangle\delta_{ij}), \tag{19}$$

Al sustituir estas expresiones en la ecuación de movimiento (14) obtenemos

$$\rho\frac{Du_i}{Dt} = \rho F_i - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\{2\mu(e_{ij} - \frac{1}{3}\triangle\delta_{ij})\}. \tag{20}$$

Esta es conocida como la ecuación de movimiento de Navier-Stokes.

En el caso en el cual la viscosidad μ es uniforme a través del fluido, la ecuación se convierte en

$$\rho\frac{Du_i}{Dt} = \rho F_i - \frac{\partial p}{\partial x_i} + \mu\left(\frac{\partial^2 u_i}{\partial x_j \partial x_j} + \frac{1}{3}\frac{\partial\triangle}{\partial x_j}\right). \tag{21}$$

Un caso aún más especial es el de un fluido incompresible. La ecuación para la conservación de masa se reduce a $\nabla \cdot \mathbf{u} = 0$ y la ecuación de movimiento se convierte en

$$\rho\frac{D\mathbf{u}}{Dt} = \rho\mathbf{F} - \nabla p + \mu\nabla^2\mathbf{u} \tag{22}$$

Estas ecuaciones describen el comportamiento de los flujos Newtonianos e incompresibles y están cerradas.

3.6. Dos soluciones a la ecuación de Navier-Stokes

Dentro del interés de esta tesis se encuentra el poder simular numéricamente el flujo de un fluido a través de un ducto cilíndrico. Particularmente el flujo de Poiseuille y el flujo de Womersley fueron utilizados para lograr validar nuestros códigos numéricos. El flujo de Poiseuille es el flujo generado por un gradiente de presiones constante en el tiempo a lo largo de un ducto mientras que en el flujo de Womersley el gradiente de presiones es dependiente del tiempo. En el siguiente desarrollo se encontrarán las soluciones a estos dos flujos al tomar un sistema coordenado cilíndrico (r, θ, x) para referir a nuestro sistema. En su forma más general, el flujo a través de un tubo requiere resolver las ecuaciones de Navier-Stokes y la ecuación de continuidad en su forma más completa. Sin embargo, las ecuaciones pueden ser simplificadas considerablemente al tomar en cuenta ciertas suposiciones.

3.6.1. Solución de estado estacionario: flujo de Poiseuille

Para encontrar el flujo de Poiseuille es necesario tomar como hipótesis que el flujo es unidireccional e independiente del tiempo, esto es:

$$\mathbf{u} = u\mathbf{e_x} \tag{23}$$

$$\frac{\partial \mathbf{u}}{\partial t} = 0 \tag{24}$$

Al hacer estas suposiciones tenemos que la ecuación de continuidad para flujos incompresibles (ecuación (12)) toma la forma $\frac{\partial \mathbf{u}}{\partial x} = 0$ por lo que la velocidad pierde su dependencia en x. Si ahora hacemos la suposición de que la simetría del ducto proporcionará un flujo independiente de la coordenada angular, entonces la velocidad será solamente una función de la coordenada radial, esto es u=u(r). Gracias a esto, el término de aceleración en las ecuaciones de Navier Stokes desaparece:

$$\mathbf{u} \cdot \nabla \mathbf{u} = u \frac{\partial u}{\partial x} \mathbf{e}_x = 0$$

Gracias a esto, las ecuaciones de Navier Stokes se reducen a:

$$\frac{\partial p}{\partial x} = \mu \left(\frac{d^2 u}{dr^2} + \frac{1}{r} \frac{du}{dr} \right) \tag{25}$$

$$\frac{\partial p}{\partial r} = 0 \tag{26}$$

$$\frac{\partial p}{\partial \theta} = 0 \tag{27}$$

Una característica importante de este tipo de flujo es que la ecuación del comportamiento es independiente de la densidad ρ. La razón de esto es que los términos de aceleración son ahora cero. Los términos restantes en la ecuación representan un balance de fuerzas entre la presión y la resistencia viscosa del fluido.

En la ecuación (25) el termino de la izquierda es una función solamente de x mientras que los términos de la derecha son funciones de r solamente. La única manera en que la ecuación puede satisfacerse es al tener ambos lados igual a una constante, la misma constante, digamos κ, entonces

$$\frac{\partial p}{\partial x} = \kappa \tag{28}$$

$$\mu \left(\frac{d^2 u}{dr^2} + \frac{1}{r} \frac{du}{dr} \right) = \kappa \tag{29}$$

Al resolver la primera ecuación obtenemos

$$p = p(0) + \kappa x \tag{30}$$

en donde x es la distancia axial medida desde la entrada del tubo.

Para poder comparar las soluciones de estas ecuaciones con las soluciones numéricas resolveremos el problema de forma adimensional. Para reescalar la

ecuación en términos de nuestros parámetros físicos debemos notar que las variables del sistema son: la coordenada r, la velocidad u_s, la presión p y la coordenada x, mientras que los parámetros del sistema son: el radio a, la viscosidad del fluido μ, y el gradiente de presiones κ. Podemos escoger algunos parámetros adecuados con dimensiones relevantes para combinarlos y entonces poder definir unas nuevas variables adimensionales. Las dimensiones de el radio del tubo, la viscosidad del fluido y a el gradiente de presiones tienen dimensiones $[a] = D$, $[\mu] = \frac{M}{DT}$, $[\kappa] = \frac{M}{D^2T^2}$ respectivamente donde D es distancia, M es masa, y T es el tiempo. Una combinación de estos tres parámetros nos permite escribir un nuevo conjunto de variables adimesnionales: $r^* = \frac{r}{a}$, $u^* = u\frac{\mu}{\kappa a^2}$, tenemos que la eciación (25) se transforma en:

$$\mu\left(\frac{\kappa a^2}{a^2\mu}\frac{d^2u^*}{dr^*2} + \frac{1}{r'a}\frac{\kappa a^2}{\mu a}\frac{du^*}{dr^*}\right) = \kappa$$

y al factorizar términos obtenemos

$$\frac{d^2u^*}{dr^{*2}} + \frac{1}{r^*}\frac{du^*}{dr^*} = 1 \tag{31}$$

Que es la ecuación escalada en términos de cantidades adimensionales r^*, y u^*. Sin embargo por cuestiones de notación dejaremos de lado los *'s y entenderemos como cantidades adimensionales a las variables r, y u. Al resolver la ecuación adimensional obtenemos

$$u(r) = -\frac{1}{4}r^2 + A\ln r + B \tag{32}$$

donde A, B son constantes de integración. Las dos condiciones de frontera para encontrarlas son la condición de no deslizamiento en la pared del tubo y velocidad finita en el centro, es decir, $u(r = 1) = 0$, $|u(r = 0)| < \infty$, por lo tanto

$$A = 0,\ B = \frac{1}{4} \tag{33}$$

Con estos valores, la solución se convierte en

$$u = \frac{1}{4}\left(1 - r^2\right) \tag{34}$$

Esta es la solución clásica para un flujo estacionario en un tubo, generalmente se le conoce como Flujo de Poiseuille en honor a su primer autor. Este perfil de velocidades tiene la forma parabólica característica comúnmente asociada al flujo en un tubo. Indica que la velocidad máxima ocurre en el eje del tubo $(r = 0)$, y velocidad cero ocurre en la pared del tubo $(r = 1)$.

3.6.2. Flujo pulsátil en un tubo: flujo de Womersley

Un flujo pulsátil se refiere a un flujo en el cual el gradiente de presión que lo genera varía en el tiempo. Una característica de este gradiente es que consiste de una parte constante que no varía en el tiempo y produce un flujo estacionario

como en el flujo de Poiseuille, más una parte oscilatoria que mueve al fluido hacia adelante y hacia atrás y que produce un flujo neto cero. Los términos "estacionario" y "oscilatorio" serán utilizados para referirse a estas dos componentes del flujo, respectivamente, y el término "pulsátil" será entendido como una combinación de los dos.

Nuevamente tomaremos la suposición de que nuestro flujo es unidireccional

$$\mathbf{u} = u\mathbf{e_x},$$

sin embargo, la presión dependerá del tiempo por lo que no podremos suponer un flujo puramente estacionario. En este caso los términos de aceleración son nuevamente cero y todas las suposiciones de la sección anterior siguen siendo validas excepto que la presión y la velocidad ahora tendrán una dependencia temporal.

Es importante notar que la ecuación de movimiento (25) es lineal con respecto a la presión $p(x,t)$ y a la velocidad $u(r,t)$. Como resultado de esta característica, la ecuación puede tratar con la parte estacionaria y la parte oscilatoria de manera separada e independiente. Si la parte estacionaria y oscilatoria de la presión y la velocidad son identificadas por los subíndices "e" y "ϕ" respectivamente, podemos escribir

$$p(x,t) = p_e(x) + p_\phi(x,t) \qquad u(r,t) = u_e(x) + u_\phi(x,t)$$

La ecuación de movimiento es la siguiente:

$$\left\{\frac{dp_e}{dx} - \mu\left(\frac{d^2u_e}{dr^2} + \frac{1}{r}\frac{du_e}{dr}\right)\right\} + \left\{\rho\frac{\partial u_\phi}{\partial t} + \frac{\partial p_\phi}{\partial x} - \mu\left(\frac{\partial^2 u_\phi}{\partial r^2} + \frac{1}{r}\frac{\partial u_\phi}{\partial r}\right)\right\} = 0 \tag{35}$$

en donde se han agrupado los términos que no dependen del tiempo (primer grupo), y los que si (segundo grupo). Esto nos indica que la ecuación ahora es lineal lo que nos permite resolver cada uno de los grupos por separado. La primera ecuación es la ecuación de flujo estacionario tratada en la sección anterior, mientras que la segunda es una ecuación que gobierna la parte oscilatoria del flujo. Ambas ecuaciones son completamente independientes una con respecto a la otra y pueden resolverse para encontrar u_e y u_ϕ respectivamente. Gracias a esta independencia, la relación entre gradientes de presión es

$$\kappa(t) = \kappa_e + \kappa_\phi(t)$$

en donde

$$\kappa(t) = \frac{\partial p}{\partial x}, \quad \kappa_e = \tfrac{dp_e}{dx} \quad \kappa_\phi(t) = \frac{\partial p_\phi}{\partial x}$$

Así, $\kappa(t)$ es el gradiente de presiones "total" en un flujo pulsátil, κ_e es su parte estacionaria, y κ_ϕ es la parte puramente oscilatoria.

En el flujo estacionario el término del gradiente de presiones en la ecuación gobernante es constante, independiente de x, debido a que todos los otros términos en la ecuación son funciones de r solamente, mientras que la presión es función de x solamente. De manera similar, en el flujo oscilatorio, el gradiente de presiones es independiente de x por las mismas razones, pero aquí puede ser una función del tiempo t. La ecuación que gobierna el flujo oscilatorio es de la forma

$$\mu\left(\frac{\partial^2 u_\phi}{\partial r^2}+\frac{1}{r}\frac{\partial u_\phi}{\partial r}\right)-\rho\frac{\partial u_\phi}{\partial t}=\kappa_\phi(t) \tag{36}$$

La solución al flujo de Womersley se obtiene al imponer un gradiente de presiones oscilatorio. En particular, estamos interesados en una solución para la cual $k_\phi(t)$ sea una función oscilatoria en el tiempo. La solución a la ecuación para el flujo oscilatorio con el gradiente de presiones oscilatorio tomado como una función senoidal o cosenoidal es simplificada si, en vez de usar una o la otra, se usa su combinación compleja:

$$\kappa_\phi(t)=\kappa_e\exp(i\omega t)=\kappa_e(\cos\omega t+i\sin\omega t)$$

en donde $i=\sqrt{-1}$. Ya que la ecuación es lineal, la solución con esta opción de k_ϕ consistirá de la suma de dos soluciones, una para la cual $k_\phi(t)=k_s\cos\omega t$ y otra para la cual $k_\phi(t)=k_s\sin\omega t$. La primera es obtenida al tomar la parte real de la solución y la segunda al tomar la parte imaginaria. De este modo, la ecuación 36 toma la forma

$$\mu\left(\frac{\partial^2 u_\phi}{\partial r^2}+\frac{1}{r}\frac{\partial u_\phi}{\partial r}\right)-\rho\frac{\partial u_\phi}{\partial t}=\kappa_e\exp(i\omega t) \tag{37}$$

En este momento es conveniente reescalar nuestra ecuación en términos de cantidades adimensionales, como lo hicimos para el flujo de Poiseuille. Las variables del sistema que debemos adimensionalizar son: la coordenada r, la velocidad u_ϕ, y el tiempo t, mientras que los parámetros del sistema que utilizaremos para adimensionalizarlas son: el radio del ducto a , la viscosidad del fluido μ, y la frecuencia de la oscilación en el gradiente de presiones ω. Las dimensiones de el radio del tubo, la viscosidad del fluido y la frecuencia de la oscilación tienen dimensiones $[a]=D$, $[\mu]=\frac{M}{DT}$, $[\omega]=\frac{1}{T}$ respectivamente donde D es distancia, M es masa, y T es el tiempo. Una combinación de estos tres parámetros nos permite escribir un nuevo conjunto de variables adimesnionales: $r^*=\frac{r}{a}$, $u_e^*=u_e a\omega$, y $t^*=t\omega$. Al escribir nuestra ecuación para la parte oscilatoria en términos de estas cantidades adimensionales obtenemos

$$\frac{\mu\omega}{a}\left(\frac{\partial^2 u_\phi^*}{\partial r^{*2}}+\frac{1}{r^*}\frac{\partial u_\phi^*}{\partial r^*}\right)-\rho a\omega^2\frac{\partial u^*{}_\phi}{\partial t^*}=\kappa_e\exp(it^*)$$

o bién,

$$\frac{\partial^2 u_\phi^*}{\partial r^{*2}}+\frac{1}{r^*}\frac{\partial u^*{}_\phi}{\partial r^*}-\Omega^2\frac{\partial u_\phi^*}{\partial t^*}=\frac{\kappa_e a}{\mu\omega}\exp(it^*) \tag{38}$$

en donde Ω es un parámetro adimensional llamado el número de Womersley, dado por

$$\Omega = \sqrt{\frac{\rho\omega}{\mu}}a \tag{39}$$

Nuevamente dejaremos a un lado la notación de *'s y entenderemos a u, r, t como nuestras cantidades adimensionales. Así, podemos resolver la ecuación por separación de variables, esto es, al descomponer $u_\phi(r,t)$ en una parte que depende solo de r y otra que dependa solamente de t. Más aún, la forma exponencial de la función del tiempo y la forma general de la ecuación dictan que la parte de u_ϕ que depende del tiempo debe tener la misma forma exponencial que el termino del lado derecho de la ecuación. Entonces se propone la separación de variables:

$$u_\phi(r,t) = U_\phi(r)\exp(it) \tag{40}$$

Al hacer esta sustitución en la ecuación gobernante, el factor $\exp(it)$ se cancela, dejando una ecuación diferencial ordinaria para $U_\phi(r)$ solamente

$$\frac{d^2U_\phi}{\partial r^2} + \frac{1}{r}\frac{dU_\phi}{\partial r} - i\Omega^2 U_\phi = K \tag{41}$$

y $K = \frac{\kappa_e a}{\mu\omega}$ es la amplitud de la oscilación adimensional. Ahora, si hacemos un cambio de variable de la forma $\varsigma = r\Omega\sqrt{-i} = r\Omega\left(\frac{i-1}{\sqrt{2}}\right) = r\Lambda$, la ecuación se convierte en

$$\frac{d^2U_\phi}{\partial\varsigma^2} + \frac{1}{\varsigma}\frac{dU_\phi}{\partial\varsigma} + U_\phi = \frac{iK}{\Omega^2} \tag{42}$$

Esta es una ecuación de Bessel para argumentos complejos. La solución general a esta ecuación tiene la forma

$$U_\phi(\varsigma) = \frac{iK}{\Omega^2} + AJ_0(\varsigma) + BY_0(\varsigma) \tag{43}$$

en donde A, B son constantes arbitrarias y J_0 , Y_0 son funciones de Bessel de orden cero y de primer y segundo tipo respectivamente. La variable ς es una variable compleja relacionada con la coordenada radial r por

$$\varsigma(r) = \Lambda r$$

en donde Λ es una frecuencia compleja relacionada a la frecuencia adimensional Ω por

$$\Lambda = \left(\frac{i-1}{\sqrt{2}}\right)\Omega$$

Las condiciones de frontera que la ecuación debe satisfacer para el flujo en un tubo son de no deslizamiento en las paredes del tubo y una velocidad finita en el eje del tubo, es decir

$$U_\phi(r=1)=0 \qquad |U_\phi(0)<\infty|$$

La segunda condición hace que $B = 0$, por propiedades de las funciones de Bessel, y la primera condición hace que $A = \frac{-iK}{\Omega^2 J_0(\Lambda)}$. Con esto, finalmente tenemos que la solución par U_ϕ es

$$U_\phi = \frac{iK}{\Omega^2}\left(1 - \frac{J_0(\varsigma)}{J_0(\Lambda)}\right) \tag{44}$$

Si añadimos el termino temporal tenemos entonces la solución a la ecuación de flujo pulsátil

$$u_\phi(\varsigma, t) = \frac{iK}{\Omega^2}\left(1 - \frac{J_0(\varsigma)}{J_0(\Lambda)}\right)\exp(it) \tag{45}$$

Esta ecuación junto con la ecuación (34) son las soluciones a las ecuaciones de Navier-Stokes que nos servirán más adelante para comparar los resultados de las simulaciones numéricas hechas con el método de lattice-Boltzmann. Dicho método aproxima las ecuaciones de Navier Stokes y será presentado en el siguiente capítulo.

4. Método de Lattice Boltzmann

El método de lattice-Boltzmann (MLB) se ha desarrollado para convertirse en un esquema numérico alternativo y prometedor para simular flujos fluidos y la modelación de la física de fluidos [4]. Basado en la teoría cinética, el MLB simula flujos indirectamente al seguir la evolución de las distribuciones de probabilidad que aproximan la microscopía del fenómeno. El esquema es totalmente paralelo y puede fácilmente modelar flujos con condiciones de frontera complicadas [22].

Históricamente, los modelos de la ecuación de lattice-Boltzmann (ELB) evolucionaron directamente de los modelos de "lattice gas automata" (LGA) [10]. Mientras que los modelos LGA son modelos Booleanos, los modelos que utilizan la ELB son la contraparte continua de los modelos LGA correspondientes - la presencia de una partícula en el modelo LGA (representada por un numero Booleano) es reemplazada por una función de distribución de una partícula (representada por un numero real). Teóricamente, la ecuación de lattice Boltzmann se puede concebir como una discretización de la ecuación de Boltzmann tradicional [14].

Heredando su carácter de los LGA's, el MLB trata a la dinámica de los fluidos desde un nivel cinético microscópico. Sin embargo, como en la ecuación de Boltzmann, el MLB describe la evolución de poblaciones de partículas en vez de intentar seguir el movimiento de partículas individuales. El algoritmo necesita calcular un término de colisiones para después poder seguir la evolución temporal de las distribuciones de probabilidad. Esto resulta en una lógica muy simple que puede ser fácilmente implementada en computadoras masivamente paralelas.

