Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXXII, págs. 1081-1101 (artículo completo)
Carlos G. García Garino, Aníbal E. Mirasso, Mario A. Storti, Miguel E. Tornello (Eds.)
Mendoza, Argentina, 19-22 Noviembre 2013

# RESOLUCIÓN DE LAS ECUACIONES DE AGUAS POCO PROFUNDAS SOBRE MALLAS NO ESTRUCTURADAS EN *GPU*

**Nicolás D. Badano**[a,b]

[a]*Programa de Hidráulica Computacional, Laboratorio de Hidráulica Aplicada, Instituto Nacional del Agua, Ezeiza, Buenos Aires, Argentina, nicolas.d.badano@gmail.com, http://www.ina.gob.ar/*

[b]*Laboratorio de Modelación Matemática, Facultad de Ingeniería, Universidad de Buenos Aires, Buenos Aires, Argentina, http://laboratorios.fi.uba.ar/lmm/*

**Palabras Clave:** Shallow Water Equations, Volúmenes Finitos, GPU, CUDA.

**Resumen.** Las ecuaciones bidimensionales de *Aguas Poco Profundas* o *Shallow Water Equations* (SWE) describen el comportamiento de un fluido incompresible a superficie libre dentro de un dominio en el cual la profundidad es considerada reducida respecto de las dimensiones horizontales del problema. Muchos de los problemas de la hidrodinámica fluvial y costera, así como la simulación de rotura de presas, pueden abordarse mediante la resolución de estas ecuaciones. En este trabajo se describe la implementación un modelo para la resolución de las SWE mediante el *Método de los Volúmenes Finitos* paralelizado para *Unidades de Procesamiento Gráfico* (GPU) mediante CUDA. Se calculan los flujos mediante esquemas tipo Roe sobre mallas no estructuradas de triángulos, de manera que estos solo dependen de los valores en las celdas contiguas y se utiliza un esquemas temporal explícito. De esta manera se genera un esquema de cálculo eficiente y de buena escalabilidad. Se alcanzan valores de speed-up de 39 en doble precisión para mallas de cálculo del orden de las decenas de miles de elementos.

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

## 1. INTRODUCCIÓN

Muchos de los problemas de la ingeniería hidráulica civil, ambiental, fluvial y costera pueden analizarse mediante la resolución de las ecuaciones bidimensionales de *Aguas Poco Profundas* o *Shallow Water Equations* (SWEs). Estos problemas se caracterizan por tener dominios irregulares y muy extensos, que se extienden usualmente más allá de la zona de estricto interés para mitigar la influencia de las condiciones de borde, que resultan generalmente aproximadas. Por ambas razones, la utilización de mallas no estructuradas resulta particularmente apropiada, ya que tanto los contornos como el tamaño de elementos pueden adaptarse para resolver con buena precisión la zona de interés, permitiendo completar el resto del dominio con elementos de mayor tamaño.

Muchos trabajos se han publicado en la bibliografía sobre este problema, la mayor parte de los cuales ha adoptado la utilización de esquemas de volúmenes finitos, calculando los flujos como si fueran problemas de Riemman en la dirección normal a la cara, usualmente resueltos de manera aproximada con esquemas de tipo Roe. Esto permite capturar discontinuidades en el flujo, y resolver perfiles con transiciones abruptas. El primero de estos trabajos es debido a Alcrudo y García-Navarro (1993), que utilizaron un esquema de segundo orden basado en una reconstrucción de las variables de tipo MUSCL (van Leer, 1979) sobre una malla estructurada de cuadriláteros. Anastasiou y Chan (1997) extendieron esta metodología a mallas triangulares, al igual que Hubbard (1999), que propuso el uso de limitadores multidimensionales basados en la definición de "regiones de principio del máximo" en cada celda. El tratamiento del mojado y secado de fracciones del dominio y su efecto sobre el trasporte de escalares fueron estudiados por Begnudelli y Sanders (2006). Fe et al. (2009) extendieron el tratamiento de las SWEs a modelos de turbulencia de tipo k-$\epsilon$, a fin de calcular físicamente la dispersión de la cantidad de movimiento.

Si bien la naturaleza bidimensional de las SWEs da lugar a mallas de cálculo mucho más pequeñas que las que surgen en problemas tridimensionales, el rendimiento de los modelos de resolución sigue siendo crítico en muchas aplicaciones. Esto se debe a que los períodos de simulación pueden ser largos, incluso de años o décadas, como por ejemplo en casos asociados al estudio estadístico de crecidas en problemas costeros (Re, 2005; Lecertua, 2010) o a problemas geomorfológicos (Badano et al., 2012).

En este contexto resulta tentador explorar la paralelización de las SWEs mediante Unidades de Procesamiento Gráfico (GPU). En los últimos años, debido a la gran demanda de gráficos 3D de calidad creciente para la industria del entretenimiento, los dispositivos GPU han desarrollado una enorme capacidad de cómputo a través de un paradigma de elevado paralelismo usando multiprocesadores de gran cantidad de núcleos y un muy alto ancho de banda de memoria. Las GPU son particularmente apropiadas para problemas que pueden expresarse de manera que exista paralelismo de datos, es decir, cuando la misma operación (idealmente de gran carga aritmética respecto de la cantidad de accesos a la memoria) puede llevarse a cabo sobre multiples elementos de manera simultánea (NVIDIA, 2013).

En este trabajo se desarrolló el modelo utilizando Compute Unified Device Architecture (CUDA), una extensión del lenguaje de programación C desarrollada por NVIDIA para sus dispositivos (Nickolls et al., 2008). Si bien el hardware del dispositivo está diseñado originalmente para cálculo gráfico, lidiando en términos de vértices, polígonos, píxeles y texturas, CUDA expone una arquitectura abstracta que permite diseñar los algoritmos de cálculo directamente, con un lenguaje semejante a un programa en CPU. Esto incrementó el uso de GPUs para procesar problemas generales, una tendencia conocida como Computación de Propósito General en

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

Unidades de Procesamiento Gráfico o General-purpose computing on graphics processing units (GPGPU).

En su aplicación a la dinámica computacional, destacan los trabajos de Shinn y Vanka (2009), que implementaron un modelo hidrodinámico tridimensional en CUDA. Senocak et al. (2009), Thibaultl y Senocak (2009) utilizaron multiples GPU para el cálculo de dispersión de contaminantes urbanos. Menenguci et al. (2010) presentaron un modelo en diferencias finitas para la ecuación de Navier-Stokes usando un algoritmo Red-Black. Costarelli et al. (2012) aprovecharon la capacidad de cálculo en GPU para desarrollaron un solver aproximado para dichas ecuaciones para su uso en tiempo real.

En este trabajo se describe la aplicación de CUDA para el desarrollo de un modelo bidimensional basado en las SWEs.

## 2. MODELO NUMÉRICO

### 2.1. Ecuaciones básicas en forma conservativa

Las ecuaciones bidimensionales de *Aguas Poco Profundas* en forma vectorial puede escribirse como (Anastasiou y Chan, 1997):

$$\frac{\partial}{\partial t}\iint_{\Omega} \boldsymbol{Q}\, d\Omega + \oint_{S} \boldsymbol{F}\cdot\boldsymbol{n}\, dS = \iint_{\Omega} \boldsymbol{H}\, d\Omega \tag{1}$$

$$\boldsymbol{Q} = \begin{bmatrix} h & hu & hv \end{bmatrix}^{T}, \qquad \boldsymbol{F} = \boldsymbol{F}^{I} + \boldsymbol{F}^{V} \tag{2}$$

$$\boldsymbol{F}^{I} = \begin{bmatrix} hu & hv \\ hu^2 + \frac{1}{2}gh^2 & huv \\ huv & hv^2 + \frac{1}{2}gh^2 \end{bmatrix}, \qquad \boldsymbol{F}^{V} = -\nu_t h \begin{bmatrix} 0 & 0 \\ \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{bmatrix}, \tag{3}$$

$$\boldsymbol{H} = \begin{bmatrix} 0 \\ -gh\left(\boldsymbol{S_{f}}_x + \boldsymbol{S_0}_x\right) + hC_f v \\ -gh\left(\boldsymbol{S_{f}}_y + \boldsymbol{S_0}_y\right) - hC_f u \end{bmatrix} \tag{4}$$

donde $\boldsymbol{Q}$ es el vector de variables independientes en forma conservativa: $h$ el tirante y $hu$ y $hv$ los caudales específicos en dirección $x$ e $y$ respectivamente; $\boldsymbol{F}$ es la matriz de flujos, suma de los flujos invíscidos $\boldsymbol{F}^{I}$ y viscosos $\boldsymbol{F}^{V}$; $\Omega$ el dominio de cálculo y $S$ su frontera; $\nu_t$ la viscosidad de torbellino; $\boldsymbol{S_0}$ y $\boldsymbol{S_f}$ son las pendientes de fondo y de fricción respectivamente y $C_f$ es el coeficiente de Coriolis.

Si bien los términos fuentes se implementaron en el modelo, estos no se utilizan en este trabajo, ya que los ensayos se realizan en condiciones de fondo horizontal y deslizante y no se consideran fuerzas de Coriolis.

