Revista en Telecomunicaciones e Informática, Vol. 2, No. 4 p. 81 - 97
Medellín - Colombia. Julio - Diciembre de 2012, ISSN 2215-8200

Guía y fundamentos de la programación en paralelo

Fabio A. ACOSTA, Oscar M. SEGURA, Alvaro E. OSPINA
Facultad de Ingeniería Eléctrica y Electrónica, Universidad Pontificia Bolivariana; Cir. 1 #70-01, B11, Medellín. Colombia.
fabioandres.acosta@alfa.upb.edu.co,
oscartronco@gmail.com,
alvaro.ospina@upb.edu.co

Resumen
En este artículo se hace una introducción a los conceptos básicos de la programación en paralelo, basados en tres modelos diferentes como MPM (*Message Passing Model*) mediante la librería MPI, SMM (*Shared Memory Model*) mediante la librería OpenMP y CUDA (*Compute Unified Device Architecture*) se busca familiarizar al lector con las arquitecturas y los tipos de funciones más importantes que manejan las librerías, dando algunas pautas generales para diseñar un programa basado en las librerías mencionadas anteriormente. La finalidad es que se tengan conceptos para definir cuál de los modelos de programación en paralelo se adapta más a los recursos dispuestos para desarrollar una aplicación.

Abstract
This article provides an introduction to the basic concepts of parallel programming, based on three different models as MPM (Message Passing Model) by MPI library, SMM

(Shared Memory Model) by library OpenMPI and CUDA (Compute Unified Device Architecture) try to familiarize the reader with the architecture and the types of functions most important of libraries. Giving some general guidelines to design a program based on the above libraries. The aim is to have concepts to define which of the parallel programming models is more adapted to the resources provided to develop an application.

Keywords: parallel programming, library, architecture, functions, resources, programación en paralelo, modelos de programación en paralelo.

1. Introducción

Con la evolución de la computación se ha incrementado la capacidad de procesamiento de los equipos, lo que ha dado pie al desarrollo de programas y aplicaciones que manejan un alto volumen de datos. La programación en paralelo ha surgido como una nueva alternativa para aprovechar de una manera más eficiente los recursos de los computadores.

La finalidad del presente artículo es acercar al lector a la programación en paralelo, a través de tres librerías basadas en diferentes modelos de trabajo que permiten analizar las múltiples alternativas para el diseño eficiente de aplicaciones, que brinden la posibilidad de decidir con un buen criterio la librería por utilizar con base en las necesidades y recursos existentes para el desarrollo.

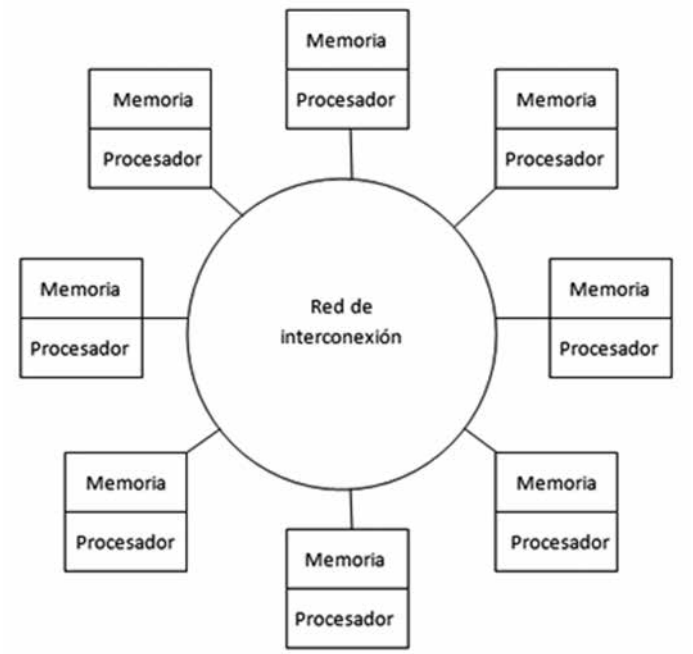
Los conceptos básicos de los diferentes modelos de programación (*Message Passing Interface – MPM, Shared Memory Model – SMM, Compute Unified Device Architecture - CUDA*) se encuentran descritos de forma breve en el presente artículo. Es imperante el apoyo de textos, guías de laboratorio, ejemplos de uso y ayudas didácticas para hacer más entendibles los conceptos de la programación en paralelo, lo que obliga a los estudiantes a compenetrarse con las nuevas tecnologías de una manera práctica que le permita a futuro aprovechar los recursos.