El desarrollo de métodos como LGA o LBM esta basado en la suposición que el comportamiento macroscópico del flujo de un fluido no es muy sensible al detalle del movimiento microscópico. Así, estos modelos fueron desarrollados en base al micromundo más simple posible que llevará a las ecuaciones de Navier-Stokes incompresibles [25]. Los métodos lograron simular flujos incompresibles, relacionando el mundo de las partículas microscópicas con el mundo macroscópico a través de los momentos de las distribuciones de probabilidad.

4.1. La Ecuación de Boltzmann

El estado mecánico de un sistema de partículas está dado por la colección de posiciones $\mathbf{x}$ y velocidades ξ. Estamos interesados en la función de distribución de una partícula $f(\mathbf{x}, \xi, t)$ definida de tal manera que

$$f(\mathbf{x}, \xi, t) d^3x d^3\xi$$

es el número de partículas que, al tiempo t, tienen posiciones que están dentro de un elemento de volumen dx alrededor del vector $\mathbf{x}$ y velocidades que caen dentro del elemento $d\xi$ del espacio de momentos alrededor de ξ. Para hacer la definición de $f(\mathbf{x}, \xi, t)$ más precisa, consideremos un punto en el espacio seis-dimensional, llamado el espacio de configuración, generado por las coordenadas $(\mathbf{x}, \xi)$ de una partícula. Un punto en este espacio representa el estado

de una partícula. Para cada instante en el tiempo, el estado de un sistema de N partículas esta representado por N puntos en el espacio de configuraciones. Si construimos un elemento de volumen $d^3xd^3\xi$ alrededor de cada punto en el espacio de configuración, y si contamos el numero de puntos en este elemento de volumen, el resultado es, por definición, $f(\mathbf{x},\xi,t)d^3xd^3\xi$. Si los tamaños de estos elementos de volumen son escogidos de tal manera que cada uno de ellos contenga una gran cantidad de puntos, y si la densidad de estos puntos no varía rápidamente de un elemento a otro, entonces $f(x,\xi,t)$ puede ser considerada como una función continua de sus elementos, y podemos tomar la aproximación

$$\sum f(\mathbf{x},\xi,t)d^3xd^3\xi \approx \int f(\mathbf{x},\xi,t)d^3xd^3\xi$$

en donde la suma de la izquierda se extiende sobre todos los centros de los elementos de volumen.

Se redefine a $f(\mathbf{x},\xi,t)$ como la masa por unidad de volumen en el espacio de configuraciones seis dimensional dado por las coordenadas espaciales y la velocidad de las partículas ξ para que esté normalizada por la masa del sistema. La diferencia entre las dos distribuciones es una cuestión de normalización solamente, las funciones de distribución de probalilidad necesariamente estan normalizadas a la unidad; la f de densidad esta normalizada a la masa total del sistema. Habiendo definido la función de distribución, podemos expresar la información de que hay N moléculas en un volumen V a través de la condición de normalización

$$\int f(\mathbf{x},\xi,t)d^3\xi = \rho$$

El propósito de la teoría cinética es el de encontrar la función de distribución $f(\mathbf{x},\xi,t)$ para una forma dada de interacción entre partículas. La forma de $f(\mathbf{x},\xi,t)$ al tomar el límite $t \to \infty$ tendría entonces todas las propiedades del equilibrio del sistema. Para lograr esto debemos de encontrar las ecuaciones de movimiento para la función de distribución. La función de distribución cambia en el tiempo debido a que las partículas constantemente entran y salen de un elemento de volumen en el espacio de configuración. Si suponemos que no existen colisiones entre las partículas y en ausencia de fuerzas externas, entonces una partícula con coordenadas $(\mathbf{x},\xi)$ en el instante t tendrán las coordenadas $(\mathbf{x}+\xi\delta t,\xi)$ en el instante $t+\delta t$. De este modo, todas las partículas contenidas en un elemento $d^3xd^3\xi$, en $(\mathbf{x},\xi)$, al instante t, se encontrarán todas en un elemento $d^3x'd^3\xi'$, situado en $(\mathbf{x}+\xi\delta t,\xi)$, al instante $t+\delta t$. Entonces, en ausencia de colisiones tenemos la igualdad

$$f(\mathbf{x}+\xi\delta t,\xi,t+\delta t) = f(\mathbf{x},\xi,t)$$

Cuando sí existen colisiones, la igualdad anterior debe de ser modificada. Escribimos

$$f(\mathbf{x}+\xi\delta t,\xi,t+\delta t) - f(\mathbf{x},\xi,t) = \left(\frac{\partial f}{\partial t}\right)_{colisiones}\delta t$$

que define el cambio en la distribución debido a las colisiones. Al hacer una expansión a primer orden en δt, obtenemos la ecuación de evolución para la función de distribución al tender $\delta t \to 0$:

$$\left(\frac{\partial}{\partial t} + \xi \cdot \nabla\right) f(\mathbf{x}, \xi, t) = \left(\frac{\partial f}{\partial t}\right)_{colisiones} \tag{46}$$

donde ∇ es el operador gradiente con respecto a $\mathbf{x}$, esta es una forma particular de la ecuación de Boltzmann [15, 25].

El lado derecho de esta ecuación, generalmente conocido como la integral de colisiones, explica los cambios de f debido a las colisiones entre partículas. Las partículas se mueven libremente por un momento; después dos partículas colisionan, y las velocidades de ambas partículas cambian. La integral de colisiones contiene la sección transversal de dispersión para una colisión en particular, y si suponemos que las colisiones ocurren entre pares de partículas (colisiones binarias), es cuadrática en la distribución f. Esta no-linealidad cuadrática es la causante de la dificultad en la ecuación de Boltzmann. En el modelo cinético más simple, la integral de colisiones es aproximada por

$$\left(\frac{\partial f}{\partial t}\right)_{colisiones} = -\frac{1}{\lambda}(f - f_{equilibrio})$$

En donde λ es una constante que representa la razón con la cual la función de distribución se aproxima a su forma en el equilibrio térmico. Esta aproximación es conocida como la aproximación BGK pues fue desarrollada por P.L. Bhatnagar, E.P. Gross, y M. Krook en los años 1950's [25]. En estos modelos se ignoran los detalles de las colisiones entre partículas reemplazándolos por el término anterior que expresa que las colisiones llevan al sistema a un estado de equilibrio local definido por la función de distribución de Maxwell-Boltzmann:

$$f_{equilibrio} \equiv \frac{\rho}{(2\pi RT)^{\frac{D}{2}}} exp(-\frac{(\xi - u)^2}{2RT}),$$

donde R es la constante del gas ideal, D es la dimensión del espacio de posiciones, y ρ, u y T son la densidad de masa macroscópica, la velocidad, y la temperatura respectivamente.

4.2. Discretización de la Ecuación de Boltzmann

En el siguiente análisis utilizaremos la ecuación de Boltzmann con la aproximación BGK, o aproximación con tiempo de relajación único:

$$\frac{\partial f}{\partial t} + \xi \cdot \nabla f = -\frac{1}{\lambda}(f - g) \tag{47}$$

En donde $f \equiv f(\mathbf{x}, \xi, t)$ es la función de distribución para una partícula, ξ es la velocidad de la partícula, λ es el tiempo de relajación debido a las colisiones

y g es la función de distribución Maxwell-Boltzmann que en la sección anterior se denominó $f_{equilibrio}$.

Las variables macroscópicas ρ, u y T son los momentos de la función de distribución f:

$$\rho = \int f d\xi = \int g d\xi, \tag{48}$$

$$\rho \mathbf{u} = \int \xi f d\xi = \int \xi g d\xi, \tag{49}$$

$$\rho \varepsilon = \frac{1}{2} \int (\xi - u)^2 f d\xi = \frac{1}{2} \int (\xi - u)^2 g d\xi. \tag{50}$$

La energía ε puede también escribirse en términos de la temperatura T como:

$$\varepsilon = \frac{D_0}{2} RT = \frac{D_0}{2} N_A k_B T, \tag{51}$$

en donde $D_0, N_{A,}$ y k_B son el numero de grados de libertad de una partícula, el número de Avogadro, y la constante de Boltzmann respectivamente.

4.3. Deducción de la ecuación de lattice Boltzmann y su función de distribución en equilibrio

La ecuación de lattice Boltzmann contiene los siguientes ingredientes: una ecuación de evolución con el tiempo y el espacio discretizados; constricciones de conservación en la forma de las ecuaciones (48), (49), y (50); y una función de distribución adecuada que nos lleve a las ecuaciones de Navier-Stokes.

4.3.1. Discretización del tiempo

La expresión correcta para la ecuación de evolución puede ser encontrada al notar que un elemento con posición $\mathbf{x}$ y velocidad ξ al tiempo t está en la posición $\mathbf{x} + \xi\delta t$ al tiempo $t + \delta t$, y el cambio en su distribución de probabilidad en el pequeño intervalo δt es

$$\frac{f(\mathbf{x} + \xi\delta t, \xi, t + \delta t) - f(\mathbf{x}, \xi, t)}{\delta t} = \left(\frac{\partial f}{\partial t} + \xi \cdot \nabla f \right) + O(\delta t). \tag{52}$$

Aqui podemos reemplazar el operador diferencial por su contraparte en diferencias finitas . Si omitimos los términos de orden $O(\delta t)$ en el lado derecho de la ecuación (52), entonces la ecuación (47) se convierte en:

$$f(\mathbf{x} + \xi\delta t, \xi, t + \delta t) - f(\mathbf{x}, \xi, t) = \frac{1}{\tau} \left[f(\mathbf{x}, \xi, t) - g(\mathbf{x}, \xi, t) \right] \tag{53}$$

donde $\tau \equiv \frac{\lambda}{\delta t}$ es el tiempo de relajación adimensional. Por lo tanto, la ecuación (53) es correcta a primer orden en δt. La ecuación (53) es la ecuación de evolución de la función de distribución f con el tiempo discreto.

Aunque g esté escrita como una función explicita de t , la dependencia temporal de g cae en las variables hidrodinámicas $\rho, \mathbf{u}$, y T, es decir, $g(\mathbf{x}, \xi, t) = g(\mathbf{x}, \xi; \rho, \mathbf{u}, T)$. Por lo tanto, uno debe primero encontrar a $\rho, \mathbf{u}$, y T antes de construir la función de distribución de equilibrio, g. Entonces, el cálculo de $\rho, \mathbf{u}$, y T se convierte en uno de los pasos más importantes en la discretización de la ecuación de Boltzmann.

4.3.2. Cálculo de los momentos hidrodinámicos

Para evaluar numéricamente los momentos hidrodinámicos dados por las ecuaciones (48), (49), y (50), se debe lograr una discretización apropiada en el espacio de velocidades ξ para que $f(\mathbf{x} + \xi\delta t, \xi, t + \delta t)$ calculada a partir de $f(\mathbf{x}, \xi, t)$ represente un valor en la malla, es decir que $\mathbf{x} + \xi\delta t$ sea un punto en la malla. Con dicha discretización, la integración en el espacio de velocidades (con función de peso g) puede ser aproximada por una cuadratura hasta cierto grado de exactitud, esto es,

$$\int \psi(\xi) g(\mathbf{x}, \xi, t) d\xi = \sum_{\alpha} W_{\alpha} \psi(\xi_{\alpha}) g(\mathbf{x}, \xi_{\alpha}, t), \tag{54}$$

en donde $\psi(\xi)$ es un polinomio de ξ, W_{α} es el coeficiente de peso para la cuadratura, y ξ_{α} es el conjunto de velocidades discreto, o las abscisas de la cuadratura. Tomando esto en cuenta, los momentos hidrodinámicos pueden ser calculados por:

$$\rho = \sum_{\alpha} f_{\alpha} = \sum_{\alpha} g_{\alpha}, \tag{55}$$

$$\rho \mathbf{u} = \sum_{\alpha} \xi_{\alpha} f_{\alpha} = \sum_{\alpha} \xi_{\alpha} g_{\alpha}, \tag{56}$$

$$\rho \varepsilon = \frac{1}{2} \sum_{\alpha} (\xi_{\alpha} - \mathbf{u})^2 f_{\alpha} = \frac{1}{2} \sum_{\alpha} (\xi_{\alpha} - \mathbf{u})^2 g_{\alpha}, \tag{57}$$

en donde

$$f_{\alpha} \equiv f_{\alpha}(\mathbf{x}, t) \equiv W_{\alpha} f(\mathbf{x}, \xi_{\alpha}, t), \tag{58}$$

$$g_{\alpha} \equiv g_{\alpha}(\mathbf{x}, t) \equiv W_{\alpha} g(\mathbf{x}, \xi_{\alpha}, t). \tag{59}$$

Es importante notar que f_{α} y g_{α} tienen unidades de $f d\xi$.

4.3.3. Aproximación de bajo numero de Mach

La función de distribución de equilibrio en la ecuación de lattice Boltzmann se obtiene por una expansión truncada de bajas velocidades (low-Mach-number approximation) [14] en donde g toma la forma:

$$g = \frac{\rho}{(2\pi RT)^{\frac{D}{2}}} exp(-\frac{\xi^2}{2RT}) exp \left\{ \frac{(\xi \cdot \mathbf{u})}{RT} - \frac{\mathbf{u}^2}{2RT} \right\}$$

$$= \frac{\rho}{(2\pi c_s^2)^{\frac{D}{2}}} exp(-\frac{\xi^2}{2c_s^2}) \times \left\{ 1 + \frac{(\xi \cdot \mathbf{u})}{c_s^2} + \frac{(\xi \cdot \mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u}^2}{2c_s^2} \right\} + O(M^3) \qquad (60)$$

en donde M es el número e Mach dado por u/c_s. Esta aproximación se logra obtener al considerar a las partículas como elementos que obedecen la ecuación de un gas ideal, en donde la velocidad del sonido c_s se obtiene al tomar la derivada

$$c_s^2 = \frac{\partial p}{\partial \rho}$$

Así, tomando la ecuación de estado propuesta por Chen & Doolen [4] $p = \rho/3$, obtenemos la relación $RT = c_s^2 = 1/3$.

Con la función de distribución de equilibrio escrita de esta manera, lo único que falta para que la ecuación (53) se convierta en la ecuación de lattice Boltzmann es la discretización del espacio fase. Por conveniencia, la siguiente notación será utilizada para la función de distribución de probabilidad con la expansión de bajo numero de Mach:

$$f^{(eq)} = \frac{\rho}{(2\pi RT)^{\frac{D}{2}}} exp(-\frac{\xi^2}{2RT}) \times \left\{ 1 + \frac{(\xi \cdot \mathbf{u})}{RT} + \frac{(\xi \cdot \mathbf{u})^2}{2(RT)^2} - \frac{\mathbf{u}^2}{2RT} \right\} \qquad (61)$$

Aunque $f^{(eq)}$ solamente retiene los términos hasta $O(\mathbf{u}^2)$, es trivial el mantener términos de orden superior de $\mathbf{u}$ en la expansión si son necesarios.

4.3.4. Discretización del espacio fase

Existen dos consideraciones para lograr la discretización del espacio fase. Primero, la discretizacion del espacio de velocidades esta acoplada a la del espacio de configuración de tal manera que una estructura de lattice (o malla) es obtenida. Esta es una característica especial de la ecuación de lattice Boltzmann [12]. Segundo, la cuadratura debe ser escogida para obtener las ecuaciones de Navier-Stokes.

Al deducir las ecuaciones de Navier-Stokes desde la ecuación de Boltzmann via el análisis Chapman-Enskog [17], los primeros dos ordenes de aproximación de la función de distribución deben de ser considerados. Por lo tanto, dada la función de distribución de equilibrio, la cuadratura utilizada debe servir para calcular los momentos hidrodinámicos ρ, $\mathbf{u}$, y T. El cálclulo de los momentos hidrodinámicos de $f^{(eq)}$ es equivalente a evaluar la siguiente integral en general:

$$I = \int \psi(\xi) f^{(eq)} d\xi = \frac{\rho}{(2\pi RT)^{D/2}} \int \psi(\xi) exp(-\frac{\xi^2}{2RT}) \times \left\{ 1 + \frac{(\xi \cdot \mathbf{u})}{RT} + \frac{(\xi \cdot \mathbf{u})^2}{2(RT)^2} - \frac{\mathbf{u}^2}{2RT} \right\} d\xi, \qquad (62)$$

en donde $\psi(\xi)$ es un polinomio de ξ. La integral anterior tiene la siguiente estructura:

$$\int e^{-x^2}\psi(x)dx,$$

que puede ser calculada numéricamente con la cuadratura de tipo Gaussiano. El objetivo es usar la cuadratura para evaluar los momento hidrodinámicos $[\rho, \mathbf{u}, T]$ dadas por las ecuaciones (48) (49) y (50). Mediante una discretización apropiada del espacio fase, podemos evaluar la integral con la exactitud deseada. Mientras tanto, la ecuación de lattice-Boltzmann con una función de distribución de equilibrio apropiada puede ser deducida.