### 2.2. Discretización espacial

El dominio de simulación se discretiza en una malla no estructurada de elementos, en este caso triangulares. La ecuación de conservación (1) puede escribirse para cualquier volumen de control $i$ como:

$$\frac{\partial}{\partial t}\iint_{\Omega_i} \boldsymbol{Q}\, d\Omega = -\oint_{\partial C_i} \boldsymbol{F}\cdot\boldsymbol{n}\, dS + \iint_{\Omega_i} \boldsymbol{H} d\Omega \tag{5}$$

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

Realizando una discretización espacial en volúmenes finitos con un tratamiento en centro de celda de las variables independientes y términos fuentes, la ecuación anterior resulta:

$$\frac{\partial \boldsymbol{Q}_i}{\partial t} = -\frac{1}{V_i}\left(\sum_{j=k(i)} \boldsymbol{F}_{i,j}\,\Delta l_{i,j}\right) + \boldsymbol{H}_i, \tag{6}$$

donde $\boldsymbol{Q}_i$ es el vector variables independientes centro de celda de area $V_i$, $\boldsymbol{F}_{i,j}$ es el flujo normal a la cara interface entre las celdas $i$ y $j$, cuya longitud es $\Delta l_{i,j}$, $k(i)$ es el conjunto de las celdas vecinas a $i$ y $\boldsymbol{H}_i$ es el término fuente en el centro de la celda.

### 2.3. Cálculo de los flujos invíscidos

Para calcular los flujos invíscidos $\boldsymbol{F}^I_{i,j}$ se adopta en cada arista la función de flujo de Roe (1981), que modela el flujo como la solución aproximada de un problema de Riemann unidimensional en la dirección normal a la arista:

$$\boldsymbol{F}^I_{i,j}\cdot\boldsymbol{n} = \frac{1}{2}\left(\boldsymbol{F}^I(\boldsymbol{Q}^+_{i,j})\cdot\boldsymbol{n} + \boldsymbol{F}^I(\boldsymbol{Q}^-_{i,j})\cdot\boldsymbol{n} - \left|\boldsymbol{A}(\tilde{\boldsymbol{Q}})\right|\left(\boldsymbol{Q}^+_{i,j} - \boldsymbol{Q}^-_{i,j}\right)\right) \tag{7}$$

donde $\boldsymbol{Q}^+_{i,j}$ y $\boldsymbol{Q}^-_{i,j}$ son los estados a ambos lados de la cara y $\boldsymbol{A}(\tilde{\boldsymbol{Q}})$ es la matriz Jacobiana de flujo $\boldsymbol{A}$ evaluada en un estado medio de Roe $\tilde{\boldsymbol{Q}}$. La matriz Jacobiana de flujo $\boldsymbol{A}$ es (Anastasiou y Chan, 1997):

$$\boldsymbol{A} = \frac{\partial \boldsymbol{F}\cdot\boldsymbol{n}}{\partial \boldsymbol{Q}} = \begin{bmatrix} 0 & n_x & n_y \\ (c^2-u^2)\,n_x - uvn_y & 2un_x + vn_y & un_y \\ (c^2-v^2)\,n_y - uvn_x & vn_x & un_x + 2vn_y \end{bmatrix}, \tag{8}$$

donde $c$ es la celeridad de la onda superficial $(gh)^{1/2}$ y $\boldsymbol{n} = \begin{bmatrix} n_x & n_y \end{bmatrix}^T$ es la normal de la cara.

En este trabajo se adoptó como estado medio el propuesto por Alcrudo y García-Navarro (1993), ampliamente utilizado en la bibliografía (Bradford y Katopodes, 1999; Bradford y Sanders, 2002; Simões, 2011):

$$\tilde{\boldsymbol{Q}} = \begin{bmatrix} \tilde{h} & \tilde{h}\tilde{u} & \tilde{h}\tilde{v} \end{bmatrix}^T, \tag{9}$$

$$\tilde{u} = \frac{\sqrt{h^+}u^+ + \sqrt{h^-}u^-}{\sqrt{h^+} + \sqrt{h^-}} \qquad \tilde{v} = \frac{\sqrt{h^+}v^+ + \sqrt{h^-}v^-}{\sqrt{h^+} + \sqrt{h^-}} \tag{10}$$

$$\tilde{h} = \sqrt{h^+h^-} \qquad \tilde{c} = \sqrt{g\frac{h^+ + h^-}{2}} \tag{11}$$

La matriz Jacobiana de flujo evaluada en el estado medio $\boldsymbol{A}(\tilde{\boldsymbol{Q}})$ puede descomponerse como:

$$\boldsymbol{A}(\tilde{\boldsymbol{Q}}) = \boldsymbol{R}\,\boldsymbol{\Lambda}\,\boldsymbol{L} \tag{12}$$

donde $\boldsymbol{R}$, $\boldsymbol{L}$ y $\boldsymbol{\Lambda}$ son los autovectores derechos e izquierdos y los autovalores de $\boldsymbol{A}(\tilde{\boldsymbol{Q}})$ respectivamente. Siguiendo la descomposición de Anastasiou y Chan (1997), los autovalores de $\boldsymbol{A}(\tilde{\boldsymbol{Q}})$ son:

$$\lambda_1 = \tilde{u}n_x + \tilde{v}n_y \qquad \lambda_2 = \tilde{u}n_x + \tilde{v}n_y - \tilde{c} \qquad \lambda_3 = \tilde{u}n_x + \tilde{v}n_y + \tilde{c}, \tag{13}$$

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

mientras que las matrices de autovectores son

$$\boldsymbol{R} = \begin{bmatrix} 0 & 1 & 1 \\ n_y & \tilde{u} - \tilde{c}n_x & \tilde{u} + \tilde{c}n_x \\ -n_x & \tilde{v} - \tilde{c}n_y & \tilde{v} + \tilde{c}n_y \end{bmatrix}, \tag{14}$$

$$\boldsymbol{L} = \begin{bmatrix} -(\tilde{u}n_y - \tilde{v}n_x) & n_y & -n_x \\ (\tilde{u}n_x + \tilde{v}n_y)/2\tilde{c} + \frac{1}{2} & -n_x/2\tilde{c} & -n_y/2\tilde{c} \\ -(\tilde{u}n_x + \tilde{v}n_y)/2\tilde{c} + \frac{1}{2} & n_x/2\tilde{c} & n_y/2\tilde{c} \end{bmatrix} \tag{15}$$

El módulo de $\boldsymbol{A}(\tilde{\boldsymbol{Q}})$ resulta simplemente de utilizar el valor absoluto de los autovalores en la matriz $\boldsymbol{\Lambda}$:

$$\left|\boldsymbol{A}(\tilde{\boldsymbol{Q}})\right| = \boldsymbol{R}\,|\boldsymbol{\Lambda}|\,\boldsymbol{L} \tag{16}$$

A fin de evitar que el módulo de los autovalores puedan anularse, lo que anularía el término estabilizante de la ecuación (7), se aplica una corrección de entropía (Harten y Hyman, 1983; Fe, 2005)

$$\left|\tilde{\lambda}_i\right| = \begin{cases} |\lambda_i| & \text{if } |\lambda_i| \geq \epsilon \\ \frac{1}{2}\left(\frac{|\lambda_i|^2}{\epsilon} + \epsilon\right) & \text{if } |\lambda_i| < \epsilon \end{cases} \tag{17}$$

### 2.4. Reconstrucción de las variables

A fin de obtener una discretización espacial de segundo orden se aplica un esquema de reconstrucción limitado tipo MUSCL (van Leer, 1979). En primer lugar la distribución de las variables dentro de cada celda se reconstruyen a partir de los resultados del paso anterior asumiendo una variación lineal:

$$\boldsymbol{Q}(x,y) = \boldsymbol{Q}_i + (\nabla\boldsymbol{Q})_i \cdot \boldsymbol{r}, \tag{18}$$

donde $\boldsymbol{Q}_i$ es el valor en el centro de celda, $(\nabla\boldsymbol{Q})_i$ es el gradiente ilimitado en el centro de celda y $\boldsymbol{r}$ es el vector que va del centro de celda a cualquier posición $(x,y)$ dentro de la celda. El gradiente en el centro de la celda para cada variable escalar se calcula a partir del valor en el centro de las tres celdas vecinas, como si fuera la pendiente de un plano apoyado en esos puntos (Begnudelli y Sanders, 2006).

Para cualquiera de las variables independientes $q$ que forman $\boldsymbol{Q}$, la diferencia entre el valor reconstruido en el centro de la cara $f$ de la celda $i$, $q_{if}$, y el valor en el centro de celda $q_i$ puede escribirse como:

$$\Delta q_{if} = q_{if} - q_i = (\nabla q)_i \cdot \boldsymbol{r}_{if}, \tag{19}$$

donde $\boldsymbol{r}_{if}$ es el vector que va del centro de celda al centro de cara.