2. *Message Passing Model* – MPM

(En español Modelo de Paso por Mensajes). Según QUINN (2004), este modelo es aprovechado principalmente por los múltiples *cores* de trabajo que están instalados en los equipos de nueva tecnología. Con MPM se asume el *hardware* como la reunión de diferentes procesadores (dentro o fuera de una máquina en el caso de *clusters*) a

través de una red de interconexión donde cada uno tiene acceso directo y exclusivo a la información almacenada en su propia memoria local, véase Fig.1.

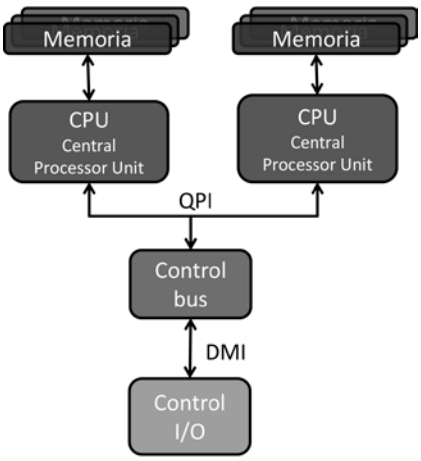
Figura 1. Esquema del modelo de paso por mensajes



MPM maneja un tipo de acceso a memoria denominado NUMA (*Nom-Uniform Memory Access).* En el que se describe que los procesos que se comunican por la red escalable de interconexión manejan un espacio específico en memoria para hacer la interacción y comunicación con los demás procesos por medio de mensajes.

En cuanto al *hardware* (Fig.2), cada memoria está directamente conectada a una CPU, en lugar de estar conectada a un controlador de memoria (modelo UMA), en este modelo los controladores de memoria se encuentran integrados a las distintas CPU que están en la red. Además, las CPU están conectadas con un I/O hub, permitiendo un ordenamiento en los datos y reduciendo los problemas de tráfico (canal común).

Figura 2. Hardware para el modelo NUMA



La comunicación y la transferencia de datos entre procesos se hace usando mensajes estructurados que contienen información clave referente a los datos, las funciones, las variables de estado y principalmente mensajes de sincronización de las memorias locales de los procesadores.

Para el uso de un programa basado en MPM se debe especificar el número de procesos con que se cuenta para la aplicación, a cada proceso se le asigna un número de identificación (ID) o *Rank,* con el fin de supervisar y controlar el flujo del programa. Para la sincronización entre procesos se usan una serie de funciones definidas en la librería.

Los mensajes de MPM generalmente contienen:

- La variable en la que reposan los datos que se envían.
- La cantidad de datos que se envían.
- El proceso receptor, el que recibe el mensaje.
- Los datos que se esperan recibir por parte del receptor.
- El tipo de dato que se envía.
- El proceso emisor del mensaje.
- La ubicación o espacio en memoria (puntero en C) donde se almacenarán los datos.

2.1. Librería MPI

El desarrollo de MPI comienza en los años ochenta por las diferentes casas productoras de equipos de cómputo, en 1993 se crea la primera versión libre de un desarrollo que contenía una librería de paso por mensajes diseñada para sistemas

de computadores intercomunicados con conexiones en paralelo denominada PVM V3 (*Parallel Virtual machine*) ganando mucha popularidad. Posteriormente, en mayo 1994 se ajustaron algunos complementos de la librería de PVM y nace la primera versión de MPI 1.0 (*Message Passing Interface*), véase HIDROBO (2005). Actualmente se sigue trabajando en el desarrollo de la librería que se encuentra en la versión 3.0.

Una característica llamativa de MPI es que permite trabajar con grupos de procesadores definidos según el programador lo disponga mediante objetos virtuales denominados comunicadores, que permiten distribuir los recursos según la tarea a realizar.

Con la librería MPI se debe tener claro que sólo se puede declarar una única vez el ambiente en paralelo (sección comprendida entre las funciones MPI_Init y MPI_ Finalize) y que todo el código que este dentro de la zona se ejecutará en simultáneo por todos los procesos.

Un programa desarrollado con MPI tiene una estructura como la que se muestra en la Fig.3.

