Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 7095-7108 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

USO DE GPUS PARA LA SIMULACIÓN DE FLUIDOS EN 3D CON EL MÉTODO DE LATTICE BOLTZMANN

P. R. Rinaldi[a,b], E. A. Dari[c], M. J. Vénere[b,c] y A. Clausse[a,b,c]

[a]CNEA-CONICET y Universidad Nacional del Centro, 7000 Tandil, Argentina.
[b]CNEA e Instituto Balseiro, 8400 Bariloche, Argentina.

Palabras Clave: GPGPU, Lattice Boltzmann Methods, CUDA, 3D, Navier-Stokes.

Resumen. Se implementó en una Unidad de Procesamiento Gráfico (GPU) un modelo de Lattice Boltzmann (LBM) para resolución de las ecuaciones de Navier-Stokes en 3 dimensiones, utilizando el lenguaje de GPU Computed Unified Device Architecture (CUDA) de NVIDIA. Se realizaron simulaciones de flujo en una cavidad cúbica con velocidad constante en la capa superior con diferentes tamaños de grillas y números de Reynolds comparando los resultados con los encontrados en la literatura mostrando gran coherencia. Al tratarse de un problema acotado por el ancho de banda de memoria, se optimizó el paso de propagación utilizando memoria compartida y accesos alineados a memoria global alcanzando el 35% del ancho de banda nominal de la placa. El *speedup* logrado con respecto a una implementación similar optimizada para CPU es de dos órdenes de magnitud.

1 INTRODUCCIÓN

El método de Lattice Boltzmann (LBM) es un modelo de Fluidodinámica Computacional (CFD) que resuelve eficientemente fenómenos de transporte, y en particular aproxima muy bien la solución de las ecuaciones de Navier-Stokes a bajo números de Reynolds (Chen y Doolen 1998, Higuera y Succi 1989, Karlin *et al* 1999; Shan y He 1998, Ansumali *et al* 2003). Básicamente, LBM es un autómata celular de estados reales (no binarios) que ejecuta un esquema explícito de colisión y advección. Una característica fundamental de LBM es la flexibilidad para realizar implementaciones en paralelo ya que al ser Autómatas Celulares (CA) con esquemas explícitos, un mismo código se ejecuta sobre todas las celdas del dominio para pasar de un estado del tiempo a otro. Si bien el campo de aplicaciones de LBM ha crecido considerablemente, existen conceptos pendientes como la estabilidad, el refinamiento de la grilla y sobre todo la performance, que dificultan la amplia aceptación de LBM para aplicaciones CFD.

El inconveniente principal de LBM es el gran volumen de información intercambiado en cada paso de propagación, lo cual produce un cuello de botella a la hora de paralelizar utilizando tecnologías clásicas basadas en pasaje de mensajes. Un esquema más apropiado para este tipo de modelos es el de memoria compartida, disponible hoy en día en placas gráficas (GPU) y que puede ser utilizado para computación de propósito general a través de diferentes lenguajes. Las GPUs modernas están optimizadas para ejecutar una instrucción simple sobre cada elemento de un extenso conjunto en paralelo utilizando múltiples procesadores de shaders que acceden a una misma memoria global compartida. Algunas de las últimas publicaciones en el área sugieren que la ejecución de LBM en GPU es prometedora para la simulación de fluidos en forma rápida y eficientes (Goodnight 2007, Tölke 2007, Zhao 2007, Kuznik *et al* 2010).

En este trabajo se presenta la implementación en GPU de un esquema LBM para resolver Navier-Stokes en 3 dimensiones. Se mostrarán las estrategias de manejo de memoria utilizadas y los resultados del análisis de eficiencia realizado en un caso de flujo en una cavidad cúbica.

2 NVIDIA CUDA

2.1 Modelo de Programación CUDA

La tecnología CUDA, fue desarrollada especialmente por NVIDIA para resolver problemas computacionales complejos utilizando las capacidades de procesamiento de las GPU a través de una interfaz de programación. El lenguaje está basado en C estándar, lo cual simplifica enormemente el desarrollo de código. La GPU, denominada *device*, se ve como una unidad computacional capaz de ejecutar múltiples *threads* paralelos trabajando como un co-procesador para la CPU principal, la cual se denomina *host*. Las aplicaciones con alto costo computacional se descargan de la CPU a la GPU utilizando llamadas a funciones CUDA. Tanto CPU como GPU mantienen su propia memoria RAM y los datos se copian de una a otra utilizando métodos DMA optimizados provistos por el lenguaje.

2.2 GPUS NVIDIA GeForce

La placa gráfica utilizada en este trabajo es una Nvidia GTX 260 de la serie

GTX200. Esta placa pertenece a la tercera generación de GPUs desarrolladas en conjunto con el modelo de programación CUDA. Está compuesta por un conjunto de multiprocesadores con arquitectura SIMT (instrucción simple para múltiples *threads*), que es una evolución de las arquitecturas SIMD (instrucción simple para múltiples datos). En este modelo de ejecución, cada procesador dentro de un multiprocesador ejecuta la misma instrucción en cada ciclo de reloj simultáneamente sobre datos diferentes como un *thread* independiente. La tabla 1 muestra las principales características de la placa utilizada.

