Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXX, págs. 2979-3008 (artículo completo)
Oscar Möller, Javier W. Signorelli, Mario A. Storti (Eds.)
Rosario, Argentina, 1-4 Noviembre 2011

RESOLUCIÓN DE LAS ECUACIONES DE NAVIER-STOKES UTILIZANDO CUDA

Santiago D. Costarelli[a], **Rodrigo R. Paz**[a,b], **Lisandro D. Dalcin**[a,b] **y Mario A. Storti**[a,b]

[a]*Facultad de Ingeniería y Ciencias Hídricas, Universidad Nacional del Litoral, Ciudad Universitaria, Paraje "El Pozo", CP3000 Santa Fe, Argentina, http://www.fich.unl.edu.ar/*

[b]*Centro Internacional de Métodos Numéricos en Ingeniería, CONICET/INTEC, Colectora Ruta Nac 168, Km 472, Paraje "El Pozo", CP3000 Santa Fe, Argentina, http://www.cimec.org.ar*

Palabras Clave: CUDA, Navier-Stokes, Poisson, FFT

Resumen. En este trabajo se propone una resolución de las ecuaciones de Navier-Stokes para el caso de fluidos Newtonianos incompresibles utilizando la arquitectura CUDA[1] provista por NVIDIA[2]. Se utiliza para la discretización espacial un esquema de diferencias finitas en grillas *staggered* (para evitar el desacople en la presión), y el método de Pasos Fraccionados (Fractional-Step) para la integración temporal. El paso predictor (problema de advección para las ecuaciones de cantidad de movimiento) se resuelve utilizando el esquema de Adams-Bashforth de segundo orden estabilizando los términos convectivos con el método QUICK. Además, el paso de Poisson (para imponer la incompresibilidad) es resuelto iterativamente mediante el método de Gradientes Conjugados precondicionando al sistema utilizando transformadas rápidas de Fourier.

En el presente desarrollo se utilizan librerías estándar de CUDA para el manejo de matrices y vectores, como ser Thrust[3] y CUSP[4], además de CUFFT[5] para las transformadas rápidas de Fourier. De esta forma, mediante las herramientas aportadas por las anteriores se confeccionan los kernels necesarios enfatizando la utilización de memoria shared, accesos fusionados a la memoria global, reduciendo al mínimo la cantidad de registros por thread, entre otros.

A continuación se presenta una serie de casos de estudio con el objetivo de validar el desarrollo y de, posteriormente, comparar las performances obtenidas con implementaciones en otras arquitecturas (CPU, unicore y multicore).

[1]http://developer.nvidia.com/cuda-downloads
[2]http://arg.nvidia.com/page/home.html
[3]http://code.google.com/p/thrust/
[4]http://code.google.com/p/cusp-library/
[5]http://developer.nvidia.com/cufft

1. CONCEPTOS

1.1. Ecuaciones de Navier-Stokes para fluidos incompresibles

Las ecuaciones que gobiernan a los fluidos incompresibles pueden ser escritas en forma conservativa como

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u} \otimes \mathbf{u}) &= -\frac{1}{\rho}\nabla p + \nu \Delta \mathbf{u} + \rho \mathbf{f}_e \\ \nabla \cdot \mathbf{u} &= 0 \end{aligned} \tag{1}$$

teniendo en cuenta que $\mathbf{u}$ es el campo velocidad (en 3D presenta 3 componentes), ρ es la densidad, p es la presión, ν es la viscosidad cinemática y $\mathbf{f}_e$ representa a las fuerzas ejercidas por el exterior sobre el fluido. Se tiene entonces que los campos incógnita son dos, a saber: presión y velocidad.

1.2. Pasos fraccionados

El método de Pasos Fraccionados (o Fractional Step, de ahora en adelante F-S) es en esencia un esquema de integración temporal que propone una resolución del sistema (1). Si bien existen varios métodos para tratarlo el principal aporte del método en cuestión es la utilización de transformadas rápidas de Fourier (FFT) como paso intermedio, que como es bien conocido, presenta implementaciones en GPGPU con alto grado de rendimiento.

Procediendo con el método en primer instancia se resuelven las ecuaciones de cantidad de movimiento considerando sólo los términos de tensiones y convectivos, así se obtienen un campo de velocidad intermedio (en el llamado *paso predictor*) que no cumple con la condición de incompresibilidad, por lo que debe ser corregido. A continuación se resuelve una ecuación de Poisson para la presión teniendo en cuenta que el campo de velocidad final debe de cumplir con la ecuación de continuidad (o la incompresibilidad). Finalmente con el campo presión resultante se corrige el campo velocidad obtenido anteriormente (*paso corrector*) y así se impone la condición de incompresibilidad.

Sin considerar las fuerzas externas y considerando un esquema de integración temporal del tipo Forward Euler se define la primer etapa de F-S como

$$\frac{\mathbf{u}^* - \mathbf{u}^n}{\Delta t} = -\nabla \cdot (\mathbf{u} \otimes \mathbf{u})^n + \nu \Delta \mathbf{u}^n, \tag{2}$$

donde se observa que sólo se resuelven los términos de convección y de tensiones. No está demás decir que se ha escogido Forward Euler como integración temporal sólo en aras de simplicidad, recordando el problema de estabilidad inherente de los métodos explícitos (Costarelli et al., 2011). En la práctica métodos semi-implícitos o implícitos son utilizados, en especial en este trabajo se ha utilizado Adams-Bashforth de segundo orden. Se tiene además que debe aplicarse algún criterio para la primer iteración, puesto que no se tiene información de una iteración previa. Es así que se propone aplicar Forward Euler para la primer iteración y posteriormente Adams-Bashforth.

Una vez resueltas las ecuaciones de cantidad de movimiento obteniendo el campo de velocidades intermedio, se tiene una ecuación que incorpora los términos de presión, esta es

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^*}{\Delta t} = -\frac{1}{\rho}\nabla p^{n+1}, \tag{3}$$

donde se observa que para obtener el valor del campo de velocidades en el paso de tiempo $n+1$ debe de tenerse el campo de presiones en ese mismo paso. Es así que aplicando divergencia a ambos miembros de la ecuación (3) e imponiendo la condición de incompresibilidad en el paso $n+1$ se obtiene

$$\begin{aligned} \mathbf{u}^{n+1} &= \mathbf{u}^* - \frac{\Delta t}{\rho}\nabla p^{n+1}, \\ \underbrace{\nabla\cdot\mathbf{u}^{n+1}}_{=0 \text{ por incomp}} &= \nabla\cdot\mathbf{u}^* - \frac{\Delta t}{\rho}\nabla\cdot\left(\nabla p^{n+1}\right), \\ \Delta p^{n+1} &= \frac{\rho}{\Delta t}\left(\nabla\cdot\mathbf{u}^*\right), \end{aligned} \tag{4}$$

es decir, una ecuación de Poisson para la presión. Una vez resuelta la ecuación (4), se procede con la ecuación (3) para obtener el campo de velocidad buscado.

Finalmente se presenta la ecuación para los pasos de tiempo $[1; N]$ que se ha seguido en este trabajo, es decir, la ecuación de cantidad de movimiento con la integración temporal de Adams-Bashforth de segundo orden

$$\mathbf{u}^* = \mathbf{u}^n + \frac{\Delta t}{2}\left[3R(\mathbf{u}^n) - R(\mathbf{u}^{n-1})\right], \tag{5}$$

en donde $R(\mathbf{u}^n)$ está definido como

$$R\left(\mathbf{u}^n\right) = -\nabla\cdot\left(\mathbf{u}\otimes\mathbf{u}\right)^n + \nu\Delta\mathbf{u}^n, \tag{6}$$

y lo mismo para $R\left(\mathbf{u}^{n-1}\right)$. Finalmente se requiere de historia del campo de velocidades en un paso anterior.

1.3. Grillas staggered

Se tiene que la falta del término de derivada de densidad (como en el caso compresible) y la discretización centrada (es decir, de segundo orden espacial) en las ecuaciones de cantidad de movimiento genera un desacople entre presión y velocidad en los métodos de corrección de presión que derivan en errores para la presión con componentes oscilatorios de alta frecuencia (Hirsch, 2007).

Una solución al problema del desacople consiste en utilizar distintas grillas para presión y velocidades. De esta forma una posible elección son grillas cuyos nodos de velocidad están desplazadas en $\delta/2$, donde δ simboliza a Δx, Δy y Δz según direcciones x, y y z para el caso 3D, con respecto a algún grilla de presión. Luego la conservación se realiza sobre celdas cuyo centro se encuentra ubicado en el nodo donde se están calculando y su tamaño está definido por el Δ de cada dimensión. Así se elimina completamente el desacople de nodos pares e impares para la presión y los campos de velocidad y presión están totalmente acoplados (balances sobre el centro y caras de la celda).

En la Figura (1) se muestra un caso 2D de grillas staggered introducidas en este Capítulo, las líneas continuas pertenecen a la grilla estructurada homogénea del campo presión, mientras que las líneas discontinuas pertenecen a aquellas pertenecientes al campo velocidad.

1.4. Estabilización del término convectivo - QUICK

Las componentes del campo de velocidad están desplazadas en media unidad de longitud de acuerdo a la dirección que le corresponda. Como será visto más adelante será necesario obtener

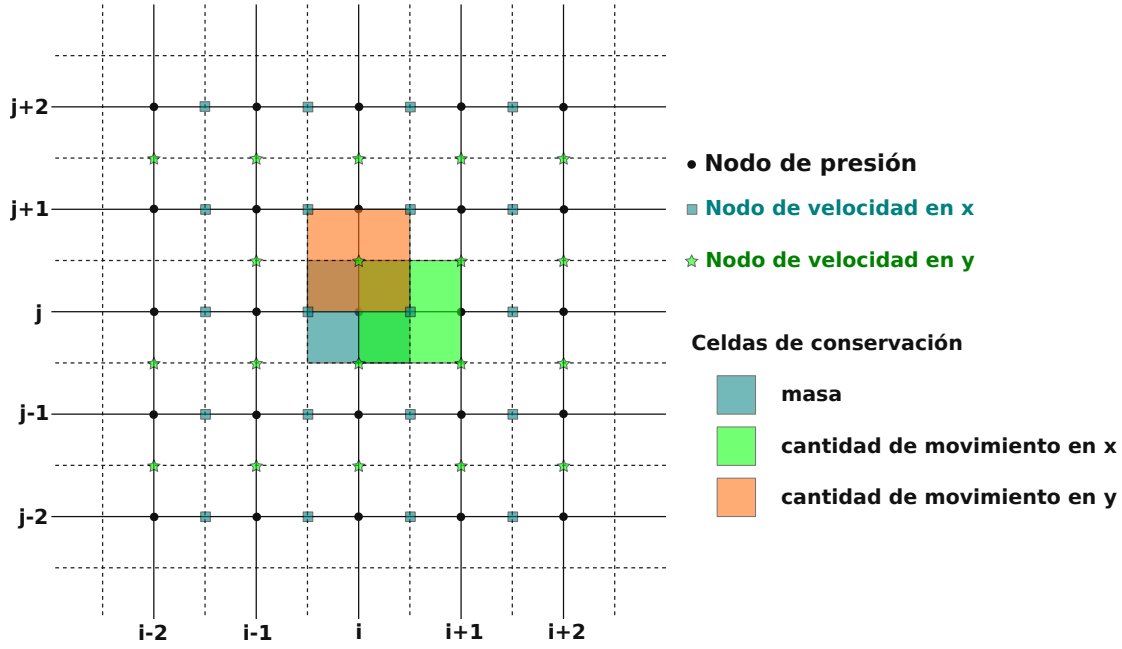


Figura 1: Ejemplo de grilla staggered 2D.

valores de presiones y velocidades descentradas, es decir, en otros valores nodales que no sean donde están definidos. Esta descentralización se realiza justo sobre media unidad de longitud, por ende es necesario un esquema de interpolación sobre esa posición que garantice un término de error de, al menos, el error de truncamiento del esquema de integración temporal. Existen varias formas de interpolar un valor entre nodos, considerando un dominio de interpolación de tres nodos uno puede definir un polinomio de segundo orden definido por sus coordenadas.

Considerando la topología de los nodos presentados en la Figura (2), y haciendo combinaciones lineales de diversas expansiones de Taylor sobre el nodo e en función a los restantes, se obtiene

$$u_e = \frac{3}{8}u_E + \frac{6}{8}u_P - \frac{1}{8}u_W - \mathcal{O}(\Delta x^3), \tag{7}$$

una interpolación de nodos desplazados en media unidad por dirección cuyo error de truncamiento es proporcional a $\mathcal{O}(\Delta x^3)$. Un proceso similar se utiliza para encontrar la expresión cuando se tiene $u_x < 0$, que resulta ser

$$u_e = \frac{3}{8}u_P + \frac{6}{8}u_E - \frac{1}{8}u_{EE} - \mathcal{O}(\Delta x^3), \tag{8}$$

y similarmente en las direcciones y y z.

Lo que no se ha introducido al momento es que este esquema de interpolación en realidad actúa como estabilización para el término convectivo en las ecuaciones de cantidad de movimiento. Su asimetría con respecto al nodo central y su forma cuadrática le brindan conservación de masa y otras características de estabilidad que, si bien no van a ser introducidas en este texto, pueden ser vistan en (Leonard, 1979).

