Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXVII, págs. 2943-2957 (artículo completo)
Alberto Cardona, Mario Storti, Carlos Zuppa. (Eds.)
San Luis, Argentina, 10-13 Noviembre 2008

PARALELIZACIÓN DE AUTÓMATAS CELULARES DE AGUAS SUPERFICIALES SOBRE PLACAS GRÁFICAS

P.Rinaldi[a,b], C. García Bauza[a,c], M. Vénere[a,b] and A. Clausse[a,b]

[a]PLADEMA, Universidad Nacional del Centro, 7000 Tandil, Argentina
pladema @exa.unicen.edu.ar,
[b]CONICET-CNEA
[c]CICPBA

Palabras Clave: Autómatas Celulares, GPU Computing, Computación en Paralelo.

Resumen: Los simuladores basados en autómatas celulares (AC) son esencialmente esquemas explícitos con operaciones locales y, por lo tanto, son especialmente aptos para ser implementados en plataformas masivamente paralelas como el hardware integrado en placas gráficas. El modelo de programación de nVIDIA® denominado Compute Unified Device Architecture (CUDA™) es una plataforma para el desarrollo de software para computación paralela de alto desempeño sobre las potentes placas gráficas de la misma marca. En este trabajo se presenta una primera implementación de AC para simulación de escurrimientos de fluidos en CUDA utilizando varias estrategias de paralelización y diferentes niveles de memoria de la placa gráfica (memoria global, compartida y bancos de registros). Se presentan implementaciones donde se consiguieron aceleraciones de hasta 6 veces en comparación con una implementación equivalente sobre la CPU. Al aumentar el tamaño de entrada al mismo tiempo que el nivel de paralelización y la cantidad de *threads*, las diferencias en performance superaron un orden de magnitud.

1 INTRODUCCIÓN

1.1 La Unidad de Procesamiento Gráfico o GPU

La GPU es el chip de las placas gráficas que efectúa las operaciones requeridas para renderizar píxeles en la pantalla. Las GPUs modernas de uso doméstico están optimizadas para ejecutar una instrucción simple (*kernel*) sobre cada elemento de un extenso conjunto simultáneamente (*i.e.* Instrucción Simple y Múltiples Datos (SIMD)). La fuerza impulsora del desarrollo de las placas gráficas ha sido el negocio millonario de la industria de los videojuegos. De hecho, la performance de las GPUs ha excedido la ley de Moore por más de 2.4 veces al año (Moreland and Angel, 2003). La causa de esto es la ausencia de circuitos de control que es lo que ocupa mayor espacio en una CPU. En contraste, las GPU dedican casi la totalidad del espacio de chip a Unidades Aritmético Lógicas (ALU) incrementando enormemente el poder de procesamiento, pero al costo de penalizar severamente las elecciones de bifurcación incorrectas. Para el uso tradicional en gráficos, esto no es un inconveniente, pero cuando se intenta utilizar la placa gráfica como un procesador de propósito general, es un problema complicado que debe ser tenido en cuenta.

El esquema de la figura 1 muestra las cantidades de transistores dedicados a diferentes tareas en la CPU comparado con la GPU.

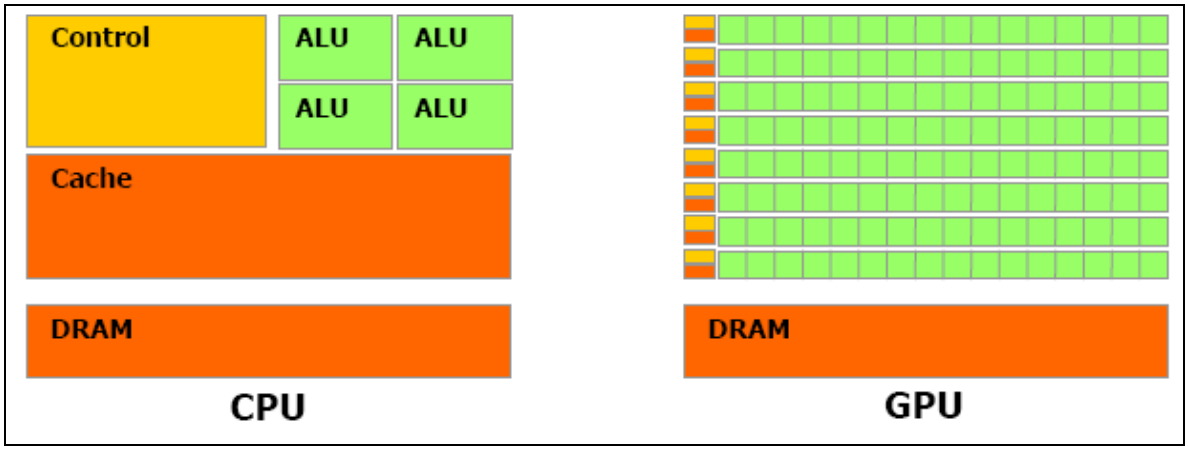

Figura 1: Comparación del uso de transistores entre CPU y GPU (NVIDIA 2007)

1.2 Arquitectura de una GPU NVIDIA GeForce de la serie 8.

La placa gráfica utilizada en este trabajo tiene una GPU NVIDIA de la serie GeForce 8000 (GeForce 8600 GT). La serie GeForce 8 de placas gráficas de la empresa NVIDIA fue desarrollada en conjunto con el modelo de programación CUDA (NVIDIA 2008) de la misma marca y por lo tanto es la primera línea de GPUs compatibles con esa arquitectura. Las GPUs de esta línea están compuestas por un conjunto de multiprocesadores como se ilustra en la figura 2. Estos multiprocesadores tienen una arquitectura SIMD, lo que significa que cada uno de los procesadores dentro de los multiprocesadores ejecuta la misma instrucción en cada ciclo de reloj, pero sobre diferentes datos simultáneamente. Cada uno de estos multiprocesadores tiene sus propias memorias de los siguientes cuatro tipos:

- Registros de 32-bits por procesador.