NVIDIA Geforce GTX 260	
Cantidad de Multiprocesadores	24
Cantidad de Procesadores	192
Tamaño de Memoria Global	896 MB
Tamaño del Bus de Memoria	448 bits
Ancho de banda de Memoria	111,9 GB/s
Performance Pico estimada	805 Gflops
Reloj del Núcleo Principal	576 Mhz
Reloj de los Procesadores	1242 Mhz
Reloj de Memoria	999 Mhz
Capacidad Computacional CUDA	1.3

Tabla 1. Especificaciones del hardware de GPU (NVIDIA, 2010a).

3 MÉTODO DE LATTICE BOLTZMANN

A diferencia de los métodos clásicos que resuelven ecuaciones diferenciales de problemas de transporte mediante discretizaciones espaciotemporales, el método de Lattice Boltzmann produce modelos de autómatas cinéticos a nivel mesoscópico de manera que las propiedades macroscópicas promediadas obedezcan las ecuaciones de conservación deseadas (Chen y Doolen, 1998).

3.1 Ecuación BGK

Dentro de los modelos de Lattice Boltzmann, el esquema de colisión más extendido para simulación de fluidos es el denominado BGK que aproxima las ecuaciones de Navier-Stokes (Bhatnagar *et al* 1954). Existen numerosas variantes de BGK según el número de direcciones utilizadas para definir una vecindad. Para la simulación de fluidos en 3D existen desde versiones con 27 direcciones hasta modelos minimalistas que logran simular procesos complejos con sólo 13 direcciones como (D'Humières *et al* 2001), también los hay de 15, 19, 23, etc. El modelo más ampliamente difundido es el d3q19 propuesto por Qian et al (Qian *et al* 1992) por presentar el mejor equilibrio entre precisión y tamaño de memoria utilizada.

La ecuación de Lattice Boltzmann para el modelo de colisión BGK en su versión de 3 dimensiones y 19 direcciones discretas se presenta de la siguiente manera:

$$f_i\left(x+\delta e_i, t+\delta\right)-f_i\left(x,t\right)=\frac{1}{\tau}\left[f_i^{(eq)}\left(x,t\right)-f_i\left(x,t\right)\right], \quad i=0,1,...,18 \tag{1}$$

donde la ecuación (1) está escrita en unidades físicas. Ambas escalas, espacial y temporal, tienen el valor δ en unidades físicas. $f_i(x,t)$ es la función de distribución de densidad de partículas en dirección e_i en la celda ubicada en x en el tiempo $t(x, t)$. El

lado derecho de la ecuación (1) representa el término de colisión donde τ es la relajación temporal simple que controla el rango de aproximación al equilibrio. La función de distribución de equilibrio $f_i^{(eq)}(x,t)$ depende solamente de la velocidad y densidad local y tienen la siguiente forma (Qian, d'Humieres y Lallemand, 1992):

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

Donde w_i son pesos asociados con las velocidades de la grilla e_i. Los pesos w_i son valores constantes derivados desde la distribución de Maxwell Boltzmann (Geier 2006). Para el modelo D3Q19 los pesos son:

$$w_0 = \frac{1}{9}, \;\; w_i = \frac{1}{18}, \; i = 1:6; \;\; w_i = \frac{1}{36}, \; i = 7:18. \tag{3}$$

Si nombramos los ejes principales como norte, sur, este, oeste, frente y atrás (en inglés N, S, E, W, F, B) y elegimos los ejes de coordenadas como se muestra en la figura 1, los vectores de velocidad de las partículas e_i quedan de la siguiente manera:

Cuadro1. Vectores de velocidad de las 19 direcciones

// C E N W S B F NE NW SW SE BE NB BW SB FE NF FW SF
e_x =0.0, 1.0, 0.0,-1.0, 0.0, 0.0, 0.0, 1.0,-1.0,-1.0, 1.0, 1.0, 0.0,-1.0, 0.0, 1.0, 0.0,-1.0, 0.0;
e_y =0.0, 0.0, 1.0, 0.0,-1.0, 0.0, 0.0, 1.0, 1.0,-1.0,-1.0, 0.0, 1.0, 0.0,-1.0, 0.0, 1.0, 0.0,-1.0;
e_z =0.0, 0.0, 0.0, 0.0, 0.0,-1.0,1.0, 0.0, 0.0, 0.0, 0.0,-1.0,-1.0,-1.0,-1.0,1.0, 1.0, 1.0, 1.0;

