Introducción a la programación de códigos paralelos con CUDA y su ejecución en un GPU multi-hilos

Gerardo A. Laguna-Sánchez,[*] Mauricio Olguín-Carbajal,[**] Ricardo Barrón-Fernández[***]

Recibido: 02 de mayo de 2011
Aceptado: 23 de mayo de 2011

Abstract
CUDA, a relatively new parallel programming tool, allows the programmers to apply the very cheap and virtually ubiquitous Graphics Processing Units (GPU) to deal with generic purpose applications, while running on common desk computers. This big computing power may be used to parallelize sequential codes by mean C style codes, in a relatively simply and straightforward way. In this paper, a basic introduction to parallel programming of NVIDIA GPUs with CUDA is presented. Finally, some practical considerations are discussed in order to reach the best GPU performance and to justify its use.

Keywords: Parallel programming, CUDA, GPU.

Resumen
Gracias a CUDA, una herramienta de programación paralela de aparición relativamente reciente, es posible escribir códigos paralelos de propósito general y ejecutarlos en los dispositivos de procesamiento de gráficos (GPU) de ciertas tarjetas gráficas, relativamente económicas y prácticamente omnipresentes, que operan en computadoras convencionales. Así, es posible aprovechar el gran poder computacional de estas tarjetas gráficas, mediante la paralelización de algoritmos secuenciales, de una manera relativamente simple, sobre todo para los programadores familiarizados con el lenguaje C. En este artículo se presenta una introducción básica a la herramienta CUDA, para la programación paralela en dispositivos GPU de marca NVIDIA, y se dan recomendaciones prácticas, tanto para garantizar el máximo aprovechamiento del GPU como para justificar el empleo del mismo.

[*]Departamento de Ingeniería Eléctrica, UAM-Iztapalapa, México, D.F. glaguna@xanum.uam.mx
[**]Centro de Investigación y Desarrollo Tecnológico (IPN).
[***]Centro de Investigación en Computación (IPN) Zacatenco, México, D.F.

Palabras clave: Programación paralela, CUDA, GPU.

Introducción

Recientemente, se propuso explotar el poder computacional disponible en las tarjetas gráficas de las computadoras personales a fin de solucionar problemas de propósito general (Owens y cols., 2007), surgiendo, con ello, la idea del GPU para procesamiento de propósito general (GPGPU, por sus siglas en inglés). Desde entonces, tanto los fabricantes como los desarrolladores, han considerado esta nueva aplicación de la computación como una prometedora área de investigación, sobre todo, por la amplia variedad de posibles aplicaciones que pueden aprovechar el paralelismo disponible en los actuales dispositivos GPU.

Los dispositivos GPU modernos tienen su origen en la arquitectura del procesador vectorial, que permite la ejecución simultánea de operaciones matemáticas sobre datos múltiples. En contraste, los procesadores de los CPU convencionales no pueden manejar más de una operación al mismo tiempo. Al principio, los procesadores vectoriales eran comúnmente usados en las computadoras científicas (Mzoughi, Lafontaine y litaize, 1996), pero más tarde fueron desplazados por arquitecturas de múltiples núcleos. Sin embargo, los procesadores vectoriales no fueron completamente eliminados ya que, en esencia, muchas de las arquitecturas diseñadas para realizar gráficas por computadora, como es el caso de los dispositivos GPU modernos, están inspirados por ellos.

Por lo general, un procesador convencional tiene que buscar la siguiente instrucción que va a ejecutar, lo que consume tiempo y genera cierta latencia durante la ejecución de las instrucciones. Para reducir la latencia, los procesadores modernos ejecutan un conjunto de instrucciones en forma paralela, lo que se conoce como "entubamiento de instrucciones" (*pipelining*). En un ciclo de máquina convencional se sigue la secuencia "traer-decodificar-ejecutar" (*fetch-*

decode-execute), en cambio, con la técnica de entubamiento de instrucciones, la decodificación de la siguiente instrucción inicia antes de que se haya concluido la ejecución de la primera, de tal manera que el decodificador de instrucciones es constantemente utilizado y, con ello, disminuye la latencia. En estas condiciones, el tiempo de ejecución para un conjunto de instrucciones es menor que en un procesador convencional, aumentando así el rendimiento total del procesador.