A fin de evitar la aparición de nuevos extremos locales durante el proceso de reconstrucción, lo que introduce oscilaciones espúreas, se limita el gradiente reconstruido:

$$(\tilde{\nabla} q)_i = \phi_{qi}\,(\nabla q)_i \tag{20}$$

donde $\phi_{qi}$ es el parámetro llamado limitador para la celda $i$. En este trabajo se limitó el gradiente de manera geométrica, aplicando un limitador minmod a los valores reconstruidos en el centro

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

de las caras (Barth y Jespersen, 1989; Anastasiou y Chan, 1997), lo que se ha demostrado alcanza para garantizar el cumplimiento del principio del máximo local (Batten et al., 1996). El valor del limitador para una variable $q$ en la celda $i$ es el mínimo que resulta limitar los valores cada una de las tres caras $f$ (Hubbard, 1999):

$$\phi_{qi} = \min_{f=1...3} (\phi_{qif}) \tag{21}$$

$$\phi_{if} = \begin{cases} \frac{\max(q_v - q_i, 0)}{\Delta q_{if}} & \text{si } \Delta q_{if} > \max(q_v - q_i, 0) \\ \frac{\min(q_v - q_i, 0)}{\Delta q_{if}} & \text{si } \Delta q_{if} < \min(q_v - q_i, 0) \\ 1 & \text{sino} \end{cases}, \tag{22}$$

donde $q_i$ es el valor en el centro de la celda $i$ y $q_v$ en el centro de la celda vecina.

La diferencia entre el valor en el centro de la celda y el reconstruido en el centro de la cara $f$ se obtiene utilizando el gradiente limitado:

$$\tilde{\Delta q}_{if} = (\tilde{\nabla q})_i \cdot \boldsymbol{r}_{if}, \tag{23}$$

Estas variaciones reconstruidas a ambos lados de una cara se suman a los valores en los centros de las celdas correspondientes para el cálculo de los flujos en las ecuaciones (7) y subsiguientes. Los flujos $\boldsymbol{F}_{i,j}$ en la cara $j$ entre las celdas $i$ y $j$ pueden expresarse en función de los valores reconstruidos a cada lado como

$$\boldsymbol{F}_{i,j} \cdot \boldsymbol{n} = \boldsymbol{F}(\boldsymbol{Q}_i + \boldsymbol{\Delta}\tilde{\boldsymbol{Q}}_{if}, \boldsymbol{Q}_j + \boldsymbol{\Delta}\tilde{\boldsymbol{Q}}_{jf}) \tag{24}$$

### 2.5. Cálculo de los flujos viscosos

Para calcular los flujos viscosos se adoptó un esquema sencillo de segundo orden. El flujo resulta:

$$\boldsymbol{F}^V_{i,j} \cdot \boldsymbol{n} = -\nu_t \left( \frac{\boldsymbol{p}_k - \boldsymbol{p}_i}{|\boldsymbol{p}_k - \boldsymbol{p}_i|^2} \cdot \boldsymbol{n} \right) \frac{h_i + h_k}{2} \begin{bmatrix} 0 \\ u_k - u_i \\ v_k - v_i \end{bmatrix}, \tag{25}$$

donde $k$ es el número de celda vecina a la cara $j$ de la celda $i$ y $\boldsymbol{p}_i$ y $\boldsymbol{p}_k$ son las coordenadas de centro de celda. El término escalar entre paréntesis, que solo depende de factores geométricos, se precalcula al principio del programa.

### 2.6. Cálculo de los flujos en las condiciones de borde

Los flujos viscosos se consideran nulos en las caras que pertenecen a condiciones de borde. Para calcular los flujos invíscidos se utilizan celdas fantasma para representar las condiciones al otro lado del borde.

Para bordes sólidos deslizantes, el estado de la celda fantasma $R$ en función del estado de la celda interior $L$, es el siguiente (Anastasiou y Chan, 1997):

$$h_R = h_L \qquad \boldsymbol{U}_{nR} = 0 \qquad \boldsymbol{U}_{tR} = \boldsymbol{U}_{tL} \tag{26}$$

donde $\boldsymbol{U}_n$ y $\boldsymbol{U}_t$ son los componentes normales y trasversales a la cara de $\boldsymbol{U}$ respectivamente. Para otras condiciones de borde, donde se pretende imponer los parámetros de subíndice $B$, los estados de la celda fantasma son:

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

- Flujo entrante subcrítico

$$h_R = h_B \qquad \boldsymbol{U}_{nR} = \boldsymbol{U}_{nL} + \sqrt{g}\left(\sqrt{h_L} - \sqrt{h_R}\right) \qquad \boldsymbol{U}_{tR} = 0 \tag{27}$$

- Flujo entrante supercrítico

$$h_R = h_B \qquad \boldsymbol{U}_{nR} = \boldsymbol{U}_{nB} \qquad \boldsymbol{U}_{tR} = 0 \tag{28}$$

- Flujo saliente subcrítico

$$h_R = h_B \qquad \boldsymbol{U}_{nR} = \boldsymbol{U}_{nL} + \sqrt{g}\left(\sqrt{h_L} - \sqrt{h_R}\right) \qquad \boldsymbol{U}_{tR} = \boldsymbol{U}_{tL} \tag{29}$$

- Flujo saliente supercrítico

$$h_R = h_L \qquad \boldsymbol{U}_{nR} = \boldsymbol{U}_{nL} \qquad \boldsymbol{U}_{tR} = \boldsymbol{U}_{tL} \tag{30}$$

Para imponer una condición de borde de nivel, se determina el sentido del flujo, entrante o saliente, a partir de la velocidad en la celda interior y el número de Froude local. En función de esos parámetros pueden obtenerse el estado de la celda fantasma utilizando las ecuaciones (27) a (30).

### 2.7. Discretización temporal

Se utilizó un esquema explícito de segundo orden propuesto por van Leer (1984) y utilizado por Hubbard (1999). La ecuación (6) se discretiza en dos semi-pasos:

$$\boldsymbol{Q}^* = \boldsymbol{Q}^n - \frac{\Delta t}{2V_i}\left(\sum_{j=k(i)} \boldsymbol{F}(\boldsymbol{Q}_i^n + \boldsymbol{\Delta}\tilde{\boldsymbol{Q}}_{if}^n,\, \boldsymbol{Q}_j^n + \boldsymbol{\Delta}\tilde{\boldsymbol{Q}}_{jf}^n)\,\Delta l_{i,j}\right) + \boldsymbol{H}_i^n \tag{31}$$

$$\boldsymbol{Q}^{n+1} = \boldsymbol{Q}^n - \frac{\Delta t}{V_i}\left(\sum_{j=k(i)} \boldsymbol{F}(\boldsymbol{Q}_i^* + \boldsymbol{\Delta}\tilde{\boldsymbol{Q}}_{if}^n,\, \boldsymbol{Q}_j^* + \boldsymbol{\Delta}\tilde{\boldsymbol{Q}}_{jf}^n)\,\Delta l_{i,j}\right) + \boldsymbol{H}_i^*, \tag{32}$$

donde $\boldsymbol{Q}^*$ es un estado intermedio de la solución, y la reconstrucción limitada $\boldsymbol{\Delta}\tilde{\boldsymbol{Q}}_{if}^n, \boldsymbol{\Delta}\tilde{\boldsymbol{Q}}_{jf}^n$ realizada para el paso de tiempo $n$ se utiliza para ambos semi-pasos, a fin de reducir el costo computacional.

## 3. IMPLEMENTACION DEL MODELO

### 3.1. Arquitectura CUDA

En CUDA, el hardware se divide en *host* (que es el CPU) y uno o más dispositivos GPU conocidos como *devices*, capaces de ejecutar simultaneamente un gran número de hilos o *threads*. La estructura de un dispositivo GPU desde el punto de vista de CUDA se presenta en la figura 1.

Desde el programa que corre en el host se ejecutan los *kernel*, funciones especiales que crean un conjunto de hilos en el dispositivo para ejecutar determinadas instrucciones. Los hilos se agrupan en conjuntos llamados bloques. Todos los hilos de un bloque se ejecutan simultaneamente en tandas de 32 llamadas *warps*. Cada warp se asigna a uno de los *streaming*

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

*multi-processors* (SM) del dispositivo, que posee una única unidad de control para controlar el flujo en todos sus núcleos o *streaming processors*(SP). Por lo tanto, cada hilo del warp ejecuta exactamente las mismas instrucciones en un SP distinto del mismo SM, típicamente sobre un juego de datos distintos. Los hilos de un mismo bloque pueden opcionalmente compartir datos mediante la memoria compartida del SM (NVIDIA, 2013).

Cada SM puede alojar un cierto número de warps simultaneamente dentro del chip, llamados residentes. Cuando un determinado warp debe esperar la transferencia de datos desde la memoria global, el SM cambia la ejecución a otro warp residente que esté listo para ejecutar operaciones. De esta manera, si la cantidad de warps residentes son suficientes es posible esconder la latencia del acceso a memoria, aprovechando al máximo todos los ciclos de reloj para realizar operaciones útiles.

La cantidad de threads residentes que puede acomodar un SM (que definen su ocupación u *occupancy*) depende fundamente de la cantidad de registros y memoria compartida disponibles en el SM y la cantidad requerida por cada hilo del kernel. Dado que los registros del GPU son típicamente de 32 bits, cada flotante de doble precisión requiere dos registros. Esto impacta negativamente sobre la performance al reducir la ocupación de los SM NVIDIA (2013).

