CUDA en Mecánica Computacional

Santiago D. Costarelli[1] Mario A. Storti[1,2]

[1] CONICET-CIMEC

[2] Facultad de Ingeniería y Ciencias Hídricas,
Universidad Nacional de Litoral.

Curso de HPC en CIMEC - UNL/FICH
Santa Fe, Argentina.

Motivación

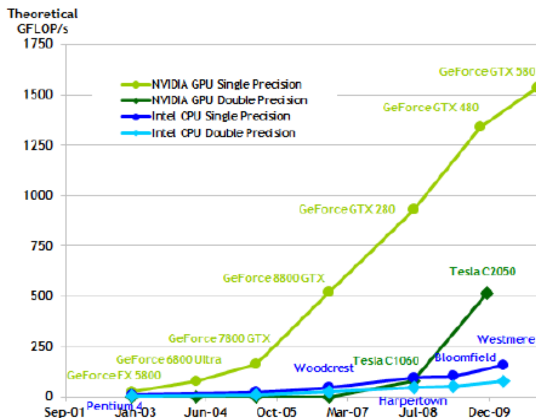

Figura 1: GFlops teóricos [?] obtenidos por diferentes GPGPUs[3] y CPU[4]. Resultados en simple y doble precisión.

[1] http://es.wikipedia.org/wiki/GPGPU

[2] http://es.wikipedia.org/wiki/Unidad_central_de_procesamiento

Motivación (cont.)

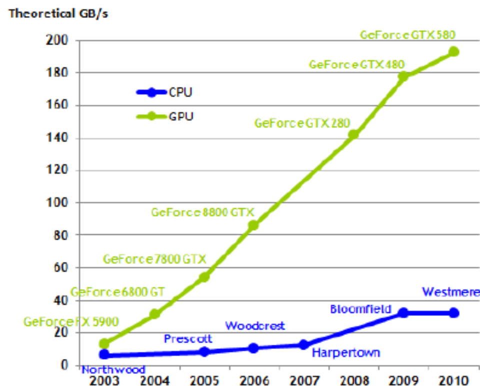

Figura 2: GBps teóricos [?] obtenidos por diferentes GPGPUs y CPU. Resultados en simple y doble precisión.

Motivación (cont.)

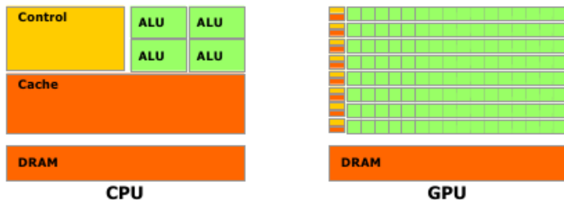

Figura 3: Comparación entre las arquitecturas CPU y GPGPU [?].

Sea el caso de la GPGPU, la clave esta en la observación de la enorme cantidad de ALUs[5] disponibles para cálculos a contrapartida de las unidades de control lógico en comparación a la CPU.

[5] http://es.wikipedia.org/wiki/Unidad_aritmético_lógica

¿Qué es CUDA entonces?

- CUDA[6] es un conjunto de herramientas, como ser un compilador, una API[7], entre otros, desarrollada por NVIDIA con el objetivo de facilitar la programación de GPGPUs.
- En esencia abstrae al programador de las dificultades inherentes a la programación de las mismas que, anteriormente, contaban con innumerables dificultades.
- El consenso y esfuerzo durante varios años derivaron en el estándar que hoy en día se conoce y que continúa evolucionando considerando la gran aceptación de los usuarios, en especial, de la comunidad científica.

