Conciencia Tecnológica No. 43, Enero-Junio 2012

Optimización en Paralelo Basada en Poblaciones Usando GPGPU

Investigación

MSC. Iliana Castro Liera[1], Dr. Marco Antonio Castro Liera[2] , M.C. Jesús Antonio Castro[2]
[1]Depto. de Sistemas y Computación, [2] División de Estudios de Posgrado e Investigación
Instituto Tecnológico de La Paz, México, Blvd. Forjadores de B.C.S., #4720 Col 8 de Octubre
La Paz, B.C.S., CP 23080, Tel: 01(612)1210424 ext 116, Fax: 01(612)1211295
icastro@itlp.edu.mx, mcastro@itlp.edu.mx, castroli48@gmail.com.

Resumen

En este artículo se muestran dos estrategias para aprovechar el poderío computacional de las unidades de procesamiento gráfico de propósito general GPGPU (General-Purpose computation on Graphics Processing Unit) con arquitectura CUDA (Compute Unified Device Architecture) para la solución de problemas de optimización utilizando algoritmos genéticos (AG) y optimización por enjambre de partículas (PSO, Particle Swarm Optimization).

Se muestra que esta implementación en una sola GPU consigue resultados de buena calidad en tiempos menores que los de un cluster de 16 computadoras con 32 núcleos.

Palabras clave: GPGPU, AG, PSO, CUDA.

Abstract

This paper describes two successful strategies for using the computing power of a General Purpose Graphics Processing Unit (GPGPU) with CUDA (Compute Unified Device Architecture) to solve hard numerical optimization problems using genetic algorithms (GA) and particle swarm optimization (PSO).

The proposed algorithm executed on a single GPU outperforms a cluster of 16 computers and 32 cores.

Key words: GPGPU, GA, PSO, CUDA.

Introducción

El presente trabajo está siendo desarrollado en el I.T. de La Paz con apoyo de la DGEST con el proyecto de investigación PAZ-MSC-2009-201 y el periodo sabático AS-1-200/2011, cuyos resultados parciales fueron publicados en [1] [2].

Las nuevas herramientas computacionales nos han permitido disminuir el tiempo necesario para encontrar la solución a los problemas de optimización y han hecho posible que abordemos nuevos problemas, tan complejos, que sin ellas simplemente no podríamos resolver. Actualmente, muchas computadoras están equipadas con procesadores que incluyen más de un núcleo y en muchos casos, gracias al interés por los videojuegos, tienen una tarjeta gráfica poderosa que la mayor parte del tiempo es desaprovechada.

En el apartado de Fundamentos Teóricos se explican las bases de los Algoritmos Genéticos y la Optimización por Enjambres de Partículas.

En la sección de Materiales y Métodos, se describen los fundamentos de la arquitectura CUDA, y la manera en que se implementaron los algoritmos en ella. Se describen también la batería de funciones utilizadas para medir la eficacia y eficiencia de la implementación.

En las secciones de Resultados y Discusión y Conclusiones se muestra que utilizando una sola GPU se obtienen soluciones de buena calidad en tiempos menores que los requeridos por un cluster de 16 computadoras.

Fundamentos Teóricos

Optimización por Enjambres de Partículas

Los métodos de optimización por enjambres de partículas, propuestos en 1995 por Russ C. Eberhart y James Kennedy [5] están basados en el comportamiento de las bandadas de aves y los cardúmenes de peces en los que su movimiento se guía por la búsqueda de alimento o para escapar de los predadores y en los que el grueso de la población sigue al miembro que posee mayor conocimiento del entorno.

Cada partícula en el enjambre recuerda su posición (x_i), su velocidad (v_i) y aptitud (f_i), así como su mejor posición y aptitud históricas.