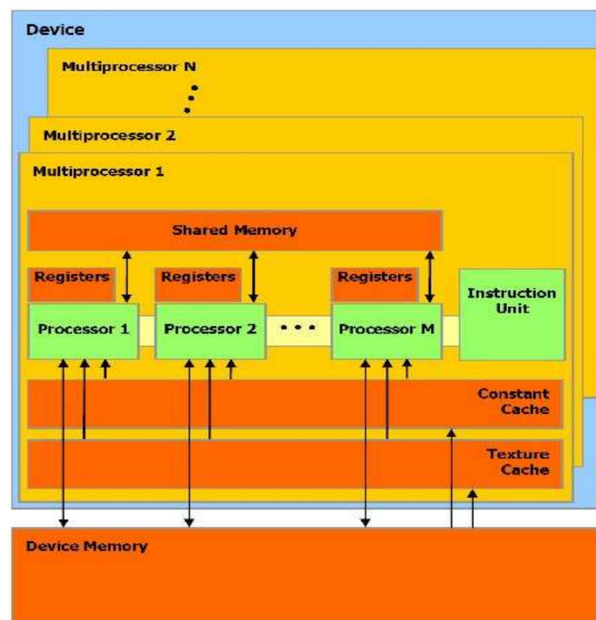



Figura 1: Estructura de multi-procesadores, núcleos y memoria de una GPU

### 3.2. Esquema de cálculo

El esquema de cálculo del modelo, llamado `fv2d`, es el siguiente:

1. Leer la malla de cálculo, generada mediante *GMSH*. La geometría (nodos, celdas y caras, vecinos) se describe mediante una serie de arrays de índices.

2. Generar los arrays de variables independientes, definidas para los centros de celda, y establecer la condición inicial del problema.

3. Transferir los arrays necesarios para definir la geometría y las variables independientes del problema a la memoria global de la placa.

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

4. Para cada paso de tiempo:

   *a*) Calcular el gradiente ilimitado de las variables de estado para cada celda.

   *b*) Limitar el gradiente de cada variables para cada celda, comparando el valor reconstruido en cada cara con el de las celdas vecinas según la ecuación (22).

   *c*) Para cada condición de borde, reconstruir un estado fantasma para cada cara y calcular los flujos, evaluando la ecuaciones (7) y (25).

   *d*) Calcular los flujos para cada una de las cara internas, evaluando las mismas ecuaciones (7) y (25).

   *e*) Totalizar los flujos para cada celda, calcular el término fuente y el estado medio del sistema evaluando la ecuación (31).

   *f*) Calcular nuevamente los flujos en las caras en el estado medio del sistema repitiendo los pasos (4*c*) y (4*d*).

   *g*) Totalizar los flujos para cada celda nuevamente, calcular el término fuente y el estado final del sistema evaluando la ecuación (32).

5. Cada cierta cantidad de pasos de tiempo, transferir los arrays de variables independientes al host y escribir los resultados.

Cada uno de los pasos (4*a*) a (4*g*) se ejecutan en el dispositivo a través de uno o más kernels, que se reemplazaron por funciones convencionales en la versión serial del código.

El modelo se desarrolló utilizando C++ y CUDA. Se utilizaron objetos para representar en el host los campos escalares que representan los distintos datos del problema. Esto permitió utilizar clases para facilitar los procedimientos de inicialización de los campos, lectura/escritura al dísco rígido, transferencia de los datos entre host y dispositivo, etc. No obstante, las rutinas de cálculo reciben como parámetros un puntero al array de datos internos de los campos, de manera que el código de los kernels escrito en CUDA no trabaja con clases ni estructuras, sino directamente sobre arrays. También se utilizaron objetos para la gestión de las condiciones de borde en el host.

### 3.3. Implementación del código

El código se desarrolló sobre cuatro principios básicos:

1. Definir una estructura de macros de pre-compilación para abstraer el código de si se realizará en CPU o GPU: de esta manera el mismo código puede compilarse como serial o paralelo, facilitando el desarrollo.

2. Estructurar todo el cálculo con paralelismo de datos: los datos de entrada y variables de estado se estructuraron mediante arrays y las operaciones se diseñaron como operaciones sobre un elemento cualquiera de esos vectores, que en principio podrían realizarse en cualquier orden o de manera simultanea. Estas operaciones se realizan de manera serial cuando el modelo corre en CPU y de manera paralela por tandas en el dispositivo. Este arreglo de operaciones no solo resulta natural para uso en el dispositivo, sino que también produce un muy buen uso del cache en la versión serial, ya que los valores de los arrays se leen en ese caso de manera consecutiva.

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

3. Estructurar las funciones de manera que al usar el dispositivo las lecturas y escrituras sean coalescentes en la mayor cantidad de casos posibles: de esta manera se optimizan la cantidad de accesos a la memoria global del dispositivo, que son el cuello de botella del rendimiento. Esto no es posible en todos los casos debido a la naturaleza no estructurada de la malla. Dado que cada elemento se lee de la memoria global una única vez, no se realiza un cacheado de los valores en la memoria compartida.

4. Minimizar la cantidad y extensión de bloques condicionales dentro de los kernels: los flujos en las caras de borde, por ejemplo, se realizan mediante kernels distintos que las caras internas.

En las secciones siguientes se describen en más detalle la estructura adoptada para los datos y el conjunto de macros de pre-compilación.

### 3.3.1. Estructura de los datos

Los datos se representaron como campos escalares, que se componen de un valor escalar individual para cada elemento (ya sea nodo, cara o celda). Estos se implementaron mediante una clase personalizada de vectores, que se definieron a través de una plantilla de clase `Vector<T>`. Esta se deriva de la clase `std::vector<T>`, permitiendo aprovechar las características del vector estándar de C++, especialmente el re-dimensionamiento dinámico, para facilitar la inicialización del modelo.

Una vez inicializado el campo en el host la clase derivada permite generar una copia de los datos en memoria del dispositivo mediante el método `transferDataToDevice`. Este método reserva lugar suficiente en el dispositivo y realiza la transferencia. El puntero al espacio reservado se almacena dentro de la clase. También existe un método inverso, `transferDataToHost`, que permite transferir los datos del dispositivo a la memoria del host. Este se utiliza para recuperar los resultados calculados para su almacenado en el disco rígido.

La clase cuenta con un método `Vector<T>::data([i])`, que devuelve un puntero al $i$-ésimo valor del vector. Este apunta a los datos almacenados en el host o en el dispositivo de acuerdo a si se usa o no el GPU. Las funciones de cálculo reciben como parámetro este puntero al primer elemento a calcular, aprovechando que los demás datos se encuentran en memoria de manera consecutiva.

La limpieza de los datos almacenados en el dispositivo se realiza automáticamente dentro del método destructor de la clase.

A partir de la clase plantilla se definen explícitamente clases concretas para campos de números enteros con signo y de punto flotante.

### 3.3.2. Macros de pre-compilación

Se definieron una serie de macros de pre-compilación para permitir la compilación del programa como un solver serial o paralelizado mediante GPU. Todos estos macros dependen de la existencia de la directiva `USECPU`, que al habilitarse transforma el programa en serial. Los macros principales, cuya definición literal se presenta en la tabla 1, se describen a continuación son:

- `VOIDKERNEL( fnName )`: permite la definición de la función parámetro ya sea como un kernel global en dispositivo o como una función convencional en CPU. Los parámetros de

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

estas funciones pueden ser tanto valores numéricos constantes o bien punteros al primer elemento de un vector sobre el que hay que operar.

- `CALLKERNEL( fnName , numElements )`: permite llamar un kernel, ya sea que vaya a ejecutarse en dispositivo o no. El entero `numElements` indica el número de elementos a procesar. En el caso de que se corra en el dispositivo, se calcula a partir de este y del tamaño de bloque la cantidad de bloques a ejecutar. Los punteros a los a procesar de los vectores se obtienen del método `Vector<T>::data()`.

- `DEVICEFUNCTION( type, fnName )`: permite la definición de una función que devuelve un valor de tipo `type`. La función se define de manera que se ejecute en el dispositivo o en CPU, y se fuerza en ambos casos que el compilador las trate como funciones inline, para evitar cambios de contexto. Todos los parámetros de estas funciones se pasan por valor o como referencias.

- `CALLDEVICEFUNCTION( type, fnName )`: permite llamar una función inline, ya sea en el dispositivo o no.

- `ITERATE( numElements )`: engloba todo el código dentro de una función definida como `VOIDKERNEL`. En el caso del CPU, el macro es reemplazado por un bucle simple que itera un índice $i$ entre $0$ y `numElements`. De esta manera el cálculo se desarrolla en forma serial para todos los elementos de los vectores. En cambio, al utilizar GPU, el macro calcula el índice $i$ para cada unidad de ejecución en función del *id* y tamaño del bloque y del *id* del threads. De esta manera cada unidad de ejecución trabaja en paralelo sobre un único elemento $i$ de los vectores de entrada y salida. A fin de evitar que el número de elementos de cálculo deba ser múltiplo del tamaño de bloque, se detienen los hilos cuyo $i$ sea mayor que `numElements`.

Para el cálculo en el dispositivo, la variable de pre-compilación `BLOCKSIZE` define el tamaño de bloque a utilizar. Esta se define al invocar al compilador.

