Implementación sobre FPGA de la estrategia evolutiva CMA-ES para optimización numérica

Leopoldo Urbina, Carlos A. Duchanoy, Marco A. Moreno-Armendáriz, Derlis Lara, Hiram Calvo

Instituto Politécnico Nacional, Centro de Investigación en Computación, México D.F., México

Resumen. En este trabajo se presenta la implementación sobre hardware del algoritmo *Covariance Matrix Adaptation Evolution Strategy* (CMA-ES). Este algoritmo se basa en la adaptación de la matriz de covarianza que se focaliza inicialmente en una región del espacio de búsqueda particular y posteriormente se mueve o crece a lo largo del espacio de búsqueda, según sea conveniente para encontrar el valor óptimo. Los resultados experimentales muestran que dicha implementación será de gran utilidad para resolver problemas de optimización numérica en un sistema embebido.

Palabras clave: Estrategia evolutiva CMA-ES, implementación sobre FPGA, optimización numérica.

FPGA Implementation of the Evolutive Strategy CMA-ES for Numerical Optimization

Abstract. In this paper, a hardware implementation of the *Covariance Matrix Adaptation Evolution Strategy* (CMA-ES) algorithm is presented. This algorithm is based on the adaptation of the covariance matrix, initially it is focused on a region of the particular search space and subsequently, it moves or grows along the search space, as appropriate to find the optimum value. The experimental results reveal that this implementation will be very useful to resolve numeric optimization issues in an embedded system.

Keywords: Evolutive strategy CMA-ES, numerical optimization, FPGA implementation.

1. Introducción

Uno de los objetivos más comunes de implementar algoritmos sobre alguna arquitectura o hardware especifico es el de obtener mayor velocidad de procesamiento de datos y menor utilización de recursos en un sistema, por lo general los algoritmos que se implementan de esta manera son los bio-inspirados que se caracterizan por consumir muchos recursos y tiempo de ejecución.

El propósito de este trabajo no es reducir el tiempo de procesamiento, el objetivo es implementar el algoritmo evolutivo CMA-ES sobre *Field-Programmable Gate Array* (FPGA), el cual pueda servir posteriormente para implementar todo un sistema de optimización numérica sobre un hardware específico, sin necesidad de una computadora de escritorio.

Se llama hardware evolutivo [7] a la integración del cómputo evolutivo y un dispositivo de hardware programable, cuyo objetivo es la reconfiguración "autónoma" de la estructura del hardware para mejorar el desempeño. En [16] se propuso la aceleración de un algoritmo genético (AG) implementándolo sobre un FPGA. Las características del AG implementado son: selección aleatoria de los padres, la cual mantiene el sistema de circuitos de selección; un modelo de memoria de estado estacionario, el cual ocupado un espacio constante al ser embebido en el chip; supervivencia de los cromosomas hijos más aptos sobre los cromosomas padres menos aptos, lo que promueve la evolución. El AG implementado sobre el FPGA está organizado en un proceso segmentado (pipeline) de seis estados, donde a cada estado se le asigna el mismo tiempo de procesamiento igual a un ciclo de reloj. Para sus experimentos, consideraron dos problemas, el problema del cubrimiento de conjuntos y el problema de plegamiento de proteínas.

En [15], se presenta el diseño de un algoritmo genético en *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) llamado *Hardware-based Genetic Algorithm* (HGA) con el objetivo de implementarlo en hardware. Debido a los procesos de segmentación, paralelización y al no sobrecargo de funciones, un AG en hardware alcanza una velocidad significativa sobre un AG en software, lo que es especialmente útil cuando el AG es usado para aplicaciones de tiempo real, por ejemplo, planificación del disco y registro de imágenes.

En [10], se realiza el diseño y la implementación de un Algoritmo Genético Compacto sobre FPGA, los resultados logrados demuestran que tener un algoritmo de búsqueda como un Algoritmo Genético Compacto en hardware, es muy útil, ya que permite realizar optimización numérica global en tiempo real, y además permite incorporarlo en un chip que forme parte de una aplicación de mayor escala. Un ejemplo de aplicación se desarrolla en [8], donde un control automático inteligente para el aterrizaje de una aeronave es diseñado, basado en un algoritmo hibrido entre redes neuronales recurrentes y algoritmos genéticos. Otros trabajos relacionados se presentan en [1,9,4,11,3].