Figura 3. Estructura general de un programa en MPI

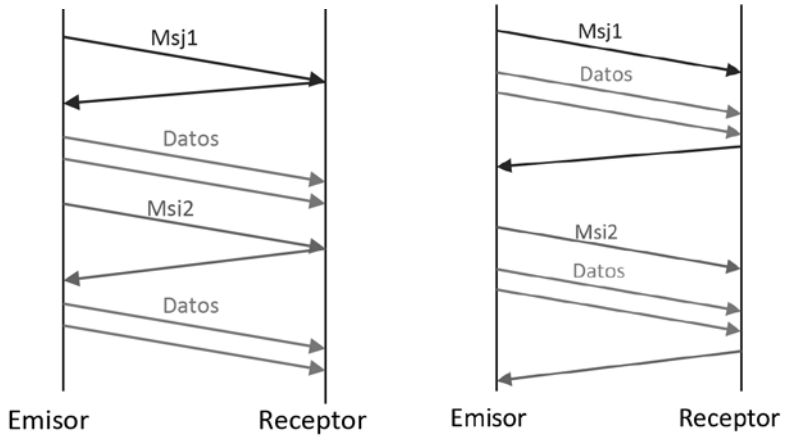
Como cualquier librería, MPI cuenta con una serie de funciones que permiten llevar a cabo tareas específicas dentro del ambiente en paralelo, la sintaxis general de una función de MPI es del tipo:

MPI_función. [argumento1, argumento2, ...] \n

Dentro de los grupos de funciones para MPI se destacan algunas como:

- Funciones de control de flujo: permiten crear y establecer parámetros de la sección en paralelo como número de procesos a usar, el comunicador, los ID de los procesos de la aplicación, etc.
- Funciones para el manejo de grupos y comunicadores: facilitan la conformación de los grupos de procesadores.
- Funciones de administración de recurso.
- Funciones de comunicación: permiten la interacción (enviar y recibir información) entre diferentes procesos. Según el número de procesos presentes en la comunicación ésta se clasifica en *punto a punto* y *multipunto*.
- Funciones para comunicación *punto a punto*: implican la interacción de dos procesos exclusivamente (maestro y esclavo), que según el tipo de petición para establecer la conexión se dividen en *m*étodo bloqueante y *no bloqueante* (véase Fig.4).
- Funciones para comunicación *multipunto*: interactúan con múltiples procesos simultáneamente, el uso de ellas requiere que el desarrollador tenga claro el recurso con el que cuenta.

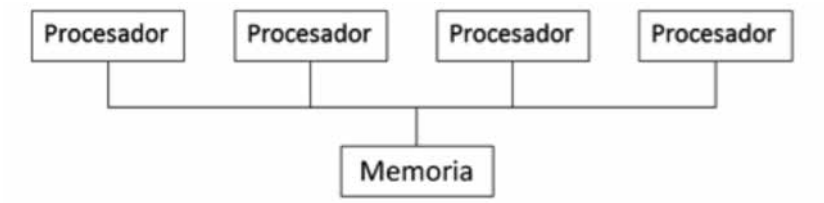
Figura 4. Esquemas de los métodos bloqueante a) y no bloqueante b)



3. *Shared Memory Model* – SMM

El modelo SMM es una abstracción del modelo de multiprocesamiento centralizado, que se compone por una colección de procesadores en donde cada uno tiene acceso a una memoria local compartida en la que se almacenan variables que pueden ser usadas por todos los procesos, véase la Fig.5.

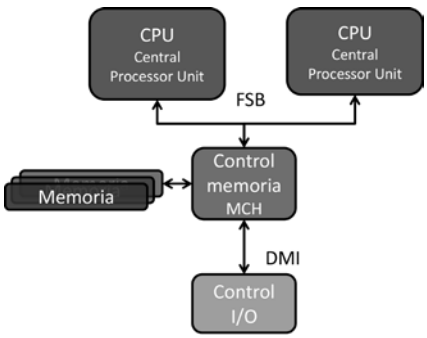
Figura 5. Esquema de hardware para SMM



El método de acceso a memoria es llamado UMA (*Uniform Memory Access*), también conocido como SMP (*Symmetric Multi- Processing*). Básicamente, describe que cada procesador de un arreglo de procesadores idénticos tienen los mismos tiempos de acceso a una memoria compartida equidistante a cada núcleo, mediante un bus de interconexión.

