Análisis del comportamiento de un AG para GPUs

Carlos Bermúdez, Carolina Salto

Facultad de Ingeniería - Universidad Nacional De La Pampa
bermudezc@yahoo.com, saltoc@ing.unlpam.edu.ar
http://www.ing.unlpam.edu.ar

Resumen Este trabajo presenta un algoritmo genético simple ejecutándose sobre GPU y empleando la tecnología CUDA para resolver el problema MaxCut. Se realiza un estudio empírico del impacto en el rendimiento del algoritmo en la elección de distintos operadores de cruce para representaciones binarias. Las pruebas mostraron un buen desempeño de las distintas variantes planteadas, aunque una mejor calidad de resultados se obtuvo con la variante utilizando un cruce de dos puntos de corte. El paso siguiente fue contrastar el rendimiento de este algoritmo con una misma versión pero ahora ejecutándose en serie sobre CPU y así poder determinar la ganancia de tiempo, reflejada por el *speedup*. Los resultados obtenidos indican que la ganancia en tiempo está relacionada con la densidad del grafo que representa cada instancia del MaxCut.

Keywords: Algoritmo genético, GPU, CUDA, MaxCut

1. Introducción

Los algoritmos genéticos (AG) desarrollados por Holland en 1975 [1] son una herramienta muy potente cuando de optimizar se trata. Y sus beneficios están ampliamente demostrados en la bibliografía existente. Los AG son algoritmos de búsqueda estocásticos basados en población y para obtener buenos resultados se necesita, a menudo, crear y evaluar muchas soluciones candidatas. Por otra parte, estos algoritmos son inherentemente paralelos debido a que una solución candidata puede crearse o evaluarse de forma independiente a las demás. Por otra parte, las Unidades de Procesamiento Gráfico (GPU) ofrecen un gran poder de cómputo a precios accesibles. Estas unidades, que pueden ser fácilmente programadas, a través de una extensión del lenguaje C (CUDA (Compute Unified Device Architecture) [4]), permiten implementar programas que hagan uso de las capacidades de las GPU. A pesar de que estos chips fueron diseñados para acelerar la rasterización de primitivas gráficas, su rendimiento ha atraído a una gran cantidad de investigadores que los utilizan como unidades de aceleración para muchas aplicaciones científicas [2]. Esto da lugar a que el uso de las GPU como soporte para la ejecución de AGs se esté popularizando. En comparación con una CPU, las GPU más recientes son unas 15 veces más rápidas que los procesadores de seis núcleos de Intel en operaciones de punto flotante de precisión

simple [12]. Dicho de otra manera, un grupo con una sola GPU por nodo ofrece un rendimiento equivalente a un cluster de 15 nodos de un sólo CPU.

En este trabajo, diseñamos un AG que se ejecuta completamente sobre una GPU (AG-GPU) para resolver el problema de corte máximo de un grafo, conocido como MaxCut. Es un problema NP-duro muy conocido y además de su importancia teórica, tiene aplicaciones en varios campos y ha sido reformulado de varias maneras. El objetivo de este trabajo es medir el desempeño del AG diseñado para ejecutar en CPU teniendo en cuenta dos tipos de crossover (un punto y dos puntos), para tratar de determinar si el beneficio obtenido en la calidad de los resultados con un operador de dos puntos es más conveniente que utilizar la versión con un solo punto de corte. Por último, se calcula el speed-up del algoritmo, para lo cual se ejecutó el algoritmo en su versión serie sobre CPU y con esto poder contrastar los tiempos obtenidos.

El trabajo se organiza de la siguiente manera. La Sección 2 introduce el problema de optimización utilizado MaxCut y la descripción del algoritmo genético utilizado en la experimentación. En la Sección 3 se detalla la parametrización utilizada en la experimentación y en la Sección 4 el análisis de los resultados obtenidos. Finalmente la Sección 5 presenta las conclusiones alcanzadas y la propuesta de trabaja futuro.