1.5. Discretización

Desagregando la ecuación (1) en sus respectivas direcciones se obtiene para el caso 3D el sistema

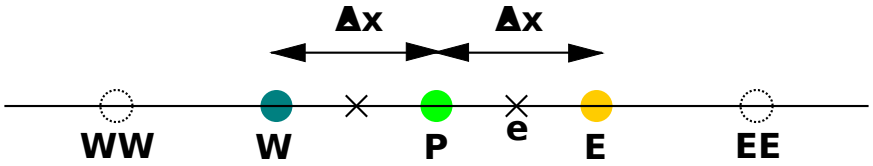


Figura 2: Nodos necesarios para el cálculo en 1D del esquema de interpolación QUICK cuando $u_x > 0$.

$$\begin{aligned}
\frac{\partial u}{\partial t} + \frac{\partial (uu)}{\partial x} + \frac{\partial (uv)}{\partial y} + \frac{\partial (uw)}{\partial z} &= -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu \Delta u, \\
\frac{\partial v}{\partial t} + \frac{\partial (vu)}{\partial x} + \frac{\partial (vv)}{\partial y} + \frac{\partial (vw)}{\partial z} &= -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu \Delta v, \\
\frac{\partial w}{\partial t} + \frac{\partial (wu)}{\partial x} + \frac{\partial (wv)}{\partial y} + \frac{\partial (ww)}{\partial z} &= -\frac{1}{\rho}\frac{\partial p}{\partial z} + \nu \Delta w, \\
\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} &= 0.
\end{aligned} \tag{9}$$

Considerando la definición del residuo del campo de velocidades $\mathbf{u}$ dada por la ecuación (6), es que se tiene que el residuo aplicado a la componente u del campo de velocidades (esto es, velocidad según la dirección x) como

$$\begin{aligned}
\left(R^*\left(u^n\right)\right)_{i+\frac{1}{2},j,k} = &- \left\{ \frac{(uu^Q)^n_{i+1,j,k} - (uu^Q)^n_{i,j,k}}{\Delta x} \right\} \\
&- \left\{ \frac{(vu^Q)^n_{i+\frac{1}{2},j+\frac{1}{2},k} - (vu^Q)^n_{i+\frac{1}{2},j-\frac{1}{2},k}}{\Delta y} \right\} \\
&- \left\{ \frac{(wu^Q)^n_{i+\frac{1}{2},j,k+\frac{1}{2}} - (wu^Q)^n_{i+\frac{1}{2},j,k-\frac{1}{2}}}{\Delta z} \right\} \\
&+ \nu \left\{ \frac{u^n_{i+\frac{3}{2},j,k} - 2u^n_{i+\frac{1}{2},j,k} + u^n_{i-\frac{1}{2},j,k}}{\Delta x^2} \right. \\
&+ \frac{u^n_{i+\frac{1}{2},j+1,k} - 2u^n_{i+\frac{1}{2},j,k} + u^n_{i+\frac{1}{2},j-1,k}}{\Delta y^2} \\
&+ \left. \frac{u^n_{i+\frac{1}{2},j,k+1} - 2u^n_{i+\frac{1}{2},j,k} + u^n_{i+\frac{1}{2},j,k-1}}{\Delta z^2} \right\},
\end{aligned} \tag{10}$$

donde se observa que el residuo se toma en la iteración n-ésima, nodo $(i+\frac{1}{2}, j, k)$ de la componente u del campo de velocidades $\mathbf{u}$ (claramente realizando el balance sobre la celda de cantidad de movimiento según x), y el superíndice Q haciendo referencia QUICK.

Se tiene además que algunos nodos que aparecen en la ecuación (10) no se encuentran en la grilla de velocidades de u, por lo cual requieren de aproximaciones que definan a los mismos. Un esquema simple para solucionar lo anterior consiste en promediar el valor mediante los dos nodos más próximos, según la dirección que se requiera. Entonces considerando esto se obtiene

$$
\begin{aligned}
u^n_{i+1,j,k} &= \frac{1}{2}\left(u^n_{i+\frac{3}{2},j,k} + u^n_{i+\frac{1}{2},j,k}\right), \\
u^n_{i,j,k} &= \frac{1}{2}\left(u^n_{i+\frac{1}{2},j,k} + u^n_{i-\frac{1}{2},j,k}\right), \\
v^n_{i+\frac{1}{2},j+\frac{1}{2},k} &= \frac{1}{2}\left(v^n_{i+1,j+\frac{1}{2},k} + v^n_{i,j+\frac{1}{2},k}\right), \\
v^n_{i+\frac{1}{2},j-\frac{1}{2},k} &= \frac{1}{2}\left(v^n_{i+1,j-\frac{1}{2},k} + v^n_{i,j-\frac{1}{2},k}\right), \\
w^n_{i+\frac{1}{2},j,k+\frac{1}{2}} &= \frac{1}{2}\left(w^n_{i+1,j,k+\frac{1}{2}} + w^n_{i,j,k+\frac{1}{2}}\right), \\
w^n_{i+\frac{1}{2},j,k-\frac{1}{2}} &= \frac{1}{2}\left(w^n_{i+1,j,k-\frac{1}{2}} + w^n_{i,j,k-\frac{1}{2}}\right).
\end{aligned}
\tag{11}
$$

Mientras que aquellos nodos que requieren de la aproximación QUICK se definen como

$$
\begin{aligned}
\left(u^Q\right)^n_{i+1,j,k} &= \begin{cases} c_0 u^n_{i+\frac{3}{2},j,k} + c_1 u^n_{i+\frac{1}{2},j,k} + c_2 u^n_{i-\frac{1}{2},j,k}, & si \;\; u^n_{i+1,j,k} \geq 0 \\ c_0 u^n_{i+\frac{1}{2},j,k} + c_1 u^n_{i+\frac{3}{2},j,k} + c_2 u^n_{i+\frac{5}{2},j,k}, & si \;\; u^n_{i+1,j,k} < 0 \end{cases}, \\
\left(u^Q\right)^n_{i,j,k} &= \begin{cases} c_0 u^n_{i+\frac{1}{2},j,k} + c_1 u^n_{i-\frac{1}{2},j,k} + c_2 u^n_{i-\frac{3}{2},j,k}, & si \;\; u^n_{i,j,k} \geq 0 \\ c_0 u^n_{i-\frac{1}{2},j,k} + c_1 u^n_{i+\frac{1}{2},j,k} + c_2 u^n_{i+\frac{3}{2},j,k}, & si \;\; u^n_{i,j,k} < 0 \end{cases}, \\
\left(u^Q\right)^n_{i+\frac{1}{2},j+\frac{1}{2},k} &= \begin{cases} c_0 u^n_{i+\frac{1}{2},j+1,k} + c_1 u^n_{i+\frac{1}{2},j,k} + c_2 u^n_{i+\frac{1}{2},j-1,k}, & si \;\; v^n_{i+\frac{1}{2},j+\frac{1}{2},k} \geq 0 \\ c_0 u^n_{i+\frac{1}{2},j,k} + c_1 u^n_{i+\frac{1}{2},j+1,k} + c_2 u^n_{i+\frac{1}{2},j+2,k}, & si \;\; v^n_{i+\frac{1}{2},j+\frac{1}{2},k} < 0 \end{cases}, \\
\left(u^Q\right)^n_{i+\frac{1}{2},j-\frac{1}{2},k} &= \begin{cases} c_0 u^n_{i+\frac{1}{2},j,k} + c_1 u^n_{i+\frac{1}{2},j-1,k} + c_2 u^n_{i+\frac{1}{2},j-2,k}, & si \;\; v^n_{i+\frac{1}{2},j-\frac{1}{2},k} \geq 0 \\ c_0 u^n_{i+\frac{1}{2},j-1,k} + c_1 u^n_{i+\frac{1}{2},j,k} + c_2 u^n_{i+\frac{1}{2},j+1,k}, & si \;\; v^n_{i+\frac{1}{2},j-\frac{1}{2},k} < 0 \end{cases}, \\
\left(u^Q\right)^n_{i+\frac{1}{2},j,k+\frac{1}{2}} &= \begin{cases} c_0 u^n_{i+\frac{1}{2},j,k+1} + c_1 u^n_{i+\frac{1}{2},j,k} + c_2 u^n_{i+\frac{1}{2},j,k-1}, & si \;\; w^n_{i+\frac{1}{2},j,k+\frac{1}{2}} \geq 0 \\ c_0 u^n_{i+\frac{1}{2},j,k} + c_1 u^n_{i+\frac{1}{2},j,k+1} + c_2 u^n_{i+\frac{1}{2},j,k+2}, & si \;\; w^n_{i+\frac{1}{2},j,k+\frac{1}{2}} < 0 \end{cases}, \\
\left(u^Q\right)^n_{i+\frac{1}{2},j,k-\frac{1}{2}} &= \begin{cases} c_0 u^n_{i+\frac{1}{2},j,k} + c_1 u^n_{i+\frac{1}{2},j,k-1} + c_2 u^n_{i+\frac{1}{2},j,k-2}, & si \;\; w^n_{i+\frac{1}{2},j,k-\frac{1}{2}} \geq 0 \\ c_0 u^n_{i+\frac{1}{2},j,k-1} + c_1 u^n_{i+\frac{1}{2},j,k} + c_2 u^n_{i+\frac{1}{2},j,k+1}, & si \;\; w^n_{i+\frac{1}{2},j,k-\frac{1}{2}} < 0 \end{cases}.
\end{aligned}
\tag{12}
$$

De la misma forma se calculan el resto de las ecuaciones de cantidad de movimiento.

La ecuación de continuidad se define mediante una simple discretización centrada realizando el balance sobre la celda centrada en el nodo de presión *(i,j,k)* obteniendo de esta manera

$$
\frac{u^{n+1}_{i+\frac{1}{2},j,k} - u^{n+1}_{i-\frac{1}{2},j,k}}{\Delta x} + \frac{v^{n+1}_{i,j+\frac{1}{2},k} - v^{n+1}_{i,j-\frac{1}{2},k}}{\Delta y} + \frac{w^{n+1}_{i,j,k+\frac{1}{2}} - w^{n+1}_{i,j,k-\frac{1}{2}}}{\Delta z} = 0. \tag{13}
$$

Con respecto a la segunda etapa del desarrollo, se debe calcular una ecuación de Poisson para la presión que presenta la forma

$$
\left(\Delta p^{n+1}\right)_{i,j,k} = \frac{\Delta t}{\rho}\left(\nabla \cdot \mathbf{u}^*\right)_{i,j,k}, \tag{14}
$$

notar que el balance se realiza en la celda de presión, es decir, centrada en el nodo *(i,j,k)*. Entonces como primer paso se calcula la divergencia y luego se aplica el factor de escala. La

divergencia se calcula como

$$(\nabla \cdot \mathbf{u}^*)_{i,j,k} = \frac{u^*_{i+\frac{1}{2},j,k} - u^*_{i-\frac{1}{2},j,k}}{\Delta x} + \frac{v^*_{i,j+\frac{1}{2},k} - v^*_{i,j-\frac{1}{2},k}}{\Delta y} + \frac{w^*_{i,j,k+\frac{1}{2}} - w^*_{i,j,k-\frac{1}{2}}}{\Delta z}, \tag{15}$$

donde se observa que el balance también se hace en la celda centrada en el nodo de presión *(i,j,k)* y el truncamiento resulta $\mathcal{O}(\Delta x^2_{max})$, donde Δx_{max} hace referencia al máximo δ de la grilla.

Además en la tercer etapa se necesitan los gradientes de presión para la corrección del campo de velocidad $\mathbf{u}^*$, luego se calculan como

$$\nabla p^{n+1} = \left\{ \begin{array}{c} \left(\frac{\partial p}{\partial x}\right)^{n+1}_{i+\frac{1}{2},j,k} \\ \left(\frac{\partial p}{\partial y}\right)^{n+1}_{i,j+\frac{1}{2},k} \\ \left(\frac{\partial p}{\partial z}\right)^{n+1}_{i,j,k+\frac{1}{2}} \end{array} \right\} = \left\{ \begin{array}{c} \frac{p^{n+1}_{i+1,j,k} - p^{n+1}_{i,j,k}}{\Delta x} \\ \frac{p^{n+1}_{i,j+1,k} - p^{n+1}_{i,j,k}}{\Delta y} \\ \frac{p^{n+1}_{i,j,k+1} - p^{n+1}_{i,j,k}}{\Delta z} \end{array} \right\}, \tag{16}$$

donde se observa el balance de los gradientes de presión sobre las celdas de velocidad y también presentan un error de truncamiento $\mathcal{O}(\Delta x^2_{max})$. Finalmente se obtiene el campo de velocidades $\mathbf{u}^{n+1}$ mediante

$$\mathbf{u}^{n+1}_{i,j,k} = \left\{ \begin{array}{c} u^{n+1}_{i+\frac{1}{2},j,k} \\ v^{n+1}_{i,j+\frac{1}{2},k} \\ w^{n+1}_{i,j,k+\frac{1}{2}} \end{array} \right\} = \left\{ \begin{array}{c} u^*_{i+\frac{1}{2},j,k} - \frac{\Delta t}{\rho}\left(\frac{\partial p}{\partial x}\right)^{n+1}_{i+\frac{1}{2},j,k} \\ v^*_{i,j+\frac{1}{2},k} - \frac{\Delta t}{\rho}\left(\frac{\partial p}{\partial y}\right)^{n+1}_{i,j+\frac{1}{2},k} \\ w^*_{i,j,k+\frac{1}{2}} - \frac{\Delta t}{\rho}\left(\frac{\partial p}{\partial z}\right)^{n+1}_{i,j,k+\frac{1}{2}} \end{array} \right\}. \tag{17}$$