---

[6] http://es.wikipedia.org/wiki/CUDA

[7] http://es.wikipedia.org/wiki/Interfaz_de_programación_de_aplicaciones

Introducción

- Se definirá entonces la asociación entre código Host, que podría correr en CPU, y device, que correrá preferentemente en GPU.
- De esta manera un código en CUDA presenta secciones que corren en CPU (como por ejemplo, la lectura de parámetros por consola, la definición de espacios de memoria, entre otros) y otras que correrán en GPU.
- El trabajo a ejecutar se modela con una grid que luego es subdividida en blocks, cuya unidad básica de intercambiando es el warp, y son estos los elementos que se derivan a los streaming multiprocessors (SM) de la GPGPU.
- Las ALUs vistas anteriormente se definen en CUDA como scalar processors (SP) y son parte constituyente del grupo SM.

División del trabajo

El dispositivo encargado de distribuir estos bloques es el driver de CUDA, sin embargo, es el warp scheduler el que subdivide estos bloques y confecciona los warps. Estos últimos son derivados a los SM para ser procesados (48 warps pueden ejecutarse en *simultáneo* en Fermi).

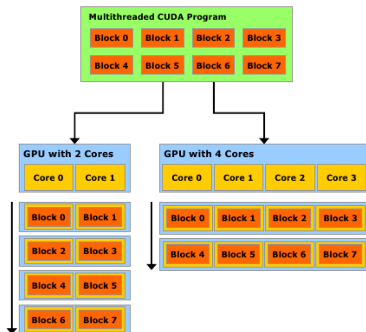

Figura 4: Esquema de división de trabajo en la arquitectura CUDA [?].

División del trabajo (cont.)

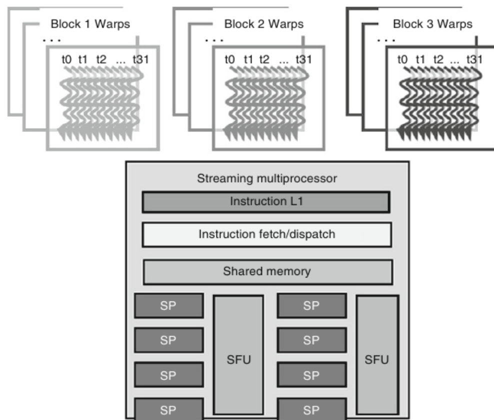

Figura 5: División de bloques en conjunto de threads denominados warps [?].

División del trabajo (cont.)

Existen básicamente dos escuelas para la división de trabajo. En la primera se recomienda

- disponer de bloques con dimensiones lo suficientemente grandes como para disponer de gran cantidad de warps y, poder así, esconder la latencia de ciertas operaciones intercambio warps;
- como contrapartida la cantidad de registros por thread puede resultar demasiado baja, y conllevar a pérdidas de eficiencia.

Existe, sin embargo, otro consenso [**?**] en el cual

- los bloques a emplear presentan dimensión menor y, de esta forma, la cantidad de registros por thread crece. Como en términos de performance, los registros son la situación óptima, esto podría resultar ventajoso;
- sin embargo no se disponen de suficientes warps para esconder posibles operaciones de gran latencia.

Jerarquía de memoria

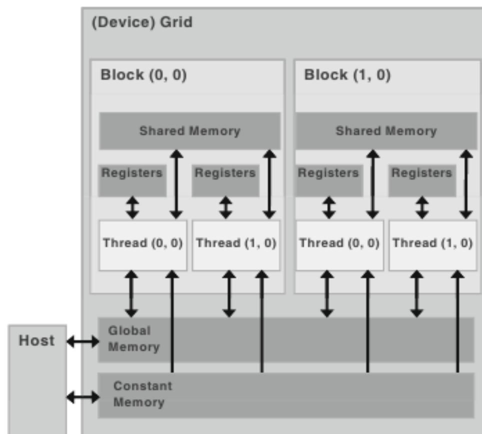

Figura 6: Jerarquía de memoria en la arquitectura CUDA [?].

Jerarquía de memoria (cont.)

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

Figura 7: Posibilidades de R/W en memorias en la arquitectura CUDA [?].