El algoritmo inicia generando aleatoriamente un conjunto de partículas, denominado enjambre. Cada partícula calcula su velocidad usando (1) y su nueva posición usando (2) para cada dimensión d; esto se realiza una cantidad preestablecida de generaciones o hasta alcanzar una condición de parada.

$$v_{id}(t+1) = wv_{id}(t) + c_1 r_1 (x_{hd} - x_{id}) + c_2 r_2 (x_{gd} - x) \quad (1)$$

donde x_h representa la mejor posición histórica de la partícula, w un peso inercial, c_1 la confianza en su propia información y c_2 la confianza en la información social; r_1 y r_2 son valores generados aleatoriamente con una distribución uniforme en el intervalo de [0,1].

$$x_{id}(t+1) = x_{id}(t) + v_{id} \quad (2)$$

Si se utiliza la estrategia global (*gbest*), x_g representa la mejor posición del enjambre; si se utiliza la estrategia local (*lbest*), x_g representa la mejor posición del vecindario. En general la estrategia *gbest* tiende a converger más rápido que *lbest*, pero es más probable quedar atrapado en un óptimo local. Podemos también decir que *gbest* es un caso especial de *lbest*, en donde el tamaño del vecindario es igual al de todo el enjambre [6].

Algoritmos Genéticos

Fueron propuestos por Holland [7] en su versión original y basan su comportamiento en la evolución de las especies, en donde sobreviven los más adaptados al entorno.

El concepto central de los algoritmos genéticos es el de **población**, que se define como un conjunto de individuos C_i donde $i=1,\ldots,\mu$.

Los algoritmos genéticos inician con una población P_g generada de forma aleatoria. Posteriormente, de manera iterativa, se calcula la aptitud de sus individuos y se aplican los **operadores genéticos** (selección, cruce y mutación) para determinar a los individuos que formarán la siguiente población P_{g+1}, donde g es el número de generaciones ó iteraciones que se han ejecutado. Generalmente, el criterio de parada es el arribo a una generación g_{max}, fijada antes de iniciar el algoritmo.

Materiales y métodos

CUDA

CUDA es una tecnología de GPGPU desarrollada por NVIDIA en 2006 cuando liberaron el lenguaje CUDA C que permite el desarrollo de aplicaciones de propósito general en paralelo utilizando los múltiples procesadores de la tarjeta gráfica [3]. La arquitectura CUDA se muestra en la figura 1.

Los elementos de la arquitectura CUDA son [4]: los *hilos* en donde se ejecutan las funciones simultáneamente y tienen su propio espacio de memoria; *bloques*, formados por un conjunto de hilos e incluyen un espacio de memoria compartida para su comunicación; una *malla*, que incluye los bloques que se ejecutan en un *dispositivo*, con distintas zonas de memoria comunes a todos los bloques (figura 1).

Una función que se ejecuta en la tarjeta gráfica (GPU) se denomina kernel. Cuando se lanza un kernel, es necesario informar a la GPU cuantos bloques (BLK) e hilos (THR) se ejecutarán, utilizando la siguiente sintaxis:

nombre_kernel<<<BLK,THR>>>(parámetros);

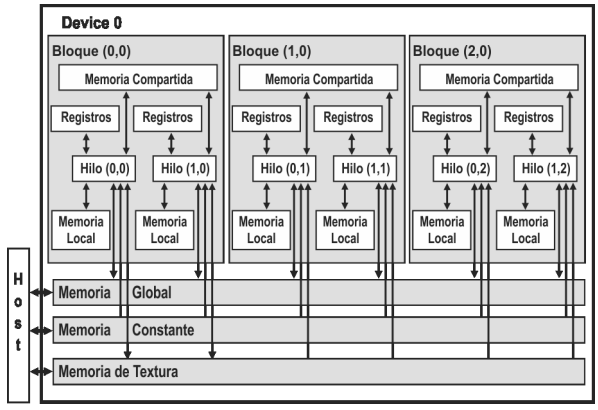

Figura 1. Arquitectura CUDA.

La GPU organiza el trabajo en sus multiprocesadores como se muestra en la figura 2 [4].

Para aprovechar al máximo la capacidad de procesamiento y evitar tener recursos desperdiciados, es importante que la cantidad de bloques sea un múltiplo de la cantidad de núcleos disponibles.

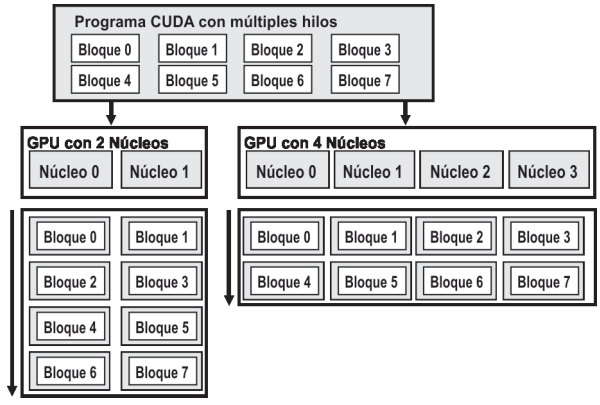

Figura 2. Distribución de bloques

En virtud de que las unidades de proceso son homogéneas, la asignación de carga se efectúa de manera estática y simétrica. Aunque la asignación de carga dinámica en estructuras de cómputo heterogéneas brinda la ventaja de una mayor eficiencia en el aprovechamiento del poder computacional, en el caso que nos compete, significaría agregar complejidad al algoritmo sin obtener beneficios de desempeño.

Implementación de PSO con CUDA

En cada bloque se procesó un enjambre de *t* partículas y cada hilo procesó una partícula, Figura 3.

Cada *PDB* iteraciones ocurre un proceso de dispersión de la información, que consiste en que cada bloque comunica la información de su mejor posición con una topología de anillo; como se muestra en la figura 4.

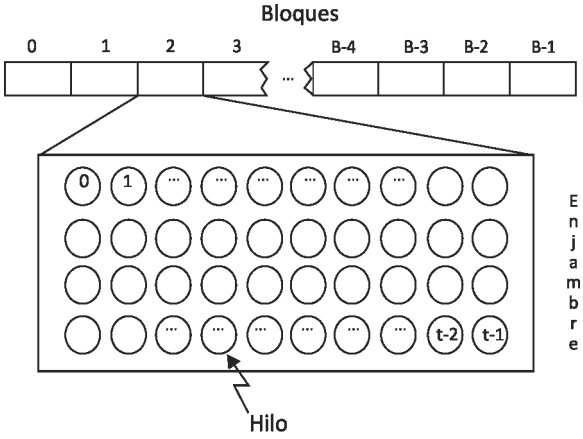

Figura 3. Asignación de enjambres en el GPU.

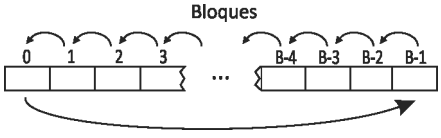

Figura 4. Dispersión.

En el siguiente pseudocódigo se muestra la asignación de tareas y se ilustra qué parte se ejecuta en el CPU y cuál se ejecuta en el GPU:

```
CPU:
Lanza el kernel para Crear BLK enjambres
para i = 1 hasta GMAX/PDB
    Lanza kernel Pso
    Lanza kernel Migración
Recibe la major posición de los BLK enjambres
Despliega la mejor posición global como la
solución del problema de optimización.

GPU:
KERNEL Crear
 para i=1 hasta P
  inicializa Xi, Vi y Fi
KERNEL Pso
 G = 0;
 Mientras que (G<PDB)
  G = G+1
  calcula Xi aptitud A(Xi)
  si A(Xi) es mejor que A(Xhi) entonces
    Xhi = Xi
  para i = 1 hasta tamaño_enjambre
    encuentra gbest del enjambre
    actualiza velocidad VI para Xi
    actualiza posición de Xi
KERNEL Dispersión
 Recibe la mejor posición de (BlockId+1) mod
 BLK
```

Implementación de AG con CUDA

En cada bloque se procesó una población de μ individuos y cada hilo procesó un individuo.

Se paralelizaron los operadores de selección, mutación y migración. El cruce se realizó en la GPU, utilizando un solo hilo para procesar todos los individuos.

La migración se llevó a cabo con un periodo del 10% del total de generaciones y una tasa de migración (MR) aproximada al 3% de la población.

En el siguiente pseudocódigo se muestra la asignación de tareas y se ilustra qué parte se ejecuta en el CPU y cuál se ejecuta en el GPU:

```
CPU:
Lanza el kernel para Crear BLK poblaciones
para i = 1 hasta GMAX/PMB
    Lanza kernel AG
    Lanza kernel Migración
Recibe la major posición de las BLK
poblaciones
Despliega la mejor posición global como la
solución del problema de optimización.

GPU:
KERNEL Crear
 para i=1 hasta MU
   inicializa Xi y Fi
KERNEL Pso
 G = 0;
 Mientras que (G<PMB)
  G = G+1
  ejecuta selección
  para i = 1 hasta tamaño_enjambre
    ejecuta cruce por torneo
  ejecuta mutación
KERNEL Ordenar
 para i=1 hasta MU
  para j=2 hasta MU-1
     si Xj mejor que Xi
        intercambia(Xi,Xj)
KERNEL Migración
  Para i=0 hasta MR-1
    Xi = XMR-i
    Fi = FMR-i
```

Primeramente se ordenó la población, para poder migrar los MR mejores del bloque siguiente en los MR peores individuos del bloque actual (ubicados al final del bloque), figura 5.

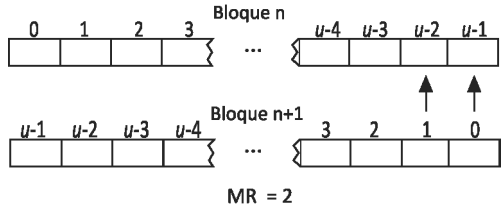

Figura 5. Migración. Se sustituyen los *MU* peores individuos con los mejores del bloque siguiente.

Funciones de prueba

Se optimizaron las funciones de Rastrigin, Ackley, Schwefel y Weierstrass, en 30 dimensiones propuestas, en la Competencia de Computación Evolutiva (CEC'05) [5].

La función de Rastrigin (5) es una función escalable, separable, multimodal con un óptimo global en el

origen y gran cantidad de óptimos locales. De acuerdo al CEC'05, el espacio de búsqueda es en el intervalo $[-5,5]^D$, donde D representa la cantidad de dimensiones del espacio de búsqueda.

$$f(x)=\sum_{i=1}^{D}(x_i^2-10\cos(2\Pi x_i)+10) \quad (5)$$

La función de Ackley (6) es un problema multimodal, no-separable y escalable. Tiene un óptimo local en el origen y un área de decremento muy pequeña cercana al punto óptimo. El espacio de búsqueda se define en el intervalo [-32,32] para cada dimensión D.

$$f(x)=-20\exp(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D}x_i^2})-\exp(\frac{1}{D}\sum_{i=1}^{D}\cos(2\Pi x_i))+20+e \quad (6)$$

La función de Schwefel (7) es un problema unimodal, desplazable, no-separable, escalable. El espacio de búsqueda se define en el intervalo [-100,100] para cada dimensión D.

$$f(x)=\sum_{i=1}^{D}(\sum_{j=1}^{i}x_j)^2 \quad (7)$$

La función de Weierstrass (8) es un problema multimodal, desplazable, no-separable, rotable escalable, continuo pero diferenciable solo en un conjunto de puntos. El espacio de búsqueda se define en el intervalo [-0.5,0.5] para cada dimensión D.

$$f(x)=\sum_{i=1}^{D}(\sum_{k=0}^{k_{max}}[a^k\cos(2\Pi b^k(x_i+0.5))])-D\sum_{k=0}^{k_{max}}[a^k\cos(2\Pi b^k\cdot 0.5)] \quad (8)$$

Resultados y discusión

Los algoritmos se implementaron utilizando GNU/Linux Fedora 13 de 32 bits, gcc 4.4.4.2 y las librerías de CUDA C versión 3.2, con un microprocesador Intel Core i5 650 @ 3.20GHz, 4 GB de memoria RAM a 1333 MHz, con una tarjeta de gráficos NVIDIA GeForce GTX 460 con 1GB de memoria global total, 7 multiprocesadores, 336 núcleos y una frecuencia de reloj de 810 MHz.

Para el caso de PSO, se lanzaron 21 bloques con 128 partículas.

Considerando los resultados en varios trabajos [4][7][8] se utilizó un valor de 1.62 para las constantes de confianza c_1, c_2 y 0.8 para el peso inercial w.

De acuerdo a los resultados obtenidos por Castro et al [9], se utilizó un periodo de dispersión entre bloques, PDB, del 1% del total de iteraciones.

En la tabla 1 se muestran los resultados obtenidos para las funciones propuestas con PSO, ejecutando 1500 muestras.

Donde: Gen=Generaciones, % Ex=Porcentaje de éxito obtenido, T=tiempo de ejecución, ErrMax=Error máximo permitido.

Función de:	Gen.	% Ex	T (s)	ErrMax
Ackley	12000	99.93	0.95	<2.5 e-6
	13000	100.00	1.02	
Rastrigin	74000	99.60	5.75	<1 e-6
	76000	100.00	5.90	
Schwefel	41000	99.80	5.24	<1 e-6
	42000	100.00	5.38	
Weierstrass	3000	99.20	5.37	<5.7 e-6
	3100	100.00	5.53	

Tabla 1. Resultados PSO

Para la implementación de AG se utilizaron 14 bloques con 64 individuos, un periodo migratorio del 10% del total de generaciones, probabilidad de cruce p_c de 0.16, probabilidad de mutación p_m de 0.7, tamaño del torneo z de 2.

En la tabla 2 se muestran los resultados obtenidos para las funciones propuestas con AG, ejecutando 1500 muestras.

Función de:	Gen.	% Ex.	T (s)	ErrMax
Ackley	50000	99.60	20.90	<2.5 e-6
	55000	100.00	22.99	
Rastrigin	7500	99.60	4.06	<1 e-6
	8000	100.00	4.33	
Schwefel	20000	--	46.59	<1 e-6
Weierstrass	6000	--	320.68	5.7 e-6

Tabla 2. Resultados AG

Conclusiones

La paralelización de problemas utilizando la arquitectura CUDA probó ser una excelente opción para la solución de este tipo de problemas, ya que con una sola tarjeta con un costo inferior a los 300 dólares se obtienen resultados de buena calidad con ahorros de tiempo cercanos al 50%, comparados con los de un cluster de 16 computadoras de doble núcleo conectadas mediante una red fast-ethernet.

En la figura 6 se muestra la ganancia en tiempo de la implementación en CUDA comparada contra la implementación en un cluster y la implementación en una sola computadora.

Debido a las limitaciones en cuanto a la memoria disponible en la GPU, como estrategia de paralelización se determinó utilizar varios bloques en los que se alojó una población con pocos individuos, (en el caso del Algoritmo Genético); o un enjambre con pocas partículas, (en el caso de la Optimización por Enjambre de Partículas).

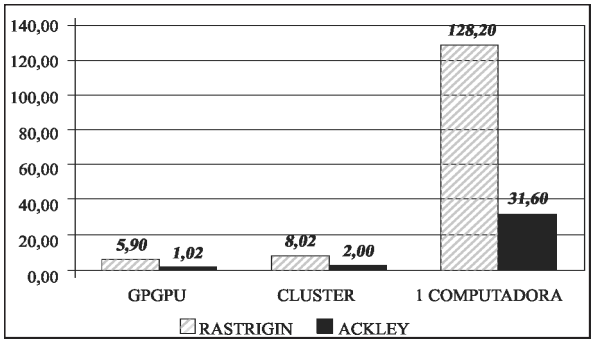

Figura 6. Comparación de tiempos de ejecución de PSO (s)

Para que los bloques pudieran colaborar a través de la migración o dispersión de información, según corresponda, fue necesario implementar en kernels separados las distintas etapas de los algoritmos; por este motivo se requirió sincronizar los hilos en el host, con el costo en tiempo de ejecución que ello implica.