2. Algoritmo Genético para GPU

Esta sección describe los detalles de implementación del AG para GPU utilizado en el trabajo para resolver el problema de corte máximo de un grafo, denominado MaxCut. Por ello esta sección se estructura en dos partes: la primera está dedicada a la descripción del problema y la segunda a introducir detalles de implementación del algoritmo AG-GPU desarrollado para resolver tal problema.

2.1. Problema MaxCut

El problema MaxCut [5] es un problema que pertenece a la clase de los NP-Completos, y se define de la siguiente manera:

Definition 1. *Dado un grafo no dirigido G=(V,E) y pesos no negativos $w_{ij} = w_{ji}$ en los arcos $(i, j) \in E$, el problema consiste en encontrar un conjunto de vértices S que maximicen el peso de los arcos en el corte $(S,\overline{S})$, esto es, el peso de los arcos que tienen un vértice en S y el otro en $\overline{S}$.*

El peso del corte queda denotado por $w(S, \overline{S}) = \sum i \in S, j \notin Sw_{ij}$, y como se menciona en la definición, dicho peso tendrá que ser máximo.

2.2. Descripción del algoritmo AG-GPU

Aprovechando las ventajas brindadas por la GPU para acelerar el proceso de optimización, se implementaron todos los operadores del AG-GPU para que se ejecuten como *Kernel* en la GPU. Por lo tanto, el bucle principal de este AG

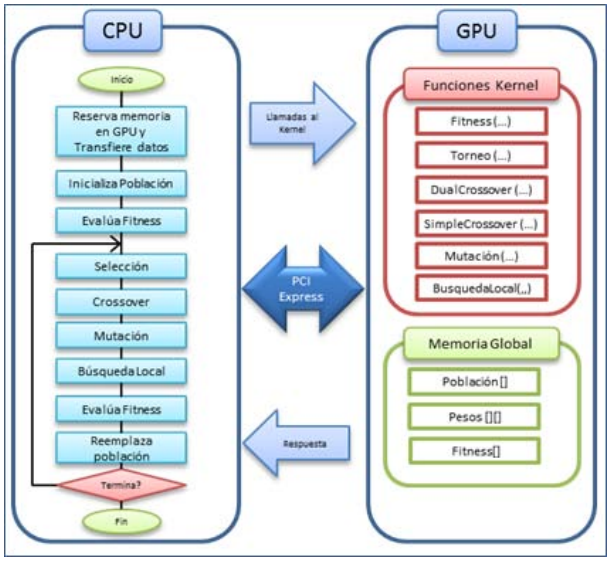

Figura 1. Configuración del AG en CPU y GPU con CUDA

se implementó para que corra completamente en el *host* y todos los operadores genéticos se implementaron para ser ejecutados en la GPU. En la Figura 1 se puede observar esta configuración. El primer paso para trabajar con la GPU es reservar la memoria necesaria donde se alojarán los arreglos utilizados. Es importante resaltar que la GPU no posee un generador de números aleatorios, por lo tanto se debe crear un vector en la CPU, llenarlo con los valores deseados y luego transferirlos a la GPU. De la misma forma se debe enviar como parámetro la posición que se utilizará de este arreglo. Para la representación de la población se utilizó codificación binaria basada en nodos, es decir que si un bit está en uno significa que el nodo que está representando ese bit forma parte del conjunto S. La representación binaria es preferible a trabajar directamente con enteros ya que, a pesar de ser un poco más complejo su manejo, utiliza mucho menos memoria. El uso de la memoria que se utiliza en el *device* es crítico, ya que se tiene que estar intercambiando información entre el *host* y el *device* continuamente.

Para almacenar un individuo se utilizó el tipo de dato unsigned char que representa la mínima pieza de información denominada palabra. Por ejemplo, un individuo que represente una instancia de 10.000 nodos, serían necesarias 1.250 palabras del tipo unsigned char para ser almacenado. Se debe contemplar también, que si el tamaño de la instancia no es múltiplo del tipo de dato van a existir datos inválidos al final de la última palabra. Para inicializar cada individuo se genera cada palabra con ceros y unos de forma aleatoria. Al evaluar el fitness del individuo se deben recuperar sus bits uno por uno, para esto, se debe utilizar una máscara auxiliar (2unsigned char -1) y se efectúa un AND con la palabra que se está evaluando, luego se desplaza a la derecha la máscara con el

operador >> y se vuelve a comparar, así hasta el final del individuo. Cuando un bit del individuo tiene que ser mutado, se necesita primero saber en qué palabra se encuentra y la posición que ocupa en la palabra. Luego se debe desplazar hacia la derecha la máscara hasta la misma posición de este bit y se efectúa un XOR entre la máscara y la palabra seleccionada.

