UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO

***FACULTAD DE INGENIERÍA ELÉCTRICA***
***DIVISIÓN DE ESTUDIOS DE POSGRADO***

# "IMPLEMENTACIÓN DE UN ALGORITMO DE EVOLUCIÓN DIFERENCIAL PARALELO BASADO EN UNIDADES DE PROCESAMIENTO GRÁFICO"

## TESIS

QUE PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS EN INGENIERÍA ELÉCTRICA

PRESENTA

ING. SERGIO JHOVANNE DOMÍNGUEZ GONZÁLEZ

DIRECTOR DE TESIS

DR. NORBERTO GARCÍA BARRIGA

MORELIA, MICHOACÁN AGOSTO DE 2011

# IMPLEMENTACIÓN DE UN ALGORITMO DE EVOLUCIÓN DIFERENCIAL PARALELO BASADO EN UNIDADES DE PROCESAMIENTO GRÁFICO

**TESIS**

Que para obtener el grado de

**MAESTRO EN CIENCIAS EN INGENIERÍA ELÉCTRICA**

presenta

**Sergio Jhovanne Domínguez González**

**Norberto García Barriga**

**Director de Tesis**

Universidad Michoacana de San Nicolás de Hidalgo

Agosto 2011

# Agradecimientos

A mi asesor el Dr. Norberto García Barriga, quien tuvo a bien apoyarme y supervisarme durante el desarrollo de esta tesis.

A la División de Estudios de Posgrado de la Facultad de Ingeniería Eléctrica de esta Universidad, por establecer las condiciones para desenvolverme dentro de un excelente ambiente de trabajo y convivencia.

Al Consejo Nacional de Ciencia y Tecnología, CONACYT, por el soporte económico brindado durante estos estudios de Maestría.

A mis revisores de tesis, el Dr. José Antonio Camarena, Dr. Francisco Rivas y de manera especial al Dr. Juan J. Flores y el Dr. Félix Calderón por su importante influencia durante mi formación académica.

A mis compañeros Abel, Luis, Rafa por su apoyo en determinados momentos, y en especial a Robert, quien además se involucró a la par conmigo en esta área de investigación.

Al MSI. Dionisio Buenrostro Cervantes, por su apoyo para poder ralizar el examen de grado en tiempo, además de la confianza depositada en mi persona.

# Lista de Publicaciones

"A HYBRID PARALLEL PROCESSING METHODOLOGY FOR SOLVING LARGE SCALE PROBLEMS USING PVM AND SSH"
R. Olmos, S. Domínguez, L. García, R. López, A. Tapia, N. García,
Transforming Research through High Performance Computing,
ISUM Conference Proceedings, págs. 161–166,
Guadalajara, Jalisco, México, Marzo 2010
ISBN: 978-607-450-348-7

# Resumen

En esta tesis se presenta la implementación de un algoritmo de Evolución Diferencial (ED) paralelo basado en Unidades de Procesamiento Gráfico (GPUs). La aplicación de la tecnología de última generación de las GPUs y el entorno de programación CUDA permiten explotar el paralelismo inherente en el algoritmo diferencial evolutivo y reducir el esfuerzo computacional asociado a operaciones tales como la creación de la población inicial, la evaluación de los individuos, la mutación y la selección. El algoritmo de ED implementado en esta tesis es una herramienta de optimización global, la cual está basada en una estrategia de búsqueda que involucra cierto nivel de voracidad, esto a través de la inclusión de información acerca de la mejor solución obtenida a cada iteración y la eliminación de la operación de recombinación. Esta implementación paralela basada en unidades de procesamiento gráfico se evalúa con un conjunto de problemas de optimización usados como referencia en la literatura especializada. Estos problemas involucran la localización y el cálculo del mínimo global de funciones tales como la función de Rosenbrock, Griewank y Ackley. Se propone una definición novedosa de los vectores mutantes, en la cual se sustituye los valores constantes de las variables de control por valores aleatorios. Además, se omite la implementación de la operación de cruza dentro del algoritmo, consiguiendo a través de estas dos variantes la reducción de los problemas asociados a la sintonización de los parámetros de ejecución del algoritmo. Por otra parte, el efecto de la variación en el tamaño de población es analizado en términos de precisión, eficacia, tiempos de cómputo y factores de aceleración. Se reportan mejoras significativas en términos de tiempos de ejecución con respecto a dos implementaciones programadas en C y ejecutadas en la CPU. La primera de estas implementaciones se refiere a la versión secuencial del algoritmo, mientras que la segunda se refiere a una versión en la cual se paraleliza tan sólo la operación de la evaluación, debido a las limitaciones del lenguaje C. Se reporta además la comparación con una versión en donde se paraleliza solamente la operación de la evaluación en la GPU, esto con la intención de presentar el beneficio obtenido por la implementación principal, en la cual se paralelizan todas las operaciones evolutivas a través de la utilización de una novedosa biblioteca incluida en las versiones más recientes de CUDA para la generación paralela de números aleatorios. La versiónes del algoritmo de Evolución Diferencial basadas en GPU fueron implementadas en una tarjeta de video GPU Tesla C2050 con 448 unidades de procesamiento.

# Abstract

A parallel Differential Evolution (DE) algorithm based on Graphics Processing Units (GPUs) approach is presented in this thesis. The application of state-of-the-art GPUs and the Compute Unified Device Architecture (CUDA) programming platform allows exploiting the inherent parallelism of the DE algorithm and reducing the computational effort associated with evolutionary operations such as creation of initial population, evaluation, mutation and selection. The DE algorithm implemented in this work is a global optimization tool which is based on a search strategy that involves a certain level of greediness. This is done through the incorporation of information about the best solution obtained at each iteration and the elimination of the recombination operation. This parallel implementation based on graphics processing units is tested on a set of well-known benchmark optimization problems reported in literature. This benchmark problems involve the location and the calculation of the global minimum in functions such as the Rosenbrock's, Griewank's and Ackley's functions. A novel mutant vector definition is proposed, which replaces the constant values of the control variables by random values. Besides, the crossover operation is not implemented, obtaining through these two variants the reduction of problems associated with the tuning of the algorithm's execution parameters. Furthermore, the effect of varying the population size is analyzed in terms of accuracy, effectiveness, runtimes and speedup factors. Significant improvements in terms of runtimes are reported with respect to two implementations coded in C language and executed on the CPU. The first of these implementations refers to the sequential version of the DE algorithm, while the second one refers to a parallel version in which only the evaluation operation is parallelized, due to the limitations of C language. It is also reported the comparison with a GPU-based version in which only the evaluation operation is parallelized, this is done with the intention of presenting the benefit achieved by the main implementation, in which all evolutionary operations are parallelized through the use of a novel library for parallel random number generation included in the most recent versions of CUDA. The GPU-based Differential Evolution algorithm versions were implemented using a video card GPU Tesla C2050 with 448 processing units.

# Contenido

# Lista de Figuras

# Lista de Tablas

# Listados

# Lista de Símbolos

| | |
|---|---|
| ACO | *Ant Colony Optimization.* |
| AE | Algoritmo Evolutivo. |
| AG | Algoritmo Genético. |
| API | *Application Programming Interface.* |
| CPU | *Central Processing Unit.* |
| CUDA | *Compute Unified Device Architecture.* |
| CURAND | CUDA *Random library.* |
| D | Número de dimensiones. |
| DRAM | *Dynamic Random Access Memory.* |
| ED | Evolución Diferencial. |
| EDCP | Evolición Diferencial Completamente Paralela. |
| $g$ | Número de generación. |
| $g_{max}$ | Número máximo de generaciones. |
| GPU | *Graphics Processing Unit.* |
| GPGPU | *Gereral-Purpose Computation in GPU.* |
| L1 | *Layer 1.* |
| L2 | *Layer 2.* |
| LD/ST | *Load/Store.* |
| MP | *Multiprocessor.* |
| MPI | *Message Passing Interface* |
| $N_{eval}$ | Número de evaluaciones. |
| $N_p$ | Tamaño de la población. |
| PCIe | *Peripheral Component Interconnect - Express.* |
| PSO | *Particle Swarm Optimization.* |
| PVM | *Parallel Virtual Machine.* |
| RGBA | *Red Blue Green Alpha.* |
| SA | *Simulated Annealing.* |
| SFU | *Special Function Unit.* |
| SM | *Streaming Multiprocessor.* |
| SP | *Streaming Processor.* |
| UAL | Unidad Aritmético-Lógica. |

Capítulo 1

# Introducción

En la década de los años 70 surge un nuevo campo dentro de la inteligencia artificial propuesto por John Henry Holland conocido como Algoritmos Genéticos (AGs) [Holland75]. Los AGs son técnicas de optimización estocásticas inspiradas en el esquema poblacional de la teoría de la evolución biológica y su base genética. Esta teoría denominada Teoría de la Selección Natural fue propuesta por Charles Darwin y expresa: "*Existen organismos que se reproducen y la progenie hereda características de sus progenitores, existen variaciones de características si el medio ambiente no admite a todos los miembros de una población en crecimiento. Entonces aquellos miembros de la población con características menos adaptadas (según lo determine su medio ambiente) morirán con mayor probabilidad. Entonces aquellos miembros con características mejor adaptadas sobrevivirán más probablemente*"[Darwin59]. La analogía entre un algoritmo genético y el proceso evolutivo formulado por C. Darwin surge al tratarse la evolución biológica de un proceso adaptativo, el cual puede ser considerado como un proceso hipotético de optimización. En su obra [Holland75] John Holland proporciona un marco general para el análisis de sistemas adaptativos (naturales y artificiales), estableciendo que cualquier problema de adaptación puede ser formulado en términos genéticos. Una vez formulados en estos términos, tales problemas pueden ser resueltos por medio de AGs. Al igual que en la naturaleza en donde las poblaciones de individuos se transforman y diversifican con el paso de las generaciones para mejorar sus características particulares y elevar la calidad de la población a través de operaciones genéticas. Los al-

goritmos genéticos son procedimientos poblacionales e iterativos que simulan la evolución biológica en donde ocurren operaciones genéticas [Koza92]. Cada iteración es denominada generación y cada cromosoma uno de los individuos dentro de la población representa una solución propuesta al problema de optimización que se esté abordando [Haupt04]. De forma similar al proceso observado en la naturaleza, se establece la esperanza de que la calidad de los individuos mejore con el transcurso de las generaciones, es decir, que las soluciones mejoren con el paso del tiempo hasta que alguna de ellas logre evolucionar hasta acercarse lo suficiente al óptimo de la función objetivo del problema de optimización que se esté abordando. Además, se realiza un proceso de selección de acuerdo a ciertos criterios definidos por las características particulares del problema y en función de los cuales se identificará a los individuos más aptos (mejores soluciones). Tal como ocurre en la naturaleza, los individuos más aptos tendrán mayor probabilidad de sobrevivir y recombinarse, ser afectados por algún proceso de mutación y de ese modo dar paso a la siguiente generación.

La simulación del proceso de la evolución biológica y de otros fenómenos naturales ha extendido el campo de los AGs hasta establecer un campo más amplio de métodos de optimización conocidos como Algoritmos Evolutivos (AEs), en el cual la mayoría de los métodos se basan en los principios básicos de los AGs. Los algoritmos evolutivos más comunes son los Algoritmos Genéticos [Holland75], el método de Recocido Simulado (SA) [Aarts89], Búsqueda Tabú (TS) [Glover97], el método de Optimización por Enjambres de Partículas (PSO) [Eberhart01], la Optimización por Colonia de Hormigas (ACO) [Dorigo97], y algunos más como el método de Evolución Diferencial (ED) (DE del inglés *Differential Evolution*) [Storn95]. Esta última variante involucra los mismo principios, pero es considerado más bien una alternativa que no representa el modelo de ningún fenómeno natural. Sin embargo, la ED ha recibido mucha atención en los últimos años dentro del campo de la computación evolutiva debido a que ha demostrado ser un método de optimización simple, preciso y robusto [Lee08]. De la misma manera que en los AEs comunes, ED es un método poblacional de optimización que ataca el problema del punto inicial de búsqueda obteniendo muestras de la función ojetivo en múltiples puntos iniciales de búsqueda aleatoriamente elegidos. Al igual que otros métodos poblacionales, ED genera nuevos puntos en el espacio de búsqueda los cuales son perturbaciones de puntos existentes. Estas perturbaciones son

la diferencia escalada de dos vectores pertenecientes a la población aleatoriamente seleccionados. Por cada vector integrante de la población se produce un nuevo vector de prueba ED a través de la suma la diferencia escalada a un tercer vector integrante de la población aleatoriamente seleccionado. Durante la etapa de selección el vector de prueba producido compite contra su vector correspondiente dentro de la población en donde el vector con mejor valor de aptitud se selecciona como miembro de la siguiente generación, convirtiéndose en los progenitores del ciclo evolutivo. [Price05] Todos los algoritmos evolutivos mencionados poseen características particulares, pero en general comparten operaciones básicas como son: selección, recombinación o cruza y mutación.

## 1.1. Antecedentes

Los AEs han demostrado ser una buena alternativa a los métodos clásicos de optimización, proporcionando soluciones a problemas prácticos en diversas áreas del conocimiento tales como el diseño [Guoyan06, Hongying10, Huicong10], finanzas [Ting07, Brabazon08, Castillo07], teoría de juegos [Niu06, Jin07, Lang10], robótica y control [McGregor92, Meng07, Shi10], bioinformática [Pal06, Fogel07, Cervantes10], procesamiento de imágenes [Aravind02, Zhang10, Shinn10] y reconocimiento de patrones [Pei-Fang, Espejo10]. Particularmente en el área de sistemas de potencia se han reportado aplicaciones para la programación de generación [Wong95, Rudolf99] y mantenimiento [Kim97, Burke00], despacho económico [Yalcinoz01, Khamsawang10], planeación, reconfiguración y expansión de redes de distribución [Vitorino09, Días09, Gallego09], ubicación óptima de dispositivos de compensación de potencia reactiva [Lirui08, Fonseca09] y observadores para prevención y diagnostico de fallas [Días09], control [McGregor92, Abido00], entre otras. En [Vitorino09], por ejemplo, es posible encontrar una aplicación de un AG en conjunto con el uso de la simulación de Monte Carlo para mejorar la confiabilidad y disminuir las pérdidas de potencia activa en un sistema de distribución radial a través de un proceso de reconfiguración de la red. Un AG mejorado es utilizado dada la enorme cantidad de posibles configuraciones y la necesidad de una búsqueda eficaz. Dichas mejoras implican probabilidades adaptativas de cruza y mutación y algunas otras características como lo son el uso de múltiples puntos de cruza y la

implementación de una operación adicional de clonación. La efectividad de esta propuesta se demostró a través del estudio de un sistema de distribución radial de 69 nodos. Una aplicación relacionada al análisis de sistemas dinámicos se describe en [Villanueva10], en donde se presenta una herramienta auxiliar en el trazo de diagramas de bifurcación a través del uso de técnicas metaheurísticas utilizadas para la búsqueda de puntos fijos, empleando por defecto optimización por enjambre de partículas. En este trabajo se presentan además las ventajas y desventajas de estas técnicas sobre los métodos tradicionales y se destaca en la experimentación un caso de estudio que involucra el análisis de una red eléctica de tres nodos. Otra aportación interesante se encuentra en [Mahdad09], donde se propone un algoritmo genético eficiente para resolver un problema multi-objetivo de flujos óptimos de potencia con restricciones ambientales. El esquema empleado de paralelismo se basa en la partición del problema principal de flujos óptimos de potencia en dos sub-problemas vinculados. Por un lado se tiene la planeación de generación de potencia real a través de un esquema redundante que realiza varias ejecuciones del AG propuesto, mientras que por otro lado es la planeación de la potencia reactiva para realizar los ajustes necesarios en los valores obtenidos por el AG paralelo. La robustez de esta propuesta se validó poniéndolo a prueba en la red de 59 buses de Algeria, y fue comparada con métodos secuenciales y convencionales de optimización global, como lo es el mismo AG y la optimización por colonia de hormigas, obteniendo como resultado que el enfoque propuesto es capaz de obtener soluciones competitivas en tiempos razonables.

Diversos trabajos han sido publicados en los últimos años sobre el desempeño exitoso de AEs en la solución de problemas que han demostrado ser difíciles o imposibles de resolver a través de la aplicación de un método determinístico. Además, se ha encontrado que las soluciones a las que se ha llegado a través de metodologías convencionales no son satisfactorias [Lee08]. Todas las variantes de los algoritmos evolutivos que se han creado a partir del surgimiento de los AGs han sido propuestas con el objetivo de mejorar el desempeño de estos procedimientos. Importantes esfuerzos se han realizado con el objeto de elevar la eficacia y eficiencia de estos métodos. Por un lado, las líneas de investigación apuntan al descubrimiento y mejora de estratégias de búsqueda, capaces de encontrar una solución precisa y eficiente, mejorando de esta manera la eficacia de los AEs. Un ejemplo

de estas propuestas se describe en [Zhixiang07], en donde se propone un esquema híbrido el cual proporciona un AG con las ventajas de un algoritmo de gradiente para reducir los inconvenientes de los AGs en términos de precisión. La estrategia seguida en [Zhixiang07] es seleccionar las mejores soluciones obtenidas como puntos iniciales de búsqueda para el algoritmo de gradiente para después usar los resultados del método de gradiente y compararlos con aquellos obtenidos por el AG, obteniendo de esta comparación las mejores soluciones. Dicha estrategia fue aplicada para la estimación de parámetros de sistemas continuos, cuyas simulaciones mostraron que el esquema empleado produjo mejores resultados en términos de rapidez y eficiencia que aquellos producidos por un algoritmo genético simple. En [Qing06] se hace la introducción a una estrategia novedosa denominada evolución diferencial dinámica, la cual es en esencia una estratégia de evolución diferencial adicionando dinámicamente una actualización de la población la cual involucra el uso de poblaciones virtuales más grandes las cuales responden rápidamente a cambios en la población original. Esta propuesta se pone a prueba a través de la comparación con el método de evolución diferencial común mostrando importantes mejoras en térmimos de eficicacia, robustez y requerimientos de memoria para diversos casos de estudio.

La evolución es un proceso altamente paralelo. Dentro de un AG todas las operaciones aplicadas a los individuos (evaluación, mutación, selección e incluso su creación, para formar parte de una población inicial) se puede llevar a cabo de manera independiente. Este paralelismo inherente existente en los AGs hace que se ajusten muy bien para su ejecución en sistemas de cómputo multiprocesador tales como workstations, clusters y grids, o mediante el uso de herramientas de procesamiento paralelo como PVM [Tasoulis04] y MPI [Kwedlo06]. Las tendencias han sido mejorar el uso de los recursos de que se disponga para implementar estos algoritmos con la intención de reducir los tiempos asociados a su ejecución. En los últimos años ha aumentado la accesibilidad a los sistemas de cómputo multiprocesador y por consiguiente se han vuelto más comunes las implementaciones paralelas de AEs. Esta ha sido una ruta que se ha seguido en la mejora contínua de los AEs es en términos de eficiencia. Los desarrollos paralelos han provisto una sustancial mejora en los tiempos de ejecución y un aumento en gran medida en el potencial de este tipo de métodos de optimización lo que ha llevado a los AEs a ser aplicados para la solución de

problemas cada vez más complejos [Lee08].

Una alternativa de reciente desarrollo que brinda un alto nivel de accesibilidad para el desarrollo de proyectos de cómputo de alto desempeño son las GPUs. El rápido aumento en la programabilidad y capacidad de procesamiento de estos dispositivos durante los últimos años ha dado lugar a una comunidad de investigación que se ha dedicado a resolver, a través del uso de esta tecnología, una amplia gama de problemas complejos y computacionalmente demandantes en diversas áreas del conocimiento. Una referencia importante puede encontrarse en [Owens08], en donde se discute el surgimiento del cómputo de propósito general en las GPUs, conocido como GPGPU (General-Purpose computation on GPUs), el cual ha posicionado en la actualidad a las GPUs no sólo como microprocesadores alternativos sino que se han convertido en los motores tradicionales de procesamiento en los sistemas de cómputo de alto rendimiento del futuro.

No obstante todas las herramientas y alternativas que existen para el procesamiento paralelo, en todas las aportaciones revisadas reportadas en la literatura especializada que han explotado el paralelismo natural de los AEs se han enfocado sólo en la paralelización de la evaluación de los individuos [Pit95, Cantú98, Haupt04, Tasoulis04, Wong06, Kwedlo06, Li07, Lee08, Mahdad09, Arora10, Veronese10, Zhu09], por mencionar algunos. Dado que la evaluación no es la única operación susceptible de ser paralelizada. Este hecho sugiere que no se ha logrado conseguir el máximo beneficio por parte de las implementaciones paralelas de AEs propuestas o presentadas y revisadas en las literatura especializada. La exclusiva paralelización de la operación de evaluación en AEs se justifica por dos razones:

1. Se considera que la evaluación representa la operación más costosa en términos computacionales [Haupt04, Lee08], por lo que realizar esta operación de manera paralela sobre cada uno de los individuos (más que cualquier otra operación) reduce en gran medida los tiempos de ejecución asociados al proceso de optimización.

2. Es necesario considerar que los AEs son procedimientos estocásticos que basan sus resultados en probabilidades que cambian con el tiempo. La naturaleza de estos métodos implica la necesidad constante de generar números aleatorios durante todo el proceso de optimización, la cual es una tarea difícil [Halprin10, Tsong07]. Dado que

la operación asociada a la evaluación de los individuos es una operación que no implica la generación de números aleatorios dentro del algoritmo, se puede justificar la paralelización exclusiva de esta operacion por motivos de simplicidad.