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

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

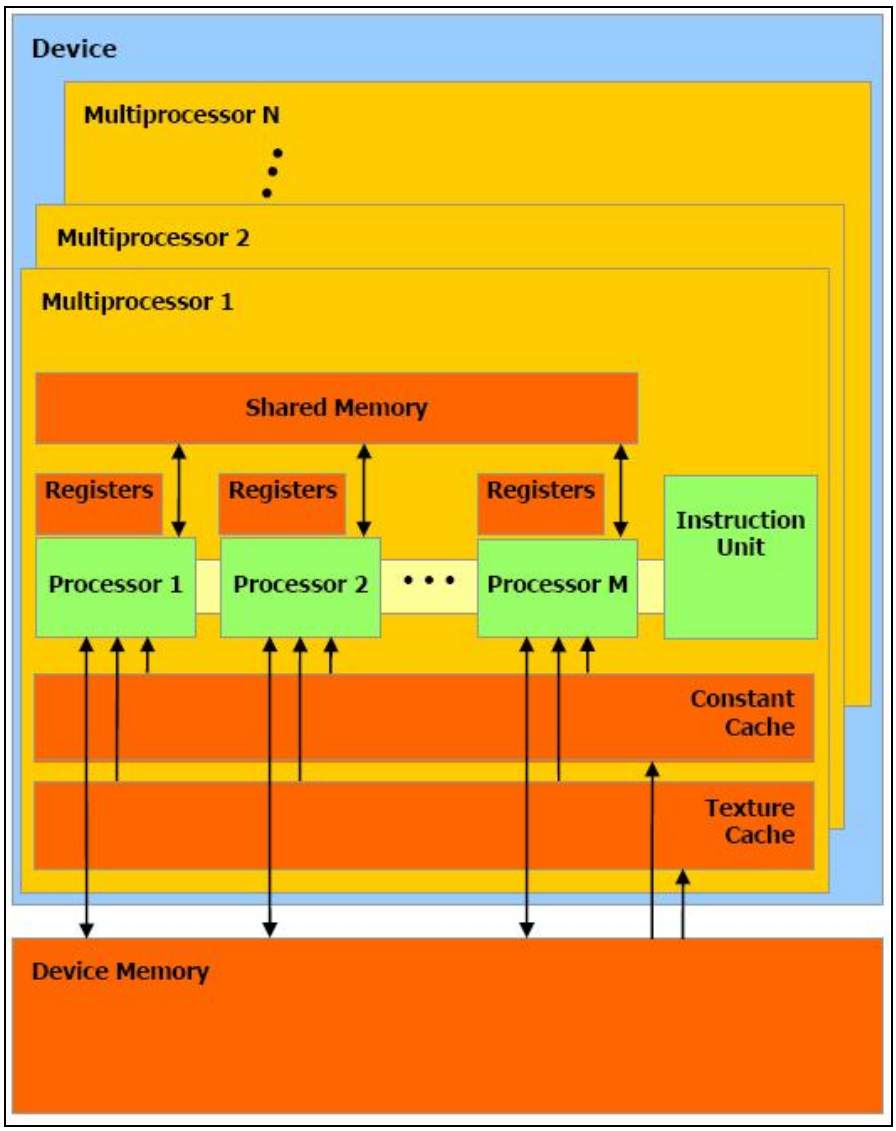

Figura 2: Conjunto de multiprocesadores SIMD con memoria compartida en una GPU NVIDIA Geforce 8. (NVIDIA 2007)

2 MODELO DE PROGRAMACIÓN CUDA.

La tecnología CUDA de NVIDA es una nueva arquitectura que permite a la placa gráfica resolver problemas computacionales complejos. CUDA facilita el acceso de aplicaciones con alto costo computacional al poder de procesamiento de las GPU NVIDIA a través de una nueva interfaz de programación. La utilización del lenguaje C estándar simplifica en gran medida el desarrollo de software. El conjunto de herramientas de CUDA es una solución completa para programar placas gráficas compatibles. El *toolkit* incluye librerías estándar FFT (Fast Fourier Transform) y BLAS (Basic Linear Algebra Subroutines), un compilador de C para la GPU y un driver específico. La tecnología CUDA es compatible con los sistemas operativos Linux y Windows® XP®. En este trabajo se utilizó la versión 1.1 sobre plataforma Windows XP.

2.1 Interfaz para la programación de aplicaciones (API).

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

2.2 Bloques de *threads*:

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

Internamente, la variable global de la función *blockDim* contiene las dimensiones del bloque. La variable interna global *threadidx* (*thread index*) contiene el índice del *thread* dentro del bloque. Para explotar el potencial del hardware eficientemente un bloque debe contener por lo menos 64 *threads* y no más de 512 (Tölke 2008).