4.3.5. Modelo 2-dimensional de 9 velocidades en una lattice cuadrada

El modelo más utilizado en dos dimensiones es el modelo de nueve velocidades en un espacio de lattice cuadrada. El sistema coordenado Cartesiano es utilizado. De esta manera y sin pérdida de generalidad $\psi(\xi)$ puede tomar la forma

$$\psi(\xi) = \xi_x^m \xi_y^n,$$

en donde ξ_x y ξ_y son las componentes x y y de ξ. La integral de los momentos, definida por la ecuación (71), se convierte en

$$I = \int \psi(\xi)_{m,n} f^{(eq)} d\xi = \frac{\rho}{\pi}(\sqrt{2RT})^{m+n}\left\{\left(1 - \frac{\mathbf{u^2}}{2RT}\right) I_m I_n + \frac{2(u_x I_{m+1} I_n + u_y I_m I_{n+1})}{\sqrt{2RT}} + \frac{u_x^2 I_{m+2} I_n + 2u_x u_y I_{m+1} I_{n+1} + u_y^2 I_m I_{n+2}}{RT}\right\}, \quad (63)$$

en donde

$$I_m = \int_{-\infty}^{+\infty} e^{-\varsigma^2}\varsigma^m d\varsigma, \;\; \varsigma = \xi/\sqrt{2RT}$$

es el m-ésimo momento de la función de peso $e^{-\varsigma^2}$ sobre el eje real. Con la variable normalizada $\varsigma = \xi/\sqrt{2RT}$. La fórmula de Hermite de tercer orden [9] es usada para evaluar I_m con el propósito de deducir el modelo de lattice Boltzmann con 9 velocidades:

$$I_m = \sum_{j=1}^{3} \omega_j \varsigma_j^m.$$

Las tres abscisas de la cuadratura son

$$\varsigma_1 = -\sqrt{3/2}, \;\; \varsigma_2 = 0, \;\; \varsigma_3 = \sqrt{3/2}, \quad (64)$$

y los coeficientes de peso correspondientes son

$$\omega_1 = \sqrt{\pi}/6, \;\; \omega_2 = 2\sqrt{\pi}/3, \;\; \omega_3 = \sqrt{\pi}/6. \tag{65}$$

De esta manera, la integral

$$I = \frac{\rho}{\pi}\sum_{i,j=1}^{3}\omega_i\omega_j\psi\left(\xi_{i,j}\right)\left\{1 + \frac{(\xi_{i,j}\cdot\mathbf{u})}{RT} + \frac{(\xi_{i,j}\cdot\mathbf{u})}{2(RT)^2} - \frac{\mathbf{u^2}}{2RT}\right\}, \tag{66}$$

en donde $\xi_{i,j} = (\xi_i, \xi_j) = \sqrt{2RT}(\varsigma_i, \varsigma_j)$. De esta expresión podemos identificar a la función de distribución de equilibrio con

$$f_{i,j}^{(eq)} = \frac{\omega_i\omega_j}{\pi}\rho\left\{1 + \frac{(\xi_{i,j}\cdot\mathbf{u})}{RT} + \frac{(\xi_{i,j}\cdot\mathbf{u})}{2(RT)^2} - \frac{\mathbf{u^2}}{2RT}\right\}. \tag{67}$$

Si utilizamos la notación de

$$\mathbf{e}_\alpha = \begin{cases} (0,0), & \alpha = 0 \\ (\cos\theta_\alpha, \sin\theta_\alpha), & \theta_\alpha = (\alpha-1)\pi/2, \quad \alpha = 1,2,3,4 \\ \sqrt{2}(\cos\theta_\alpha, \sin\theta_\alpha), & \theta_\alpha = (\alpha-5)\pi/2 + \pi/4, \quad \alpha = 5,6,7,8 \end{cases} \tag{68}$$

y

$$w_\alpha = \frac{\omega_i\omega_j}{\pi} = \begin{cases} 4/9, & i=j=2, \quad \alpha = 0 \\ 1/9, & i=1, j=2, ..., \quad \alpha = 1,2,3,4 \\ 1/36, & i=j=1, ..., \quad \alpha = 5,6,7,8 \end{cases} \tag{69}$$

Y si hacemos la sustitución de $RT = c_s^2 = 1/3$ $\left(\left\|\sqrt{2RT}\varsigma_1\right\| = \sqrt{3RT} = 1\right)$, obtenemos la función de districuión de equilibrio del modelo de lattice Boltzmann con 9-velocidades:

$$f_\alpha^{(eq)} = w_\alpha\rho\left\{1 + \frac{(e_\alpha\cdot\mathbf{u})}{c_s^2} + \frac{(e_\alpha\cdot\mathbf{u})}{2c_s^4} - \frac{\mathbf{u^2}}{2c_s^2}\right\}. \tag{70}$$

Esta función de distribución junto con la ecuación de evolución (53) conforman la ecuación de lattice Boltzmann en el esquema dos dimensional con nueve velocidades.

4.3.6. Modelo 3-dimensional de 15 velocidades en una lattice cuadrada

El modelo de 15 velocidades en un espacio de lattice cuadrada en 3D es una extensión directa del modelo dos dimensional con 9 velocidades. Las abscisas ξ_i, al igual que los coeficientes de peso correspondientes ω_i, de la cuadratura para evaluar los momentos es la misma. La función de distribución de equilibrio para el modelo de 15 velocidades está dada por

$$f_\alpha^{(eq)} = w_\alpha \rho \left\{ 1 + \frac{(e_\alpha \cdot \mathbf{u})}{c_s^2} + \frac{(e_\alpha \cdot \mathbf{u})}{2c_s^4} - \frac{\mathbf{u^2}}{2c_s^2} \right\}, \tag{71}$$

en donde

$$\mathbf{e}_\alpha = \begin{cases} (0,0,0), & \alpha = 0 \\ (\pm 1,0,0),(0,\pm 1,0),(0,0,\pm 1), & \alpha = 1,2,3,4,5,6 \\ (\pm 1,\pm 1,\pm 1), & \alpha = 7,8,...,14 \end{cases} \tag{72}$$

y

$$w_\alpha = \frac{\omega_i \omega_j \omega_k}{\pi^{3/2}} = \begin{cases} 2/9, & i=j=k=2, \quad \alpha = 0 \\ 1/9, & i=j=2, k=1,..., \quad \alpha = 1,2,...,6 \\ 1/72, & i=j=k=1,..., \quad \alpha = 7,8,...,14 \end{cases}, \tag{73}$$

Aquí podemos notar que $f^{(eq)}$ del modelo de 15 velocidades es exactamente la misma que la del modelo de 9 velocidades excepto por los valores de los coeficientes w_α. También, La velocidad del sonido de ambos modelos es igual ya que en los dos $RT = c_s^2 = 1/3$.

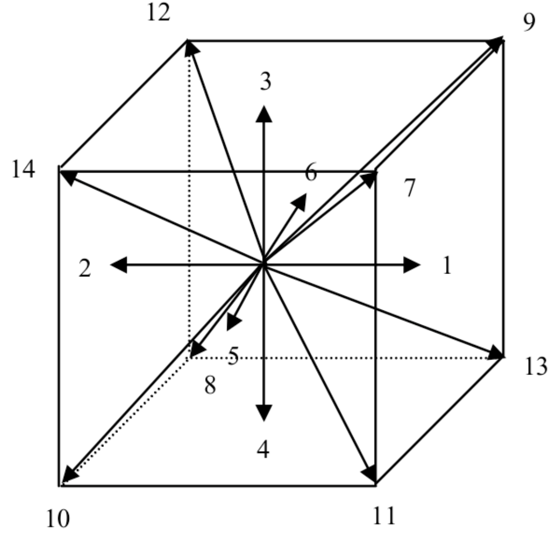


Figura 20: Celda característica para el modelo de lattice Boltzamnn de 15-velocidades.

Hasta aquí, hemos deducido modelos de lattice Boltzmann desde la ecuación BGK, la cual es completamente independiente de los autómatas de gases. La deducción conecta directamente la ecuación de lattice Boltzmann con la ecuación

de Boltzmann; así, el alcance de la ecuación de lattice Boltzmann puede descansar en el de la ecuación de Boltzmann, y los resultados rigurosos de la ecuación de Boltzmann pueden ser extendidos a la ecuación de lattice Boltzmann vía esta conexión explícita.

4.4. Incompresibilidad

En efecto, la ecuación de lattice Boltzmann fue propuesta para simular las ecuaciones de Navier-Stokes en su límite incompresible. De hecho, dichas ecuaciones pueden ser deducidas desde la ecuación de lattice Boltzmann mediante una aproximación de Chapman-Enskog [17] si las fluctuaciones en la densidad son supuestamente insignificantes. Desafortunadamente, este no es siempre el caso en las simulaciones computacionales al hacer uso del método de lattice Boltzmann. El efecto de compresibilidad en los modelos deducidos en la sección anterior pueden producir errores en las simulaciones numéricas [13]. Han habido intentos de reducir o eliminar el efecto compresible en el método de Lattice Boltzmann , sin embargo, los resultados no han sido satisfactorios debido a que los modelos no logran reproducir fenómenos no estacionarios. Debido a esto, es necesario mejorar el método para simulaciones en el límite incompresible, y más específicamente para flujos no estacionarios.

Idealmente, la incompresibilidad puede alcanzarse solamente cuando la densidad se convierte en una constante. Sin embargo, es prácticamente imposible mantener la densidad como una constante en los métodos de lattice Boltzmann. Sin embargo, en las simulaciones numéricas se utiliza un gradiente de presión para forzar el sistema, y el gradiente de presiones se alcanza al mantener un gradiente de densidades en el sistema. Bajo estas circunstancias, la suposición de una densidad constante se vuelve inválida, y la magnitud de la fluctuación en la densidad puede llegar a ser significativa.

Es sabido que el método de lattice Boltzmann es solamente aplicable a la hidrodinámica de bajo número de Mach, debido a que una expansión de bajas velocidades es utilizada (implícitamente) en la deducción de las ecuaciones de Navier Stokes desde la ecuación de lattice Boltzmann [14]. Debe resaltarse que el límite de bajo número de Mach es equivalente al límite incompresible. Así, a continuación se presentará el modelo ideado por He et. al [13] para el límite incompresible de las ecuaciones de Navier Stokes. En el siguiente análisis, la deducción de dichas ecuaciones se hará a través el ejemplo del modelo de 15-velocidades en tres dimensiones, sin embargo esta aproximación se puede aplicar a otros modelos en dos y tres dimensiones en general.

Como acabamos de ver, el modelo BGK de 15 velocidades en una lattice cuadrada evoluciona en base a las siguientes 15 velocidades discretas,

$$e_\alpha = \begin{cases} (0,0,0), & \alpha = 0 \\ (\pm 1,0,0)c, (0,\pm 1,0)c, (0,0,\pm 1)c, & \alpha = 1,2,3,4,5,6 \\ (\pm 1,\pm 1,\pm 1), & \alpha = 7,8,...,14 \end{cases}$$

en donde $c = \delta_x/\delta_t$, y δ_x y δ_t son la distancia entre nodos y el tamaño del

paso de tiempo, respectivamente. La ecuación de evolución del sistema es

$$f_\alpha(\mathbf{x}+\mathbf{e}_\alpha\delta_t, t+\delta_t) - f_\alpha(\mathbf{x},t) = -\frac{1}{\tau}\left[f_\alpha(\mathbf{x},t) - f_\alpha^{(eq)}(\mathbf{x},t)\right]$$

en donde τ es el tiempo de relajación, y la función de distribución de equilibrio $f_\alpha^{(eq)}$está dada por:

$$f_\alpha^{(eq)} = w_\alpha\rho\left\{1 + \frac{3\,(e_\alpha\cdot\mathbf{u})}{c^2} + \frac{9\,(e_\alpha\cdot\mathbf{u})}{2c^4} - \frac{3\mathbf{u^2}}{2c^2}\right\}.$$

con coeficientes de peso

$$w_\alpha = \begin{cases} 2/9, & i=j=k=2, \quad \alpha=0 \\ 1/9, & i=j=2, k=1,..., \quad \alpha=1,2,...,6 \\ 1/72, & i=j=k=1,..., \quad \alpha=7,8,...,14 \end{cases},$$

y haciendo énfasis en que en la función de distribución de equilibrio anterior,

$$\frac{\mathbf{u}}{c} \approx M.$$

Es claro que en un fluido incompresible la densidad es (aproximadamente) una constante, digamos ρ_0, y la fluctuación en la densidad, $\delta\rho$, debe de ser del orden $O(M^2)$ en el límite $M \to 0$. Si nosotros hacemos la sustitución explícita $\rho = \rho_0 + \delta\rho$ en la función de distribución de equilibrio, $f_\alpha^{(eq)}$, y despreciamos los términos proporcionales a $\delta\rho(\mathbf{u}/c)$, y $\delta\rho(\mathbf{u}/c)^2$, entonces, la función de distribución de equilibrio se convierte en

$$f_\alpha^{(eq)}(\mathbf{x},t) = w_\alpha\left\{\rho + \rho_0\left[\frac{3\,(e_\alpha\cdot\mathbf{u})}{c^2} + \frac{9\,(e_\alpha\cdot\mathbf{u})}{2c^4} - \frac{3\mathbf{u^2}}{2c^2}\right]\right\} \tag{74}$$

La función de distribución anterior es la función de distribución de equilibrio del modelo de lattice Boltzmann incompresible.

Ya que es una práctica común el utilizar la presión, p, como una variable independiente en las ecuaciones de Navier Stokes, introducimos una función de distribución para presión local

$$p \equiv c_s^2 f_\alpha \tag{75}$$

en donde c_s es la velocidad del sonido y $c_s = c/\sqrt{3}$ para el modelo de 15 velocidades. De este modo, la ecuación de evolución del sistema se convierte en

$$p_\alpha(\mathbf{x}+\mathbf{e}_\alpha\delta_t, t+\delta_t) - p_\alpha(\mathbf{x},t) = -\frac{1}{\tau}\left[p_\alpha(\mathbf{x},t) - p_\alpha^{(eq)}(\mathbf{x},t)\right] \tag{76}$$

donde

$$p_\alpha^{(eq)} = c_s^2 f_\alpha^{(eq)} = w_\alpha\left\{p + p_0\left[\frac{3\,(e_\alpha\cdot\mathbf{u})}{c^2} + \frac{9\,(e_\alpha\cdot\mathbf{u})}{2c^4} - \frac{3\mathbf{u^2}}{2c^2}\right]\right\}, \tag{77}$$

con $p = c_s \rho$, y $p_0 = c_s^2 \rho_0$. Con la representación de presiones, la presión, p, y la velocidad, $\mathbf{u}$, estan dadas por

$$p = \sum_\alpha p_\alpha \tag{78}$$

$$p_0 \mathbf{u} = \sum_\alpha \mathbf{e}_\alpha p_\alpha \tag{79}$$

El modelo incompresible es el compuesto por las ecuaciones (81) y (82). Nuevamente, al hacer una aproximación Chapman-Enskog [13], las ecuaciónes de Navier-Stokes deducidas de este modelo incompresible son:

$$\frac{1}{c_s^2} \frac{\partial P}{\partial t} + \nabla \cdot \mathbf{u} = 0 \tag{80}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla P + \nu \nabla^2 \mathbf{u} \tag{81}$$

en donde $P = p/\rho_0$, y la viscocidad cinemática es

$$\nu = \frac{(2\tau - 1)}{6} \frac{\delta_x^2}{\delta_t}$$

.

Si escribimos las ecuaciones (83) y (84) de manera adimensional :

$$\frac{1}{T} \frac{\partial P'}{\partial t'} + \frac{c_s}{L} \nabla' \cdot \mathbf{u}' = 0 \tag{82}$$

donde $P' = P/c_s^2$, $t' = t/T$, $\nabla' = L\nabla, \mathbf{u}' = \mathbf{u}/c_s$, y L y T son un tiempo y una distancia características.

En el caso de flujos estacionarios, $\partial P/\partial t = 0$, y la condición de flujo incompresible, $\nabla \cdot \mathbf{u} = 0$ se satisface idénticamente. Así, la única condición que uno debe satisfacer en las simulaciones numéricas de flujos incompresibles es

$$M \ll 1 \tag{83}$$

En el caso de flujos no estacionarios, una condición adicional debe de ser cumplida. De la ecuación (85) se puede ver que para que el término $\partial P'/\partial t'$ sea despreciable, la siguiente condición debe de ser satisfecha:

$$T \gg L/c_s \tag{84}$$

Es decir, que el tiempo, T, en el cual el flujo atraviesa un cambio macroscópico (en el rango de la distancia L) debe de ser mayor que el tiempo, L/c_s, que le toma a una señal de sonido atravesar la distancia L, para que la propagación de las interacciones (a través de una onda de presión o fluctuación de la densidad) en el fluido pueda ser tomada como instantánea. Así, en el método de lattice Boltzmann, la variación temporal de la presión impuesta en las condiciones de frontera no debe de ser muy rápida por la razón antes mencionada,

y la variación espacial de la presión (o densidad) no deberá de ser muy grande. Entonces, las condiciones (86) y (87) deberán ser cumplidas simultáneamente en las simulaciones de flujos incompresibles dependientes del tiempo.

4.5. Condiciones de Frontera

Las condiciones de frontera en el método de lattice Boltzmann han sido directamente adaptadas del método de gases autómatas. El método de "halfway bounce-back", y un método de extrapolación son discutidos a continuación. el primer método nos servirá para implementar una condición de "no-slip", o velocidad cero en las fronteras sólidas mientras que el segundo nos permitirá imponer una presión en las fronteras abiertas. Este método es necesario ya que los flujos de Poiseuille y Womersley son generados por gradientes de presión conocidos en las fronteras.

4.5.1. Halfway Bounce-Back

Por el esquema "halfway bounce-back" nos referimos que cuando una partícula alcanza un nodo de una pared, la partícula se dispersará de regreso al campo de flujo en dirección opuesta a la que llevaba antes de llegar al muro. En el MLB, la operación de "halfway bounce-back" nos lleva a la conservación de la masa y una velocidad cero en los muros. La manera en la que se manejan las fronteras en el MLB es bastante simple comparada con otros esquemas numéricos. Sin embargo, el esquema "halfway bounce-back" solamente tiene una exactitud a primer orden en el número de Mach. Esto degrada el método ya que los puntos dentro de la malla tienen una exactitud a segundo orden. El método de bounce back es muy sencillo de implementar y esta basado en la ecuación

$$f_{-\alpha}(\mathbf{x}, t + \delta t) = f_{\alpha}(\mathbf{x} + \xi \delta t, t + \delta t) \tag{85}$$

en donde $f_{-\alpha} = f(\mathrm{x}, -\mathbf{e}_{\alpha}, t)$. La idea principal de estas condiciones es la de reflejar la partícula entrante para alcanzar las condiciones de no deslizamiento. Esto se logra al calcular la colisión en los nodos sólidos para después invertir la dirección en de la partícula entrante. La colisión de las partículas en el nodo sólido simplemente manda a la partícula incidente hacia dentro del fluido. Se ha demostrado que este esquema nos lleva a una velocidad cero en los nodos sólidos.

4.5.2. Esquema de extrapolación

A continuación presentaremos el esquema de extrapolación ideado por Martínez et. al [5] para lograr un segundo orden de exactitud en las fronteras con condiciones de presión o velocidad impuestas. La idea del procedimiento es relativamente simple: Para cualquier flujo fluido, podemos suponer que existe una capa adicional de nodos que se encuentran fuera del dominio (como la capa F-G-H en la Figura 21).

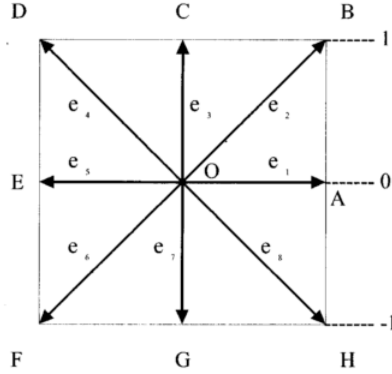


Figura 21: Celda característica para el modelo de lattice Boltzamnn de 9-velocidades.

Cada paso de tiempo antes de enviar las distribuciones a los nodos adyacentes, las funciones de distribución de los nodos fuera del dominio son calculadas utilizando una extrapolación de segundo orden, basada en el valor de la función de distribución en la capa fronteriza (E-O-A) y la capa de nodos que están un paso dentro del fluido (D-C-B). Esto implica que forzamos la siguiente condición en los nodos imaginarios para cada paso de tiempo antes de distribuir las funciones de distribución:

$$f_i^{-1} = 2f_i^0 - f_i^1, \tag{86}$$

en donde f_i^{-1} , f_i^0 y f_i^1 son las funciones de distribución en la capa imaginaria, la capa fronteriza y la primera capa dentro del fluido, respectivamente.