A pesar de los inconvenientes mencionados, se han realizado importantes contribuciones que han explotado el paralelismo de los AEs. La paralelización del método de evolución diferencial se describe en [Tasoulis04], en donde se reporta una mejoría significativa en la velocidad de ejecución y el desempeño de dicho algoritmo implementando una solución basada en PVM, herramienta que permite emular una computadora con varios procesadores a través de una red homogénea o heterogénea de computadoras, sumando de manera virtual su poder de cómputo por moderado que este sea. En [Kwedlo06] se describe el uso de un algoritmo de evolución diferencial para su aplicación en el proceso de entrenamiento de una red neuronal. En dicha referencia se emplea un esquema basado en MPI, una herramienta de intercambio de mensajes que permite usar una red de computadoras y la arquitectura maestro-esclavo para aprovechar la existencia de múltiples procesadores y de esa manera agilizar los cálculos. En dos contribuciones más recientes [Veronese10] y [Zhu09], se propone un enfoque basado en GPU para la aceleración de un algoritmo de evolución diferencial aplicado a la minimización de un conjunto de funciones bien conocidas en el área de optimización. En [Veronese10] se obtienen resultados muy buenos en términos de reducción del esfuerzo computacional y precisión en las soluciones. Sin embargo, se emplea un esquema poco eficiente de generación de números aleatorios, en el cual se generan de manera secuencial en la CPU una cantidad suficiente de números aleatorios que usará el algoritmo de manera paralela. En [Zhu09] se presenta además una mejora al algoritmo de ED en términos de precisión, a través de la adición de una etapa de búsqueda local para cada solución propuesta por el algoritmo. En esta aportación se logró también una disminución importante en el esfuerzo de cómputo, aunque la evaluación de las implementaciones fué a menor escala. En [Li07] se presenta la implementación de un algoritmo genético finamente granulado y basado en el uso de GPUs como motor principal de procesamiento. Dicha propuesta emplea además un esquema de manejo de datos que utiliza la memoria usada para la representación de texturas lo cual permite acelerar la ejecución el algoritmo. De igual manera, podemos encontrar en [Arora10] una contribución en la paralelización de

un AG binario y uno real a través del uso de una GPU donde se describen factores de aceleración significativos los cuales fueron medidos respecto a la versión secuencial de los mismos algoritmos. En [Wong06] los autores presentan un AG convencional implementado para ejecutarse en unidades de procesamiento gráfico en donde se implementan variantes a la operación de mutación y una selección pseudo-determinística. En este algoritmo genético paralelo todos los pasos son llevados a cabo de manera paralela a excepción de la generación de números aleatorios. Las implementaciones paralela y secuencial son comparadas analizando la eficacia y eficiencia de la estrategia de selección empleada usando un conjunto de problemas de optimización.

Además del uso de las GPUs para la paralelización de AEs, también se encuentran reportadas en la literatura varias contribuciones acerca de la aplicación de GPUs dentro del campo de la Ingeniería Eléctrica. En [Zainud-Deen09] se utiliza un dispositivo GPU para resolver el problema de dispersión electromagnética mediante el método de diferencias finitas en el dominio de la frecuencia. La implementación en la GPU mostró una aceleración mayor a trés veces sobre el código ejecutado en el CPU implementado en MATLAB. En relación a esta última aportación, en [Unno09] y [Demir10], se describe una técnica de aceleración basada en GPU del método de diferencias finitas en el dominio del tiempo, destacando en la primera de estas contribuciones la simulación masivamente paralela de un campo electromagnético, mostrando un alto desempeño cuando el algoritmo es programado en la arquitectura de una GPU. Finalmente, se tienen dos aportaciónes más relacionadas a los sistemas de potencia. En [Gopal07] se presenta la implementación del análisis de contingencias basado en flujos óptimos de potencia de corriente directa en una unidad de procesamiento gráfico. En esta aportación se realiza la evaluación de la implementación empleando sistemas estándar de prueba de la IEEE. Los resultados obtenidos mostraron un factores de aceleración cercanos a 4, los cuales crecen de acuerdo a la complejidad del sistema. En [García10] se utiliza una GPU para obtener la solución de flujos de potencia el sistema del sistema de 118 nodos del IEEE. En esta contribución se implementó el método de gradiente biconjugado y el método Newton-Raphson para resolver las correcciones de voltaje, en donde la tarea más demandante de este método iterativo es realizada en la GPU. La exactitud y la eficiencia de las soluciónes de flujos de potencia calculadas en la GPU son reportadas

en esta contribución con una tarjeta C870 con 118 unicades de procesamiento. Por último, en [Jalili09] se hace uso de estos dispositivos para lograr una rápida y precisa simulación de estabilidad transitoria de sistemas de potencia de gran escala. El dispositivo usado es una GPU NVIDIA GeForce GTX 280 con 240 núcleos de procesamiento y 1GB de memoria física. La precisión de la simulación propuesta fue validada empleando el software PSS/E y varios casos de estudio, obteniendo un factor de aceleración de 345 sobre la simulación ejecutada solamente en la CPU para un sistema de 1248 buses y 320 generadores.

## 1.2. Descripción del problema

La precisión de las soluciones proporcionadas por los algoritmos evolutivos ha sido una de las desventajas más mencionadas por los críticos de estos métodos. El pobre nivel de precisión de los AEs se acentúa cuando la función objetivo es continua y diferenciable, problemas para los cuales la mayoría de las veces se requiere que la solución encontrada tenga un alto grado de exactitud. Debido a esta limitante se han dirigido importantes esfuerzos con la intención de mejorar el desempeño de estos procedimientos heurísticos. La mayoría de las contribuciones que se han reportado posterior al surgimiento de los AGs han tenido el objetivo de superar este inconveniente.

Aunado a la falta de precisión en sus soluciones, los AEs comparten un serio problema con los métodos de optimización convencionales, el cual está asociado a que la capacidad de encontrar el óptimo de una función objetivo disminuye conforme aumenta la dimensionalidad del problema. Para el caso particular de los AEs, encontrar la solución a problemas de gran dimensionalidad implica aumentar significativamente el número de individuos dentro de la población en proporción al número de dimensiones y tamaño del espacio de búsqueda dentro de cada una de estas dimensiones. Es necesario considerar además, que los tiempos de ejecución crecen en proporción al tamaño de población, lo cual afecta directamente el desempeño del algoritmo.

El problema asociado a los grandes tiempos de ejecución requeridos por parte de los AEs es otro importante inconveniente y ha sido la razón de muchas críticas. En el área de ingeniería, por ejemplo, es común el uso de funciones objetivo que involucren simula-

ciones complejas para la solución de un problema empleando un algoritmo evolutivo. Estas funciones objetivo requieren un esfuerzo considerable de cómputo y es necesario realizar un gran número de evaluaciones de la misma a lo largo de todo el proceso de optimización [Haupt04]. La cantidad de evaluaciones en un AE está definido por el tamaño de la población y el número de iteraciones totales, por lo que al incrementar ambos o alguno de estos dos parámetros aumenta significativamente los tiempos de ejecución.

El uso de sistemas de cómputo multiprocesador han permitido la implementación de desarrollos paralelos de AGs, gracias a su paralelismo inherente. Estos desarrollos han provisto una sustancial mejora en los tiempos de ejecución y un aumento en gran medida en el potencial de este tipo de métodos de optimización. Aunque en la actualidad los sistemas de cómputo de alto rendimiento se han vuelto más accesibles, su uso aún se encuentra restringido a un sector muy reducido de la comunidad científica. Los costos de los sistemas multiprocesador aún son considerables, lo cual limita el gran poder de cómputo de ciertos sistemas a un número relativamente pequeño de usuarios. Una alternativa a los costosos sistemas multiprocesador es el uso de sistemas de cómputo distribuido, pero se tiene que considerar que el uso de este tipo de sistemas sólo es redituable cuando el tiempo requerido para la comunicación e intercambio de datos entre los nodos de procesamiento es considerablemente menor que los tiempos asociados al esfuerzo computacional que se va a ejecutar en cada uno de estos nodos.

A pesar de los avances obtenidos en cuanto a la distribución del esfuerzo computacional requerido por los AEs entre múltiples unidades de procesamiento, el desempeño de versiones paralelas de AEs ha sido limitado debido a que los desarrollos paralelos de AEs se han enfocado únicamente en la ejecución paralela de la evaluación de los individuos, tal como es mencionado en la literatura. Este hecho implica un deterioro en el desempeño global de estos algoritmos dado que la paralelización parcial de los AEs disminuye el beneficio que puede ser obtenido por una implementación de este tipo. Sin embargo, se ha justificado el no paralelizar el resto de las operaciones evolutivas debido a que requieren de la generación de números aleatorios, lo cual es un problema complejo en el sentido de que es una tarea difícil de lograr. En la actualidad, existen bibliotecas para diversos lenguajes de programación las cuales incluyen funciones para la generación de números aleatorios. Las

más eficientes son capaces de generar grandes series de números pseudo-aleatorios. Cada una de estas series tiene asociado un periodo, el cual es directamente proporcional a la calidad de la serie producida. En este sentido, cuanto más grande sea el periodo de la serie, mejor será la calidad del generador de números pseudo-aleatorios. No obstante, ninguna de estas biliotecas es capaz de generar números aleatorios de forma paralela, es decir, las funciones asociadas a la generación de números aleatorios no pueden ser llamadas bajo un contexto de programación multi-hilo y no funcionan correctamente durante la ejecución simultánea de multiples hilos. Por esta razón no es posible paralelizar operaciones tales como la creación de la poblacion inicial, mutación y la cruza dentro de un AE.

Un problema adicional se refiere al hecho de que todos los AEs requieren la elección de diversos parámetros para su funcionamiento. Es importante mencionar que los valores de dichos parámetros han tenido la necesidad de ser definidos empíricamente, requiriendo una enorme cantidad de extenuantes experimentos. El conocimiento teórico de los efectos de estos parámetros en la convergencia sigue siendo un problema abierto en la actualidad [Lee08].

## 1.3. Objetivos de la investigación

Esta tesis de investigación tiene como objetivo general la implementación de un sistema eficiente de minimización global paralelizado basado en Evolución Differencial y Unidades de Procesamiento Gráfico (GPUs) para la solución de problemas de optimización de gran escala. La implementación paralela de este algoritmo evolutivo tiene como principal finalidad la reducción de los tiempos asociados a la ejecución de los algoritmos de optimización ejecutados comúnmente en la CPU.

Los objetivos particulares de este trabajo se describen a continuación:

1. Incrementar la eficiencia de un algoritmo de Evolución Diferencial mediante su implementación paralela usando una GPU y el entorno de desarrollo CUDA.

2. Implementar variantes al método original de ED con el objetivo de aumentar su eficacia y reducir sus requerimientos externos para el control de su ejecución.

3. Realizar la paralelización de todas las operaciones evolutivas dentro del algoritmo de Evolución Diferencial en la GPU a través del uso de la biblioteca CURAND para la generación de números aleatorios de manera masivamente paralela en la GPU.

4. Comparar el desempeño de la total paralelización del algoritmo de ED en la GPU contra una segunda implementación en la GPU en la cual se paraleliza exclusivamente la operación de la evaluación.

5. Comparar el desempeño de la variante de ED implementada contra la implementación del esquema original de ED, ambos ejecutados en la GPU.

6. Analizar el desempeño de la implementación paralela basada en unidades de procesamiento gráfico programada en CUDA y compararla con las versiones secuencial y paralela implementadas para ejecutarse en la CPU.

7. Poner a prueba la implementación en la GPU, así como las implementaciones en la CPU, para solucionar problemas de optimización ampliamente conocidos en la literatura tales como la función de Rosenbrock, Griewank y la función de Ackley, todas estas continuas y diferenciables.

## 1.4. Justificación

En general los algoritmos evolutivos son menos precisos y mucho más demandantes computacionalmente que los métodos clásicos de búsqueda local. Sin embargo, un algoritmo evolutivo tiene la ventaja de no depender de una condición inicial, lo cual puede afectar el desempeño y la eficiencia de un método basado en gradiente. Durante el desarrollo del presente trabajo de investigación se identificó al método de Evolución Diferencial como un método robusto de búsqueda, preciso y relativamente rápido, capaz de tratar funciones objetivo de gran dimensionalidad con una proporción relativamente pequeña en los tamaños de las poblaciones. El método de Evolución Diferencial posee ventajas sobre el resto de los AEs tales como simplicidad y versatilidad [Storn95]. Todo esto ha resaltado este método como una excelente alternativa a los AEs convencionales, superando algunos inconvenientes relacionados a la mayoría de los AEs.

Debido a la naturaleza paralela que poseen los AEs es posible realizar implementaciones paralelas en sistemas multiprocesador, lo cual beneficia directamente el desempeño de estos procedimientos, disminuyendo su tiempo de ejecución en función del número de procesadores existentes en el sistema de cómputo que los ejecuta. La aceleración en el proceso de ejecución ha sido la razón más citada en la literatura para el uso del cómputo paralelo en algoritmos evolutivos. Dado que las evaluaciones y operaciones evolutivas a las que son sometidas las poblaciones pueden ser completadas de manera simultánea para cada individuo, es común distribuir estas tareas entre múltiples procesadores o núcleos de procesamiento. El paralelismo en algoritmos evolutivos se ha vuelto importante, dado el incremento en el uso de sistemas de cómputo paralelo y las mejoras permanentes de las tecnologías computacionales asociadas al desarrollo de multi-procesadores cada vez más rápidos y eficientes. Sin embargo, la gran amayoría de las contribuciones encontradas durante el estudio del estado del arte de esta tesis han explotado el paralelismo natural de los AEs sólo de manera parcial enfocandose únicamente en la paralelización de la evaluación de los individuos.

Las GPUs se han convertido en dispositivos altamente paralelos con una gran cantidad de núcleos de procesamiento y gran poder de cómputo. En comparación con los sistemas convencionales multiprocesador (workstations, clusters, grids y supercomputadoras), las GPUs poseen un gran ancho de banda, evitando en gran medida los problemas existentes en los sistemas multiprocesador convencionales, problemas asociados a la sincronización y comunicación entre los nodos de cómputo. En la actualidad las GPUs son un medio alternativo mucho más accesible para la obtención de un gran poder computacional, brindando el mismo poder de cómputo que los sistemas quad-core más recientes, a una décima parte del precio y consumiendo una vigésima parte de la energía [NVIDIA11].

## 1.5. Metodología

Para alcanzar los objetivos establecidos en esta tesis se establece en primera instancia el estudio y comprensión del modelo de programación de la plataforma CUDA, el cual puede verse en términos generales como un conjunto de extenciones del lenguaje de programación C. Sin embargo, es necesario comprender el paradigma empleado para la de-

finición y ejecución de las tareas paralelas. Para llevar a cabo la parte práctica del análisis de la plataforma CUDA se requiere una GPU capaz de soportar la ejecución de la misma. La GPU usada en este trabajo de investigación es una tarjeta NVIDIA Tesla C2050, con 2688 MB de memoria física y la versión 3,2 de la plataforma CUDA. Esta GPU cuenta con 14 multiprocesadores, cada uno con 32 núcleos resultando en un total de 448 procesadores de flujo.

La GPU se encuentra montada en un sistema de cómputo CPU AMD Phenom II, 3.0 GHz quad-core con 3.9 GB de memoria física. El sistema operativo empleado es Linux en su distribución Ubuntu versión 10.04 de 64 bits. En esta plataforma se ejecutarán las implementaciones secuenciales y paralelas programadas en C.

Se implementarán dos versiones en la CPU que se usarán para comparar el desempeño de la propuesta implementada en la GPU. La primera de estas implementaciones se refiere a la versión secuencial del método, mientras que la segunda se trata de una versión multi-hilo. En esta última, únicamente se paraleliza la operación de evaluación, debido a las limitantes del lenguaje de progración C. Estos códigos se ejecutarán en el sistemas multiprocesador de 4 núcleos y usando técnicas de programación paralela.

Una vez completada la etapa de implementación en la CPU, se pasará a la etapa de desarrollo en la GPU para la cual se necesita en primera instancia la obtención de una versión funcional del algoritmo, para la cual se propone paralelizar todas las operaciones evolutivas: evaluación, selección mutación e incluso la creación de la población inicial. A continuación es necesario realizar una optimización en la programación, esto con el objetivo de aumentar la eficiencia de la aplicación a través de la explotación de los recursos de cómputo contenidos dentro del dispositivo GPU, como son los espacios de memoria constante, memoria compartida y texturas.

Las dificultades en la generación de números aleatorios que impiden llevar a cabo una paralelización total de un AE serán superadas haciendo uso de la biblioteca CURAND, incluida en la versión más reciente de CUDA. Dicha biblioteca proporciona un conjunto de funciones para generar secuencias de números pseudo-aleatorios de manera eficiente y paralela con un periodo mayor a $2^{190}$ en la GPU. Las funciones integradas en la biblioteca CURAND generan secuencias que satisfacen la mayoría de las propiedades estadísticas de

una verdadera secuencia de números aleatorios, además de brindar las facilidades para generar dichas secuencias normalmente distribuidas y uniformemente distribuidas [NVIDIA10].

Dos versiones más serán implementadas en la GPU, las cuales corresponden a: 1).- el esquema original de ED y 2).- la exclusiva paralelización de la operación de evaluación. La primera de estas se realiza con el objetivo de comparar el desempeño de la variante implementada al algoritmo original de ED, mientras que la segunda de estas versiones servirá como referencia para determinar la rentabilidad de la completa paralelización de las operaciones evolutivas en el algoritmo de ED por parte del desarrollo principal.

Se realizarán experimentos con cuatro funciones continuas de las cuales tres de ellas son diferenciables. Estas funciones han sido ampliamente utilizadas en la literatura como pruebas de desempeño de algoritmos de optimización. Estas funciones son la función de Rosenbrock, Griewank, Ackley y la función $f_4$ reportada en [Lee04], [Veronese10] y [Zhu09].

## 1.6. Descripción de la Tesis

El contenido general de los capítulos que conforman esta tesis se describe a continuación:

En el Capítulo 1 se proporciona una breve introducción al campo de los algoritmos evolutivos, estableciendo sus antecedentes como técnicas de optimización y destacando sus principales ventajas y las propuestas plasmadas en la literatura para subsanar sus principales desventajas. Además, se presenta la propuesta de este trabajo para resolver las principales desventajas de los algoritmos evolutivos y la metodología empleada para cumplir el objetivo general de este trabajo de investigación.

En el Capítulo 2 se presenta una introducción a la arquitectura de las GPUs y el estudio del estado del arte referente al uso de GPUs en la solución de problemas en diversas áreas de la ciencia y la tecnología. Se explican los conceptos más importantes asociadas al entorno de programación de CUDA y se describe el paradigma de programación de dicha plataforma. Por último, se hace referencia al uso de los recursos de cómputo de que disponen este tipo de dispositivos, resaltando sus principales características y la manera en cómo pueden ser explotados.

En el Capítulo 3 se describen los principales conceptos y el esquema de optimización empleado por los algoritmos evolutivos en la solución de problemas. Se explican más detalladamente el ajuste de los algoritmos evolutivos a un esquema de procesamiento palalelo y se explica la metodología empleada por el algoritmo de evolución diferencial desarrollado a lo largo del presente trabajo de investigación, enfatizando en el uso de una GPU como herramientas de procesamiento paralelo de propósito general.

Con el objetivo de que el presente trabajo pueda servir de referencia para futuros desarrollos basados en GPUs, en el Capítulo 4 se presenta un ejemplo de desarrollo, el cual describe la implementación de la etapa de evaluación del algoritmo de evolución diferencial, programada para su ejecución en la GPU mediante el uso de la plataforma de programación CUDA.

En el Capítulo 5 se presenta la experimentación y obtención de resultados para diversos casos de estudio, los cuales involucran un conjunto de problemas de optimización altamente estudiados, mediante los cuales se ponen a prueba las aplicaciones desarrolladas basado en GPUs y en la plataforma de cómputo unificado como CUDA. Los resultados obtenidos son comparados con la versiones secuenciales y paralelas de los mismos algoritmos implementados en el lenguaje de programación C.

En el Capítulo 6 se presentan las conclusiones generales de la tesis, aportaciones y trabajos futuros.

Capítulo 2

# Unidades de procesamiento gráfico

Impulsados por la gran demanda del mercado de los gráficos interactivos en 3D de alta definición, las unidades de procesamiento gráfico (GPUs por sus siglas en inglés *Graphic Processor Unit*), han evolucionado hasta convertirse en arquitecturas paralelas masivas con una gran cantidad de procesadores y con un gran poder de cómputo.

Una GPU es un microprocesador especializado que permite acelerar la representación de los gráficos desde la unidad central de proceso o CPU, con el objetivo de aligerar su carga de trabajo. De esta manera, mientras la GPU se encarga de la representación de los gráficos, la CPU se puede encargar de otros cálculos como la inteligencia artificial usada en videojuegos y aplicaciones 3D interactivas. Estos dispositivos se utilizan en sistemas que van desde teléfonos móviles, computadoras personales, estaciones de trabajo y consolas de videojuego. Las GPUs modernas son muy eficientes en la manipulación de los gráficos en una computadora, con una arquitectura altamente paralela que ha mostrado ser más eficiente que las CPUs de propósito general para la implementación y ejecución de ciertos algoritmos [NVIDIA10d]. Una GPU puede estar presente en una PC en la forma de una tarjeta de vídeo, o en la actualidad puede incluso ser la tarjeta base.

La representación de gráficos 3D interactivos es una tarea computacionalmente demandante y altamente paralela. Este proceso requiere del cálculo complejo desarrollado por una computadora cuyo objetivo es generar una imagen 2D a partir de una escena 3D, considerando información acerca de la geometría de los elementos en la escena, perspectiva,

texturas, iluminación y sombreado. En aplicaciones de procesamiento multimedia tales como post-procesamiento y escalamiento de imagenes, codificación y decodificación de video, visión estéreo y reconocimiento de patrones, es posible dividir la información en bloques de datos para ser procesados de manera paralela. El esquema de cómputo paralelo en una GPU mapea cada elemento en el conjunto de datos con el procesamiento paralelo de hilos de ejecución o *threads*. Muchas aplicaciones que procesan grandes conjuntos de datos pueden emplear un modelo de programación de datos en paralelo para acelerar el cómputo. Este enfoque puede producir un rendimiento mayor en comparación con el desempeño de una CPU convencional.

NVIDIA introdujo en 2006 una arquitectura de cómputo paralelo de propósito general denominada CUDA (del inglés *Compute Unified Device Architecture*), capaz de aprovechar la capacidad de cómputo contenida en una GPU con el objetivo de resolver problemas computacionalmente complejos de una manera más eficiente. CUDA es un entorno de programación integral que permite a los desarrolladores utilizar el lenguaje C como lenguaje de implementación de alto nivel. CUDA tiene una interfaz de programación de aplicaciones API (del inglés *Application Programming Interface*) la cual incluye un conjunto de extensiones para el lenguaje de programación C y permite declarar funciones específicas de un programa normal de C para ejecutarse en una GPU. Lo anterior permite que los programas escritos en C puedan tomar ventaja de la capacidad de una GPU sin dejar de hacer uso de la CPU cuando sea apropiado. CUDA es la primera API que permite a las aplicaciones basadas en la CPU acceder directamente a los recursos de una GPU, sin las limitaciones de la utilización de una API de gráficos.

