Síntesis booleana con programación genética paralela en CPU y GPU

Genetic parallel programing-based Boolean synthesis with CPU and GPU

Recibido: 27 de Enero de 2013
Aprobado: 9 de Febrero de 2013

César A. Pedraza*, Jaime V. Oyaga**, Ricardo C. Gómez***

Resumen

La síntesis booleana o combinacional es un proceso mediante el cual se optimiza una red de puertas lógicas, con el fin de reducir su consumo, minimizar costos, minimizar área y aumentar el rendimiento a la hora de ser implementada. Por otra parte, la programación genética es una alternativa importante para generar estructuras de *hardware* interesantes y eficientes. Se ha demostrado que los algoritmos evolutivos (AE) tienen mejor rendimiento si se implementan en sistemas paralelos. Este artículo presenta la implementación de un algoritmo genético paralelo (PGP) para realizar síntesis booleana en una plataforma basada en CPU-GPU. Esta implementación emplea el modelo de islas, el cual permite la evolución paralela e independiente del PGP a través de las múltiples unidades de procesamiento de la GPU y los múltiples núcleos de un procesador de última generación. Se probaron diferentes alternativas de mapeo del PGP en la plataforma en orden de optimizar el tiempo de respuesta. Como resultado se muestra una aceleración superior a 41.

Palabras clave

Programación paralela, síntesis booleana, GPU, algoritmo evolutivo.

* Ingeniero Electrónico, Universidad Santo Tomás, Bogotá. Magíster en Ingeniería Electrónica y de Computadores, Universidad de los Andes y Doctor en Ingeniería Informática, Universidad Rey Juan Carlos de Madrid, España. Profesor de tiempo completo, Universidad Santo Tomás. E-mail: cesarpedraza@usantotomas.edu.co

** Ingeniero Electrónico, Universidad Santo Tomás, Bogotá. Especialista en Instrumentación Electrónica, Universidad Santo Tomás y Magíster en Teleinformática, Universidad Distrital. Profesor de tiempo completo Universidad Santo Tomás. E-mail: jaimevitola@usantotomas.edu.co

*** Ingeniero Electrónico, Universidad Nacional de Colombia. Candidato a Magíster en Ingeniería Electrónica y de Computadores, Universidad de los Andes. Profesor de tiempo completo Universidad Santo Tomás. E-mail: ricardogomez@usantotomas.edu.co

Abstract

The Boolean or combinational synthesis is a process that optimizes a logic gates network, in order to reduce power consumption, minimize costs, minimize area and increase the performance when it is implemented. Moreover genetic programming is an important alternative to generate interesting and efficient *hardware* structures. It has been shown that evolvable algorithms are faster when implemented in parallel systems. This paper presents the implementation of a parallel genetic programming (PGP) for boolean synthesis on a GPU-CPU based platform. Our implementation uses the island mode which allows the parallel and independent evolution of the PGP through the multiple processing units of the GPU and the multiple cores of a new generation desktop processors. We tested multiple mapping alternatives of the PGP on the platform in order to optimize the PGP response time. As a result we show that our approach achieves a speedup up to 41.

Keywords

Parallel programming, Boolean synthesis, GPU, evolutionary algorithm.

I. Introducción

El principal objetivo de la síntesis booleana es encontrar expresiones booleanas compactas como suma de productos compuesta por la menor cantidad de variables y términos posible. Esta técnica se emplea para encontrar expresiones óptimas que superen las dificultades de las técnicas tradicionales del álgebra booleana.

Por su parte, los algoritmos evolutivos son un mecanismo de optimización inspirado en el comportamiento evolutivo de la naturaleza, que busca solucionar problemas que no logran otras estrategias de optimización. Su funcionamiento se basa en el proceso de la evolución de las especies formulado por Darwin [1], donde los individuos tienen la capacidad de replicarse mediante operadores genéticos. Estos operadores genéticos dan la posibilidad de realizar variaciones en el genoma (cadena de ADN) durante generaciones de una población, para poder encontrar individuos óptimos que se adapten plenamente a la naturaleza. Esta estrategia es usada para resolver problemas en donde el espacio de búsqueda no es continuo, o incluso, en sistemas NP completos, como es el caso de la reducción y satisfacción booleana.

Hallar una solución satisfactoria puede tomar poco tiempo, sin embargo, de acuerdo con el tamaño del problema, la carga computacional puede tener una magnitud tal, que solo después de días, o incluso años de evaluación, se pueda encontrar una solución adecuada, por lo que una buena alternativa para reducir el tiempo en encontrar una solución es una implementación paralela como lo describe Cantú-Paz en [2].

En este artículo se propone un sistema para la solución al problema de la carga computacional y del tiempo de respuesta de la síntesis booleana. Para ello, se empleó la programación genética paralela como algoritmo evolutivo, el cual se implementó en una arquitectura compuesta por una CPU y varias GPU. Para encontrar la configuración con

el mejor desempeño se empleó el modelo de islas y se probaron varios esquemas para su implementación en las plataformas.