Los procesadores vectoriales llevan la misma idea más lejos, al realizar el entubamiento tanto de las instrucciones como de los datos. En este caso, una sola instrucción opera en muchos elementos de datos, lo que ahorra tiempo en la decodificación de instrucciones y produce un gran poder de cálculo. Considerando que, sobre todo durante la realización de gráficas por computadora, las imágenes tienen una representación natural con el formalismo de las matrices y que es deseable contar con la capacidad de realizar operaciones paralelas sobre múltiples datos, se comprende que los procesadores vectoriales hayan sido ampliamente adoptados para resolver estas exigencias de un modo directo. Lo expuesto arriba explica la gran aceptación de los procesadores vectoriales en el campo de realización de gráficas por computadora y su influencia en el desarrollo de los modernos dispositivos GPU, destinados a satisfacer la gran demanda del poder computacional requerido para el procesamiento de imágenes en las más recientes aplicaciones de vídeo-juegos.

Arquitectura del GPU de marca NVIDIA

La tendencia actual, observada tanto en el desarrollo como en la aplicación de los dispositivos GPU ofrecidos por el fabricante NVIDIA, permite anticipar la consolidación del nuevo modelo de programación paralela soportado por la herramienta de programación CUDA, donde el GPU, además de ofrecer una mayor capacidad de cálculo paralelo, tiene un papel preponderante como administrador de múltiples hilos (*NVIDIA CUDA C Programming Guide, Version 3.2*, 2010). Este GPU proporciona una arquitectura unificada, tanto para gráficos como para cálculos, que NVIDIA llama arquitectura Tesla y que es fundamentalmente un arreglo escalable de multiprocesadores multi-hilo.

Cada multiprocesador consiste en ocho núcleos de procesamiento, una unidad de instrucción multihilos y memoria compartida, todo dentro del circuito integrado (*on-chip*) (ver Figura 1). Cada multiprocesador se encarga de la creación, manejo y ejecución de los hilos que están físicamente activos en el dispositivo GPU, manejando cientos de hilos (teóricamente hasta 512) con el esquema denominado por NVIDIA como "una instrucción y múltiples hilos" (SIMT, por sus siglas en inglés) (*NVIDIA CUDA C Programming Guide, Version 3.2*, 2010). Debido a que un multiprocesador asigna cada uno de sus hilos a un núcleo y que cada hilo es ejecutado independientemente de los demás, con su propia dirección de instrucción y sus propios registros de estado, las herramientas de programación de NVIDIA ofrecen algunas funciones que son enfocadas precisamente al manejo y optimización de múltiples hilos.