En este trabajo se logró la implementación del algoritmo CMA-ES en el Nios II. Este trabajo se desarrollará en las siguientes secciones: en la sección 2, se describirá a detalle la estructura del algoritmo CMA-ES; en la sección 3 se explicará el desafío que fue lograr la implementación del algoritmo; los resultados logrados por el trabajo se reportarán en la sección 4; la interpretación de resultados y las concluciones obtenidas, se justificarán en las secciones 5 y 6 respectivamente; por último, las ideas propuestas para continuar con el desarrollo de este trabajo, están redactadas en la sección 7.

2. Algoritmo CMA-ES

Las estrategias evolutivas fueron inventadas por Ingo Rechenberg [12] y Hans-Paul Schwefel [14] en los años 60's. El operador principal para mejorar el desempeño del individuo es la mutación. Este operador se basa en la distribución normal con media $\mathbf{m}$ y desviación estándar $\boldsymbol{\sigma}$. $\mathbf{m}$ y $\boldsymbol{\sigma}$ son parámetros de estrategia del algoritmo y se incluyen en el genoma del individuo. Por esta razón, la característica principal de las estrategias evolutivas es la auto-adaptación. El CMA-ES [5] es usado para problemas de minimización y se describe a continuación:

Paso 1. Ajuste de parámetros de la estrategia. Establecer el número de puntos de búsqueda n, el número de hijos λ, el número de padres μ, los pesos $w_i, i = 1, \ldots, \mu$ y los parámetros $c_\sigma, d_\sigma, c_c, c_1$, y c_μ que es usada para cálculos complementarios.

Paso 2. Inicialización. Inicializar el número de generación g = 0, la matriz de covariancia $\mathbf{C}^{(0)} = \mathbf{I}$, el tamaño del paso $\sigma^{(0)} = 0.5$, el camino de evolución $\mathbf{p}_\sigma = 0, \mathbf{p}_c = 0$, y el valor inicial de la media $\mathbf{m}^{(0)} \in \mathbb{R}^n$.

Paso 3. Inicio. Generar una población de puntos de búsqueda usando una distribución normal, de la siguiente manera:

$$\mathbf{x}^{(\mathbf{g+1})}{}_k \sim \mathcal{N}(\mathbf{m}^{(\mathbf{g})}, (\sigma^{(\mathbf{g})})^2 \mathbf{C}^{(\mathbf{g})}) \text{ for } k{=}1,\ldots,\lambda. \tag{1}$$

Paso 4. Selección y recombinación. Obtener el valor de aptitud $f(\mathbf{x}_\mathbf{k}^{(\mathbf{g+1})})$ de los puntos de búsqueda y seleccionar los μ mejores puntos $\mathbf{x}_{i:\lambda}^{(g+1)}$ para $i = 1, \ldots, \mu$. Actualizar el valor de la media de la distribución usada para la búsqueda como sigue:

$$m^{(g+1)} = \sum_{i=1}^{\mu} w_i \mathbf{x}_{i:\lambda}^{(g+1)}. \tag{2}$$

Paso 5. Adaptación de la matriz de covariancia. La matriz de covarianza se actualiza como sigue.

$$\begin{aligned} \mathbf{C}^{(g+1)} = (1 - c_1 - c_\mu)\mathbf{C}^{(g)} + c_1 \left(\mathbf{p}_c \mathbf{p}_c^T + \delta(h_\sigma)\mathbf{C}^{(g)} \right) \\ + c_\mu \sum_{i=1}^{\mu} w_i \left(\frac{\mathbf{x}_{i:\lambda} - \mathbf{m}}{\sigma} \right) \left(\frac{\mathbf{x}_{i:\lambda} - \mathbf{m}}{\sigma} \right)^T \end{aligned} \tag{3}$$