2.3 Grillas de bloques de *threads*.

Existe un número máximo de *threads* por bloque limitado por el hardware, el cual incluso puede ser menor según la cantidad de memoria local y compartida utilizada por cada uno. Sin embargo, bloques que ejecuten el mismo *kernel* pueden ser agrupados en una grilla de bloques, con lo que el número de *threads* que pueden ser lanzados en un solo llamado es mucho mayor. Esto tiene como contrapartida una menor cooperación entre *threads*, ya que los *threads* de diferentes bloques no pueden comunicarse de forma segura o sincronizarse entre sí. Cada bloque se identifica por su *block ID* y la estructura de una grilla se especifica en el llamado a la función del mismo modo que la dimensión de los bloques. Los diferentes bloques de una grilla pueden correr en paralelo y para aprovechar el hardware eficientemente se

deberían utilizar al menos 16 bloques por grilla (Tölke 2008).

2.4 Tipos de funciones

Los calificadores de funciones o Function Type Qualifiers se utilizan para definir el tipo de función, es decir:

- ***device*** es una función que se ejecuta en la placa gráfica y se puede llamar sólo desde la placa.
- ***global*** es una función *kernel*. Se ejecuta en el dispositivo y se puede llamar sólo desde el *host*. Cualquier llamada a una función *global* debe especificar la configuración de ejecución (dimensiones de la grilla de bloques y de los bloques).
- ***host*** es una función que se ejecuta en el *host* y que sólo se puede invocar desde el *host*. Similar a una función C clásica.

2.5 Tipos de variables

- ***device*** es una variable que reside en el espacio global de memoria de la placa. Es accesible desde todos los *threads* de la grilla (con una latencia aproximada de 200-300 ciclos de reloj) y desde el *host* a través de las librerías.
- ***shared*** es una variable que se almacena en el espacio compartido de memoria de un bloque y es accesible únicamente desde todos los *threads* del bloque (pero con una latencia de sólo 2 ciclos de reloj).

2.6 Administración de la memoria

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

- ***cudaMalloc(…)*** ubica una cantidad especificada de bytes de memoria consecutiva y retorna un puntero a ese espacio que puede utilizarse para cualquier tipo de variable.
- ***cudaMemcpy(…)*** copia bytes de un área de memoria a otra. Pudiendo copiar del *host* al *host*, del *host* al dispositivo, del dispositivo al *host* y dentro del dispositivo.

Ambas funciones sólo pueden ser llamadas desde el *host*.

2.7 Sincronización:

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

3 MODELO AQUA DE ESCURRIMIENTO SUPERFICIAL

3.1 Evolución de AQUA

En el año 2002 se presentó el modelo de escurrimiento superficial AQUA (Vénere and Clausse, 2002) de autómatas celulares que tenía varias ventajas con respecto a los modelos clásicos de hidrología en cuanto a precisión y nivel de detalle. Este modelo fue reformulado y calibrado con datos provenientes de inundaciones reales en (Dalponte et al. 2007, Rinaldi et al. 2007a). AQUA utiliza una grilla regular como representación de la geometría del terreno. La información sobre la topografía se encuentra discretizada en celdas de igual tamaño que contienen un valor de altura del terreno y cota de agua en un determinado instante. El modelo

básicamente toma conjuntos de 9 celdas siguiendo la vecindad de Moore y redistribuye el agua contenida buscando la cota más baja.

El presente trabajo se basa en el autómata AQUA-GRAPH (Dalponte et al. 2005), el cual introduce en AQUA la vecindad de von Neumann como un grafo no dirigido, determinando el volumen de agua transferido de una celda a otra en un paso en base a las diferencia de alturas totales y a parámetros de los arcos del grafo. Posteriores extensiones de este modelo como GRAPHFLOW (Rinaldi et al. 2007b) permiten utilizar células de diferentes tamaños para reducir el costo computacional en base a diversos tipos de discretizaciones temporales y espaciales. Este autómata ha demostrado capacidad de simular eventos reales de tormenta para amplias regiones de llanura con alta performance debido a la baja complejidad computacional de sus cálculos y aprovechando el alto nivel de detalle de los Modelos Digitales del Terreno (MDT) generados en base a imágenes satelitales.

3.2 Reglas de AQUA-GRAPH

En AQUA-GRAPH cada celda de una grilla regular, identificada como $n_{(x,y)}$, representa una porción del terreno y contiene la información de la cota promedio $h_{(x,y)}$, el nivel de agua $w_{(x,y)}$, y fuentes-sumideros (Dalponte et al. 2005, Rinaldi et al. 2007a).