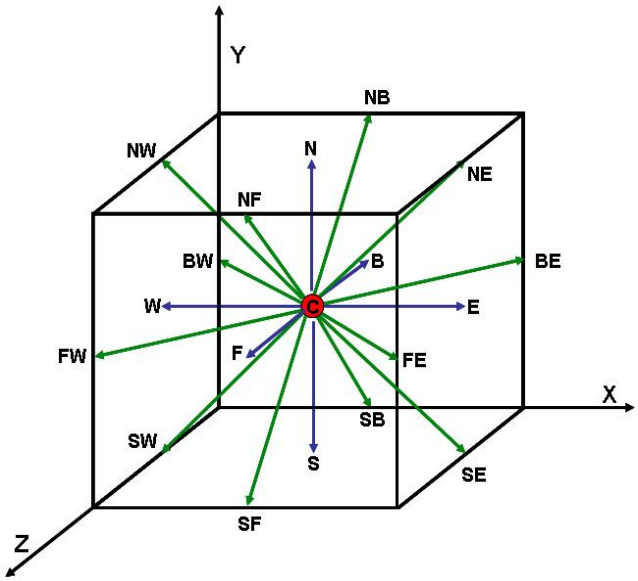


Fig. 1. LBM: Funciones de distribución en el modelo D3Q19.

La densidad de cada nodo, ρ, y la velocidad macroscópica del fluido, $u=(u_x, u_y, u_z)$, se definen en términos de la función de distribución de la siguiente manera:

$$\sum_{i=0}^{18} f_i = \rho, \qquad \sum_{i=0}^{18} f_i e_i = \rho u \tag{4}$$

La presión está dada por $p=c_s^2\rho$, donde c_s es la velocidad del sonido con $c_s^2=1/3$, y la

viscosidad cinemática v está dada por:

$$\nu = \left[\frac{(2\tau - 1)}{6}\right]\delta \tag{5}$$

4 IMPLEMENTACIÓN

Los esquemas de acceso a memoria y la representación interna de los datos cobran importancia en el diseño de un algoritmo eficiente al aumentar considerablemente el volumen de datos a transferir por celda con respecto a los modelos de dos dimensiones, (Rinaldi *et al* 2009). Existen estrategias básicas, como la utilización de una sola grilla con funciones macros de direccionamiento que han sido utilizadas exitosamente en entornos distribuidos de CPUs (Wellein *et al* 2006) y son aplicables a GPU. Pero quedan por explorar otros aspectos, como el acceso a memoria, la secuencia de ejecución, o la manera en que se recorren las estructuras, los cuales deben implementarse de manera diferente para obtener una mejor performance dada la particular arquitectura de las placas gráficas.

4.1 Estructuras de Datos

La primera etapa de la implementación consiste en definir la representación interna que se dará a los datos —*i.e.* la forma en que se almacenan las distribuciones de partículas— que tiene un impacto substancial en la performance (Wellein *et al* 2006). En este trabajo se utilizó un solo arreglo unidimensional y definiciones Macro de acceso, lo cual permite modificar fácilmente la disposición de los datos y evaluar diferentes alternativas. La macro en cuestión utiliza 3 índices dimensionales *(x, y, z)* para referenciar la celda de la matriz, un índice q que indica la velocidad (de 0 a 18) y el parámetro t que sirve para alternar entre los dos pasos de tiempo. También recibe como parámetro las dimensiones totales de la grilla y la cantidad total de direcciones (Q=19).

El cuadro 1 muestra el código de la macro para la disposición de almacenamiento utilizada. Con este esquema, los valores de una misma dirección e_i para todas las celdas del dominio quedan en posiciones consecutivas de memoria al movernos en la dirección x de la grilla tridimensional representada.

Cuadro2. Código para disposición de almacenamiento

```
int dirQ(int x, int y, int z, int q, int t, int DIMX, int DIMY, int DIMZ, int Q){
    return (x + y*DIMX + z*DIMX*DIMY + DIMX*DIMY*DIMZ*f +
    DIMX*DIMY*DIMZ*Q*k);
}
```

4.2 Algoritmo

El algoritmo principal se implementó siguiendo un esquema "pull" (Wellein *et al,* 2006) donde para cada celda, primero se obtienen las distribuciones *fs* "trayendo" los valores desde celdas vecinas, y luego se aplican las condiciones de contorno, se calcula el paso de colisión y por último el de relajación. Al realizar todos los pasos del algoritmo como uno solo, se reduce notablemente la cantidad de datos transferidos desde y hacia la memoria principal. El uso de dos copias de los datos una para t y otra para $t+1$ facilita la paralelización eliminando la dependencia de datos. Los datos se leen de una copia y se escriben en otra durante un paso del algoritmo completo.

4.3 Accesos a memoria agrupados

Programar el algoritmo CUDA de manera que los accesos a la memoria sean siempre alineados repercute enormemente en la performance del código. Al asignar celdas consecutivas en *x* a un bloque de *threads* paralelos, los accesos a memoria de éstos se realizan a elementos consecutivos. Es decir, el *thread x* lee la dirección *base+x*, el *thread x+1* lee la dirección *base+x+1* y así sucesivamente. Esto hace que los accesos a memoria principal de la placa gráfica de cada grupo de 16 *threads* (denominado *half warp* en CUDA) se realicen como uno solo y no como una secuencia de 16 accesos (NVIDIA 2010b).