II. Síntesis booleana con AE

La síntesis booleana es un proceso mediante el cual se optimiza una red de puertas lógicas, con el fin de reducir su consumo, minimizar costos, minimizar área y aumentar el rendimiento a la hora de ser implementada. Existen diversas técnicas para síntesis booleana, entre las más destacadas se encuentran los mapas de Karnaugh, el algoritmo de Quine-McCluskey, el algoritmo Reed-Muller y otros métodos heurísticos. Sin embargo, algunos de estos algoritmos presentan desventajas, tales como su crecimiento exponencial, falta de restricciones y múltiples soluciones a un mismo problema.

Por su parte, los algoritmos y la programación genética ofrecen una alternativa al problema de la síntesis booleana, produciendo nuevas estructuras para la implementación de bloques combinacionales que no resultarían mediante los métodos tradicionales mencionados en Cheang, Lee y Leung [3], Jozwiak, Ederveen y Postula [4], Nadia, Ajith y Luiza [5].

Adicionalmente, mediante la estrategia evolutiva en la síntesis booleana, se pueden añadir al algoritmo restricciones, tales como la velocidad de propagación, las puertas lógicas empleadas, el área, entre otros, mejorando los resultados para cada problema. La síntesis booleana es una herramienta útil en el diseño de *hardware* evolutivo (HE), pero presenta problemas si su implementación es *onchip*, dado que la carga computacional requerida para un algoritmo evolutivo aplicado a la síntesis booleana puede ser alta como lo demuestra Stoica [6,7], lo que representa una dificultad a la hora de ser desarrollados en sistemas embebidos. Hay implementaciones relacionadas con la síntesis booleana mediante algoritmos genéticos (AG) y programación genética (PG) realizados de forma intrínseca por Thompson [8] y extrínseca Kalganova [9] pero con un número muy reducido de variables (cuatro variables), orientados a obtener estructuras distintas de componentes básicos y donde se proponen plataformas de *hardware* y *software* para el desarrollo del algoritmo evolutivo. También se han hecho esfuerzos por aumentar las restricciones de la síntesis y aumentar el número de variables mediante su desarrollo en sistemas paralelos, con AG como lo presentan Higuchi y Manderick [10] o programación genética descrita en Cheang, Lee y Leung [3] y Chang, Hou y Su [11] proponiendo siempre arquitecturas de *hardware* y distintas representaciones de redes booleanas.

III. La programación genética y la síntesis booleana

La figura 1 muestra los pasos más importantes en la ejecución de un algoritmo evolutivo. El primer paso consiste en crear una población de forma aleatoria pero con una estructura bien definida, dependiendo de la representación que se emplee para los individuos. A continuación, una función de aptitud que determina qué tan bueno es un individuo, asigna un valor a cada uno de los miembros de la población, que expresa la adaptabilidad en el medio, o lo que es lo mismo, la capacidad de dar solución al problema.

Luego, dentro del ciclo del algoritmo se escogen algunos de los mejores individuos que brindan dicha solución como se muestra en Ashlock [12], llamado proceso de selección. Algunos de estos individuos reemplazarán a otros, y serán sometidos a procesos de alteración o modificación mediante los operadores de cruce y mutación, que serán de tipo aleatorio. El principal objetivo de estos dos operadores es el de generar una nueva población para aumentar la probabilidad de encontrar una solución, bien sea mediante el intercambio de estructuras entre individuos (cruce) o mediante la alteración de una parte de estas (mutación) Sekanina [13].

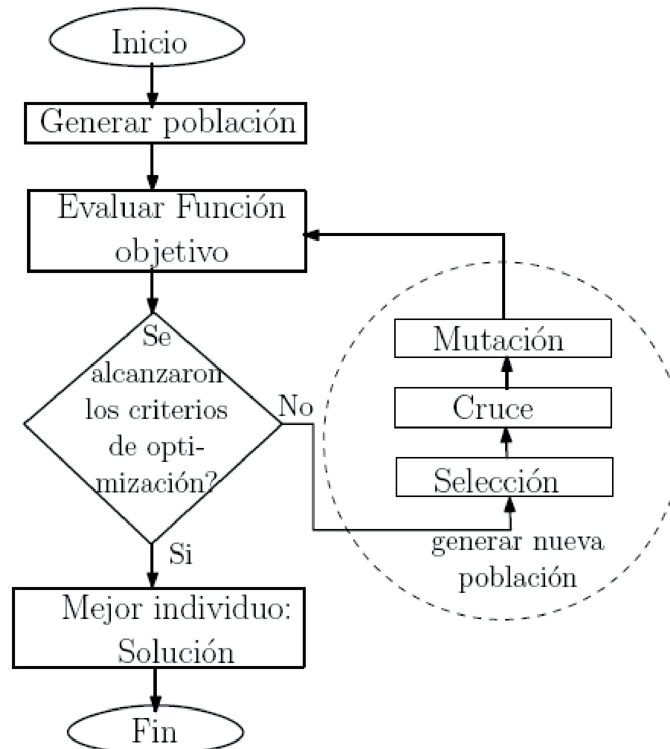

Figura 1. Pasos de un algoritmo evolutivo.

Este ciclo se repetirá tantas veces como sea necesario hasta que se cumplan las condiciones de finalización del algoritmo, basadas en los requerimientos del problema, evitando que puedan existir mejores soluciones, o lo que en términos de optimización se denomina óptimo local según Mitchell [14] y Goldberg y Holland [15].