Es de suma importancia considerar que no todos los problemas complejos pueden ser resueltos eficientemente en una GPU [Veronese10]. Una GPU resulta especialmente adecuada para resolver problemas complejos que puedan ser expresados como el cálculo de datos en paralelo. Además de esta limitante, se debe tomar en cuenta que la programación de propósito general será mucho más compleja que la realizada en una arquitectura convencional basada en CPU. El modelo de programación de CUDA puede potencializar las deficiencias en las habilidades de los programadores, al no existir la recursividad ni apuntadores a funciones. Esto forza a los desarrolladores a escribir métodos que pasen y

reciban una gran cantidad de datos. Un problema más está asociado a lo especializada de la documentación existente, por lo cual puede resultar difícil comenzar a hacer desarrollos empleando el entorno y paradigma de programación de CUDA. A pesar de los inconvenientes mencionados, tanto el entorno de programación de CUDA como la arquitectura de las GPU continua evolucionando, aumentando cada vez más la programabilidad de estos dispositivos y mejorando las caractarísticas de este lenguage de programación.

**Nota:** Durante el presente capítulo será común el uso de terminología en inglés debido a su ámplia y actual aceptación dentro de esta área del conocimiento.

## 2.1. Glosario de términos

A pesar de que CUDA es considerado, en términos generales, como un conjunto de extensiones del lenguaje de programación C, este entorno de programación introduce también una serie de nuevos términos, sintaxis y palabras reservadas. A continuación se presenta la terminología más importante, con el objetivo de facilitar el entendimiento de los conceptos relacionados a este modelo de programación.

### GPU

Una GPU es una unidad especializada de procesamiento dedicada a la representación de gráficos, comúnmente llamadas tarjetas de video. En los últimos años estas unidades han superado el objetivo principal para el que fueron creadas, convirtiéndose en poderosos motores no sólo para el procesamiento de gráficos sino también para el cómputo de propósito general.

### CUDA

CUDA es una arquitectura de cómputo paralelo de propósito general que permite aprovechar el gran poder de cómputo contenido en las GPUs. Se compone de un compilador y un conjunto de herramientas de desarrollo que permiten realizar la programación de algoritmos en la GPU con una sintáxis parecida a la del lenguaje de programación C.

### Procesadores de flujo de CUDA

Se denomina procesador de flujo (*Streaming Processor*, SP) a cada uno de los núcleos de procesamiento de flujo contenidos en una GPU. En la actualidad, una GPU puede contener hasta 448 núcleos. Estos núcleos de procesamiento se encuentran agrupados en unidades denominadas multiprocesadores de flujo (*Streaming Multiprocessors*) o SMs.

### *Thread* de CUDA

Se refiere a cada uno de los hilos de ejecución que se encargan de realizar una operación de manera paralela sobre determinado conjunto de datos. Estos hilos de ejecución o *threads* tienen un identificador único al cual se puede tener acceso a través de la variable integrada `threadIdx`.

### Bloque de *threads*

Un bloque de *threads* es un conjunto de threads organizados en arreglos unidimensionales, bidimensionales o tridimensionales, de tal manera que la variable integrada `threadIdx`, empleada para identificar cada *thread* es un vector de 3 componentes, al cual se puede referir como `threadIdx.x`, `threadIdx.y` y `threadIdx.z`. El tamaño de los bloques de threads se encuentra disponible a través de la variable integrada `blockDim`, llamados `blockDim.x`, `blockDim.y` y `blockDim.z`. Este esquema proporciona un método natural para realizar cómputo a través de estructuras de datos tales como vectores, matrices y arreglos de 3 dimensiones.

### Grid de bloques de *thread*

Los bloques de threads a su vez se encuentran dispuestos dentro de un arreglo en una o dos dimensiones denominado *grid*. El número y la organización de los bloques de threads en un grid son determinados por el tamaño de los datos que se procesan y además por la cantidad de procesadores disponibles en el sistema. Para identificar los bloques de threads dentro de un grid se emplea la variable integrada `blockIdx`. la cual es un vector de 2 componentes que se denominan `blockIdx.x` y `blockIdx.y`.

### Warp

Para poder ser atendidos, los *threads* requieren ser calendarizados y formados en una cola de ejecución. Para esto, los calendarizadores agrupan a los *threads* en conjuntos de 32 dentro de estructuras denominadas *warps*.

### Host

Dentro del entorno de programación de CUDA se denomina *host* a la parte del sistema de cómputo que involucra la unidad central de proceso o CPU y los recursos de cómputo de que dispone directamente. La declaración de una función precedida por el calificador `__host__` especifica una función que será ejecutada en la CPU y que puede ser llamada únicamente por otras funciones que sean ejecutadas en la CPU.

### Device

Se denomina *device* al conjunto de recursos contenidos dentro de una GPU (núcleos de procesamiento y memoria), es decir, se refiere a la GPU en si misma. Además, la declaración de una función precedida por el calificador `__device__` especifica una función que será ejecutada en la GPU y podrá ser llamada únicamente por funciones que hayan sido declaradas para ejecutarse en la GPU. De la misma manera, la declaración de una variable que esté precedida por el calificador `__device__` hace referencia a una variable que va a residir en la memoria de la GPU.

### Kernel de cómputo

Un kernel de cómputo es la parte fundamental en el modelo de programación de CUDA. Se refiere a una función que se va ejecutar en el *device* y que puede ser llamada solamente desde el *host*. El esquema de procesamiento paralelo en CUDA se basa en la declaración, llamado y ejecución de una función kernel. El calificador `__global__` sirve para declarar una función como un kernel de cómputo, el cual podrá ser ejecutado N veces en paralelo por N CUDA *threads* diferentes.

**Textura**

Las texturas son conjuntos de datos (común mente una imagen) los cuales se utilizan para cubrir la superficie de un objeto gráfico bidimensional o trimensional. En una GPU la disposición nativa de los datos es bidimensional, debido al hecho de que están diseñadas para representar y desplegar geometría en dos dimensiones para gráficos. Los datos correspondientes a una textura son almacenados en espacios de memoria bidimensionales que ofrecen diferentes modos de direccionamiento y filtrado para formatos de datos específicos. Los espacios de memoria para texturas residen en el *device* y son almacenados en caché, por lo que la extracción de un dato almacenado en textura tiene el costo de una lectura a la memoria del *device* sólo cuando ocurra un error de caché. De otra manera, el costo será únicamente una lectura del caché de textura.

**Texel**

Cada uno de los elementos en las texturas se denomina texel y representa la unidad mínima de una textura aplicada a una superficie. Texel es la contracción del inglés *texture element*. De la misma forma que una imagen digital se representa mediante una matriz de pixeles, una textura es representada mediante un matriz de texeles. El proceso de aplicar una textura a un elemento gráfico es conocido como mapeado de texturas, el cual consiste en asignar texeles a los pixeles correspondientes que aparecerán en la imagen final en una pantalla.

**CUDA array**

Los CUDA *arrays* son estructuras de datos integradas y optimizadas empleadas para la escritura de datos en los espacios de memoria para texturas. Para almacenar datos en dichos espacios de memoria es necesario copiarlos primero a un CUDA *array*, a partir del cual se realiza un ligado a una textura que haya sido declarada previamente. Los CUDA *array* están ligados a las texturas sólo por el número de elementos contenido en el arreglo y los datos contenidos en una textura sólo pueden ser leídos desde un kernel de cómputo a través de una operación denominada extracción de textura.

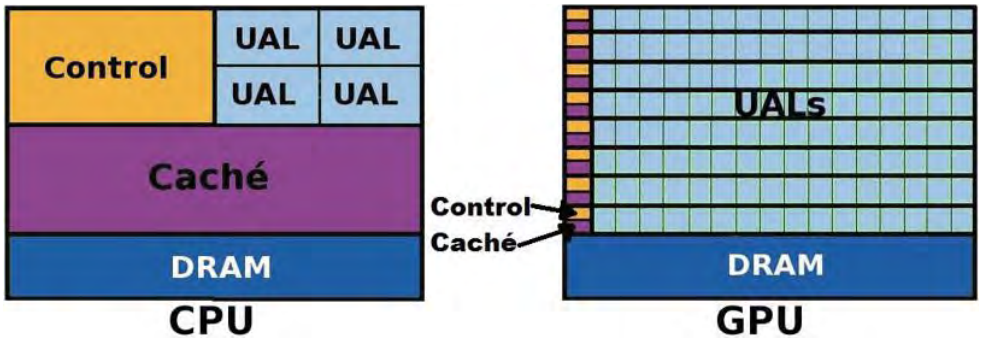



Figura 2.1: Diferencia entre las arquitecturas CPU y GPU.

## 2.2. Arquitectura de las GPUs

Una GPU se especializa en realizar cálculo intensivo y altamente paralelo en tareas asociadas a la representación de gráficos. En comparación con una CPU, ésta tiene un diseño optimizado para maximizar el desempeño de programas secuenciales, en los cuales se hace uso de una sofisticada lógica de control para llevar a cabo la administración de un número pequeño de unidades aritmético-lógicas o UALs. Además, una CPU está provista de grandes espacios de memoria caché para reducir las operaciones de memoria física en aplicaciones grandes y complejas. Por otra parte, una GPU se ajusta muy bien para la solución de problemas que pueden ser expresados como cálculos de datos en paralelo a través de un número mayor de UALs. La Figura 2.1 ilustra esquemáticamente las diferencias entre la arquitectura de una CPU y una GPU. Se puede apreciar que la GPU posee un mayor número de unidades aritmético-lógicas.

Debido a que el esquema de paralelismo implementado por las GPUs es a nivel de datos, una GPU está diseñada de tal forma que se dediquen más transistores al procesamiento de datos en lugar de emplearlos para el almacenamiento de datos en caché o al control de flujo. En una GPU todos los procesadores reciben la misma instrucción, pero operan sobre diferentes conjuntos de datos, la misma instrucción es ejecutada de manera síncrona por todas las unidades de procesamiento, lo cual se conoce en computación como SIMD (siglas en inglés de *Single Instruction, Multiple Data*). Dado que las mismas instruc-

ciones son ejecutadas para muchos conjuntos o elementos de datos, y que estas se realizan de manera síncrona, existe un requerimiento muy bajo de control de flujo [Kirk10], teniéndo únicamente que asegurarse de que cada unidad de procesamiento cumpla con su tarea.

Las principales diferencias entre una CPU y un GPU se producen en términos del manejo de los hilos de ejecución y la administración de memoria. A continuación se describen tres puntos importantes de comparación [NVIDIA10a]:

**Ejecución de *threads***

Las CPU soportan un número limitado de *threads* concurrentes, por ejemplo, un servidor con cuatro procesadores quad-core pueden correr sólo 16 *threads* concurrentemente (32 en algunas arquitecturas recientes usando *HyperThreading*). En comparación, la unidad más pequeña ejecutable de paralelismo en un dispositivo CUDA comprende 32 *threads*. Todas las GPUs NVIDIA soportan al menos 768 *threads* concurrentemente activos por SM, y algunas GPUs soportan 1024 o más *threads* activos por SM. Lo anterior significa que dispositivos con 30 SMs permiten la ejecución de más de 30,000 *threads* concurrentemente activos.

**Complejidad de los *threads***

Los *threads* que se ejecutan en una CPU son entidades de mucha importancia, los cuales demandan una cantidad considerable de recursos. Para que un sistema operativo incorpore la capacidad de *multi-threading*, deben realizarse una gran cantidad de intercambios de *threads* entre los canales de ejecución que conforman una aplicación, denominados intercambios de contexto. Estos intercambios de contexto durante la ejecución de una aplicación son lentos y costosos. Por otra parte, los *threads* en una GPU son extremadamente simples. En un sistema típico, cientos de *threads* son formados en una cola de ejecución en conjuntos de 32 denominados *warps*. Debido a que registros separados se asignan a todos los *threads* activos, no es necesario el intercambio de registros o de estado entre los *threads* de la GPU sino que los recursos permanecen asignados a cada *thread* hasta que completa su ejecución.

**Uso de memoria**

Tanto el *host* como el *device* poseen memoria de acceso aleatorio (RAM). En el *host*, la RAM es general y equitativamente accesible por el código de un programa dentro de las limitaciones impuestas por el sistema operativo. En el *device*, la memoria RAM es dividida tanto virtual como físicamente en diversos tipos de memoria tales como memoria global, constante, compartida y texturas, cada una de las cuales tiene su propósito particular y es capaz de satisfacer diferentes necesidades.

### 2.2.1. Beneficio de una arquitectura masivamente paralela

La principal motivación de la programación masivamente paralela llevada a cabo en las GPUs es el incremento en la velocidad de las aplicaciones. Cuando es posible ajustar una aplicación a un esquema de procesamiento en paralelo, una implementación eficiente en la GPU puede lograr una aceleración de hasta 100×, es decir, que se ejecutará hasta 100 veces más rápido que la versión secuencial de la misma aplicación. En [Kirk10] se presenta un caso de estudio el cual implica la reconstrucción de una imagen de resonancia magnética, para el cual la reconstrucción más eficiente empleando una GPU NVIDIA Quadro FX-5600 alcanza una aceleración de 108× a través de la explotación de la memoria caché y las unidades de funciones especiales a través de las herramientas provistas por el entorno CUDA. La aceleración de una aplicación depende de muchos factores, sin embargo, en la medida que una apliación incluya paralelismo en sus datos es sencillo para una GPU lograr una aceleración de al menos 10×. La magnitud del beneficio en la aceleración de una aplicación implementada en una GPU depende de la medida en que puede ser paralelizada. En general, el código que no pueda ser paralelizado eficientemente deberá ejecutarse en el *host*, a menos que esto implique excesivas transferencias de datos entre el *host* y el *device*.

La ley de Amdahl [Amdahl67] es un modelo que especifica la aceleración esperada (en inglés *speedup*) debido a la paralelización de ciertas partes en un programa. Esencialmente, la aceleración de un programa que utiliza múltiples procesadores para su ejecución es limitada por el tiempo requerido por la parte secuencial del programa. Esto se define

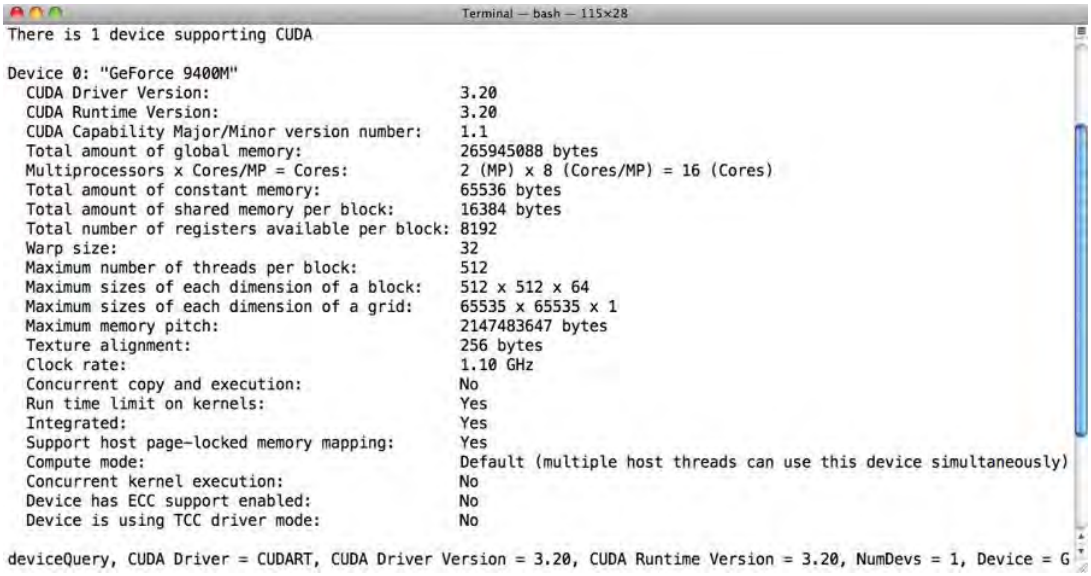



Figura 2.2: Ejemplo de los datos de configuración de un dispositivo habilitado para CUDA.

como sigue,

$$1 \leq Speedup = \frac{1}{r_s + \frac{r_p}{n}} \leq n \tag{2.1}$$

en donde $r_s + r_p = 1$, $r_s$ representa la proporción de la parte secuencial y $r_p$ la parte paralelizada en un programa y $n$ representa el número de unidades de pricesamiento. Todas las contribuciones en la literatura sugieren enfocar esfuerzos para incrementar $r_p$, dado que el punto clave es que entre más grande sea $r_p$ más grande es el beneficio en términos de aceleración.

Además de la medida en que una aplicación puede ser paralelizada, el beneficio que se puede obtener de una arquitectura paralela depende de la capacidad de cómputo que ésta posea. La capacidad de cómputo de una GPU en particular depende de sus características de hardware, el conjunto de instrucciones soportadas por el dispositivo y de otras especificaciones tales como los tamaños de los espacios de memoria, el número de registros disponibles y el orden máximo que pueden tener los arreglos de *threads* y bloques de *threads*. En la Figura 2.2 se presentan las características y especificaciones que denotan la capacidad de cómputo de la GPU GeForce 9400M, comúnmente integrada en computadoras portátiles. En este sen-

tido, los desarrolladores deben identificar principalmente el número de multiprocesadores (SMs) en cada dispositivo, el número de núcleos de procesamiento (SPs) por multiprocesador, la cantidad de memoria disponible, y aquellas características especiales que éste posea. Dentro del entorno de programación CUDA, toda la información descrita en la Figura 2.2 puede ser obtenida a través del llamado de la función `cudaGetDeviceProperties()`.

A continuación se listan los aspectos más relevantes de la información presentada en la Fig. 2.2,

**Total amount of global memory:** Se refiere al total de la memoria global del dispositivo, es decir la memoria DRAM.

**Multiprocessors × Cores/MP = Cores:** Describe la cantidad de unidades SM o MPs contenidas en el dispositivo así como la cantidad de unidades SP (denominadas *Cores*) dentro de cada SM. El total de unidades SP en cada SM multiplicado por el número de unidades SM contenidas en el dispositivo nos da el total de SPs, *Cores*, núcleos o unidades de procesamiento de que se dispone.

**Total amount of constant memory:** Es el total de memoria constante de que se dispone. La memoria constante es un tipo de memoria de sólo lectura y es la memoria de más rápido acceso de manera global. Aunque su espacio es muy limitado, es conveniente explotar este recurso cada vez que sea posible. El aprovechamiento de este espacio de memoria puede mejorar de manera sustancial el rendimiento de cualquier aplicación que necesite realizar una gran cantdad de accesos a determinado conjunto de datos [NVIDIA10d].

**Total amount of shared memory per block:** Se refiere al total de memoria compartida disponible para todos los *threads* dentro de un bloque. La memoria compartida forma parte de la memoria caché de las unidades SM.

**Maxium number of threads per block:** Es el número máximo de *threads* que podemos crear dentro de en un bloque.

**Maxium sizes of each dimension of a block:** Considerando que los bloques de *threads* pueden ser unidimensionales, bidimensionales o tridimensionales, estos datos descri-

ben el máximo número de *threads* que podemos tener en $x, y, z$, respectivamente. Es importante señalar que para realizar la organización de los *threads* que integrarán un bloque, es necesario considerar estos datos al mismo tiempo que se considera el número máximo de *threads* que puede contener el bloque.

**Maxium sizes of each dimension of a grid:** Los *grids* de bloques de *threads* pueden ser unidimensinales o bidimensionales, por lo cual estos datos expresan el número máximo de bloques que puede contener un *grid* tanto en la dimensión $x$ como en la dimensión $y$, obviamente para la dimensión $z$ este número es 1.

**Clock rate:** Representa la frecuencia de reloj de cada uno de las unidades SP.

**Concurrent copy and execution:** Indica si el dispositivo permite iniciar el procesamiento de los datos aún cuando estos no han sido copiados en su totalidad en la memoria de la GPU, de esta manera no es necesario esperar a que todo el bloque de datos sea almacenado en la memoria del dispositivo para poder iniciar con el procesamiento de los mismos, disminuyendo de esta manera el tiempo de respuesta por parte de la GPU.

**Run time limit on kernels:** Indica si el dispositivo es capaz de forzar la terminación de un kernel si este se ha demorado en finalizar su ejecución. Este atributo de la GPU sólo es válido en entornos gráficos, en donde en tétminos visuales también evita que la imagen se congele en la pantalla. Si el entorno gráfico es desactivado, esta característica también se desactiva permitiendo a los kérneles de cómputo realizar su ejecución durante el tiempo que sea necesario.

**Compute mode:** Indica la manera en que puede ser utilizado el dispositivo. El valor por defecto (*default*) indica que varios *threads* de la CPU pueden utilizar la GPU simultáneamente.

**Concurrent kernel execution:** Indica si es posible ejecutar múltiples kerneles de cómputo simultáneamente.

**Device has ECC support enabled:** Indica si el dispositivo es capaz de ejecutar códigos para la la corrección de errores en los datos.

## 2.3. La arquitectura de cómputo Fermi

Se conoce como Fermi a la arquitectura más moderna de cómputo en las GPUs desarrollada por la corporación NVIDIA. El surgimiento de la arquitectura Fermi establece una nueva generación de GPUs, cuya característica principal es la capacidad para realizar cálculos con desempeño de doble presición y un mayor nivel de programabilidad y eficiencia [NVIDIA09]. La arquitectura de cómputo Fermi, la cual puede estar configurada hasta con 14 SMs, cada uno con 32 SPs. Tal como se puede apreciar en la Figura 2.3, la arquitectura Fermi cuenta con un calendarizador de trabajo de tecnología denominada GigaThread (Figura 2.3), el cual se encarga de distribuir los bloques de CUDA *threads* entre los SMs disponibles, balanceando dinámicamente la carga de trabajo en la GPU. Una innovación que incluye la tecnología GigaThread es la capacidad para llevar a cabo la ejecución de múltiples kernels simultáneamente [Nickolls10].