En versiones CUDA previas a la 3.0 (utilizada en este trabajo), o en placas con compatibilidad inferior a 1.2, las restricciones para que este tipo de accesos se agrupen son mayores. El campo base debe ser múltiplo del tamaño del dato y cada *thread* debe traer un dato diferente de los otros. El principal problema que esto acarrea en una implementación de LBM está en el paso de advección y en la actualización de las *fs* con valores provenientes de celdas que no tengan el mismo índice *x* (*e.g.* cualquiera de las *fs* denominadas con E o W en la figura 1). Algunos trabajos que utilizan estas versiones como (Tölke 2007, Kuznik *et al* 2010) proponen esquemas de dos recorridos a la grilla para garantizar que el paso de advección se realice de manera alineada.

En versiones CUDA de la 3.0 en adelante y placas con compatibilidad 1.2 o superior, para alinear todos los accesos solamente debe cumplirse que los 16 *threads* paralelos accedan a datos que se encuentran en un espacio de memoria continuo de tamaño dependientes del tipo de dato accedido. Utilizando el esquema propuesto en este trabajo, la peor situación se daría cuando los accesos se agrupan como 2 en lugar de 1 sólo, pero nunca como los 16 accesos originales. Por lo tanto no es necesario realizar dos pasadas para implementar un LBM eficientemente.

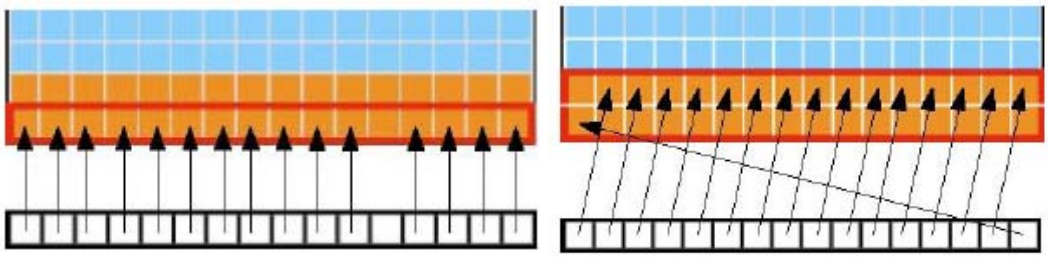


Figura 2 Ejemplos de16 *theads* que acceden a memoria y que se realizan como uno solo. (NVIDA 2010b)

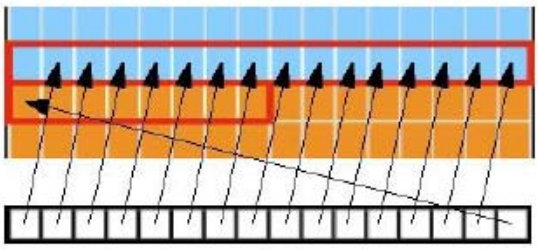


Figura 3 Ejemplo de16 accesos que se agrupan en 2 accesos a memoria. (NVIDA 2010b)

4.4 Uso de memoria compartida

Como el acceso a la memoria principal de la placa es de 100 a 150 veces más lento que el acceso a los otros tipos de memoria que posee el dispositivo (registros, constantes

y memoria compartida), en cada iteración el primer paso es copiar alineadamente los datos de memoria principal de la placa (denominada *global*) a la memoria compartida (*shared*), para luego trabajar sobre éstos. Para el código 3D implementado, la actualización de una celda se compone de los siguientes pasos:

1. Leer las funciones de distribución de las celdas adyacentes en memoria global, *i.e.* $f_i(x\text{-}e\delta t,\ t\text{-}\delta t)$ a memoria compartida.
2. Sincronizar los *threads*.
3. Aplicar las condiciones de contorno para obtener las *fs* faltantes.
4. Calcular ρ, u y $f_i^{(eq)}$
5. Sincronizar.
6. Escribir los valores actualizados en la celda actual, i.e. $f_i(x,t)$ a memoria global.

Los pasos 1 y 6 son los que acceden a memoria global y tienen que hacerse de manera alineada como se explicó en la sección 4.3

4.5 Configuración de ejecución

El dominio se divide en bloques de *threads* donde los índices *y* y *z* son constantes y a cada índice *x* le corresponde un *thread*. Por lo tanto cada *thread* ejecuta el código de una sola celda y el nivel de paralelismo alcanzado es total (figura 4).

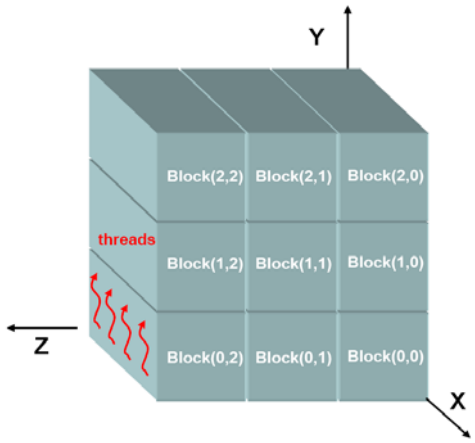


Figura 4.Esquema de división del dominio en bloques de *threads*.