A fin de ilustrar cómo queda estructurado el código de un kernel típico utilizando estos macros, se presenta en las figuras 3.3.3 el código de la función que totaliza los flujos en las caras de cada celda y actualiza las variables de estado se muestra en la figura, ligeramente simplificado. La correspondiente sintaxis para su invocación se presenta en la figura 3. Se observa que cada iteración del bucle, en el caso del programa serial, o cada hilo en el dispositivo trabaja sobre una única celda $i$. Se lee el índice de cada una de sus tres caras vecinas, cuyos flujos para cada variable, ya calculados, se leen de memoria y suman en la función inline `AddFlux`.

### 3.3.3. Coalescencia de los accesos a memoria global

A fin de maximizar la tasa de acceso a la memoria global es importante buscar la coalescencia de los accesos. En dispositivos CUDA con capacidad de cálculo 2.x la coalescencia se logra cuando threads consecutivos acceden simultaneamente a valores alojados en memoria global de manera consecutiva (NVIDIA, 2013).

La naturaleza no estructurada de la malla hace imposible tener acceso coalescente a la memoria global en todos los casos. El cálculo de los diversos kernels involucra básicamente campos que pueden corresponder a un valor para cada celda o bien un valor para cada cara. Utilizando la estructura de datos presentada en este trabajo, los kernels que iteran sobre las celdas realizan

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

```
VOIDKERNEL( UpdateVariables ) (
  UInt   numCells,    //Cantidad de celdas
  Float  dt,          //Paso de tiempo
  Int   *facec0s,     //Índice de la primer celda vecina de cada cara
  Float *cellAs,      //Área de cada celda
  //Índice de las caras vecinas de las celdas
  Int   *cellf0s,     Int   *cellf1s,     Int   *cellf2s,
  //Variables independientes en cada celda
  Float *cellh,       Float *cellhu,      Float *cellhv,
  //Flujos en cada cara
  Float *faceFluxh,   Float *faceFluxhu,  Float *faceFluxhv
) {
  ITERATE( numCells ) {
    // Acumular los flujos
    Float dFh = 0;  Float dFhu = 0; Float dFhv = 0;
    CALLDEVICEFUNCTION(void, AddFlux) (dFh, dFhu, dFhv, i, cellf0s[i],
        facec0s, faceFluxh, faceFluxhu, faceFluxhv);
    CALLDEVICEFUNCTION(void, AddFlux) (dFh, dFhu, dFhv, i, cellf1s[i],
        facec0s, faceFluxh, faceFluxhu, faceFluxhv);
    CALLDEVICEFUNCTION(void, AddFlux) (dFh, dFhu, dFhv, i, cellf2s[i],
        facec0s, faceFluxh, faceFluxhu, faceFluxhv);

    // Calcular términos fuentes Sh, Shu, Shv ....

    // Actualizar las variables
    Float A = cellAs[i];
    cellh[i]  = cellh[i]  + (Sh - dFh)   / A * dt;
    cellhu[i] = cellhu[i] + (Shu - dFhu) / A * dt;
    cellhv[i] = cellhv[i] + (Shv - dFhv) / A * dt;
  }
}
```

Figura 2: Código para la definición de un kernel

```
CALLKERNEL( UpdateVariables , numCells ) (
  numCells,               //Cantidad de celdas
  dt,                     //Paso de tiempo
  facecns[0]->data(0),    //Índice de la primer celda vecina de cada cara
  cellAs.data(0),         //Área de cada celda
  //Índice de las caras vecinas de las celdas
  cellfns[0].data(0), cellfns[1].data(0),  cellfns[2].data(0),
  //Variables independientes en cada celda
  cellh.data(0),      cellhu.data(0),      cellhv.data(0),
  //Flujos en cada cara
  faceFluxh.data(0),  faceFluxhu.data(0),  faceFluxhv.data(0)
);
```

Figura 3: Código para el llamado de un kernel

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

| Macro | `VOIDKERNEL( fnName )` |
|---|---|
| CPU | `void cpu##fnName` |
| GPU | `__global__ void gpu##fnName` |
| Macro | `CALLKERNEL( fnName, numElements )` |
| CPU | `cpu##fnName` |
| GPU | `gpu##fnName <<<((numElements)-1)/BLOCKSIZE+1,BLOCKSIZE>>>` |
| Macro | `DEVICEFUNCTION( type, fnName )` |
| CPU | `type cpu##fnName` |
| GPU | `__forceinline__ __device__ type gpu##fnName` |
| Macro | `CALLDEVICEFUNCTION( fnName )` |
| CPU | `cpu##fnName` |
| GPU | `gpu##fnName` |
| Macro | `ITERATE( numElements )` |
| CPU | `for (int i=0; i< (numElements); i++)` |
| GPU | `unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;`<br>`if (i >= (numElements))return;` |

Tabla 1: Macros de pre-compilación

lecturas/escrituras coalescentes sobre los arrays de celdas y no coalescentes sobre los arrays de caras. Para los kernels que iteran sobre las caras la situación se revierte.

Tomando como ejemplo el kernel presentado en la figura , que totaliza los flujos en las caras de la celda, calcula el término fuente y actualiza las variables de estado, es necesario accedear a una serie de datos almacenados por celda y otros datos almacenados por cara. Dado que cada thread procesa una celda, los datos almacenados por celda, como el área, índices de las caras vecinas, elevación del fondo, variables de estado, etc, se acceden de manera coalescente, ya que cada thread del warp lee simultáneamente valores enteros o flotantes consecutivos de cada array de datos. Los resultados también se escriben de manera coalescente. No osbtante, los datos almacenados por cara, como los flujos de cada variable, no pueden realizarse en general de manera coalescente, ya que las caras vecinas de una determinada celda no siguen ningún orden específico.

Por lo tanto, el modelo emplea una mezcla de accesos coalescentes y no coalescentes.

## 4. VALIDACIÓN

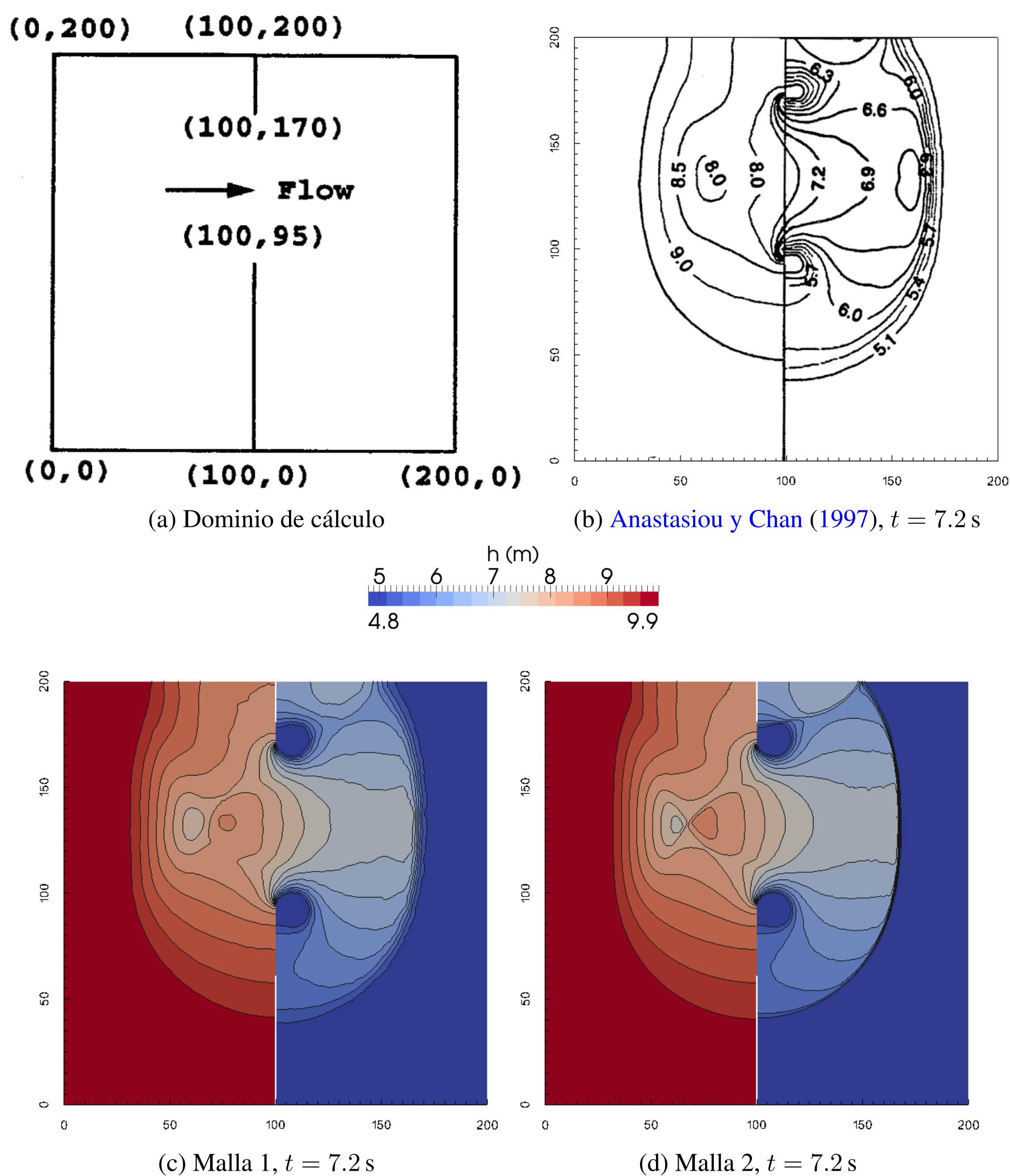
Para la validación del modelo se reprodujeron diversos casos publicas en la bibliografía. En todos estos casos la ecuación de conservación (1) es homogenea, es decir, no se consideran tensiones de corte externas ni fuerzas de Coriolis.