Es conveniente recordar de esta forma que

- las memorias on chip son más rápidas pero su capacidad es muy limitada;
- las memorias off chip presentan un rendimiento extremadamente inferior a las anteriores pero, como contrapartida, son mucho mas vastas.

Ejemplo 1

El ejemplo a resolver consiste en la ecuación de Laplace estacionaria en un dominio rectangular $D \in \Re^2 : \ [0;1] \times [0;1]$. De esta forma, término dependiente $\phi(\mathbf{x})$ podría representar temperatura, concentración, entre otros.
El problema a resolver resulta entonces

$$\Delta\phi(\mathbf{x}) = 0,$$

sujeto a las condiciones de contorno

$$\begin{aligned} \phi|_{x=0} &= f(y), \\ \phi|_{x=L} &= 0, \\ \phi|_{y=0} &= 0, \\ \phi|_{y=H} &= 0, \end{aligned}$$

donde $\mathbf{x} = (x, y)$ hace referencia a la posición espacial.

Ejemplo 1 (cont.)

La solución analítica al problema siendo

$$\phi(\mathbf{x}) = \sum_{m=1}^{\infty} c_m \; sin\left(\frac{m\pi y}{H}\right) sinh\left[\frac{m\pi\,(x-L)}{H}\right],$$

y como propuesta se tiene que

$$f(y) = sin\left(\frac{\pi y}{H}\right),$$

de esta forma los coeficientes c_m quedan definidos por

$$c_m = \frac{H \; sin\left[\frac{(1-m)\pi}{H} y\right]}{2(1-m)\pi} - \frac{H \; sin\left[\frac{(1+m)\pi}{H} y\right]}{2(1+m)\pi}.$$

Las curvas de nivel de la solución son presentadas en la Figura (8).

Ejemplo 1 (cont.)

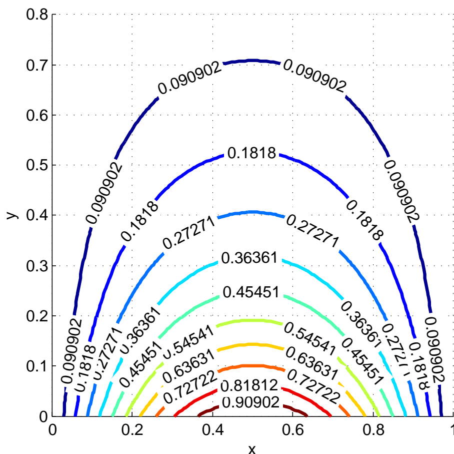

Figura 8: Solución analítica al problema propuesto.

Ejemplo 1 (cont.)

Para la discretización se utilizarán diferencias finitas centradas de segundo orden con paso de malla constante ($\Delta x = \Delta y = h = L/N = H/N$ donde N hace referencia a la cantidad de nodos por dirección).
De esta forma, el término h^2 de la ecuación puede ser tirado y lo que resta presenta la forma

$$\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i,j+1} - 4\phi_{i,j} = 0,$$

es decir, el típico stencil estrella de 5 nodos. La notación introducida haciendo referencia a $\phi_{i,j} = \phi(ih, jh)$.

Respecto a las condiciones de contorno, y considerando que i y j varían libremente, se tiene que $\phi_{i,N} = \phi_{N,j} = \phi_{i,0} = 0$, mientras que $\phi_{0,j} = f(jh)$.

Ejemplo 1 (cont.)

Finalmente, del ensamble de las ecuaciones nodales se obtiene un sistema de ecuaciones de la forma $\mathbf{A}\phi = \mathbf{b}$ que deberá ser resulto.

Un análisis detallado del problema propuesto muestra que la eficiencia del algoritmo se corresponde únicamente con aquella obtenida por el método que resuelve el sistema lineal.

En particular, la arquitectura CUDA tiene como principal benefactor a los algoritmos del tipo autómata-celular[8], que en esencia son aquellos en donde existe localidad en los datos. En otras palabras, para computar un determinado nodo, sólo una *pequeña* cantidad de nodos vecinos son requeridos.