Después de esta extrapolación, las funciones de distribución son repartidas para todos los nodos, incluyendo la capa adicional imaginaria. En el momento de la colisión utilizamos el tipo de relajación BGK para todos los nodos, excepto los nodos fronterizos, en los cuales las condiciones de frontera para la presión (o la velocidad) son impuestas en la función de distribución de equilibrio.

Si quisiéramos imponer un gradiente de presiones entre la entrada y la salida del ducto, debemos escoger la densidad (que es proporcional a la presión) con valores ligeramente distintos en la entrada y la salida del canal. La velocidad puede o no tomar algún valor. De esta manera, (ver figura) $f_1, f_{2,}$ y f_8 en la capa imaginaria de la entrada del tubo, y$f_4, f_{5,}$ y f_6 en la capa imaginaria de la salida del tubo, son calculadas utilizando el esquema de extrapolación aqui mencionado.

4.6. Simulaciones

En esta sección se presentan los resultados que se obtuvieron al comparar la rapidez del método de lattice Boltzmann D2Q9 implementado en el CPU con un programa escrito en C, contra la rapidez del mismo método escrito en CUDA e implementado sobre la tarjeta de video utilizando la memoria de texturas. La comparación consistió en medir el tiempo que le llevaba a cada uno de los dos programas alcanzar un total de 10,000 pasos de tiempo de la ecuación de evolución del método variando el número de nodos que conforman al sistema. En la gráfica de la Figura 22 se presentan los resultados obtenidos.

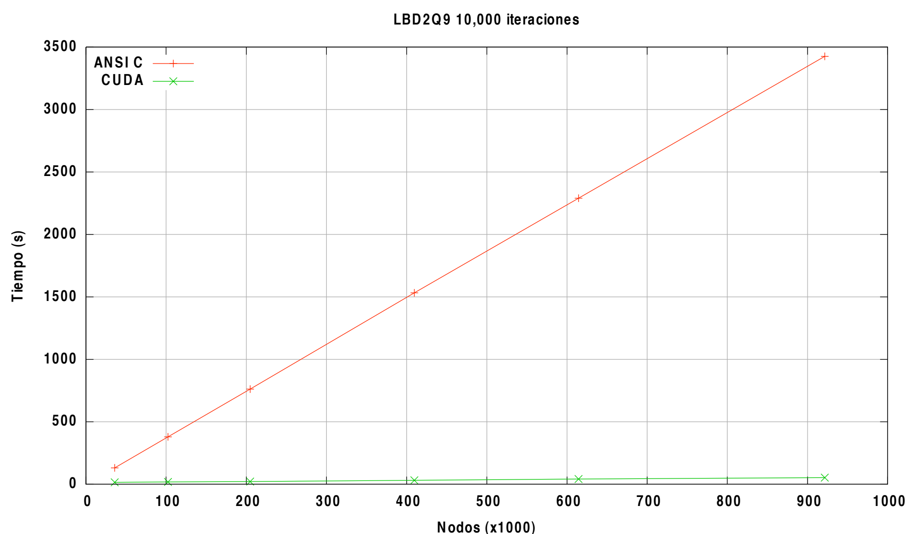


Figura 22: Comparación de la rapidez del método LBD2Q9 escrito en CUDA y en C.

Es posible concluir de las gráficas anteriores que la utilización del GPU para acelerar los cálculos del método de lattice Boltzmann D2Q9 brinda mejoras en el tiempo de ejecución de mas de dos ordenes de magnitud. Mientras que al CPU le tarda mas de una hora llegar a las 10,000 iteraciones en un sistema de $1,228,800$ nodos, el GPU las logra alcanzar en un minuto con diez segundos. Para las simulaciónes en tres dimensiones esperamos que éstas diferencias en la aceleración del método persistan.

Una vez comprobado el hecho de que el GPU brinda grandes mejorías en el tiempo de ejecución del método, se decidió explorar la diferencia en los tiempos de ejecución del método utilizando dos tipos distintos de las memorias que provee CUDA. Se desarrollaron dos códigos del método de lattice Boltzmann con esquema D2Q9, uno que utliza la memoria global y otro que utiliza la memoria de texturas como memoria principal. También aqui se midió el tiempo de ejecución que le tomó a los programas alcanzar 10,000 iteraciones del método

al variar el tamaño del sistema. Los resultados se muestran en la Figura 23:

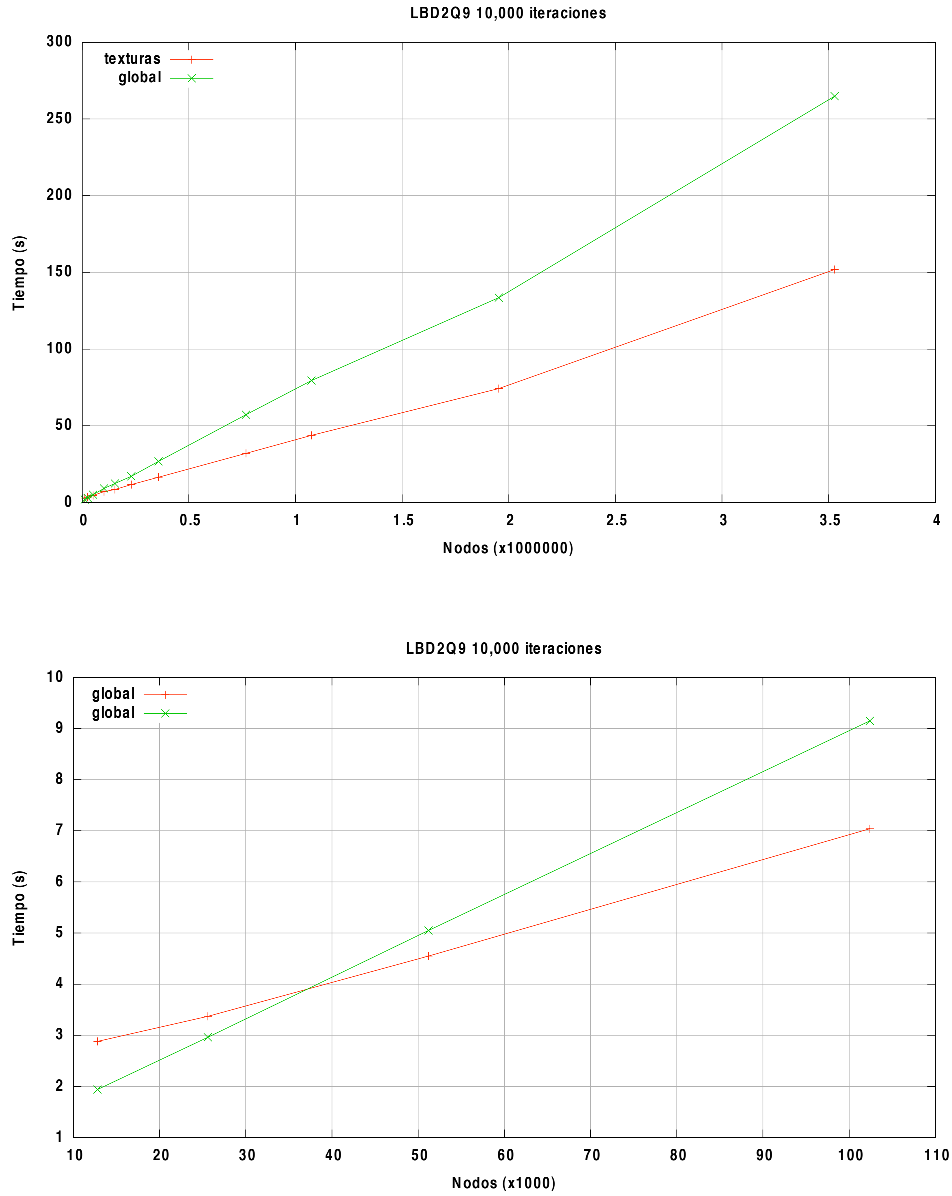


Figura 23: Desempeños de memoria para la ecuación de lattice Boltzmann 2D.

Nuevamente podemos notar que la implementación de la memoria de texturas alcanza un desempeño mucho mas alto para sistemas con un gran número de partículas. Aunque la memoria global brinda un menor tiempo de ejecución

para sistemas menores a 30,000 nodos, la memoria de texturas es mucho más rápida para sistemas con un mayor número de partículas. En particular, para un sistema de 3.5 millones de nodos a la memoria de texturas le lleva la mitad del tiempo de ejecución que le toma a la memoria global alcanzar 10,000 iteraciones de método, terminado el trabajo en menos de tres minutos.

En el siguiente capítulo se presentanlos resultados de distintas simulaciones utilizando el método de lattice Boltzmann pero con el esquema tridimensional D3Q15.

5. Validación y Resultados

Para validar los códigos desarrollados para realizar el propósito de esta tesis, a continuación se presentan los resultados obtenidos al simular dos flujos conocidos en la física de fluidos, el flujo de Poiseuille y el flujo de Wommersley, ambos discutidos en el capítulo 2 de esta tesis. En esta sección presentaremos los resultados obtenidos de las simulaciones hechas con un programa del método de Lattice Boltzmann con esquema de 15 velocidades en tres dimensiones. El programa fue desarrollado en el lenguaje de programación en paralelo CUDA y todas las simulaciones fueron hechas en una tarjeta NVIDIA GeForce 260 GTX. El código se desarrolló en el laboratorio de Acústica del departamento de Física de la Facultad de Ciencias de la UNAM. Los resultados numéricos de las simulaciones del flujo de Poiseuille y del flujo de Womersley son comparadas con su contraparte analítica. Mas adelante se presentarán los resultados de unas simulaciones sin contraparte analítica que consisten en visualizar las líneas tangentes al campo de velocidades de un flujo oscilatorio a la salida de un ducto inmerso en un fluido. Estas simulaciones estuvieron inspiradas en el flujo oscilatorio que se presenta en el SIBEO. Por último se presentarán las simulaciones de un flujo alrededor de un obstáculo esférico. Para todas las simulaciones descritas a continuación, el flujo es supuesto como laminar ($Re \ll 100$) y el numero de Mach es pequeño. Para todas las paredes se ha utilizado la condición de "halafway bounce-back" en los nodos para alcanzar condiciones de frontera de no deslizamiento; para la entrada y la salida del ducto se han utilizado fronteras de presión impuesta utilizando el método de extrapolación presentado en el capítulo anterior, fronteras periódicas o una combinación de ellas. En cada caso se especifica en detalle como se obtuvieron las simulaciones.

5.1. Flujo de Poiseuille

Hemos estudiado el flujo dentro de un ducto rígido con sección transversal circular debido a un gradiente de presión constante $\partial p/\partial x = \kappa_e$. El gradiente de presión fue implementado al fijar los valores de la presión en la entrada y en la salida utilizando el método de extrapolación presentado en la sección anterior. A continuación se presentan los resultados para distintos tamaños del sistema para flujos con un numero de Reynolds ≈ 5. El número de Reynolds fue calculado utilizando el radio del ducto como distancia característica y la velocidad máxima del flujo de Poiseuille como velocidad característica. Se observaron soluciones numéricas que concuerdan con las soluciones analíticas.

Para la simulación con un ducto de radio $a = 45$ nodos se utilizó un sistema con dimensiones de $N_x \times N_y \times N_z = 288 \times 96 \times 96$ con $2,654,208$ nodos. En los nodos de la entrada $x = 0$ y de la salida $x = N_x - 1$ del ducto, la condición de frontera implementada fue la de una presión constante. Los valores de la presión fueron fijados utilizando los valores $\rho = 2.8$ en la entrada, y $\rho = 3.1$ en la salida (Figura 24). En las paredes del tubo, se implementaron condiciones de no deslizamiento mediante el esquema de "halfway bounce-back" en los nodos sólidos para todas las simulaciones. La condición inicial fue $\mathbf{u} = 0.1$ en el interior

del canal para las tres simulaciones en donde se utilizó la velocidad máxima del flujo de Poiseuille para adimensionalizar la velocidad. El tiempo de relajación del método fué de $\tau = 2.75$, y como es común en los métodos de lattice Boltzmann $\delta_x = \delta_t = 1$.

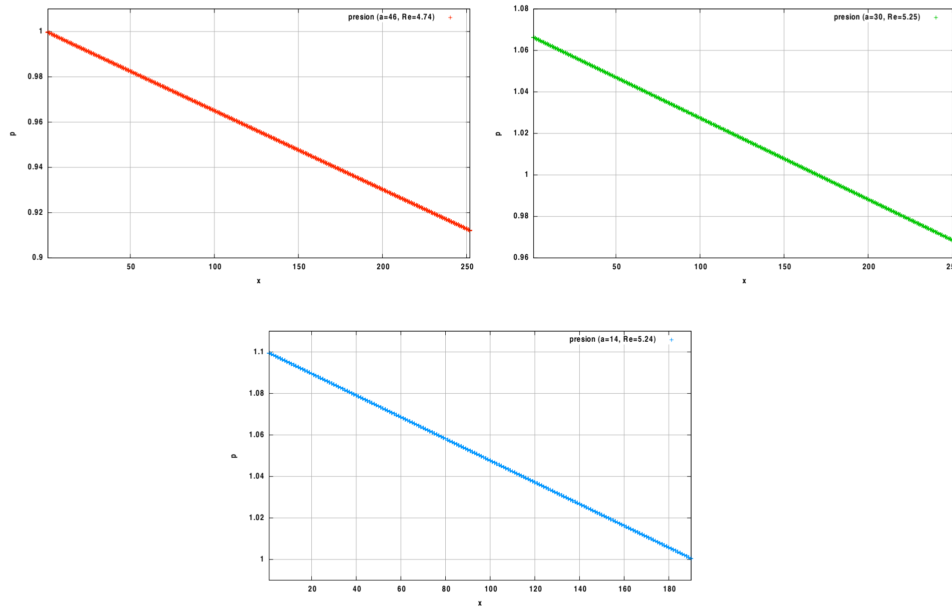


Figura 24: Presion a lo largo del ducto para simulación de radio a=46 a=30 y a=15 respectivamente

Para la simulación con un ducto de radio $a = 30$ se utilizó un sistema con dimensiones de $N_x \times N_y \times N_z = 256 \times 64 \times 64$ con $1, 048, 576$ nodos. La presión en los nodos de la entrada fue fijada utilizando un valor de $\rho = 2.9$ y en la salida en un valor de $\rho = 3.2$. El tiempo de relajación fué de $\tau = 1.75$.

Por último, para la simulación con un ducto de radio $a = 14$ se utilizó un sistema con dimensiones de $N_x \times N_y \times N_z = 192 \times 32 \times 32$ con $196, 608$ nodos. La presión en los nodos de la entrada fue fijada utilizando un valor de $\rho = 1.0$ y en la salida en un valor de $\rho = 1.1$. El tiempo de relajación fue de $\tau = 0.95$. La Figura 25 muestra los perfiles de la velocidad con las condiciones de presión fija en la entrada y salida del canal. Estas simulaciones fueron hechas para comprobar que el método lograra reproducir el mismo flujo de manera independiente a los parámetros utilizados. Es decir, se reprodujo el mismo flujo (mismo número de Reynolds) para distintos parámetros del sistema.

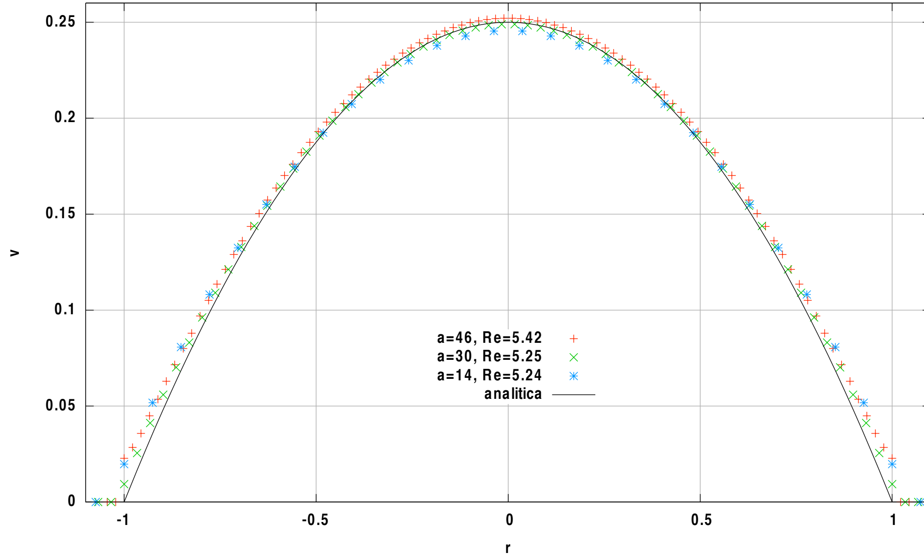


Figura 25: Perfiles de velocidad para flujode Poiseuille. En la leyenda 'a' representa el radio del ducto y 'Re' es el número de Reynolds de la simulación.

Para obtener las gráficas se dejó relajar el sistema por un numero finito de iteraciones hasta alcanzar un estado estacionario. El criterio de convergencia para todas las simulaciones fue obtenido al comparar los resultados de dos iteraciones sucesivas con un criterio menor a 10^{-3}. El criterio de estado estacionario esta dado por

$$\sum_i \frac{\|\mathbf{u}(\mathbf{x}_i, t+1) - \mathbf{u}(\mathbf{x}, t)\|}{\|\mathbf{u}(\mathbf{x}_i, t+1)\|} \leq 10^{-3}$$

en donde la suma es sobre todo el sistema. Generalmente se alcanza el estado estacionario después de ciertos miles de iteraciones, dependiendo del valor de el tiempo de relajación, o la viscosidad, y las condiciones iniciales del sistema. Dos tipos de mediciones fueron tomadas en las simulaciones. Una es la medida de la velocidad **u** en una sección transversal arbitraria del ducto. La otra es la medida de la presión a lo largo del canal. Todas las mediciones fueron tomadas después de alcanzar el estado estacionario.

Podemos notar claramente de las Figuras 24 y 25 que los resultados numéricos acuerdan con los resultados analíticos dentro de las capacidades del método reportadas en la literatura [1] . La combinación de los resultados mostrados en las Figuras 24 y 25 muestran que, no solamente los perfiles de velocidad son correctos en forma (parabólica) a lo largo del canal, sino que también la velocidad máxima del perfil es cuantitativamente correcta. Mas aún, la distribución de presiones es lineal a lo largo del canal. Todos los resultados numéricos están en acuerdo con los resultados analíticos dentro de los márgenes conocidos del

método. Por ejemplo, en la gráfica podemos notar que las simulaciones se alejan de la solución analítica cerca de las paredes del tubo. Esta discrepancia es atribuida a la manera de implementar condiciones de frontera, y esta analizada en la literatura [24].