El primer caso de validación es la rotura parcial de presa estudiada por Fennema y Chaudhry (1990), y reproducida por diversos autores (Alcrudo y García-Navarro, 1993; Anastasiou y Chan, 1997; Hubbard, 1999; Jiwen y Ruxun, 2001). La geometría del problema consiste en un recinto cuadrado de 200 m de lado, dividido en dos mitades (figura 4a). En el instante inicial, existe una diferencia de nivel entre ambos lados de una brecha de 75 m. El nivel inicial de agua lado derecho es 10 m, mientras que del lado izquierdo es de 5 m. Todas los bordes se consideran paredes rígidas libremente deslizantes.

El caso se estudió para dos mallas de cálculo distintas: M1 y M2, de 18 876 y 379 182 celdas respectivamente. La simulación se desarrolló durante 7.2 s con un paso de tiempo de 0.001 s.

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

Los niveles de agua al final de la simulación se presentan en la figura 4, junto con los resultados presentados por Anastasiou y Chan (1997). Existe un muy buen acuerdo entre las simulaciones y la bibliografía. Se observa en los resultados un frente de choque que se propaga hacia aguas abajo, muy bien representado en la M2, y una onda de depresión que se propaga hacia aguas arriba.




(a) Dominio de cálculo

(b) Anastasiou y Chan (1997), $t = 7.2\,\mathrm{s}$

(c) Malla 1, $t = 7.2\,\mathrm{s}$

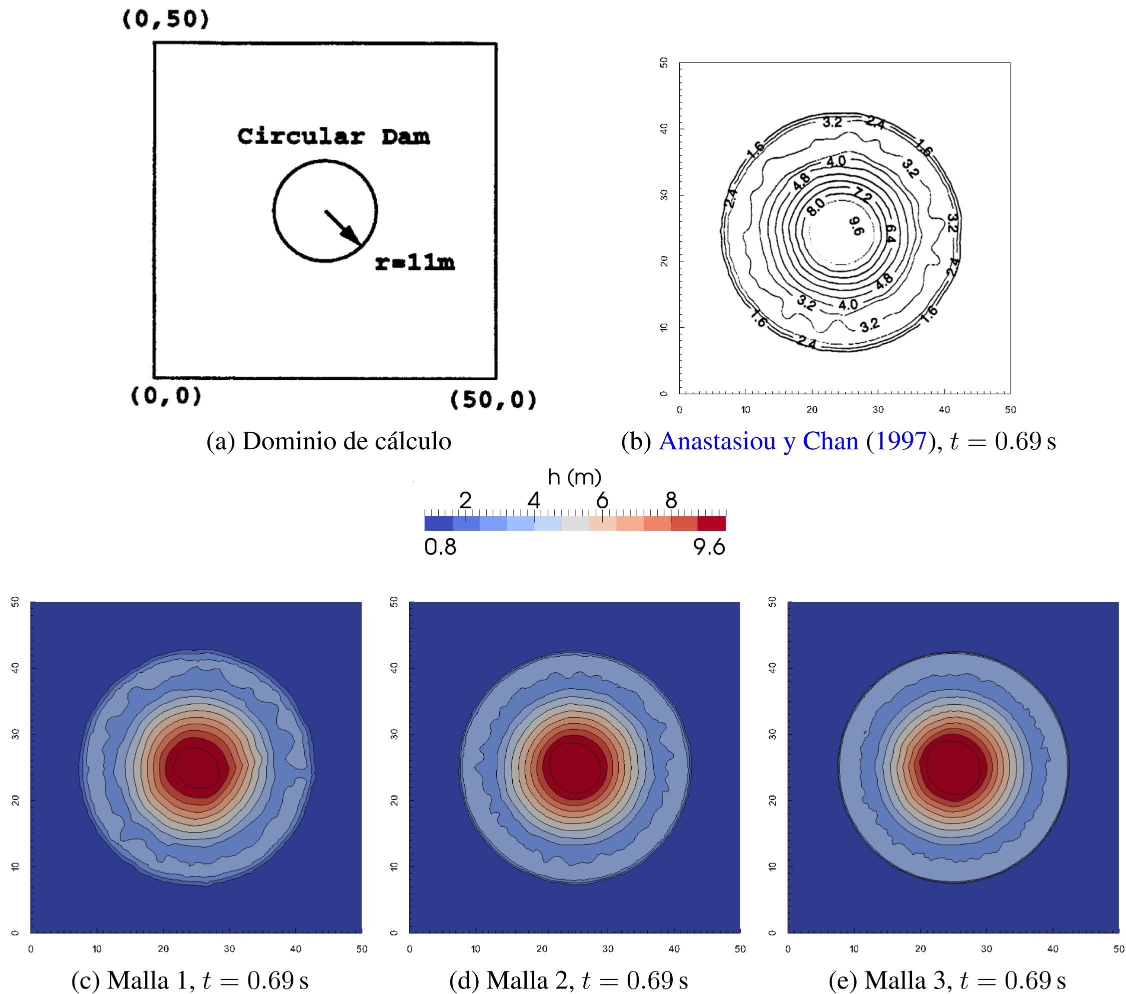
(d) Malla 2, $t = 7.2\,\mathrm{s}$

Figura 4: Rotura parcial de presa (Fennema y Chaudhry, 1990)

Se reprodujo también un caso hipotético de rotura instantanea de una presa circular, propuesto por Alcrudo y García-Navarro (1993), muy estudiado en la bibliografía (Anastasiou y Chan, 1997; Hubbard, 1999; Jiwen y Ruxun, 2001). El dominio de cálculo es cuadrado, de $50\,\mathrm{m}$ de lado (figura 5a). El nivel inicial en una región circular de $11\,\mathrm{m}$ de diámetro es de $10\,\mathrm{m}$, mientras que en el resto del dominio es de $1\,\mathrm{m}$. La simulación se desarrolló durante $7.2\,\mathrm{s}$, con un paso de tiempo de $1 \times 10^{-4}\,\mathrm{s}$. En la figura 5 se presentan los resultados del presente modelo, realizados sobre tres mallas de cálculo: M1, M2 y M3, de $8688$, $35\,168$ Y $140\,200$ celdas respectivamente.

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

Se comparan los resultados con los publicados por Alcrudo y García-Navarro (1993), presentados en la figura 5b, con los que existe un muy buen acuerdo. Al igual que en el caso anterior, la onda de choque que se propaga hacia afuera se observa más difundida en las mallas más gruesas.