Tanto para el operador de torneo como el de mutación se lanza un thread por cada individuo. En el operador de torneo binario cada thread evalúa el fitness de los individuos involucrados en el torneo y entonces almacena la posición del ganador en la memoria global para que luego pueda ser utilizado por el operador de crossover. En la mutación, cada thread decide cuándo el individuo tiene que ser mutado, utilizando para esto, un valor random de la memoria global. Si el individuo tiene que ser mutado se utiliza otro valor random de la memoria para ubicar el bit a alterar.

Como el objetivo del trabajo se centra en analizar cual operador de crossover es el más adecuado para resolver el problema, es que se consideraron dos operadores: el operador de un punto y el operador de dos puntos. Además como es sabido dentro de la comunidad de computación evolutiva , el rendimiento del algoritmo está fuertemente vinculado con el tipo de operador seleccionado [13].

El crossover de un punto (SPX) selecciona un punto en el vector de los padres. Todos los bits más allá de este punto se intercambiarán entre los dos padres. Para implementar el SPX se necesita calcular en qué palabra y en qué posición de ésta se hará el corte. Si la posición es al comienzo de la palabra simplemente se copiarán las palabras completas y no son necesarias operaciones a nivel de bit. En caso contrario se necesitan operaciones a nivel de bit en la palabra seleccionada y las palabras restantes se copian de forma directa. La idea del algoritmo es desplazar los bits hacia un lado para descartar la información innecesaria y luego desplazarlos en sentido contrario para dejar los bits requeridos en la ubicación correcta.

El crossover de dos puntos (DPX) requiere seleccionar dos puntos en los vectores de los padres. Todos los datos entre los dos puntos se intercambian entre los padres, creando dos hijos. El crossover de dos puntos es similar al SPX pero es necesario contemplar situaciones especiales. En el caso más general, los dos puntos de corte caen dentro de palabras distintas y se debe aplicar el mismo procedimiento que SPX para ambas palabras. Por otra parte cuando los dos puntos de corte caen dentro de la misma palabra las operaciones de desplazamiento para obtener el bit requerido se deben efectuar sobre la misma palabra. Otro caso especial que requiere atención, sobre todo para ganar eficiencia, es cuando los dos puntos de crossover caen en la misma palabra y al comienzo de ésta.

El operador de crossover sobre GPU se organizó en forma de bloques de threads, la cantidad de bloques es igual a la mitad de la población. Cada thread maneja varios elementos de la solución, para esto toma el identificador de bloque *blockid* al que pertenece y copia el primer bit en la posición *threadid*, luego el segundo bit en la misma posición del segundo bloque y así hasta completar la solución. Para mejorar el rendimiento del operador se utilizó acceso coalescente

a memoria global, así los threads contiguos acceden a localizaciones de memoria adyacentes.

Después de aplicar los operadores genéticos de crossover y mutación, los nuevos individuos creados pasan a un proceso de optimización local con una cierta probabilidad. Este algoritmo es una variación eficiente de la fase de búsqueda local propuesta por Festa et al. [6] . Cuando es aplicado sobre una solución x, el procedimiento comienza creando un conjunto T de todos los nodos i. El procedimiento iterativo selecciona de forma aleatoria un nodo i del conjunto T y entonces cambia este nodo desde un subconjunto al otro en x de acuerdo a las siguientes reglas:

$$\textbf{If} \quad (xi = 0 \, and \, \sigma s(i) - \sigma \overline{s}(i) > 0) \quad \textbf{then} \quad xi = 1 \tag{1}$$

$$\textbf{If} \quad (xi = 1 \, and \, \sigma \overline{s}(i) - \sigma s(i) > 0) \quad \textbf{then} \quad xi = 0 \tag{2}$$

Donde, para cada nodo $i : 1, \ldots, n$; $\sigma s(i) = \sum_{j \in S} w_{ij}$ y $\sigma \overline{s}(i) = \sum_{j \in \overline{S}} w_{ij}$ representan el cambio en la función objetivo asociada con mover un nodo i desde un subconjunto del corte al otro. Todos los posibles movimientos de una solución son investigados simplemente haciendo un intercambio entre todos los nodos. La solución actual es reemplazada por la mejorada. El seudocódigo del procedimiento de búsqueda local se puede observar en el algoritmo 1.

Algoritmo 1: Heurística de búsqueda local

```
T ← V ;
while T ≠ 0 do
    i ← verticealeatorioenT ;
    if (i ∈ s)and(σ_s(i) − σ_s̄(i) > 0) then
        S ← S\ {i} ;
        S̄ ← S̄∪ {i} ;
    end
    if (i ∈ s̄)and(σ_s̄(i) − σ_s(i) > 0) then
        S̄ ← S̄\ {i} ;
        S ← S∪ {i} ;
    end
    T ← V{i} ;
end
```

Para el funcionamiento de este operador sobre GPU se lanza un thread por cada individuo en donde se evalúa el beneficio de cambiar un nodo de subconjunto, de ser positivo el intercambio se incorpora a la solución y se continua evaluando el siguiente nodo hasta terminar con el cromosoma.

3. Experimento

Para realizar la experimentación se utilizó el conjunto de instancias generadas por Helmberg y Rendl [7]. Éstas consisten en grafos toroidales planos y generados aleatoriamente variando el tamaño y la densidad, con pesos que toman el valor 0, 1 o -1. El tamaño del grafo V varía de 800 a 2000 nodos. La densidad fluctúa

desde 0.25 % a 6.12 % como se observa en la Tabla 1. Festa et al. [6], Martí et al.[8] , y Arráiz et al [9] usan estos grafos en sus experimentos, por lo tanto es una elección conveniente para propósitos comparativos.

Cuadro 1. Cantidad de nodos y densidad para cada instancia.

Instancias	—V—	Densidad (%)
G1, G2, G3	800	6.12
G11, G12, G13	800	0.63
G14, G15, G16	800	1.58
G22, G23, G24	2000	1.05
G32, G33, G34	2000	0.25
G43, G44, G45	1000	2.10

Para seleccionar el valor correspondiente a cada uno de los parámetros que controlan a los operadores genéticos, se efectuaron diferentes pruebas con las instancias G1 y G22 (se las consideraron representativas del conjunto ya que poseen distintas cantidades de nodos y de densidad). Estas pruebas consistieron en variar los siguientes parámetros: probabilidad de mutación, probabilidad de crossover, tamaño de la población, cantidad de generaciones y probabilidad de la búsqueda local. Para cada una de estas pruebas se efectuaron 30 ejecuciones y se tomaron los promedios. Los parámetros utilizados fueron: población de 200 y 480 individuos, 100, 200 y 300 generaciones y las probabilidades de los operadores genéticos tomaron valores de 60 %, 80 % y 100 %. Los resultados de estas pruebas sugirieron que los siguientes valores son los que arrojaron la mejor calidad de resultados: tamaño de población igual a 480 individuos, 200 generaciones, probabilidad de mutación igual al 100 % (todos los individuos se mutan y cada gen tiene una probabilidad 1/—V— de ser cambiado), probabilidad de crossover del 60 % y probabilidad de la búsqueda local igual al 100 % (por motivos de espacio no se incluyen las tablas con estos resultados). Estos resultados fueron sometidos a una validación estadística utilizando tests no paramétricos con un valor de significancia del 95 %. Si bien una probabilidad del 100 % para la búsqueda local puede parecer extraña, este mecanismo ayuda al algoritmo a converger en forma más temprana sin caer en óptimos locales.