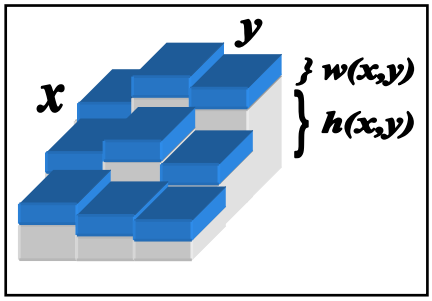

Figura 3: Representación interna del modelo

$T_{(x,y)}(t)$ define la altura total de la celda ubicada en la coordenada (x, y), en un instante determinado y es la suma de la cota de terreno $h_{(x,y)}$ más el nivel de agua en ese instante $w_{(x,y)}(t)$:

$$T_{(x,y)}(t) = h_{(x,y)} + w_{(x,y)}(t) \quad (1)$$

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

1. Para cada celda determinar el entorno de recepción comprendido por las celdas vecinas con altura total $T_{(x,y)}$ inferior a la celda considerada (Figura 4):

$$R_{(x,y)}(t) = \left\{ x_{(u,v)} / x-1 \le u \le x+1, y-1 \le v \le y+1 \; y \; T_{(u,v)}(t) < T_{(x,y)}(t) \right\} \quad (2)$$

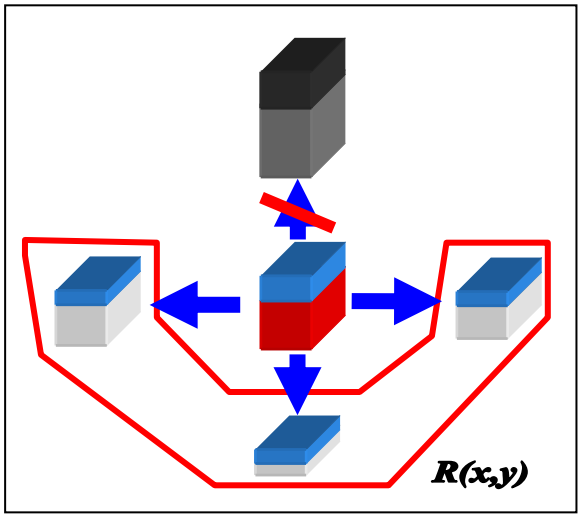

Figura 4: Entorno de recepción de una celda (centro).

2. Calcular el nivel que alcanzará el agua en la celda y su entorno si todo el líquido escurriera a su mínima posición, llamada altura de equilibrio $Teq_{(x,y)}(t)$.

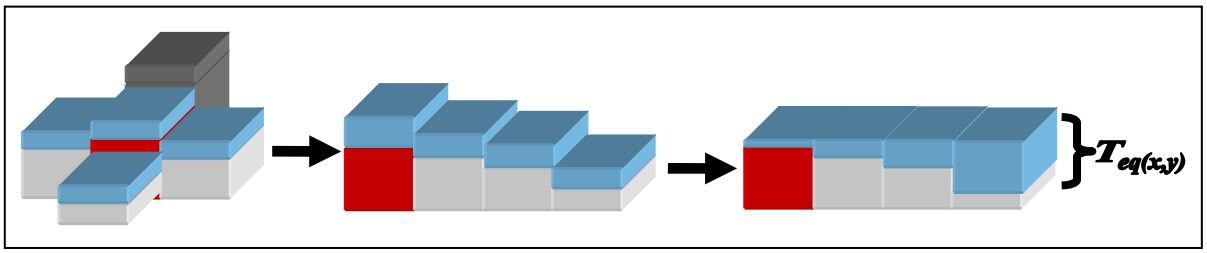

Figura 5: Cálculo de $Teq_{(x,y)}$ para la celda $n_{(x,y)}$ (en rojo) y su entorno $R_{(x,y)}$ (gris). El nodo oscuro pertenece a la vecindad pero no al entorno.

3. Determinar la cantidad de agua que saldrá de la celda $\Delta w_{(x,y)}(t)$ como:

$$\Delta w_{(x,y)}(t) = \min\left(T_{(x,y)}(t) - T_{eq(x,y)}(t),\ w_{(x,y)}(t)\right) \tag{3}$$

4. Distribuir $\Delta w_{(x,y)}(t)$ entre las celdas del entorno $R_{(x,y)}$ pesando con la raíz cuadrada de las diferencias de altura siguiendo la ley de pérdida de carga de sistemas hidráulicos (Goldstein 1965):

$$\Delta w_{(x,y,u,v)}(t) = \Delta w_{(x,y)}(t)\frac{\sqrt{T_{(x,y)}(t) - T_{(u,v)}(t)}}{\sum_{k \in R_{(x,y)}(t)} \sqrt{T_{(x,y)}(t) - T_{(u,v)}(t)}} \tag{4}$$

5. Transferir a cada celda $n_{(u,v)}$ en $R_{(x,y)}$ el nivel de agua correspondiente $\Delta w_{(x,y,u,v)}$

multiplicado por un coeficiente de relajación α:

$$w_{(u,v)}\left(t+1\right) = w_{(u,v)}(t) + \Delta w_{(x,y,u,v)}\left(t\right) \bullet \alpha \tag{5}$$

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

$$w_{(x,y)}\left(t\right) = w_{(x,y)}(t+1) \tag{6}$$

4 IMPLEMENTACIÓN

4.1 Representación de los datos.

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

4.2 Kernels

La ejecución se divide en 2 pasos principales consecutivos. En el primer paso, denominado *Flow(),* cada celda ejecuta los cinco primeros pasos del algoritmo presentado en la sección 3, y se almacenan los resultados parciales en el arreglo auxiliar *dAux*.

Para evitar problemas de concurrencia se acumula en uno de los campos del arreglo auxiliar los incrementos de agua en las celdas vecinas y en el otro campo los decrementos de la celda que se analiza. Por ejemplo, una celda ubicada en la posición *"i"* del arreglo tiene a *dTerreno[i].x* como altura de terreno y *dTerreno[i].y* como cota de agua en un determinado tiempo *t*. De esta manera, si esta celda transfiere un nivel de agua *w* a la celda ubicada en *"j"*, el incremento en la altura de agua de la celda vecina se almacena en *dAgua[j].y* y el decremento en la celda origen se almacena en *dAgua[i].x*. Este primer paso es el que trae problemas a la hora de paralelizar porque la ejecución sobre una celda debe leer y actualizar varias celdas vecinas.