5.2. Flujo de Womersley

También se hicieron simulaciones de flujos dependientes del tiempo, en este caso el flujo pulsátil en tres dimensiones se utilizó para validar el modelo para flujos no estacionarios. Como hemos mencionado, la configuración geométrica del flujo de Womersley es idéntica a la del flujo de Poiseuille, con la diferencia de que el flujo esta generado por un gradiente de presiones periódico en el tiempo $\partial p/\partial x = \kappa_e + \kappa_\phi(T)$, en donde T es el periodo de la oscilación. Para todas las simulaciones se utilizo una función senoidal $\kappa_\phi = A \cdot \sin(\omega t)$ donde $\omega = 2\pi/T$. Primero llevamos a cabo un conjunto de simulaciones con distintos números de Womersley $\Omega = 2, 3, 4$. En estas simulaciones se midieron los perfiles de velocidad para distintos tiempos dentro de un periodo. En las simulaciones se mantuvo fijo el tamaño del sistema en $N_x \times N_y \times N_z = 64 \times 64 \times 64$ o $262, 644$ nodos. Se llevaron a cabo simulaciones para tres distintos números de Womersley, para los cuales la amplitud de la oscilación del gradiente de presiones se mantuvo constante a lo largo del tiempo, sin embargo se varió el periodo y el valor del tiempo de relajación para alcanzar los distintos valores del numero de Womersley. La magnitud de la caída de presión total a lo largo del canal es de $\Delta p = 0.0033$ ($A = \Delta p/N_x$), ajustando la densidad en la salida en $\rho = 3.0$. Para la simulación con $\Omega = 4$, el periodo de el gradiente de presiones fue de $T = 2100$ pasos temporales, y el teimpo de relajación $\tau = 1.0$. Para la simulación con $\Omega = 3$, $T = 3000$ y $\tau = 1.1$. Y por último para la simulación con $\Omega = 2$, el periodo $T = 4600$ y $\tau = 1.4$. Las condiciones iniciales del campo de velocidades siempre fueron ajustadas a cero en todos los puntos el sistema.

Las Figuras 26, 27 y 28 muestran los perfiles de velocidad a través de una sección transversal a la mitad del canal $x = 31$ en cinco tiempos distintos $t = T, T/10, T/5, 3T/10$, y $4T/10$.

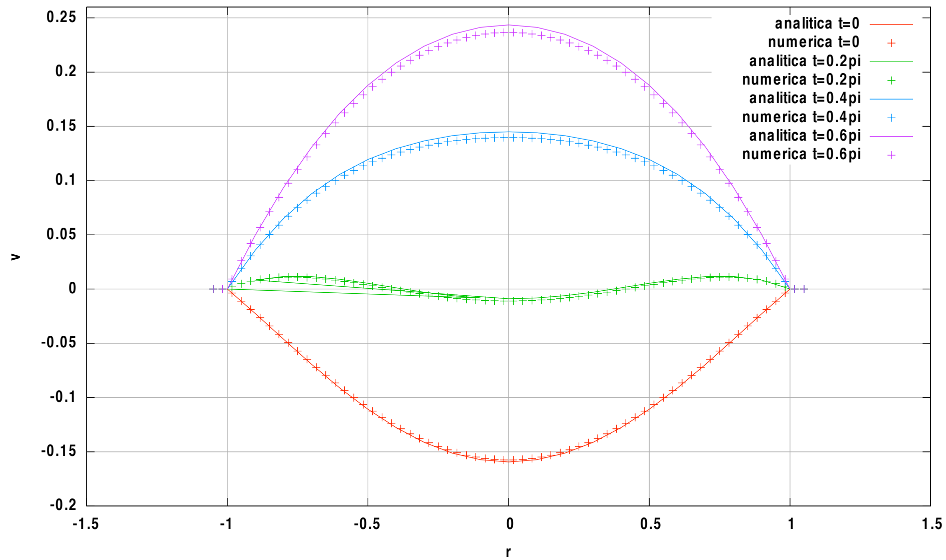


Figura 26: Perfil de velocidades oscilatorias para el flujo de Womersley en distinntos tiempos de la oscilación. $\Omega = 2$

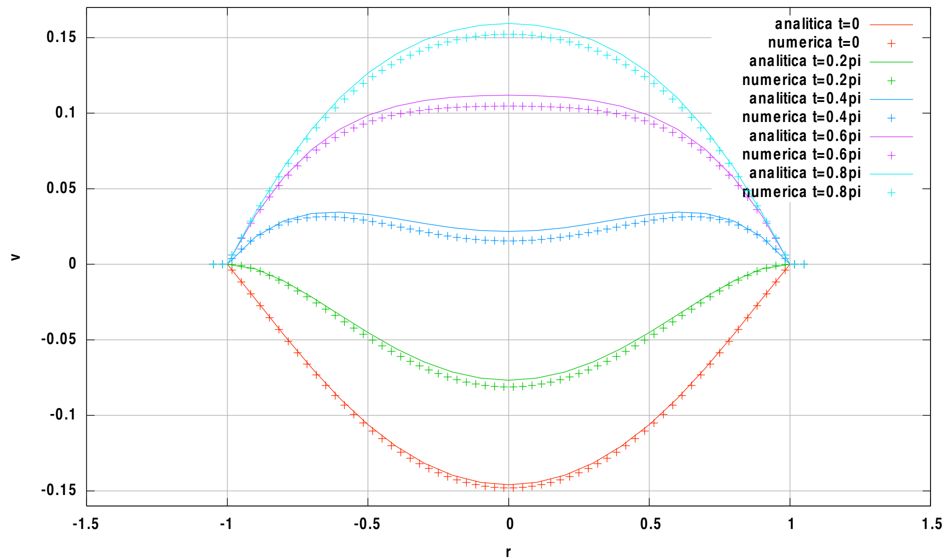


Figura 27: Perfil de velocidades oscilatorias para el flujo de Womersley en distinntos tiempos de la oscilación. $\Omega = 3$

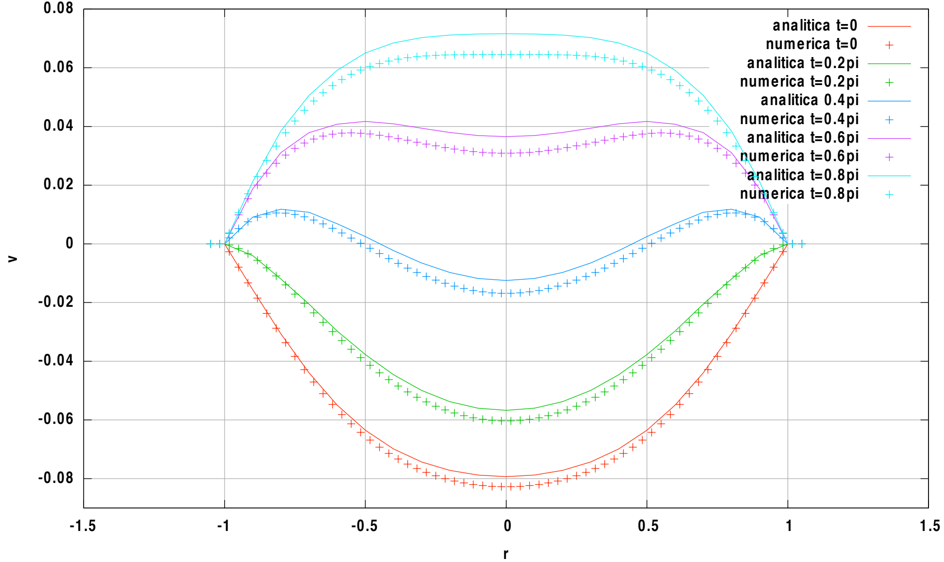


Figura 28: Perfil de velocidades oscilatorias para el flujo de Womersley en distinntos tiempos de la oscilación. $\Omega = 4$

En los resultados presentados anteriormente, el cálculo del campo de velocidades siempre comenzó con $2T$ pasos iniciales para obtener un criterio de convergencia:

$$\sum_i \frac{\|\mathbf{u}(\mathbf{x}_i, t+T) - \mathbf{u}(\mathbf{x}, t)\|}{\|\mathbf{u}(\mathbf{x}_i, t+T)\|} \leq 10^{-3}$$

en donde la suma es sobre el sistema en su totalidad.

En las Figuras 25, 26 y 27 se puede observar que existe un pequeño desfase entre las simulaciones y la teoría. Se ha encontrado este fenómeno reportado en la literatura [1] en donde se propone que este desfase es una función del tiempo y de τ. Artoli et al. encontraron que si se supone de entrada que la teoría se aparta de la simulación con un medio paso de tiempo, es decir, $t_{sim}\delta t = t_{teo} + 0.5\delta t$, el error se reduce por lo menos en un orden de magnitud para todos los valores de τ. Este desfase temporal puede ser atribuido a la manera en que las condiciones de frontera fueron implementadas en el método, es decir, cuando se utiliza el "halfway bounce-back" en los nodos podemos observar que el error es máximo cuando $t = T/4$, y esto es atribuido al alto gradiente de presión en este tiempo.

5.3. Ducto abierto

Al lograr validar el código para casos con solución analítica, se decidió cambiar la geometría y las condiciones del sistema para simular otro tipo de flujos. En particular, el flujo que ocurre en la boca del tubo resonante del SIBEO (Sistema de Bombeo por Energía de Olas) sirvió de inspiración para realizar un par de simulaciones que se presentan a continuación.

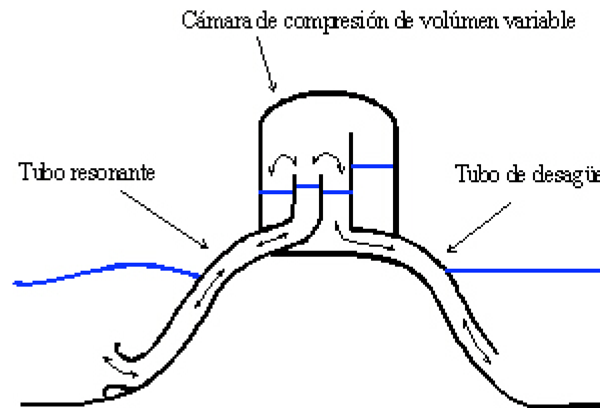


Figura 29: Geometrías de las dos simulaciones.

El SIBEO, esquematizado en la Figura 29, es un sistema de bombeo de agua autónomo que aprovecha la energía de oleaje del mar. El sistema fue diseñado por investigadores del Instituto de Limnología y Ciencias del Mar de la UNAM y está diseñado para bombear agua marina a puertos y lagunas costeras con el fin de reactivar su circulación e intercambio con el mar. El SIBEO utiliza la señal de presión producida por el oleaje para hacer resonar la cámara de compresión y generar un flujo en el tubo resonante que derrame agua dentro de la cámara. Ya que ésta agua no puede regresar al ducto resonante debido a la altura del ducto dentro de la cámara, el agua desciende a la reserva terrestre por la fuerza de gravedad [8]. Es sabido que la eficiencia de esta bomba se ve afectada por la geometría de la boca del ducto resonante.

Las simulaciones que se hicieron en relación a este sistema de bombeo consistieron en reproducir los flujos presentes en la boca del ducto resonante para dos distintas geometrías del ducto. La primera es la boca de un ducto cilíndrico con sección transversal circular, mientras que a la segunda se le añadió una boca en forma de trompeta generada por un medio círculo al final del ducto. Las geometrías se muestran en la siguiente figura.

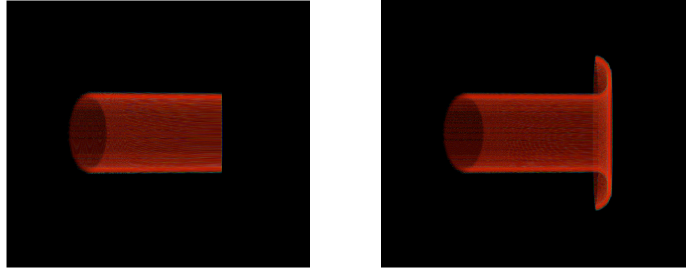


Figura 30: Geometrías de las dos simulaciones.

A continuación se presenta el campo de velocidades, proyectado en un cierto plano, de la simulación de un flujo tridimensional oscilatorio en la salida de estos ductos inmersos en un medio fluido. El plano que se muestra en las figuras siguientes es el plano que secciona el ducto transversalmente por la mitad. Se utilizo el método de lattice Boltzmann con esquema D3Q15. En ambos casos el tamaño del sistema fue de $N_x \times N_y \times N_z = 256 \times 96 \times 96$ o $2,359,296$ nodos. El radio del ducto fue de $a = 18$ nodos y el largo del ducto fue de 90 nodos.

El número de Reynolds de ambas simulaciones fué $Re = 3.44$ y el número de Womersley es $\Omega = 9.51$. Para las simulaciones se utilizó una función senoidal $\kappa_\phi = 0.1 \cdot \cos(\omega t)$ donde $\omega = 2\pi/T$, y T=3000 pasos de tiempo. El tiempo de relajación fue $\tau = 0.51$. Las Figuras 31 y 32 muestran el campo de velocidades para distintos tiempos dentro del periodo de oscilación de la presión para el ducto con sección transversal circular.

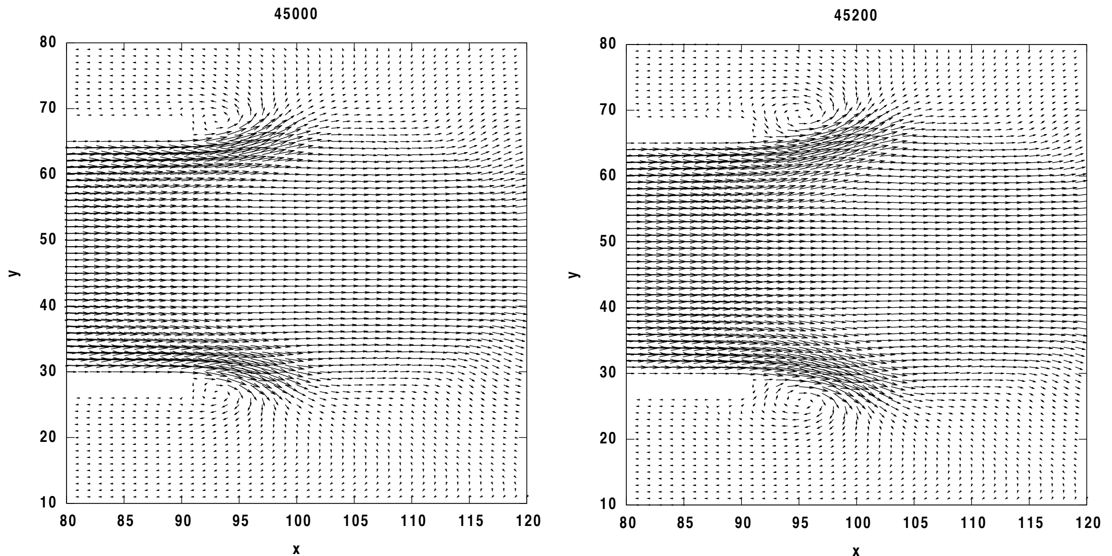


Figura 31: Campo de velocidades para el ducto cilíndrico para T=200, 400.

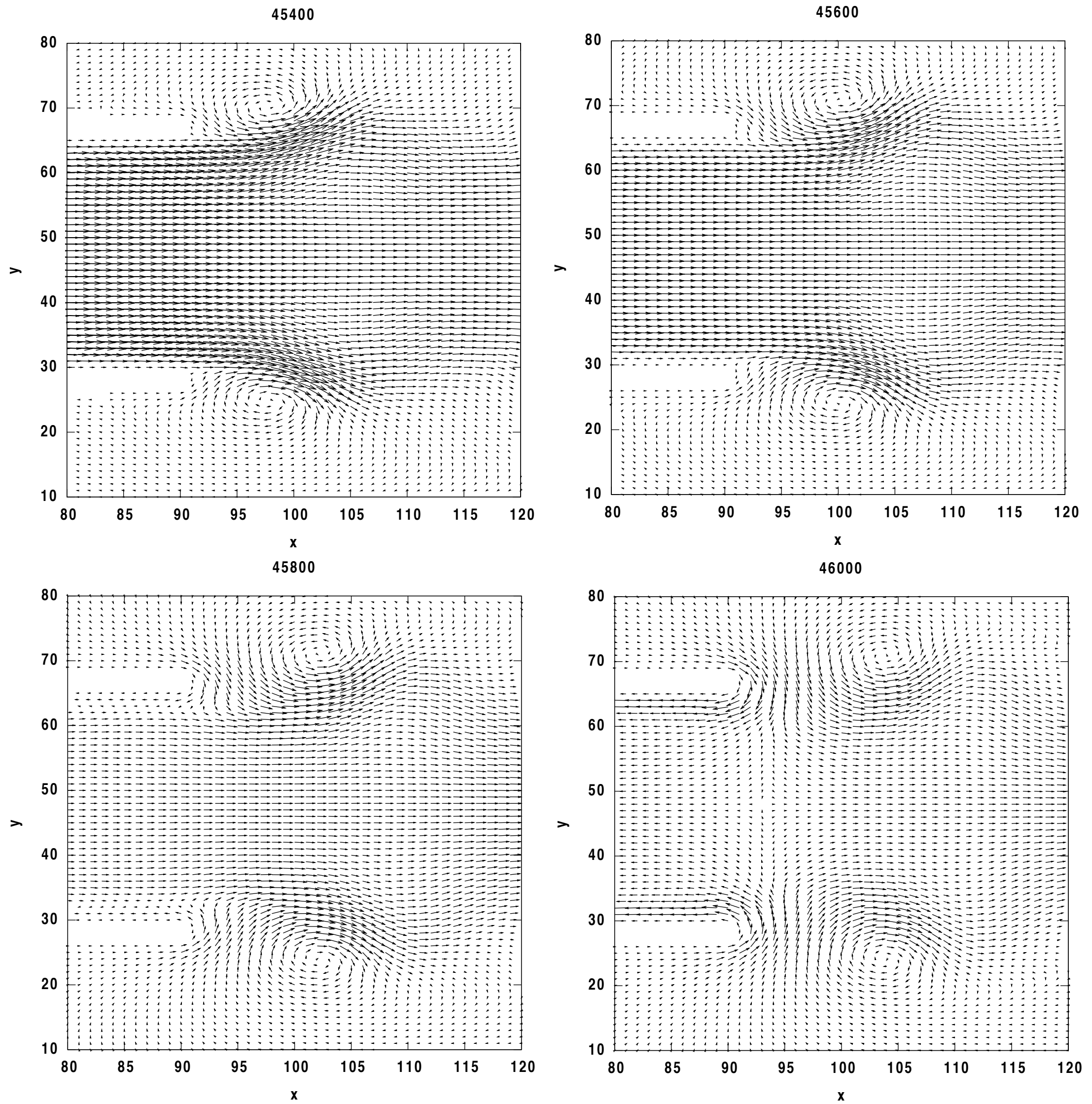


Figura 32: Campo de velocidades para el ducto cilíndrico para T=600, 800, 1000, y 1200.