1.6. Resolvedor rápido de Poisson

Considerando que

$$\underbrace{\mathbf{P}^{-1}\mathbf{A}}_{*}\mathbf{x} = \mathbf{P}^{-1}\mathbf{b}, \tag{18}$$

en donde $\mathbf{P}$ hace referencia a la matriz de precondicionamiento, lo ideal sería que el término sobre la etiqueta (*) resultara en la identidad, o bien diagonal, con el objetivo de tener una inversa trivial. El problema de Poisson para la presión es requerido por la segunda etapa de F-S, en donde la discretización sobre grillas homogéneas del operador lineal Laplaciano (forzando la propiedad de positivo definido mediante el signo introducido) considerando sin pérdida de generalidad condiciones de borde periódicas, determina una matriz simétrica $\mathbf{A}$ de la forma

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \dots & 0 & 0 & -1 \\ -1 & 2 & -1 & \dots & 0 & 0 & 0 \\ 0 & -1 & 2 & \dots & 0 & 0 & 0 \\ 0 & 0 & -1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -1 & 2 & -1 \\ -1 & 0 & 0 & \dots & 0 & -1 & 2 \end{bmatrix}. \tag{19}$$

Se tiene entonces que es deseable encontrar una matriz de precondicionamiento que, actuando sobre la matriz Laplaciana, reproduzca una matriz diagonal tal y como se ha introducido anteriormente. Considere la transformada discreta de Fourier unidimensional definida como

$$X_k = \sum_{j=1}^{N} x_j e^{i\frac{2\pi kj}{N}}, \tag{20}$$

donde X_k es la componente frecuencial k-ésima, x_j es el valor del nodo j, y la exponencial es la base de la transformada discreta de Fourier (periódica, de período N). Sea la matriz $\mathbf{F}$ el resultado de evaluar dicha base exponencial, se puede demostrar que sus columnas representan los autovectores de la matriz $\mathbf{A}$. De esta forma (Loan, 1992, Sección 4.5.3), se tiene

$$\mathbf{D} = \mathbf{F}^{-1}\mathbf{A}\mathbf{F}, \tag{21}$$

donde $\mathbf{D}$ es una matriz diagonal cuyas componentes resultan los N autovalores de $\mathbf{A}$. Considerando que $\mathbf{A} = \mathbf{F}\mathbf{D}\mathbf{F}^{-1}$ luego se obtiene finalmente

$$\mathbf{P}^{-1} = \mathbf{A}^{-1} = \mathbf{F}\mathbf{D}^{-1}\mathbf{F}^{-1}, \tag{22}$$

con lo cual

$$\mathbf{x} = \mathbf{F}\mathbf{D}^{-1}\mathbf{F}^{-1}\mathbf{b}, \tag{23}$$

de esta forma el algoritmo de Gradientes Conjungados (CG) converge en una iteración aplicando sólo tres cálculos, dos transformadas de Fourier (una hacia delante y otra inversa) con $\mathcal{O}(Nlog_2(N))$ operaciones para cada transformación, y una multiplicación de N elementos con $\mathcal{O}(N)$ operaciones. Esta precondición puede extender a otras condiciones de borde, como ser, las fijas, o bien en grillas no homogéneas. La extensión 2D y 3D sigue el mismo camino.

Cuando se tienen cuerpos embebidos en el dominio la solución del precondicionador es tan buena que el CG converge en pocas iteraciones. Si bien en este trabajo no se tratan casos de estudio en esta temática, un análisis profundo puede ser encontrado en (Storti et al., 2010).

2. CUDA

2.1. Introducción

En esencia la arquitectura CUDA fue introducida al mercado a inicios de 2007 por la empresa NVIDIA y básicamente se ideó con el objeto de facilitar la programación de componentes GPU (graphics processing units). Dado que la GPU fue concebida para un cálculo intensivo (principalmente dado a que libera al CPU de la tarea de la renderización de los objetos en la pantalla) es que al ver la creciente curva de rendimiento obtenida a lo largo de estos últimos años, se comenzó a utilizar la misma para cálculos de propósito general.

En la jerga GPU una GPU/GPGPU (general purpose graphics processing units) se identifica como un device, mientras que el hardware que la soporta como un host. Un host puede soportar varios devices (en principio tantos como puertos PCI-Express se tengan). Cada GPU implementa básicamente 4 tipos de componentes: los *streaming multiprocesors* (SM), las *unidades especiales* (SFU), las de *doble precisión* y el *caché*. Sin entrar en detalles (para tecnologías Tesla[6] -arquitectura GPU utilizada en este trabajo- y anteriores) los SM presentan un conjunto de 8 procesadores escalares denominados SP (scalar processors) capaces de procesar evaluandos de

[6]http://www.nvidia.com/object/tesla_computing_solutions.html

precisión simple. Las unidades SFU implementan esquemas de evaluación optimizados (a veces por hardware, otras por software) para funciones trascendentales y recíprocos (entre otros). Las unidades de doble precisión realizan operaciones con evaluandos de doble precisión (siguiendo los estándares del IEEE 754). Finalmente el caché realiza operaciones análogas a aquel presente en un CPU.

Una función que se ejecuta en un device se denomina *kernel*. Para su procesamiento CUDA conforma lo que se denomina grilla (o *grid*). La grilla presenta en cada nodo un bloque de threads (o *threadblock*). Un thread se define como la mínima unidad de ejecución posible. Un threadblock entonces es un conjunto de threads.

El fundamento por detrás de estas subdivisiones se encuentra de la mano de como la arquitectura distribuye estos threadblocks sobre los SM para que sean procesados. El orden de reparto es desconocido y es efectuado por el driver de CUDA.

Existe una (al menos en Tesla y anteriores) unidad que se encarga de la organización de los threadblocks dentro de cada SM, denominada unidad SIMT. Ella es la encargada de mantener, siempre que se pueda, a los SP activos. Para ello cada threadblock es subdividido en *warps*, un conjunto de 32 threads.

Así la unidad SIMT desglosa estos threadblocks y distribuye warps sobre los SP. La cantidad de warps activos por SM es dependiente del *Compute Capability* del dispositivo, para la Tesla (con Compute Capability 1.3) se tiene que se puede mantener 32 warps activos por SM. El objetivo de tener dicha cantidad de warps activos es que cuando los threads de un determinado halfwarp (conjunto de 16 threads, división de un warp, es la unidad de ejecución por SP en arquitecturas TESLA y anteriores) realizan operaciones de alta latencia (como ser la lectura/escritura sobre memoria local) luego dicho warp es enviado a la cola de espera y un nuevo warp toma el hardware para ser procesado. De esta forma el hardware evita la desocupación de sus SM. Lo anterior se conoce como *zero overheading schedule*.

Si bien el intercambio de procesos en CPU es una tarea costosa, dado a que requiere almacenar el estado del programa para que en el próximo quantum de tiempo de procesador que le sea asignado pueda conocer el punto desde donde debe continuar (pila de instrucciones) y los datos que tenia disponible (así también los recursos a los que poseía acceso); contrariamente en una GPU los recursos son asignados por thread, por lo cual no hay intercambio de estados entre ellos. Así el intercambio de threads se hace sin una carga adicional de rendimiento. El resultado entonces es una unidad que mediante el intercambio masivo de threads puede reducir notablemente la latencia de operaciones, en comparación a otras arquitecturas (p.e. la PC).

El modelo de ejecución de la arquitectura CUDA se basa en lo que se conoce como SIMT *(*single instruction multiple thread), es decir, el objetivo es que todos los threads dentro de un mismo halfwarp (válido en Tesla y anteriores) ejecuten la misma instrucción cada uno a sus respectivos datos, y de esta forma se amortiza la búsqueda de la instrucción en memoria.

Como se ha presentado anteriormente la arquitectura CUDA se basa en el esquema SIMT, y como punto crítico se tiene que, en el mejor de los casos, todos los threads de un halfwarp *deberían* ejecutar la misma instrucción para el conjunto de datos que tienen cargados. Esto en la práctica no resulta generalmente así puesto que los threads de un mismo halfwarp pueden seguir caminos distintos en condicionales (p.e. IF-THEN), el resultado son varias cargas de instrucciones (tantas como caminos condicionales se tengan) y la serialización de los caminos de ejecución.

La arquitectura CUDA presenta actualmente 5 tipos distintos de memorias, en lo que sigue se introducirán los conceptos básicos que rigen el funcionamiento de cada una.

2.1.1. Memoria global y accesos fusionados

La memoria global (global memory) es una vasta área de memoria que permite tanto lectura como escritura, con una latencia media de 400-600 ciclos de reloj. Si bien no presenta accesos vía caché (al menos no en la tecnología Tesla), su gran capacidad la hace fundamental cuando se requiere de gran cantidad de datos en la GPU. La unidad Tesla C1060 que se dispone presenta un total de 4 Gbytes de memoria global.

Para reducir su gran latencia, se implementan lecturas/escrituras optimizadas en caso de cumplir ciertos requisitos, denominadas *coalesce readings/writings* o lecturas/escrituras fusionadas.

Suponiendo que se tiene un determinado warp en ejecución asociado a una operación de lectura o escritura, entonces se tienen diferentes escenarios donde el acceso fusionado puede ocurrir dependiendo del valor de *Compute Capability* que se disponga. El acceso fusionado es fundamental en cualquier desarrollo en CUDA, puesto que de utilizarse se aprovecha la máxima eficiencia efectiva otorgada por este tipo de memoria. Una discusión más extensa se accesos fusionados puede encontrarse en (Kirk y mei W. Hwu, 2010; Sanders y Kandrot, 2010).

2.1.2. Memoria constante

La memoria constante (constant memory) es una memoria (de 64 Kbytes para la Tesla) que presenta caché (de 8 Kbytes para la Tesla) y tiene la particularidad que sólo permite lecturas. Presenta accesos optimizados de acuerdo a como se efectúen las lecturas sobre esta.

En particular si todos los threads dentro de un mismo halfwarp acceden a la misma posición de memoria, luego sólo una lectura es realizada (con lo cual en principio se evitarían 15 lecturas extra) y es replicada vía broadcast a todos los threads de ese halfwarp. Además, en caso de que en el próximo halfwarp se requiera de la misma posición de memoria, el acceso se realizaría sobre el caché. Como resultado, estos accesos resultan (en general) casi tan rápidos como el acceso a un registro.

Sin embargo si los threads dentro de un halfwarp accesan a distintas porciones de memoria, los accesos reducirían notablemente su rendimiento, puesto que el broadcast ya no se realizaría a nivel de los 16 threads, sino al número de threads que accedan a la misma posición. El peor caso es cuando todos los threads acceden a posiciones distintas con lo cual el rendimiento se verá seriamente afectado, pudiendo resultar en un acceso con menor rendimiento que uno sobre memoria global (recordando que sobre este último se tiene la capacidad del acceso fusionado).

2.1.3. Memoria compartida

Puede ser considerada como la memoria de mayor importancia dentro de la arquitectura CUDA. Su profundo entendimiento y posterior utilización puede resultar en enormes incrementos de rendimiento en comparación al uso de la memoria global. En esencia la memoria compartida (shared memory) es una memoria de alto rendimiento tanto para lecturas como para escrituras, con visibilidad para todos los threads de un mismo threadblock. La latencia que tienen operaciones sobre ellas resultan 100-200 veces menores a las mismas realizadas sobre la memoria global. En las Tesla se disponen de 16 Kbytes de memoria disponible por SM.

Se utiliza generalmente cuando cada thread dentro de un mismo threadblock requiere de valores que pueden ser cargados por otro thread, así cada uno carga su propio conjunto de datos y lo almacena en memoria compartida, para que luego no haya accesos redundantes a memoria global por cada thread.

La organización de la memoria compartida se desprende del hecho de que está conformada por palabras de 4 bytes. Se tiene además una subdivisión de 16 bancos de memoria, donde palabras sucesivas pertenecen a bancos sucesivos. Es importante notar que la cantidad de bancos coincide con la cantidad de threads en un halfwarp, puesto que están íntimamente ligados.

2.1.4. Memoria local

La memoria local (local memory) es una memoria dedicada por thread y se utiliza para almacenar datos privados. Los datos que allí residen presentan visibilidad sólo al thread que pertenecen. En esencia en una porción de memoria global que actúa como soporte a la memoria de registros. Cuando uno define variables en un kernel de CUDA estas serán depositadas en memoria local cuando: sean arreglos, existan demasiados registros utilizados o bien la estructura completa pueda utilizar demasiada cantidad de espacio de registros.

2.1.5. Memoria de texturas

La memoria de texturas (texture memory) es una memoria con visibilidad para todos los threads sin importar a que threadblock pertenezcan. Es de sólo lectura y cacheada. La disposición de los datos dentro del hardware es 1D o 2D, con lo cual si la naturaleza del problema presenta cierta localidad en la utilización de los datos (p.e. un stencil de diferencias finitas en mallas estructuradas) luego el acceso a esos datos se realiza con lecturas muy eficientes.

2.1.6. Registros

Memoria dedicada por thread de rápido acceso. La GPGPU Tesla C1060 dispone de 16384 registros por SM, lo cual permite concluir que en promedio cada thread dispone de $16384/(32 \times 32) = 16$ registros, considerando que la GPGPU en cuestión puede disponer de 32 warps activos por SM, cada uno con 32 threads.

3. IMPLEMENTACIÓN

Como bien fue argumentado anteriormente los objetivos principales a tener en cuenta para un desarrollo eficiente de un kernel en CUDA, y teniendo en cuenta las características del problema planteado, son:

1- Accesos fusionados,

2- utilización de memoria compartida y

3- minimización de la cantidad de registros por thread.

Si bien existen muchas otras ventajas de la arquitectura CUDA, estas resultan ser las que se pueden utilizar en el planteo propuesto.

3.1. Las estructuras de datos y las condiciones de borde

Es sabido que los campos incógnitas en el problema estudiado son dos: un escalar *presión*, y un vector *velocidad*. Una vez introducido el concepto de grillas staggered y el problema de desacople para nodos de presión, es directo notar que los cálculos en 3D tendrán que efectuarse sobre 4 grillas independientes (en principio). Luego una formulación sencilla de estructura de datos consistiría en disponer de dos grillas, una para presión y otra velocidad.