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

4.3 Threads y Bloques

En una primera implementación, con grillas de hasta 256 × 256 celdas, todas las simulaciones se realizaron con un solo bloque de *threads*. Cada *thread* recorre una única fila de la grilla de principio a fin ejecutando el algoritmo. La fila sobre la que ejecuta cada *thread*

se determina por la variable interna *threadidx.y*. Los *threads* se sincronizan al comenzar y finalizar el recorrido. El inconveniente de esta paralelización es que *threads* que comparten alguna celda vecina actualizan al mismo tiempo celdas de la grilla auxiliar (figura 6).

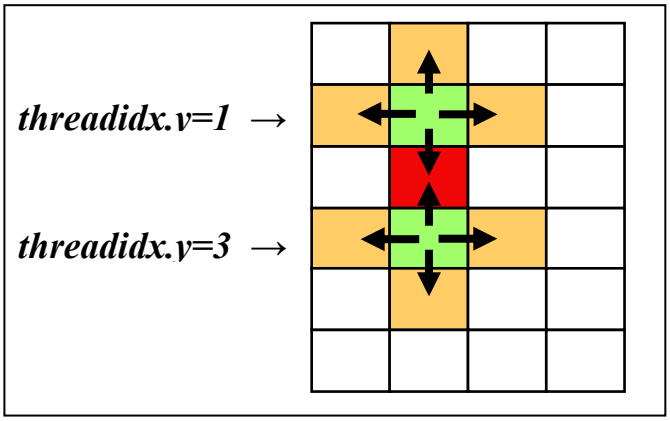

Figura 6: Ambos *threads* actualizan la celda 1,3 (en rojo) al mismo tiempo

La mayoría de los simuladores que han sido implementados exitosamente sobre GPUs (Goodnight 2007, Tölke 2008, Zhao 2007) son modelos que para cada celda leen el estado de las vecinas, computan el nuevo estado de la celda y lo actualizan en memoria auxiliar. Estos simuladores no tienen problemas de concurrencia al ser paralelizados independientemente de la tecnología aplicada.

En (Podlozhnyuk 2007) se implementan índices desfasados sobre memoria compartida para que evitar errores de concurrencia cuando varios *threads* escriben el mismo dato. En nuestro caso se optó por una solución similar desfasando los índices en función de la fila que recorren y cada línea se trata como un arreglo circular. Los *threads* empiezan a recorrer la fila desde un punto intermedio y al llegar al final saltan al inicio para completar el recorrido. El corrimiento de índices se muestra en el figura 7. También deben incluirse sincronizaciones dentro de los ciclos de cada *thread* para que los índices sigan conservando la separación a lo largo del las filas. En la figura 8 se muestra el código del *kernel* que maneja los índices.

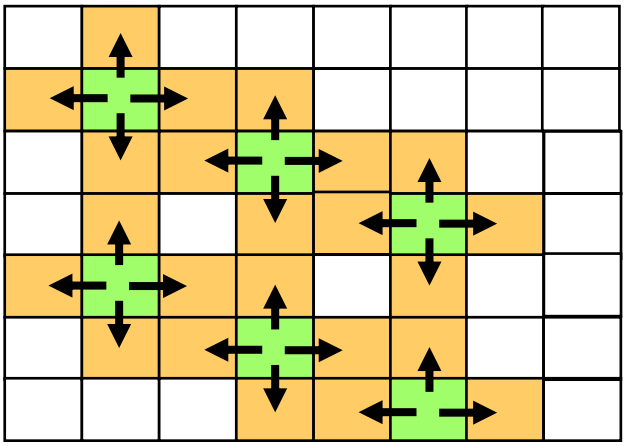

Figura 7: Cada *thread* comienza 3 celdas más adelante que el anterior para no producir errores de concurrencia.

```
__global__ void Flow( float2* dTerreno, float2* dAux, int DIMX, int DIMY)
{
      int idy = threadIdx.y;
      int desp = ((idy)*2)%6; //el desplazamiento del inicio en cada thread
      __syncthreads();        //sincronización

      for (int p=0; p<DIMX; p++){   //recorre toda la dimensión en x
            __syncthreads();
            int indice = p + desp; //suma el desplazamiento por desfasaje
            indice = indice%DIMX;  //tomo modulo para que caiga dentro
            …
      }
}
```

Figura 8: Desplazamiento de índices dentro del *kernel*.

Para la GPU utilizada en este trabajo, el esquema de cálculo descrito sólo es válido para grillas que no excedan las 256 celdas por lado, lo cual constituye el límite de *threads* que se pueden lanzar en simultáneo dentro de un bloque. Para grillas mayores la ejecución debe dividirse en bloques cuadrados de igual tamaño donde se ejecuta el mismo código *kernel* pero en diferentes bloques de *threads*. En este caso surge un inconveniente en las filas de los límites entre bloques, donde *threads* de un bloque deben actualizar celdas que están siendo analizadas por *threads* de otro bloque. Si bien este problema no es común a otros AC, existen generalmente problemas similares al unir porciones de datos procesados en paralelo. La solución adoptada es similar a la del método *LBExchange()* (Tölke 2008).