En éstas gráficas podemos ver que existen vórtices que se desprenden del ducto en distintos tiempos de la oscilación. Estos vórtices han sido observados experimentalmte por el grupo de investigación del SIBEO y parecen ser los responsables por el cambio de eficiencia de la bomba. Dentro de ese proyecto de investigación se propuso una distinta geometría de la boca del ducto para eliminar los vórtices y aumentar la eficiencia del SIBEO. La geometría propuesta

es similar a la geometría de una trompeta y no es exactamente la presentada en esta tesis, pero la aproximación de semicírculos parece acercarse mucho. En las Figuras 33 y 34 se muestra el campo de velocidades en un plano céntrico para esta geometría en distintos tiempos del periodo del flujo pulsátil. Como un ejemplo de la visualización del flujo en la Figura 33 se muestra la magnitud del campo de velocidades en el ducto con sección transversal circular para un momento de la oscilación.

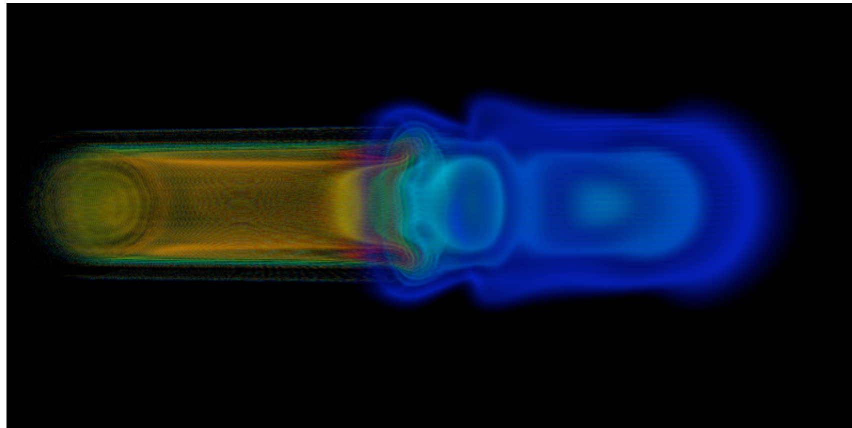


Figura 33: Magnitud del campo de velocidades dentro y fuera del ducto.

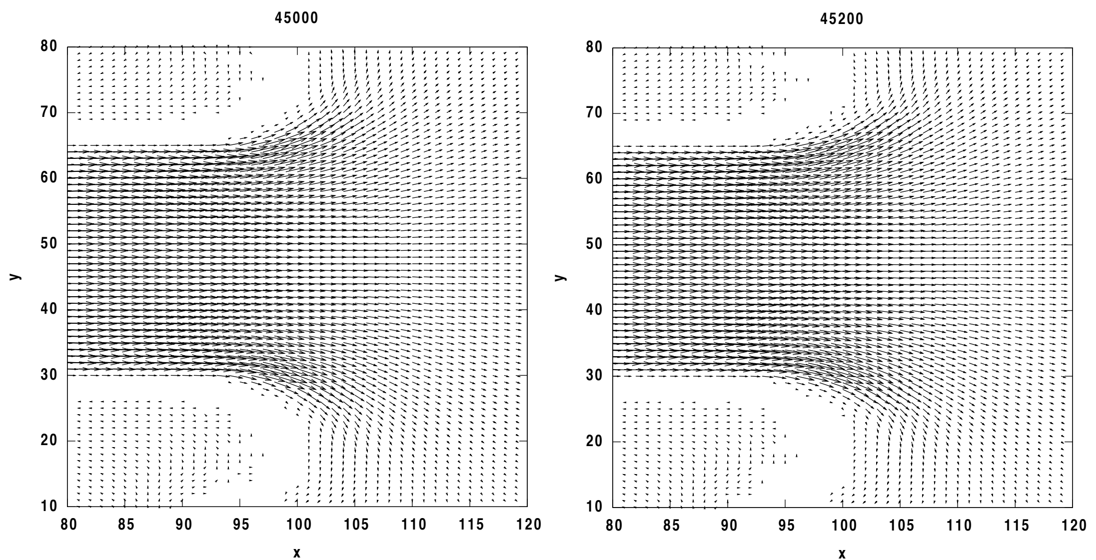


Figura 34: Campo de velocidades para el ducto del SIBEO para T=200 y 400.

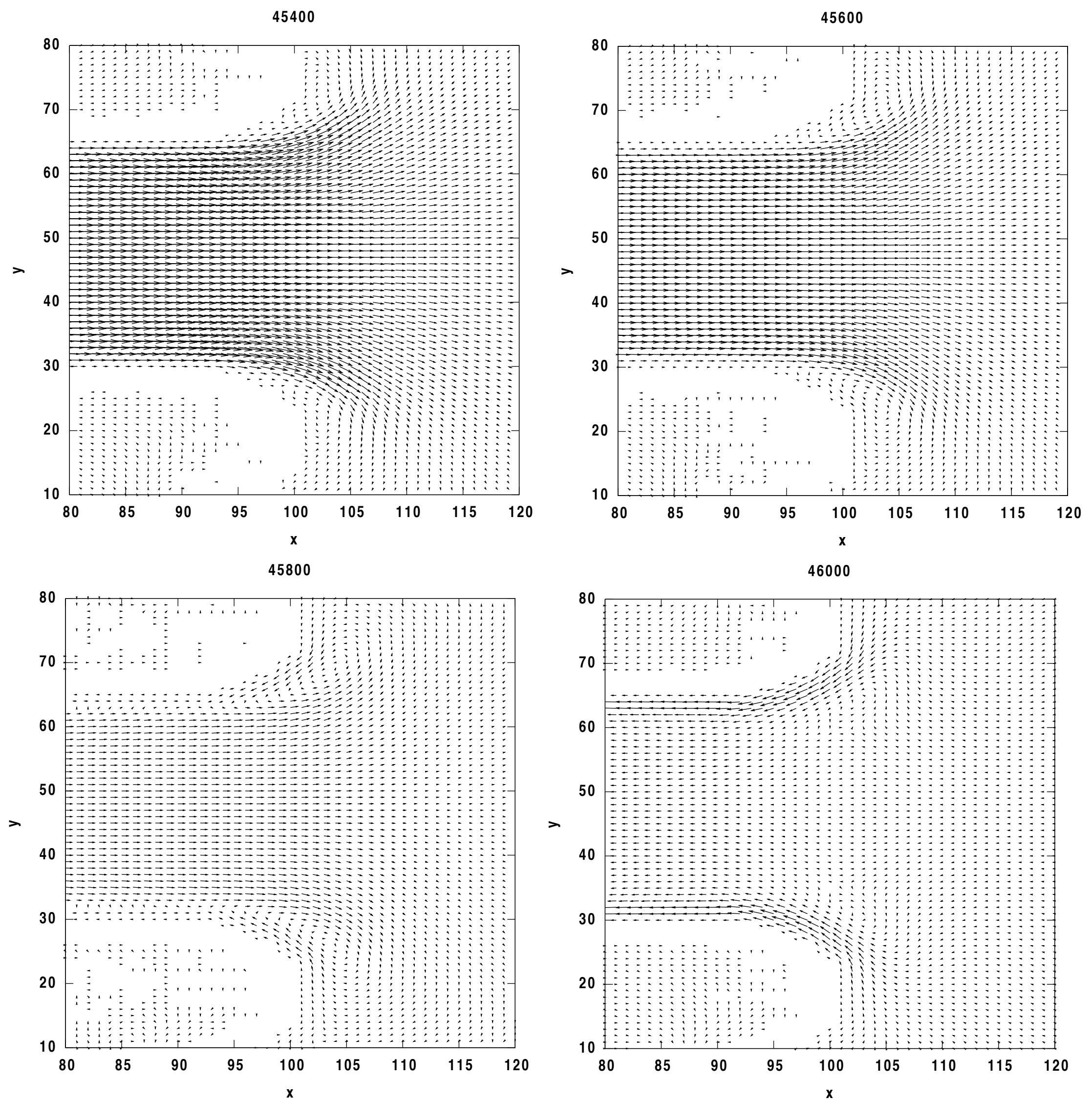
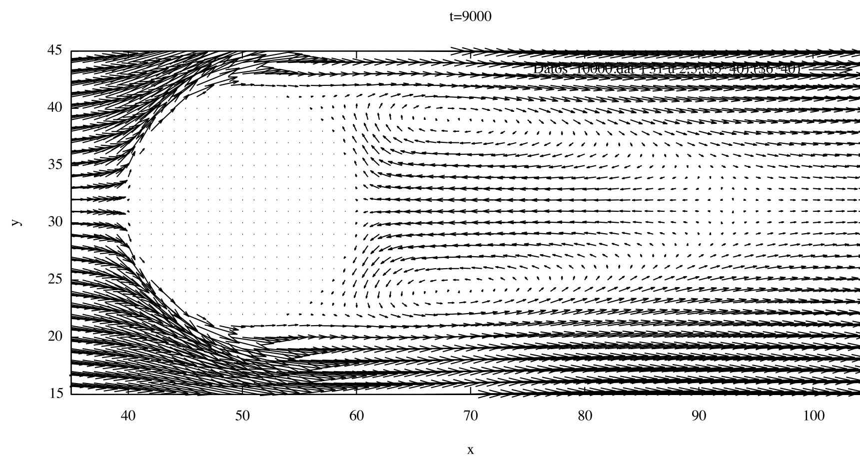


Figura 35: Campo de velocidades para el ducto del SIBEO para T= 600, 800. 1000 y 1200.

Esta simulación coincide cualitativamente con los flujos observados experimentalmente en donde los vórtices fuera de la trompeta desaparecen gracias a la geometría del ducto. Esta ausencia de vórtices aumenta la eficiencia de la bomba.

5.4. Flujo alrededor de una esfera

Como un último ejercicio, se simuló el flujo alrededor de una esfera para explorar las capacidades del método de lattice Bolzmann D3Q15. Para esta simulación se utilizó un sistema con dimensiones $N_x \times N_y \times N_z = 256 \times 64 \times 64$ o $1,048,576$ nodos. Se llevaron a cabo simulaciones para distintos números de Reynolds, para los cuales el tamaño del sistema se mantuvo constante, sin embargo se varió el valor del tiempo de relajación, asociado a la viscosidad, para alcanzar los distintos valores del numero de Reynolds. El número de Reynolds fue calculado con el radio de la esfera y la velocidad del flujo impuesta en la entrada del dominio. Para la simulación con número de Reynolds = 97 el tiempo de relajación fué de $\tau = 0.54$ mientras que para la simulación con número de Reynolds = 195 el tiempo de relajación fué de $\tau = 0.52$. Las condiciones de la frontera fueron nuevamente " halfway bounce back" en los nodos sólidos mientras que se impuso una velocidad inicial en $x = 0$ y condiciones de derivada cero en el plano $x = N_x - 1$ y condiciones periódicas en y y z. En las Figuras 35 y 36 se muestra el campo de velocidades al igual que imágenes de la simulación para distintos tiempos. Las simulaciones muestran una transición de un flujo estacionario a uno dependiente del tiempo alrededor de un número de Reynolds=130 que concuerda con las observaciones experimentales.



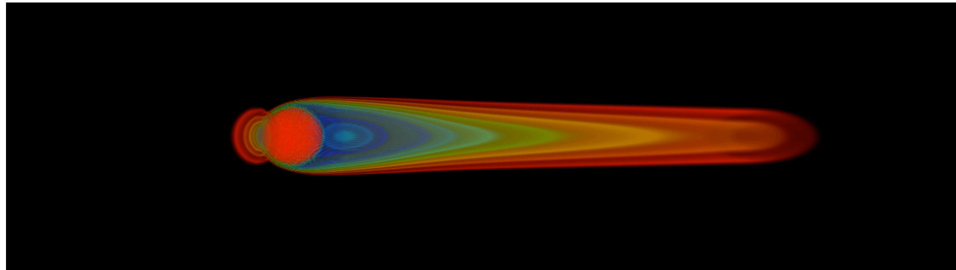
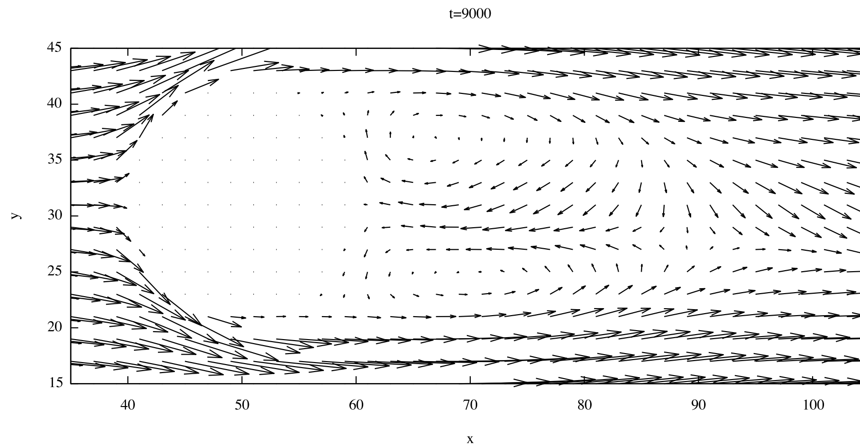


Figura 36: Flujo alrededor de una esfera. Reynolds=97. Arriba se muestra el campo de velocidades en el plano céntrico. Abajo se muestra una imágen de la magnitud de la velocidad en la simulación. El color rojo representa un valor de la magnitud de la velocidad de 0.86 veces la magnitud de la velocidad en la entrada y el color azul representa una velocidad de 0.026.



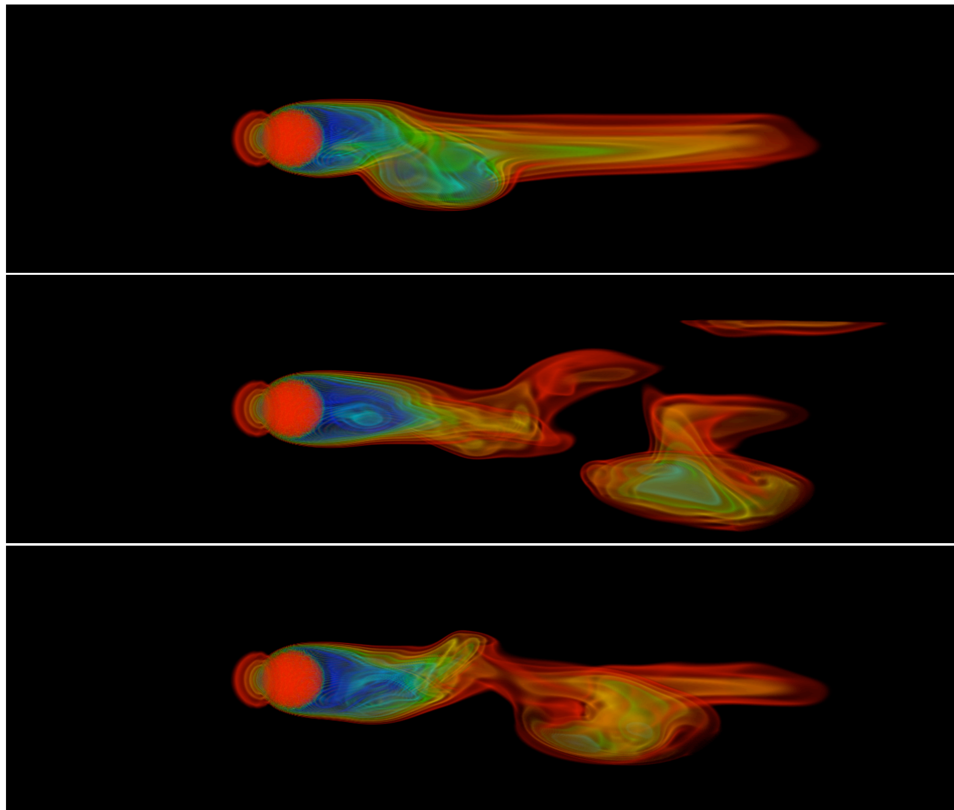


Figura 37: Flujo alrededor de una esfera. Reynolds=195. Arriba se muestra el campo de velocidades en el plano céntrico. Abajo se muestra una imágen de la magnitud de la velocidad en la simulación. El color rojo representa un valor de la magnitud de la velocidad de 0.86 veces la magnitud de la velocidad en la entrada y el color azul representa una velocidad de 0.026.

6. Conclusiones

El objetivo principal de esta tesis consistió en aprender a programar tarjetas de video (GPUs) para hacer cálculas numéricos en paralelo para implementar un método en el área de la dinámica de fluidos. El uso de GPUs para el cómputo científico es relativamente nuevo y representa hoy en dia la opción más económica para hacer cómputo de alto rendimiento. El trabajo presentado en esta tesis corresponde a uno de los primeros esfuerzos por manipular estas nuevas tecnologías dentro de nuestra comunidad. Este objetivo fue cumplido al simular el flujo tridimensional de un fluido haciendo uso del lenguaje de computo paralelo CUDA. Podemos concluir que es posible utilizar los GPUs para acelerar los cálclulos numéricos de programas del cómputo científico que pueden ser paralelizados; en particular, en la dinámica de fluidos y los fenómenos de transporte existen métodos con un alto nivel de paralelismo que son óptimos candidatos a ser paralelizados. El método de lattice Boltzmann en su esquema bidimensional D2Q9 y su esquema tridimensional D3Q15 fueron implementados en un GPU. El método bidimensional sirvió también para comparar la rapidez de la tarjeta de video respecto a la rapidez del CPU, encontrando mejorías de más de dos ordenes de magnitud en el tiempo de ejecución entre el programa paralelo y el programa serial. Los resultados que se obtuvieron al simular el flujo de Poiseuille y el flujo de Womersley utilizando estos métodos concuerdan con las soluciones analíticas.

Con los resultados presentados en esta tesis también es posible concluir que el método de lattice Boltzmann puede ser utilizado para predecir distintos fenómenos presentes en la dinámica de los fluidos. Mediante un tratamiento cuidadoso de las condiciones de frontera el método puede ser utilizado con distintos fines. Por ejemplo, parece posible simular el flujo sanguíneo utilizando éste método, y como una propuesta a futuro, se podría explorar a detalle las distintas condiciones de frontera para lograr este propósito. Este tipo de simulaciones son de mucha utilidad en el campo de la física médica y pueden ayudar a la comunidad de médicos a realizar tratamientos y diagnósticos cada vez más acertados. Conseguir que estas simulaciones se realizen en GPUs reducirá costos y tiempos de espera.

Las tarjetas de video proveen distintos tipos de memoria que pueden ser utilizados para mejorar el desempeño y la rapidez de los cálculos. Se encontró que para problemas con localidad en los datos, la memoria de texturas es la más eficiente, brindando una velocidad de cálculo dos veces más rápida que las memoria global o la memoria compartida. Esto nos muestra que el rendimiento de los GPUs puede ser optimizado para brindar mejores tiempos de ejecución dependiendo del método numérico que se quiera implementar. Para poder gozar de estos beneficios es necesario hacer un análisis del método deseado para determinar si contiene secciones que presenten paralelismo en los datos y una vez hecho esto es posible optimizar los algoritmos para que utilicen las distintas memorias del GPU en beneficio del método numérico.