El *hardware* en SMM está basado en FSB (*front-side bus*) que a su vez es el modelo acceso a memoria usado en UMA (*Uniform Memory Access)*, véase Fig.6. Consta de un controlador de memoria (MCH) al que se conecta a la memoria general, las CPU interactúan con el MCH cada vez que necesitan acceder a memoria. También, cuenta con un controlador de I/O que se conecta al MCH, por lo tanto, se genera un problema de cuello de botella con el bus cuando se requiere de la memoria, generando limitaciones de escalabilidad y una velocidad finita.

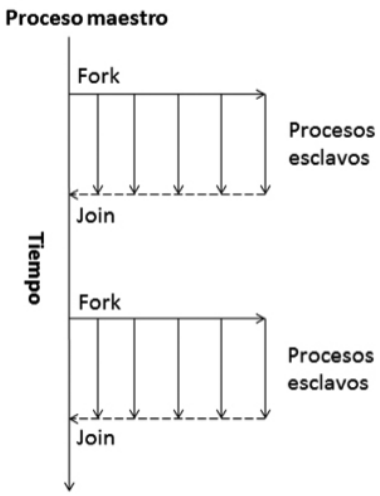
Figura 6. Hardware para el modelo UMA



SMM está fundamentado en un modelo de ejecución denominado *fork/join* que básicamente describe la posibilidad de pasar de una zona secuencial ejecutada por un único hilo maestro (*master thread*) a una zona paralela ejecutada por varios hilos esclavos (*fork*), posteriormente, cuando finalice la ejecución en paralelo, la información y resultados se agrupan de nuevo mediante un proceso de escritura en memoria en un proceso llamado *join*, véase Fig.7.

A diferencia de un modelo de memoria compartida (SMM), en un programa basado en MPM los procesos típicos permanecen activos en todo momento de la ejecución del programa; por el contrario, en el SMM el número de conexiones activas es dinámico (se pueden crear y destruir tantos procesos como sean necesarios).

Figura 7. Esquema del modelo de ejecución fork/join



3.1. Librería OpenMP

Es una interfaz portable programable para computadores o redes que soportan SMM, que fue adoptada como un estándar informal en 1997 por científicos que buscaban generalizar los programas basados en SMM, véase CHANDRA (2000).

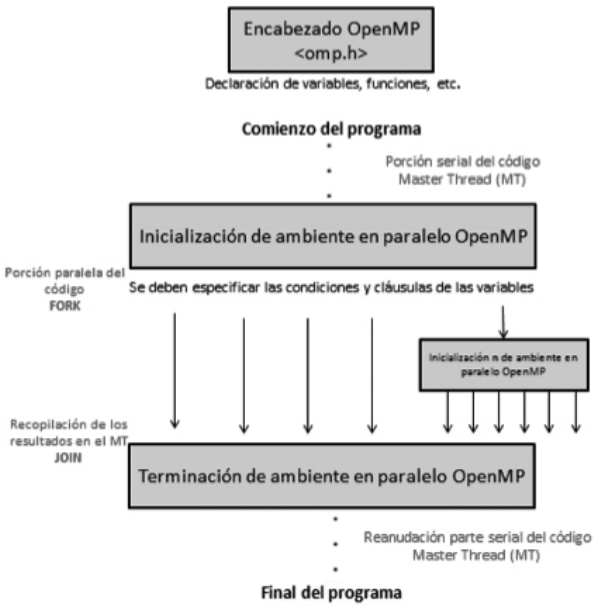
La gran portabilidad es una característica de MPI debido a que está soportado para C, C++ y Fortran, disponible para sistemas operativos como Solaris, AIX, HP-UX, GNU/Linux, MAC OS, y Windows.

Además, OpenMP soporta la interacción con el modelo de paso por mensajes (MPM) permitiendo la integración de la librería MPI en las aplicaciones, lo que amplía aún más las alternativas de programación.