Para evitar problemas de concurrencia, cada bloque ejecuta sobre las filas internas de la grilla y al terminar esta ejecución se corre otra función *kernel* (de un solo bloque) que calcula el modelo en los bordes horizontales de las divisiones de la grilla. Esta función se denomina *FlowUnion()*. El esquema se muestra en la figura 9 donde las líneas rojas marcan la división en bloques. El algoritmo se ejecuta con el *kernel Flow()* para las celdas naranja y con *FlowUnion()* para las celdas verdes. Las flechas indican el punto de inicio y el sentido de ejecución de los *threads* de cada bloque.

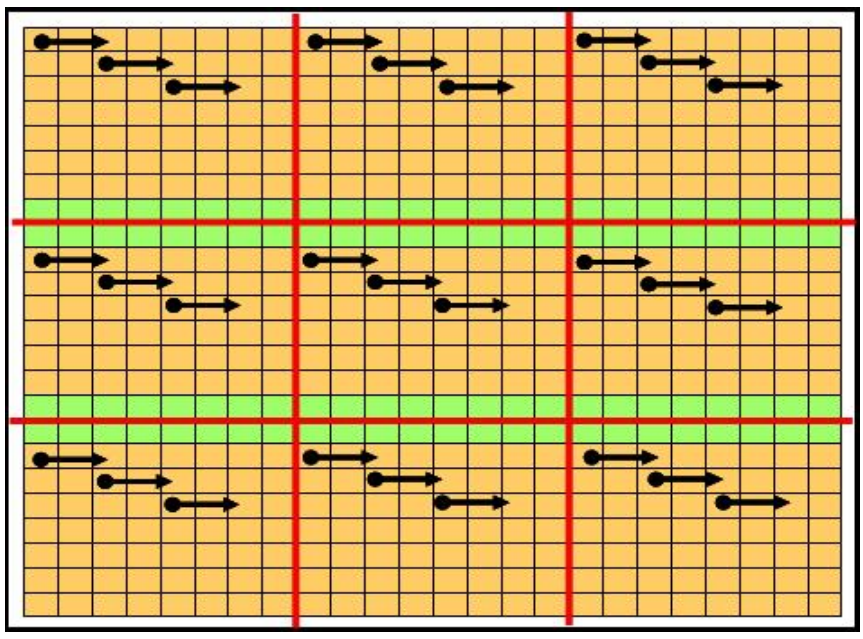

Figura 9: Configuración de bloques de *threads* para grandes tamaños de grillas.

4.4 Comparación de performance

Para realizar comparaciones de tiempo de respuesta con un cálculo equivalente sin placa gráfica, se implementó el mismo algoritmo en C++. Los datos de entrada son exactamente los mismos pero sin necesidad de copiarlos a la memoria del dispositivo. Esta implementación es de un único *thread* y con dos pasos que recorren la grilla: el escurrimiento o *Flow()* donde se implementa hasta el paso 5 del algoritmo de la sección 3, y la actualización o *Actualize()* que implementa el último paso.

Se compararon los resultados de ambos códigos en una simulación del escurrimiento superficial sobre un plano inclinado con un pulso fuente de agua en el contorno superior. El gráfico de la figura 10 muestra la evolución de altura de agua en el punto central comparando las implementaciones del algoritmo en GPU y CPU.

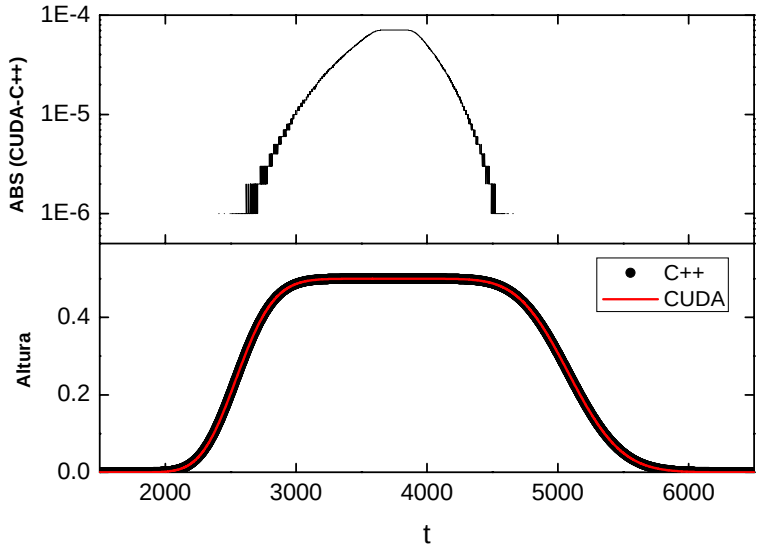

Figura 10: Altura de agua en la celda central (grilla de 512×512) de un plano inclinado con un pulso de agua en su borde superior. Se muestran las curvas de ambas implementaciones de AQUA en la parte inferior y la diferencia entre ambas en la parte superior en escala logarítmica.

5 PERFORMANCE

Los experimentos realizados en esta sección consisten un plano inclinado en el que inicialmente todas las celdas tienen el mismo nivel de agua y un factor de relajación α de 0,005. Estas condiciones fuerzan a que el algoritmo se ejecute por completo para todas las celdas de la grilla durante toda la simulación. La pendiente del plano se eligió de manera que quede perpendicular a las líneas de ejecución de los *threads* para que el movimiento del agua genere interacción entre estos últimos.