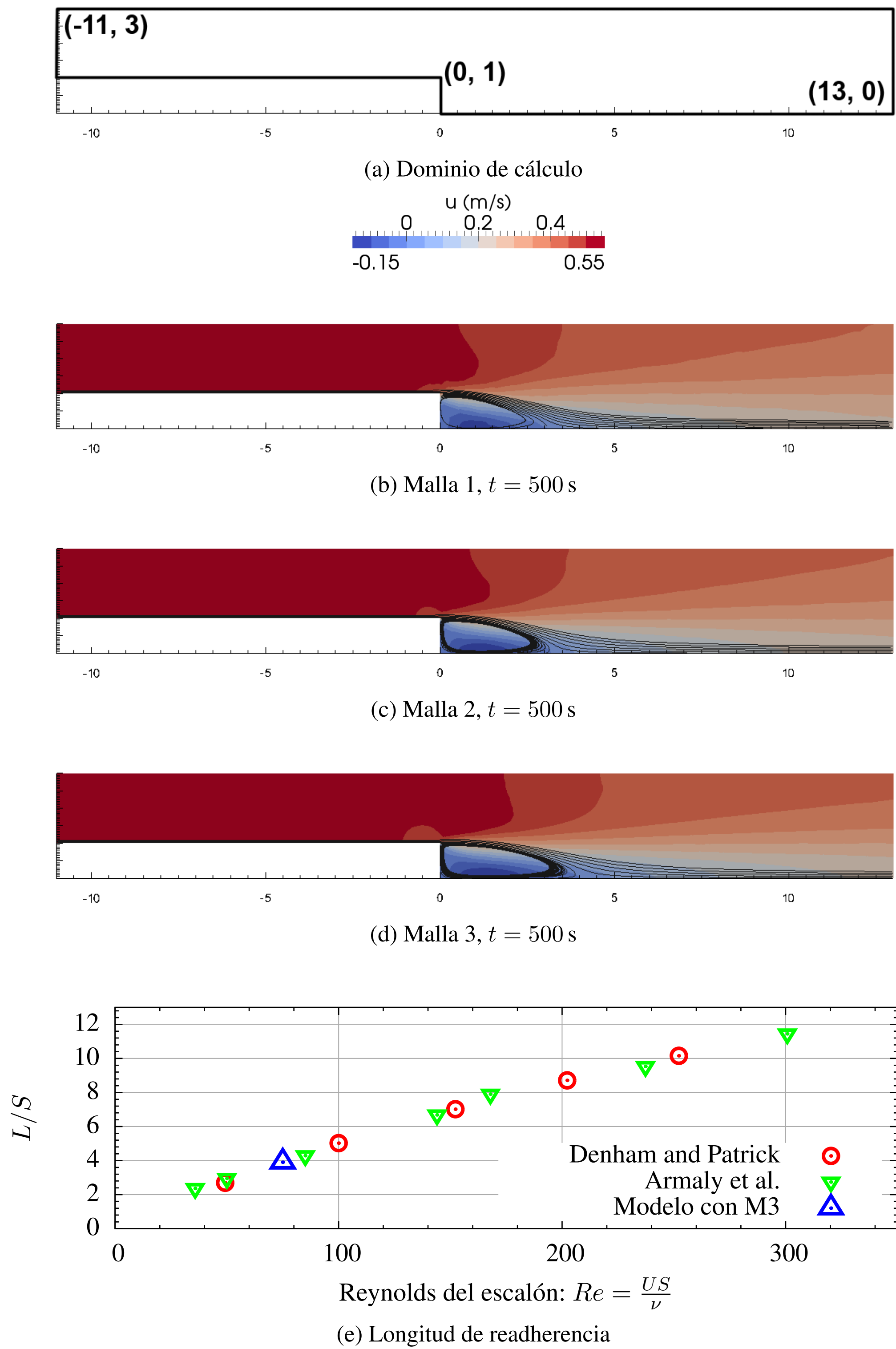

(a) Dominio de cálculo (b) Anastasiou y Chan (1997), $t = 0.69$ s

(c) Malla 1, $t = 0.69$ s (d) Malla 2, $t = 0.69$ s (e) Malla 3, $t = 0.69$ s

Figura 5: Rotura de presa circular (Alcrudo y García-Navarro, 1993)

Por último, se ensayó un caso de flujo en canal pasando un escalón a fin de validar el cálculo de los flujos viscosos. La geometría del problema es la propuesta por Anastasiou y Chan (1997), similar a la estudiada por Denham y Patrick (1974). La geometría se presenta en la figura 6a, y consiste en un canal de $2$ m de ancho que se expande a $3$ m. En el borde de entrada se fijó una velocidad de $0.5$ m/s y en el borde de salida un tirante de $1$ m. Se fijó un viscosidad de torbellino de $0.006\,67$ m$^2$/s, que resulta en un número de Reynolds del escalón de $75$. La simulación se desarrolló sobre $500$ s con un paso de tiempo de $0.002$ s. En la figura 6 se presentan los resultados para el instante final para tres mallas de cálculo: M1, M2 y M3, de $3294$, $13\,894$ y $56\,168$ respectivamente. Asimismo se presentan lineas de corriente para ilustrar la longitud de separación del flujo. En el caso de la malla M3 se obtuvo una longitud de readherencia de $3.9$ m, que coincide muy bien con los datos medidos que se presentan en la figura 6e (Denham y Patrick, 1974; Armaly et al., 1983). Para las mallas más gruesas se observan longitudes algo menores, debidas al efecto de la difusión numérica.

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar




(a) Dominio de cálculo

(b) Malla 1, $t = 500\,\mathrm{s}$

(c) Malla 2, $t = 500\,\mathrm{s}$

(d) Malla 3, $t = 500\,\mathrm{s}$

(e) Longitud de readherencia

Figura 6: Flujo pasando un escalón

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

## 5. RENDIMIENTO DE LA PARALELIZACIÓN

Se realizaron una serie de pruebas para evaluar el rendimiento de la estrategia de paralelización mediante GPU.

Cabe destacarse que el algoritmo utilizado por el modelo es básicamente el mismo, ya sea que se compile para host o dispositivo. De hecho, dada la diferencia de arquitecturas, la manera de estructurar el código de manera que sea eficiente en GPU no necesariamente conduce al máximo rendimiento posible en CPU. No obstante, el algoritmo no se considera particularmente nocivo para el desempeño en CPU, ya hace un uso eficiente del cache del procesador, factor que probablemente sea determinante en la performance.

En este trabajo se compara la performance en paralelo en el dispositivo con la performance serial en CPU, por lo que se utiliza un único hilo en el procesador. Se planea en el futuro modificar los macros de precompilación para incorporar opcionalmente paralelización en CPU mediante OpenMP. No obstante, dado que el algoritmo tiene una carga bastante elevada de accesos a memoria respecto de las operaciones matemáticas que realiza, no se espera un incremento de rendimiento muy significativo.

Para esta evaluación del rendimiento se adoptaron los mismos tres casos utilizados para la validación. En cada caso se cronometraron los tiempos de simulación para todas las mallas de cálculo, a fin de evaluar cómo escala el desempeño a medida que crece el número de elementos.

En el caso de rotura de presa circular, se realizaron las simulaciones en GPU para distintos tamaños de bloque: $64$, $128$, $256$ y $512$. Para los demás casos solo se ensayó un tamaño de bloque de $256$.

Las simulaciones en CPU se realizaron utilizando un único hilo en un procesador i7 950 3.07GHz, utilizando el compilador de GCC 4.4.7 bajo Linux, con nivel de optimización 3 (`-O3` ). Las simulaciones en GPU se realizaron en una placa NVIDIA Tesla C2075, instalada en un sistema idéntico. El código del dispositivo se compiló con NVCC 4.2. Algunas características de la placas utilizada se presentan en la tabla 2. Todas las pruebas presentadas se realizaron utilizando doble precisión, con números de punto flotante de 64 bits. No obstante, pruebas preliminares mostraron que, como es de esperarse, los resultados de speed-up al utilizar el dispositivo son bastante mayores en precisión simple de 32 bits.

| Dispositivo | nVidia Tesla C2075 |
|---|---|
| Multi-procesadores (SM) | 14 |
| CUDA Cores por SM | 32 |
| CUDA Cores totales | 448 |
| Compute Capabilities | 2.0 |
| Frecuencia de los CUDA Cores | $1.15$ GHz |

Tabla 2: Características del dispositivo GPU

En la tabla 3 se presenta los resultados de duración de cálculo para las distintas condiciones, tanto en CPU como en GPU. También se presentan los resultados de speed-up y los rendimientos del código en cantidad de celdas y pasos de tiempo procesados por unidad de tiempo. Estos resultados de rendimiento se presentan gráficamente en la figura 7. Los tiempos informados, así como las métricas derivadas, solo incluyen el período propiamente de cálculo, obviando la inicialización y transferencias de datos al dispositivo. Se evitó la impresión de resultados en todos los casos para no influenciar los tiempos.

Se observan los siguientes resultados:

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

| Caso | Malla | Cantidad de celdas | Tamaño de bloque | Tiempo de corrida i7 950 3.07 Ghz | Tiempo de corrida Tesla C2075 | Rendimiento pasos·celda/s | Speed-up |
|---|---|---|---|---|---|---|---|
| Alcrudo & García-Navarro | M1 | 8688 | 512 | 67.94 s | 2.17 s | $27.68 \times 10^6$ | 31.37 |
| | | | 256 | | 2.07 s | $28.90 \times 10^6$ | 32.76 |
| | | | 128 | | 2.10 s | $28.55 \times 10^6$ | 32.35 |
| | | | 64 | | 2.09 s | $28.71 \times 10^6$ | 32.54 |
| | M2 | 35 168 | 512 | 264.17 s | 7.63 s | $31.79 \times 10^6$ | 34.61 |
| | | | 256 | | 7.59 s | $31.96 \times 10^6$ | 34.80 |
| | | | 128 | | 7.66 s | $31.68 \times 10^6$ | 34.49 |
| | | | 64 | | 7.59 s | $31.97 \times 10^6$ | 34.81 |
| | M3 | 140 200 | 512 | 1021.30 s | 36.83 s | $26.26 \times 10^6$ | 27.73 |
| | | | 256 | | 36.63 s | $26.41 \times 10^6$ | 27.88 |
| | | | 128 | | 36.65 s | $26.40 \times 10^6$ | 27.87 |
| | | | 64 | | 36.29 s | $26.65 \times 10^6$ | 28.14 |
| Fennema & Chaudhry | M1 | 18 876 | 256 | 186.69 s | 5.81 s | $32.51 \times 10^6$ | 32.16 |
| | M2 | 379 182 | 256 | 4654.37 s | 137.29 s | $27.62 \times 10^6$ | 33.90 |
| Denham & Patrick | M1 | 3294 | 256 | 713.63 s | 41.74 s | $19.73 \times 10^6$ | 17.10 |
| | M2 | 13 894 | 256 | 3393.81 s | 106.31 s | $32.67 \times 10^6$ | 31.32 |
| | M3 | 56 168 | 256 | 14 895.90 s | 384.87 s | $36.49 \times 10^6$ | 38.70 |