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

4.6 Condiciones de contorno

Las condiciones de contorno implementadas son las originalmente presentadas en (Maier *et al* 1996), extendidas a segundo orden en dos dimensiones en (Zou y He 1997). Estas condiciones fueron generalizadas al modelo d3Q19 en (Hecht y Harting 2010) permitiendo combinar condiciones de contorno del tipo "on-site no slip" con condiciones externas impuestas de presión o velocidad. Para simular el problema descrito en la sección 5, se definieron 19 tipos de celdas diferentes con su

correspondiente cálculo de *fs* restantes. Los tipos de celdas se muestran en el cuadro 3.

Cuadro 3. Tipos de celda

```
FL=0;  //Flujo(Flow) (celdas internas, sen condicion de contorno)
 L=1;   //Capa superior (Lid) (Velocidad constante Umax Oeste a Este)
 B=2;   //Pared trasera (Back) (Ux=0 y Uy=0)
 E=3;   //Pared este (East) (Uy=0 y Uz=0)
 F=4;   //Pared del Frente (Front) (Ux=0 y Uy=0)
 W=5;   //Pared Oeste (West) (Uy=0 y Uz=0)
 S=6;   //Piso (South) (Ux=0 Y Uz=0)
 BE=7;  //Vértice trasero este (Back East) (Uy=0)
 EF=8;  //Vértice frente este (Front East) (Uy=0)
 FW=9;  //Vértice frente oeste (Front West) (Uy=0)
 WB=10; //Vértice trasero oeste (West Back) (Uy=0)
 BS=11; //Vértice trasero del piso (Back South) (Ux=0)
 ES=12; //Vértice este del piso (East South) (Uz=0)
 FS=13; //Vértice frontal del piso (Front South) (Ux=0)
 WS=14; //Vértice oeste del piso (West South) (Uz=0)
 BSE=15;//Esquina (Back South East) (sin forzar velocidades)
 ESF=16;//Esquina (East South Front) (sin forzar velocidades)
 FSW=17;//Esquina (West South Front) (sin forzar velocidades)
 WSB=18;//Esquina (West South Back) (sin forzar velocidades)
```

Si bien en algunos casos las velocidades forzadas son las mismas, el cálculo es diferente porque las *fs* faltantes en el paso de advección son distintas. Para implementar la capa superior con velocidad constante impuesta, se tomó toda la matriz de celdas con $x = DIMX-1$, *i.e.* sin considerar los bordes de manera diferente. En las cuatro esquinas inferiores no se aplica ninguna condición de velocidad específica ya que estas celdas no tienen partículas que se dirijan hacia el flujo.

Una de las recomendaciones más importantes de CUDA en cuanto a perfomance es intentar reducir las posibles bifurcaciones del código, ya que un grupo de *threads* paralelos que entra en diferentes ramas del código se ejecuta como si todos los *threads* recorrieran cada una de las ramas consecutivamente (CUDA 2010b). En el caso particular de las condiciones de contorno de LBM no existe otra solución, pero si se analiza el dominio y la división de *threads* propuesta, en el peor de los casos un grupo de 16 *threads* puede entrar a lo sumo en 3 condiciones diferentes. Y, dado que el código ejecutado en las diferentes condiciones de contorno es muy simple, el problema sigue estando acotado por el acceso a memoria.

5 SIMULACIÓN DE CAVIDAD CÚBICA

El problema estacionario de cavidad cúbica fue el problema elegido como caso de estudio. Por no existir solución analítica al problema, los resultados se comparan con otros encontrados en la literatura. El problema se describe como el escurrimiento estacionario de un fluido incompresible en una cavidad cúbica, compuesta de paredes impermeables rugosas en la que se impone un movimiento en la pared superior de la cavidad. Dicha pared se mueve con velocidad constante (*Umax*) en la dirección *x* mientras que las demás paredes permanecen inmóviles como lo muestra la figura 5.

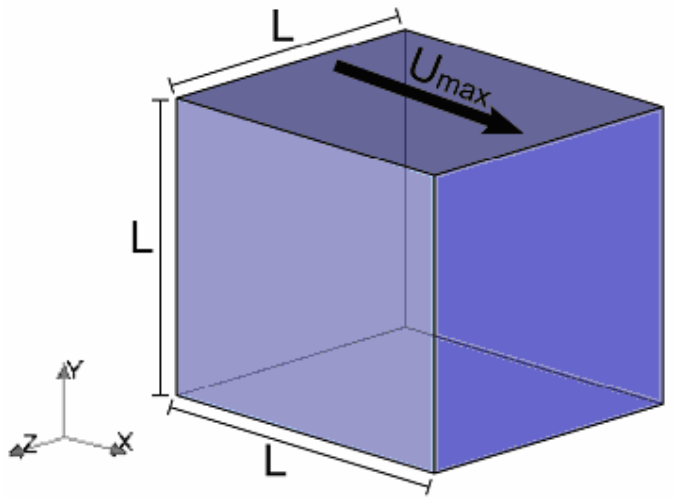


Figura 5. Esquema del problema de escurrimiento en cavidad cúbica.