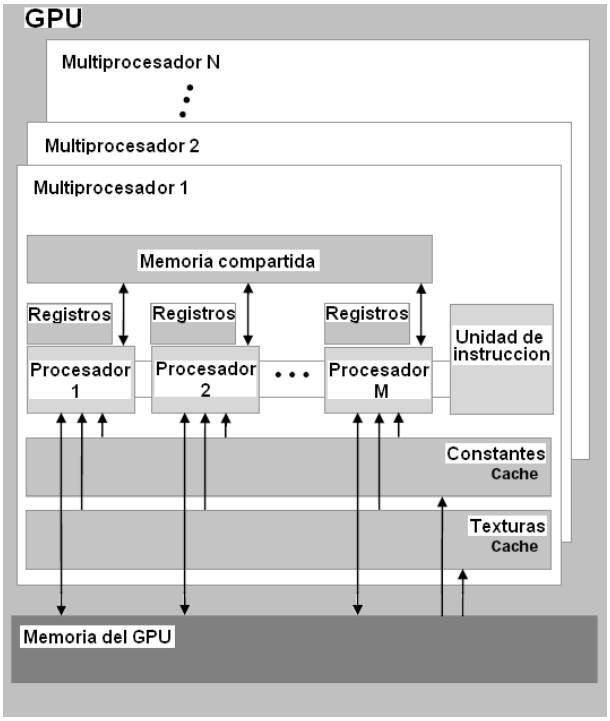

Figura 1. Arquitectura del GPU NVIDIA empleado

Arquitectura CUDA

La herramienta de programación conocida como CUDA está diseñada para soportar el esquema de "una instrucción y múltiples hilos" (SIMT), de tal manera que múltiples hilos pueden ser ejecutados sobre muchos datos. La herramienta CUDA permite que los programadores escriban el código paralelo, usando lenguaje C estándar más algunas extensiones de NVIDIA. La herramienta CUDA permite organizar el paralelismo en un sistema jerárqui-

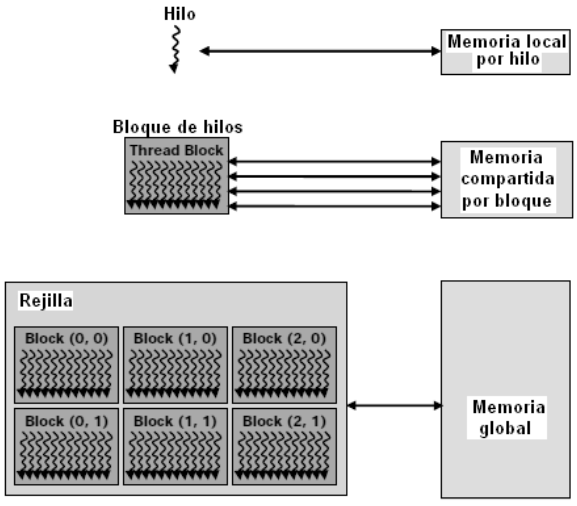

Figura 2. Jerarquía de la memoria del GPU NVIDIA

co de tres niveles: rejilla, bloque, e hilo. El proceso comienza cuando el procesador anfitrión (CPU anfitrión) invoca una función para el dispositivo GPU, llamada *kernel*, luego de lo cual se crea una rejilla (o arreglo) con bloques de múltiples hilos, para distribuirla en algún multiprocesador disponible. Con CUDA, dentro del programa se arranca la ejecución de los kernels paralelos mediante la siguiente sintaxis extendida de llamada a función:

```
kernel <<<dimGrid, dimBlock>>>
(...parameters...);
```

donde `dimGrid` y `dimBlock` son parámetros especializados que especifican, respectivamente, la dimensión (en bloques) de la rejilla de procesamiento paralelo y la dimensión (en hilos) de los bloques.

Durante la ejecución del kernel, los hilos tienen acceso a seis tipos de memoria dentro del dispositivo GPU, según la siguiente jerarquía (o niveles de acceso) predefinidos por NVIDIA (ver Figura 2):

- Memoria global. Es una memoria de lectura/escritura y se localiza en la tarjeta del GPU.
- Memoria para constantes. Es una memoria rápida (*cache*) de lectura y se localiza en la tarjeta del GPU.
- Memoria para texturas. Es una memoria rápida (*cache*) de lectura y se localiza en la tarjeta del GPU.
- Memoria local. Es una memoria de lectura/escritura para los hilos y se localiza en la tarjeta del GPU.
- Memoria compartida. Es una memoria de lectura/escritura para los bloques y se localiza dentro del circuito integrado del GPU.
- Memoria de registros. Es la memoria más rápida, de lectura/escritura para los hilos y se localiza dentro del circuito integrado del GPU.