La interfaz con el *host* (*Host Interface*) conecta la memoria DRAM de la GPU

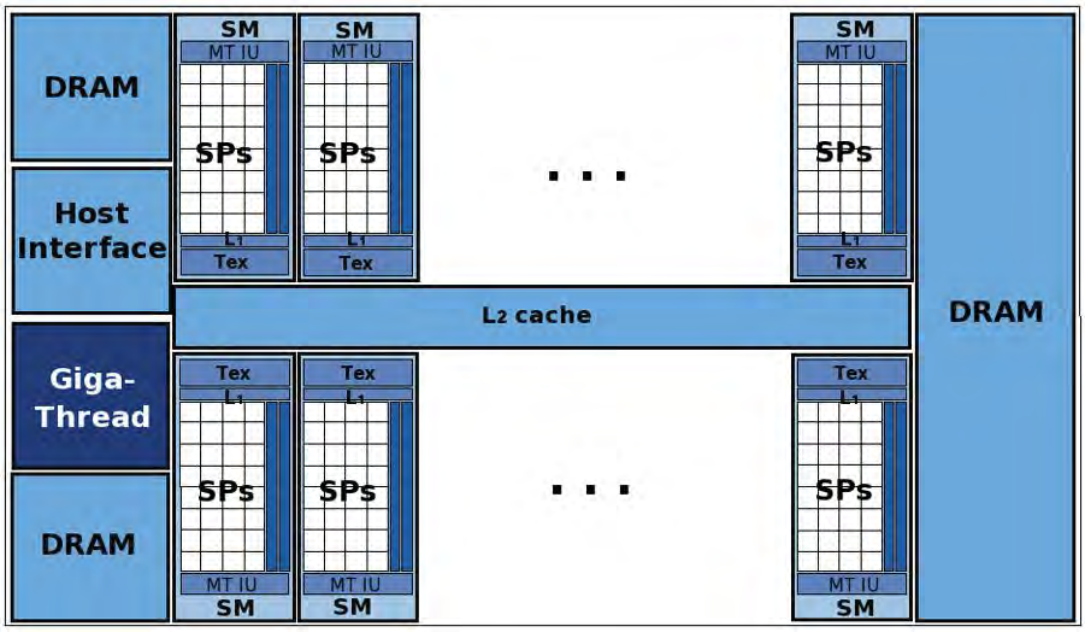



Figura 2.3: Arquitectura Fermi.

con el sistema de memoria de la CPU a través del puerto PCI. De esta manera el sistema de cómputo heterogéneo CPU+GPU lleva a cabo un co-procesamiento y una transferencia de datos bidireccional a través de esta interfaz [NVIDIA09].

Cada SM tiene en un primer nivel (L1) su propio caché de datos y al mismo tiempo todos los SMs comparten un segundo nivel (L2) de caché común y unificado de 768 KB cuya principal función es actuar como una extención de memoria para texturas [Nickolls10]. El caché L2 se conecta con las interfaces DRAM y la interfaz PCIe la cual se encuentra conectada al sistema de memoria de la CPU. De esta manera, en el nivel de caché L2 es capaz de almacenar datos tanto de la memoria global de la GPU como páginas de memoria en la CPU accedidas a través de la interfaz PCIe [Nickolls10].

### 2.3.1. El multi-procesador de flujo (SM)

La unidad fundamental de configuración en la arquitectura de una GPU son los multi-procesadores de flujo (SMs). Cada SM es capaz de proporcionar suficientes SPs y memoria compartida para ejecutar uno o más bloques de CUDA *threads*. La memoria compartida permite a las tareas paralelas ejecutadas en estos núcleos compartir datos sin la necesidad de ser enviados a través del bus de memoria del sistema. Las GPUs capaces de soportar el entorno de programación CUDA poseen varias SMs que en total pueden conjuntar hasta 448 SPs, los cuales pueden ejecutar de manera simultánea y colectiva miles de *threads*.

Los SMs integrados en la arquitectura Fermi han aumentado su programabilidad y eficiencia en comparación con generaciones anteriores. En la actualidad cada SM presenta hasta 32 SPs, en donde cada SP está integrado por una unidad de punto flotante (FP unit) y una unidad aritmético-lógica de enteros (INT unit) (ver Fig. 2.4). Ambas unidades proveen una precisión de 32 bits para todas las instrucciones y están eficientemente optimizadas para soportar una precisión de hasta 64 bits. Las unidades SM soportan varias instrucciones: boleanas, desplazamientos, movimientos, comparaciones y conversiones [Nickolls10].

Cada SM posee 16 unidades LD/ST (load/store units, ver Fig. 2.4), las cuales permiten la lectura y el almacenamiento de datos en memoria caché o directamente en la memoria DRAM de la GPU. Así mismo, cada SM cuenta con cuatro unidades para fun-

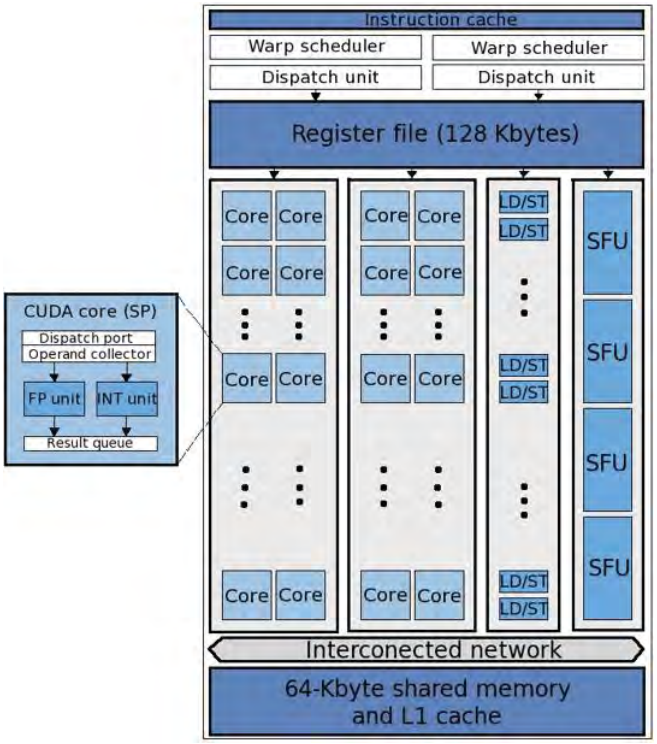



Figura 2.4: Multi-procesador de flujo en la arquitectura Fermi.

ciones especiales (SFU), las cuales ejecutan funciones trascendentales como senos, cosenos, funciones inversas y raíces cuadradas.

Los SMs ejecutan bloques de *threads* canalizándolos para su ejecución en grupos de 32 llamados *warps*. Cada SM es capaz de atender hasta 1,536 *threads* concurrentes. Cada SM cuenta con dos calendarizadores de warps o *Warp schedulers* y dos unidades despachadoras de instrucciones o *Dispatch units*(ver Fig. 2.4), las cuales permiten que dos *warps* sean distribuidos y ejecutados paralelamente. De esta manera, la mayoría de las instrucciones pueden ser doblemente ejecutadas al mismo tiempo; dos instrucciones enteras, dos instrucciones flotantes, o incluso una mezcla de instrucciones enteras, de punto flotante, de lectura, de escritura y de funciones especiales. Por otra parte, las instrucciones de doble precisión no soportan despacho doble con ninguna otra operación.

En la arquitectura Fermi, cada SM cuenta con 64 Kbytes de memoria integrada

la cual puede ser configurada como 48 Kbytes de memoria compartida y 16 Kbytes de memoria caché L1 (ver Fig. 2.4). El uso de memoria compartida proporciona importantes beneficios en el desempeño de aplicaciones restringidas por el ancho de banda, ya que se reduce el tráfico desde la memoria DRAM. Además, aplicaciones que no utilicen memoria compartida, se ven automáticamente beneficiadas por el caché L1 [NVIDIA09].

## 2.4. El modelo de programación de CUDA

El modelo de programación concurrente de CUDA proporciona un paralelismo anidado que guía a los programadores a dividir un problema en sub-problemas que pueden ser resueltos en paralelo por bloques de *threads*. Al mismo tiempo, estos sub-problemas pueden ser divididos en piezas más finas las cuales pueden ser resueltas cooperativamente en paralelo por los *threads* que integran un bloque. Este esquema de particionado de un problema se describe en la Figura 2.5, el cual se ajusta perfectamente al esquema de paralelismo anidado en la jerarquía de *threads* (ver Sección 2.1), en dicha sección se especifica la consideración de los *grids* de bloques de *threads* como arreglos hasta en dos dimensiones (2D), mientras que los bloques de *threads* son considerados como arreglos hasta en 3D. Debido a la jerarquía especificada es posible indexar los *threads* de CUDA a través de cinco dimensiones (5D). Como consecuencia, la Figura 2.5 es útil para describir tanto la granularidad de un problema como la disposición de los *threads* dentro de un bloque y los bloques de *threads* dentro de un *grid*.

Otra característica importante de este modelo de programación es la escalabilidad automática que posee. Este escalamiento automático implica que una GPU con más SMs ejecutará automaticamente un programa paralelo en menos tiempo que una GPU con menor número de núcleos en forma transparente al programador. En la Figura 2.6 se ilustra esquemáticamente la ejecución de un mismo programa en dos GPUs con diferente número de multiprocesadores de flujo. Como se puede apreciar en la Figura 2.6, la GPU con 2 SMs requiere un tiempo $T_1$ para completar la ejecución del programa en cuestión. Por su parte, la segunda GPU con 4 SMs requiere de un tiempo $T_2$, en donde $T_2 < T_1$.

La ejecución de cualquier programa en CUDA se realiza esencialmente en 4 etapas.

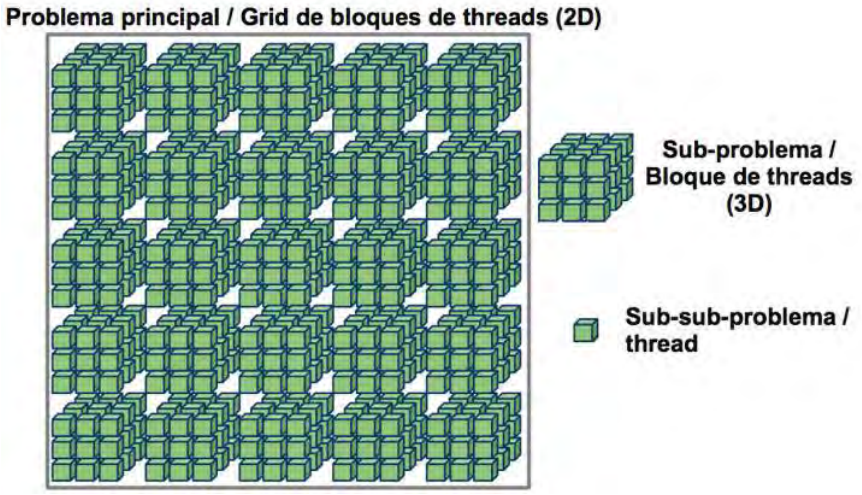



Figura 2.5: Analogía entre la partición de un problema y la jerarquía de los threads en CUDA.

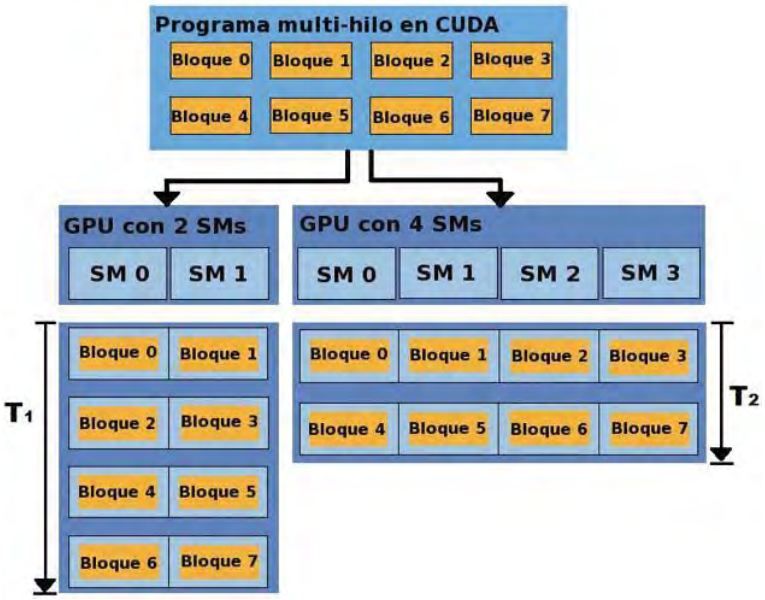



Figura 2.6: Escalabilidad automática.

Estas etapas se ilustran esquemáticamente en la Figura 2.7 y son descritas a continuación en el orden en que se llevan a cabo:

1. Se copian los datos desde la memoria principal en la CPU a la memoria de la GPU.
2. La CPU instruye a la GPU el procesamiento que debe realizarse sobre los datos.
3. La GPU ejecuta en paralelo la misma instrucción sobre diferentes conjuntos de datos.
4. Se copia el resultado del procesamiento de la memoria de la GPU a la memoria principal en la CPU.

En general, un programa en CUDA es un código unificado que involucra tanto código en la CPU como en la GPU. Tal como se aprecia en la Figura 2.7, cualquier programa basado en GPU comienza y finaliza su ejecución en la CPU.

### 2.4.1. Estructura de un programa en CUDA

La arquitectura CUDA y su software asociado fueron diseñados con dos objetivos principales [NVIDIA10c]:

- Proveer con un conjunto de extensiones para lenguajes de programación estándar (C y C++, Fortran, OpenCL, Python) que permitan una programación directa de algoritmos paralelos con un requerimiento mínimo de control de flujo. De esta forma, los programadores se pueden enfocar en la tarea de la paralelización de los algoritmos en lugar de invertir demasiado tiempo en su implementación.
- Brindar soporte para computación heterogénea en donde las aplicaciones se puedan ejecutar tanto en la CPU como en la GPU. Las partes seriales de las aplicaciones son ejecutadas en la CPU y las partes paralelas son cargadas para su ejecución en la GPU. La CPU y GPU se consideran dispositivos separados con sus propios espacios de memoria. Este esquema permite la realización de cómputo simultáneo en la CPU y GPU sin conflicto por los recursos de memoria.

La programación en CUDA involucra la ejecución de código en un sistema con dos plataformas de cómputo diferentes: la CPU o *host*, y las GPUs o *devices* alojadas en el

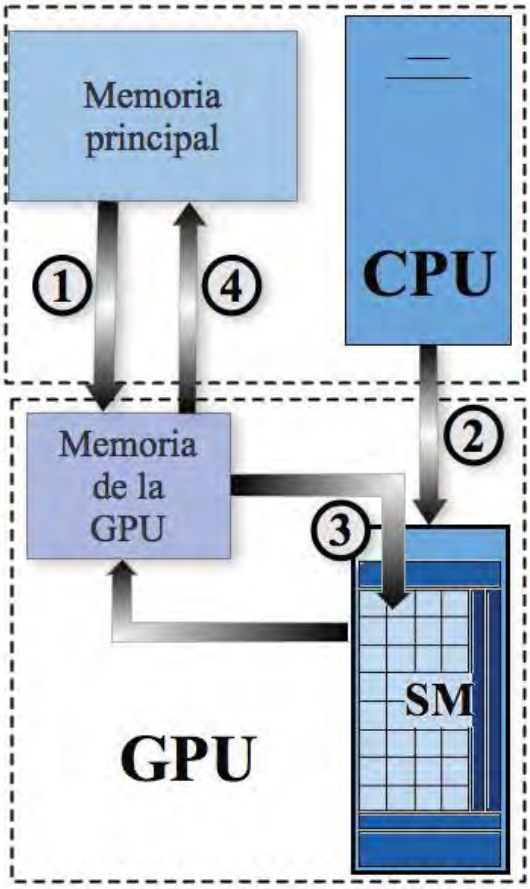



Figura 2.7: Flujo de ejecución de una aplicación basada en GPU.

mismo gabinete que resguarda la CPU. Ambos sistemas de cómputo se basan arquitecturas distintas, por lo que es importante entender las características de cada uno de estos para incrementar el rendimiento de las aplicaciones basadas en una GPU.

Debido a las diferencias en las características del *host* y el *device*, es importante dividir las aplicaciones de tal manera que cada sistema de hardware realice el trabajo que se ajuste mejor a sus características [NVIDIA10a]. Por un lado, la CPU está optimizada para realizar la ejecución de las partes secuenciales de un programa, mientras que la GPU es ideal para los cálculos que se pueden ejecutar simultáneamente sobre una gran cantidad de conjuntos de datos. La Figura 2.8 muestra la secuencia que se debe seguir para ejecutar

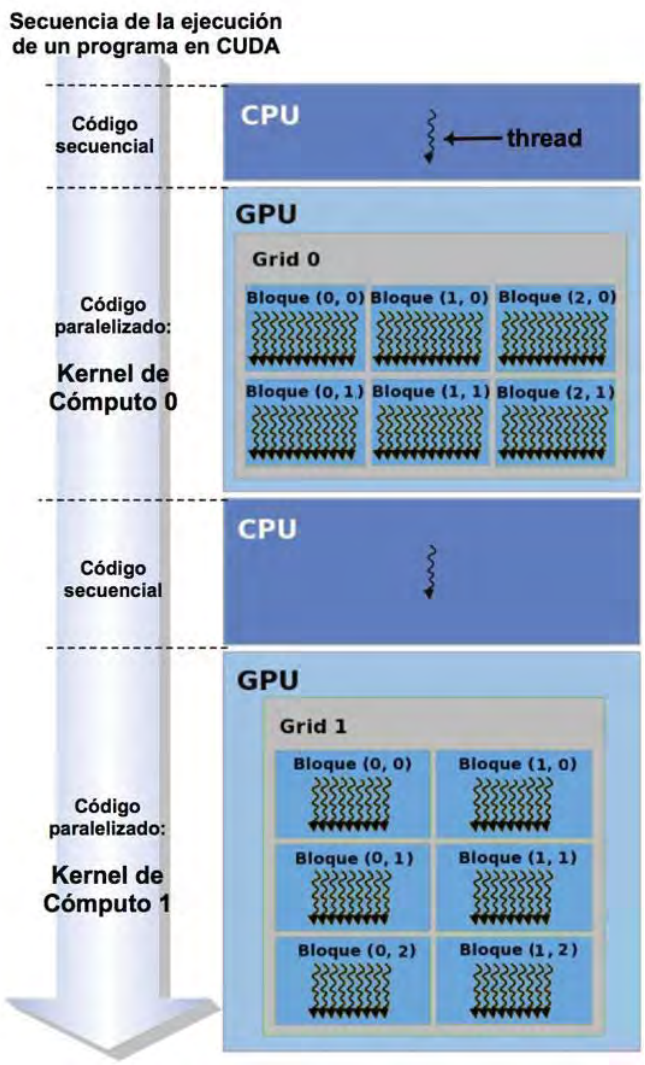



Figura 2.8: Ejecución de un programa típico en CUDA.

un programa en CUDA. Como se puede observar en la Figura 2.8, el código secuencial es ejecutado en el *host* por un único hilo, mientras que el código paralelo se ejecuta en el *device* por una gran cantidad de hilos simultáneamente.

Existen algunas consideraciones que se deben tomar en cuenta para determinar las partes de una aplicación que pueden ser ejecutadas en la GPU. Estas consideraciones son:

- Debe existir cierta coherencia en los accesos a memoria por parte del código en el *device*. Los datos que no puedan ser establecidos o que no sean del tamaño adecuado para ser almacenados en los espacios de memoria de texturas, no verán un beneficio significativo en su desempeño cuando se utilice un esquema basado en GPU.

- Para usar CUDA, los datos deben ser transferidos desde el *host* al *device* a través del bus PCIe. Estas transferencias son costosas desde el punto de vista de tiempos de transferencia, por lo que es recomendable hacer las modificaciones necesarias a la programación del código para minimizarlas. En este sentido es recomendable tomar en cuenta las siguientes consideraciones:

  - La complejidad de las operaciones debe justificar la transferencia de datos hacia y desde el *host*. Transferencias de datos para un uso breve por parte de los *threads* producirán muy poco o nulo beneficio en el rendimiento de la aplicación. Por ejemplo, transferir dos matrices desde el host a la GPU para realizar su suma y después transferir el resultado de nuevo al host no implica mucho beneficio en el desempeño. Asumiendo que las matrices son de orden $N \times N$, hay $N^2$ operaciones (sumas) y $3N^2$ elementos transferidos. Por lo tanto, la relación entre operaciones y elementos transferidos es de 1:3, es decir O(1). En el caso de la multiplicación de las mismas dos matrices, se requiere $N^3$ operaciones (multiplicación-suma), de tal manera que la proporción entre operaciones y elementos transferidos es O(N), en cuyo caso la ventaja en el rendimiento es mayor.

  - Dado que las transferencias de datos entre el *host* y el *device* deben minimizarse, los datos deben permanecer en el *device* el mayor tiempo posible. Es recomendable que en programas que ejecutan múltiples kernels sobre los mismos datos se dejen dichos datos en el *device* entre las llamadas a los kerneles de cómputo, en lugar de transferir los resultados intermedios al *host* y luego enviarlos de nuevo al *device* para cálculos posteriores. Este enfoque debe ser utilizado incluso si alguno de los pasos en una secuencia de cálculos se puede realizar más rápido en el *host*. Un kernel relativamente lento puede mostrar un beneficio si se evita una o más transferencias a través del bus PCIe.

### 2.4.2. Jerarquía de threads

Los *threads* pueden ser organizados e identificados dentro de estructuras unidimensionales, bidimensionales y tridimensionales. Con el objetivo de ilustrar la definición y la ejecución de un kernel de cómputo por un bloque unidimensional de *threads*, en el Listado 2.1 se presenta un código resumido. Este código realiza la suma de dos vectores **a** y **b** de orden $N$ y almacena el resultado dentro del vector **c**. En este ejemplo, cada *thread* lleva a cabo la suma de las componentes correspondientes de los dos vectores **a** y **b** almacenando la componente resultante correspondiente en **c**. Por otra parte, es posible ejemplificar la ejecución de un kernel por un bloque bidimensional de *threads* a través del código resumido presentado en el Listado 2.2, el cual realiza la suma de dos matrices **A** y **B** de orden $N \times N$ almacenando el resultado en la matriz **C**.

El número de *threads* que ejecutan dicho kernel se especifica empleando una nueva sintaxis de configuración de ejecución definida como,

```
Kernel_Name<<<Grid_Dim, Block_Dim>>>(arg1, arg2, ..., argN);
```

donde `Grid_Dim` y `Block_Dim` son variables del tipo `dim3`, un tipo de dato integrado a través del cual se puede definir las dimensiones del *grid* de bloques de *threads* y las dimensiones de los bloques de `threads` respectivamente. Sin embargo, estas variables pueden ser también valores enteros cuando se trata de arreglos unidimensionales de bloques y/o *threads*.