En el caso del *hardware*, este es representado mediante un cromosoma y bajo el concepto de la selección natural de Darwin [1] es mutado y cruzado con otros cromosomas a fin de crear una población de individuos, para que mediante una función de aptitud se determine si cumple con una función objetivo propuesta, de forma similar a como se describe en la figura 1. Dado que la función objetivo determinará la viabilidad de un circuito o individuo, para el caso específico de la síntesis de *hardware*, deberá contar con una serie de restricciones que por lo general son:

- Que se cumpla con el comportamiento a las entradas y las salidas (tabla de verdad).

- Que se obtenga el menor número de puertas lógicas posible.

Adicionalmente, es posible añadir más restricciones tales como tiempos de propagación, tipo y cantidad de puertas lógicas empleadas, entre otras. Todas estas modificaciones hechas al algoritmo evolutivo para que sea viable para la síntesis de *hardware*, conllevan a emplear una variación del algoritmo genético simple (AGS) conocida como programación genética.

La programación genética como se presenta en Koza et al. [16] y en Nadia, Ajith y Luiza [5] es una técnica basada en los algoritmos genéticos que no distingue el espacio de búsqueda ni tampoco el espacio de representación del cromosoma, basando su estrategia en la variación de la longitud de este último y la creación de nuevos operadores de mutación y de cruce. Los programas que se evolucionan pueden representarse de forma lineal o mediante árboles, siendo esta última la que es de interés en el área de la síntesis booleana.

A. Representación del cromosoma

La codificación adecuada de una estructura de *hardware* es uno de los aspectos más importantes en el diseño de un sistema evolutivo, consiste en la forma como se debe representar un circuito lógico mediante una cadena de bits a fin de poder ser manipulado en el proceso de evolución. Rothlauf [17] Pedraz y Córdoba [18] establecen una serie de condiciones que debe reunir una buena representación genética:

- Se debe poder representar todas las posibles soluciones al problema propuesto.

- Los operadores de cruce y mutación no deben generar individuos irreales.

- Se debe cubrir todo el espacio de soluciones de modo continuo para que la búsqueda sea aleatoria.

Existen distintas formas de representar *hardware* combinacional para un algoritmo genético. Hernández, Coello y Buckles [19] y Koza et al. [16] emplean una representación basada en árboles en 2-D para evolucionar *hardware* haciendo uso de la programación genética. Higuchi [20] propuso una estructura del tipo PLD (Programmable Logic Device) para evolucionar circuitos lógicos, Miller y Harding [21] proponen una representación cartesiana para la evolución de circuitos lógicos mediante un listado de números enteros que serán mapeados directamente a unos grafos. La representación mediante una estructura de árbol 2-D es adecuada para la implementación de sistemas paralelos, dado que permiten fragmentar los cromosomas a fin de repartir la carga computacional como lo demuestra Pedraz y Córdoba [18].

Por simplicidad se estableció una estructura de árbol básica que permita representar una función booleana simple o hasta tres con hasta 4 variables de entrada codificados de forma binaria. La figura 2 muestra la estructura de la celda y la forma como se codifica dentro del cromosoma.

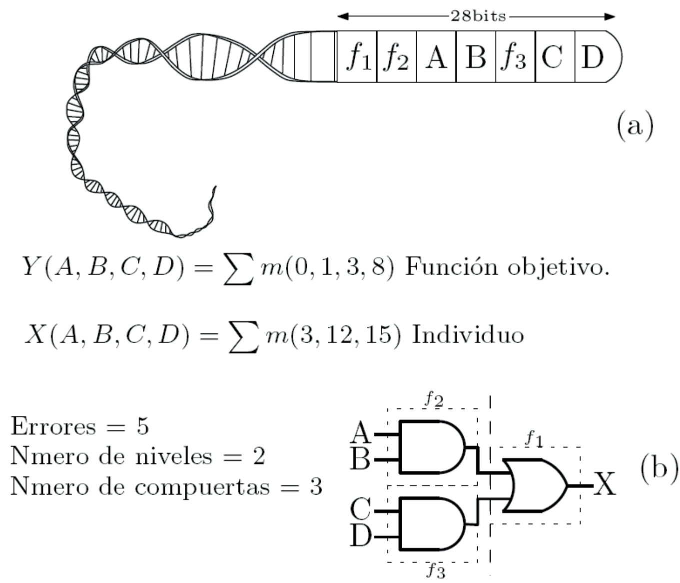

Figura 2. Estructura de una celda básica y su representación dentro del cromosoma.

Es necesario destacar que la longitud del cromosoma es variable dado que se desconoce el tamaño de la solución al problema de la síntesis. Alander [22] mediante estudios empíricos argumenta que un tamaño de población comprendido entre *l* y 2*l* es suficiente para dar solución a la mayoría de los problemas de algoritmos evolutivos, siendo *l* el tamaño del cromosoma. Lo anterior implica que si se tiene una longitud de cromosoma variable, seguramente el tamaño de la población tendrá que serlo también.