Un programa desarrollado en OpenMP debe manejar la siguiente estructura básica (esquematizada en la Fig.8):

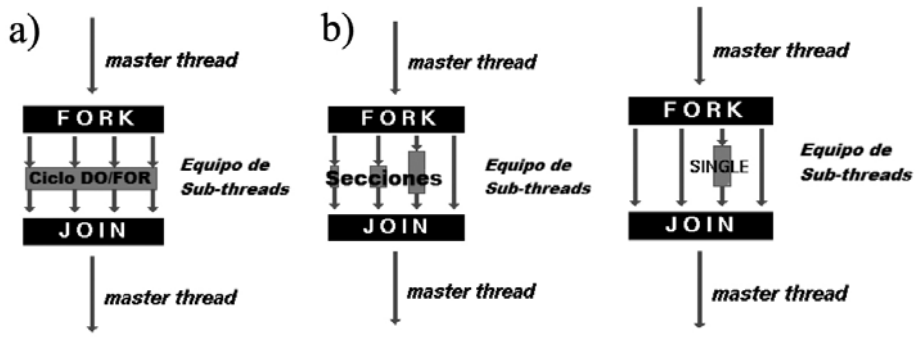
- Fichero principal: <omp.h>: hace el llamado de la librería.
- Directivas #pragma de compilación: son sentencias especiales que determinan la forma de trabajar de los procesadores asignados a una aplicación, se usa la siguiente sintaxis.
- #pragma omp nombre_directiva. [cláusulas, ...] \n
- Clausulas: son los parámetros que definen una condición en las funciones.
- Variables de entorno: son los posibles valores que cambian dinámicamente, afectando cualquier proceso definido en un ambiente de programación OpenMP.

Figura 8. Estructura general de un programa en OpenMP



Una particularidad de OpenMP es que permite administrar el recurso en rutinas que contienen cálculos iterativos, mediante la distribución de iteraciones de los ciclos en los diferentes *threads* (hilos) que maneja la aplicación mediante funciones especiales denominadas constructores, cuyo esquema está expuesto en la Fig.9.

Figura 9. Constructores Do/for a), secciones b) y single c)



Los constructores se combinan con una cláusula especial llamada *Schedule* que contiene parámetros que definen la forma de distribución de las iteraciones.

Con OpenMP es posible trabajar el anidamiento, que consiste en crear nuevas secciones paralelas dentro de los mismos *threads* formando una especie de ramificación que facilitan el procesamiento de grandes volúmenes de información.

Al igual que MPI, OpenMP también cuenta con rutinas especiales para atención y detección de errores, que son de gran ayuda para comprender el modo de trabajo de las diferentes directivas.

4. *Compute Unified Device Architecture*- CUDA

Nace a partir de la necesidad de desarrollar nuevas alternativas para el procesamiento de grandes volúmenes de información, aplicables en el desarrollo de videojuegos, reproductores de video y simulaciones complejas (meteorología y distribución térmica). Con el tiempo surge Nvidia como una compañía especializada en el desarrollo de tecnologías para el procesamiento de información, que a mediados del 2000 lanzó la primera GPU (*Graphic Processing Unit*) que se caracterizaba por la gran capacidad para hacer operaciones de punto flotante, esta cualidad hizo que el interés de la comunidad se volcará a la tecnología. En 2003 se desarrolla el primer modelo de programación extendido a C usando una GPU que posteriormente fue sometido a algunos ajustes de *hardware* y *software*, generando así en 2006 lo que hoy se conoce como CUDA, véase la historia completa en NVIDIA *Corporation* (2012). Arquitectura de CUDA

Básicamente es un estándar que migra de un modelo centralizado (se caracteriza por que todas las operaciones eran asumidas exclusivamente por la CPU) a un

modelo de cooprocesamiento en el que se dividen los datos ordenadamente entre la CPU y la GPU.

El hardware de CUDA se compone por la CPU de un equipo principal o *host* y una GPU dispuesta en un dispositivo externo o *device.*

La GPU es un procesador vectorial (generalmente contenido en una tarjeta de video Nvidia) en el que se destinan mayor cantidad de transistores para el procesamiento de datos de punto flotante que para el almacenamiento de información y control de flujo. A nivel de código de programación con las GPU se obtienen ventajas como:

- Código compacto: una instrucción define N operaciones.
- Reduce la frecuencia de los saltos en el código.
- No se requiere de *hardware* adicional para detectar el paralelismo.
- Permite la ejecución en paralelo asumiendo N flujos de datos paralelos.
- No maneja dependencias.
- Usa patrones de acceso a memoria continua.

Una GPU está compuesta por un número escalable de multiprocesadores paralelos (*streaming multiprocessor* - SM). La cantidad de SM varía según el modelo de la tarjeta de video. Cada SM cuenta con 48 procesadores para operaciones aritméticas que suman un promedio de 384 procesos, 2 unidades especiales de función (SFU), 1 unidad de multiplicación y suma (MAD); y 1 unidad adicional de multiplicación (MUL). Además, todos los procesadores de un SM comparten la unidad de búsqueda y lanzamiento de instrucciones de tal forma que se ejecuta la misma instrucción al mismo tiempo en todos los procesadores.

Un SM cuenta con varias memorias (véase Fig. 10) que pueden ser accesadas en cualquier parte del código:

- Memoria compartida de lecto/escritura.
- Memoria de constantes.
- Memoria de texturas

Por el contario una CPU cuenta con bloques únicos para el manejo de caché, para el control y para el manejo de la RAM; y en muchos casos múltiples ALU (dependiendo de las características). Por su parte, la GPU cuenta con una unidad de control o Kernel y una caché por cada ALU dispuesta (según el modelo de la tarjeta de video), lo que permite aumentar el poder de cómputo de manera considerable, véase Fig.11.

Figura 10. Esquema de memoria en una GPU

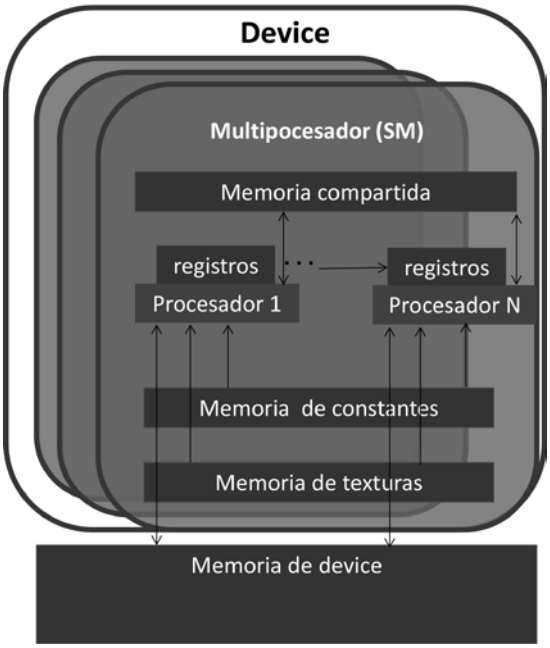
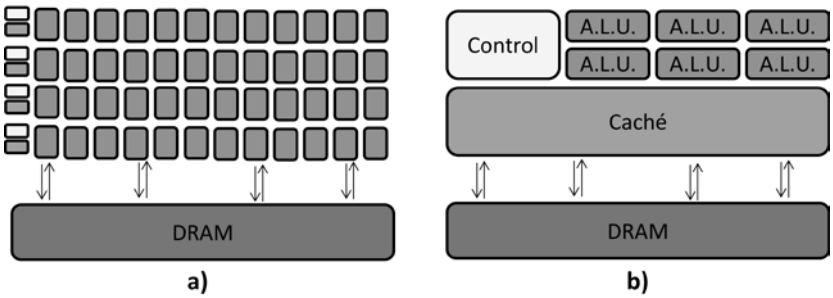
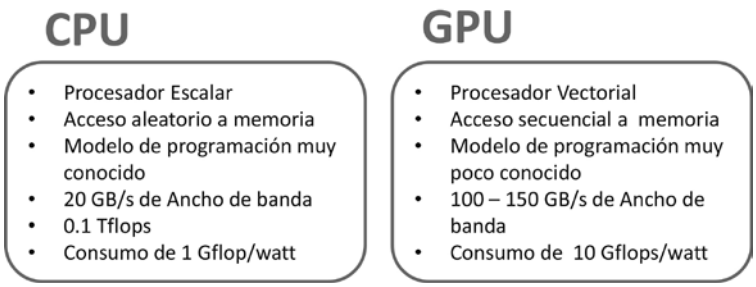


Figura 11. Composición de la GPU a) vs. CPU b)



Técnicamente existen grandes diferencias entre una CPU y la GPU como las que se muestran en la Fig.12.

Figura 12. CPU vs. GPU

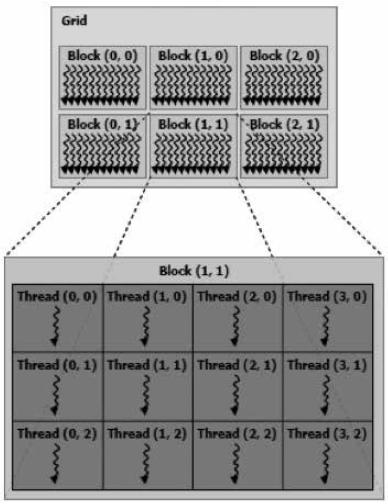


4.2. Modelo de programación

Para CUDA se maneja un concepto de programación en el que se puede ejecutar código de forma secuencial mediante la CPU o paralela usando la GPU. El kernel hace la paralelización del código, asignando tareas específicas a los diferentes *threads* dispuestos para la aplicación.

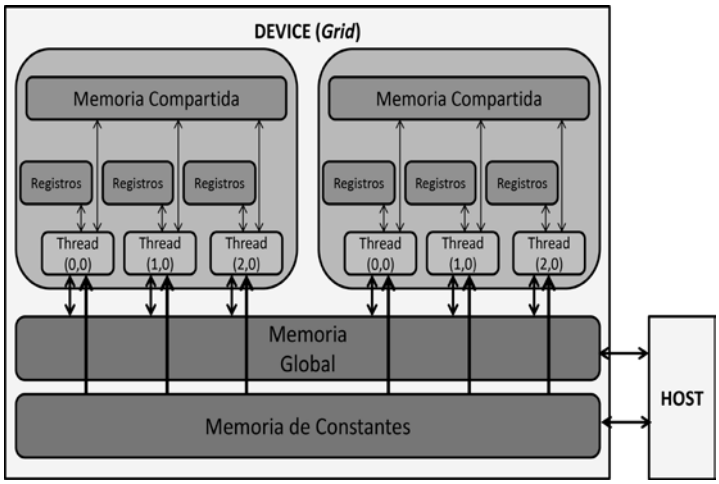
Para definir la cantidad de *threads* se maneja una jerarquía donde los *threads* se agrupan en estructuras de 1, 2 ó 3 dimensiones denominados **bloques**, los bloques a su vez, se agrupan en un ***grid*** de bloques que también puede ser de 1, 2 ó 3 dimensiones, como se indica en la Fig.13.

Figura 13. Esquema de división de *threads*



Para la comunicación entre *threads* en CUDA se usan una serie de memorias que tienen jerarquía en donde los resultados de las operaciones son comunes entre los *threads*. Cada bloque maneja una memoria compartida visible por todos los *threads* del bloque, que dura poblada el tiempo que dure activo el bloque, por otra parte, los bloques también manejan otra memoria común entre ellos mismos y el *grid* denominada memoria global (DRAM), véase Fig.14.

Figura 14. Mapa de memoria del *device*



La estructura de un código basado en CUDA contiene:

- Declaración de funciones y kernel (función que realiza el *host*).
- Copiado de variables de la memoria del *Host* al *Device*.
- Paralelización y ejecución de las tareas por realizar en la GPU.
- Ordenamiento de los resultados de las operaciones paralelas.
- Copiado de los resultados de la memoria del *Device* al *Host*.
- Liberación de memorias.

Para CUDA existen dos API que controlan el entorno de programación; la primera de alto nivel llamada **CUDA Runtime API** para C++ que maneja funciones con prefijo cudaFunción y está definida en el fichero "cuda_runtime_api.h". Estas funciones sólo se pueden procesar a través del compilador dispuesto por nVidia llamado nvcc. La API usada con compiladores de bajo nivel se llama **CUDA driver API** y contiene casi todas las mismas funciones que la primera librería, pero con la diferencia de que inician con el prefijo cuFunción.

En cuanto a la declaración de una función en el kernel (código que se ejecuta de forma paralela en la GPU) se usa una instrucción con la siguiente sintaxis, así:

global void FuncionKenel (tipo_dato vblesDeFuncion)
{Definición de la función}

Al momento de hacer el llamado de esta función en el código inmediatamente se pasa el flujo al *device* para la ejecución en paralelo por parte de los *threads* que fueron definidos en los en el *grid.*

CUDA maneja funciones diseñadas exclusivamente pare el trabajo en paralelo con el *device* (GPU), las funciones se usan con múltiples propósitos como el manejo de memoria, operaciones lógicas, control de flujo, entre otras. Además hay funciones con características especiales que permiten el manejo del entorno gráfico mediante texturas y superficies (gráficos 3D) basadas en OpenGL.