Es necesario considerar, en la ejecución de un kernel de cómputo existe un límite en el número de *threads* que pueden integrar un bloque (hasta 1024, en las GPUs más recientes). Por lo tanto, si los vectores o matrices tuvieran un número de elementos mayor al máximo número de *threads* por bloque, esta operación no podría ser realizada por un sólo bloque, siendo necesario crear más de un bloque *threads* para poder ejecutar la operación correctamente. Lo anterior implica considerar adicionalmente los índices de los bloques de *threads* para poder calcular la posición de los elementos en el vector o matriz resultante que va a calcular determinado *thread*.

```
// Definición del Kernel de Cómputo
__global__ void VecAdd(float *a, float *b, float *c){
        int i = threadIdx.x;
        c[i] = a[i] + b[i];
}

// Función principal
int main (int argc, char **argv){
        ⋮
        // Ejecución del Kernel por un único bloque de threads de dimensión N
        VecAdd <<<1, N>>>(a, b, c); //Recibe los vectores a, b y c
        ⋮
}
```

Listado 2.1: Suma de dos vectores de orden $N$ en un kernel de cómputo.

```
// Definición de Kernel de Cómputo
__global__ void MatAdd(float **A, float **B, float **C){
        int i = threadIdx.x;
        int j = threadIdx.y;

        C[i][j] = A[i][j] + B[i][j];
}

// Función principal
int main (int argc, char **argv){
        ⋮
        dim3 GridDim(1);
        dim3 BlockDim(N, N);
        // Ejecución del Kernel por un único bloque de threads de dimensión N×N
        MatAdd<<<GridDim, BlockDim>>>(A, B, C); //Recibe las matrices A, B y C
        ⋮
}
```

Listado 2.2: Suma de dos matrices de orden $N \times N$ en un kernel de cómputo.

En CUDA los índices y los identificadores de los *threads* se relacionan entre sí de una manera directa. Para un bloque unidimensional (únicamente en al dimensión $x$), el índice y el identificador son los mismos. Para un bloque bidimensional de orden $D_x \times D_y$ el identificador del *thread* con índices $[x, y]$ se obtiene realizando la operación $(x + yD_x)$. Por su parte, un *thread* dentro de un bloque tridimensional de orden $D_x \times D_y \times D_z$ con índices $[x, y, z]$ obtendrá su identificador mediante la operación $(x+yD_x+zD_xD_y)$. Analógicamente, cada bloque de *threads* puede ser identificado a través de sus índices dentro del *grid* y las dimensiones propias del *grid*.

En el Apéndice A se reporta el programa completo para realizar la ejecución del código presentado en los listados 2.1 y 2.2

## 2.5. Jerarquía de memoria

Dentro del entorno de programación de CUDA, los *threads* son capaces de acceder a múltiples espacios de memoria durante su periodo de vida. En la Figura 2.9(a) se describe el acceso a la memoria local por parte de cada *thread*. Los bloques de *threads* tienen un espacio de memoria compartida a la cual pueden acceder todos los *threads* dentro del bloque (ver Fig. 2.9(b)). Así mismo, todos los *threads* tienen acceso a la memoria global de la GPU durante la ejecución de un kernel (ver Fig. 2.9(c)). Estos espacios de memoria tienen un periodo de vida igual al tiempo de ejecución del kernel que ha sido invocado por los *threads*.

Existen adicionalmente dos espacios de memoria a los cuales pueden tener acceso todos los *threads*, los cuales han sido optimizados para diferentes usos: el espacio de memoria constante y el espacio de memoria asignado para texturas. La memoria constante es un tipo de memoria de "sólo lectura" por parte de los *threads*. Se trata de la memoria caché de la GPU por lo que es el espacio de acceso más rápido. Tanto la memoria constante como la memoria compartida son recursos escasos, 16 KB y 64 KB, respectivamente, por lo que su uso debe ser limitado de acuerdo a las necesidades particulares de cada aplicaión.

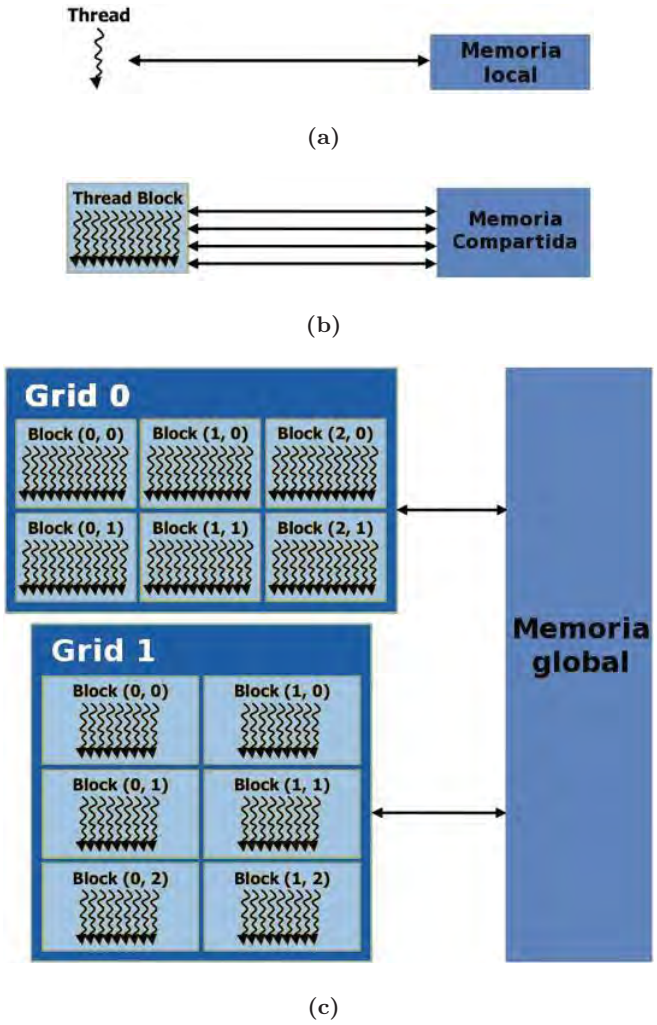



Figura 2.9: Tipos de memoria en la GPU (a).- memoria local, (b).- memoria compartida y (c).- memoria global.

### 2.5.1. Memoria compartida

La memoria compartida es un espacio de memoria de acceso rápido dentro de las unidades SM por parte de los *threads* que integran un bloque, tal como se muestra en la Figura 2.9(b). Dado que un bloque de *threads* se ejecuta en un solo SM, este espacio de memoria permite la cooperación entre *threads* de un mismo bloque para agilizar la reutilización de datos empleados por el bloque. Este enfoque reduce en gran medida el

tráfico de datos desde la memoria global del dispositivo. Este tipo de memoria puede ser reservada de manera estática durante la ejecución de un kernel empleando el calificador `__shared__`. Por ejemplo,

```
__shared__ int array[5] = {1,2,3,4,5}
```

Con el objeto de agilizar su acceso, la memoria compartida es dividida en módulos de memoria de tamaño identico llamados bancos, a los cuales se puede acceder simultáneamente sin crear conflictos. La memoria compartida es mucho más rápida de acceder que la memoria global, de tal manera que se recomienda aprovechar cualquier oportunidad de reemplazar los accesos a la memoria global por accesos a la memoria compartida.

### 2.5.2. Memoria global

Como se mencionó con anterioridad, el modelo de programación de CUDA asume un sistema de cómputo compuesto por un *host* (CPU) y un *device* (GPU); cada uno de estos con su propio espacio de memoria. Así mismo, CUDA proporciona funciones para reservar, liberar, copiar y transferir datos entre la memoria del *host* y la del *device*. La memoria global del *device* puede ser asignada bajo un esquema de memoria lineal, en el cual se ve a la memoria como un arreglo unidimensional de localidades de un Byte y a la cual se puede hacer referencia a través de apuntadores. La memoria lineal puede ser asignada empleando la función `cudaMalloc()`, cuya sintaxis está definida como sigue,

```
cudaMalloc(void** devPtr, size_t n)
```

esta función reservará *n* Bytes de memoria en el *device* y regresa en **devPtr* un apuntador a la memoria reservada. La memoria asignada está preparada, por conveniencia, para soportar cualquier tipo de variable.

Todo espacio de memoria reservado en el *device* debe ser liberado empleando la función `cudaFree()`. Esta función tiene la sintaxis,

```
cudaFree(void *devPtr)
```

a través de la cual se libera el espacio de memoria apuntado por *devPtr*.

Las transferencias de datos entre el *host* y el *device* se realizan usando la función `cudaMemcpy()` descrita a continuación,

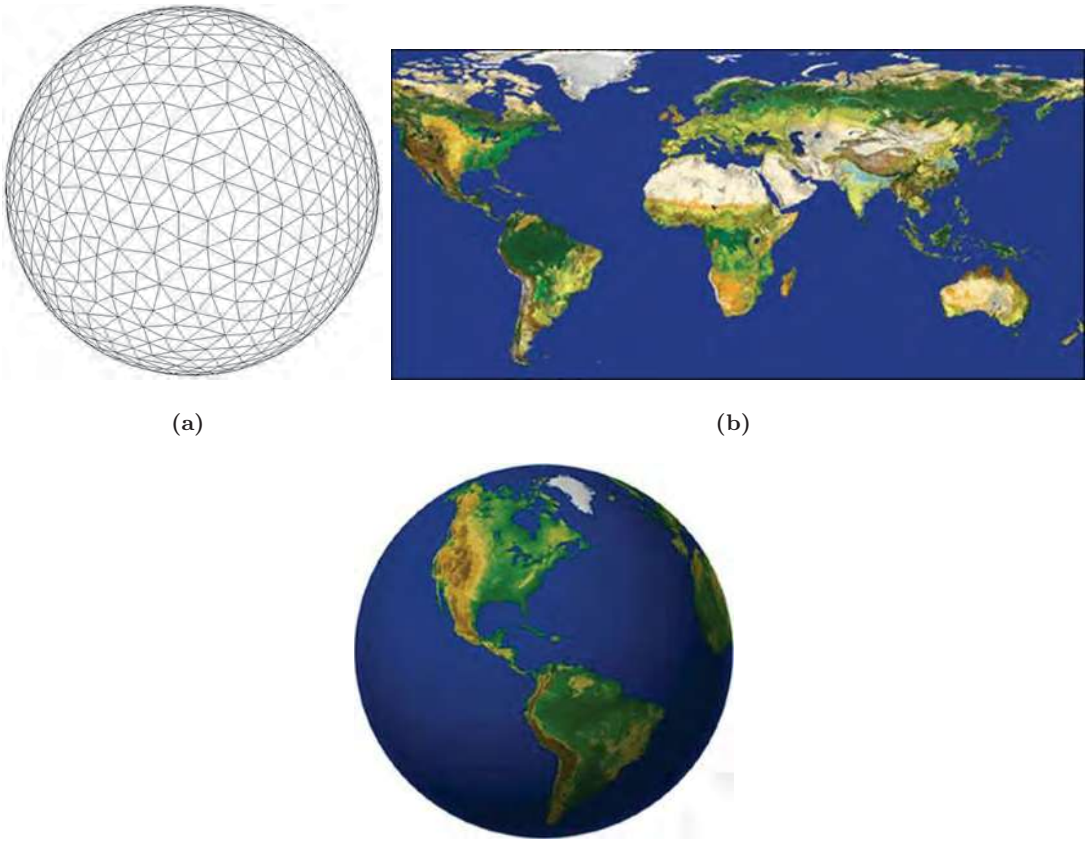
```
cudaMemcpy(void *dst, const void *src, size_t count,
                    enum cudaMemcpyKind kind)
```

Esta función se encarga de copiar *count* Bytes desde el espacio de memoria *src* hacia el espacio de memoria *dst*. El parámetro *kind* especifica el tipo y el sentido en que se están copiando los datos. Las opciones disponibles para el parámetro *kind* son:

- `cudaMemcpyHostToHost`
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`

## 2.6. Texturas

En una GPU la disposición nativa de los datos es bidimensional. Los datos son almacenados en estos arreglos bidimensionales llamados texturas. Este tipo de arreglo es esencial debido al hecho de que las GPUs están originalmente diseñadas para desplegar geometría bidimensional [Gopal10]. Una textura es en esencia una matriz que almacena en sus elementos datos referentes a la intensidad de los tres canales del esquema de color RGB y el canal alpha o A. Estos datos se utilizan para cubrir la superficie de un objeto gráfico en dos o tres dimensiones. La Figura 2.10 muestra la aplicación de texturas a un elemento tridimensional. Sea por ejemplo una esfera sin textura como la mostrada en la Figura 2.10(a) y una imagen de textura (ver Figura 2.10(b)), entonces la esfera con textura resultante se muestra en la Figura 2.10(c). Tal como se puede apreciar en esta imagen, las texturas se emplean principalmente para agregar realismo a los gráficos. El realismo al desplegar los gráficos en una pantalla es la principal motivación en la creación de aplicaciones 3D interactivas cada vez más complejas.



**(a)** **(b)**

**(c)**

Figura 2.10: Aplicación de texturas a un elemento tridimensional (a).- Esfera sin textura, (b) Imagen de textura y (c).- Esfera con textura.

Los espacios de memoria para texturas residen en el *device* almacenados en caché. Esto quiere decir que la extracción de un dato almacenado en textura tiene el costo de una lectura a la memoria del *device* sólo cuando ocurra un error de caché. De otra manera, el costo será únicamente una lectura del caché de textura. En [NVIDIA10b] se sugiere que los datos de una aplicación que no puedan ser establecidos o que no sean del tamaño adecuado para ser almacenados en los espacios de memoria de texturas, no verán un beneficio significativo en su desempeño cuando se utilice un esquema basado en GPU. De acuerdo a [NVIDIA10d], dependiendo de la capacidad de cómputo de la GPU, es posible crear texturas en 2D de hasta 65536 texeles de ancho y 65535 texeles de alto.

Para la implementación del algoritmo de Evolución Diferencial basado en GPU relacionado a esta tesis, el recurso de cómputo asociado a los espacios de memoria de texturas es aprovechado con la finalidad de mejorar su desempeño. La manera en que se explotan estos espacios de memoria de rápido acceso es generando una textura bidimensional a partir de las poblaciones (principal y de prueba). En CUDA, las texturas se generan al vincularse a una región lineal de memoria a través de un `cudaArray`, por lo que para generar una textura a partir de las poblaciones de individuos en el algoritmo de ED, es necesario copiar previamente estos datos en un `cudaArray`. La textura tendrá que almacena las poblaciones tendrá un ancho igual a a longitud de los vectores o individuos, mientras que de alto tendrá el tamaño de la población. Este hecho limita el tamaño de la población a 65535, para tamaños de población más grandes no sería posible aprovechar los espacios de memoria de texturas y el beneficio sería menor.

Como un dato adicional, los `cudaArrays` están compuestos de elementos, cada uno de los cuales pueden tener hasta cuatro componentes de valores enteros con o sin signo de 8, 16 y 32 bits, o flotantes de 16 y 32 bits. Estos elementos con hasta cuatro componentes representarían en un contexto gráfico la intensidad de los colores rojo, verde, azul y transparencia (los canales RGBA). Por lo tanto, una operación simple de extracción de datos desde una textura puede regresar hasta cuatro valores encapsulados, dependiendo de cuantos componentes existan en los elementos del `cudaArray` al que se vinculó la textura. Este hecho establece la posibilidad de tener hasta cuatro poblaciones al mismo tiempo, lo cual permitiría la ejecución simultánea de 4 algoritmos de ED, bajo un esquema de paralelismo similar al empleado en [Gopal10].

### 2.6.1. Declaración de una referencia a textura

Algunos atributos de las referencias a texturas se especifican en la declaración de la misma. Una referencia a textura es declarada como una variable del tipo `texture`, a través de la siguiente sintaxis,

```
texture<Type, Dim, ReadMode> texRef;
```

en donde,

- `Type` especifica el tipo de datos que es regresado cuando se hace una extracción desde la textura. `Type` está restringido a enteros y flotantes de precisión sencilla, además del tipo de dato float4 el cual es capaz de encapsular cuatro números flotantes de presición sencilla.

- `Dim` especifica la dimensionalidad de la referencia a textura, y es igual a 1, 2 o 3; en el último caso se le debe indicar a CUDA la declaración de texturas tridimensionales. `Dim` es un argumento opcional cuyo valor por defecto es 1.

- `ReadMode` es un parámetro que puede tomar `cudaReadModeNormalizedFloat` o `cudaReadModeElementType`; si es `cudaReadModeNormalizedFloat` y Type es un entero, todo el rango del tipo entero es mapeado al rango [0.0, 1.0] para enteros sin signo y [-1.0, 1.0] para enteros con signo, lo que significa que el valor que se regresa en realidad es un flotante. Si se establece `cudaReadModeElementType`, el cual es el valor por defecto, esta conversión no es efectuada.

Una referencia a textura se puede declarar únicamente como una variable global estática y no puede ser pasada como argumento a una función. Además, a diferencia de la memoria de una CPU, no es posible acceder de manera aleatoria a estos espacios de memoria en un punto determinado durante el cómputo en una aplicación. Una textura puede ser sólo de lectura o sólo de escritura.

### 2.6.2. Sistema de coordenadas de texturas

La memoria de texturas es leída desde los kérneles empleando funciones denominadas "capturas de textura", en las cuales se extraen los datos almacenados. Cada uno de los elementos en las texturas se denomina texel y el acceso a los texeles se realiza a través de índices denominados "coordenadas de textura". Las coordenadas de textura son equivalentes a los índices de un arreglo en la CPU, con la diferencia de que estas coordenadas hacen referencia a elementos que pueden encapsular hasta cuatro valores, mientras que el índice de un arreglo se refiere únicamente a una entidad.

Las texturas poseen atributos que definen el tipo de datos que se maneja dentro de ellas, el tipo de procesamiento que debe realizarse con sus datos y la manera en como

deben ser interpretadas las coordenadas de textura. Por defecto las texturas son referenciadas empleando coordenadas en valores de punto flotante. Si se declara que las coordenadas de textura sean normalizadas, es decir, `textureReference.normalized = true`, las coordenadas toman valores en el rango [0.0, 1.0) en cada dimensión. Las coordenadas de textura normalizadas se ajustan naturalmente a los requerimientos de algunas aplicaciones en las cuales es preferible que dichas coordenadas sean independientes del tamaño de la textura [NVIDIA10d]. Por otra parte, si este atributo se declara como no-normalizado, es decir, `textureReference.normalized = false`, el sistema de coordenadas toma valores en el rango [0.0, n). Un sistema de coordenadas no-normalizado se emplea en aplicaciones en las cuales las coordenadas de textura dependan del tamaño de la textura, en el cual se denota explícitamente una equivalencia entre las coordenadas de una textura y los índices de un arreglo en la CPU.

Para extraer los datos almacenados en los texeles es necesario que las coordenadas de textura apunten al centro del texel [Gopal10]. En un sistema de coordenadas de textura no-normalizado cada texel representa una unidad. Por lo tanto, para extraer los datos almacenados en estos elementos de textura se debe hacer un desplazamiento de 0.5 en los valores de las coordenadas. La Figura 2.11 muestra esquemáticamente el sistema de coordenadas empleado en las referencias a textura no-normalizadas. Como se puede observar, para hacer referencia al primer texel en cada dimensión, por ejemplo, es necesario emplear las coordenadas (0.5, 0.5) en lugar de (0, 0).

Dos atributos más que se deben especificar para poder hacer referencia a texturas son el modo de direccionamiento y de interpretación de los datos (filtrado de textura). El atributo referente al modo de filtrado `filterMode` de textura especifica el modo en que son regresados los valores cuando se hace una extracción de textura cuando las coordenadas no indican el centro de un texel. Este atributo puede ser `cudaFilterModePoint` o `cudaFilterModeLinear`; si se especifica como `cudaFilterModePoint`, el valor regresado será el dato almacenado en el texel cuyas coordenadas se encuentren más cercanas; si este atributo es activado como `cudaFilterModeLinear` el valor regresado será la interpolación lineal de los cuatro vecinos más cercanos. `cudaFilterModeLinear` sólo se puede emplear para texturas declaradas para almacenar valores de punto flotante.

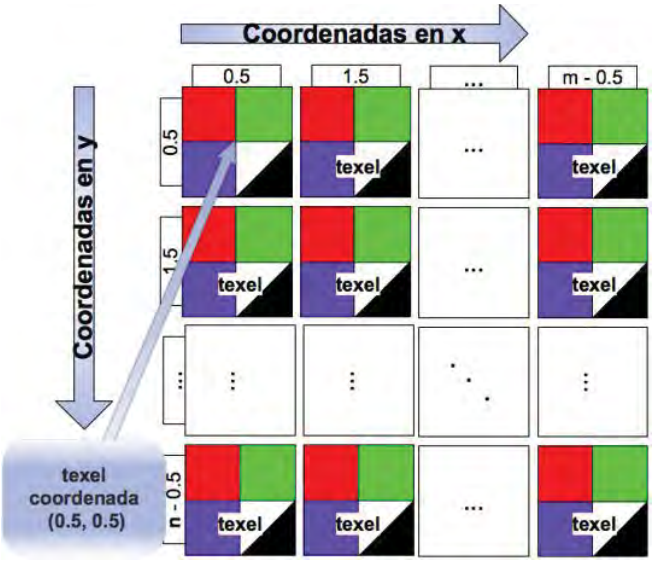



Figura 2.11: Sistema de coordenadas en una textura no normalizada de $m \times n$.

El modo de direccionamiento `addressMode` define lo que ocurre cuando una referencia se sale del rango de las coordenadas de la textura. Por defecto, cuando se usan coordenadas no-normalizadas, se usa un modo de direccionamiento `cudaAddressModeClamp`, en donde valores por debajo de 0.5 son considerados como 0.5 mientras que valores superiores a $n-0.5$ son fijados en $n-0.5$. Si se especifica un tipo de direccionamiento de envoltura `cudaAddressModeWrap` se considera a la textura como periódica [NVIDIA10d].

Además de los atributos de las texturas mencionados con anterioridad, existe un último e importante atributo que se debe tomar en cuenta y que define el formato en que los datos son almacenados y extraídos empleando texturas. El atributo `cudaChannelFormatDesc` es del siguiente tipo:

```
struct cudaChannelFormatDesc {
  int x, y, z, w;
  enum cudaChannelFormatKind f;
}
```

donde x, y, z y w son iguales al número de bits que sería empleado para representar la intensidad de color en cada uno de los cuatro canales RGBA y que en este caso representa el número de bits para cada componente del valor extraído de la textura. El parámetro f puede ser cudaChannelFormatKindSigned o cudaChannelFormatKindUnsigned si se trata de enteros con signo y sin signo, respectivamente. Por su parte, se usa para valores de punto flotante la opción cudaChannelFormatKindFloat. A continuación se describe la sintaxis para establecer este atributo dentro del entorno de programación de CUDA,

```
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(x,
                                                    y, z, w, f);
```

### 2.6.3. Ligado de texturas

Antes de que un kernel pueda hacer referencia a una textura para lectura de datos, es necesario escribir los datos en dicha textura. A esta operación se le llama ligado de textura y se lleva a cabo vinculándola a un cudaArray a través de la función cudaBindTextureToArray. El código presentado en el Listado 2.3 muestra el proceso completo para realizar el ligado de una textura a un cudaArray.

Una vez que se tengan los datos en la memoria de la textura estos pueden ser extraídos desde un kernel o función declarada en el *device* a través de la siguiente sitáxis,

```
value = tex2D(texRef, x_coord, y_coord);
```

Después de haber hecho referencia o uso de dicha textura es necesario eliminar la vinculación entre el cudaArray y la textura en cuestión, lo cual se hace a través de la función cudaUbindTexture. Esta instrucción es equivalente a liberar la memoria, pero con la diferencia que para cálculos posteriores es posible volver a ligar esta textura para almacenar en ella datos futuros.

```
// Declaración de una referencia a textura para una textura 2D
texture<float, 2, cudaReadModeElementType>texRef;

int main(int argc, char** argv){
        int width = 128; int height = 2048; // Ancho y alto de la textura
        // Datos en el host que se van a almacenar en una textura
        float *host_data

        // Creamos la descripción de los canales del esquema RGBA
        cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0, 0,
                                                cudaChannelFormatKindFloat);
        // Reservamos memoria en el device para un CUDA array
        cudaArray *cuArray;
        cudaMallocArray(&cuArray, &channelDesc, width, height);

        // Copiamos al CUDA array los datos ubicados en la direccion host_data
        cudaMemcpyToArray(cuArray, 0, 0, *host_data,
                          size, cudaMemcpyHostToDevice);

        // Establecemos los atributos de la textura
        texRef.addressMode[0] = cudaAddressModeWrap;
        texRef.addressMode[1] = cudaAddressModeWrap;
        texRef.filterMode = cudaFilterModeLinear;
        texRef.normalized = false;
        ⋮
        // Ligamos la textura al CUDA array
        cudaBindTextureToArray(texRef, cuArray, channelDesc);

        /* Ejecucion del kernel */
        ⋮
        // Deslindamos la textura del CUDA array
        cudaUnbindTexture(texRef);
}
```

Listado 2.3: Proceso de ligado de textura.

## 2.7. Sumario

Debido a la creciente demanda de los gráficos 3D de alta definición en aplicaciones interactivas, las GPUs han evolucionado en arquitecturas de procesamiento masivamente paralelas. La arquitectura de las GPUs modernas es muy eficiente, con la capacidad para realizar cómputo de propósito general ejecutando ciertos algoritmos más rápida y eficientemente que una CPU. En el 2006 NVIDIA introduce un entorno de programación integral denominado CUDA, el cual permite aprovechar el poder de cómputo contenido en las GPUs para realizar procesamiento de propósito general. Es necesario considerar que no todos los problemas complejos pueden ser resueltos eficientemente en una GPU, una GPU es adecuada para resolver problemas complejos que puedan ser expresados principalmente como el cálculo de datos en paralelo. CUDA tiene una API integrada por un compilador y herramientas que permiten declarar funciones, con una sintaxis muy parecida a la del lenguaje de programación C, para ser ejecutadas en procesadores GPU. Además, permite a las aplicaciones basadas en la CPU acceder directamente a los recursos de una GPU, sin las limitaciones de la utilización de una API para gráficos. Lo anterior hace que los programas de C puedan tomar ventaja de la capacidad de una GPU sin dejar de hacer uso de la CPU cuando sea apropiado.

Capítulo 3

# Evolución Diferencial Paralela en la GPU

La mayoría de los AEs son modelos reducidos de fenómenos manifestados en la naturaleza. Un algoritmo genético por ejemplo, es una metáfora de la evolución Darwiniana; PSO representa una analogía del comportamiento de los bancos de peces y parvadas de aves [Parsopoulos02]; ACO obtuvo sus baese de las colonias de hormigas [Dorigo97]; los Sistemas Inmunes Artificiales fueron creados a partir de la observaión del funcionamiento de los sistemas inmunológicos en los seres vivos [Kephart94]. Incluso se han obtenido modelos funcionales a partir de fenómenos físicos regenerativos basados en funciones de temperatura como lo es SA [Kircpatrick83]. Todos estos modelos representan procesos con un éxito notable de algún tipo de optimización en la naturaleza. En la actualidad, los algoritmos evolutivos se han establecido como modernos procedimientos heurísticos de búsqueda que han logrado trascender debido a su capacidad de sobrellevar procesos de optimización correspondientes tanto al modelo continuo como al discreto.

## 3.1. Optimización

Optimización es el proceso o metodología intentar variantes sobre un concepto inicial y utilizar información asociada con el fin de mejorar dicho concepto, haciéndolo tan

funcional o efectivo como sea posible. Matemáticamente es el proceso minimizar o maximizar una función objetivo, esto implica la selección sistemática del mejor elemento de un conjunto de alternativas posibles con la finalidad de obtener la mejor solución. Cuando se emplea el término "la mejor solución" implica que existe más de una solución y que éstas no son de la misma calidad [Haupt04]. Esta definición es relativa al problema en cuestión, su método de solución y la tolerancia permitida. Algunos problemas tienen soluciones exactas como la solución a una ecuacion diferencial de primer orden. En la vida diaria se presentan varias opciones para llevar a cabo un proceso de optimización. A qué hora se debe despertar una persona para maximizar el tiempo de sueño sin llegar tarde al trabajo, cuál es la mejor ruta hacia el trabajo, etc. En la industria, por ejemplo, cuando se diseña algo es posible minimizar su costo de producción o maximizar su atractivo.

Optimización es el proceso de ajustar los parámetros de entrada o las características de un dispositivo, un proceso matemático o un experimento con el objetivo de encontrar el máximo o mínimo resultado. En cualquier proceso de optimización las entradas consisten en variables, el proceso o la función se conoce como función de costo o función objetivo y las salidas representan el costo. En [Haupt04] se definen varios tipos de optimización, las cuales se enlistan a continuación:

1. **Optimización por prueba y error** se refiere al proceso de ajustar las variables que afectan la salida sin conocer mucho acerca del proceso que la genera.

2. **Optimización unidimensional** y **multi-dimensional** si existe una sola variable o si el problema es en más de una dimensión, respectivamente. Cabe señalar que el proceso de optimización se vuelve más complejo al aumentar el número de dimensiones.

3. **Optimización dinámica** y **estática** dependiendo si la salida se encuentra o no en función del tiempo. Encontrar la distancia más corta entre dos puntos dentro de una ciudad es un problema estático. Por otro lado, encontrar la ruta más rápida es un problema dinámico cuya solución depende de la hora del día, el clima, etc.

4. **Optimización discreta** y **continua** si las variables tienen una representación discreta (número finito de valores) o continua (número infinito de valores).

5. **Optimización con restricciones** y **sin restricciones** dependiendo si las variables tienen límites o restricciones, o si simplemente pueden tomar cualquier valor.

Los problemas pueden ser clasificados entonces de acuerdo a la naturaleza de la función objetivo y las restricciones que sus parámetros implican (lineales, no lineales, convexas), el número de variables (grande o pequeño) y si la función es diferenciable o no diferenciable. Posiblemente la distinción más importante es entre aquellos problemas que tienen restricciones en el dominio de sus variables y los que no. Cuando tanto la función objetivo como todas sus restricciones son funciones lineales se tiene un problema de programación lineal. Cuando al menos una de las restricciones o la función objetivo son funciones no lineales, se trata de problemas de programación no lineal. Los problemas de programación no lineal tienden a surgir de manera natural dentro de la ingeniería y ciencias físicas, y son cada vez más ampliamente utilizados en administración y ciencias económicas [Nocedal99].

Para poder llevar a cabo un proceso de optimización, existen algoritmos determinísticos capaces de encontrar una solución local a partir de un punto inicial de búsqueda. Estos algoritmos son muy rápidos y eficientes, pero tienden a estancarse en óptimos locales y no siempre encuentran la mejor solución. Las soluciones u óptimos globales son necesarias o al menos altamente requeridas en la mayoría de las aplicaciones. Sin embargo, la mayoría de las veces es difícil identificarlas e incluso más dificil localizarlas. Un caso especial son las funciones objetivo que denotan una parábola o paraboloide, tal es el caso de una función cuadrática, en cuyo caso no existen soluciones locales, sólo una solución global.

Por otra parte, existen problemas no lineales (con o sin restricciones) para los cuales puede existir un gran número de óptimos locales. Para abordar problemas como éstos, existen métodos estocásticos como los algoritmos evolutivos, los cuales parten de más de un punto inicial y emplean cálculos probabilísticos para encontrar puntos en el espacio de búsqueda. Estos últimos son metodologías más robustas de optimización que tienden a ser más lentos pero más eficientes para encontrar el óptimo global.

### 3.1.1. Optimización analítica

La teoría del cálculo proporciona herramientas suficientes para encontrar el mínimo de muchas funciones de costo de una manera elegante y eficiente. Si se simplifica el proceso a una sola variable, es posible encontrar el óptimo obteniendo la primera derivada de la función de costo, igualar la derivada a cero y resolver para encontrar el valor de la variable. Si la segunda derivada es mayor o igual que cero, el óptimo encontrado es un mínimo. Si la segunda derivada es menor que cero se trata de un máximo.

Una manera de encontrar el óptimo de una función de dos o más variables $f$ es tomar el gradiente de la función e igualarlo a cero, $\nabla f = 0$, con lo cual se tiene un sistema de ecuaciones. Una vez resuelto este sistema de ecuaciones es necesario calcular el Hessiano, la solución encontrada es un mínimo cuando $\nabla^2 f \geq 0$. Desafortunadamente este proceso no garantiza que la solución encontrada sea el mínimo global, esto debido a que el sistema de ecuaciones puede tener más de una solución. En el caso de que existiera más de una solución, es necesario encontrar el conjunto completo de soluciones, evaluar todas ellas y entonces identificar el mínimo global. Enfoques como estos resultan ser matemáticamente elegantes en comparación con aquellos de busqueda exhaustiva o que implican búsquedas aleatorias, sin embargo, las funciones de costo necesariamente tienen que ser continuas y diferenciables [Haupt04].

En [Nocedal99] se destacan dos estrategias de optimización sin restricciones: los métodos de búsqueda lineal (*line search method*) y los métodos de búsqueda en región de confianza (*trust region*), los cuales esencialmente comienzan en un punto $x_0$ y proceden a generar una secuencia de iteraciones $\{x_k\} \ \ \forall\ k = 0, 1, 2, 3, ..., \infty$ la cual termina cuando no es posible obtener más beneficio. En general, tanto los métodos de búsqueda lineal como los métodos de búsqueda en regiones de confianza utilizan un modelo cuadrático de la función objetivo, aunque de diferentes maneras. Los métodos de búsqueda lineal emplean el modelo cuadrático para generar una dirección de búsqueda y después enfocan sus esfuerzos en encontrar una longitud de paso adecuada en esta dirección. Por otro lado, los métodos de búsqueda en regiones confiables definen un modelo que represente adecuadamente la función objetivo alrededor de la solución actual, entonces eligen la dirección y una longitud

un paso que se aproxime a la minimización del modelo en esta región.

En las estratégias de búsqueda lineal los algoritmos calculan una dirección $p_k$ y buscan en esa dirección un valor menor en el rango de la función. La distancia que se recorre en $p_k$ puede ser encontrada resolviendo el siguiente problema de minimización en una dimensión para encontrar una longitud de paso $\alpha$ [Nocedal99]:

$$\min_{\alpha} \; f(x_k + \alpha p_k), \qquad\qquad \alpha > 0 \tag{3.1}$$

A través de la solución de (3.1) es posible obtener el máximo beneficio en la dirección $p_k$. Sin embargo, una minimización exacta es costosa e innecesaria [Nocedal99]. En lugar de esto, el algoritmo genera un número limitado de longitudes de paso de prueba hasta encontrar uno que se aproxime al mínimo. En el nuevo punto encontrado, se calcula una nueva dirección de búsqueda, una longitud de paso y el proceso es repetido.

Cada iteración de un método de búsqueda lineal calcula una dirección de búsqueda $p_k$ y después decide que tan lejos avanzar a lo largo de esa dirección. Esta iteración está dada por

$$x_{k+1} = x_k + \alpha_k p_k \tag{3.2}$$

en donde el escalar positivo $\alpha_k$ se denomina longitud de paso. El éxito de un método de búsqueda lineal depende en una elección efectiva tanto de la dirección de búsqueda $p_k$ como de la longitud de paso $\alpha_k$.

En un proceso de minimización se requiere que $p_k$ sea una dirección de descenso para el que $p_k^T \nabla f_k < 0$. Para cumplir con esta desigualdad, es necesario que alguno de los dos factores sea negativo. Esta propiedad garantiza desplazarse en cualquier dirección excepto en aquella en la que la función crece más escarpadamente. De esta forma, la función $f$ puede ser reducida a lo largo de esta dirección. La dirección de búsqueda siempre tiene la forma

$$p_k = -B_k^{-1} \nabla f_k \tag{3.3}$$

donde $B_k$ es una matriz simétrica y no singular. Para el método de máximo descenso de gradiente $B_k$ es simplemente la matriz identidad $I$, mientras que en el caso del método de Newton $B_k$ es el Hessiano $\nabla^2 f(x_k)$. En los métodos denominados quasi-Newton, $B_k$ es

una aproximación a la matriz Hessiana la cual es actualizada en cada iteración a través de una fórmula de bajo rango [Nocedal99]. Para los métodos de búsqueda lineal se puede generalizar la definición de la dirección de búsqueda como:

$$p_k \propto -\mathrm{U}\nabla f_k \tag{3.4}$$

En el caso de la estrategia de búsqueda en región confiable, esta reúne información acerca de $f$ para construir un modelo de una función $m_k$ cuyo comportamiento cerca del punto actual $x_k$ sea similar al de la función objetivo $f$. Dado que para puntos lejanos a $x_k$ el modelo puede no ser una buena aproximación, la búsqueda debe restringirse en alguna region cercana a $x_k$.

Es común que la región de confianza tenga forma circular con un radio $\delta > 0$. Sin embargo, se suelen utilizar también regiones rectangulares y elípticas [Nocedal99]. El tamaño de la región confiable resulta crítica en la efectividad de cada paso que se da en la búsqueda. Si la región es pequeña, el algoritmo pierde la oportunidad de realizar pasos sustanciales que lo pudieran llevar cerca del mínimo de la función objetivo $f$. Si esta región es muy grande, la solución candidata puede no producir un decremento suficiente en $f$ o incluso no producir decremento alguno. Esto significa que el modelo $m_k$ es una representación inadecuada de la función $f$ en la región actual, de tal manera que es necesario reducir la región y calcular de nuevo la dirección de paso. El modelo $m_k$ en (**??**) está definido comúnmente como una función cuadrática de la forma

$$m_k(x_k + p) = f_k + p^T \nabla f_k + \frac{1}{2} p^T B_k p \tag{3.5}$$

donde $f_k$ y $\nabla f_k$ son la función original y su gradiente en el punto $x_k$, mientras que $B_k$ es una matriz la cual puede ser el hessiano $\nabla^2 f_k$ o alguna aproximación al mismo.

Estos métodos eligen una dirección y una longitud de paso simultaneamente. Para obtener cada paso $p$, se busca la solución al subproblema

$$\min_p \; m_k(p) = f_k + \nabla f_k^T p + \frac{1}{2} p^T B_k p \tag{3.6}$$

donde $|p| \leq \delta$, es decir que $x_k + p$ reside dentro de la región de confianza. Si un paso no es aceptable, se reduce el tamaño de la región, lo cual implica un cambio también en la dirección del paso.

## 3.2. Algoritmos evolutivos

Un algoritmo evolutivo parte de los principios originales de los algoritmos genéticos, es decir, son métodos estocástico de búsqueda que basan su funcionamiento en la selección y la variación de un conjunto de soluciones denominada población. Cada individuo dentro de esta población representa entonces una solución candidata para un problema de optimización y los AEs se encargan de aplicar operadores a estos individuos con el objeto de generar nuevas soluciones a partir de ellos o reubicarlos en mejores lugares en el espacio de búsqueda de la solución óptima. Los operadores que actúan sobre estas poblaciones han sido diseñados como modelos simples aspectos de la naturaleza. El principal ejemplo es el algoritmo genético, el cual es un modelo simple de la evolución biológica a través de mutaciones y recombinaciones genéticas. Las soluciones propuestas por el algoritmo se encuentran codificadas y cada uno de los elementos en el código producido almacena información acerca de los parámetros del problema de optimización que se intenta resolver, del mismo modo que el código genético o genotipo de un individuo almacena información acerca de las características que este puede expresar en su entorno. La expresión del genotipo en adición de la influencia del entorno en el que se desenvuelve se conoce como fenotipo.

Los métodos determinísticos de optimización han sido calibrados para encontrar rápidamente la solución de una función analítica y convexa con pocas variables. Para estos casos los métodos determinísticos superan el rendimiento de un algortitmo genético, encontrando rápidamente el mínimo mientras un AE se encontraría realizando las primeras evaluaciones en la primera iteración. De cualquier manera, la principal desventaja de los enfoques determinísticos de optimización es que no se ajustan para la solución de la mayoría de los problemas en el mundo real, comúnmente encuentran el óptimo incorrecto y no funcionan correctamente con variables discretas [Haupt04]. En este caso, los métodos determinísticos que suelen ser más rápidos que un AG no resultan adecuados.

Por otra parte, algunas de las ventajas de un AE son las siguientes:

- Capacidad para optimizar empleando variables continuas y discretas.
- No requiere información del gradiente de la función.

- Realiza búsquedas simultáneas en diferentes áreas del espacio.
- Se ajusta muy bien a la ejecución en arquitecturas de cómputo paralelas.
- Es menos susceptible de estancarse en óptimos locales.
- Puede proveer una lista de soluciones en lugar de una sola.

Estas ventajas se pueden encontrar en la mayoría de los algoritmos evolutivos. Por esta razón, este conjunto de procedimientos heurísticos de optimización es capaz de producir buenos resultados cuando los métodos tradicionales de optimización no resultan adecuados.

En la actualidad existe una creciente variedad de AEs. Las diferencias entre estos diversos métodos están asociadas principalmente a la estratégia empleada en la búsqueda de la mejor solución y a la estrategia de reemplazo de los individuos para dar paso a la siguiente generación de soluciones. El esquema de reemplazo de los individuos puede tomar tres modelos [Haupt04],

1. El modelo generacional: la población entera es reemplazada por sus descendientes en cada generación.
2. El modelo de estado estable: sólo algunos individuos reemplazan a los padres menos aptos en cada generación.
3. Los modelos en los cuales sólo un determinado porcentaje de la población es reemplazada por la descendencia.

Tanto la estratégia de búsqueda como la de reemplazo de individuos definirán el desempeño del algoritmo, por esta razón es necesario prestar especial atención en los esquemas de selección, recombinación y mutación empleados.

### 3.2.1. Evaluacion

Cada solución generada por un AE, denotada por $\mathbf{x}_i$, es evaluada y le es asignado un valor de aptitud, esto es,

$$aptitud = f(\mathbf{x}_i) = f(x_{i1}, x_{i2}, ..., x_{iD}) \tag{3.7}$$

en donde $f$ representa la función de aptitud del AE. De la misma manera que un costo, la aptitud de un individuo denota que tan buenas son las soluciones obtenidas por el AE. La función de aptitud en un AE se encuentra estrechamente relacionada a la función objetivo del problema de optimización que se está abordando. En problemas de optimización con parámetros reales, tal como las funciones continuas, es común que la función de aptitud y la función de costo sean la misma, incluso la morfología de la solución real es la misma que la de su cromosoma.

### 3.2.2. Recombinación

La recombinación o cruza es una de las operaciones evolutivas implicadas en la producción de nuevos individuos. Esta operación es llevada a cabo en el nivel genotípico, comenzando con la participación de dos o más cromosomas (soluciones codificadas) establecidos como padres. A partir de estos padres se generan descendientes cuyos rasgos o características serán definidas por una combinación de los genes de los padres que intervinieron en su creación. Dependiendo del esquema de reemplazo que el algoritmo esté utilizando, se puede establecer la posibilidad de que los padres sobrevivan para formar parte también de la siguiente generación. Los individuos o soluciones se encuentran codificados comúnmente en vectores, podemos generalizar que la recombinación toma componentes de cada padre y crea nuevos individuos realizando la suma de estas componentes. Esta mezcla de información es realizada de manera aleatoria y es la que brinda mucho del poder existente en un AG, caso para el cual la recombinación o cruza representa la principal operación evolutiva.

Durante una recombinación, es posible que los mejores genes de cada padre sean heredados a sus descendientes y de esta manera crear un descendiente cuyos genes expresen las mejores características de cada uno de sus padres. Sin embargo, también existe la posibilidad de que ocurra lo contrario, hecho por el cual esta operación ha sido motivo de controversia debido a la posibilidad de que se presente un comportamiento disruptivo, originando descendientes de mala calidad a partir de padres cuya aptitud sea considerada buena [Pit95]. Debido a la controversia generada, algunos AEs han optado por reducir el impacto que tienen esta operación en la generación de nuevos individuos. Esto lo han realizado estableciendo una tasa de recombinación, la cual define la cantidad de nuevos individuos que

se van a generar a través de la recombinación, o simplemente eliminando esta operación del proceso de evolución [Lee08].

### 3.2.3. Mutación

La poeración de mutación fue introducida originalmente en los AGs como una operación genética muy simple. Se estableció como un cambio aleatorio en el valor de un gen, originando un individuo nuevo con características diferentes. Al igual que en la naturaleza, la taza de mutación se estableció para ser muy pequeña, por lo que se consideró en un principio como un mecanismo secundario en el funcionamiento de los AGs. En la literatura referente a los AEs se ha reflejado un reconocimiento creciente acerca de la importancia de la operación de mutación, convirtiéndola en los últimos tiempos en el principal motor de búsqueda de estos métodos de optimización. El esquema de mutación puede ser tan simple como el empleado originalmente por los AGs, o tan complejo como se desee, dependiendo de la estrategia que se utilice para explorar el espacio de búsqueda. En la actualidad la mayoría de los AEs realizan la mutación induciendo cambios en los individuos empleando información obtenida en base a la experiencia y haciendo uso además de una componente aleatoria. La magnitud de la componente aleatoria que participa y la manera en como participa en el proceso de mutación se define por la variante de AE que se este empleando [Haupt04].

La mutación brinda diversidad a la población a través de la introducción de nuevas soluciones, lo cual puede proteger al algoritmo de convergencia prematura. Una convergencia prematura implica la posibilidad de una rápida obtención no necesariamente del óptimo global. Si se da el caso de que en la función objetivo existan óptimos locales es necesario evitar la tendencia del algoritmo a converger prematuramente. Para superar este problema es necesario mantener cierto nivel de diversidad a través de las generaciones, lo cual permita la exploración de otras áreas del espacio de búsqueda a parte del área donde un comportamiento de rápida convergencia intensificaría la búsqueda. La mutación ayuda a mantener la diversidad en la población y su importancia se acentúa en las generaciones finales, cuando la mayoría de los individuos presentan características similares [Lee08].

### 3.2.4. Selección

La selección es la operación responsable de determinar las características en la convergencia de los AEs. En [Back94] se introduce el término "presión de selección" el cual se refiere al grado en que los mejores individuos serán favorecidos. Durante el proceso de selección se decide cuales individuos son suficientemente aptos para procrear nuevos individuos y/o formar parte de la siguiente generación. De los $N_p$ individuos existentes en la población en determinada generación $g$, sólo algunos son elegidos mientras que el resto se descartan para dejar su lugar a los nuevos individuos.

En general, los esquemas de selección empleados por los diferentes AEs implican que aquellos individuos con grandes valores de aptitud tendrán mayores probabilidades de contribuir en la procreación de uno o más miembros en la siguiente generación. De la misma manera, en la naturaleza la aptitud de los individuos es determinada por su habilidad de sobreponerse a los obstáculos que ofrece su entorno, por lo que los individuos más capaces tendrán más oportunidades de sobrevivir y heredar los rasgos que los hicieron exitosos dentro del medio en el que se desenvuelven.

## 3.3. Paralelismo en algoritmos evolutivos

Cuando la diversidad está garantizada, el enorme poder de búsqueda de los AEs está dado por las grandes poblaciones. Sin embargo, esto resulta al mismo tiempo ser una desventaja de estos métodos en términos de velocidad de convergencia. Es necesario recordar que cada una de las soluciones propuestas por el algoritmo debe ser evaluada, contar con cierta probabilidad de ser elegida para ser recombinada y mutada con el objetivo de generar nuevas soluciones. Cuando todas estas operaciones se realizan bajo un esquema de procesamiento convencional los tiempos de ejecución crecen enormemente. El tiempo de ejecución depende directamente del tamaño de las poblaciones y el número de iteraciones que debe realizar el algoritmo. No obstante, gracias a la naturaleza de los algoritmos evolutivos es posible reducir los tiempos asociados a su ejecución. Cuando se dispone de una arquitectura paralela de cómputo, cada procesador puede llevar a cabo estas operaciones concurrentemente sobre cada individuo y de este modo acelerar su ejecución a través del

uso de técnicas de procesamiento en paralelo.

### 3.3.1. Procesamiento paralelo

Cuando los recursos de cómputo de que se dispone son considerables, es recomendable realizar el máximo aprovechamiento de los mismos. En la actualidad es común el uso de sistemas de cómputo multiprocesadores los cuales van desde PCs con más de un núcleo de procesamiento, hasta enormes clusters o grids de cómputo con cientos o miles de unidades de procesamiento. Sea cual sea el caso, es necesario tomar ventaja del hecho de tener múltiples procesadores y plantear esquemas para la solución de problemas haciendo uso de técnicas de programación paralela. Estas técnicas deben permitir a la vez reducir los inconvenientes que involucran los procesos de comunicación y sincronización necesarios entre los nodos de cómputo de que se disponga [NVIDIA10a].

El concepto básico detrás de un esquema de procesamiento paralelo es la división de un proceso en varios sub-procesos y los cuales son resueltos simultáneamente empleando múltiples procesadores. En la Figura 3.1, se ilustra un ejemplo de procesamiento en paralelo basado en el esquema antes mencionado. En esta figura se ilustra esquemáticamente el procesamiento paralelizado de la aplicación de un filtro detector de bordes a una imagen. Primeramente se particiona la imagen original en tantas partes como tengamos unidades de procesamiento. A continuación cada unidad de procesamiento se dedica a aplicar el filtro a la porción de la imagen que le corresponde y finalmente se conjuntan las particiones en una imagen resultante.

En los últimos años se ha logrado resolver problemas complejos en diferentes áreas de la ciencia aplicada a través del uso exitoso de los AEs. Esto ha originado crecientes demandas a los AEs tales como grandes espacios de búsqueda, costosas funciones de evaluación y grandes tamaños de población que en la mayoría de los casos requiere enormes tiempos de procesamiento [Lee08].

No obstante las crecientes aplicaciones basadas en los AEs, se ha extendido también la necesidad de implementaciones rápidas. Una manera de reducir el tiempo de cómputo requerido para la ejecución de los algoritmos evolutivos es minimizando el número de llamadas a la función de costo o aptitud, sin embargo, la reducción en los tiempos de ejecución

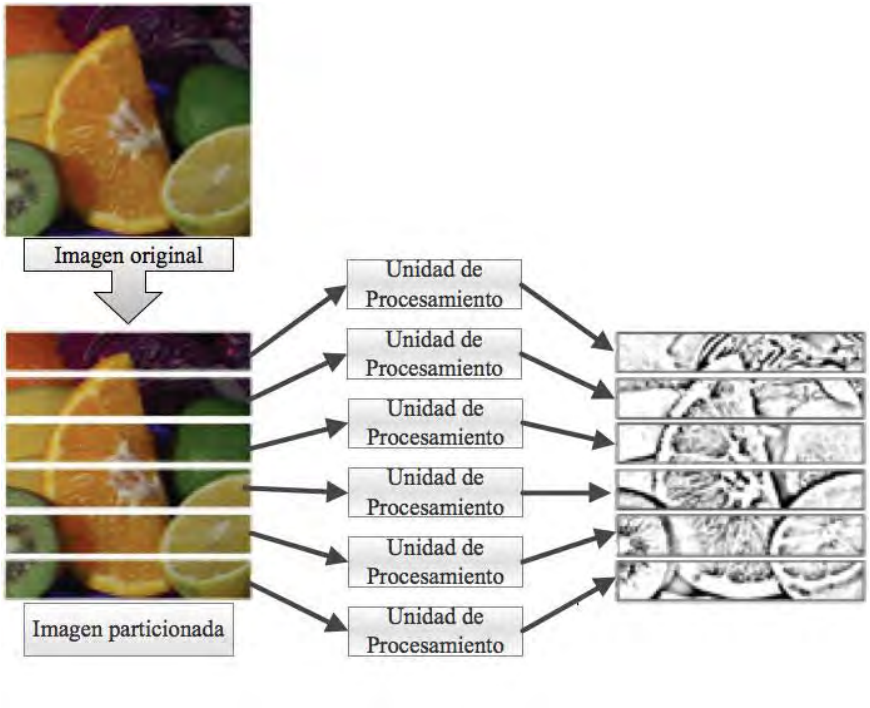



Figura 3.1: Procesamiento paralelo sobre una imagen.

tendría un impacto insignificante [Pit95].

Una ruta natural de exploración ha sido el procesamiento paralelo aplicado a los AEs. Los AEs se ajustan muy bien a esquemas de procesamiento en paralelo ya que las evaluaciones de la función de costo, las recombinaciones genéticas, las mutaciones, la selección e incluso la creación de la población inical son tareas independientes de cada individuo por lo que pueden llevarse a cabo de manera paralela. La realización de estas operaciones en múltiples procesadores de manera paralela sobre los individuos es una manera muy eficiente de reducir los tiempos de ejecución de acuerdo a la ley de Amndahl [Amdahl67]. Otro modelo que puede ser adoptado, es la ejecución de múltiples AEs de manera simultánea y la implementación adicional de algún esquema de interacción entre las diversas ejecuciones [Haupt04].

Los algoritmos evolutivos paralelos se han establecido en la actualidad como un nuevo y promisorio campo de investigación [Pit95]. No obstante, la paralelización de un AE

requiere tomar en cuenta consideraciones importantes. Por ejemplo, la paralelización de operaciones con requerimientos moderados de procesamiento, resulta contraproducente cuando la sincronización y comunicación entre los nodos de procesamiento implica más tiempo de lo que tardaría una implementación secuencial de la operación [NVIDIA10b]. Considerando el caso base de un AG, a continuación se desarrolla la descripción de los diferentes esquemas de procesamiento paralelo empleados por estos procedimientos de optimización.

### 3.3.2. Aceleración esperada por parte de una implementación paralela

Considerando el caso en el que sólo se paraleliza la evaluación de los individuos en un AE, el tiempo de ejecución total de cada generación puede ser calculado, de acuerdo a [Haupt04], como,

$$T_p = \frac{N_p}{P} T_f + \rho(P-1)T_c \tag{3.8}$$

en donde $N_p$ es el tamaño de la población, $T_f$ es el tiempo para evaluar la aptitud de un individuo, $T_c$ es el tiempo promedio de la comunicación entre procesadores, $P$ es el número de procesadores y $\rho$ es un parámetro que depende del metodo de selección y paralelización.

El tiempo total está compuesto por dos términos, el primero se refiere al tiempo requerido para evaluar la aptitud de los cromosomas, mientras que el segundo se refiere al tiempo total de comunicación. La aceleración depende entonces de la proporción entre el tiempo para realizar la evaluación y el tiempo requerido para llevar a cabo la comunicación $(T_f/T_c)$, el número de procesadores disponibles, el tamaño de la población y de los detalles en la metodología empleada para realizar la selección y la paralelización.

## 3.4. Evolución diferencial

La evolución diferencial es un algoritmo evolutivo relativamente nuevo, el cual ha atraído la atención para ser aplicado a una amplia variedad de problemas en la ingeniería [Lee08]. A diferencia de los AEs convencionales que dependen de una distribución de probabilidad predefinida para realizar el proceso de mutación, la evolución diferencial emplea las diferencias de vectores elegidos aleatoriamente para llevar a cabo este proceso. El método

de ED fue introducido en [Storn95] como una alternativa robusta y fácil de usar en comparación con los algoritmos de optimización adaptativos clásicos. De manera similar que en los AEs comunes, un problema de optimización es solucionado codificando soluciones candidatas en vectores, en donde las componentes representan los parámetros del problema de optimización. En el caso de ED las soluciones propuestas se denominan agentes.

La evolución diferencial utiliza $N_p$ agentes que integran una población en cada generación $g$, donde $N_p$ no cambia a lo largo del proceso de optimización. En cualquier momento, cada agente se encuentran representado por un vector con componentes $x_{ij}$ tal que,

$$\mathbf{x}_i^{(g)} = [x_{i1}^{(g)}, x_{i2}^{(g)}, ..., x_{iD}^{(g)}]^T \tag{3.9}$$

en donde $i = 1, ..., N_p$ representa el índice del agente dentro de la población, $j = 1, ..., D$ representa los parámetros en el problema de optimización D-dimensional y $g$ denota la generación o iteración actual del algoritmo.

Como miembro de la familia de los AEs, el método de ED se basa en la generación de una población inicial, mutación, recombinación y selección a través de repetidas generaciones hasta que se cumpla con algún criterio de terminación. El algoritmo de ED se ilustra esquemáticamente en la Figura 3.2. En general, los procesos de generación de una población inicial y la evaluación de los individuos para obtener su aptitud no varían entre los AEs. Sin embargo, este no es el caso de los procesos de selección, mutación y recombinación, los cuales, definen específicamente las diferencias entre los diversos algoritmos evolutivos. Estas diferencias son principalmente en cuanto al esquema de reemplazo utilizado y la estrategia de búsqueda empleada. A continuación se explica con mayor detalle cómo se llevan a cabo el resto de las etapas dentro del algoritmo de ED.

### 3.4.1. Mutación diferencial

Una población $\mathbf{V}$ de vectores de prueba $\mathbf{v}_i$ es producida durante el proceso de mutación (ver Fig. 3.2). En particular, la mutación diferencial agrega la diferencia ponderada de dos vectores elegidos aleatoriamente a un tercer vector para crear un vector mutante $\mathbf{v}_i^{(g)}$

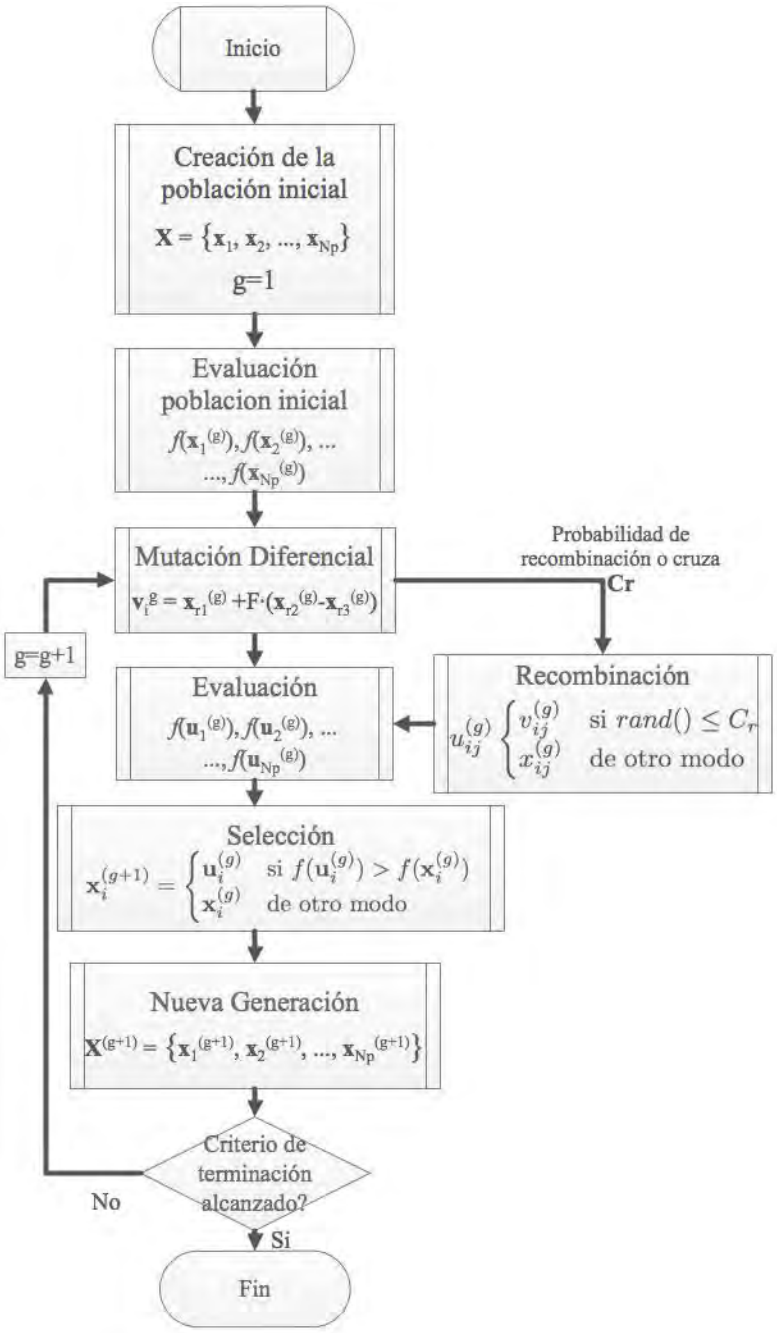



Figura 3.2: Diagrama de flujo del algoritmo de evolución diferencial.

de la forma,

$$\mathbf{v}_i^{(g)} = \mathbf{x}_{r1}^{(g)} + F \cdot (\mathbf{x}_{r2}^{(g)} - \mathbf{x}_{r3}^{(g)}) \tag{3.10}$$

La ecuación (3.10) representa el esquema básico de mutación en ED, en donde $F \in (0, 2]$ se denomina constante de mutación y es empleado para escalar la diferencia entre dos agentes y $r_1, r_2$ y $r_3$ son enteros diferentes entre sí y diferentes del índice $i$ del agente base, los cuales son seleccionados aleatoriamente.

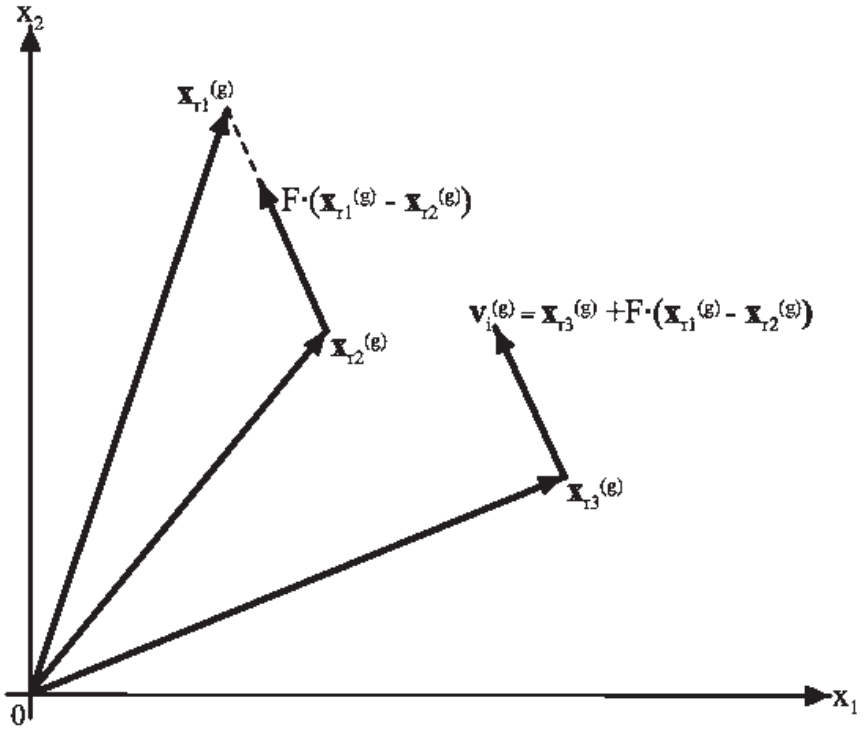



Figura 3.3: Mutación diferencial: la diferencia escalada, $F \cdot (\mathbf{x}_{r1}^{(g)} - \mathbf{x}_{r2}^{(g)})$, es sumada al vector base $\mathbf{x}_{r3}^{(g)}$ para producir un mutante $\mathbf{x}_i^{(g)}$.

La Figura 3.3 muestra la mutación de un vector base $\mathbf{x}_{r3}^{(g)}$ para la optimización de la función de Rosenbrock produciendo un mutante $\mathbf{v}_i^{(g)}$, el cual se encuentra más cercano al valor óptimo. Las líneas tenues denotan las curvas de nivel de esta función.

Varias alternativas han sido propuestas durante los últimos años con el objeto de mejorar la efectividad de este algoritmo. Dependiendo del esquema empleado, un vec-

tor mutante puede ser determinado usando alguna de las siguientes expresiones [Storn95] [Price96],

$$\mathbf{v}^{(g)} = \mathbf{x}_i^{(g)} + \lambda(\mathbf{x}_{best}^{(g)} - \mathbf{x}_i^{(g)}) + F \cdot (\mathbf{x}_{r1}^{(g)} - \mathbf{x}_{r2}^{(g)}) \tag{3.11}$$

$$\mathbf{v}^{(g)} = \mathbf{x}_{best}^{(g)} + F \cdot (\mathbf{x}_{r1}^{(g)} - \mathbf{x}_{r2}^{(g)}) \tag{3.12}$$

$$\mathbf{v}^{(g)} = \mathbf{x}_{r1}^{(g)} + F \cdot (\mathbf{x}_{r2}^{(g)} - \mathbf{x}_{r3}^{(g)}) + F \cdot (\mathbf{x}_{r4}^{(g)} - \mathbf{x}_{r5}^{(g)}) \tag{3.13}$$

$$\mathbf{v}^{(g)} = \mathbf{x}_{best}^{(g)} + F \cdot (\mathbf{x}_{r1}^{(g)} - \mathbf{x}_{r2}^{(g)}) + F \cdot (\mathbf{x}_{r3}^{(g)} - \mathbf{x}_{r4}^{(g)}) \tag{3.14}$$

en donde los parámetros $r_1, r_2, r_3, r_4, r_5$ son enteros seleccionados aleatoriamente diferentes entre sí y diferentes del índice $i$ del vector base. Una variable adicional de control $\lambda$ es introducida en la eq. (3.11), la cual proporciona un medio para aumentar la voracidad del método a través de la incorporación de información acerca de la mejor solución encontrada hasta el momento $\mathbf{x}_{best}^{(g)}$ [Price05]. Las ecuaciones (3.12), (3.13) y (3.14) [Tasoulis04], presentan variantes a partir de los esquemas básicos de ED propuestos originalmente por Rainer Storn y Kenneth Price.

### 3.4.2. Recombinación

Para complementar la estrategia de búsqueda de la mutación diferencial, el algoritmo de ED además emplea una cruza uniforme. En particular, se lleva a cabo una cruza de cada vector $\mathbf{x}_i^{(g)}$ con el correspondiente vector mutante $\mathbf{v}_i^{(g)}$ en la población de prueba. En cada cruza se obtiene un nuevo vector de prueba $\mathbf{u}_i^g$ obteniendo sus parámetros a través de,

$$u_{ij}^{(g)} \begin{cases} v_{ij}^{(g)} & \text{si } rand() \leq C_r \\ x_{ij}^{(g)} & \text{de otro modo} \end{cases} \tag{3.15}$$

en donde $rand()$ es un número aleatorio uniformemente generado en el intervalo $[0, 1]$ para cada parámetro $j$. El parámetro $C_r \in [0, 1]$ se define constante y representa la probabilidad de cruza (ver Fig. 3.2). La probabilidad de cruza ayuda a controlar la fracción de los valores de un mutante que serán usados para formar parte de los nuevos vectores de prueba $\mathbf{u}_i$.

### 3.4.3. Selección

En la operación de selección, se elige la mejor solución entre el vector base $\mathbf{x}_i^{(g)}$ y el correspondiente vector de prueba $\mathbf{u}_i^{(g)}$ de acuerdo a su valor de aptitud. Esto es,

$$\mathbf{x}_i^{(g+1)} = \begin{cases} \mathbf{u}_i^{(g)} & \text{si } f(\mathbf{u}_i^{(g)}) > f(\mathbf{x}_i^{(g)}) \\ \mathbf{x}_i^{(g)} & \text{de otro modo} \end{cases} \tag{3.16}$$

Una vez que la nueva población se ha formado, los procesos de mutación, evaluación, recombinación y selección son repetidos hasta que el valor óptimo es localizado, o hasta que se cumple con un criterio de convergencia establecido (ver Fig. 3.2), por ejemplo, un número máximo de generaciones $g_{max}$.