Si se considerara a M, N y P como las dimensiones del dominio computacional según direcciones x, y y z, una estructura de datos para la presión consistirá en una arreglo unidimensional con $M \times N \times P$ valores, cada uno haciendo referencia a un valor de presión en una determinada coordenada *(i,j,k)*. La segunda estructura consistirá en otro arreglo unidimensional con con $M \times N \times P$ valores donde cada valor es una subestructura con componentes u, v y w, o componentes del campo velocidad según direcciones x, y y z.

A su vez los valores de las grillas se almacenarán según z, y y x, es decir, las primeras P posiciones de ambas estructuras de datos contienen los valores de los nodos $(0, 0, k)$ con $k = 0, 1, \cdots, P-1$, los siguientes P valores son aquellos con coordenadas $(0, 1, k)$ con $k = 0, 1, \cdots, P-1$, y así sucesivamente hasta los primeros $N \times P$ valores. Luego se tiene $(1, 0, k)$ con $k = 0, 1, \cdots, P-1$, posteriormente $(1, 1, k)$ con $k = 0, 1, \cdots, P-1$, hasta que se carguen los respectivos $M \times N \times P$ valores.

En la práctica el disponer de 4 grillas por separado no ha rendido como se esperaba, mientras que multiplicaba el código innecesariamente. Finalmente se optó disponer de sólo dos grillas, una de presión y otras para las velocidades.

La utilización de grillas staggered requiere, por definición, de una reducción de un nodo por dirección sobre uno de los campos (presión o velocidad) dado a que sus nodos estarán centrados en la celda conformada por 4 nodos en 2D, o por 8 en 3D (velocidad o presión). El disponer de la misma cantidad de nodos para ambos campos evita bifurcaciones de predicados (condicionales IF-THEN-ELSE) pero, sin los recaudos necesarios, sólo funciona para condiciones de borde periódicas. De otra forma, es necesario determinar cual de los campos será el que tenga una capa de nodos menos, puesto que las condiciones de borde depende de ello. El cálculo de los flujos es independiente de como se tomen los datos, aunque cada condición de borde deberá de cargar los mismos de manera consistente. El caso de estudio (y por tanto, la carga de datos) que se propone en este trabajo utiliza condiciones de borde periódicas, de esta forma y sin falta de generalidad, lo que sigue del texto trata con este tipo de condiciones.

Considerando que se dispone de todo aquello anterior y posterior al lazo temporal, se presenta en el pseudocódigo (1) la metodología propuesta según Pasos Fraccionados.

Código 1: Lazo temporal y las etapas requeridas por la integración temporal de Pasos Fraccionados.

```
1  for(int n=0;n<nsteps;n++){
2    \\1- Resolución de las ecuaciones de cantidad de movimiento. u^n → R(u^n)
3    \\2- Integración temporal.
4      \\Iteración 0, FE. u^n + Δt R(u^n) → u*
5      \\Iteración 1 a nsteps-1, AB. u^n + Δt/2 [3R(u^n) - R(u^{n-1})] → u*
6    \\3- Cálculo de divergencia de velocidad. ∇·u*
7    \\4- CG+FFT. -Δp^{n+1} = -ρ/Δt ∇·u*
8    \\5- Cómputo de gradientes de presión y corrección de velocidad.
9          u* - Δt/ρ ∇p^{n+1} → u^{n+1}
10 }
```

Las etapas 1 y 4 son las que más tiempo de cómputo requieren y serán descriptas en secciones subsecuentes. La etapa 2 es realizada mediante operaciones algebraicas de nivel 1 (vector-vector) provistas por la API Thrust. La etapa 3, el cálculo de la ecuación Laplaciana en la 4 y el cómputo de gradientes en la 5 se realizan de manera similar y seguirán un esquema simplificado de la etapa 1. Mientras que la corrección de velocidad de la etapa 5 será realizada en un kernel actuando conjuntamente por los gradientes de presión para aprovechar las lecturas sobre la memoria global que ya se han realizado, aunque puede haberse optado por utilizar una actualización vectorial vía Thrust como ha sido utilizado en la etapa 2.

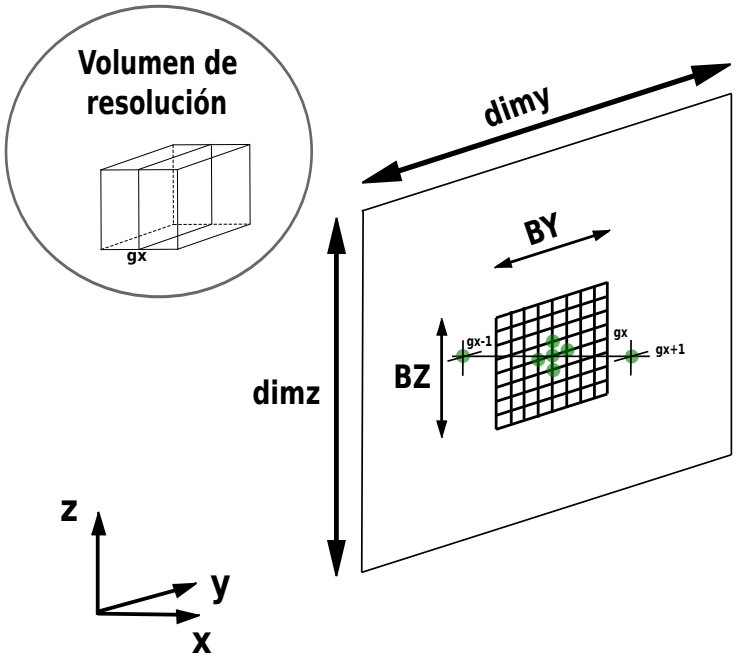


Figura 3: Rodaja número gx del dominio de resolución, se observa un stencil con sus 3 nodos tanto en la dirección y como en la z, y luego los 3 nodos en la dirección x. Además se presenta la subdivisión del plano YZ que es procesada por un SM en particular, o en otras palabras, la subdivisión en theadblocks de tamaños $(\mathrm{BY}, \mathrm{BZ})$.

Con los recursos de la GPGPU Tesla C1060 en mente, se decidió proceder con una resolución sobre el dominio computacional considerando sucesivos planos del mismo. Esto es, si se supone el esquema de almacenado de campos como ha sido introducido en la sección anterior, es decir, según direcciones z, y y x, luego resulta natural la división en sucesivos planos YZ, confeccionando stencils con información de planos contiguos, y una vez finalizados los cálculos, tomar el siguiente plano YZ hasta que finalmente se hayan procesados todos. Entonces se tendrán tantos ¨planos de avance¨ como nodos en la dirección x.

En la Figura (3) se presenta la forma de procesar el plano completo mediante la subdivisión del mismo, y los nodos requeridos para la conformación de stencils. Cada nodo está asociado con un thread, y cada uno de ellos lee al menos información de 3 nodos, el valor actual del campo de velocidad en la posición $(\mathrm{gx}, \mathrm{gy}, \mathrm{gz})$ y los nodos pertenecientes a los planos en dirección x contiguos, es decir, con $\mathrm{gx} \pm 1$.

Considerando las ecuaciones discretas derivadas anteriormente y la Figura (5), se tiene que son necesarios tres planos en dirección x para el cómputo de las ecuaciones de cantidad de movimiento, luego en base al esquema de indexación utilizado dichos planos presentan distintas dimensiones, a saber, $(1+\mathrm{BY}+1, 1+\mathrm{BZ}+1)$ para el plano del medio, $(\mathrm{BY}+1, \mathrm{BZ}+1)$ para el plano de atrás y $(1+\mathrm{BY}, 1+\mathrm{BZ})$ para el plano de adelante. La posición de los 1s dentro de las dimensiones hace referencia para donde se expande esa fila o columna considerando un bloque genérico cuyas dimensiones serían $(\mathrm{BY}, \mathrm{BZ})$. Así el plano de atrás por ejemplo tendrá una fila arriba y una columna a la derecha del bloque genérico, mientras que el plano de adelante tendrá un fila abajo y una columna a la izquierda del bloque genérico. El plano central por su parte, tendrá 2 filas (una arriba y otra abajo) y 2 columnas (una a izquierda y otra a derecha) considerando el bloque genérico. Esta situación se observa en la Figura (5).

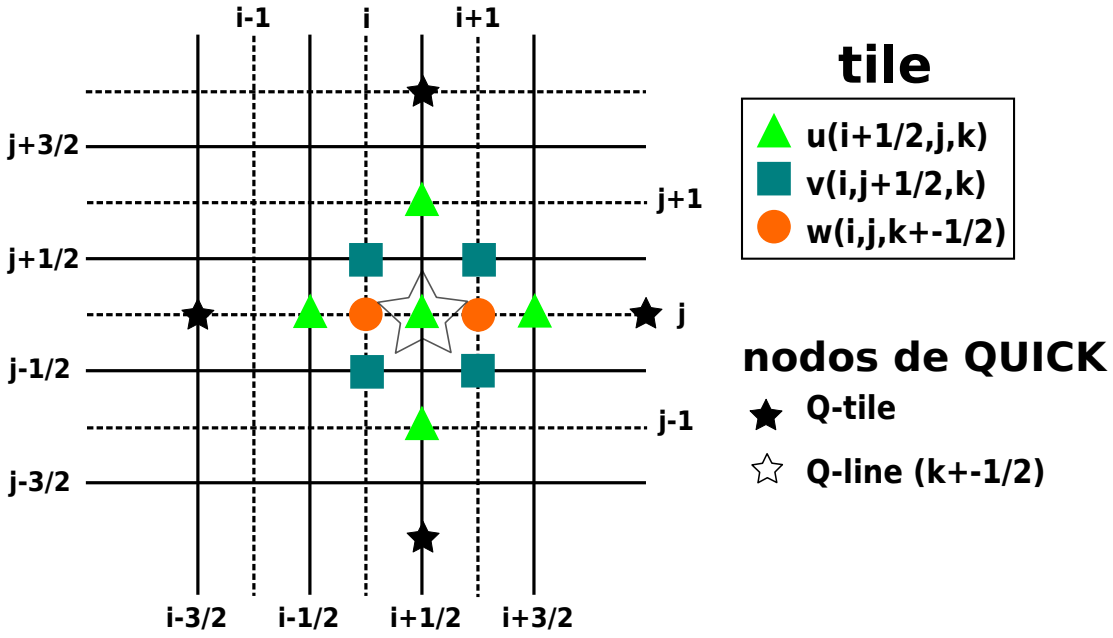


Figura 4: Nodos requeridos para el cálculo del residuo en u. La expresión *TILE* hace referencia al threadblock (o división del plano YZ), mientras que *LINE* a los nodos según la dirección de avance (x).

En el Código (2) se exponen tanto las variables requeridas como finalmente las etapas involucradas en el cómputo de las ecuaciones de cantidad de movimiento, a saber, la etapa de carga de datos, la convección y la difusión (recordando que no se tienen en cuenta términos de fuerzas externas).

Código 2: Kernel de cantidad de movimiento definicion e inicializacion de variables.

```
1  //Planos en memoria compartida.
2  __shared__ velocity<scalar> tileb [BY+1][BZ+1]; //ty+1, tz+1
3  __shared__ velocity<scalar> tilep [1+BY+1][1+BZ+1];
4  __shared__ velocity<scalar> tilef [1+BY][1+BZ]; //ty-1, tz-1
5
6  //Indices globales
7  const int gy = blockIdx.y * blockDim.y + threadIdx.y;
8  const int gz = blockIdx.x * blockDim.x + threadIdx.x;
9
10 //Indices del bloque (BY,BZ) actual
11 const int bty = threadIdx.y;
12 const int btz = threadIdx.x;
13
14 //Macro para indexación
15 #define grid(i,j,k) (dimy*dimz*(i) + dimz*(j) + 1*(k))
16
17 //Auxiliares.
18 velocity<scalar> rpdaux = {0,0,0}, qvel = {0,0,0};
19 scalar velprom;
20
21 //Lazo en dirección x.
22 for(int gx=0;gx<dimx;gx++){
23   //1-Carga de datos.
24   //2-Convección + QUICK.
25   //3-Difusión.
26 }
```

Observar que parece resultar que en cada iteración existe cierta información que vuelve a ser cargada de memoria, es decir, suponiendo que en la iteración actual se tienen cargado tres planos, luego tanto el plano intermedio con el delantero serán reutilizados en la próxima iteración.

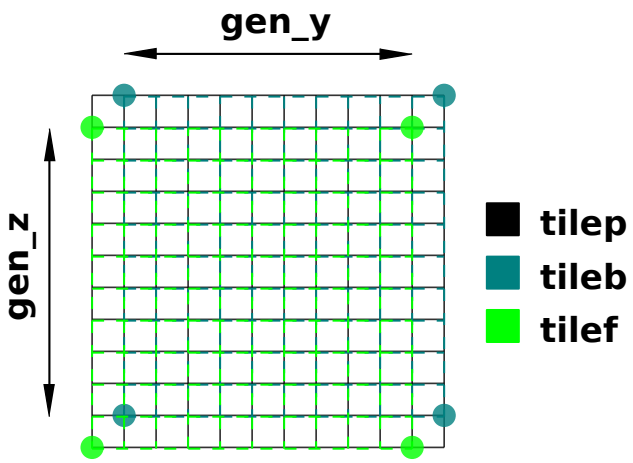


Figura 5: Arreglos bidimensionales necesarios para la confección de los stencils. Se observa que tilep, el plano del medio, tiene como dimensiones (1+BY+1,1+BZ+1), mientras que tileb, el plano de atrás, (BY+1,BZ+1) y tilef, el de adelante, (1+BY,1+BZ).