Los resultados mostrados en la siguiente Sección son el promedio de 30 ejecuciones independientes. Se efectuó un análisis de estadístico por medio del test no-paramétrico Kruskal Wallis para distinguir las diferencias más relevantes a través de la media de los resultados para cada algoritmo. Se consideró un nivel significativo de $\alpha = 0.05$, para indicar un nivel de confianza del 95 % en los resultados.

Los algoritmos fueron implementados en C++ utilizando la librería CUDA, y fueron ejecutados en una máquina con un microprocesador Intel© Core™ i7 2600 CPU @ 3.40GHZ con 4GB de RAM y una placa de video G-FORCE GTX-580 con 512 cores y 3 GB de memoria.

Cuadro 2. Resultados de las variantes de AG-GPU.

	AG-GPU_SPX			**AG-GPU_DPX**			**test**
	Mejor	**Media**	σ	**Mejor**	**Media**	σ	
G1	11624	11561.77	± 25.82	11624	11589.10	± 19.63	+
G2	11605	11557.87	± 15.22	**11616**	11588.83	±11.95	+
G3	11602	11568.97	± 15.15	**11617**	11595.60	± 12.82	+
G11	**556**	547.67	± 3.83	520	498.33	± 11.03	+
G12	514	496.80	±10.30	**546**	537.33	± 4.18	+
G13	550	527.73	± 8.51	**572**	560.87	±5.35	+
G14	**3058**	3049.43	± 3.80	3048	3034.77	± 5.50	+
G15	**3036**	3028.43	±3.52	3032	3016.53	± 7.23	+
G16	**3039**	3032.80	±3.40	3033	3023.07	± 5.67	+
G22	13248	13187.80	±24.75	**13309**	13268.27	± 18.14	+
G23	13259	13192.23	±30.21	**13309**	13273.47	± 14.14	+
G24	13241	13184.93	±30.58	**13301**	13265.60	± 20.14	+
G32	1234	1187.67	±17.68	**1336**	1322.93	± 8.17	+
G33	1188	1155.73	±16.31	**1318**	1296.93	±11.55	+
G34	1200	1155.67	±17.97	**1314**	1295.80	±9.21	+
G43	6630	6601.30	± 14.56	**6649**	6625.43	± 12.41	+
G44	6626	6592.10	±16.01	**6644**	6626.20	±10.20	+
G45	6641	6596.70	±17.71	**6644**	6622.17	±13.75	+

4. Análisis de resultados

Esta sección está organizada en dos partes. La primera está orientada a determinar cuál variante del AG-GPU es la que mejor rendimiento obtiene. La segunda parte muestra una comparativa en el rendimiento del AG-GPU y su versión secuencial ejecutada en serie en CPU.

En el Cuadro 2 se puede observar el comportamiento de las dos variantes del AG-GPU para cada una de las instancias del MaxCut. En el Cuadro se muestra la siguiente información: mejor solución encontrada por el AG-GPU en las 30 ejecuciones (columna Mejor), promedio de las mejores soluciones (columna Media) junto con el desvío estándar (columna σ). En negrita se resaltan las mejores soluciones para cada instancia. Comparando las mejores soluciones del AG-GPU_DPX y AG-GPU_SPX, se puede distinguir el buen desempeño del AG-GPU_DPX, ya que en 14 de las 18 instancias obtuvo mejores resultados que AG-GPU_SPX. Este mejor desempeño se hace más evidente en las instancias que tienen una menor complejidad, como son las instancias G32, G33 y G34 que poseen una densidad de 0,25 % y el mejor valor encontrado supera en más de 100 unidades al de AG-GPU_SPX. Por el contrario, en las instancias con mayor densidad como las G1, G2 y G3, que tienen una densidad de 6,12 % , la ganancia que se obtiene está en el orden de las 10 unidades. Por último, no parece haber una relación directa entre la cantidad de nodos y la calidad de los resultados, ya quelos mejores resultados se obtuvieron en instancias que tienen 2000 nodos, mientras que en las que se arrojaron valores menores tienen 800 nodos. Por otra parte, podemos observar que existen diferencias estadísticamente significativas entre las dos variantes de AG-GPU, ya que los respectivos valores de p para la prueba estadística es mayor que el nivel de confianza de 0,05 (símbolo + en la columna test).