Tabla 3: Tiempos de simulación y speed-ups

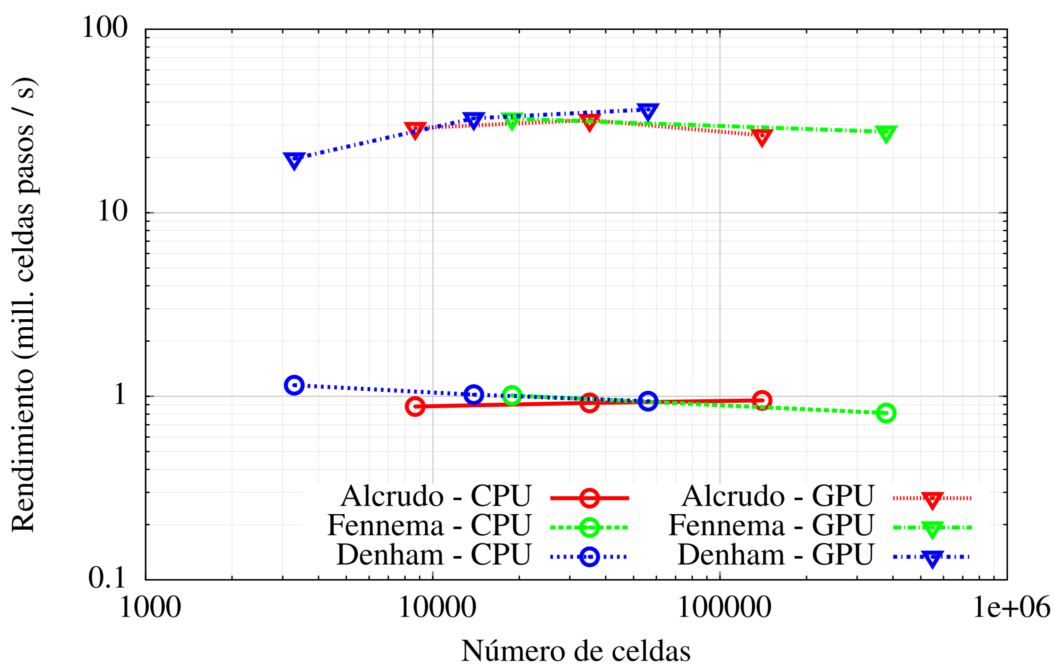



Figura 7: Rendimiento del modelo

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

1. Los resultados de speed-up al utilizar el dispositivo son muy buenos, obteniéndose valores de entre $17$ y $39$ para los distintos casos.

2. La eficiencia de la paralelización es mayor para mallas en el entorno de varias decenas de miles de elementos. Se ve una merma del speed-up para mallas tanto más grandes como más pequeñas.

3. El rendimiento en función del tamaño de bloque no presenta una tendencia definida, obteniéndose valores máximos para 256 o 64, con valores menores tanto para 512 como para 128. De cualquier manera, las diferencias entre el rendimiento mínimo y máximo para distintos tamaños de bloque es solo de entre 4 % y 1 %.

4. El modelo es capaz de procesar en el dispositivo entre $26.41$ y $36.49$ millones de pasos y celdas por segundo, mientras que el rendimiento serial es de entre $0.81$ y $1.15$ millones de pasos y celdas por segundo.

5. Se observa un rendimiento en dispositivo mucho bajo para la malla más gruesa del caso de Denham and Patrick que en cualquiera de los demás casos. Esto se debe sin duda a la baja cantidad de elementos en esa malla.

## 6. CONCLUSIONES

Se implementó un modelo de segundo orden para la resolución de las ecuaciones de *Aguas Poco Profundas* paralelizado mediante CUDA para GPU. Este se estructuró utilizando un conjunto de clases y macros que permiten fácilmente compilarlo tanto para simulaciones seriales en CPU como para procesamiento en GPU.

El modelo fue validado en condiciones homogeneas tanto para flujo invíscido como para flujo viscoso reproduciendo satisfactoriamente resultados publicados en la bibliografía.

Se evaluó la eficiencia de la paralelización obteniéndose speed-ups de hasta $39$ veces, respecto del rendimiento serial en CPU, utilizando precisión doble de 64 bits.

## REFERENCIAS

Alcrudo F. y García-Navarro P. A high-resolution Godunov-type scheme in finite volumes for the 2D shallow-water equations. *International Journal for Numerical Methods in Fluids*, 16:489–505, 1993.

Anastasiou K. y Chan C. Solution of the 2D shallow water equations using the finite volume method on unstructured triangular meshes. *International Journal for Numerical Methods in Fluids*, 24:1225–1245, 1997.

Armaly B., Durst J., Pereira J., y Schönung B. Experimental and theoretical investigation of backward-facing step flow. *jfm*, 127:443–496, 1983.

Badano N., Sabarots Gerbec M., y Re M. A coupled hydro-sedimentologic model to assess the advance of the Parana River delta front. En *International Conference on Fluvial Hydraulics - River Flow 2012*. 2012.

Barth T. y Jespersen D. The design and application of upwind schemes on unstructured meshes. *AIAA Paper*, 89-0366, 1989.

Batten P., Lambert C., y Causon M. Positively conservative high-resolution convection schemes for unstructured elements. *International Journal for Numerical Methods in Engineering*, 39:1821–1838, 1996.

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

Begnudelli L. y Sanders B. Unstructured grid finite-volume algorithm for shallow-water flow and scalar transport with wetting and drying. *Journal of Hydraulic Engineering*, 132:371–384, 2006.

Bradford S. y Katopodes N. Hydrodynamics of turbid underflows I: Formulation and numerical analysis. *Journal of Hydraulic Engineering*, 125:1006–1015, 1999.

Bradford S. y Sanders B. Finite-volume model for shallow-water flooding of arbitrary topography. *Journal of Hydraulic Engineering*, 128:289–298, 2002.

Costarelli S., Storti M., Paz R., y Dalcin L. Solving incompressible 3D viscous fluid flows using CUDA. En *X Congreso Argentino de Mecánica Computacional (MECOM)*. Salta, Salta, 2012.

Denham M. y Patrick M. Laminar flow over a downstream-facing step in a two-dimensional flow channel. *Transactions of the Institute of Chemical Engineering*, 52:361–367, 1974.

Fe J. Aplicación del método de volúmenes finitos a la resolución numérica de las ecuaciones de aguas someras con incorporación de los esfuerzos debidos a la turbulencia. *International Journal for Numerical Methods in Fluids*, 2005.

Fe J., Navarrina F., Puertas J., Vellando P., y Ruiz D. Experimental validation of two depth-averaged turbulence models. *International Journal for Numerical Methods in Fluids*, 60:177–202, 2009.

Fennema R. y Chaudhry M. Explicit methods for 2D transient free-surface flows. *Journal of Hydraulic Engineering*, 116:1013–1034, 1990.

Harten A. y Hyman J. Self-adjusting methods for one-dimensional hyperbolic conservation laws. *Journal of Computational Physics*, 50:235–269, 1983.

Hubbard M. Multidimensional slope limiters for MUSCL-type finite volume schemes and unstructured grids. *Journal of Computational Physics*, 155:54–74, 1999.

Jiwen W. y Ruxun L. The composite finite volume method on unstructured meshes for the two-dimensional shallow water equations. *International Journal for Numerical Methods in Fluids*, 37:933–949, 2001.

Lecertua E. *Análisis de riesgo de inundaciones en las áreas costeras del Río de la Plata considerando Cambio Climático*. Tesis de grado, Facultad de Ingeniería, Universidad de Buenos Aires, Buenos Aires, 2010.

Menenguci W., Valli A., Cantabriga L., y Veronese L. Un algoritmo CUDA em diferenças finitas para a discretizaçao das equaçoes de Navier-Stokes. En *IX Congreso Argentino de Mecánica Computacional (MECOM), XXXI Iberian-Latin-American Congress on Computational Methods in Engineering, II Congreso Sudamericano de Mecánica Computacional*. Buenos Aires, 2010.

Nickolls J., Buck I., Garland M., y Skadron K. Scalable parallel programming with CUDA. *ACM Queue*, 6:14–53, 2008.

NVIDIA. *CUDA C Programming Guide v5.5*. 2013. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

Re M. *Impacto del Cambio Climático Global en las costas del Río de la Plata*. Tesis de maestría, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Buenos Aires, 2005.

Roe P. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43:357–372, 1981.

Senocak I., Thibaultl J., y Caylor M. Rapid-response urban CFD simulations using GPU computing paradigm on desktop supercomputer. En *Proceedings of the Eighth Symposium on the Urban Environment*. Phoenix, Arizona, 2009.

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

Shinn A. y Vanka S. Implementation of a semi-implicit pressure-based multigrid fluid flow algorithm on a graphics processing unit. *ASME Conference Proceedings*, 43864:125–133, 2009.

Simões F. Finite volume model for two-dimendional shallow environmental flow. *Journal of Hydraulic Engineering*, 137:173–182, 2011.

Thibaultl J. y Senocak I. CUDA implementation of a Navier-Stokes solver on a multi-GPU desktop platforms for incompresible flows. En *Proceedings of the 7th AIAA Aerospace Sciences Meeting Including New Horizons Forum and Aerospace Exposition*. Orlando, Florida, 2009.

van Leer B. Towards the ultimate conservation difference scheme. V. a second order sequel to Godunov's method. *Journal of Computational Physics*, 32:101, 1979.

van Leer B. On the relation between the upwind-differencing schemes of Godunov, Engquist-Osher and Roe. *SIAM Journal on Scientific and Statistical Computing*, 5, 1984.

Copyright © 2013 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar