Comparación del uso de GPU y cluster de multicore en problemas con alta demanda computacional

Erica Montes de Oca[1], Laura De Giusti[1], Armando De Giusti[1 2], Marcelo Naiouf [1]

[1]Instituto de Investigación en Informática LIDI (III-LIDI)
Facultad de Informática, Universidad Nacional de La Plata.
La Plata, Buenos Aires, Argentina.
[2] CONICET
{emontesdeoca,ldgiusti,degiusti,mnaiouf}@lidi.info.unlp.edu.ar

Resumen. Este trabajo realiza una comparación del uso de dos arquitecturas multiprocesador, tomando como caso de aplicación un problema con alta demanda computacional como el de N-body. Se presentan las implementaciones paralelas con memoria compartida (usando Pthreads) y pasaje de mensajes (con MPI) en cluster de multicore, y una solución sobre GPU (con CUDA). Se describen y analizan los resultados experimentales obtenidos, que muestran la buena performance lograda con el uso de GPU.

Palabras claves: multicore, cluster de multicore, GPU, GPGPU, CUDA, N-Body.

1 Introducción

Los intentos por acelerar el procesamiento secuencial a través del mejoramiento del hardware alcanzó su máxima capacidad cuando ya no se pudo sostenerse el incremento de la frecuencia de reloj. [1]. El gap producido entre el rendimiento esperado y el obtenido de las arquitecturas fabricadas hasta entonces motivó a la industria del hardware a orientar su producción hacia la construcción de máquinas paralelas. La tecnología del semiconductor comenzó a tomar dos caminos principales en la fabricación de las nuevas arquitecturas de cómputo: los multicores y los manycore [2].

Los multicores resolvieron el problema al que se enfrentaban los arquitectos de hardware, reduciendo la complejidad de los procesadores además de doblar la cantidad de núcleos de procesamiento con cada nueva generación [3]. El impacto producido en el mercado desplazando a las supercomputadoras permitió reducir los costos y la infraestructura garantizando la ejecución de problemas complejos con alta demanda computacional [4].

Los cluster son un conjunto de equipos independientes que pueden ser vistos como un sola máquina unificada [5]. Cada cluster puede estar conformado por máquinas iguales (homogéneo) o diferentes (heterogéneo). El cluster de multicores nace como una consecuencia del avance tecnológico y la necesidad de un mayor poder de cómputo.

La GPU (Unidad de Procesamiento Gráfico) ha emergido en los últimos años como una plataforma alternativa para el procesamiento paralelo a costo accesible. Es

una arquitectura manycore caracterizada por un número masivo de procesadores simples, cuyo uso avanzó en otros campos además del procesamiento de imágenes. Las GPUs modernas pueden ser descriptas como un vector de procesadores que ejecutan la misma instrucción sobre diferentes datos de forma paralela [6][7][8][9].

Un problema con alta demanda computacional ampliamente estudiado es el de los N-body, caracterizando una variedad de casos que van desde la interacción de las partículas electrostáticas hasta el movimiento de los cuerpos astronómicos en el espacio.

Este trabajo presenta una comparación de soluciones al problema de la atracción gravitacional de los cuerpos celestes utilizando cluster de multicore y GPU. La Sección 2 plantea el problema de los N-body. En la Sección 3 se describen las soluciones desarrolladas, mientras que en la Sección 4 se muestran los resultados experimentales obtenidos. La Sección 5 presenta las conclusiones y trabajos futuros.

2 El problema de los N-body

Caracteriza una variedad de problemas que van desde la interacción de las partículas electrostáticas hasta el movimiento de los cuerpos astronómicos en el espacio [10]. El presente artículo se basa en una solución al problema de la atracción gravitacional de los cuerpos celestes, apoyada en la teoría desarrollada por Newton, quien descubrió que la fuerza de atracción entre dos cuerpos es proporcional a sus masas e inversamente proporcional al cuadrado de la distancia que los separa.

Cada cuerpo cuenta con una masa (m), una posición inicial (p) y una velocidad (v). La gravedad hace que los cuerpos se aceleren y se muevan, provocando la atracción entre ellos. La relación entre masa y peso se puede obtener del Principio de Masa de Newton (Ecuación 0):

$F = m \times a$ donde F = cantidad de fuerza, m = masa, a = aceleración **(0)**

Durante un pequeño intervalo de tiempo (dt), la aceleración del cuerpo i (a_i) es aproximadamente constante, por lo que el cambio de velocidad del cuerpo (dv_i) es el indicado en la Ecuación 1:

$$dv_i = a_i \times dt \quad \textbf{(1)}$$

El cambio de la posición del cuerpo (dp_i) puede ser calculado como la integral de la velocidad y la aceleración en el intervalo de tiempo dt, que es aproximadamente el indicado en la Ecuación 2:

$$dp_i = v_i \times dt + (a_i/2) \times dt^2 \quad \textbf{(2)}$$

También, puede calcularse la magnitud de la fuerza de gravedad entre dos cuerpos i y j a través de la Ecuación 3 expresada por Newton:

$F = (G \times m_i \times m_j) / r^2$ *con r = distancia ,G = cte gravitacional (6,67 x 10-11)* **(3)**

Si los cuerpos se hallan en un plano espacial de dos dimensiones, las posiciones pueden ser representadas como puntos coordenados en el plano tal como: ($p_i.x$, $p_i.y$) y ($p_j.x$, $p_j.y$). El plano espacial donde se hallan los cuerpos se lo denomina espacio euclídeo. La Ecuación 4 mide la distancia euclídea entre dos puntos:

$$\sqrt{(p_i.x - p_j.x)^2 + (p_i.y - p_j.y)^2} \quad \textbf{(4)}$$

Si la distancia de dos cuerpos es pequeña significa que los mismos están muy cerca de colisionar. En esos casos se considera que se producen choques entre los cuerpos, que es lo que sucede naturalmente en el espacio [10][11].

3 Descripción de las soluciones

A continuación se describen los algoritmos desarrollados en este trabajo para resolver el problema.

3.1 Solución secuencial

La implementación de la versión secuencial se realizó mediante el algoritmo Fast N-Body Simulation [12], que realiza básicamente las siguientes acciones:

1. Calcula la fuerza de atracción del cuerpo *i*, con respecto a los demás cuerpos.
2. Calcula la nueva posición y la velocidad del cuerpo *i*.

El cálculo de la fuerza de gravedad de un cuerpo no modifica las fuerzas de gravedad de los demás. Cada uno actualiza la suya para un momento dado en el tiempo bajo un escenario determinado. Se utilizan estructuras de datos de tipo arreglo unidimensional de tamaño N para almacenar: las fuerzas de atracción (*vf*), las posiciones (*vp*), las masas (*vm*) y las velocidades (*vv*).

El procesamiento de la fuerza de atracción de los cuerpos se realiza dividiendo los vectores *vf*, *vp*, y *vm* en bloques para aprovechar la localidad espacial y temporal de la caché, y de este modo, reducir su tasa de fallo. Para cada bloque de *vf* se realizan los cálculos necesarios utilizando los vectores *vp* y *vm* por bloques. Es decir que para calcular el valor de los elementos de un bloque de *vf* se toma el primer bloque de *vp* y *vm*, se realizan todos los cálculos necesarios entre ellos para actualizar el bloque de *vf*, luego se toma el segundo bloque de *vp* y *vm* para realizar las mismas acciones que con los primeros, y así sucesivamente hasta que se actualiza con los vectores completos *vp* y *vm*. Una vez realizado el cálculo de la fuerza de atracción gravitacional para todos los cuerpos del ese bloque de *vf*, se repite el cómputo con el siguiente bloque de *vf* hasta terminar con el vector completo.

3.2 Solución paralela en memoria compartida

En el caso de la solución en memoria compartida se utilizó Pthread [13] como herramienta para el manejo paralelo de los threads. El hilo principal crea T-1 threads, y entre todos realizan el trabajo.

En un paso de ejecución cada cuerpo calculará su fuerza de atracción independiente de la de los demás. A cada cuerpo *i* le interesa el valor de la posición

de los demás cuerpos, la cual no cambiará hasta que todos hayan calculado su fuerza de atracción.

Partiendo del algoritmo secuencial descripto en 3.1, esta solución paralela realiza las mismas acciones sólo que sobre porciones más pequeñas de datos (cada porción de *N/t* datos, siendo *N* la cantidad de cuerpos y *t* la cantidad de threads). La cantidad de bloques totales en los cuales se dividen los vectores *vf*, y *vm* es igual a la del secuencial. Por ejemplo, si en el algoritmo secuencial se realizó una división de 200 bloques, en el algoritmo paralelo con 8 threads se tienen 25 bloques del vector de fuerza para procesar por cada thread. En tanto, los vectores de posiciones y masa serán divididos en 200 bloques para todos los hilos, ya que se lo necesita completo para el cálculo de la fuerza de atracción de cada cuerpo.

Una vez que las fuerzas de atracción de todos los cuerpos fueron actualizadas, se realiza el cálculo de las velocidades y posiciones. La velocidad de cada cuerpo dependerá de su fuerza de atracción y es independiente de la de los demás cuerpos. La posición se calcula a partir de la velocidad del cuerpo y es independiente de los demás. Realizados dichos cálculos, los threads deben esperar en una barrera a que todos los demás hayan terminado antes de realizar otro paso de simulación, ya que para calcular la fuerza del cuerpo *i* el thread correspondiente necesita conocer las posiciones de los N cuerpos restantes.

3.3 Solución paralela en memoria distribuida

La solución en memoria distribuida es similar a la anterior en lo referido al cálculo de los datos. Solo que en lugar de tener threads ejecutándose, se tienen procesos que utilizan pasaje de mensajes de la librería MPI [14].

El proceso con rank cero es el que inicializa las estructuras de datos, y una vez realizada, reparte los datos entre todos los demás procesos. Luego cada proceso actualiza los vectores de fuerza, posición y velocidad sobre una porción de datos (*N/p* , donde N es la cantidad de cuerpos y *p* es la cantidad de procesos). La división de los datos en bloques se lleva a cabo como se explicó en la Sección 3.2.

Una vez que cada proceso ha terminado de realizar todos los cálculos correspondientes a la fuerza de atracción, actualiza el vector de velocidad y posición; al terminar dicho cálculo transmite los resultados a los demás. Por lo tanto, todos los procesos tendrán los datos resultantes al finalizar la aplicación.

3.4 Solución paralela en GPU-CUDA

En 2007 Nvidia lanzó la implementación de CUDA, plataforma hardware y software que extiende al lenguaje C con un reducido conjunto de abstracciones para la programación paralela de propósito general en GPU. La arquitectura de una GPU-Nvidia CUDA está organizada en Streams Multiprocessors (SMs), los cuales tienen un determinado número de Streams Processors (SPs) que comparten la lógica de control y la caché de instrucciones [9][15][16].

La organización de los threads puede verse por nivel o jerárquicamente de la siguiente manera: grilla o *grid* (formado por bloques de threads), bloque (conformado por threads) y threads. En un bloque, los hilos están organizados en warps (en general de 32 hilos). Todos los hilos de un warp son planificados juntos durante la ejecución.

El sistema de memoria de la GPU está compuesto por distintos tipos de memorias que se diferencian tanto en la forma de acceso de los hilos, las limitaciones en las operaciones permitidas y su ubicación dentro del dispositivo. Está compuesto por: *Memoria Global, Memoria de Textura, Memoria Constante, Memoria Shared o compartida , Memoria Local y Registro*.

CUDA permite definir una función denominada *kernel* que es ejecutada *n* veces, siendo *n* la cantidad de threads de la aplicación. Dicha función solo corre en la GPU o Device, y puede invocar código secuencial para ser ejecutado en la CPU. El kernel debe ser configurado con el número de hilos por bloque y por grid [15][16]. Como el kernel se ejecuta en el device, la memoria debe ser alocada antes de que las función sea invocada y los datos que requiera utilizar copiados desde la memoria del host a la memoria del device. Una vez ejecutada la función kernel, los datos de la memoria del device deben ser copiados a la memoria del host.

El algoritmo Fast N-Body Simulation, es apto para ser ejecutado en la GPU por la independencia de cómputo en el cálculo de la fuerza de atracción gravitacional de los cuerpos. Cada cuerpo debe resolver su fuerza gravitacional dependiendo de las posiciones de los demás sin modificar otro dato que no sea su propia fuerza. Del mismo modo, la actualización de posiciones y velocidades se realiza de manera independiente.

Se almacenan en la memoria global de la GPU las mismas estructuras utilizadas por la CPU (*vf, vv, vp* y *vm*). El paso siguiente es la comunicación de los datos previamente inicializados por la CPU a través del PCI-E. A continuación, la CPU delega la ejecución a la GPU, y queda en espera a la respuesta de esta.

En el kernel o función que calcula la fuerza de atracción gravitacional, cada thread calcula su posición en la memoria global que le permite acceder al cuerpo para el que tiene que calcular dicha fuerza. Se utilizan tres vectores con dimensión igual a la cantidad de threads por bloque, dos de los cuales son para almacenar una porción del vector de posiciones (una para las coordenadas en *x* y otro en *y*), y el tercero para una porción del vector de masas. Eso permite computar por bloque reduciendo el acceso a memoria global.

Además de su identificador en la memoria global, cada thread calcula su identificador dentro del vector en memoria compartida. Cada uno trae de memoria global la posición y la masa del cuerpo que le corresponde, por lo que el vector almacenado en memoria compartida es cargado por todos los threads, y la posición del cuerpo que le corresponde será almacenada en la dirección correspondiente a su identificador dentro de la memoria shared. Antes de comenzar a calcular la fuerza de atracción, los thread son sincronizados para garantizar que todas las posiciones de los vectores de la memoria compartida hayan sido cargadas.

Luego, cada thread realizará el cálculo correspondiente a la fuerza de atracción para su cuerpo con las posiciones que se encuentran en los vectores de posiciones de la memoria compartida. Cuando todos los threads del bloque hayan utilizado todas las posiciones de los vectores de la memoria shared, deben sincronizarse para volver a traer una porción más de las posiciones y de las masas de los cuerpos de la memoria global a la memoria compartida. Este ciclo se repetirá hasta que todo el vector de posiciones y masas haya sido utilizado. Finalmente, cada thread calcula la velocidad y posición del cuerpo que le corresponde. Cuando los pasos de simulación ha realizar se

han completado, la GPU enviará a la CPU los resultados calculados, y le devolverá el control de la ejecución.

4 Resultados experimentales

Las arquitecturas utilizadas para la experimentación fueron:

- un Blade de 16 servidores (hojas). Cada hoja posee 2 procesadores Quad Core Intel Xeón e5405 de 2,0 GHz, con cachés L2 2x6MB compartida de a par de procesadores, y sistema operativo Fedora 12 de 64bits.
- una GPU Geforce TX 560TI, con 384 procesadores con una cantidad máxima de thread de 768 y 1 GB de memoria RAM.

En las pruebas realizadas se varía la cantidad de cuerpos (256000 y 512000) y los pasos de simulación (5,6 y7). Los resultados se obtuvieron a partir de un promedio de ejecuciones.

En las Tabla 1 se muestra el tiempo de ejecución (en seg) del algoritmo secuencial y del algoritmo en MPI para 2, 4 y 8 procesos. En la Tabla 2 se presentan los tiempos de ejecución para los algoritmos secuenciales y Pthread con 2, 4 y 8 threads.

Tabla 1. Tiempos de ejecución (en segundos) para el algoritmo MPI en CPU. Con P = cantidad de procesos.

Tamaño del vector de señales	Cantidad de pasos de simulación	Secuencial	MPI (P = 2)	MPI (P = 4)	MPI (P = 8)
256000	5	19357,36	9711,28	4856,14	2429,29
	6	23227,00	11653,19	5827,61	2914,85
	7	27098,72	13596,05	6798,60	3400,58
512000	5	77426,49	38848,18	19430,01	9716,82
	6	92918,43	46620,05	23316,08	11659,93
	7	108358,99	54384,43	27200,91	13602,60

Tabla 2. Tiempos de ejecución (en segundos) para el algoritmo Pthread en CPU. Con T = cantidad de threads.