Todas estas características hacen de CUDA una excelente alternativa para hacer desarrollos en paralelo que impliquen el procesamiento de grandes volúmenes de información. Nvidia ha dispuesto en el paquete de instalación una carpeta que contiene gran cantidad de ejemplos que le dan al usuario un soporte en caso de dudas. En este artículo se han mostrado tres diferentes modelos de programación en paralelo en los que es posible resaltar que el costo de la implementación de los modelos depende mucho de los recursos con que se cuente. En el caso de que se tenga con una red de máquinas como los clúster o salas de computadores es más fácil utilizar MPM, ya que no es necesario realizar cambios significativos a nivel de *hardware* y *software* para su implementación.

La capacidad de procesamiento *(speedup)* también depende del recurso, ya que en los modelos de MPM el rendimiento está ligado al número de *cores* con los que se cuenta (a mayor número de *cores* mayor rendimiento, teniendo en cuenta el grado de desfragmentación de las tareas), que diferencia de SMM con el aumento de recursos, también se logra un aumento pero en menor medida, debido a que se llega a un punto de cuello de botella en el bus de datos.

La aplicabilidad de los modelos está ligada al nivel de granularidad de las tareas; es decir, en el caso de que una tarea se pueda dividir en muchas partes se obtienen mejores resultados usando MPI, puesto que los múltiples recursos se dividen el trabajo. En el caso de que se tenga el recurso (GPU) CUDA sería la mejor opción porque ofrece gran cantidad de *cores* para el procesamiento de las tareas y libera la CPU. En el caso de que las tareas sean más densas de nada sirve tener muchos recursos, porque no todos están trabajando sobre la tarea, en este caso sería mejor usar OpenMP porque el direccionamiento de información es más eficiente al interactuar directamente con la memoria compartida.

Conclusiones

La paralelización de tareas usando la librería MPI está basada en el modelo de paso por mensajes que conceptualmente es más asimilable por que las tareas se dividen según el número de procesadores físicos con los que cuenta el sistema. MPI permite un control más riguroso sobre los procesos que llevan a cabo las tareas dentro del flujo, debido a que es posible administrar quién ejecuta una tarea en específico, dando como resultado un nivel de paralelización más modificable que facilita el desarrollo de la aplicación. Además, la estructura del código es la más semejante de las tres librerías a la usada a un código de C.

Cuando el diseño de una aplicación requiere varias secciones secuenciales y paralelas, OpenMP facilita el trabajo, gracias a su modelo de programación que combina la interacción entre rutinas secuenciales y paralelas que lo diferencia de MPI en donde sólo se puede declarar un único ambiente en paralelo. Cuando se usa OpenMP se debe ser cuidadoso en el manejo de la memoria de los *threads* debido a que por la naturaleza del modelo de memoria compartida, es posible que se pueda presentar *aliasing* al momento de escribir la memoria.

El mayor rendimiento de CUDA se obtiene cuando se realiza una operación puntual que no requiere de una lógica muy extensa dentro del kernel, puesto que CUDA está diseñado para manejar una gran cantidad de datos durante un corto tiempo para optimizar el procesamiento.

Teniendo en cuenta que existen parámetros que hacen imposible medir exactamente el rendimiento en las mismas condiciones para las tres librerías de programación en paralelo, es posible afirmar que cualquiera de las tres opciones optimiza los tiempos de procesamiento de un código respecto a la ejecución secuencial del mismo, de acuerdo con esto, CUDA es la librería que presenta los tiempos más reducidos de procesamiento cuando se ejecutan tareas que implican el manejo de un gran volumen de datos.

Referencias

CHANDRA, Rohit, et al (2000). *"Parallel Programming in OpenMP".* 1st Ed. . Londres : Academic Press, pp. 13-14

HIDROBO, Francisco y HOEGER, Herbert. (2005) "Introducción a MPI: (Message Passing Interface)". [En línea] 16 de marzo de 2006. [Citado el: 12 de febrero de 2011.] Disponible en: http://webdelprofesor.ula.ve/ingenieria/hhoeger/Introduccion_MPI.pdf.

NVIDIA Corporation (2012). NVIDIA CUDA C: Programming Guide. [En línea] 16 de abril de 2012. [Citado el: 10 de mayo de 2012.] Disponible en: http://developer.nvidia.com/cuda-gpus.

QUINN, Michael J. (2004) *"Parallel programming: in C with MPI and OpenMP ".* 1st Ed. Oregon : McGraw-Hill Higher Education, 529 páginas.