B. Función de aptitud

La función de aptitud es otro de los problemas más susceptibles en el diseño de sistemas evolutivos, ya que se encarga de cuantificar la forma en que un cromosoma o individuo se adapta a los requerimientos que se tienen. Se han establecido tres parámetros a tener en cuenta para la función de aptitud:

- El número de coincidencias del individuo X para todas las posibles combinaciones a la salida, con las de la función objetivo.
- El número de puertas lógicas que se tienen dentro del individuo.
- El número de niveles del circuito lógico que determinará el tiempo de propagación máximo del circuito.

$$\text{fitness} = \omega_1 . \left[\sum_{j=1}^{m} \sum_{i=1}^{n} Y(j,i) - X(j,i) \right] + \omega_2 . p(X) + \omega_3 . l(X) \qquad (1)$$

En (1) aparece la función de aptitud (*fitness*) para el sistema evolutivo. Las constantes ω_1, ω_2 y ω_3 determinan los pesos de cada uno de los parámetros que determinan la aptitud de la función. La función p(X) servirá para calcular el número de puertas de un cromosoma teniendo en cuenta algunos de los *introns* o segmentos de la cadena de genotipo que

no tendrán función alguna y que no contribuyen en el resultado del circuito lógico que representa. La función l(X) servirá para determinar el número de niveles que tiene el circuito representado, o en otras palabras, el número de puertas que atraviesa la ruta crítica en el circuito. La constante *m* hace referencia al número de salidas del circuito y *n* al número de combinaciones posibles a las entradas del circuito.

C. Operadores genéticos

Operador de selección: es el encargado de identificar los mejores individuos de toda la población teniendo en cuenta la explotación y la exploración, ampliado en Pedraz y Córdoba [18]. La primera permite que los individuos con un buen valor de aptitud sobrevivan más y se reproduzcan más, la segunda hace referencia a la característica que debe tener el algoritmo de realizar búsquedas en más zonas y dar la posibilidad de encontrar mejores resultados. Se ha empleado el método de selección de Boltzman que permite controlar el equilibrio de exploración y explotación durante la ejecución del algoritmo genético mediante un factor de temperatura (T) de forma similar al temple simulado en Kirkpatrick, Gelatt y Vecchi [23]. La ecuación (2) describe el valor esperado de un individuo en función de T y del valor de aptitud del mismo, para una determinada iteración o instante t.

$$E(i,t) = \frac{e^{f(i)/T}}{\left(e^{f(i)/T}\right)_t} \quad (2)$$

De esta forma, un individuo es seleccionado el número de veces que su valor de aptitud coincida con la parte entera del valor esperado calculado en (2).

Mutación: para diversificar la búsqueda, este operador de vez en cuando realiza alteraciones al cromosoma de forma aleatoria que pueden ser de dos tipos: de operador o variable y de un segmento del cromosoma. Las dos se realizan de forma aleatoria y con una cierta probabilidad que si varía durante la ejecución del algoritmo (mutación evolutiva) descrita en Krohling, Zhou y Tyrrell [24] es más efectiva en el desarrollo de *hardware* evolutivo.

Cruce: este operador es el encargado de recombinar dos individuos seleccionados a fin de generar dos nuevos para la población. Se ha implementado un sistema de cruce de uno o dos puntos de corte, seleccionados de forma aleatoria, dado que es más eficiente para el sistema evolutivo como se describe en Miller y Thomson [25].

D. Programación genética paralela

Una de las ventajas de los algoritmos evolutivos es su capacidad de ser implementados de forma paralela, aumentando el rendimiento mediante el uso de más microprocesadores, memorias y sistemas de comunicación para el intercambio de datos. Un algoritmo evolutivo, específicamente, un programa genético paralelo (PGP) se puede paralelizar de dos formas sin irrumpir drásticamente en su estructura: primero mediante la paralelización de la evaluación de los individuos, dado que esta consume muchos recursos, siendo su cálculo de forma distribuida una buena opción para la paralelización, y en segunda instancia ejecutando varios AE al mismo tiempo en diferentes procesadores, lo que equivale

a evolucionar varias poblaciones de forma paralela. Seguidamente, se intercambian sus mejores individuos con cierta frecuencia para obtener mejores resultados.

Para optimizar el rendimiento del algoritmo paralelo presentado en Pedraz y Córdoba [17] será necesario tener en cuenta los siguientes aspectos:

- Las topologías de comunicación (anillo, todos con todos, maestro-esclavo).
- La proporción de intercambio de datos que puede aumentar la probabilidad de encontrar mejores soluciones. Deberá haber un equilibrio en el número de individuos que se intercambian ya que demasiados no producirán nuevas soluciones y muy pocos no ayudarán a que el algoritmo termine de encontrar una solución.
- La frecuencia de migración, ya que la periodicidad del intercambio de individuo solución de la población entre nodos de un sistema paralelo, puede ser un factor que ayude a encontrar mejores soluciones y más rápidamente.

IV. Implementación del algoritmo evolutivo

A continuación se muestran las formas en que fue implementado el algoritmo evolutivo para la síntesis booleana.

A. Programa genético paralelo y el modelo de islas

Los algoritmos evolutivos son altamente paralelizables ya que la evaluación de la función objetivo de cada individuo de la población en cada iteración es independiente. Empleando el modelo de islas descrito en Di Chio et al. [26] y Luong, Melab y Talbi [27], la población inicial es dividida en μ-poblaciones las cuales están en capacidad de evolucionar de forma independiente, aunque comparten la misma función objetivo. Sin embargo, estas islas no están completamente aisladas, en lugar de esto se encuentran conectadas en una estructura tipo anillo. Luego de varias iteraciones de evolución se realiza un intercambio de individuos entre las μ-poblaciones con sus vecinos, este procedimiento es conocido como migración.

B. Aceleración mediante GPU

En la implementación del AE en tarjetas gráficas mediante procesadores gráficos (GPU) se diferenciaron dos partes fundamentales: la primera corresponde a la generación de números aleatorios y la segunda al programa evolutivo como tal. La primera parte obedece a que la GPU no puede generar números aleatorios haciendo uso de las librerías convencionales de programación en alto nivel como se presenta en Sussman, Crutchfield y Papakipos [28]. Dichos números que serán empleados en la generación de nuevos individuos, cruce y mutación podrían ser transportados desde la memoria de la CPU hacia la GPU, pero este proceso, causaría un incremento significativo en el tiempo de respuesta del AE. Por esta razón, se ha optado por usar una librería de *mersenne-twister* para ser ejecutada en la GPU y establecer un *buffer* de números suficiente en la memoria global durante la ejecución posterior del AE.

La figura 6 muestra la forma como se implementó la segunda parte del sistema en la GPU, que corresponde al algoritmo evolutivo (AE). En el esquema empleado, se hizo uso de la estrategia de evolución por islas, donde cada hilo se encarga de evolucionar una μ-población.

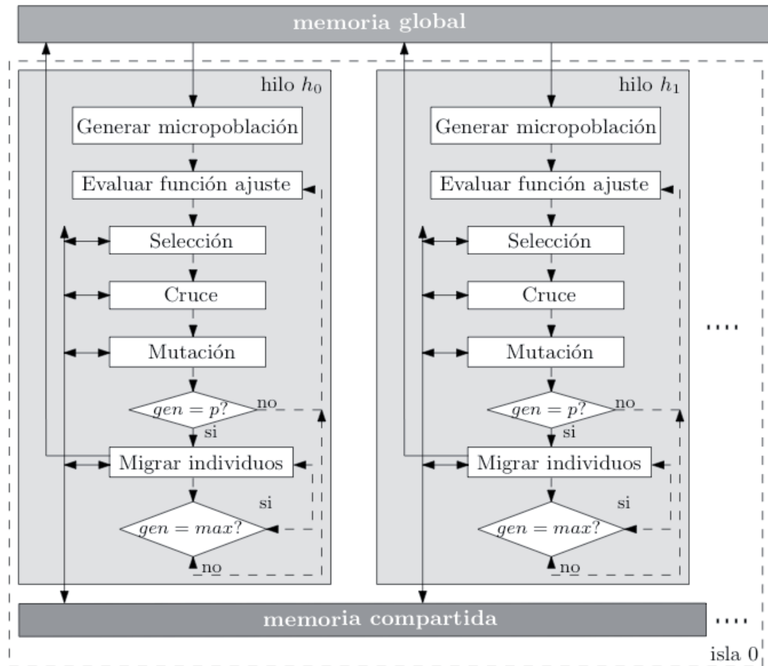

Figura 3. Implementación del algoritmo evolutivo en la GPU.

C. Aceleración mediante CPU con múltiples procesadores

Como se mencionó anteriormente una técnica para mejorar el desempeño de un Algoritmo Genético (AG) es implementarlo de forma paralela. En la actualidad las CPU cuentan con múltiples procesadores lo que permite que la CPU ejecute procesos independientes de forma paralela. En este sentido, para paralelizar el algoritmo genético siguiendo el mismo modelo de islas, se lanzaron procesos independientes los cuales tiene copias idénticas del algoritmo evolutivo pero con μ-poblaciones diferentes. En esta plataforma el operador de migración emplea tuberías para permitir el intercambio de individuos entre las μ-poblaciones.

A pesar del aumento del rendimiento que se logra al lanzar procesos independientes que se ejecuten de forma paralela en los distintos núcleos, esto se ve limitado como lo expresa la ley de Amdahl. De acuerdo con esta ley cuando un elemento de un sistema es mejorado, el desempeño total mejora en una fracción del tiempo empleado por el componente.

Para mitigar el efecto de esta ley los procesadores Intel Core i7 cuentan con una característica llamada *turbo boost* la cual permite incrementar la frecuencia de los procesadores por un corto lapso de tiempo dependiendo de la cantidad de núcleos activos, el consumo de corriente estimado, el consumo de potencia estimado, y la temperatura del procesador. Con los experimentos descritos se obtuvo una mejora en el rendimiento entre

3% y 12%, sin embargo, las medidas en su mayoría tuvieron una mejora entre el 4% y el 6%. Estos resultados son similares a los que describen Charles et al. [29].

V. Experimentos

A. Configuración del experimento

El programa genético paralelo fue probado en dos arquitecturas. La primera es una plataforma multi-GPU basada en dos dispositivos *NVIDIA Tesla*, cada una de las cuales cuenta con 448 núcleos CUDA. La segunda plataforma es una CPU con un procesador *Intel Core i7* y 4GB de memoria RAM.