5.1 Resultados

Se realizaron sucesivas simulaciones de 5000 pasos de tiempo cada una sobre escenarios cuadrados calculando tiempos de ejecución mediante el *timer* provisto por la librería. Se tomaron en cuenta solamente los tiempos de ejecución para ambos ejemplos y no la creación de estructuras ni la copia de datos entre la memoria principal y la del dispositivo. Para cada tamaño de grilla se buscaron diferentes cantidades de bloques de *threads* para la implementación en CUDA como se detalló en la sección 4.3.

En la tabla 1 se muestran el tiempo de ejecución promedio de 1000 iteraciones para cada tamaño de grilla. La última columna indica la relación entre las implementaciones para CPU y para la placa gráfica.

Tamaño de la Grilla	**Implementación C++**	**Implementación en CUDA**		**Relación**
	Tiempo de CPU (1000 it.)	**Esquema de bloques y *threads***	**Tiempo de GPU (1000 it.)**	***(t CPU / t GPU)***
128×128	17.8	1 *b* × 128 *ths*	2.43	7.32
		4 *bs* × 64 *ths*	2.25	7.91
256×256	71.41	1 *b* × 256 *ths*	7.34	9.72
		16 *bs* × 64 *ths*	6.76	10.56
512×512	279.58	16 *bs* × 128 *ths*	24.07	11.61
		64 *bs* × 64 *ths*	23.65	11.68
1024×1024	961.62	16 *bs* × 256 *ths*	113.62	8.45
		64 *bs* × 128 *ths*	107.30	5.18
		256 *bs* × 64 *ths*	107.09	8.97
		1024 *bs* × 32 *ths*	72.61	13.24
		4096 *bs* × 16 *ths*	75.80	12.68
		16384 *bs* x 8 *ths*	111.19	8.64

Tabla 1: Comparación de tiempos de ejecución para las diferentes implementaciones según el tamaño de la grilla.

Si bien la ejecución en un solo bloque es más simple porque no requiere del paso de unión *UnionFlow()*, al dividir la ejecución en bloques de *threads* la cantidad de celdas que recorre cada uno se reduce y aumenta el nivel de paralelización. Es por esto que al aumentar la cantidad de bloques, y por lo tanto de *threads*, los tiempos de ejecución disminuyen para un mismo tamaño de grilla.

En la figura 11, se grafica el tiempo que demora promedio de una iteración completa en función de la dimensión de la grilla. La versión para CPU se muestra en negro mientras que la de GPU en rojo. Para este último caso se grafican los tiempos de las mejores configuraciones.

La figura 12 muestra la cantidad de celdas calculadas en un microsegundo, donde se puede observar claramente como este valor permanece prácticamente constante para la implementación de CPU, mientras que para la versión GPU el nivel de *throughtput* aumenta notablemente con el nivel de paralelización.

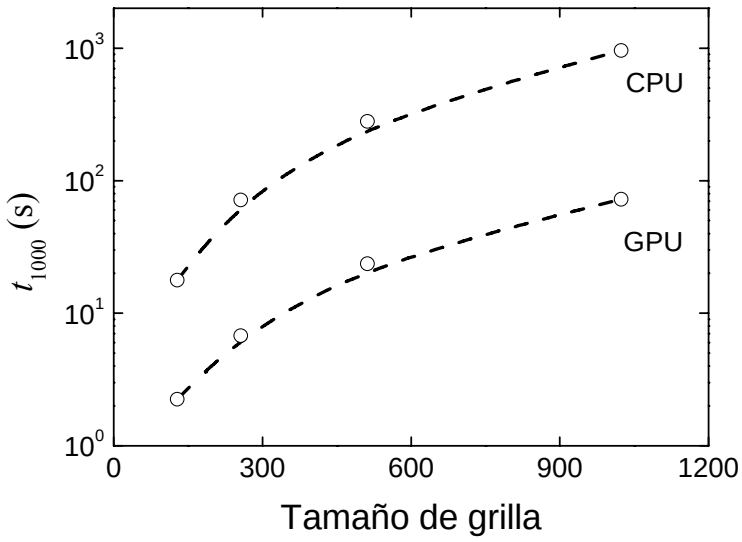

Figura 11: Comparación de tiempos de procesamiento (tiempo requerido para ejecutar 1000 pasos de tiempo).

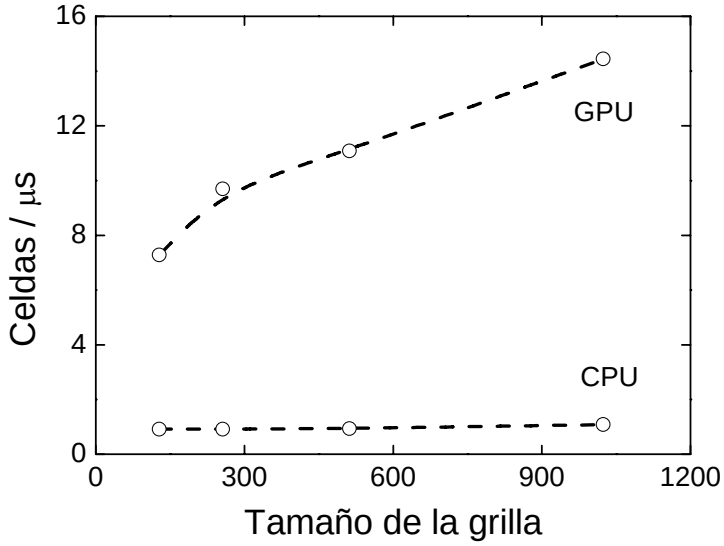

Figura 12: Comparación de *throughtput*.