Sin embargo, pruebas que han sido efectuadas haciendo explícito el reuso de dicha información, no mejora los resultados (en performance) obtenidos, por lo que se concluye que de alguna manera el mismo driver de CUDA realiza estas optimizaciones.

Continuando con el análisis del Código (2) las variables globales gx, gy y gz se utilizan para acceder a cualquier elemento del dominio computacional. Las variables locales ty y tz sólo sirven para acceder a posiciones dentro de un threadblock en particular, observar que existen sólo dos registros para acceder a los 3 planos en memoria compartida, luego el centrado y el de adelante aplican un incremento sobre ese registro cuando efectúan indexaciones, es decir, se utilizan como $\mathrm{tx}+1$ y $\mathrm{ty}+1$ respectivamente. Esto es así puesto que se considera la coordenada $(0,0)$ como el nodo ubicado en $(0,0)$ para el plano de atrás y $(1,1)$ para el plano del medio y el de adelante.

Luego se tiene una macro, y varias variables auxiliares. La utilización de CUDA-GDB para conocer la ubicación física en memoria indica que en presencia de vectores o estructuras (como es el caso de rpdaux y qvel) el almacenamiento se produce en memoria local y no en registros, esto es realizado por el compilador como ha sido introducido anteriormente. Sin embargo parece ser que realiza algún tipo de optimización y opera como si se trataran de registros (puede ser considerado como que conoce que se utilizarán en reiteradas oportunidades), ya que si uno creara variables que no sean arreglos (como por ejemplo los índices, que según CUDA-GDB sí residen en memoria de registros), operara sobre ellas y luego finalmente confeccionara las estructuras con estas variables ninguna ventaja resultará de este proceso.

3.2. Carga de datos

La carga de datos tendrá lugar en básicamente dos secciones del kernel, en la primera se leerán los datos de la memoria global y se asignarán a los planos según corresponda. En esta etapa no se leerán valores nodales en las fronteras requeridos para el cómputo de QUICK, estos serán cargados en la segunda etapa y a medida que se vayan utilizando. De esta forma sólo es necesario un registro que guarde la lectura, y no memoria compartida o lecturas adicionales.

Inicialmente cada thread debe leer los valores del nodo $(\mathrm{gx},\mathrm{gy},\mathrm{gz})$ según corresponda, y además los nodos pertenecientes a planos anteriores y posteriores $\mathrm{gx}\mp 1$, situación que se observa en el Código (3).

Código 3: Lectura de los planos sin considerar valores en frontera.

```
tileb[bty][btz] = (gx==0) ? udev[grid(dimx-1,gy,gz)]
                          : udev[grid(gx-1,gy,gz)];
tilep[(bty+1)][(btz+1)] = udev[grid(gx,gy,gz)];
tilef[(bty+1)][(btz+1)] = (gx==dimx-1) ? udev[grid(0,gy,gz)]
                                       : udev[grid(gx+1,gy,gz)];
```

Donde se observa que dado a la condiciones periódicas que fueron asumidas, como el plano de atrás consiste en el plano YZ en posición $\mathrm{gx}-1$, si el plano del medio actual es $\mathrm{gx}=0$ luego el plano de atrás consistirá en el plano YZ con $\mathrm{gx}=\mathrm{dimx}-1$, y eso es lo que se observa en la primer linea de código. Una situación similar sucede con el plano hacia adelante.

Los términos de las fronteras son cargados por aquellos threads más externos del threadblock, el Código (4) refleja este hecho.

Código 4: Lectura de nodos frontera.

```
if(gy==0){
  tilep[(bty+1)-1][(btz+1)] = udev[grid(gx,dimy-1,gz)];
  tilef[(bty+1)-1][(btz+1)] = (gx==dimx-1) ? udev[grid(0,dimy-1,gz)]
                                           : udev[grid(gx+1,dimy-1,gz)];
}
else if(gy==dimy-1){
  tileb[bty+1][btz] = (gx==0) ? udev[grid(dimx-1,0,gz)]
                              : udev[grid(gx-1,0,gz)];
  tilep[(bty+1)+1][(btz+1)] = udev[grid(gx,0,gz)];
}
else if((bty+1)==1){
  tilep[(bty+1)-1][(btz+1)] = udev[grid(gx,gy-1,gz)];
  tilef[(bty+1)-1][(btz+1)] = (gx==dimx-1) ? udev[grid(0,gy-1,gz)]
                                           : udev[grid(gx+1,gy-1,gz)];
}
else if ((bty+1)==BY){
  tileb[bty+1][btz] = (gx==0) ? udev[grid(dimx-1,gy+1,gz)]
                              : udev[grid(gx-1,gy+1,gz)];
  tilep[(bty+1)+1][(btz+1)] = udev[grid(gx,gy+1,gz)];
}

if(gz==0){
  tilep[(bty+1)][(btz+1)-1] = udev[grid(gx,gy,dimz-1)];
  tilef[(bty+1)][(btz+1)-1] = (gx==dimx-1) ? udev[grid(0,gy,dimz-1)]
                                           : udev[grid(gx+1,gy,dimz-1)];
}
else if(gz==dimz-1){
  tileb[bty][btz+1] = (gx==0) ? udev[grid(dimx-1,gy,0)]
                              : udev[grid(gx-1,gy,0)];
  tilep[(bty+1)][(btz+1)+1] = udev[grid(gx,gy,0)];
}
else if((btz+1)==1){
  tilep[(bty+1)][(btz+1)-1] = udev[grid(gx,gy,gz-1)];
  tilef[(bty+1)][(btz+1)-1] = (gx==dimx-1) ? udev[grid(0,gy,gz-1)]
                                           : udev[grid(gx+1,gy,gz-1)];
}
else if ((btz+1)==BZ){
  tileb[bty][btz+1] = (gx==0) ? udev[grid(dimx-1,gy,gz+1)]
                              : udev[grid(gx-1,gy,gz+1)];
  tilep[(bty+1)][(btz+1)+1] = udev[grid(gx,gy,gz+1)];
}
```

Se puede observar entonces como primero se realiza la consulta de si un nodo pertenece a la frontera del dominio computacional, pues si así resultara debería leer los datos de la frontera contraria. De otra forma, conoce que no está en ninguna frontera por ende está necesariamente en el interior, es más, el rango de los índices globales resulta $\mathrm{gy} \in [\mathrm{BY}-1; \mathrm{dimy}-(\mathrm{BY}-1)]$ y $\mathrm{gz} \in [\mathrm{BZ}-1; \mathrm{dimz}-(\mathrm{BZ}-1)]$.

Al momento se han cargado totalmente aquellos elementos a compartir, pero necesita de

alguna forma asegurar que todos los threads que están colaborando terminaron de cargar los datos, pues de otra forma cuando se calculen los flujos se podrían estar leyendo datos que no sean los correctos (pues no habían sido cargados). Esto se realiza mediante barreras de sincronización que establecen una marca en dicha línea de código y devuelve el control de ejecución a la aplicación sólo si todos los threads de un mismo threadblock llegaron hasta allí.

Código 5: Sincronización.

```
__syncthreads(); //Sincronizando carga de memoria local.
```

Finalmente para terminar de cargar los datos necesarios para la confección de stencils se realizan dos lecturas más, fuera de la barrera de sincronización porque son utilizados por el thread actual. Además se tiene que los nodos en los vértices del plano central no han sido cargados, y sólo dos de ellos se utilizan. Si uno revisara las ecuaciones, notaría que estos nodos corresponden a los nodos cuyas coordenadas son $(0, \mathrm{BZ}-1)$ y $(\mathrm{BY}-1, 0)$. De esta forma las cargas resultante se muestran en el Código (6).

Código 6: Carga de datos en vértices.

```
if((bty+1)==1 && (btz+1)==BZ){
  if(gy==0 && gz==dimz-1){
    tilep[(bty+1)-1][(btz+1)+1] = udev[grid(gx,dimy-1,0)];
  }
  else if(gy==0 && gz!=dimz-1){
    tilep[(bty+1)-1][(btz+1)+1] = udev[grid(gx,dimy-1,gz+1)];
  }
  else if(gy!=0 && gz==dimz-1){
    tilep[(bty+1)-1][(btz+1)+1] = udev[grid(gx,gy-1,0)];
  }
  else{
    tilep[(bty+1)-1][(btz+1)+1] = udev[grid(gx,gy-1,gz+1)];
  }
}
else if((bty+1)==BY && (btz+1)==1){
  if(gy==dimy-1 && gz==0){
    tilep[(bty+1)+1][(btz+1)-1] = udev[grid(gx,0,dimz-1)];
  }
  else if(gy==dimy-1 && gz!=0){
    tilep[(bty+1)+1][(btz+1)-1] = udev[grid(gx,0,gz-1)];
  }
  else if(gy!=dimy-1 && gz==0){
    tilep[(bty+1)+1][(btz+1)-1] = udev[grid(gx,gy+1,dimz-1)];
  }
  else{
    tilep[(bty+1)+1][(btz+1)-1] = udev[grid(gx,gy+1,gz-1)];
  }
}
```

3.3. Términos convectivos y la estabilización vía QUICK

Con los datos requeridos por los stencils de cada nodo cargados en memoria compartida tenemos garantizado un acceso con ancho de banda superior al acceso al de la memoria global de entre 150-200 veces más eficiente. Además, el problema de lecturas fusionadas ya no tiene sentido y sólo ha sido tenido en cuenta en el proceso de carga, donde en general los nodos de borde realizaban accesos sin fusión.

No es la intención del presente documento explicar todas las expresiones presentes en el kernel de cantidad de movimiento, más sí la mecánica de indexación que se tiene para llevar el término de la ecuación discreta a un valor de un plano en particular.

Considerando el cálculo de convección según x y sólo un término en dirección x, y además recordando la ecuación que lo gobierna como

$$\frac{(uu^Q)^n_{i+1,j,k} - (uu^Q)^n_{i,j,k}}{\Delta x}, \tag{24}$$

donde

$$u^n_{i+1,j,k} = \frac{1}{2}\left(u^n_{i+\frac{3}{2},j,k} + u^n_{i+\frac{1}{2},j,k}\right), \tag{25}$$

y

$$\left(u^Q\right)^n_{i+1,j,k} = \begin{cases} c_0 u^n_{i+\frac{3}{2},j,k} + c_1 u^n_{i+\frac{1}{2},j,k} + c_2 u^n_{i-\frac{1}{2},j,k}, & si \;\; u^n_{i+1,j,k} \geq 0 \\ c_0 u^n_{i+\frac{1}{2},j,k} + c_1 u^n_{i+\frac{3}{2},j,k} + c_2 u^n_{i+\frac{5}{2},j,k}, & si \;\; u^n_{i+1,j,k} < 0 \end{cases}, \tag{26}$$

entonces se tiene que de acuerdo al signo de la velocidad $\left(u^Q\right)^n_{i+1,j,k}$ podría como no ser necesario un nodo desplazado en dos posiciones en dirección x. De esta forma existen dos formas de cargar dicho valor: en la primera se carga sin más, como desventaja podría tenerse una lectura inútil, que considerando la latencia de las operaciones sobre la memoria global, en principio la situación debería de ser evitada; en la segunda la lectura es realizada sólo si fuera necesario, que como principal problema introducirá divergencia de threads y según las pruebas que se han realizado presenta un rendimiento peor al primer caso. Finalmente se optó por realizar la lectura, independientemente de su utilización. De esta forma la sección del kernel que realiza la operación descripta por las ecuaciones (24-26) se presenta en el Código (7),

Código 7: Cómputo de un término de la ecuación de cantidad de movimiento - conveccion.

```
qvel = ((gx==dimx-1) || (gx==dimx-2)) ? udev[grid(gx-(dimx-2),gy,gz)]
                                      : udev[grid(gx+2,gy,gz)];

velprom = 0.5*(tilef[(bty+1)][(btz+1)].u+tilep[(bty+1)][(btz+1)].u);

if(velprom >= 0.){
  rpdaux.u = -velprom*(c0*tilef[(bty+1)][(btz+1)].u+
                       c1*tilep[(bty+1)][(btz+1)].u+
                       c2*tileb[bty][btz].u)/deltax;
}
else{
  rpdaux.u = -velprom*(c0*tilep[(bty+1)][(btz+1)].u+
                       c1*tilef[(bty+1)][(btz+1)].u+
                       c2*qvel.u)/deltax;
}
```

en donde se observa como en la primer linea se carga en un auxiliar el valor del nodo desplazado en dos planos según dirección x, posteriormente se realiza el promedio de velocidades como es propuesto en la ecuación (25) y finalmente se hace el cálculo de QUICK según ecuación (26). Los coeficientes c0, c1 y c2 son los coeficientes de QUICK y se encuentran almacenados en la memoria constante con el objetivo de (1) reducir la cantidad de registros y (2) utilizar el caché que dispone; de esta forma sólo una lectura sobre la memoria constante es requerida y luego se tienen replicaciones vía broadcast de lecturas al caché. Esto se observa en el Código (8).

Código 8: Coeficientes de QUICK en memoria constante.

```
//Coeficientes de QUICK.
__device__ __constant__ float c0 = 0.375, c1 = 0.75, c2 = -0.125;
```

Se analizará entonces la indexación utilizada. Los nodos $(i+\frac{1}{2},j,k)$, $(i,j+\frac{1}{2},k)$ e $(i,j,k+\frac{1}{2})$ son accedidos bajo la misma secuencia $\mathrm{tilep}[(\mathrm{bty}+1)][(\mathrm{btz}+1)]$, sólo diferenciando la disposición de las grillas por los campos u, v y w de la estructura adoptada para el campo de velocidades. De otra forma podría decirse que la grillas staggered son accedidas como si fueran una misma grilla, centradas en los nodos de presión.