Se puede concluir que el uso de la técnica de los GPUs para los cálclulos con propósitos científicos es viable. La utilización de estas tarjetas gráficas abre

las puertas del computo paralelo de alto rendimiento a centros de investigación con pocos recursos económicos. Las posibilidades que se abren al utilizar esta técnica para el cómputo científico son muy altas. La paralelización de distintos algoritmos en casi todas las áreas de la física pueden ayudar a traer el cómputo de alto rendimiento que antes estaba limitado a muy pocos centros de investigación, a cualquier computadora equipada con un GPU. El área de las simulaciones computacionales de alto rendimiento es ahora, mediante el uso de esta técnica, accesible prácticamente a todos.

7. Apéndices

7.1. Apéndice A. Instrucciónes de localización y copiado de datos

CUDA provee instrucciones específicas para lograr la localización de memoria en el GPU y las copias de datos entre el CPU y el GPU. La función $cudaMalloc()$ puede ser llamada desde el código del host para localizar un pedazo de la memoria global y hacer la copia de los datos. Es importante darse cuenta de la similitud entre $cudaMalloc()$ y la función estándar de C, $malloc()$. Esto es intencional; CUDA es C con un número de extensiones. CUDA utiliza la función $malloc()$ de las librerías estándar de C para localizar la memoria del host y añade $cudaMalloc()$ como extensión de C para localizar la memoria global del device.

Una vez que un programa ha localizado memoria global para los datos que se enviarán al dispositivo, entonces puede hacerse la transferencia del host hacia el device. Esto se logra al llamar otra de las funciones que son extensiones a C, $cudaMemcpy()$ para transferencias de memoria. La función de copiado de memoria puede ser utilizada también para copiar memoria desde una ubicación en la memoria del GPU a otra ubicación en la misma memoria. La misma función puede ser utilizada para transferir información en ambas direcciones haciendo uso de calificadores definidos en las librerías del lenguaje. También existen funciones como $cudaMemcpy3D()$ o $cudaMemcpyToArray()$ y distintios tipos de calificadores que permiten utilizar los distintos tipos de memoria del dispositivo para lograr procesos y algoritmos deseados por el programador. Al mantener la interfase cercana a la sintaxis típica de C , CUDA minimiza el tiempo que un programador de C necesita para aprender el uso de estas extensiones. A continuación se presenta una colección de las instrucciones principales que permiten gestionar la memoria del GPU:

- cudaMalloc (**void** ** devPtr, **size_t** size)

Esta función tiene dos parámetros. El primer parámetro es la dirección de un apuntador que apunta al objeto después de la localización en el GPU. La dirección de la variable apuntadora debe ser especificada como $(void**)$ ya que la función espera un valor de apuntador genérico; es decir, la función de localización de memoria es una función genérica que no esta restringida a ningún tipo particular de objeto. El segundo parámetro de la función es el tamaño del objeto que será localizado en términos de bytes. Este segundo parámetro debe ser consistente con el tamaño del objeto utilizado en la función de C $malloc()$. En la Figura 39 se muestra un extracto de código que ejemplifica como utilizar la función $cudaMalloc()$. Si suponemos que Md es una matriz cuadrada con dimensión Dim, se utliza $cudaMalloc()$ para localizar memoria en el GPU con las dimensiones deseadas.

```
28  float *Md;
29  int tamaño=Dim*Dim*sizeof(float);
30
31  cudaMalloc((void**)&Md, tamaño);
32  ...
33  cudaMemcpy(Md, M, tamaño, cudaMemcpyHostToDevice);
34  ...
35  cudaMemcpy(P, Pd, tamaño, cudaMemcpyDeviceToHost);
36  ...
37  cudaFree(Md);
```

Figura 38: Ejemplo del uso de la función $cudaMalloc()$ y $cudaMemcpy()$.

- cudaMallocPitch (**void** $**$ devPtr, **size_t** $*$ pitch, **size_t** width, **size_t** height)

Si se tiene un objeto de datos bidimensional, CUDA provee una instrucción que permite localizar memoria lineal ordenada de tal manera que al hacer una lectura o escritura sobre el apuntador en cualquier renglón, éste preserve la alineación requerida para tener una lectura o escritura coalescente. Esta función localiza width(en bytes)*height(en bytes) de memoria lineal y regresa en *devPtr un apuntador a dicha memoria. el parámetro de tipo size_t llamado pitch es el tamaño en bytes de cada renglón de la matriz. Este parámetro nos sirve para calcular direcciones dentro del arreglo 2D. Por ejemplo, dadas la columna y el renglón de un elemento, su dirección es calculada de la siguiente manera:
$elemento = (T*)((char*)apuntador + renglón * pitch) + columna$

Para localizaciones de arreglos 2D se recomienda el uso de esta función debido a las restricciones de alineación del hardware. Sobre todo cuando se efectúen copias de memorias en 2D entre distintas regiones de la memoria del dispositivo.

- cudaMalloc3D (**struct** cudaPitchedPtr $*$ pitchedPtr, **struct** cudaExtent extent)

Esta instrucción es útil cuando se trabaja con arreglos de datos tridimensionales. La instrucción tiene dos parámetros, ambos son estructuras. El primero se conoce como un "Pitched Pointer" y es un apuntador con una organización de memoria específica para arreglos en tres dimensiones. El segundo parámetro es una estructura que guarda el tamaño de la localización en bytes. Esta instrucción localiza $width * height * depth$ bytes de memoria lineal en el dispositivo y regresa el Pitched Pointer a la memoria localizada. Al igual que la instrucción para dos dimensiones ésta instrucción permite mantener la alineación cuando se hacen copias de memoria de arreglos tridimensionales. Un Pitched Pointer es una estructura con cuatro tipos de datos, *pitchedPtr.ptr* es el apuntador a la memoria, *pitchedPtr.pitch* es nuevamente el tamaño de un renglón de la localización en bytes. *pitchedPtr.xsize* y *pitchedPtr.ysze* son el tamaño lógico de un renglón y de una columna que son equivalentes a los parámetros especificados por el programador en la estructura *extent*. Para calcular una dirección (i, j, k) dentro del arreglo 3D debemos accesar a la memoria de la siguiente manera:

$char * devPtr = (char*)plot_ptr.ptr;$
$size_t\ pitch = plot_ptr.pitch;$
$size_t\ slicePitch = pitch * extent2.height;$
$char * slice = devPtr + k * slicePitch;$
$float * row = (float*)(slice + j * pitch)$

En donde $row\,[i]$ tiene el elemento (i, j, k).

- cudaMallocArray (**struct cudaArray** ** array, **const struct** cudaChannelFormatDesc *desc, **size_t** width, **size_t** height)

Esta instrucción se utiliza en la implementación de texturas. Nuevamente es un apuntador con un formato específico que cumple con restricciones del hardware para accesos coalescentes de memoria. Tiene 4 parámetros $**array$ es el apuntador a la memoria del dispositivo con el formato específico para el uso de texturas. El parámetro$*desc$ es una estructura que especifica el formato de los canales del arreglo texturizado. Los parámetros *width* y *height* especifican el tamaño del arreglo en bytes. El formato del segundo parámetro va de acuerdo a la estructura $cudaChannelFormatDesc$ que se define como:

$struct\ cudaChannelFormatDesc\{$
$int\ x,\ y,\ z,\ w;$
$enum\ cudaChannelFormatKind\ f;$
$\}$

en donde $cudaChannelFormatKind$ es uno de $cudaChannelFormatKindSigned$, $cudaChannelFormatKindUnsigned$, o $cudaChannelFormatKindFloat$. Estos descriptores le dicen a la textura que tipo de datos estará guardando.

- cudaMemcpy (**void** * dst, **const void** * src, **size_t** count, **enum cudaMemcpyKind** kind)

La función $cudaMemcpy()$ tiene cuatro parámetros. El primer parámetro es un apuntador al destino de la copia. El segundo parámetro apunta a los datos de origen los cuales serán copiados. El tercer parámetro especifica el número de bytes de la copia de datos. El cuarto parámetro indica los tipos de memoria involucrados en la copia: desde la memoria del host hacia la memoria del host, desde la memoria del host hacia la memoria del device, desde la memoria del device hacia la memoria del host, o desde la memoria del device hacia la memoria del device. Cabe mencionar que la función $cudaMemcpy()$ no puede ser utilizada para hacer copias entre distintos GPUs en sistemas con multiples dispositivos. La misma función puede ser utilizada para transferir información en ambas direcciones haciendo uso de calificadores definidos en las librerías del lenguaje $cuaMemcpyHostToDevice$ y $cudaMemcpyDeviceToHost$ y ordenando correctamente los apuntadores a las direcciones de origen y destino. Esto puede ser visto en el código de la Figura 39.

- cudaMemcpy2D (**void** $*$ dst, **size_t** dpitch, **const void** $*$ src, **size_t** spitch, **size_t** width, **size_t** height, **enum cudaMemcpyKind** kind)

Esta instrucción copia una matriz desde la memoria especificada por $*src$ hacia la memoria apuntada por $*dst$. La variable *kind* es una de las siguientes variables: $cudaMemcpyHostToHost$, $cudaMemcpyHostToDevice$, $cudaMemcpyDeviceToHost$, o $cudaMemcpyDeviceToDevice$, y especifica la dirección de la copia. *dpitch* y *spitch* son los tamaños de los renglones en términos de bytes de los arrelos bidimensionales apuntados por *dst* y *src*, incluyendo un "padding" al final de cada renglón que sirve para que la copia mantenga un formato preciso. Las áreas de memoria no deben de sobreponerse una con la otra, y la variable *width* no debe exceder el tamaño de las variables *dpitch* o *spitch* o la instrucción devolverá un error.

7.2. Apéndice B. Declaración de variables en los distintos tipos de memoria

En la Figura 40 se muestra la sintaxis para declarar las variables de un programa en los distintos tipos de memoria. Cada una de estas declaraciones también le da a la variable un rango "scope" y un tiempo de vida "lifetime".

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__, __shared__, int SharedVar;`	Shared	Block	Kernel
`__device__, int GlobalVar;`	Global	Grid	Application
`__device__, __constant__, int ConstVar;`	Constant	Grid	Application

Figura 39: Calificadores de variables (tomado de [16])

El rango define cuales de los hilos podrán accesar a la variable: un único hilo, todos los hilos de un bloque, o por todos los hilos de todas las mallas. Si el rango de una variable es únicamente un hilo, una versión privada de la variable será creada para cada hilo; cada hilo podrá accesar solamente a su versión privada de la variable. Por ejemplo, si una variable es declarada con un rango de un solo hilo y se lanza para un millón de hilos, entonces se crearan un millón de versiones de esa variable de tal modo que cada hilo inicializa y utiliza su propia versión de la variable.

El tiempo de vida representa la duración de la ejecución en la cual la variable esta disponible para ser utilizada: durante el llamado a un kernel o durante todo el tiempo de ejecución del programa. Si la vida de una variable está dentro del llamado a un kernel, debe de ser declarada en el código dentro del cuerpo del kernel y solamente podrá ser utilizada por el código del kernel. Si el kernel es llamado varias veces, el contenido de la variable no es guardado entre estos llamados. Cada llamado debe inicializar la variable para poder utilizarla. Por otro lado, si la vida de una variable es de todo el tiempo del programa, debe ser declarada fuera del cuerpo de cualquier función. Los contenidos de la variable son mantenidos durante toda la ejecución del programa y están disponibles a todos los kernels.

Si la declaración de una variable esta precedida por la palabra clave $__shared__$ (cada "__" consiste de dos "_" caracteres), declara una variable compartida. Estas declaraciones residen típicamente dentro del kernel o una función del dispositivo. El rango de una variable compartida es todo un bloque de hilos; esto es, todos los hilos dentro de un bloque ven la misma versión de una variable compartida. Una versión privada de la variable compartida es creada y utilizada por cada bloque durante la ejecución de un kernel. Cuando un kernel termina su

ejecución, los contenidos de la variable compartida dejan de existir. Las variables compartidas son una manera eficiente para que los hilos logren una cooperación entre ellos. El acceso a la memoria compartida es extremadamente rápido y altamente paralelo. Los programadores generalmente utilizan la memoria compartida para guardar una porción de la memoria global que es utilizada muchas veces dentro de una fase de ejecución de un kernel.

Si la declaración de una variable esta precedida por la palabra clave __*constant*__, declara una variable constante. La declaración de las variables constantes debe de estar fuera del cuerpo de cualquier función. El rango de estas variables es para todas las mallas, causando que todos los hilos de todas las mallas ven la misma versión de la variable. La vida de las variables constantes permanece durante toda la ejecución del programa. Estas variables se utilizan generalmente para variables que proveen valores de entrada a los kernels. Las variables constantes son guardadas en la memoria global pero tiene un "cache" para un acceso eficiente. Actualmente, el tamaño total de una variable constante esta limitado a 65,536 bytes.

Una variable que esta precedida por la palabra clave __*device*__, declara una variable global que será guardada en la memoria global. Los accesos a las variables globales son relativamente lentos; sin embargo, las variables globales son visibles a todos los hilos de todos los kernels. Sus contenidos también permanecen durante toda la ejecución de un programa. Así, las variables globales son una buena manera de cooperar entre bloques. Sin embargo, es importante tomar en cuenta que no existe ninguna manera de sincronizarse entre hilos de distintos bloques o de asegurar una consistencia en los datos entre hilos que accesan a la memoria global. Por lo tanto estas variables son utilizadas generalmente para pasar información de un llamado de un kernel al siguiente.

7.3. Apéndice C. Utilización de la memoria de texturas

Debido a que la memoria de textura es únicamente una manera de guardar los datos en un cache de manera optimizada para su lectura "local", para poder utilizarla debemos de acomodar nuestros datos de una manera muy especial. CUDA provee distintos tipos de variables que deben de ser utilizadas para acomodar nuestra información de manera útil para las texturas. Lo primero que debemos hacer es crear un *cudaArray* en la memoria del dispositivo. El *cudaArray* es un arreglo que tiene los datos ordenados en un formato especial diseñado para poder ser guardados en la memoria de texturas. Para poder crear este arreglo debemos de decirle a CUDA que tipo de datos estamos trabajando, ya que las texturas se diseñaron para trabajar con datos gráficos, tienen canales para guardar información en bits de RGBA, sin embargo también pueden guardar información en punto flotante. De este modo para especificar esta información debemos de crear un "channel format descriptor" o un descriptor del formato de canales. En el descrpitor determinaremos el tamaño de los canales RGBA y el tipo de datos que están guardados nuestros variables. Cuando trabajamos con datos científicos no es muy típico pensar en los canales RGBA así que CUDA permite dejar esos parámetros en blanco y solamente decidir el tipo de variables que guardara nuestro arreglo (signed/unsigned ints, o floats). Esto se muestra en la Figura 41

```
76    cudaChannelFormatDesc desc;
77    desc = cudaCreateChannelDesc<tipo de datos>(rBits, gBits, bBits, aBits);
78    desc = cudaCreateChannelDesc<float>();
```

Figura 40: declaración de un descriptor de cudaArrays

Habiendo especificado el tipo de datos con los se trabajará, es posible crear el cudaArray. Para hacer esto se hace uso de la función $cudaMallocArray()$ del CUDA API. En este punto se decide el tamaño de los datos. Dependiendo del tamaño de la memoria a localizar, se tiene un arreglo de datos ordenado (Figura 42).

```
80    cudaArray* cuArray;
81    cudaMallocArray(&cuArray, &channelDesc, width, height);
82    cudaMallocArray(&cuArray, &desc, ni, nj));
```

Figura 41: Localización de memoria de un cudaArray

Después se hace una copia de datos desde el apuntador hacia el cudaArray que se encuentra en el GPU. Para hacer esto es necesario utilizar otra de las funciones que provee el API, $cudaMemcpyToArray()$ que permite copiar la información de un apuntador en el CPU al arreglo ordenado en el GPU:

```
84      cudaMemcpyToArray(cuArray, 0, 0, cpuPtr, width*height*sizeof(T), cudaMemcpyHostToDevice);
85      cudaMemcpyToArray(cuArray, 0, 0, *datos, pitch, sizeof(float)*ni, nj,cudaMemcpyHostToDevice);
```

Figura 42: copia de datos

El último paso para poder accesar a la memoria de texturas en el código es el de crear una referencia de la textura, es decir un nombre para poder accesar a los datos de la textura. Al declarar una referencia de la textura especificamos la manera en la que se quiere leer los datos al fijar las dimensiones de la textura y el tipo de datos que hay en ella. También se tiene que especificar el tipo de lectura que se llevará a cabo. Existen variables predefinidas como “filtermode” y “normalized” que especifican la manera en que la lectura se comportará si se hace un llamado a los datos que se encuentran fuera del dominio, y si estos datos están normalizados o no. CUDA puede hacer una interpolación lineal o fijar el valor del llamado fuera del dominio al valor de la frontera. Esto se hace de la siguiente manera:

```
92      texture<float, 2> tex;
93      tex.filterMode = cudaFilterModePoint;
94      tex.normalized = false;
```

Figura 43: referencia de la textura

Una vez hecho esto debemos unir la referencia de la textura al *cudaArray* para que el compilador sepa que datos son los que debe de cachear en el dispositivo para poder accesar a los datos dentro de la memoria. Esto se hace utilizando de la función *cudaBindTextureToArray*() que esta diseñada para llevar a cabo esta acción:

```
99      cudaBindTextureToArray(tex, cuArray, desc);
100     cudaBindTextureToArray(tex, cuArray);
```

Figura 44: binding de cudaArray a referencia de la textura

Por último se puede accesar a la memoria de texturas desde cualquier parte del kernel. El acceso a la memoria se hace de la siguiente manera:

```
102     float variable = tex2D(tex, x, y);
```

Figura 45: lectura de la memoria de texturas

en donde x y y son las coordenadas cartesianas de la malla. Así mediante un simple desfase de las coordenadas podemos accesar a datos que se encuentran juntos físicamente en el método.