Donde,

$$\mathbf{p}_c^{(g+1)} = (1-c_c)\mathbf{p}_c^{(g)} + h_\sigma \sqrt{c_\sigma(2-c_\sigma)\mu_{eff}}\frac{\mathbf{m}^{(g+1)}-\mathbf{m}^{(g)}}{\sigma}, \tag{4}$$

$$c_c = \frac{4+\mu_{eff}/n}{n+4+2\mu_{eff}/n}, \tag{5}$$

$$c_1 = \frac{2}{(n+1.3)^2+\mu_{eff}}, \tag{6}$$

$$c_\mu = min\left(1-c1, \alpha_\mu \frac{\mu_{eff}-2+1/\mu_{eff}}{(n+2)^2+\alpha_\mu\mu_{eff}/2}\right), \tag{7}$$

$$\alpha_\mu = 2, \tag{8}$$

$$\delta(h_\sigma) = (1-h_\sigma)c_c(2-c_c), \tag{9}$$

$$h_\sigma = \begin{cases} 1 & \text{if } \frac{\|\mathbf{p}_\sigma\|}{\sqrt{1-(1-c_\sigma)^{2(g+1)}}} \\ & < (1.4+\frac{2}{n+1})E\|\mathcal{N}(\mathbf{0},\mathbf{I})\|, \quad , \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

$$E\|\mathcal{N}(\mathbf{0},\mathbf{I})\| = \sqrt{n}\left(1-\frac{1}{4n}+\frac{1}{21n^2}\right). \tag{11}$$

Paso 6. Control del tamaño de paso. El tamaño del paso se actualiza de la siguiente manera:

$$\sigma^{(g+1)} = \sigma^{(g)} exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|\mathbf{p}_\sigma\|}{E\|\mathcal{N}(\mathbf{0},\mathbf{I})\|}-1\right)\right), \tag{12}$$

Donde,

$$\begin{aligned}\mathbf{p}_\sigma^{(g+1)} =& (1-c_\sigma)\mathbf{p}_\sigma^{(g)} + \sqrt{c_\sigma(2-c_\sigma)\mu_{eff}} \\ & \cdot \left(\mathbf{C}^{(g)}\right)^{-\frac{1}{2}} \frac{\mathbf{m}^{(g+1)}-\mathbf{m}^{(g)}}{\sigma^{(g)}}.\end{aligned} \tag{13}$$

Paso 7. Criterio de finalización. Si se cumple con alguna de las siguientes condiciones como lo es el máximo número de generaciones, el máximo número de evaluaciones de aptitud o se llega al valor de umbral de la aptitud; entonces se termina el proceso de búsqueda. Si no se cumple ninguna de las condiciones anteriores, entonces g = g +1 y se regresa al **Paso 3**.

En cada iteración del algoritmo, se actualizará la matriz de covarianza **C** y por lo tanto, también se actualizarán sus vectores y valores propios, esto significa que la magnitud del espacio de búsqueda de la solución deseada ira incrementando y adaptando en cada iteración, siendo así el espacio de búsqueda una elipse cada vez más enfocada (adaptada) a encontrar lo que se está buscando, como se puede apreciar en la Figura 1.

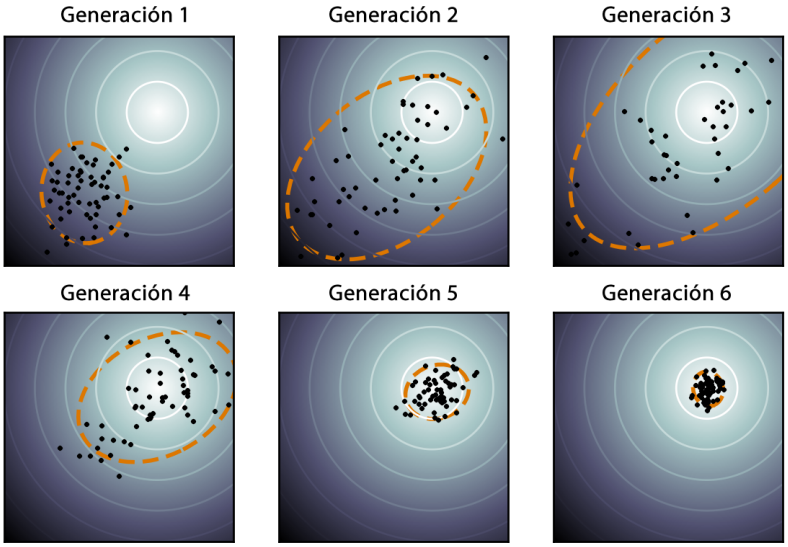

Fig. 1: Ejemplo de búsqueda adaptativa mediante la actualización de la matriz de covarianza C