Tamaño del vector de señales	Cantidad de pasos de simulación	Secuencial	Pthread (T = 2)	Pthread (T = 4)	Pthread (T = 8)
256000	5	19357,36	9680,89	4839,94	2419,80
	6	23227,00	11617,94	5809,11	2903,56
	7	27098,72	13553,43	6776,39	3387,41
512000	5	77426,49	38727,09	19364,25	9679,42
	6	92918,43	46472,62	23230,21	11613,76
	7	108358,99	54221,36	27101,45	13551,05

Las pruebas experimentales en GPU se realizaron con bloques de 256 threads que es el óptimo para la arquitectura utilizada en la experimentación. Los resultados se presentan en la Tabla 3.

Tabla 3. Tiempos de ejecución (en segundos) para el algoritmo CUDA en GPU. Con T = cantidad de threads.

Tamaño del vector de señales	Cantidad de pasos de simulación	GPU (T = 256)
256000	5	37,90
	6	45,47
	7	53,05
512000	5	151,37
	6	181,64
	7	211,90

Las Fig. 1 y 2 muestran el Speedup (calculado como la relación entre el tiempo de la solución secuencial y el tiempo paralelo) para las soluciones MPI y Pthread con 256000 y 512000 cuerpos.

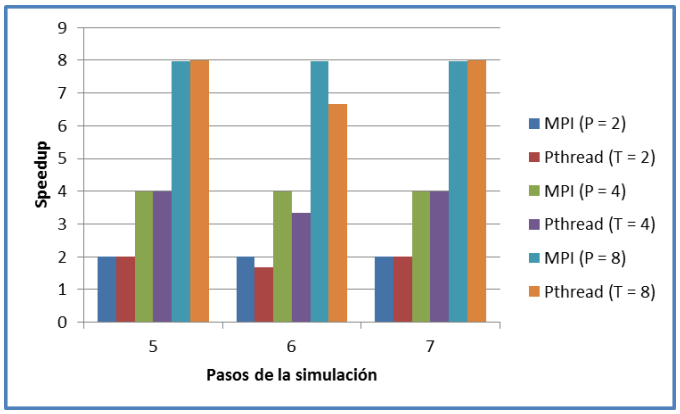

Fig. 1. Speedup de los algoritmos paralelos MPI y Pthread para 256000 cuerpos.

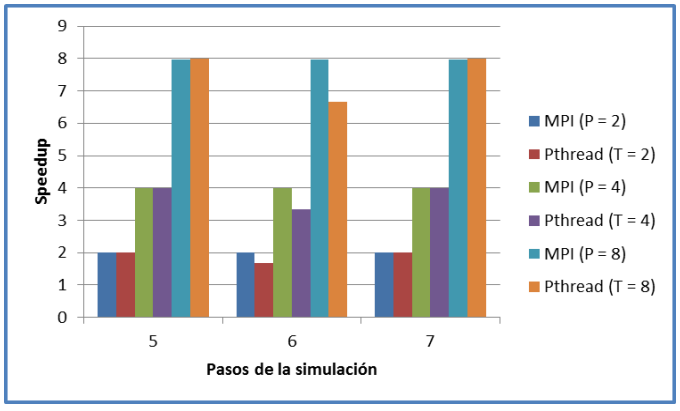

Fig. 2. Speedup de los algoritmos paralelos MPI y Pthread para 512000 cuerpos.

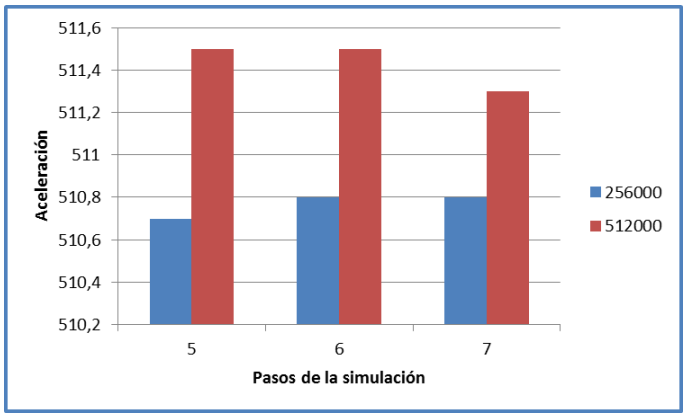

Fig. 3. Aceleración obtenida para el algoritmo N-Body en GPU.