7.4. Apéndice D. Programas que resuelven la ecuacion de calor con memoria compartida y memoria de texturas

A continuación se muestra el encabezado del programa que resuelve la ecuación de calor en un dominio bidimensional utilizando la memoria global o la memoria compartida de CUDA para guardar nuestras variables.

```
12  // Librerias
13  #include <stdio.h>
14  #include <stdlib.h>
15  #include <cutil.h>
16
17  // Tamaño del bloque
18  #define TILE_I 16
19  #define TILE_J 16
20
21  // Punteros en el CPU
22  float *t0;
23
24  // Punteros, arreglos y texturas en el GPU
25  float *t_data;
26  float *t_data_old;
27
28  // Escalares globales:
29  float dx,dy,dp,dt,kcond;
30  int ni,nj,paso;
31  size_t pitch;
32
33  // Definicion del CUDA kernel:
34  __global__ void ec_calor_kernel (int ni, int nj, int pitch, float kcond,
35          float dt, float dx, float dy, float *t_data, float *t_data_old);
36
37  // Definición de subrutinas de C:
38  void ec_calor(void);
39  void Imprimir(void);
```

Figura 46: Encabezado de programa con uso de memoria global o com.partida

En el encabezado se encuentran las librerías que incluirá el programa; la librería $< stdio.h >$ de C permite hacer uso de las funciones de entrada y salida como $printf()$, etc. también se utiliza la librería $< stdlib.h >$ de C que contiene funciones de control de procesos y manejo de memoria como $malloc()$, etc. Por último se incluye la librería $< cutil.h >$ que contiene las funciones de CUDA. La definición de las variables globales $TILE_I$ y $TILE_J$ son utilizadas mas adelante para definir el tamaño de los bloques de hilos en el llamado a los kernels. Después se tienen las definiciones de los apuntadores para las variables globales en el CPU y en el GPU. Por último se encuentran ciertas variables que son utilizadas en la simulación y los prototipos de las funciones que serán utilizadas por el programa, en donde se encuentra también el prototipo de la función kernel. Como podemos notar, la única diferencia de este encabezado con el encabezado de cualquier programa escrito en C es la definición de nuestro prototipo de la función kernel, que esta precedida por la palabra clave $__global__$ que indica

que será una función que hará uso del GPU. Los apuntadores se definen de la misma manera para el CPU y para el GPU, la diferencia será el tipo de memoria que se utilizará mas adelante en el código para localizarlos.

Ahora mostraremos el código que se utiliza para resolver la misma ecuación haciendo uso de la memoria compartida proporcionada por CUDA. El encabezado del programa, el cuerpo de la función principal *main*(), y el cuerpo de la función *solveheat*() son exactamente iguales a los del programa con memoria global. Estos son representados por las Figuras 47 y 12 respectivamente. El kernel sin embargo es distinto; la matriz del campo de temperaturas esta definida como una variable compartida. Podemos ver que la palabra clave *__shared__* precede la definición de la matriz:

```
113  __global__ void ec_calor_kernel (int ni,int nj,int pitch, float kcond, float dt,
114                                  float dx, float dy, float *t_data, float *t_data_old)
115  {
116      int i, j, ii, jj, i2d, i2d2, i2d3, i2d4, i2d5;
117      float told,tnow;
118
119      i = blockIdx.x*TILE_I + threadIdx.x;
120      j = blockIdx.y*TILE_J + threadIdx.y;
121
122      i2d = i + j*pitch/sizeof(float);
123      i2d2= (i+1) + (j)*pitch/sizeof(float);
124      i2d3= (i-1) + (j)*pitch/sizeof(float);
125      i2d4= (i) + (j+1)*pitch/sizeof(float);
126      i2d5= (i) + (j-1)*pitch/sizeof(float);
127
128      told = t_data_old[i2d];
129
130      if (i ==ni-1) i2d2= ni-1 + (j)*pitch/sizeof(float);
131      if (i == 0) i2d3= 0 + (j)*pitch/sizeof(float);
132      if (j ==nj-1) i2d4= i + (nj-1)*pitch/sizeof(float);
133      if (j == 0) i2d5= i + (0)*pitch/sizeof(float);
134
135      /// Declaracion de la matriz en memoria compartida
136      __shared__ float t[TILE_I + 2][TILE_J + 2];
137
138      /// Indices de la matriz en memoria compartida
139      ii = threadIdx.x + 1;
140      jj = threadIdx.y + 1;
141
142      /// Lectura colectiva de memoria global y escritura en compartida
143      t[ii][jj] = told;
144
145      if (ii == 1)      t[ii-1][jj] = t_data_old[i2d3];
146      if (ii == TILE_I)   t[ii+1][jj] = t_data_old[i2d2];
147      if (jj == 1)      t[ii][jj-1] = t_data_old[i2d5];
148      if (jj == TILE_J)   t[ii][jj+1] = t_data_old[i2d4];
149
150      __syncthreads();
151
152      tnow = told + dt*kcond*( ( t[ii+1][jj] - 2.0f*told + t[ii-1][jj] )/(dx*dx)
153                           + ( t[ii][jj+1] - 2.0f*told + t[ii][jj-1])/(dy*dy));
154
155      t_data[i2d] = tnow;
156  }
```

Figura 47: Kernel para resolver la ecuación de calor con memoria compartida

Por último se muestra el código que hace uso de la memoria de texturas para resolver la ecuación de calor. Podemos ver que el acceso a los elementos del arreglo es mucho mas sencillo e intuitivo en esta manera de lectura de memoria. Primero, el encabezado del programa cambia ligeramente,

```
11  // Librerias
12  #include <stdio.h>
13  #include <stdlib.h>
14  #include <cutil.h>
15
16  // Tamaño del bloque
17  #define TILE_I 16
18  #define TILE_J 16
19
20  // Punteros en el CPU
21  float *t0;
22
23  // Punteros, arreglos y texturas en el GPU
24  float *t_data;
25  cudaArray *t_array;
26  texture<float, 2> t_tex;
27
28  // Escalares globales
29  float dx,dy,dp,dt,kcond;
30  int   ni,nj,paso;
31  size_t pitch;
32
33  // Definicion del CUDA kernel
34  __global__ void ec_calor_kernel (int ni, int nj, int pitch, float kcond,
35                                  float dt, float dx, float dy, float *t_data);
36  // Definicion de subrutinas de C
37  void ec_calor(void);
38  void Imprimir(void);
```

Figura 48: Encabezado de programa con uso de memoria de texturas

La diferencia entre este encabezado y el encabezado de los programas anteriores es la definición de un *cudaArray* y la declaración de la referencia a la textura en la línea 26. Podemos ver que se define una textura de escalares de punto flotante de dos dimensiones. A continuación se presenta la función main:

```
int main(void)
{
    int i;
    int totpoints;
    cudaChannelFormatDesc desc;

    dt = 0.01f;
    dx = 0.1f;
    dy = 0.1f;
    kcond = 0.01f;
    ni=1120;
    nj=1120;
    totpoints = ni*nj;

    printf ("ni = %d\n", ni);
    printf ("nj = %d\n", nj);
    printf ("Numero de puntos = %d\n", totpoints);

    // Aloca la memoria en el CPU (host)
    t0 = (float *)malloc(ni*nj*sizeof(float));

    // Aloca la memoria en el GPU (device)
    CUDA_SAFE_CALL(cudaMallocPitch((void **)&t_data, &pitch,
                                sizeof(float)*ni, nj));

    desc = cudaCreateChannelDesc<float>();
    CUDA_SAFE_CALL(cudaMallocArray(&t_array, &desc, ni, nj));

    // Valores iniciales del campo t
    for (i=0; i<totpoints; i++) {
    t0[i] = 0.f;
    }
    t0[totpoints/2 + ni/2] = 1000.f;

    // Copia valores iniciales al GPU
    CUDA_SAFE_CALL(cudaMemcpy2D((void *)t_data, pitch, (void *)t0,
                            sizeof(float)*ni,sizeof(float)*ni, nj,
                            cudaMemcpyHostToDevice));

    paso = 0;

    for (i=1;i<=10000;i++){
    paso = paso + 1;
    ec_calor();
    if (paso%1000 == 0) printf ("Iteracion: %d\n", paso);
    }

    Imprimir();

    return 0;
}
```

Figura 49: Función main() para el programa con uso de memoria de texturas

La diferencia entre el cuerpo de esta función y la de los otros programas es la existencia del descriptor del formato de canales *cudaChanneFormatDesc* en donde se especifica que los datos son escalares de punto flotante. La otra diferencia es que ahora se tiene un *cudaArray* y un arreglo normal localizados en la memoria del GPU. En este caso además de las actualización de datos en el GPU, también se encuentra la definición de nuestro *filtermode* y la unión del arreglo con la referencia a la textura con *cudaBindTextureToArray*(). Más allá de eso el algoritmo sigue los mismos pasos que los otros programas.

```
void ec_calor(void)
{
    CUDA_SAFE_CALL(cudaMemcpy2DToArray(t_array, 0, 0, (void *)t_data, pitch,
                            sizeof(float)*ni, nj, cudaMemcpyDeviceToDevice));

   t_tex.filterMode = cudaFilterModePoint;
    CUDA_SAFE_CALL(cudaBindTextureToArray(t_tex, t_array));

    dim3 grid = dim3(ni/TILE_I, nj/TILE_J);
    dim3 block = dim3(TILE_I, TILE_J);

    ec_calor_kernel<<<grid, block>>>(ni,nj,pitch,kcond,dt,dx,dy,t_data);

    CUDA_SAFE_CALL(cudaUnbindTexture(t_tex));
    CUT_CHECK_ERROR("ec_calor falló.");
}
```

Figura 50: subrutina para la ecuacción de calor con memoria de texturas

Por último se presenta el kernel para resolver la ecuación de calor con memoria de texturas. Debido a la optimización de lectura para arreglos de datos bidimensionales, la lectura de la temperatura de los nodos vecinos es completamente directa a través de dos índices. Primero se definen estos índices en términos de las variables $threadIdx.x$, $blockIdx.x$, etc. para luego poder hacer los llamados a la memoria de texturas vía la función $tex2D()$.

```
__global__ void ec_calor_kernel (int ni,int nj,int pitch, float kcond, float dt,
                                              float dx, float dy, float *t_data)
{
    int i, j, i2d;
    float told,tnow,tip1,tim1,tjp1,tjm1;

    i = blockIdx.x*TILE_I + threadIdx.x;
    j = blockIdx.y*TILE_J + threadIdx.y;

    i2d = i + j*pitch/sizeof(float);

    told = tex2D(t_tex, (float) i,   (float) j);
    tip1 = tex2D(t_tex, (float) i+1, (float) j);
    tim1 = tex2D(t_tex, (float) i-1, (float) j);
    tjp1 = tex2D(t_tex, (float) i,   (float) j+1);
    tjm1 = tex2D(t_tex, (float) i,   (float) j-1);

    if (i ==ni-1)  tip1 = tex2D(t_tex, (float) i, (float) j);
    if (i == 0)    tim1 = tex2D(t_tex, (float) i, (float) j);
    if (j ==nj-1)  tjp1 = tex2D(t_tex, (float) i, (float) j);
    if (j == 0)    tjm1 = tex2D(t_tex, (float) i,   (float) j);

    tnow = told + dt*kcond*((tip1-2.0f*told+tim1)/(dx*dx)
                      + (tjp1-2.0f*told+tjm1)/(dy*dy));
    t_data[i2d] = tnow;
}
```

Figura 51: Kernel para resolver la ecuación de calor con memoria compartida

7.5. Apéndice E. Kernel Para resolver la ecuación de lattice Boltzmann con esquema D3Q15

En este apéndice se proporciona el código utilizado para resolver la ecuación de lattice Boltzmann con esquema D3Q15.

```
250  __global__ void LB_step_kernel (cudaExtent extent2, float *f_data,
251  float *f_old_data, int * solid_data, int *cx_data, int *cy_data,
252  int *cz_data, float *face_data, float tau, int N, float roout)
253  {
254      int i, j, k, m, ii, jj, kk, i3d, i3d2, idx, idx2, idx3;
255      float rtau, rtau1, v_sq_term, ro, vx, vy, vz;
256      float fnow[15], feq[15];
257      int op[15];
258
259      rtau = 1.f/tau;
260      rtau1 = 1.f - rtau;
261
262      //vector de opuestos
263      op[0]  =  0;
264      op[1]  =  3;
265      op[2]  =  4;
266      op[3]  =  1;
267      op[4]  =  2;
268      op[5]  =  6;
269      op[6]  =  5;
270      op[7]  = 13;
271      op[8]  = 14;
272      op[9]  = 11;
273      op[10] = 12;
274      op[11] =  9;
275      op[12] = 10;
276      op[13] =  7;
277      op[14] =  8;
278
279
280      i = blockIdx.x*blockDim.x + threadIdx.x;
281      j = (blockIdx.y*blockDim.y)%extent2.height+threadIdx.y;
282      k = floorf(blockIdx.y/(extent2.height/blockDim.y))*blockDim.z + threadIdx.z;
283      i3d = i + j*(extent2.width) + k*(extent2.width*extent2.height);
284
285      //si soy fluido:
286      if (solid_data[i3d] == 1){
287
288          // lectura de todas las f's y guardamos en registros
289          //#pragma unroll
290          for (m=0; m<N; m++){
291          i3d2 = i3d + (extent2.width*extent2.height*extent2.depth)*m;
292          fnow[m] = f_old_data[i3d2];
293          }
```

Figura 52: Kernel para resolver la ecuación de lattice Boltzmann con esquema D3Q15

```
294
295            // calculamos macros:
296            ro =  fnow[0] + fnow[1]  + fnow[2]  + fnow[3]  + fnow[4]  + fnow[5]  +
...            fnow[6] +  fnow[7] +  fnow[8] + fnow[9]  + fnow[10] + fnow[11] +
...            fnow[12] + fnow[13] + fnow[14];
297
298            vx = (fnow[1]  - fnow[3]  + fnow[7]   + fnow[8] - fnow[9] - fnow[10] +
...            fnow[11] +  fnow[12] - fnow[13] - fnow[14]) / roout;
299
300            vy = (fnow[2]  - fnow[4]  + fnow[7]   + fnow[8] + fnow[9] + fnow[10] -
...            fnow[11] -  fnow[12] - fnow[13] - fnow[14]) / roout;
301
302            vz = (fnow[6]  - fnow[5]  + fnow[7]   - fnow[8] - fnow[9] + fnow[10] +
...            fnow[11] -  fnow[12] - fnow[13] + fnow[14]) / roout;
303
304            v_sq_term = ((vx*vx) + (vy*vy) + (vz*vz));
305
306            for (m=0; m<N; m++){
307
308                ii = i + cx_data[m];
309                jj = (j + cy_data[m])%extent2.height;
310                kk = (k + cz_data[m])%extent2.depth;
311                //encuentra a los vecinos:
312                idx = ii + extent2.width*jj + extent2.width*extent2.height*kk;
313                    //si no eres frontera:
314                    if ( ii >= 0 && ii<extent2.width ){
315                    // calculamos f's de equilibrio
316                    feq[m] = face_data[m] * (ro + roout * (3.f*((cx_data[m]*vx) +
...                    (cy_data[m]*vy) + (cz_data[m]*vz)) + 4.5f*((cx_data[m]*vx) +
...                    (cy_data[m]*vy) + (cz_data[m]*vz))*((cx_data[m]*vx) +
...                    (cy_data[m]*vy) + (cz_data[m]*vz)) - (1.5f*v_sq_term)));
317                    // si mi vecino es fluido
318                    if (solid_data[idx] == 1 ) {
319                        idx2 = idx + (extent2.width*extent2.height*extent2.depth)*m;
320                        // calculamos los nuevos valores de las f's y los guardamos
...                        en el vecino
321                        f_data[idx2]= rtau1 * fnow[m] + rtau * feq[m];
322                    }
323                    // si mi vecino es solido
324                    else {
325                        idx3 = i3d
...                        +(extent2.width*extent2.height*extent2.depth)*op[m];
326                        //calculamos los nuevos valores de las f's y los guardamos
...                        en direccion opuesta local (halfway bounceback)

321                        f_data[idx2]= rtau1 * fnow[m] + rtau * feq[m];
322                    }
323                    // si mi vecino es solido
324                    else {
325                        idx3 = i3d
...                        +(extent2.width*extent2.height*extent2.depth)*op[m];
326                        //calculamos los nuevos valores de las f's y los guardamos
...                        en direccion opuesta local (halfway bounceback)
327                        f_data[idx3] = rtau1 * fnow[m] + rtau * feq[m];
328                    }
329                }
330            }
331        }
332
333    }
```

Figura 53: Kernel para resolver la ecuación de lattice Boltzmann con esquema D3Q15

Ludwig Boltzmann, 1844-1906, que con su teorema H abrió la puerta hacia el entendimiento del mundo macroscópico en base a la dinámica molecular.

Referencias

[1] A M Artoli, A G Hoekstra, and P M A Sloot. 3D Pulsatile flow with the lattice Boltzmann BGK method. *International Journal Of Modern Physics*, 13(8):1119–1134, 2002.

[2] G.K. Batchelor. *An introduction to fluid dynamics*. Cambridge Univ Pr, 2000.

[3] C. Boyd. Data-parallel computing. *Queue*, (April 2008):30–39, 2008.

[4] S. Chen and Gary D. Doolen. Lattice Boltzmann Method for Fluid Flows. *Annual Review of Fluid Mechanics*, 30(1):329–364, January 1998.

[5] S. Chen, D. Martinez, and R. Mei. On boundary conditions in lattice Boltzmann methods. *Physics of Fluids*, 8(9):2527, 1996.

[6] S. Chen, Z. Wang, X. Shan, and Gary D. Doolen. Lattice Boltzmann computational fluid dynamics in three dimensions. *Journal of Statistical Physics*, 68(3):379–400, 1992.

[7] T.J. Chung. *Computational Fluid Dynamics*, volume 206. Cambridge University Press, 2002.

[8] S. Czitrom. Sea-water pumping by resonance I. In *Proceedings of the Second European Wave Power Conference, Lisbon, November*, pages 324–328, 1995.

[9] P.J. Davis and P. Rabinowitz. *Methods of numerical integration*, volume 1. Academic press New York, 1975.

[10] U. Frisch and B. Hasslacher. Lattice-gas automata for the Navier-Stokes equation. *Physical Review Letters*, 56(14), 1986.

[11] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel Computing Experiences With CUDA. *Micro, IEEE*, 28(4):13–27, 2008.

[12] X. He and L. Luo. A priori derivation of the lattice Boltzmann equation. *Physical Review E*, 55(6):6333–6336, 1997.

[13] X. He and L. Luo. Lattice Boltzmann Model for the Incompressible Navier-Stokes Equation. *Journal of Statistical Physics*, 88(3/4):927–944, August 1997.

[14] X. He and L. Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review E*, 56(6):6811–6817, December 1997.

[15] K. Huang. *Statistical Mechanics*, volume 8.

[16] D.B. Kirk and W.H. Wen-mei. *Programming massively parallel processors: A Hands-on approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2010.

[17] R.L. Liboff. *Kinetic theory: classical, quantum, and relativistic descriptions*, volume 59. Springer Verlag, 2003.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla : a unified and graphics computing arquitechture. *Micro, IEEE*, 28(2):39–55, 2008.

[19] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[20] J. Sanders and E. Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming, 2010.

[21] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54, September 2005.

[22] Y. Wang, X. He, G. Tang, and W. Tao. Simulation of two-dimensional oscillating flow using the lattice Boltzmann method. *International Journal of Modern Physics C*, 17(5):615–630, 2006.

[23] M. Zamir. *The Physics of Pulsatile Flow*. Springer, 2000.

[24] Q. Zou and X. He. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Physics of Fluids*, 9(6):1591, 1997.

[25] R Zwanzig. *Nonequilibrium statistical mechanics*. Oxford University Press, USA, 2001.