5.2 Configuración ideal de bloques.

Se ensayaron en la GPU seis configuraciones diferentes de bloques y *threads* utilizando la grilla más grande (1024 × 1024). Se dividió la grilla en 16 bloques cuadrados cada uno con 256 *threads* que ejecutan sobre 256 celdas cada uno. A partir de ahí se fue aumentando la cantidad de bloques, y correspondientemente la cantidad de *threads*. En la última prueba se utilizaron 16384 bloques de 8 *threads* cada uno. En la figura 13 pueden verse los tiempos de

ejecución para 1000 iteraciones en función de la cantidad de *threads* utilizados. La configuración ideal es de 1024 bloques de 32 *threads* cada uno.

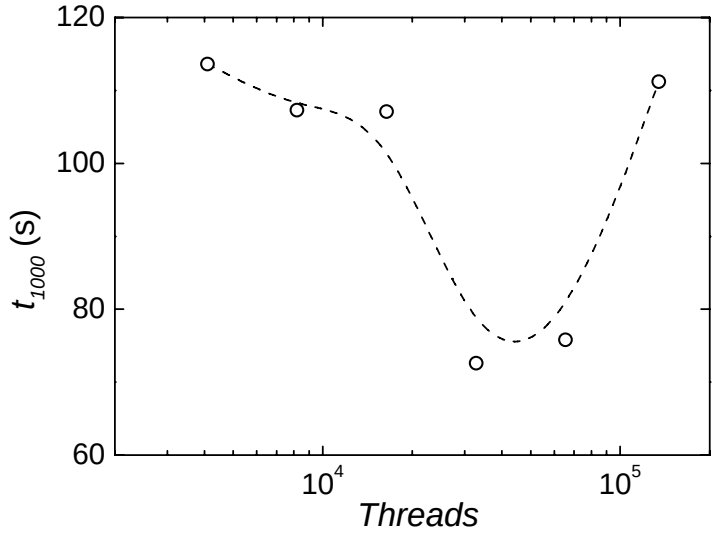

Figura 13: Comparación de tiempos de procesamiento de 1000 iteraciones de la GPU para diferentes configuraciones de *threads*.

6 CONCLUSIONES

Se presentó un estudio de performance de la tecnología de computación genérica sobre placas gráficas, mostrándose que es una técnica viable a la hora de acelerar procesos computacionales costosos y paralelizables como los Autómatas Celulares. La placa utilizada es una de las más económicas del mercado compatibles con la tecnología CUDA y destinada principalmente a PCs hogareñas, pero sin embargo se lograron diferencias en tiempos de ejecución mayores al orden de magnitud.

Diferentes autores dan configuraciones recomendadas u óptimas de bloques de *threads* sin embargo no siempre se comprobaron. Esto se debe fundamentalmente a particularidades del modelo y de las decisiones de implementación. El modelo AQUA tiene una muy baja complejidad computacional y por lo tanto no aprovecha al máximo el potencial de esta arquitectura pero sirve para analizar estrategias de implementación aplicables a otros modelos de AC como LBM. En trabajos futuros se prevé extender este estudio a cálculos numéricos más complejos.

7 REFERENCIAS

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

D.D. Dalponte, P.R. Rinaldi, M.J. Vénere, A. Clausse. Algoritmos de grafos y autómatas celulares: Aplicación al la simulación de escurrimientos. *Mecánica Computacional, Vol. XXIV*. ISSN 1666-6070. pp. 19. 2005.

S. Goldstein (Ed.). Modern Developments in Fluid Dynamics. Vol. I. Cap. VII: Flow in pipes and channels and along flat plates. Dover Publications, Inc. New York. 1965.

N. Goodnight. CUDA/OpenGL Fluid Simulation. [Online]. 2007. Disponible en: http://new.math.uiuc.edu/MA198-2008/schaber2/fluidsGL.pdf

K. Moreland, E. Angel. The FFT on a GPU. *SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware*, M. Doggett, W. Heidrich, W. Mark, and A. Schilling, Eds., Eurographics Association. San Diego, California. Eurographics Association. pp. 112–119. 2003.

NVIDIA, NVIDIA CUDA Programming Guide Version 1.0. 2007.

NVIDIA, NVIDIA CUDA Home Page. [Online]. 2008. Disponible en: http://www.nvidia.com/object/cuda_home.html

V. Podlozhnyuk. Histogram Calculation in CUDA. [Online]. 2007. Disponible en: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram256/doc/histogram.pdf

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

P.R. Rinaldi, D.D. Dalponte, M.J. Vénere, A. Clausse. Autómatas Celulares sobre Grafos de Nodos Irregulares: Aplicación a la Simulación de Escurrimientos Superficiales en Zonas de Llanura. *CACIC 2007*. Corrientes. 2007b.

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

M.J. Vénere, A. Clausse. A computational environment for water flow along floodplains. *International Journal on Computational Fluid Dynamics*. Vol. 16, pp. 327-330. 2002

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