La memoria compartida y los registros son los más rápidos, pero su tamaño está limitado porque es memoria ubicada dentro del circuito integrado. Por otra parte, la memoria localizada en la tarjeta del dispositivo (local, global, para constantes y para texturas) es grande pero presenta mayor latencia en los accesos, en comparación con la memoria alojada dentro del circuito integrado. Ya que el multiprocesador ejecuta paralelamente los hilos en grupos de 32, llamados "tejidos" (*warp*), los hilos pueden tener un acceso más eficiente a la memoria global, siempre que este acceso se realice en bloques de justo la "mitad de un tejido" (*half-warp*), mediante lecturas/escrituras simultáneas de/a memoria, fusionadas en una sola transacción en masa (*coalesced*) de memoria, ya sea de 32, 64, o 128 bytes ("NVIDIA CUDA C Programming Guide, Version 3.2", 2010).

Consideraciones para la puesta en práctica

Es muy importante hacer énfasis en que, debido a los tiempos, relativamente grandes, de retardo (latencia) y al bajo ancho de banda en las trasferencias de memoria, entre la computadora anfitrión y el dispositivo GPU, es altamente recomendable dividir a la aplicación, de tal manera que cada parte del sistema (hardware) haga únicamente el trabajo que mejor realiza. El uso del GPU es solamente recomendado si (*NVIDIA CUDA C Programming Guide, Version 3.2*, 2010; *CUDA C Best Practices Guide, Version 3.2*, 2010):

- La complejidad de las operaciones justifica el costo de mover datos, de y hacia el dispositivo GPU. El escenario ideal es aquél en el que muchos hilos ejecutan una cantidad considerable de trabajo. Entonces, ya que las transferencias deben ser minimizadas, los datos deberían mantenerse en le GPU tanto como sea posible.

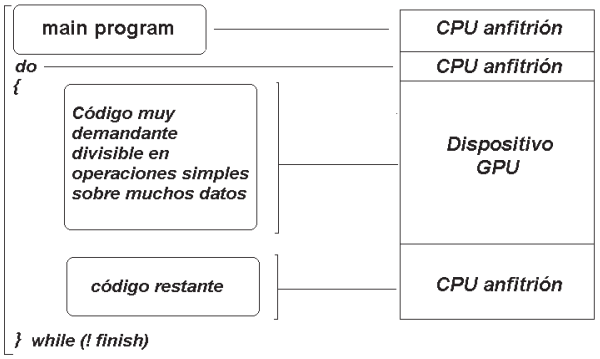

Figura 3. Estrategia de paralelización maestro-esclavo

- La aplicación tiene numerosos datos que pueden ser calculados simultáneamente en paralelo. Esto típicamente involucra operaciones aritméticas sobre un gran conjunto de datos, donde la misma operación puede ser realizada sobre miles de elementos al mismo tiempo.

- La aplicación puede ser dividida en operaciones simples, que pueden ser asignadas a numerosos hilos ejecutándose en paralelo.

- El tamaño de los tipos, usados para variables y arreglos dentro del código del GPU, son congruentes tanto con ciertos patrones de memoria como con las instrucciones aritméticas más ágiles, a fin de alcanzar el mejor desempeño del GPU (por ejemplo, el empleo de accesos de memoria en masa, así como el uso de tipos enteros, flotantes de precisión simple y funciones aritméticas intrínsecas compatibles).

Con estas consideraciones en mente, lo primero que se tiene que hacer es determinar cuál es la parte del código secuencial que se puede paralelizar mejor usando el GPU, de acuerdo con las recomendaciones de arriba. Típicamente se elige como posibles candidatos a todos los segmentos de código que son especialmente demandantes de recursos computacionales (tanto tiempo de procesamiento como memoria). Finalmente, se procede a delegar al GPU aquellos segmentos de código que cumplen con las recomendaciones prácticas antes mencionadas (ver Fig. 3).