3. Implementación sobre FPGA

Para llevar a cabo esta implementación fue necesario usar un procesador NIOS II [2]. El primer paso es realizar su inicialización con las siguientes características: Bloques de memoria de 4kbit de 32 bits, Dos puertos (uno de salida y otro de entrada) de 8 bits y el JTAG UART para tener comunicación con una computadora de escritorio. El segundo paso fue codificar el algoritmo de la sección 2, un problema que se encontró fue como realizar el cálculo de los eigenvectores y eigenvalores de matriz de covarianza **C**, después de revisar la literatura, se decidió usar el algoritmo QR, el cual se basa en la descomposición QR, desarrollada en la década de 1950 por John G.F. Francis (Reino Unido) y Vera N. Kublanovskaya (URSS).

Una vez implementado el Nios II y cargado el algoritmo CMA-ES sobre este, se verificó su correcto funcionamiento, un detalle a resolver fue la generación de la población inicial, esta se debe generar aleatoriamente, para hacerlo en lenguaje C se usa la función *rand*, esta función está programada con base a un algoritmo que toma como referencia un valor semilla y a partir de este se generan números que en realidad son pseudoaleatorios, por *default* la semilla es estática, es decir aunque se ejecute de nuevo el programa se obtendrán los mismo valores, para corregir esta situación se utiliza el reloj del sistema con la función *srand (time(0))*, con lo cual a cada ejecución la semilla tendrá un valor diferente y así los números pseudoaleatorios generados serán diferentes. Para más detalles se puede consultar [6].

4. Experimentos y resultados

Para realizar la evaluación de la implementación realizada, se usaron 7 funciones de aptitud, estas funciones representan problemas de optimización mono objetivo [13]. Además, se programaron versiones del CMA-ES en Matlab® y lenguaje C, con el fin de validar el funcionamiento del algortimo.

Tabla 1: Valores de inicialización del CMA-ES.

Valor	**Uso**
$n = 4$	Dimensión del problema y tamaño de la población
stopfitness $= 1 * E^{-11}$	Valor mínimo a alcanzar en el valor de aptitud
stopeval $= N^2 * 10^3 = 16000$	Máximo número de evaluaciones
$\lambda = 4 + floor(3 * log(n)) = 8$	Número de hijos
$\sigma = 0.5$	Tamaño del paso
$\mu = \frac{\lambda}{2}$	Número de puntos para la recombinación
$w_i = log(\mu + 0.5) - log(1 : \mu)'$	Pesos para la recombinación

Nikolaus Hansen, el creador del algoritmo CMA-ES, explica detalladamente en [5] el funcionamiento del algoritmo, asi como consejos para su correcto uso e implementación. En la Tabla 1 se muestran los valores iniciales requeridos para la ejecucción del algoritmo CMA-ES.

Tabla 2: Valores finales de las funciones de aptitud.

Función	**FPGA**	**Matlab®**	**Lenguaje C**
Esfera	2.557E-07	7.221E-07	2.523E-09
Rosenbrock	9.998E-01	1.000E+00	1.000E+00
Elli	-5.166E-08	-3.456E-08	-2.324E-09
Diffpow	6.739E-03	2.155E-03	-5.409E-03
Cigar	1.183E-07	-1.377E-07	-8.210E-09
Tablet	6.536E-08	3.634E-07	1.954E-09
Cigtab	-8.436E-09	8.827E-08	1.457E-03

Las pruebas del algoritmo CMA-ES sobre el Nios II se lograron pese a algunos inconvenientes, en el caso particular de las funciones *Cigtab*, *Diffpow* y *Tablet*,

estas no convergían en la mayor parte de las ejecuciones, esto fue debido a que la dispersión de los valores aleatorios generados como solución influyen en la rapidez de convergencia del algorítmo, es decir, si los valores iniciales generados son muy dispersos entre si, es muy probable que al ser procesados por el CMA-ES no se llegue a una solución aceptada, ya que cabe la posibilidad de que los valores en la matriz de covarianza **C** sean ínfimos o excesivos, esto repercutiría en el aumento de recursos computacionales para obtener los eigenvalores y eigenvectores, a tal grado que se puede estar iterando por muchos ciclos sin conseguir un resultado aceptable. Para solucionar este problema, se puede ejecutar el algoritmo nuevamente esperando que la propuesta inicial de solución no sea tan dispersa y se pueda procesar por el CMA-ES de manera adecuada, o en su defecto de ser posible asignar una mayor cantidad de recursos en el diseño del Nios II. En nuestro caso se optó por omitir las ejecuciones que no tuvieron un resultado concluyente, es decir, solo se tomaron en cuenta las ejecuciones que hayan concluido de manera satisfactoria (llegar al valor de umbral de la aptitud).

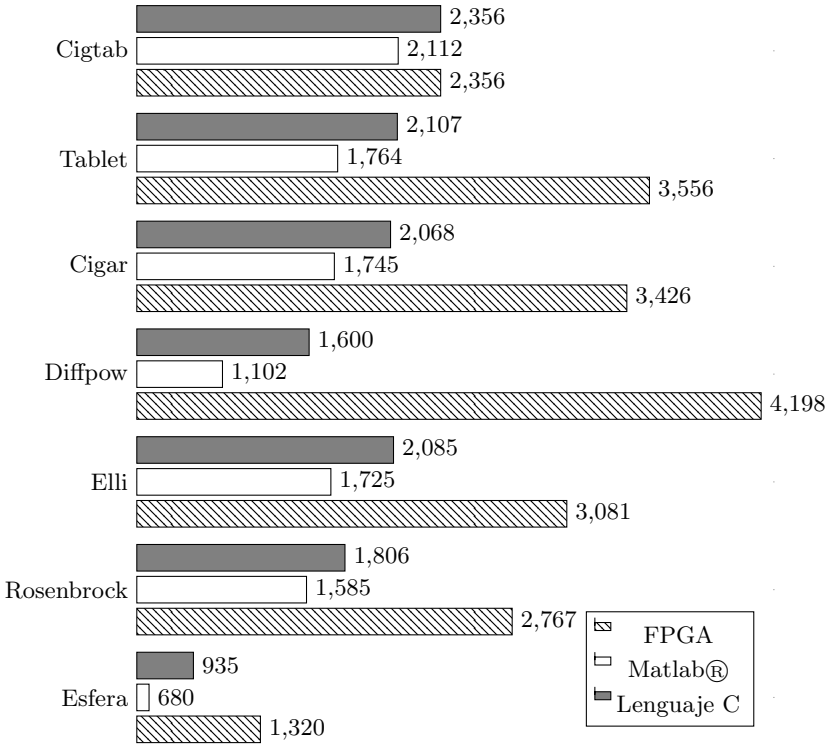

Fig. 2: Número de evaluaciones realizadas por la función de aptitud.

Se reportan tres resultados importantes del algoritmo CMA-ES, como son: número de iteraciones realizadas (Ver Figura 2), último valor obtenido para la

función de aptitud (Ver Tabla 2), valor óptimo obtenído por el algoritmo (Ver Tabla 3). Por cada función de aptitud, el algoritmo se ejecutó 45 veces, solamente se reportó el promedio de cada función.

La Tabla 2 reporta el valor numérico de la función de aptitud evaluado por el valor óptimo, el valor de la función de aptitud debe ser cero o cercano a este, con excepción de la función *Rosenbrock* cuyo valor de aptitud debe tender a uno. La Tabla 3 contiene información referente al valor óptimo para cada función de aptitud, los valores alcanzados deben de ser lo más próximo a cero.

Tabla 3: Valor óptimo obtenído por el algoritmo.

Función	**FPGA**	**Matlab®**	**Lenguaje C**
Esfera	5.231E-12	5.669E-12	5.287E-15
Rosenbrock	6.191E-12	6.806E-12	1.821E-14
Elli	5.508E-12	4.611E-12	1.065E-14
Diffpow	6.286E-12	6.777E-12	6.072E-13
Cigar	5.270E-12	7.157E-12	1.377E-14
Tablet	6.810E-12	7.546E-12	1.057E-14
Cigtab	6.031E-12	5.549E-12	1.068E-13

4.1. Discusión

Los resultados obtenidos en los experimentos presentados demuestran que la estrategia evolutiva CMA-ES se implementó de manera exitosa sobre el FPGA, sin embargo, presenta ciertas desventajas respecto a su ejecución sobre una computadora personal, como ejemplo, la velocidad y capacidad de procesamiento en las computadoras personales es mayor en comparación al Nios II del FPGA.

Como ya se mencionó la velocidad del procesamiento del CMA-ES depende del hardware en el cual se está ejecutando, lo cual indica que aún y cuando se ejecute en una computadora, la rapidez con que llegue a una solución dependerá de las características de dicha computadora por ejemplo, el tipo de procesador y memoria RAM con los que cuente, otro aspecto importante de mencionar que también afecta la rapidez de convergencia, es la forma en que se haya programado el CMA-ES, por ejemplo para los experimentos realizados convergió más rápido la versión del CMA-ES programada en Matlab® que la versión en lenguaje C ejecutándose sobre una computadora, esto debido a que Matlab® cuenta con un lenguaje de programación propio que permite operaciones de vectores y matrices, mientras que en lenguaje C, se programaron versiones de dichas funciones que tardan un poco más en dar resultados, ya sea por el algoritmo programado o por la forma de crear arreglos o matrices en lenguaje C, que es mediante ciclos "*for*", esto depende de la experiencia y habilidad que se tenga en lenguaje C.

Analizando el gráfico presentado en la Figura 2, podemos concluir que el lenguaje de Matlab® presentó el mejor desempeño, a diferencia de la implementación en FPGA, esto se debe al modo en que se genera la población inicial en el FPGA. Los resultados reflejados en la Tabla 2, demuestran que el Lenguaje C presenta una mayor precisión, seguido por el FPGA, dejando al lenguaje de Matlab® en tercera posición, esto demuestra que la implementación realizada en el FPGA compite con los lenguajes de alto desempeño. En la Tabla 3 se reporta el valor óptimo obtenido para cada función, este valor está totalmente ligado a los resultados de la Tabla 2, por lo tanto se obtuvieron los mismos resultados el Lenguaje C obtuvo la mejor precisión, caso contrario del lenguaje en Matlab®.

El resultado de embeber el algoritmo CMA-ES en un FPGA es exitoso y comparado con los lenguajes C y Matlab®, es competitivo, ya que como en un principio se aclaró, el objetivo de este trabajo no es el de competir en tiempo de procesamiento, si no, embeber de manera funcional el algoritmo genético.

5. Conclusiones

Se ha logrado implementar un algoritmo evolutivo como el CMA-ES sobre FPGA, capaz de obtener los mejores individuos de una generación, los cuales servirán para diversas aplicaciones, basta modificar la función objetivo a evaluar por el CMA-ES, por la propia del sistema sobre el cual se quiera aplicar el modelo realizado.

La rapidez de convergencia de este tipo de heurísticas depende de los siguientes factores: el hardware en el cual son procesadas; la herramienta con la cual son programadas; la forma en la que son programadas y los valores aleatorios que adquiere la población inicial. Se concluyó que Matlab® es una herramienta eficiente que facilita la programación de este tipo de heurísticas, sin embargo para implementar algoritmos sobre el Nios II es necesario programarlos en C/C++ o ensamblador, la eficiencia de la implementación dependerá de la experiencia, habilidad y destreza que se tenga en estos lenguajes de programación, para programar funciones de manera eficiente. La ventaja de realizar implementaciones de este tipo es la de tomar señales directas del sistema real, lo que da un mayor de certidumbre. Al contrario de sólo realizarlo mediante un modelo matemático. Utilizar heurísticas para la solución a diversos problemas de optimización resulta una alternativa muy competitiva, puesto que no existe un método riguroso para realizar esta actividad, si además es apoyada con un hardware tan versátil como un FPGA, los resultados son positivos para este tipo de sistemas computacionales aplicados a diversa tareas de ingeniería.