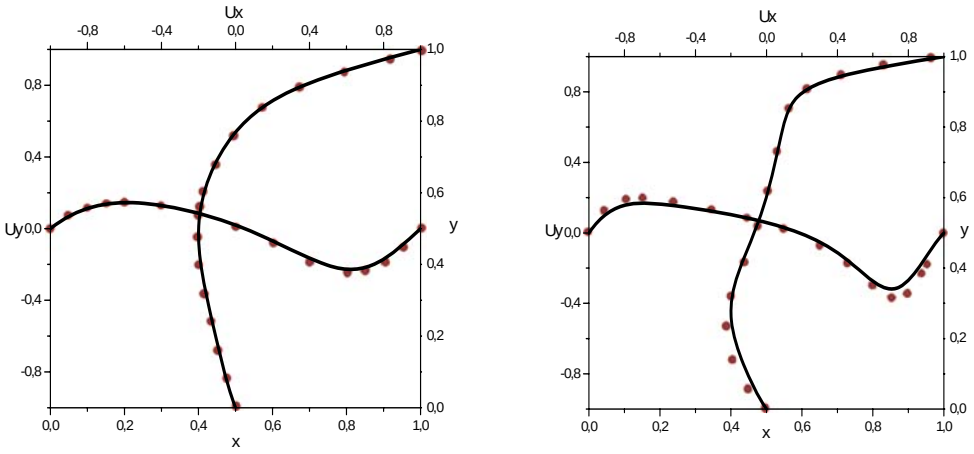
El problema de la cavidad cúbica se caracteriza con el número adimensional de Reynolds el cual se mide en este caso como

$$\mathrm{Re} = \frac{LU_{\max}}{\nu} \tag{6}$$

Donde ν es la viscosidad cinemática del fluido (Ec. 5). Se realizaron simulaciones numéricas de este problema para tres diferentes números de Reynolds 100, 400 y 1000 en grillas con dimensiones 16^3, 64^3 y 128^3 respectivamente, con las condiciones de contorno descritas anteriormente. Las simulaciones fueron hechas partiendo de una velocidad inicial cero en todas las direcciones de las celdas y una densidad específica de 1. Como criterio de convergencia se tomó la diferencia relativa media de los campos de velocidad entre un paso de tiempo y el siguiente. (eq. 7)

$$\sum_i \frac{\left\| U\left(x_i, t+\delta\right) - U(x_i, t) \right\|}{\left\| U(x_i, t+\delta \right\|} \leq 10^{-6} \tag{7}$$

En las figuras 6, 7 y 8, se grafican las componentes de velocidad normalizada u_y a lo largo de la dirección x que pasa por el centro geométrico de la cavidad cúbica y las componentes de velocidad u_x a lo largo de la dirección y que pasa por el mismo centro para los diferentes números de Reynolds. Las gráficas se comparan con los resultados de (Yang *et al* 1998).



Figuras 6 y 7. Componentes de velocidad u_x e u_y en el centro de la cavidad para *Re*:100 y 400 respectivamente.

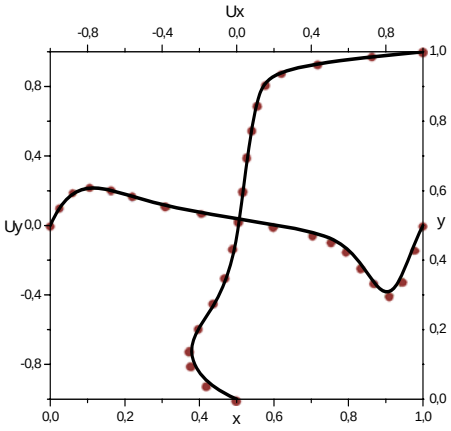


Figura 8: Componentes de velocidad u_x e u_y en el centro de la cavidad para *Re*:1000.

En las figuras 9, 10 y 11 se muestran imágenes 3D con las líneas de corriente que pasan por el eje central ($x=0,5$; $y=0,5$) calculadas a partir del campo de velocidades, donde se puede observar el vórtice central generado en cada caso y la naturaleza tridimensional del escurrimiento de fluido. Puede apreciarse cómo el vórtice central cambia de posición y forma en función del número de Reynolds.