Típicamente, una aplicación y código con CUDA debe incluir los siguientes pasos:

1. El computador (CPU) anfitrión llama al cuerpo principal del programa (`main()`).

2. Se reserva memoria dentro del dispositivo GPU.

3. Se copian los datos del CPU anfitrión al dispositivo GPU.

4. El CPU anfitrión llama a la función kernel.

5. El dispositivo GPU ejecuta el código paralelamente.

6. Se copian los resultados de vuelta a la memoria del CPU anfitrión.

7. Se libera la memoria reservada dentro del dispositivo GPU.

La aplicación, así conformada, delega al GPU la ejecución paralela de tantos hilos como lo determina el tamaño de la rejilla de bloques y el tamaño de cada bloque de hilos. Concretamente, cuando se arranca la ejecución del kernel se define tanto el número de hilos por bloque (`blockDim`) como el número de bloques que conforman a la rejilla (`gridDim`). La multiplicación de estos dos parámetros resulta en el total de hilos que serán administrados por el multiprocesador que el sistema designó para la ejecución del kernel.

En todo esto, se asume que el multiprocesador puede ejecutar concurrentemente, cuando mucho, el número de hilos en un solo bloque. Sin embargo, el multiprocesador puede planificar la ejecución de un número total de hilos que rebase, por mucho, al tamaño de un solo bloque. Esto lo logra distribuyendo el trabajo de todos los bloques, durante los intervalos de tiempo disponibles, conforme se va presentando la oportunidad.

Por ejemplo, para realizar una misma operación aritmética, a todo lo largo de un arreglo de tamaño variable, se puede usar un número fijo de hilos si se emplea el código genérico mostrado en la Fig. 4. En este código, todos los hilos ejecutados concurrentemente al mismo tiempo se identifican tanto por el índice `threadIdx` como por el identificador del bloque vigente `blockId`, así que al mismo tiempo se podrán actualizar `blockDim` elementos del arreglo. Si el identificador del bloque se emplea como factor para la constante de desplazamiento `blockDim`, entonces el mismo número de hilos actualizarán, a la primera oportunidad, otro bloque de

```
static __global__ void PointwiseAritmeticOperation( Array1,Array2, size_of_arrays)
{
   const int numThreads = blockDim.x * gridDim.x;
   const int threadID = blockIdx.x * blockDim.x + threadIdx.x;
   for (int i = threadID; i < size_of_arrays; i += numThreads)
      Array1[threadID] = AritmeticOperation(Array1[threadID],Array2[threadID]);
}
```

Figura 4. Código paralelo genérico para kernel aritmético

igual dimensión. Aún más, si el número total de hilos de la rejilla no fuera suficiente, por sí solo, para el total de elementos del arreglo, no hay problema porque el ciclo `for` lo soluciona mediante el uso, iterativo y secuencial, del total de hilos para actualizar al arreglo por bloques adyacentes de tamaño `blockDim* gridDim`.

Conclusión

En este artículo se han expuesto los conceptos básicos, el potencial y las restricciones de los dispositivos de procesamiento de gráficos (GPU), al emplearse como procesadores paralelos de propósito general. En particular, se ha dado énfasis a las recomendaciones prácticas que procuran el mejor aprovechamiento de esta poderosa herramienta de cómputo. En este sentido, y para finalizar, es muy importante saber reconocer en cuáles aplicaciones es conveniente usar un GPU y en cuáles no, ya que, en el caso de elegir una aplicación incompatible con las mejores prácticas de programación del GPU, se puede llegar a la situación de que el incremento de velocidad en la ejecución de la aplicación y la correspondiente disminución en el tiempo consumido, no justifique el empleo del GPU, ni por desempeño respecto del CPU anfitrión, por sí solo, ni por el tiempo y esfuerzo requerido para paralelizar la aplicación en el GPU.

Referencias

1. *CUDA C Best Practices Guide, Version 3.2* [Manual de software informático]. (2010). California, USA.
2. Mzoughi, A., Lafontaine O., y Litaize, D. (1996) Performance of the vectorial processor VECSM2* using serial multiport memory. En *International conference on supercomputing* (pp 390–397). Toulouse Cedex, Francia.
3. *NVIDIA CUDA C Programming Guide, Version 3.2* [Manual de software informático]. (2010). California, USA.
4. Owens J., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. y cols (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1), 80–113.

cs