El acceso a la posición $u^n_{i+\frac{3}{2},j,k}$ se realiza como $\mathrm{tilef}[(\mathrm{bty}+1)][(\mathrm{btz}+1)].\mathrm{u}$, tilef pues está leyendo un nodo desplazado en una unidad del actual desplazamiento de $1/2$ en dirección x. Una situación similar ocurriría si se deseara acceder a los datos del nodo $u^n_{i-\frac{1}{2},j,k}$, solamente que el plano de lectura resultaría tileb, dado a que se alinea con el campo de presión en $p_{i-1,j,k}$. Finalmente los nodos en otras direcciones, o bien cruzados, siguen una lógica similar incrementando o decrementando alguna de las dimensiones de los bloques.

3.4. Términos difusivos

Con respecto a los términos difusivos, siguen el esquema de discretización del operador Laplaciano en 3D que se muestra en el Código (9).

Código 9: Cómputo de un término de la ecuación de cantidad de movimiento - difusión.

```
rpdaux.u += nu * ((tilef[(bty+1)][(btz+1)].u
                  -2.*tilep[(bty+1)][(btz+1)].u
                  +tileb[bty][btz].u)/(deltax*deltax)
                  +
                  (tilep[(bty+1)+1][(btz+1)].u
                  -2.*tilep[(bty+1)][(btz+1)].u
                  +tilep[(bty+1)-1][(btz+1)].u)/(deltay*deltay)
                  +
                  (tilep[(bty+1)][(btz+1)+1].u
                  -2.*tilep[(bty+1)][(btz+1)].u
                  +tilep[(bty+1)][(btz+1)-1].u)/(deltaz*deltaz));
```

3.5. El problema de Poisson para la presión

Para la resolución del problema de Poisson para la presión se implementó un gradiente conjugado, y se utilizó la FFT (transformada rápida de Fourier) provista por la API CUFFT. El concepto es sencillo, resolver el sistema considerando cuerpos inmersos en el fluido utilizando la FFT (Storti et al., 2010) y con su solución precondicionar al Laplaciano que será introducido en el CG. En general la solución obtenida con la FFT es tan buena, que el CG alcanza a converger en pocas iteraciones (si sólo se tiene fluido, en una sola iteración). Cuando existen cuerpos, estos últimos introducen errores que luego el CG deberá amortiguar.

Como se sabe el RHS de la ecuación de Poisson es una divergencia. Esta se calcula como un esquema simplificado de la resolución presentada para las ecuaciones de cantidad de movimiento.

Con el objeto de resolver todas las etapas de Pasos fraccionado en GPU se optó por una implementación sencilla de cálculo de divergencias. Recordando, se tiene que la divergencia se calcula como un balance sobre la celda de presión, es decir, centrada en el nodo *(i,j,k)*

$$(\nabla\cdot\mathbf{u}^*)_{i,j,k}=\frac{u^*_{i+\frac{1}{2},j,k}-u^*_{i-\frac{1}{2},j,k}}{\Delta x}+\frac{v^*_{i,j+\frac{1}{2},k}-v^*_{i,j-\frac{1}{2},k}}{\Delta y}+\frac{w^*_{i,j,k+\frac{1}{2}}-w^*_{i,j,k-\frac{1}{2}}}{\Delta z}, \tag{27}$$

así considerando sólo un plano con dimensiones $(1+\mathrm{BY},1+\mathrm{BZ})$ se tiene

Código 10: Cálculo de la divergencia del campo velocidad.

```
lineu = (gx==0) ? udev[grid(dimx-1,gy,gz)]
                : udev[grid(gx-1,gy,gz)];

udivdev[index] = (tileu[ty][tz].u-lineu.u)/(deltax)+
                 (tileu[ty][tz].v-tileu[ty-1][tz].v)/(deltay)+
                 (tileu[ty][tz].w-tileu[ty][tz-1].w)/(deltaz);
```

Finalmente se aplica el factor de escala $-\rho/\Delta t$ vía Thrust, el signo negado surge pues se resuelve el negado del Laplaciano de la presión, dado a que el Laplaciano es un operador definido negativo y el CG necesita que la matriz LHS sea definida positiva y simétrica (requisito no sólo del CG, sino para garantizar la no singularidad de **A**).

Los detalles de la implementación del CG y la precondición vía FFT no serán introducidos en este documento. Pero en esencia lo que se hace es aplicar tanto al inicio del CG como por cada iteración sobre el sistema, el operador de precondición obtenido vía FFT.

3.6. Corrección del campo de velocidades

Recordando las ecuaciones para el cálculo de gradientes de presión como

$$\nabla p^{n+1} = \left\{ \begin{array}{c} \left(\dfrac{\partial p}{\partial x}\right)^{n+1}_{i+\frac{1}{2},j,k} \\ \left(\dfrac{\partial p}{\partial y}\right)^{n+1}_{i,j+\frac{1}{2},k} \\ \left(\dfrac{\partial p}{\partial z}\right)^{n+1}_{i,j,k+\frac{1}{2}} \end{array} \right\} = \left\{ \begin{array}{c} \dfrac{p^{n+1}_{i+1,j,k} - p^{n+1}_{i,j,k}}{\Delta x} \\ \dfrac{p^{n+1}_{i,j+1,k} - p^{n+1}_{i,j,k}}{\Delta y} \\ \dfrac{p^{n+1}_{i,j,k+1} - p^{n+1}_{i,j,k}}{\Delta z} \end{array} \right\}, \tag{28}$$

y la posterior actualización como

$$\mathbf{u}^{n+1}_{i,j,k} = \left\{ \begin{array}{c} u^{n+1}_{i+\frac{1}{2},j,k} \\ v^{n+1}_{i,j+\frac{1}{2},k} \\ w^{n+1}_{i,j,k+\frac{1}{2}} \end{array} \right\} = \left\{ \begin{array}{c} u^{*}_{i+\frac{1}{2},j,k} - \dfrac{\Delta t}{\rho}\left(\dfrac{\partial p}{\partial x}\right)^{n+1}_{i+\frac{1}{2},j,k} \\ v^{*}_{i,j+\frac{1}{2},k} - \dfrac{\Delta t}{\rho}\left(\dfrac{\partial p}{\partial y}\right)^{n+1}_{i,j+\frac{1}{2},k} \\ w^{*}_{i,j,k+\frac{1}{2}} - \dfrac{\Delta t}{\rho}\left(\dfrac{\partial p}{\partial z}\right)^{n+1}_{i,j,k+\frac{1}{2}} \end{array} \right\}, \tag{29}$$

luego un esquema de resolución constaría de un plano con dimensiones $(\mathrm{BY}+1, \mathrm{BZ}+1)$, cuya carga de datos sería similar (pero reducido) al presentado en la resolución de las ecuaciones de cantidad de movimiento. Finalmente el código que procesa los gradientes y aplica las correcciones se presenta en el Código (11).

Código 11: Cálculo de gradientes de presión y corrección de velocidad.

```
pu = (gx==dimx-1) ? pdev[grid(0,gy,gz)] : pdev[grid(gx+1,gy,gz)];

//Se actualiza la velocidad con los gradientes de presión.

aux = udev[grid(gx,gy,gz)];

aux.u += -dT/rho*((pu-tile[ty][tz])/deltax);
aux.v += -dT/rho*((tile[ty+1][tz]-tile[ty][tz])/deltay);
aux.w += -dT/rho*((tile[ty][tz+1]-tile[ty][tz])/deltaz);

udev[grid(gx,gy,gz)] = aux;
```

4. RESULTADOS

4.1. La inestabilidad de Kelvin-Helmholtz

El caso de estudio propuesto se conoce como la inestabilidad de Kelvin-Helmholtz. Considere el caso particular de dos fluidos inmiscibles dentro de un recinto ubicados espacialmente según $z > 0$ y $z < 0$, con densidades y presiones ρ_1, p_1 y ρ_2, p_2 y finalmente con velocidades $U_1\vec{i}$ y $U_2\vec{i}$ respectivamente. Considerando la gravedad este problema se denomina *la inestabilidad de Kelvin-Helmholtz*, pero en aras de simplicidad, no se tendrá en cuenta la misma. Con el objetivo de estudiar la inestabilidad, se introducirá una perturbación $\zeta(x,t)$ que actúa como término de desestabilización por acción de gravedad.

Los resultados obtenidos en las simulaciones se presentan en las Figuras (4).

4.2. Cálculos de ancho de banda

Considerando que un kernel en general presenta 4 etapas, esto es, (1) la inicialización de variables, (2) la carga de datos, (3) el cómputo y (4) la escritura, entonces el cálculo de ancho de banda tendrá en cuenta las etapas 1, 2 y 4, sin tener en cuenta la escritura de los datos leídos sobre los registros (en otras palabras, no se tienen en cuenta las asignaciones en memoria compartida). De esta forma se mide el rendimiento en la carga de memoria global, y posteriormente la escritura sobre la misma. Eso es lo que se quiere comparar, en que medida el kernel actual está aprovechando el máximo rendimiento efectivo otorgado por la memoria global.

De las especificaciones de la Tesla $C1060$ se tiene que su reloj trabaja a 1600 [MHz] y su interfaz es de 512 [bits], por ende el rendimiento teórico máximo resulta 102.4 GBps. Utilizando las herramientas otorgadas por el SDK de CUDA se obtiene que el rendimiento efectivo de la memoria global resulta de 74 [GBps], y este es el número contra el cual se harán las comparaciones.

	Doble	Simple	
	8x8	8x8	16x16
	[GBps]	[GBps]	[GBps]
32x32x32	31.45 (42.50 %)	21.29 (28.77 %)	16.45 (22.29 %)
64x64x64	45.39 (60.53 %)	37.08 (50.37 %)	40.38 (54.56 %)
128x128x128	50.12 (67.72 %)	43.14 (58.29 %)	47.59 (64.31 %)

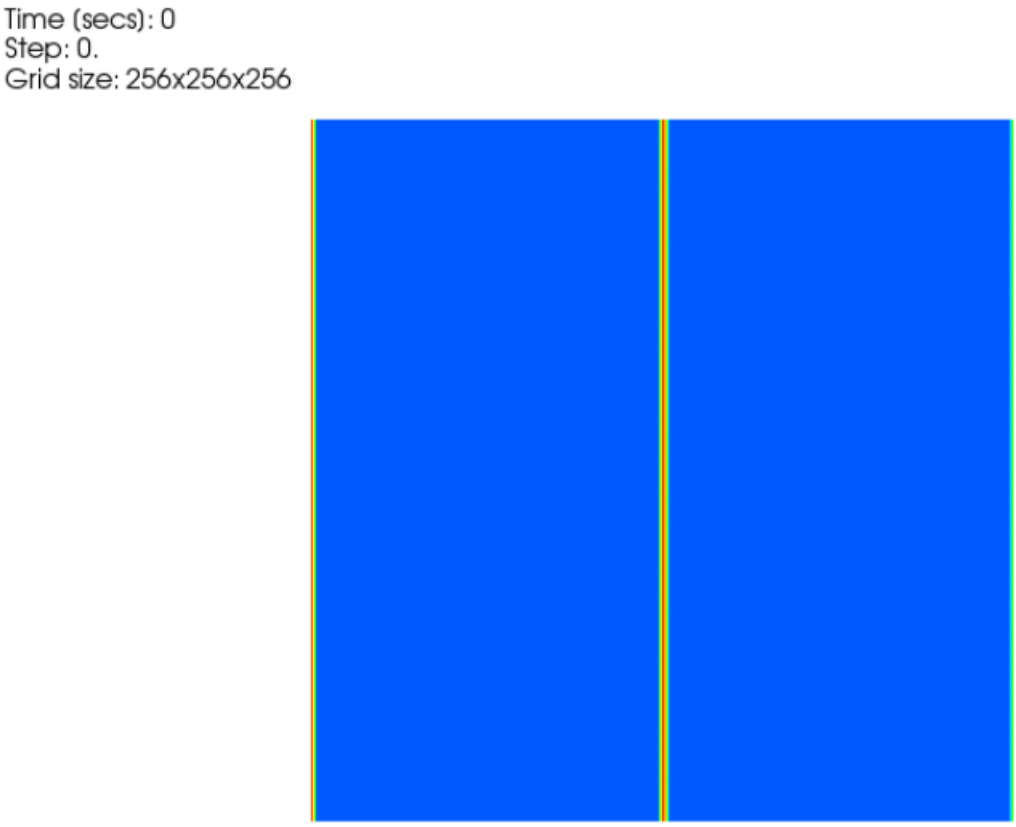
Tabla 1: Cuadro comparativo presentando los rendimientos del kernel de cantidad de movimiento. Los valores en rojo representan los GBps obtenidos, mientras que los valores entre paréntesis son los porcentajes de utilización del ancho de banda efectivo.