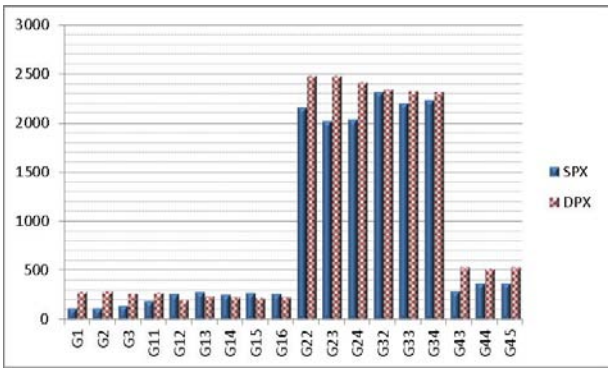

Figura 2. Comparación de los tiempos de ambos operadores.

En las Figuras 2 y 3 se puede observar los tiempos y las evaluaciones que consumieron las distintas variantes de AG. En general, la variante AG-GPU_DPX insume más tiempo y cantidad de evaluaciones que AG-GPU_SPX, excepto en las instancias G12, G13, G14, G15 y G16. Estas últimas también se corresponden con las que AG-GPU_DPX obtuvo peores resultados. Estas diferencias fueron corroboradas con los estudios estadísticos realizados.

En resumen, la variante AG-GPU_DPX fue la que mejor comportamiento ha exhibido (en 14 de las 16 instancias analizadas), por lo que se utilizará para realizar la siguiente experimentación.

Otro de los ejes de este trabajo es medir la ganancia de tiempo que se obtiene de ejecutar el algoritmo en paralelo (ambiente GPU) respecto de su ejecución en serie (entorno CPU). Para ello se efectuó la misma experimentción y con los mismos parámetros utilizados en la versión paralela, pero ahora con un algoritmo modificado para que pueda correr completamente en un CPU, esta implementación es con un único hilo de ejecución. Con estos resultados se pueden relacionar los mejores tiempos de la versión paralela con los mejores tiempos de su versión serie, y así obtener una medida de rendimiento conocida como *speed-up* [12]. En la Figura 4 se muestran los valores obtenidos para esta métrica para cada una de las instancias. En este caso se observa que el valor de *speed-up* está muy relacionado con la densidad que exhibe cada una de las instancias. Por ejemplo, el *speed-up* es mucho mayor a medida que aumenta la densidad de las instancias, las instancias G32, G33 y G34 que presentan una densidad de 0,25 % se obtiene un valor de *speed-up* de 1.18 mientras que en las instancias más densas G1,G2 y G3 con 6,12 % de densidad, el valor de *speed-up* es de 17, aproximadamente. Si bien los valores de *speed-up* no son importantes, vale destacar que la comparación se realiza con un procesador de última generación muy rápido.

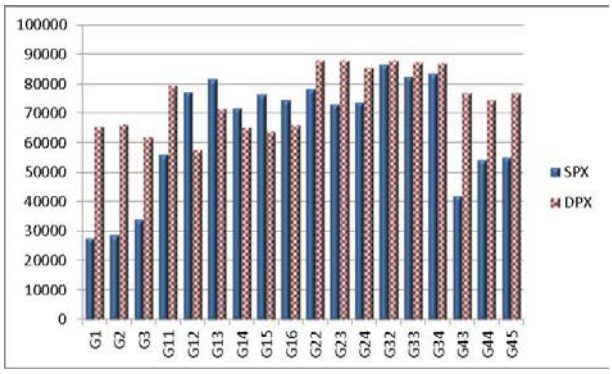

Figura 3. Comparación de las evaluaciones de ambos operadores.