[8] http://es.wikipedia.org/wiki/Autómata_celular

Ejemplo 2

Se considerará entonces el caso $\phi\left(\mathbf{x};t\right)$ gobernado por

$$\frac{\partial\phi}{\partial t}=\Delta\phi\left(\mathbf{x}\right),$$

sujeto a las condiciones de contorno

$$\begin{aligned}\phi|_{x=0}&=0,\\ \phi|_{x=L}&=0,\\ \phi|_{y=0}&=0,\\ \phi|_{y=H}&=0,\end{aligned}$$

y a la condición inicial

$$\phi|_{t=0}=\phi_0\left(\mathbf{x}\right).$$

Ejemplo 2 (cont.)

La solución analítica al problema siendo

$$\phi(\mathbf{x}) = \sum_{m=1}^{\infty} c_m exp\left(-\beta_m^2 t\right) sin\left(\frac{m\pi x}{L}\right) sin\left(\frac{m\pi y}{H}\right),$$

y como propuesta se tiene que

$$\phi_0(\mathbf{x}) = sin\left(\frac{\pi x}{L}\right) sin\left(\frac{\pi y}{H}\right),$$

de esta forma los coeficientes c_m quedan definidos por

$$c_m = \frac{4}{LH}\int_0^L sin\left(\frac{\pi x}{L}\right) sin\left(\frac{m\pi x}{L}\right) dx \int_0^H sin\left(\frac{\pi y}{H}\right) sin\left(\frac{m\pi y}{H}\right) dy,$$

resultados que podemos obtener en base a tablas, por ejemplo, usando

$$\int sin(ax)sin(bx)\ dx = \frac{sin[(a-b)x]}{2(a-b)} - \frac{sin[(a+b)x]}{2(a+b)}, \quad a^2 \neq b^2.$$

Ejemplo 2 (cont.)

Usando una discretización temporal de Forward Euler, se obtiene

$$\frac{\phi_{ij}^{n+1} - \phi_{ij}^{n}}{\Delta t} = (\Delta\phi)_{ij}^{n}\,,$$

y de esta forma

$$\phi_{ij}^{n+1} = Fo\{\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i,j+1} - 4\phi_{i,j}\} + \phi_{ij}^{n},$$

donde Fo[9] es el número adimensional de Fourier, definido como

$$Fo = \frac{\nu\Delta t}{h^2},$$

donde se ha usado $\nu = 1$. Se ve entonces el típico esquema de autómatas-celulares, para avanzar en el tiempo la posición (i, j) sólo requiere consultar a sus vecinos más cercanos.

[9] El límite de estabilidad resulta $\nu\Delta t/h^2 \leq 1/2$.

Código CUDA

Se usará para la implementación los contenedores y algoritmos provistos por Thrust[10]. De esta forma la creación/destrucción de vectores, tanto en el host como en el device, son tratados naturalmente.

Finalmente se presentarán dos resoluciones

- la primera resuelve el problema haciendo uso de la memoria global,
- la segunda consiste en resolver el problema utilizando la memoria shared.

[10] http://code.google.com/p/thrust/

Código CUDA - Consideraciones generales

Con el objetivo de evitar las extensas definiciones de tipos de datos, una opción es definir tipos de datos propios con menor extensión.

```
1 typedef float ScalarType;
2 typedef typename thrust::tuple<ScalarType, ScalarType> tuple;
3 typedef typename thrust::host_vector  <ScalarType>   host_vector;
4 typedef typename thrust::device_vector<ScalarType> device_vector;
```

Código 1: Definiendo tipos de datos.

Se ve allí, la definición de vectores tanto del lado host como del device. De esta forma vectores en CPU y GPU serán definidos y tratados indistintamente.

Código CUDA - Consideraciones generales (cont.)