Continuando se tiene entonces una fórmula para el cálculo del ancho de banda de un determinado kernel como

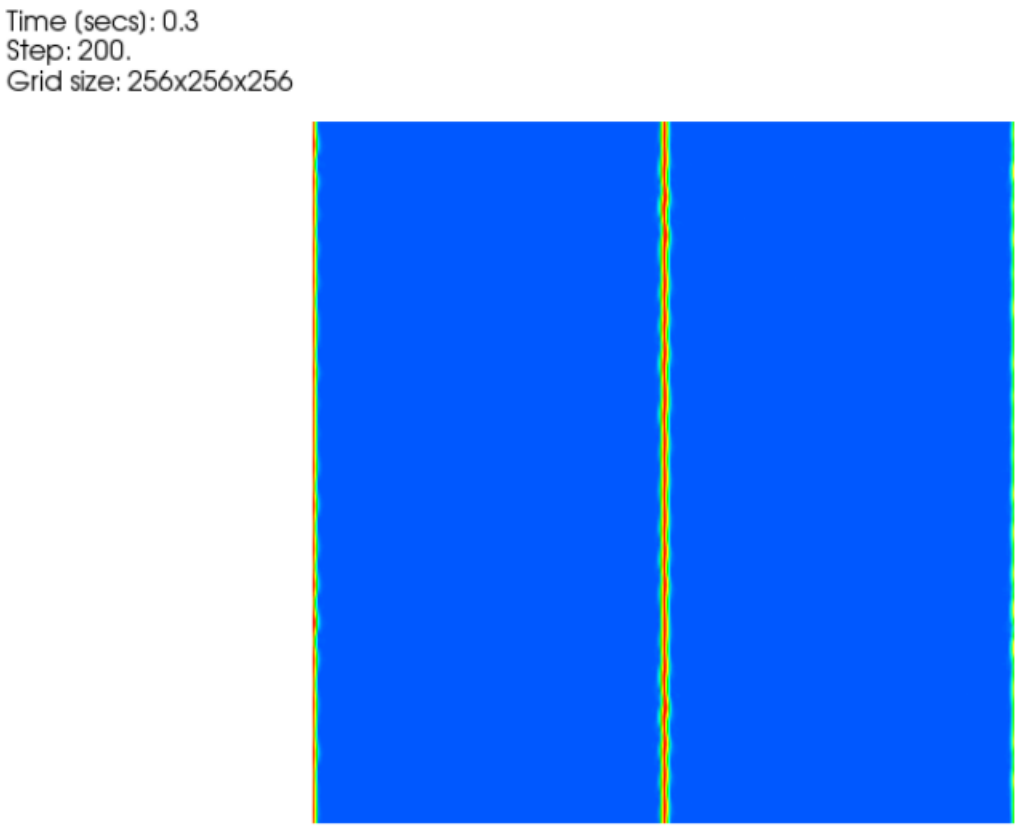
$$[(B_R + B_W) \cdot 1024^{-3}]/t_{total}, \tag{30}$$

donde t_{total} es el tiempo calculado como la resta del tiempo tomado inmediatamente tanto antes de la primer como después de la cuarta etapa y el factor $1/1024^3$ es un factor de escala a GBps.

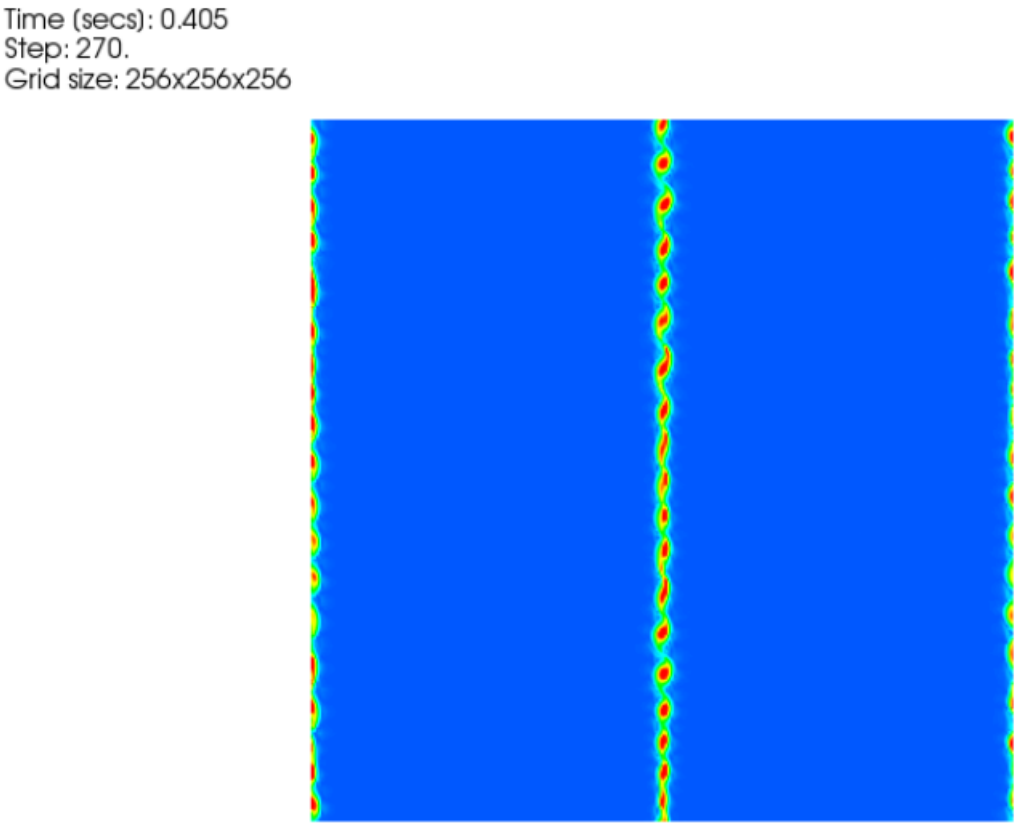
Con respecto al kernel de cantidad de movimiento se necesitan N^3 lecturas sobre la memoria global, cada una accediendo a un campo de velocidad. En 3D se tienen 3 componentes de velocidad que bien podrían ser flotantes de simple o doble precisión, de esta forma se tienen $3N^3 \cdot B$ lecturas, donde B hace referencia al alineamiento en Bytes requerido por ambos tipos



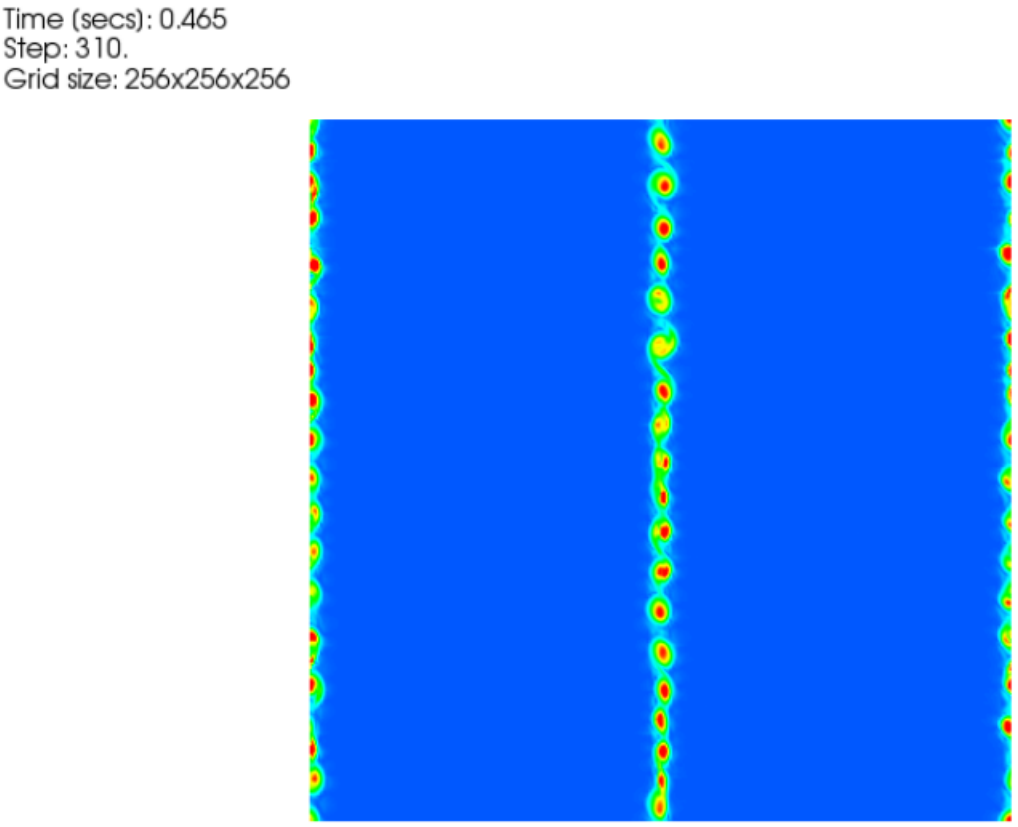
(a) Condiciones iniciales.



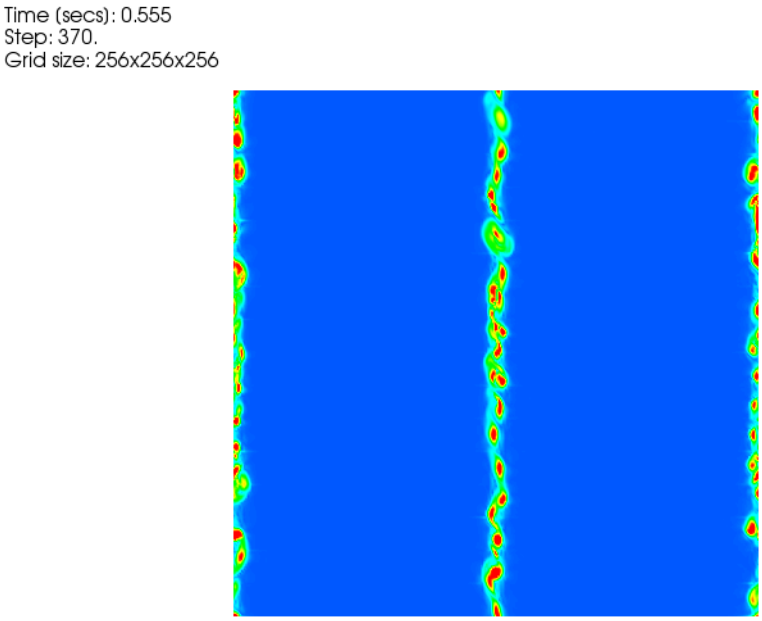
(b) Comienzan a notarse las inestabilidades.



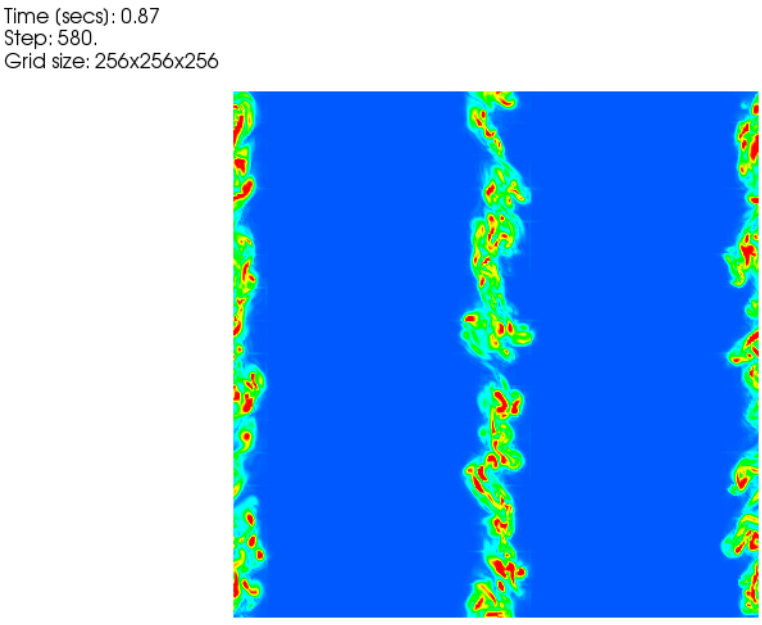
(c) La capa vorticosa va tomando forma.



(d) Ya se tienen vórtices bien definidos.



(e) La convección empieza a desprender los vórtices de la capa central.



(f) Los vórtices empiezan a desparramarse por el resto del dominio.

Figura 4: Los parámetros que se han utilizado fueron: dimensiones 256^3, $\nu = 1.004 \cdot 10^{-6}$, $\Delta t = 3.1 \cdot 10^{-3}$. En la simulación se utilizo una tangente hiperbólica para simular una capa de corte continua sobre una de las componentes del campo de velocidad, y además se introdujeron perturbaciones en todos los campos del orden de $1 \cdot 10^3$ que recreaban la inestabilidad.

de datos, 4 para simple y 8 para doble. Asimismo la cantidad de escrituras es necesariamente la misma que la de lecturas, con lo cual $B_R + B_W = 6N^3 \cdot B$. El Cuadro (1) muestra los resultados obtenidos.

El hecho de no disponer del $100\,\%$ de eficiencia reduce el rendimiento general del kernel, aunque como será visto a continuación, no resulta el punto crítico a optimizar. Esta caída de rendimiento se debe principalmente a las condiciones de borde y la forma en que los nodos del dominio de resolución se mapean a un arreglo 1D.

4.3. Comparaciones GPU vs CPU

Con el objetivo de tener una referencia del rendimiento obtenido en el desarrollo propuesto, se procederá a comparar los rendimientos contra una versión en CPU del mismo, considerando versiones unicore y multicore utilizando para ello OpenMP, un estándar de programación multithread en memoria compartida. La CPU utilizada en las pruebas es un intel i7 950 que presenta 4 núcleos y 12 MB de caché, mientras que la GPU fue una Tesla C1060 con 240 SP.

Es de interés mencionar que en las pruebas

- la optimización con OpenMP sólo es aplicada a la resolución de las ecuaciones de cantidad de movimiento, y
- para el cálculo de las FFT se utilizaron dos versiones, la FFT provista por FFTW (sin threads), y CUFFT (NVIDIA).