Sobre estas dos plataformas se ejecutó el programa genético paralelo para 4, 8 y 12 variables, diferentes tamaños de poblaciones y diferente número de hilos o procesos e islas ejecutados.

Las figuras 4, 5 y 6 muestran el tiempo de respuesta del algoritmo ejecutado en la plataforma multi-GPU, con 1 y 2 GPU para poblaciones de 1024 o 32786 individuos y diferente cantidad de islas.

B. Rendimiento en la GPU

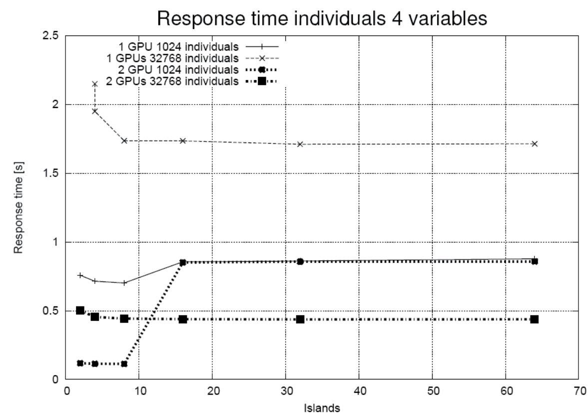

Figura 4. Tiempo de respuesta con 1 y 2 GPU para 4 variables y diferentes tamaños de población

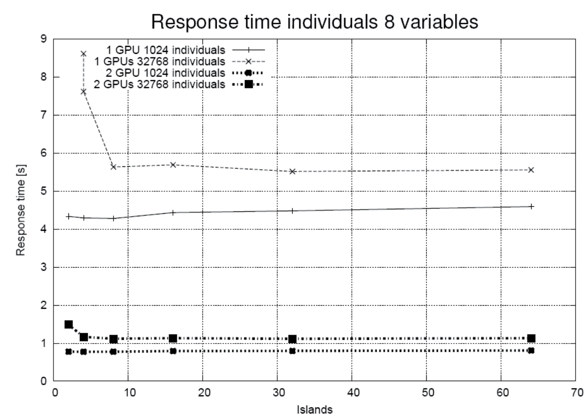

Figura 5. Tiempo de respuesta con 1 y 2 GPU para 8 variables y diferentes tamaños de población

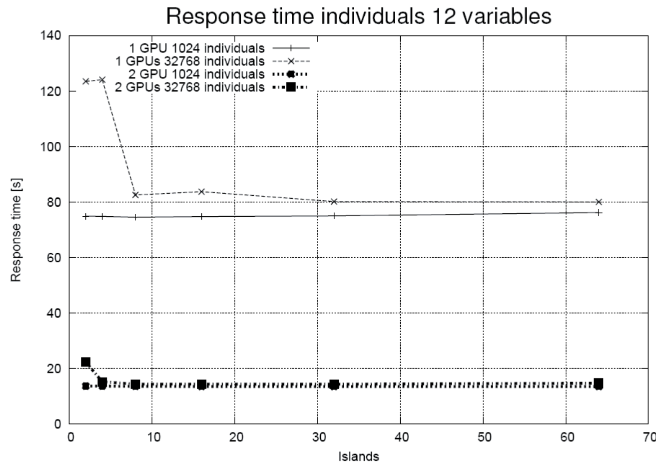

Figura 6. Tiempo de respuesta con 1 y 2 GPU para 12 variables y diferentes tamaños de población

Este experimento demuestra que el tiempo de respuesta depende del tamaño del problema debido a que complejidad de los individuos crece exponencialmente con la cantidad de variables del problema y estos son evaluados por *software*. En contraste el tiempo de respuesta mostrado por Pedraza et al. en [30] no tiene una gran dependencia del tamaño de problema ya que se evalúa en *hardware*.

También se observa que para evaluar el PGP con un tamaño de población grande (32.768 individuos) tiene mejor desempeño en 2 GPU que en una sola. Sin embargo, al aumentar la población de 1024 a 32.768 el tiempo de respuesta no se incrementa en la misma proporción ya que el tiempo empleado para las comunicaciones está incluido en los dos escenarios.

Por otra parte, las figuras 7, 8 y 9 muestran los tiempos de respuesta del PGP evaluado en la plataforma CPU. En esta plataforma se lanzaron de 1 a 8 procesos paralelos en cada una de las unidades de procesamiento del procesador *Intel Core i7*.

Rendimiento en la GPU.