Se definen inicialmente los parámetros de gobierno.

```
/**
   Parametros
   L : dimension fisica (en la direccion x) de la geometria.
   H : dimension fisica (en la direccion y) de la geometria.
   N : numero de nodos por direccion.
   h : paso de la malla, en este caso constante.
   dt: paso de tiempo.
   nu: viscosidad cinematica asumida como 1.
   t : contador de tiempo.
   T : tiempo final.
*/

ScalarType L = 1;
ScalarType H = 1;
assert(L == H);

int N = 16;
assert(N > 1);

ScalarType h = L / (ScalarType)(N - 1);

ScalarType dt = 0.25 * h * h / 1. * 0.9, t = 0, T = 0.26;

// Fourier.
ScalarType Fo = 1 * dt / h / h;
assert(Fo <= 0.5);
```

Código 2: Definiendo parámetros del problema.

Código CUDA - Consideraciones generales (cont.)

Se definen ahora los vectores con los ejes coordenados donde se hace uso de dos funciones provistas por Thrust

- thrust::sequence toma un vector, un valor inicial y un paso y genera un vector cuyas sucesivas entradas difieren en la cantidad paso;
- thrust::copy es similar al copy estándar de las STL[11].

Luego se cargan las condiciones iniciales, del lado host, y luego se realiza la copia al device. Podría iniciarse directamente del lado del device también.

```
1   // Vectores con ejes coordinados.
2   host_vector x (N), y (N);
3   thrust::sequence(x.begin(), x.end(), (ScalarType)0, h);
4   thrust::copy(x.begin(), x.end(), y.begin());
5
6   // Cargando condicion inicial.
7   host_vector hphi (N*N);
8   initial_condition(x, y, hphi, N, L, H);
9
10  device_vector dphi     (N * N),
11                dphi_new(N * N);
12  thrust::copy(hphi.begin(), hphi.end(), dphi.begin());
```

Código 3: Definiendo ejes y condiciones iniciales.

[11] http://en.wikipedia.org/wiki/Standard_Template_Library

Código CUDA - Consideraciones generales (cont.)

Ahora se define el tamaño de la grilla y los bloques que conformaran las mismas. Esto se define con una estructura propia de CUDA, dim3, que para este caso consisten en $N \times N$ y $BX \times BY$, respectivamente. Los valores BX y BY son escogidos por el usuario y son cruciales en cualquier código en CUDA.

```
1  // Dimensiones del bloque.
2  dim3 dimBlock(BX, BY);
3
4  // Dimensiones (en bloques) de la grilla.
5  dim3 dimGrid( (N + BX - 1 ) / BX, (N + BY - 1) / BY);
```

Código 4: Definiendo los parámetros de CUDA del problema.

Código CUDA - Consideraciones generales (cont.)

A continuación se define el bucle temporal, que corresponde a la integración, y se itera hasta un valor de tiempo prefijado, $t = T$.

```
1   // Lazo temporal.
2   while (t < T)
3     {
4       // Calculando nuevo paso de tiempo.
5       global_kernel(dimBlock, dimGrid, dphi, Fo, N, dphi_new);
6
7       // Copiando informacion al vector de incognitas.
8       thrust::copy(dphi_new.begin(), dphi_new.end(), dphi.begin());
9
10      // Incrementamos el contador de tiempo.
11      t += dt;
12    }
```

Código 5: Lazo temporal.

Lo importante aquí es la función global_kernel, ella implementa la versión del código en memoria global.

Código CUDA - Consideraciones generales (cont.)