6. Trabajo a futuro

Con el deseo de continuar con el desarrollo alcanzado por este trabajo, se proponen las siguientes ideas: Proponer otro algoritmo para la obtención de los valores y vectores propios de una matriz; modificar el método de generación aleatoria de la población inicial; por último, aplicar el sistema desarrollado en

diversas tareas específicas, por ejemplo, la sintonización de controladores de diversos tipos.

Agradecimientos. Los autores agradecen el apoyo del Instituto Politécnico Nacional (SIP-IPN, COFAA-IPN, BEIFI-IPN) y del gobierno mexicano (SNI y CONACYT).

Referencias

1. Affi, Z., Badreddine, E., Romdhane, L.: Advanced mechatronic design using a multi-objective genetic algorithm optimization of a motor-driven four-bar system. Mechatronics 17(9), 489–500 (2007)
2. Altera: Tutoriales sobre cómo usar el procesador Nios II (2015), `https://www.altera.com/`
3. Calle, A., Pazmiño, P., Ponguillo, R.: Algoritmo de detección de bordes en imágenes con Nios II (2014)
4. Cupertino, F., Mininno, E., Lino, E., Naso, D.: Optimization of position control of induction motors using compact genetic algorithms. In: IECON 2006, 32nd Annual Conference on IEEE Industrial Electronics. pp. 55–60. IEEE (2006)
5. Hansen, N.: The CMA evolution strategy: A tutorial. Vu le 29 (2005)
6. Hernández Lara, D.: Implementación sobre FPGA de la estrategia evolutiva CMA-ES pp. 1–139 (2014)
7. Higuchi, T., Yao, X.: Evolvable hardware, vol. 11. Springer Science and Business Media (2006)
8. Juang, J.G., Chiou, H.K., Chien, L.H.: Analysis and comparison of aircraft landing control using recurrent neural networks and genetic algorithms approaches. Neurocomputing 71(16), 3224–3238 (2008)
9. Kim, D.H., Abraham, A., Cho, J.H.: A hybrid genetic algorithm and bacterial foraging approach for global optimization. Information Sciences 177(18), 3918–3937 (2007)
10. Moreno-Armendáriz, M.A., Cruz-Cortés, N., Duchanoy, C.A., León-Javier, A., Quintero, R.: Hardware implementation of the elitist compact genetic algorithm using cellular automata pseudo-random number generator. Computers & Electrical Engineering 39(4), 1367–1379 (2013)
11. Raygoza, J.J., Ortega, S., Chirino, C.A., Rivera, J.: Implementación en hardware de un SVPWM en un soft-core Nios II, Parte I. e-Gnosis 7 (2009)
12. Rechenberg, I.: Evolution strategy: Optimization of technical systems by means of biological evolution. Fromman-Holzboog (1973)
13. Ross, O.H.M., Sepulveda, R.: High Performance Programming for Soft Computing. CRC Press (2014)
14. Schwefel, H.P.P.: Evolution and Optimum Seeking: The Sixth Generation. John Wiley & Sons, Inc. (1993)
15. Scott, S.D., Seth, S., Samal, A.: A synthesizable VHDL coding of a genetic algorithm. Tech. rep., Technical Report UNL-CSE-97-009, University of Nebraska-Lincoln (1997)
16. Shackleford, B., Snider, G., Carter, R.J., Okushi, E., Yasuda, M., Seo, K., Yasuura, H.: A high-performance, pipelined, FPGA-based genetic algorithm machine. Genetic Programming and Evolvable Machines 2(1), 33–60 (2001)