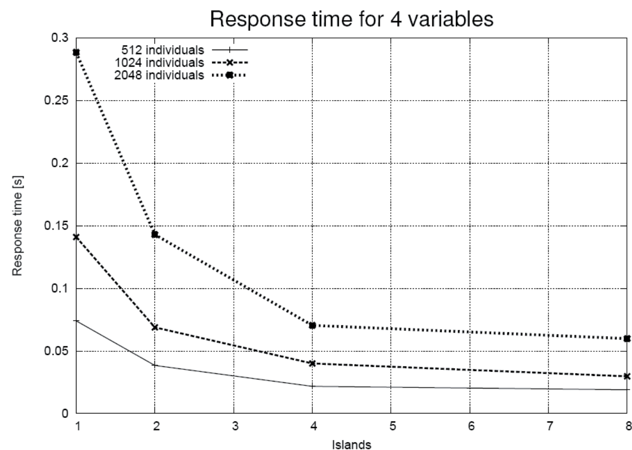

Figura 7. Tiempo de respuesta con CPU para 4 variables y diferentes tamaños de población

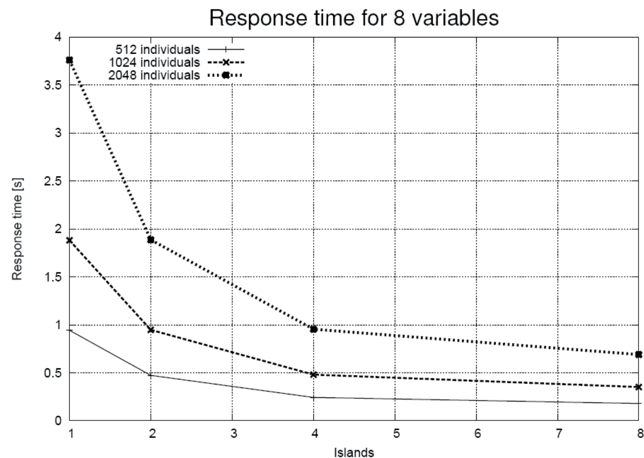

Figura 8. Tiempo de respuesta con CPU para 8 variables y diferentes tamaños de población

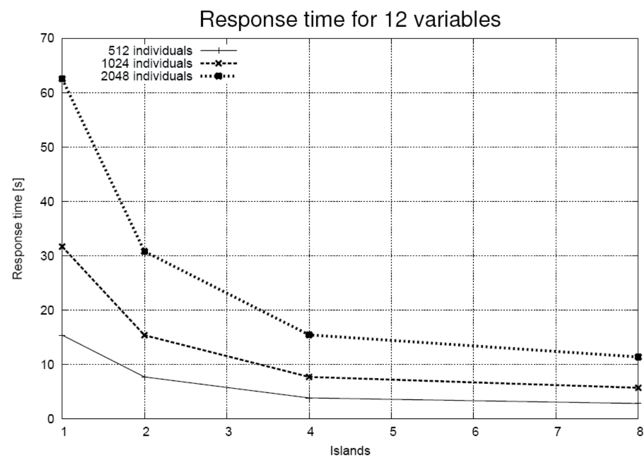

Figura 9. Tiempo de respuesta con CPUs para 12 variables y diferentes tamaños de población

Al comparar el tiempo de respuesta de todos los escenarios probados se encuentra que la mayor aceleración se logra comparando el PGP evaluado con una población de 32K sobre dos GPU contra el tiempo empleado por la CPU con un solo proceso ejecutando el algoritmo con la misma población.

VI. Conclusiones

Se presentó una estrategia alternativa y novedosa para evaluar individuos de un algoritmo evolutivo sobre plataformas basadas en CPU y GPU. Los resultados de los experimentos mostraron una aceleración superior a 41 evolucionando más de 32K individuos en una plataforma multi-GPU comparado con la plataforma CPU. Además se demostró que para poblaciones pequeñas (1024 individuos) es menos eficiente ejecutarlo en las dos GPU en comparación con ejecutarlo en la plataforma CPU ya que el tiempo empleado en las comunicaciones se vuelve importante. Por otra parte, comparando el desempeño de una GPU contra dos GPU, la primera plataforma evoluciona más rápidamente las poblaciones pequeñas mientras que para poblaciones grandes resulta más rápido ejecutar el algoritmo sobre dos GPU.

Los experimentos demostraron que el algoritmo es más efectivo para los problemas de 4 y 8 bits. Los problemas de 12 bits tienen un excelente desempeño al ser evaluados en GPU; sin embargo, debido a que el espacio de búsqueda es muy grande es difícil que el algoritmo converja a una solución adecuada. Este problema puede ser solucionado a futuro empleando una plataforma GPU con una mayor cantidad de núcleos, y por tanto, mayor capacidad.

Mediante la medición de tiempos en escenarios con distinto número de hilos e islas, se determinó que es conveniente lanzar un número de hilos no superior al número de núcleos de procesamiento de la GPU, con un número de islas que no exceda los recursos de memoria compartida. La principal razón es porque la mayor parte de la memoria de este tipo disponible, es usada para las operaciones a nivel de subpoblación. De esta manera, si se desea aumentar el tamaño de la población será conveniente aumentar también el número de islas y de hilos, a fin de conservar el número de individuos en la micropoblación. Esto representa un problema de escalabilidad desde el punto de vista de la tarjeta gráfica, ya que se necesitarán más llamadas a la función *kernel* del programa genético, que define el programa de cada hilo.

Finalmente, también se comprobó el funcionamiento de la herramienta tecnológica *Turbo Boost* de los procesadores *Intel Core i7*.

Referencias

[1] Ch. Darwin, *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life*. New York: D. Appleton, 1859.

[2] E. Cantú-Paz, *Efficient and accurate parallel genetic algorithms*. Estados Unidos, 2001.