Con el objetivo de utilizar vectores de Thrust dentro de kernels de CUDA es necesario realizar casteos a arreglos de C. Es así como se utiliza la función thrust::raw_pointer_cast provista por Thrust, ella retorna un puntero que puede ser pasado como argumento a los kernels de CUDA.

```
1  void global_kernel (const dim3 & dimBlock,
2                      const dim3 & dimGrid,
3                      const device_vector & phi_old,
4                      const ScalarType Fo,
5                      const int N,
6                      /* */ device_vector & phi_new)
7  {
8    /**
9       Los kernels de CUDA necesitan que los vectores de Thrust
10      esten casteados como un vector de C, para ello se usa
11      'thrust::raw_pointer_cast'.
12   */
13   global_kernel_unwrapped<<<dimGrid,dimBlock>>>(thrust::raw_pointer_cast(&phi_old[0]),
14                                                  Fo,
15                                                  N,
16                                                  thrust::raw_pointer_cast(&phi_new[0]));
17   cudaError err = cudaGetLastError();
18   if (cudaSuccess != err) throw std::runtime_error(cudaGetErrorString(err));
19 }
```

Código 6: El wrapper de global kernel.

Código CUDA - Memoria global

Cada thread dispone de una tupla de valores que lo identifica univocamente dentro del conjunto. Estos son obtenidos haciendo uso de las estructuras provistas por CUDA

- blockIdx
- blockDim
- threadIdx

```
1  const int gx = blockIdx.x * blockDim.x + threadIdx.x;
2  const int gy = blockIdx.y * blockDim.y + threadIdx.y;
```

Código 7: Definiendo cada thread dentro del conjunto.

Código CUDA - Memoria global (cont.)

Es posible que las dimensiones del bloque no sean un múltiplo de las dimensiones globales del problema. Es así como se hace necesario un chequeo para conocer si existen threads fuera de dominio computacional pues estos, inicialmente, no tendrían porqué computar valor alguno.

```
1 // Chequeo de sanidad.
2 if (gx >= N || gy >= N) return;
```

Código 8: Chequeo de sanidad.

Finalmente se computa el nuevo valor en función de los anteriores.

Código CUDA - Memoria global (cont.)

Respecto a las macros se tiene que

- GRID permite utilizar tratamiento 2D sobre un arreglo 1D, es decir, se pueden utilizar dos indices para hacer referencia a un dato virtualmente bidimensional;
- CHECKBC asigna el valor 0 al campo ϕ si el valor del campo en el contorno es solicitado.

```
#define GRID(gx, gy) ( (gy) * N + (gx) )
#define CHECKBC(phi_old, gx, gy, N) \
  ( ( (gx) < 0 || (gy) < 0 || (gy) >= (N) || (gy) >= (N) ) ? 0 : (phi_old)[GRID((gx), (gy))] )

  // Stencil de 5 puntos.
  phi[GRID(gx, gy)] = Fo * (
    CHECKBC(phi_old, gx - 1, gy    , N) + 1 *
    CHECKBC(phi_old, gx + 1, gy    , N) + 1 *
    CHECKBC(phi_old, gx    , gy - 1, N) + 1 *
    CHECKBC(phi_old, gx    , gy + 1, N) - 4 *
    CHECKBC(phi_old, gx    , gy    , N)
    ) +
    CHECKBC(phi_old, gx    , gy    , N) ;

  if (gx == 0 || gy == 0 || gx == N - 1 || gy == N - 1)
    phi[GRID(gx, gy)] = 0; // Condicion de contorno.

#undef CHECKBC
#undef GRID
```

Código 9: Cómputo global.

Código CUDA - Memoria global (cont.)

Algunos problemas que pueden enumerarse son los siguientes

- si bien se hace uso de la posibilidad de correr una masiva cantidad de threads para resolver el problema, existe una gran cantidad de accesos redundantes;
- es conocido que la memoria global tiene un rendimiento muy por debajo del otorgado por la memoria shared o bien por el brindado por los registros.

De esta forma, una versión en memoria shared elimina los problemas enumerados anteriormente.

Código CUDA - Memoria shared

La memoria shared es bastante acotada, para una GPGPU Fermi [?] se tienen entre 16/48 KB que pueden ser escogido por el usuario[12]. La elección de uno o otro caso dependerá del problema a tratar.