A partir de los resultados obtenidos puede observarse que las soluciones Pthread y MPI presentan características similares en tiempo, no habiendo una diferencia significativa entre ellas en este aspecto. Esto se debe a que la etapa de sincronización entre todas las tareas es más significativa a la de comunicación.

En cambio, para la implementación GPU-CUDA se logran tiempos sensiblemente inferiores con respecto a los mencionados, denotando una alta adecuación de la arquitectura (algoritmo, solución) al problema. Un análisis equivalente puede realizarse a partir de los valores de speedup y aceleración mostrados.

5. Conclusiones y trabajos futuros

La reducción de los tiempos de ejecución en algunas aplicaciones requiere que su solución deba realizarse por medio del procesamiento paralelo para la obtención de tiempos de respuesta aceptables que no pueden ser cumplidos a través del procesamiento secuencial. Una diversidad de arquitecturas multiprocesador están disponibles con este fin, al igual que varias herramientas de programación que permiten explotar el paralelismo en las mismas.

En la última década, ha surgido una arquitectura de propósito específico como una alternativa para el cómputo de altas prestaciones: las GPUs han evolucionado al punto de ofrecer un poder de cómputo que permite alcanzar una gran performance, reduciendo los tiempos de ejecución de las aplicaciones adaptables a dicha arquitectura como se ha mostrado en el presente trabajo para problemas que comparten las mismas características que el problema de los N-Body.

Cabe agregar además, que el costo de una placa GPU es sensiblemente inferior al de la arquitectura blade utilizada para la comparación, reforzando de esta forma una de las ventajas de su empleo en problemas de este tipo.

Como trabajos futuros pueden mencionarse el uso de clusters de GPUs y el estudio del paradigma de programación híbrida utilizando MPI-CUDA para dicha arquitectura, así como avanzar en el análisis de consumo energético en el uso de GPU. Por último, interesa estudiar las nuevas arquitecturas MIC de próxima aparición.

Referencias

1. Moore Gordon E.: Cramming more components onto integrated circuits. Electronics, vol. 38, Número 8 (1965)
2. Kirk David B., Hwu Wen-mei W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann (2010)
3. AMD, "Evolución de la tecnología de múltiple núcleo". 2009. http://multicore.amd.com/es-ES/AMD-Multi-Core/resources/Technology-Evolution.
4. Balasubramonian R., Jouppi N., Mularimanohar N.: Multi-Core Caché Hierarchies, Morgan & Claypoll (2011)
5. Thiruvathukal G. K.: Cluster Computing. Copublicado por la IEEE CS y la API. (2005)
6. Chen J. Y.: GPU Technology Trends and Future Requirements. IEEE (2009)
7. Nvidia Corporation: GPU Gems, Pearson Education (2003)
8. Nickolls J., Dally W. J.: The GPU Computing Era. IEEE (2010)
9. Perez C., Piccoli M. F.: Estimación de los parámetros de rendimiento de una GPU. Mecánica Computacional, vol. XXIX, pp. 3155-3167 (2010)
10. Bruzzone Sebastian, "LFN10, LFN10-OMP y el Método de Leapfrog en el Problema de N Cuerpos ", Instituto de Física, Departamento de Astronomia, Universidad de la República y Observatorio Astronómico los Molinos, Uruguay. (2011)
11. Andrews Gregory R., "Foundations of Multithreaded, Parallel, and Distributed Programming", Addison-Wesley. (2000).
12. Lars Nyland, Mark Harris, and Jan Prins. Fast N-body simulation with CUDA. In GPU Gems 3, chap-ter 31, pages 677{695. Addison-Wesley Professional, 2007.
13. https://computing.llnl.gov/tutorials/pthreads.
14. Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J., "MPI: The Complete Reference". The MIT Press, Cambridge, Massachusetts. 1996.
15. Nvidia Corporation: NVIDIA CUDA C Programming Guide. (2011)
16. Nvidia Corporation: CUDA C Best Practices Guide. (2012)