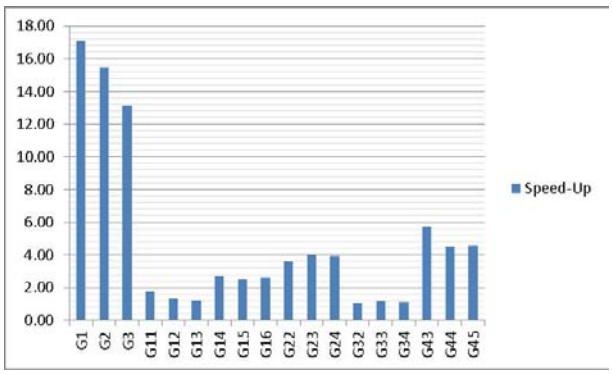

Figura 4. Speed-Up

5. Conclusiones

En este trabajo hemos presentado una implementación de un AG para GPU (AG-GPU) para resolver el problema de MaxCut. Para implementar el AG para GPUs se utilizó la tecnología CUDA, obteniéndose conocimiento sobre el manejo de AGs y su optimización en estos entornos. Se analizó el comportamiento del AG para GPUs con dos tipos de crossover: de uno (SPX) y dos (DPX) puntos. Hemos analizado el rendimiento del algoritmo utilizando 16 instancias del problema MaxCut. La variante AG-GPU_DPX mejora notablemente la calidad de las soluciones obtenidas por AG-GPU_SPX. El speedup de nuestra implementación fue investigada usando una tarjeta GeForce GTX 580 y un procesador rápido Intel Core i7 920 a 3.4 GHz. Para nuestra opción los valores de *speed-up* varían entre 1 y 17, y están relacionados con la densidad del grafo de corte que representa cada instancia. Una de las ventajas de nuestra implementación es que se puede ejecutar en cualquier GPU nVidia y plataforma Linux/Windows.

Nuestro trabajo futuro estará enfocado sobre estrategias de optimización específicas de CUDA para mejorar de ser posible los valores de speedup. También nuestra investigación estará orientada a la implementación de algoritmos genéticos paralelos, en especial bajo el modelo isla, para una plataforma GPU usando la tecnología CUDA.

Reconocimientos

Los autores agradecen el apoyo de la UNLPam y ANPCYT (proyecto 09F-049) y PICTO-UNLPAm (0278).

Referencias

1. Holland J., Adaptation in Natural and Artificial Systems. University of Michigan: Press, Ann Arbor 1975
2. Kirk D., Mei W., Hwu W., "Programming Massively Parallel Processors,ïn A Hands-on Approach., 2010, p. 280.
3. Lee V. et al, "Debunking the 100X GPU vs. CPU myth architecture ,ïn Proceedings of the 37th annual international symposium on Computer ISCA '10., 2010, p. 451.
4. NVIDIA, ÇUDA Toolkit 4.0"., http://developer.nvidia.com.
5. Karp R., Reducibility among combinatorial problems., 85103rd ed., Thatcher J (eds) Complexity of computer computations. In: Miller R, Ed., 1972.
6. Pardalos P., Resende M., Ribeiro C., Festa P., Randomized heuristics for the MAX-CUT problem. Optimization Methods and Software, 2002.
7. Rendl F., Helmberg C., A spectral bundle method for semidefinite programming, 2000.
8. Duarte A., Laguna M., Martí R., Advanced scatter search for the Max-Cut problem. , 2112638th ed., 2009.
9. Olivo O., Arráiz E., Competitive simulated annealing and tabu search algorithms for the Max-Cut problem. In: GECCO '09:proceedings of the 11th annual conference on genetic and evolutionary, computation. ACM, New York, pp 1797–1798, 2009.
10. Williamson D., Goemans M., Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. J ACM 42(6):1115–1145, 1995.
11. Salto C., Alba E., Designing heterogeneous distributed AGs by efficiently self-adapting the migration period, DOI 10.1007/s10489-011-0297-9.
12. Lee V. et al., "Debunking the 100X GPU vs. CPU myth," in Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10, 2010, p. 451.
13. Nannen, V. and Smit, S.K. and Eiben, A.E., "Costs and Benefits of Tuning Parameters of Evolutionary Algorithms," in Proceedings of the 10th international conference on Parallel Problem Solving from Nature: PPSN X, 2008, p. 528-538.