[3] S. M. Cheang, K. H. Lee, y K. S. Leung, «*Applying Genetic Parallel Programming to Synthesize Combinational Logic Circuits*». Evolutionary Computation, IEEE Transactions on, Volumen 11. pp. 503 – 520, 2007.

[4] L. Jozwiak, N. Ederveen, y A. Postula, «*Solving synthesis problems with genetic algorithms*». Euromicro Conference, Volumen 1, pp.10001, 1998.

[5] N. Nadia, A. Ajith, y L. M. de Macedo, *Genetic Systems Programming*, ed. Springer, 2006.

[6] A. Stoica, «*Evolvable hardware: from on-chip circuit synthesis to evolvable space systems*». Multiple-Valued Logic, (ISMVL 2000) Proceedings. 30th IEEE International Symposium on, 2000: p. 161 - 169, 2000

[7] A. Stoica, *Evolvable Hardware for Autonomous Systems*. CEC Tutorial, 2004.

[8] A. Thompson, «*An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics*». Lecture notes in computer science, pp. 390-405, 1997.

[9] T. Kalganova, «*An Extrinsic Function-Level Evolvable Hardware Approach*», Lecture notes in computer science, Berlín:Springer, pp.60-75, 2000.

[10] T. Higuchi, T. Niwa, T. Tanaka, H. H. Iba, H. de Garis, y F. Tatsumi, «*Evolving hardware with genetic learning: a first step towards building a Darwin machine*» Proceedings of the second international conference, pp. 417-424, 1993.

[11] S-J. Chang, H-S. Hou, y Y-K. Su, «Automated synthesis of passive filter circuits including parasitic effects by genetic programming». *Microelectronics Journal*, volumen 37, pp. 792 – 799, 2006.

[12] D. Ashlock, *Evolutionary Computation for Modeling and Optimization*. Guelp, Ontario: Springer, 2005, 571p.

[13] L. Sekanina, *Evolvable components, from theory to hardware implementations*, Natural computing series, República Checa: Springer, 1998.

[14] M. Mitchell, *An introduction to genetic algorithms*. Massachusetts: The MIT Press, 1998.

[15] D. Goldberg, y J. Holland, *Genetic Algorithms and Machine Learning*. Machine Learning, Paises bajos: Springer, 1988.

[16] J. Koza, F. Bennett, D. Andre, y M. Keane, *Genetic programming III: darwinian invention and problem solving* [Book Review]. Evolutionary Computation, 1999.

[17] F. Rothlauf, *Representations for Genetic and Evolutionary Algorithms*, ed. Springer, 2006.

[18] J. C. Pedraz, y C. Córdoba, «*Implementación de un algoritmo genético paralelo sobre hardware gráfico de última generación*». recolecta.net. 2005.

[19] A. Hernández, C. Coello, y B. Buckles, «*A genetic programming approach to logic function synthesis by means of multiplexers*». Proceedings of the First NASA/DoD Workshop on Evolvable, pp. 46-53, 1999.

[20] T. Higuchi, y B. Manderick, «*Hardware realizations of evolutionary algorithms*». Evolutionary Computation 2. (Noviembre), pp. 253 -263, 2000.

[21] J. F. Miller, y S. Harding, «*Cartesian genetic programming*». Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference, pp. 3489-3512, 2009.

[22] J. Alander, «*On optimal population size of genetic algorithms*». CompEuro'92. Computer Systems and Software Engineering' proccedings, pp 65-70, 1992.

[23] S. Kirkpatrick, C. Gelatt,. y M. Vecchi, «*Optimization by Simulated Annealing*». Science, Volumen 220, Número 4598, pp. 671-680, 1983.

[24] R. Krohling, Y. Zhou, y A. Tyrrell, «*Evolving FPGA-based robot controllers using an evolutionary algorithm*» 1st International Conference on Artificial Immune Systems, 2002.

[25] J. Miller, y P. Thomson, «*Aspects of Digital Evolution: Geometry and Learning*». Lecture notes in computer science, 1998.

[26] C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. Esparcia-Alcazar, et al. *Applications of Evolutionary Computation*. Berlin, Heidelberg: Springer, Volumen 6024, 2010, pp. 442-451.

[27] T. V. Luong, N. Melab, y E. G. Talbi, «*GPU-based island model for evolutionary algorithms*». Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10. New York, USA: ACM Press, p. 1089, 2010.

[28] M. Sussman, W. Crutchfield, y M. Papakipos, «*Pseudorandom number generation on the GPU*». Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics *hardware*, pp. 87-94, 2006.

[29] J. Charles, P. Jassi, N.S. Ananth, A. Sadat, y A. Fedorova, Evaluation of the Intel Core i7 Turbo Boost feature. In: 2009 IEEE International Simposium on Workload Characterization (IISWC), pp. 188-197. IEEE (2009). DOI 10.1109

[30] C. Pedraza, E. Castillo, J. Castillo, C. Camarero, J. Bosque, J. Martinez, y R. Menendez, «*Cluster architecture based on low cost reconfigurable hardware*». International conference on field programmable logic and applications, FPL. Heidelberg, Alemania, pp. 595 – 598, 2008.