Una limitante de la arquitectura es la precisión en los resultados de las funciones matemáticas implementadas para CUDA C, con error de hasta 8 ULP (Unit in the Last Place, dígito menos significativo) en algunos casos de funciones matemáticas para números flotantes de simple precisión, que puede reducirse a 3 ULP en su equivalente para números flotantes de doble precisión [10]. Para lograr optimizar la función Weierstrass, que requiere el uso de la función pow(), cuyo error es de 8 ULP en simple precisión y 3 ULP en doble precisión, y para no limitar aún más la cantidad de partículas (en el caso de Optimización por Enjambre de Partículas) o individuos (en el caso de Algoritmos Genéticos), se implementó la función de aptitud utilizando variables de doble precisión para realizar los cálculos y para el valor de retorno se definió una variable de simple precisión. Esta estrategia también se utilizó con éxito para la función de Schwefel.

Para la mayoría de las funciones probadas en este trabajo es preferible utilizar enjambres de partículas en lugar de algoritmos genéticos, pues su desempeño, en cuanto a tiempo de ejecución, es mejor. Sólo en el caso de la función de Rastrigin, debido a que tiene una enorme cantidad de óptimos locales, el algoritmo genético mostró un mejor desempeño que el de PSO.

El Algoritmo Genético no pudo obtener un resultado aceptable para las funciones de: Schwefel, que aún excediendo en 40 segundos el tiempo de PSO no pudo obtener el óptimo; y Weierstrass, que excedió en más de 300 segundos el tiempo de PSO, sin obtener el resultado óptimo.

Trabajo Futuro

Actualmente se está trabajando en la optimización de los parámetros de un sistema de identificación difusa para modelar un proceso fermentativo (embriogénesis somática).

Será interesante la implementación con varios GPU´s trabajando colaborativamente, en un solo equipo de cómputo ó instalando un cluster de computadoras.

Referencias

[1] Castro M.A., J.A. Morales, I. Castro, J.A. Castro, y L.A. Cárdenas, (2011), "Distributed Particle Swarm Optimization Using Clusters and GPGPU", *I Congreso Internacional de Ingeniería Electrónica y Computación* (Minatitlán, Veracruz, México) (22-24 de Junio de 2011).

[2] Castro I., M.A. Castro, and J.A. Castro, (2011), "Parallel particle swarm optimization using GPGPU", *Evento de Automática y Sistemas Computacionales de la XIV Convención de Ingeniería Eléctrica (CIE2011)*, (Universidad Central de Las Villas, Cuba) (14-18 de Junio de 2011).

[3] NVIDIA Corp., (2009), "CUDA Architecture Introduction & Overview", NVIDIA Cor., Santa Clara California. Version 1.1.

[4] NVIDIA Corp., (2010), "CUDA C Programming Guide Version 3.2", NVIDIA Cor., Santa Clara California.

[5] Kennedy, J. y Eberhart, C., (1995), *Proceedings of IEEE International Conference on Neural Networks* Particle Swarm Optimization. En Proceedings, volume IV, páginas 1942-1948.

[6] Van de Berg, Frans, (2001) *"An Analysis of Particle Swarm Optimizers"*, PhD Dissertation Faculty of Agricultural and Natural Science, Pretoria University, Pretoria South Africa.

[7] Vose, M.D., (1999), *The Synple Genetic Algorithm: Foundations and Theory*. MIT Press.

[8] Sugantan, P. N., N. Hansen, et al. (2005). Problem Definitions and Evaluation Criteria for the CEC 2005, Special Session on Real-Parameter Optimization. Singapore, Kanpur Genetic Algorithms Laboratory, Nanyang Technological University: 40.

[9] Clerc, M., (2006), *"Particle Swarm Optimization"*, 1st English Ed. Newport Beach ISTE.

[10] Mussi, L Daolio, F and Cagnoni, S., (2009) "GPU-based Road Sign Detection Using Particle Swarm Optimization", *9th International Conference on Intelligent Systems Design and Applications*, (Pisa, Italia) (30 de Noviembre al 2 de Diciembre de 2009) p. 152-157.

[11] NVIDIA Corp., (2010), "CUDA C Programming Guide.", NVIDIA Cor., Santa Clara California. Version 3.2.

Artículo recibido: 5 de julio de 2011

Aceptado para publicación: 9 de abril de 2012