Además se tiene que la cantidad de registros también es bastante limitada, para el caso Fermi se tienen 32768.

Tratar con la memoria shared requiere de un entendimiento de la arquitectura CUDA que escapa de la corriente presentación, más el lector interesado se puede instruir en temas como: organización en bancos de memoria, conflicto de bancos, entre otros.

[12] Con la directiva -Xptxas -dlcm.

Código CUDA - Memoria shared (cont.)

La memoria shared define un nuevo espacio de memoria, es así como se necesita una tupla de índices que identifique una dada posición, tal y como fue realizado para el caso de memoria global.

```
1  // Indices.
2  const int gx = blockIdx.x * blockDim.x + threadIdx.x;
3  const int gy = blockIdx.y * blockDim.y + threadIdx.y;
4
5  // Chequeo de sanidad.
6  if (gx >= N || gy >= N) return;
7
8  const int tx = threadIdx.x + 1;
9  const int ty = threadIdx.y + 1;
```

Código 10: Definiendo tuplas que determinen posiciones tanto en la memoria global como en la shared.

De esta forma se tienen identificadores

- globales, como gx y gy;
- locales, como tx y ty.

Código CUDA - Memoria shared (cont.)

Se define entonces el espacio de memoria, observar el tipo __shared__ utilizado, propio de CUDA, indicando que el arreglo a continuación esta definido en el espacio de memoria shared.

```
1  // Se considera una capa de HALO a cada lado.
2  __shared__ ScalarType smem [1 + BY + 1][1 + BX + 1];
```

Código 11: Definiendo un arreglo en memoria shared.

Tal y como se ha utilizado anteriormente en Life, existe un HALO, que no es más que una capa de nodos extra que permite computar a los nodos en la frontera del bloque.

Código CUDA - Memoria shared (cont.)

```
/**
   El acceso se hace de esta forma para evitar ineficiencias
   en el acceso a la memoria shared.
 */
smem[ty][tx] = phi_old[GRID(gx, gy)];

// Sincronizacion.
__syncthreads();

// Cargando HALO.
if      (tx == 1 )
  {
    smem[ty    ][0      ] = (gx == 0    ) ? 0 : phi_old[GRID(gx - 1, gy    )];
  }
else if (tx == BX)
  {
    smem[ty    ][BX + 1] = (gx == N - 1) ? 0 : phi_old[GRID(gx + 1, gy    )];
  }
if      (ty == 1 )
  {
    smem[0     ][tx     ] = (gy == 0    ) ? 0 : phi_old[GRID(gx    , gy - 1)];
  }
else if (ty == BY)
  {
    smem[BY + 1][tx     ] = (gy == N - 1) ? 0 : phi_old[GRID(gx    , gy + 1)];
  }

// Aqui no se requiere barrera de sincronizacion.
```

Código 12: Cargando los datos en la memoria shared.

Código CUDA - Memoria shared (cont.)

Finalmente se computa, similar al caso estudiado con memoria global.

```
// Stencil de 5 puntos.
phi[GRID(gx, gy)] = Fo * ( smem[ty    ][tx - 1] +
                           smem[ty    ][tx + 1] +
                           smem[ty - 1][tx    ] +
                           smem[ty + 1][tx    ] - 4 * smem[ty    ][tx    ]
                         ) +
                    smem[ty    ][tx    ];

if (gx == 0 || gy == 0 || gx == N - 1 || gy == N - 1)
  phi[GRID(gx, gy)] = 0; // Condicion de contorno.
```

Código 13: Computando.

Validación

La validación del código se muestra en las Figuras (**??**) y (**??**), donde se utilizó $N = 16$ y $\Delta t = 1{,}0 \times 10^{-3}$.

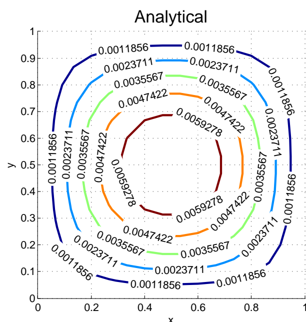

Figura 9: Solución analítica al problema propuesto.