4.3.1. Resultados en simple precisión

Los resultados obtenidos al efectuar los cálculos en simple precisión son presentados a continuación. El Cuadro (2) presenta los rendimientos en [segs/MCels] obtenidos para un paso de tiempo completo con datos en simple precisión.

	CPU OpenMP (1 thread)	CPU OpenMP (1 thread) +FFT(GPU)	CPU OpenMP (4 threads)	CPU OpenMP (4 threads) +FFT(GPU)	GPU
	[segs/MCels]	[segs/MCels]	[segs/MCels]	[segs/MCels]	[segs/MCels]
32x32x32	0.6391	0.5627	0.5085	0.4329	0.1173
64x64x64	0.7242	0.5852	0.5887	0.4477	0.0294
128x128x128	1.2815	0.5712	1.2107	0.4389	0.0184

Tabla 2: Rendimiento obtenido por paso de tiempo. Tipo de dato simple precisión.

Donde se observa que a medida que las dimensiones del problema crecen la opción de CUDA resulta favorable. Los resultados de speedups se observan en el Cuadro (3), teniendo como referencia la versión en GPU, es decir, un factor $2x$ hace referencia a una versión de GPU dos veces más rápida que aquella en CPU.

Es necesario introducir el hecho de que el calcular la FFT en GPU y el resto en CPU requiere de traspasos de datos por el bus PCI-Express lo que, dado a su escaso ancho de banda (de alrededor de 8 GBps para la versión 2.0), limita el rendimiento general del desarrollo.

Con respecto a la toma de tiempos en las ecuaciones de cantidad de movimiento se consideraron dos etapas, la resolución y la integración temporal. Finalmente los resultados son presentados en el Cuadro (4).

	CPU OpenMP (1 thread)	CPU OpenMP (1 thread) +FFT(GPU)	CPU OpenMP (4 threads)	CPU OpenMP (4 threads) +FFT(GPU)
32x32x32	5.44x	4.79x	4.33x	3.69x
64x64x64	24.63x	19.90x	20.02x	15.22x
128x128x128	69.64x	31.04x	64.79x	23.85x

Tabla 3: Speedups obtenidos en la resolución de un paso de tiempo completo, GPU vs los distintos desarrollos en CPU. Tipo de dato simple precisión.

	CPU OpenMP (1 thread)	CPU OpenMP (4 threads)	GPU
	[segs/MCels]	[segs/MCels]	[segs/MCels]
32x32x32	0.1769	0.0592	0.0341
64x64x64	0.1905	0.0541	0.0104
128x128x128	0.1917	0.0603	0.0093

Tabla 4: Rendimiento obtenido en la resolución de las ecuaciones de cantidad de movimiento. Tipo de dato simple precisión.

El Cuadro (5) recoge los resultados en speedup, considerando como referencia los resultados obtenidos en GPU.

	CPU OpenMP (1 thread)	CPU OpenMP (4 threads)
32x32x32	5.18x	1.73x
64x64x64	18.31x	5.20x
128x128x128	20.61x	6.48x

Tabla 5: Speedups obtenidos en la resolución de las ecuaciones de cantidad de movimiento, GPU vs los distintos desarrollos en CPU. Tipo de dato simple precisión.

Para obtener el porqué de los resultados obtenidos se utiliza una herramienta provista por NVIDIA denominada *computeprof*, ella otorga un perfil completo del código CUDA que se este testeando incluyendo tiempos de ejecución, utilización de memoria compartida, registros por threads, información de accesos y escrituras fusionados (y aquellos que no logran la fusión), divergencia de threads, entre otros. De esta forma se obtiene una cantidad de 41 registros requeridos por thread y considerando que el tamaño de bloque escogido para el kernel de cantidad de movimiento fue de 16×16 (superior en rendimiento a bloques con dimensiones menores), se tiene que un sólo bloque activo puede residir por SM. En términos de warps resultan 8 con lo cual se obtiene una ocupación (cociente entre warps activos y warps disponibles, 32 para la Tesla C1060) de 0.25.

Cabe aclarar que con las dimensiones tratadas sólo un pequeña cantidad de los 30 SMs de los que se disponen son utilizados, entonces si se consideraran problemas con mayores dimensiones se podrían distribuir los threadblocks a mayor cantidad de SM con lo cual una mejora de rendimiento resulta evidente.

La principal mejora entonces consistiría en reducir la cantidad de registros por thread, una mejora en la eficiencia de las operaciones en memoria resultarían en menor performance relati-

va.

4.3.2. Resultados en doble precisión

Los resultados utilizando el tipo de dato doble precisión se muestran en los Cuadros (6 y 7), en donde los tiempos se toman teniendo en cuenta todas la operaciones requeridas por un paso de tiempo completo.

	CPU OpenMP (1 thread)	CPU OpenMP (1 thread) +FFT(GPU)	CPU OpenMP (4 threads)	CPU OpenMP (4 threads) +FFT(GPU)	GPU
	[segs/MCels]	[segs/MCels]	[segs/MCels]	[segs/MCels]	[segs/MCels]
32x32x32	0.6464	0.5939	0.5252	0.4645	0.2310
64x64x64	0.7894	0.6411	0.6594	0.5109	0.0901
128x128x128	1.5012	0.6384	1.3437	0.5086	0.0602

Tabla 6: Rendimiento obtenido por paso de tiempo. Tipo de dato, doble precisión.

	CPU OpenMP (1 thread)	CPU OpenMP (1 thread) +FFT(GPU)	CPU OpenMP (4 threads)	CPU OpenMP (4 threads) +FFT(GPU)
32x32x32	2.79x	2.57x	2.27x	2.01x
64x64x64	8.76x	7.11x	7.31x	5.67x
128x128x128	24.93x	10.60x	22.32x	8.44x

Tabla 7: Speedups obtenidos en la resolución de un paso de tiempo completo, GPU vs los distintos desarrollos en CPU. Tipo de dato doble precisión.

Los Cuadros (8) y (9) muestran los resultados obtenidos en la resolución de las ecuaciones de cantidad de movimiento.

	CPU OpenMP (1 thread)	CPU OpenMP (4 threads)	GPU
	[segs/MCels]	[segs/MCels]	[segs/MCels]
32x32x32	0.1854	0.0598	0.1071
64x64x64	0.1921	0.0673	0.0602
128x128x128	0.1930	0.0739	0.0435

Tabla 8: Rendimiento obtenido en la resolución de las ecuaciones de cantidad de movimiento. Tipo de dato, doble precisión.

El porqué de los resultados similares para la resolución de las ecuaciones de cantidad de movimiento en las implementaciones GPU y CPU multithread está en que, la GPU como dispositivo de cálculo masivo de propósito general, no tiene resultados satisfactorios cuando opera en doble precisión (en Fermi se tiene una notable mejora en performance para datos estos tipos de datos). Esto es pues la arquitectura (al menos la Tesla y anteriores) está ideada para cómputos en simple precisión, luego tiene más unidades de simple precisión que de doble. La CPU en cambio, que tiene un grado de evolución para los cálculos de propósito general superior a la

	CPU OpenMP (1 thread)	CPU OpenMP (4 threads)
32x32x32	1.73x	0.55x
64x64x64	3.19x	1.11x
128x128x128	4.43x	1.69x

Tabla 9: Speedups obtenidos en la resolución de las ecuaciones de cantidad de movimiento, GPU vs los distintos desarrollos en CPU. Tipo de dato, doble precisión.

GPU, presenta más cantidad de registros de 64 bits, es por ello que la utilización de datos en simple o doble precisión en general no tienen gran diferencia de rendimiento.

El principal problema del kernel en GPU para este caso particular no resulta ni la utilización de memoria compartida, ni su eficiencia de acceso, sino la cantidad de registros por thread requeridos. Utilizando *computeprof* se tiene que cada thread requiere de 88 registros. Considerando threadblocks de 8×8 (cuyo rendimiento resultó superior a threadblocks con dimensiones de 2×2 y 4×4, y además que los bloques de 16×16 no son válidos pues exceden los recursos disponibles) luego necesita 5632 registros, con lo cual sólo dos bloques por SM pueden ser desglosados por la unidad SIMT (pues con 3 bloques se supera la cantidad de registros asociados a SM que son 16384). Luego en términos de warps activos como se tienen threadblocks de 8×8, sólo se disponen de dos warps, considerando entonces que se tienen dos bloques por SM, la cantidad de warps activos por SM resultan 4, de esta forma la ocupación es de 0.125. Un factor bajo, que únicamente podría ser incrementado con la adición de más registros asociados a un SM.

El problema de disponer de pocos warps activos ya fue introducido anteriormente, pero básicamente se puede decir que la unidad SIMT no tiene los recursos suficientes como para mantener a los SPs activos cuando operaciones de gran latencia (como ser la gran cantidad de operaciones de lecturas requeridas para la confección de stencils) se llevan a cabo.

Se puede decir entonces que una mejor utilización del ancho de banda efectivo no resultaría en un incremento notable puesto que el kernel de cantidad de movimiento se estaría ejecutando sobre la misma cantidad de SPs. Una verdadera mejora consistiría en reducir la cantidad de registros por thread (del lado del programador, o desde el planteo matemático), o bien mediante la introducción de más registros en las GPUs (del lado del proveedor).

4.4. Consideraciones de los esquemas temporales

Como bien es introducido en (Molemaker et al., 2008) los esquemas de integración temporal adoptados para el primer paso de F-S resultan incondicionalmente inestable para FE mientras que condicionalmente estable para AB2 (Chen y Falconer, 1992; Costarelli et al., 2011), siempre teniendo en cuenta el esquema de estabilización QUICK para el término convectivo de las ecuaciones de cantidad de movimiento.

5. CONCLUSIONES

Se puede concluir, en base a los resultados obtenidos, que la tecnología CUDA

- para problemas con dimensiones relativamente pequeñas y utilizando datos en doble precisión, no otorga mejora alguna con respecto a un desarrollo en CPU utilizando múltiples threads. Cuando las dimensiones comienzan a crecer la opción en CUDA resulta favorable dado por su escalabilidad natural;

- con respecto a los desarrollos en simple precisión, presenta mejoras más pronunciadas superando en general a aquellos resultados obtenidos por una versión con múltiples threads;
- presenta el problema fundamental de tener que adaptar el problema a la arquitectura (esto es con el objetivo de utilizar eficientemente la bondades de la misma), luego requiere de un gran esfuerzo para el logro de tareas similares en CPU (a excepción de operaciones triviales);
- dada por la escasez de registros por thread, posee una baja ocupación (en términos de warps activos por SM) que está directamente relacionado con operaciones de gran latencia, limitando así su capacidad de dispositivo de cálculo masivo. Es por ello que la confección de stencils resulta más costosa que la posterior etapa de cálculos.

Finalmente

- considerando propuestas de esquemas sencillos como ser diferencias finitas en grillas estructuradas, cuya formulación dada por la localidad de los datos y el trivial acceso entre vecinos contiguos (al contrario que con esquemas no estructurados, clásicos de elementos finitos) debiera tener la posibilidad de explotar al máximo el procesamiento masivo de una GPU, se ha visto sin embargo que la escasez de registros impide ese resultado y pone en duda la utilización de esta arquitectura para la mecánica de fluidos computacional;
- extrapolando a esquemas más complejos (como podría ser la utilización de elementos finitos con datos en doble precisión), no se esperan resultados satisfactorios (en comparación a los obtenidos por un CPU), al menos con la arquitectura Tesla, aunque se espera que en un futuro esta tecnología madure y pueda sobrepasar aquellas dificultades expuestas a lo largo de este trabajo.

6. AGRADECIMIENTOS

Este trabajo ha recibido soporte financiero de

- Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT, Argentina, grants PICT-1141/2007, PICT-0270/2008),
- Universidad Nacional del Litoral (UNL, Argentina, grants CAI+D 2009-65/334, CAI+D-2009-III-4-2) y
- European Research Council (ERC) Advanced Grant, Real Time Computational Mechanics Techniques for Multi-Fluid Problems (REALTIME, Reference: ERC-2009-AdG, Dir: Dr. Sergio Idelsohn).

Además se han utilizado para el desarrollo herramientas de Free Software como GNU/Linux OS, compiladores GCC/G++, Octave, y *Open Source* como VTK entre muchos otros.

REFERENCIAS

Chen Y. y Falconer R.A. Advection-diffusion modelling using the modified quick scheme. *International journal for numerical methods int fluids*, 1992.

Costarelli S.D., Paz R.R., y Dalcin L.D. Resolución de las ecuaciones de navier-stokes utilizando cuda. *Tesina de grado en Ingeniería Informática - Facultad de Ingeniería y Ciencias Hídricas, Universidad Nacional del Litoral*, 2011.

Hirsch C. *Numerical computation of internal & external flows : Volume I*. Elsevier, 2th edición, 2007. ISBN 978-0-7506-6594-0.

Kirk D.B. y mei W. Hwu W. *Programming massively parallel processors: A hands-on approach*. Elsevier, 2010. ISBN 978-0-12-381472-2.

Leonard B.P. A stable and accurate convective modelling procedure based on quadratic upstream interpolation. *Computer Methods in Applied Mechanics and Engineering*, 1979.

Loan C.V. *Computational frameworks for the fast Fourier transform*. Siam, 1992. ISBN 0-89871-285-8.

Molemaker J., Cohen J.M., Patel S., y Noh J. Low voscosity flow simulations for animation. *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, 2008.

Sanders J. y Kandrot E. *CUDA by example: An introduction to general-purpose GPU programming*. Addison-Wesley, 2010. ISBN 978-0-13-138768-3.

Storti M.A., Paz R.R., Dalcin L.D., y Costarelli S.D. A fft preconditioning technique for the solution of incompressible flow with fractional step methods on graphic processing units. *Mecánica Computacional*, 2010.