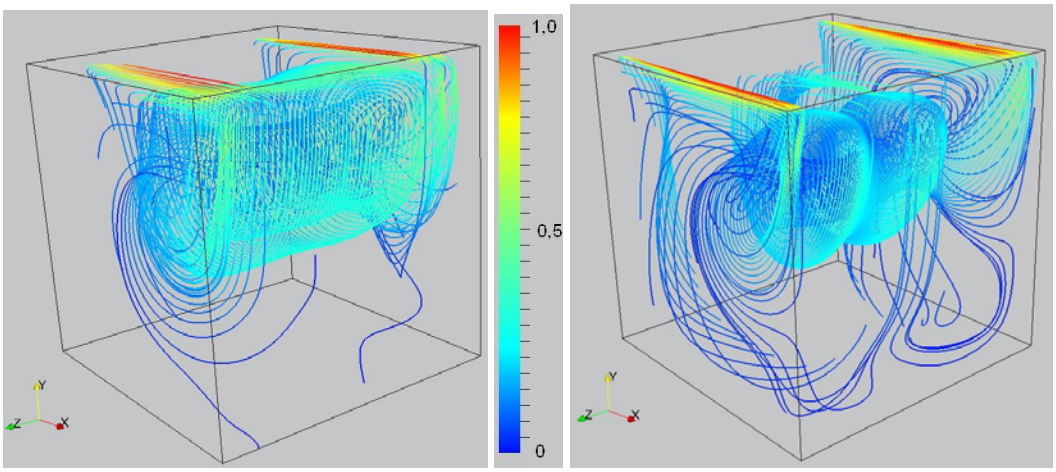


Figura 9 y 10: Líneas de corriente calculadas a partir del campo de velocidades para *Re*: 100 y 400 respectivamente.

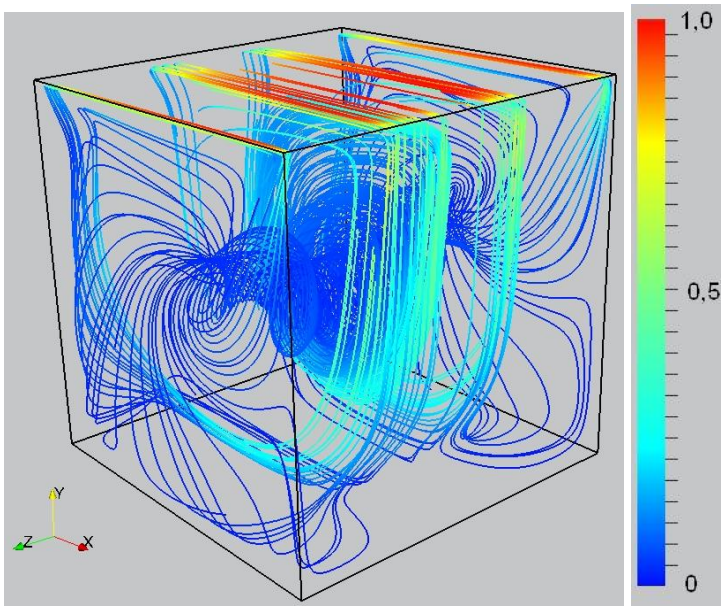


Figura 11: Líneas de corriente calculadas a partir del campo de velocidades para *Re*: 1000.

6 PERFORMANCE

La unidad utilizada en la medición de performance de los modelos de Lattice Boltzmann es el número de celdas actualizadas en un segundo o LUPS (Lattice-Site Updates per Second) (Lammers y Küster 2007). En la tabla 2 se muestra, en función del tamaño de grilla, la comparación de tiempo promedio de una iteración para toda la grilla y los correspondientes MLUPS aproximados para las versiones del modelo LBM en C y CUDA ejecutados sobre las correspondientes plataformas (CPU y GPU) y el *speedup* alcanzado. Puede verse que a partir de grillas de 64 celdas cúbicas, el *speedup* es de dos órdenes de magnitud.

Tamaño de la Grilla	Bloques y *threads*	Tiempo de CPU (ms)	MLUPS CPU	Tiempo de GPU (ms)	MLUPS GPU	Speedup
32^3 32768	32 x 32 32	15.3	2,1	0,165	198	92
64^3 262144	64 x 64 64	124	2,1	1,066	245	116
96^3 884736	96 x 96 96	429	2,06	3,410	259	125
128^3 2097152	128 x 128 128	1092	1,92	9,629	217	113
144^3 2985984	144 x 144 144	1580	1,88	12,815	233	123
160^3 4096000	160 x 160 160	2164	1,89	16,512	248	131

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

Cabe señalar que la versión para CPU se implementa como tres ciclos anidados, uno para cada dirección y tanto el sentido de recorrido de la grilla como la macro de direccionamiento están optimizadas para este tipo de arquitecturas con memoria caché. La CPU utilizada tiene un procesador de doble núcleo AMD Athlon 64 X2 Dual Core de 2,41 GHz y 4 Gb. de memoria RAM. Al tratarse de una aplicación con un solo hilo de ejecución, la performance no mejora con procesadores más modernos de 4 u 8 núcleos.

6.1 Ancho de banda

Para analizar el ancho de banda real alcanzado en las simulaciones se calcula la cantidad de datos transferidos medidos en bytes desde y hacia memoria global por celda para cada iteración. Este valor se multiplica por el número de LUPS máximo alcanzado y se compara con el ancho de banda teórico de la placa. En el paso de advección se leen 19 *floats* (1 de cada dirección) más un *int* que define el tipo de celda. Luego de los cálculos se reescriben los 19 valores de punto flotante. Por lo tanto se transfieren 38×4 bytes + 1×2 bytes = 154 bytes, a 248 MLUPS el ancho de banda alcanzado es de 38.2 Gbytes/s, que es aproximadamente el 35% del ancho de banda pico de la placa.