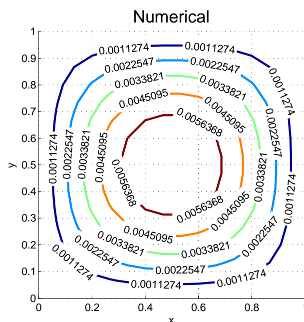

Figura 10: Solución numérica al problema propuesto.

Performance obtenido

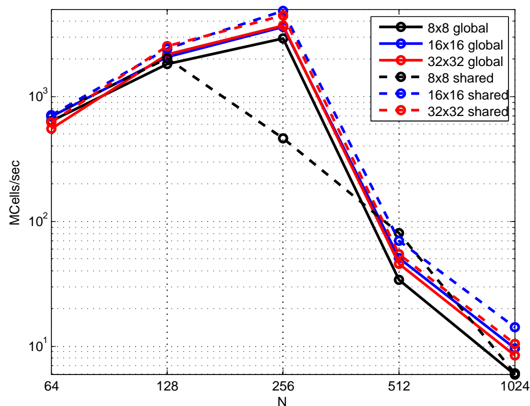

Figura 11: Performance obtenido[14] por la implementación bajo los ejemplos propuestos, global y shared, variando el tamaño del bloque. Simple precisión.

[13] Usando la estructura provista por CUDA cudaEvent_t.

Utilización del profiler de CUDA

NVIDIA provee una herramienta gráfica muy interesante, denominada nvvp[15], cuyo objetivo es brindar información de rendimiento de la aplicación analizada.
Entre los cuales se pueden numerar

- accesos fusionados [**?**], o coalesced, de la memoria global. Si el rendimiento resultara bajo, deberían de revisarse los accesos a memoria global;
- hits al caché (ya sea L1 o L2), pues de no resultar significativos, podría darse de baja el caché para evitar que se realice una búsqueda allí;
- ocupación, u occupancy, de los SM disponibles;
- divergencia de threads, en esencia se da cuando threads de un mismo warp ejecutan diferentes caminos en un condicional. De esta forma es necesario leer más instrucciones lo que va a contramano del modelo SIMT.

[15] NVIDIA Visual Profiler.

Actividades propuestas

- Ejercicio 1: Implementar[16] un método, directo o iterativo, que resuelva lo planteado en el *Ejemplo 1*.
- Ejercicio 2: Modificar lo presentado en el *Ejemplo 2* para poder introducir otros esquemas de integración temporal[17]. Luego realice pruebas con esquemas explícitos de un paso y multipasos.
- Ejercicio 3: Realizar una gráfica de performance similar a la presentada anteriormente pero considerando dobles como tipo escalar. Los resultados obtenidos ¿Mantienen la performance del caso en simple precisión? De no ser así, justifique debidamente[18].
- Ejercicio 4: Utilizando nvvp, ¿Podría indicar alguna optimización al ejercicio propuesto que utiliza memoria shared? De ser así justifique.

[16] http://code.google.com/p/cusp-library/

[17] Investigue y haga uso de functores usando Thrust.

[18] La relación de performance en doble precisión respecto a la de simple se enuncia en los documentos oficiales de NVIDIA como un cociente.

Bibliografía

C CUDA.
Best practices guide.
NVIDIA Corporation, 2012.

David B Kirk and W Hwu Wen-mei.
Programming massively parallel processors: a hands-on approach.
Morgan Kaufmann, 2010.

CUDA Nvidia.
Nvidia cuda programming guide, 2011.

Vasily Volkov.
Better performance at lower occupancy.
In *Proceedings of the GPU Technology Conference, GTC*, volume 10, 2010.

Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu.
Fermi gf100 gpu architecture.
Micro, IEEE, 31(2):50–59, 2011.