7 CONCLUSIONES

Se presentó una implementación en paralelo sobre GPU un modelo de Lattice Boltzmann de tres dimensiones y 19 velocidades para simulación de fluidos. El código se validó con la simulación del problema clásico de la cavidad cúbica con velocidad constante en una cara. Para evaluar la performance global del modelo, se realizó una implementación equivalente para CPU. Se obtuvieron aceleraciones de dos órdenes de magnitud de la versión GPU respecto de la CPU. Mediante uso de memoria compartida y accesos alineados se logró alcanzar el 35% del ancho de banda teórico de la placa. En ambos se trató de equipos de escritorio convencionales cuyos valores no exceden los 2000 dólares para el equipo completo, y tanto la GPU sola como el CPU tienen un valor aproximado de 350 dólares.

8 REFERENCIAS

Ansumali S., Karlin I. V. and Öttinger H. C. Consistent Lattice Boltzmann Method, *Europhys. Lett*. 63, 798, 2003.

Bhatnagar, P., Gross, E. and Krook, M. A model for collisional processes in gases I: small amplitude processes in charged and neutral one-component system. *Phys. Rev.* 94, 511, 1954

Boon J.-P. and Rivet J.-P., Lattice Gas Hydrodynamics. *Cambridge University Press,* Cambridge, 2001.

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

D'Humières, D., Bouzidi, M. and Lallemand, P. Thirteen-velocity three-dimensional lattice Boltzmann model. Physical Review E, Vol 63, 066702. 2001.

Geier, M.C., AB Initio Derivation of the Cascaded Lattice Boltzmann Automaton. Tesis de Doctorado, University of Freiburg – IMTEK, September 2006.

Goodnight, N. CUDA/OpenGL Fluid Simulation. (Online): http://new.math.uiuc.edu/MA198-2008/schaber2/fluidsGL.pdf 2007.

Hecht, M., Harting, J. Implementation of on-site velocity boundary conditions for D3Q19 lattice Boltzmann simulations. Journal of Statistic Mechanics. 1742-5468/10/P01018. 2010.

Higuera F. and Succi S., Simulating the flow around a circular cylinder with a lattice Boltzmann equation, *Europhys, Lett.* 8, 517, 1989

Karlin I. V., Ferrante A. and Öttinger H. C. Perfect entropy functions of the Lattice Boltzmann method, *Europhys. Lett.* 47, 182, 1999.

Kuznik F., Obrecht C., Rusaouen G., Roux J., LBM based flow simulation using GPU computing processor, Computers and Mathematics with Applications, 59, 2380-2392, 2010.

Lammers, P. and Küster U. Recent Performance Results of the Lattice Boltzmann Method, *High Performance Computing on Vector Systems* 2006. Part 2. 51-59. 2007.

Maier, R. S., Bernard, R. S., Grunau, D. W. Boundary conditions for the lattice Boltzmann method. Phys. Fluids Vol. 8, 1070-6631/96/8(7)/1788/14. 1996.

NVIDIA, NVIDIA CUDA Home Page. (Online): http://www.nvidia.com/object/cuda_home.html 2008.

NVIDIA. NVIDIA CUDA Compute Unified Device Architecture – Programing Guide Version 3.1. (Online): http://developer.download.nvidia.com 2010a.

NVIDIA. NVIDIA CUDA Compute Unified Device Architecture – NVIDIA CUDA C Best Practice Guide Version 3.1. (Online): http://developer.nvidia.com/object/cuda_3_1_downloads.html 2010b.

Qian, Y., d'Humie`res, D., and Lallemand, P. Recovery of Navier–Stokes equations using a lattice-gas Boltzmann method. *Europhys. Lett.* 17, 479, 1992.

Rinaldi, P., Dari, E., Dalponte, D., Vénere, M., Clausse, A. Métodos de Lattice Boltzmann sobre GPU para simulación de fluidos. Comparación con NS mediante Elementos Finitos. Mecánica Computacional. Vol. XXVIII, pp. 273, ISSN 1666-6070. 2009.

Shan X. and He X., Discretization of the Velocity Space in the Solution of the Boltzmann Equation. *Phys. Rev. Lett.* 80, 65, 1998.

Tölke, J. Implementation of a Lattice Boltzmann kernel using the Compute Unified Architecture Developer by nVIDIA. (Online): http://www.irmb.tu-bs.de/UPLOADS/toelke/Publication/toelked2q9.pdf 2007.

Wellein, G., Zeiser, T., Hager, G., Donath, S. On the single processor performance of simple lattice Boltzmann kernels, *Computers & Fluids*, vol. 35, 910-919 (2006)

Yang, J.Y., Yang, S.C., Chen, Y.N., Hsu, C.A., Implicit weighted eno schemes for the three-dimensional incompressible navier-stokes equations. J. Comput. Phys., 146(1):464–487, 1998. ISSN 0021-9991. 1998.

Zhao, Ye. Lattice Boltzmann based PDE Solver on the GPU. *Visual Comput 2007*. Springer-Verlag, 10.1007/s00371-007-0191-y, 2007.

Zou, Q. and He, X., On pressure and velocity boundary conditions for the lattice Boltzmann BGK model, *Phys. Fluids*, vol.9 (6), pp.1591-1598, 1997.