## 3.5. Evolución diferencial en la GPU

La evolución diferencial es un algoritmo estocástico que requiere la generación de números aleatorios. La falta de una librería eficiente de números aleatorios ha degradado el desempeño de implementaciones previas de AEs paralelos. Dentro de las librerías del lenguaje C, por ejemplo, se encuentra una librería cuyas funciones permiten la generacíon aleatoria de números reales en el intervalo $(0, 1]$. Sin embargo, las funciones que la integran no pueden ser llamadas bajo un esquema multi-hilo y, por lo tanto, en la implementación de cualquier AE paralelo, todas las operaciones que impliquen la generación de números aleatorios tendrán que realizarse secuencialmente. Las propuestas analizadas en la literatura especializada [Tasoulis04] [Kwedlo06] [Zhu09] [Veronese10], por mencionar algunas, sólo presentan la paralelización de la operación de evaluación . Esto se ha justificado argumentando que el resto de las operaciones evolutivas tienen un bajo requerimiento de cómputo y por lo tanto el beneficio de paralelizarlas sería muy pequeño. Es por esto que las propuestas presentadas hasta la fecha se han enfocado en realizar de manera paralela la evaluación, la cual se considera la tarea más demandante dentro del algoritmo y no requiere de la generación de números aleatorios.

En este trabajo de investigación, las dificultades en la generación de números aleatorios en un esquema multi-hilo han sido superadas haciendo uso de la biblioteca CURAND

del entorno CUDA. Gracias al uso de esta biblioteca, también las operaciones de mutación, selección y la creación de la población inicial han sido paralelizadas, quedando sólo una tarea secuencial, la cual está asociada a la determinación de la mejor solución en cada generación $\mathbf{x}_{best}^{(g)}$. La operación asociada a la recombinación o cruza es omitida, esto con el propósito de reducir las variables de control del algoritmo y los problemas relacionados a su ajuste. Además, se incrementa la voracidad del método realizando una búsqueda más directa.

**Nota:** La determinación de la mejor solución es una tarea que también puede ser realizada mediante técnicas de procesamiento paralelo, sin embargo, debido a que es una tarea relativamente sencilla y a que no se adapta al esquema de procesamiento paralelo de CUDA, se optó por realizarla secuencialmente por cuestiones de simplicidad.

### 3.5.1. Esquema de mutación

El algoritmo de ED se basa en la generación de nuevos vectores de parámetros a través de la adición de una diferencia escalada de dos miembros de la población a un tercer miembro. Para el caso del algoritmo de ED paralelo que será ejecutado en la GPU se implementaron algunas variantes. La operación de mutación es realizada empleando una propuesta que reduce el número de variables de control, de este modo, se evita la realización de exhaustivos experimentos para ajustar los valores de estas variables. Se agrega además cierto nivel de voracidad a la estrategia de búsqueda, de acuerdo a [Storn95], a través de la inclusión de la información acerca de la mejor solución global que se ha encontrado.

Durante la mutación, la constante $F$ es sustituida por lo que puede representarse como un vector aleatorio con componentes $\lambda_j \in (0, 1]$, lo cual elimina la necesidad de tener que ajustar $F$ a un valor adecuado. Cada una de las componentes $\lambda_j$ es usada para escalar la diferencia entre las componentes $j$ de dos vectores miembros de la población elegidos aleatoriamente. Como complemento, un factor $(1 - \lambda_j)$ es empleado para escalar la diferencia entre la componente $x_{i,j}$ y la componente $x_{best,j}$ del vector que representa la mejor solución encontrada en la historia. Esta combinación lineal normalizada de diferencias se suma finalmente a cada una de las componentes $x_{i,j}^{(g)}$ del vector base $\mathbf{x}_i^{(g)}$ produciendo un mutante $\mathbf{v}_i^{(g)}$.

De acuerdo a lo anterior, para cada vector en la población en la generación $g$ denotado como $\mathbf{x}_i^{(g)}$, un vector mutante $\mathbf{v}_i^{(g)}$ es creado a tarvés del cálculo de sus componentes por medio de la siguiente definición:

$$v_{i,j}^{(g)} = x_{i,j}^{(g)} + \lambda_j \cdot (x_{r1,j}^{(g)} - x_{r2,j}^{(g)}) + (1 - \lambda_j) \cdot (x_{best,j}^{(g)} - x_{i,j}^{(g)}), \quad \forall \;\; i = 1, 2, ..., N_p, \;\; j = i, 2, ..., D \tag{3.17}$$

en donde $r1$, $r2 \in [1, N_p]$ son enteros aleatorios diferentes entre sí. Empleando este esquema ya no es un valor fijo el que escala las diferencias entre vectores, sino que cada diferencia de componentes es escalada de manera independiente. Esto se hace con el objetivo de reducir la influencia del vector $\mathbf{x}_{best}^{(g)}$, la cual puede afectar la diversidad y producir convergencia prematura.

La ecuación (5.10) establece que la perturbación aplicada a $\mathbf{x}_i^{(g)}$ tendrá dos componentes; la primer componente está asociada a la diferencia entre dos miembros de la población seleccionados aleatoriamente, para la cual se obtiene un escalamiento de manera aleatoria. La segunda de estas componentes está definida por el escalamiento de la diferencia entre la mejor solución y el vector base, dicho escalamiento es limitado a ser $s < 1$, en donde $s = (1 - \lambda_j)$, debido al escalamiento calculado para la primer componente. A través de los dos escalamientos complementarios empleados en la mutación diferencial se pretende establecer un equilibrio en donde se limite la voracidad del método y se mantenga cierto nivel de diversidad en la población, obteniendo como resultado un esquema de búsqueda más directo y que pueda ser capaz de evitar la convergencia prematura.

Esta propuesta en el esquema de mutación surge a partir de conceptos interesantes que forman parte de los métodos de optimización por enjambres de partículas o PSO [Eberhart01]. En donde se establece que el movimiento de las partículas es generado en base a dos componentes: 1).- la componente social, la cual representa la influenicia de la mejor solución global encontrada y 2).- la componente cognitiva, la cual representa la mejor solución conocida por cada partícula. Las magnitudes de esas componentes están dadas también por un escalamientos aleatorios los cuales permiten que en ocaciones influya más la componente social que la cognitiva y viceversa, además de que se establece la posibilidad de que exista un equilibrio entre ambas componentes, es decir que sus magnitudes sean iguales.

### 3.5.2. Esquema de selección

De acuerdo estudios realizados en [Montgomery10] y [Lin10], acerca de los efectos de la variación del parámetro Cr en el desempeño del algoritmo de ED y diferentes métodos de recombinación (incluyendo ED únicamente con mutación), en la presente implementación se decidió probar con la eliminación del efecto de este parámetro al omitir la operación de cruza en la ED. El efecto de la omisión de la operación de recombinación se analiza más adelante en la sección 5.3, en donde se compara la variante implementada con el algoritmo original de ED.

La operación de selección es llevada a cabo de tal manera que el $i$-ésimo mutante es $\mathbf{v}_i^{(g)}$ es seleccionado diréctamente para formar parte de la siguiente generación, sólo si su aptitud es superior a la aptitud del $i$-ésimo miembro de la población en la generación actual $\mathbf{x}_i^{(g)}$. Por ejemplo, si se tarta del proceso de la minimización de la función $f$, esto es,

$$\mathbf{x}_i^{(g+1)} = \begin{cases} \mathbf{v}_i^{(g)} & \text{si } f(\mathbf{v}_i^{(g)}) < f(\mathbf{x}_i^{(g)}) \\ \mathbf{x}_i^{(g)} & \text{de otro modo} \end{cases} \tag{3.18}$$

En general, el desempeño de un algoritmo de ED depende del tamaño de la población, la constante de mutación $F$, la constante de cruza $C_r$ y del número máximo de generaciones $g_{max}$ [Lee08]. Al utilizar el esquema propuesto de ED, los parámetros evolutivos son reducidos debido a que la constante de mutación $F$ fue reemplazada por un número aleatorio $\lambda$, mientras que la constante de cruza $C_r$ fue eliminada. Como consecuencia, sólo el tamaño de la población $N_p$ y el número máximo de generaciones $g_{max}$ restan de ser controladas.

### 3.5.3. Esquema de procesamiento en paralelo

El algoritmo de evolución diferencial implementado en la GPU adopta el esquema *master-slave* de una sola población, en donde el nodo maestro se encuentra representado por la CPU mientras que los esclavos serán cada uno de los SPs dentro de la GPU. Una innovación que se propone, es la realización de todas las operaciones evolutivas en paralelo. Cada uno de los agentes es creado, evaluado, mutado y seleccionado de manera independiente dentro de un kernel de cómputo por un CUDA *thread*. En la Figura 3.4 se presenta

el esquema que muestra la aplicación en paralelo de cada una de estas operaciones sobre cada solución propuesta. Para acelerar la lectura de datos, se forma una textura a partir de la población de individuos; los datos son leídos desde espacios de memoria de texturas y los resultados son escritos en espacios de memoria global.

Empleando el esquema de la Figura 3.4, los datos permanecen todo el tiempo en la memoria de la GPU. El mantener los datos todo el tiempo en la memoria del dispositivo elimina la necesidad de realizar grandes transferencias de datos entre la CPU y la GPU. A pesar de que las GPUs modernas poseen un gran ancho de banda dedicado a la comunicación con el procesador central [NVIDIA10d], es de gran utilidad reducir el tráfico de datos en la medida posible. Una reducción eficiente en el tráfico de datos entre el *host* y el *device* aumenta considerablemente el desempeño de aplicaciones de gran escala [NVIDIA10b].

En la Figura 3.4 cada proceso predefinido en el diagrama de flujo (creación de la población inicial, evaluaciones, mutación y selección) es realizado por un *grid* de bloques de *threads* por lo que cada una de estas operaciones es implementado como un kernel de cómputo. Los *threads* son dispuestos en bloques unidimensionales de longitud 16, esta longitud fue definida epíricamente. El número máximo de *threads* por bloque permitido por la arquitectura Fermi es hasta 1024, sin embargo, para la presente aplicación una longitud de 1024 presentaba un desempeño menor. Una vez definida la longitud de los bloques es necesario integrarlos dentro de un *grid*, los bloques de *threads* fueron integrados en un *grid* unidimensional cuya longitud es calculada como,

$$Grid\_Size = \left\lceil \frac{N_p}{16} \right\rceil \tag{3.19}$$

en donde $N_p$ representa el tamaño de la población y el resultado al entero inmediato superior, para de esta manera asegurar la creación de un número suficiente de bloques para satisfacer la demanda de $N_p$ *threads* requeridos para atender a toda la población. Una vez definido el orden en que serán dispuestos los *threads* para su ejecución, se procede a invocar cada uno de los kernels de cómputo de acuerdo al orden lógico del algoritmo de ED. Las operaciones son aplicadas independiente y paralelamente sobre cada uno de los individuos en la población, restando sólo para su ejecución de manera secuencial la tarea de identificar la mejor solución obtenida en cada iteración (ver Fig. 3.4).

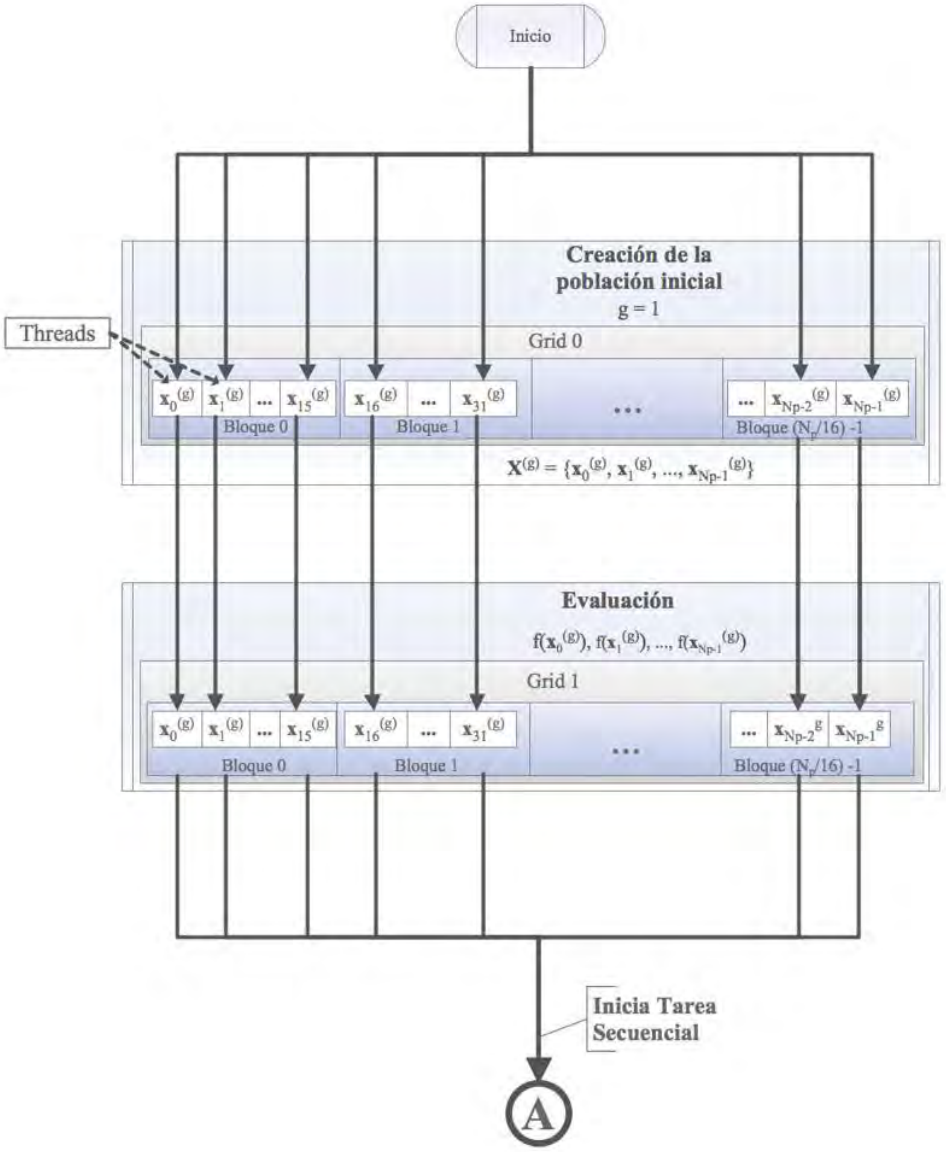



Figura 3.4: Esquema de paralelización del algoritmo de ED en la GPU.

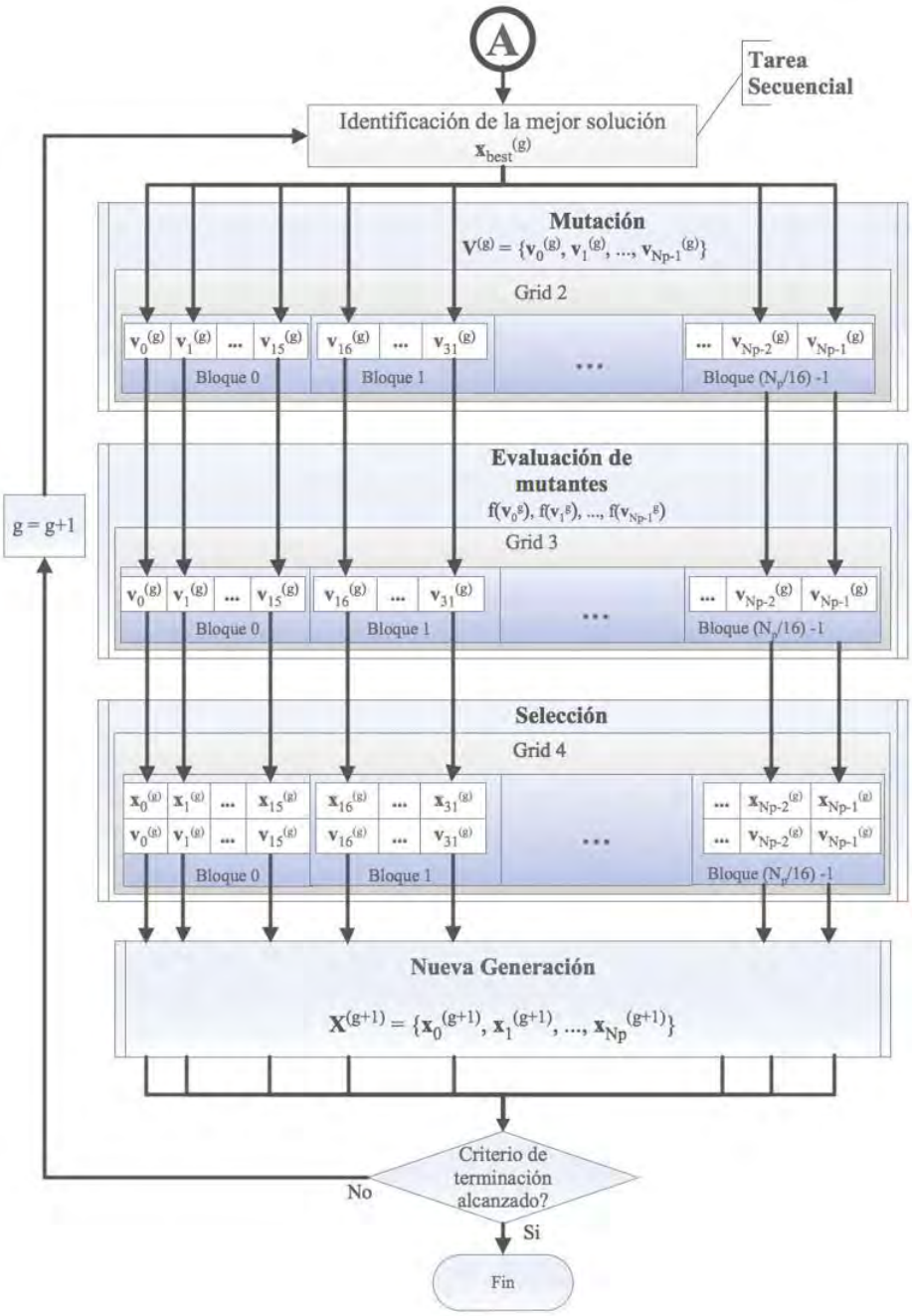



Figura 3.5: Esquema de paralelización del algoritmo de ED en la GPU (continuación).

### 3.5.4. Generación de números aleatorios

La versión más reciente de CUDA incluye la biblioteca CURAND. Esta biblioteca proporciona las facilidades para generar grandes series de números pseudo-aleatorios. Una serie pseudo-aleatoria es una secuencia de números generada determinísticamente pero que satisface en su mayoría las propiedades estadísticas de una verdadera secuencia de números aleatorios. La librería CURAND soporta además la generación de bits y la generación de números pseudo-aleatorios con una distribución específica [NVIDIA10].

#### Inicialización de una secuencia de números aleatorios

Para poder hacer uso de la biblioteca CURAND, es necesario incluir el archivo `curand_kernel.h` en aquellos archivos que declaren kernels de cómputo o funciones en el *device* que usen esta biblioteca. Este archivo define las funciones del *device* para la creación de un generador de números aleatorios y generar secuencias de números aleatorios.

Las series de números aleatorios en la biblioteca CURAND son creadas empleando algoritmos determinísticos [NVIDIA10]. Estos algoritmos utilizan un valor inicial denominado semilla, en base al cual establecen un estado inicial en una serie de números aleatorios. La función `curand_init()` establece un estado inicial para una secuencia aleatoria empleando una semilla dada. Esta función tiene la sintaxis,

```
void curand_init (unsigned long seed,unsigned long sequence,
                  unsigned long offset, curandState *state)
```

De esta manera se establece un estado inicial, el cual es almacenado en el parámetro *state* del tipo `curandState`, un tipo de dato integrado en esta biblioteca. Esta función permite también, identificar a la secuencia a través del parámetro *sequence*. El parámetro *offset* representa un salto hacia adelante en la secuencia. Si se establece *offset*= 100, por ejemplo, el primer número aleatorio generado será el centécimo en la secuencia. Esto permite que múltiples ejecuciones ejecuciones del mismo programa puedan generar resultados de la misma secuencia sin traslaparse [NVIDIA10]. Para una misma semilla (*seed*), esta función siempre producirá el mismo estado inicial y la misma secuencia.

**Uso de una secuencia de números aleatorios**

Después de haber establecido el estado inicial para una secuencia aleatoria (a través de la función `curand_init()`), es posible obtener números aleatorios de esta secuencia a través de las funciones `curand()` y `curand_uniform()`. Estas funciones regresan secuencias de números pseudo-aleatorios con un periodo igual a $2^{67} \cdot sequence + offset$ [NVIDIA10].

La invocación de la función `curand()` regresa un número entero sin signo, con distribución uniforme. Está función está definida como,

```
unsigned int curand (curandState *state)
```

en donde el parámetro *state* se refiere al estado actual de la secuencia; *state* es actualizado en cada llamada a la función `curand()`.

Otra función de gran ayuda es la funcion `curand_uniform()` definida como sigue,

```
float curand_uniform (curandState *state)
```

la cual regresa en cada llamada una secuencia de números flotantes pseudo-aleatorios uniformente distribuidos dentro del rango $(0.0, 1.0]$.

**Notas de rendimiento**

Para realizar una generación de números aleatorios de manera paralela y óptima, cada *thread* debe generar su propia secuencia de números empleando una semilla única. Además de esta consideración, es necesario tomar en cuenta las siguientes recomendaciones:

- Una llamada a la función `curand_init()` demanda un tiempo de ejecución mayor que llamar a las funciones `curand()` o `curand_uniform()`. Por lo tanto, se recomienda almacenar en memoria global el estado actual de la serie y emplearlo como estado inicial en futuras llamadas a estas funciones, en lugar de estar calculando un estado inicial repetidamente [NVIDIA10].

- Grandes valores de *offset* al llamar la función `curand_init()` requerirán más tiempo para establecer el estado inicial de una secuencia.
- La iniciailización del estado de una secuencia requiere más registros y memoria local que la generación de los números aleatorios. Por lo tanto, se recomienda inicializar todos los generadores (secuencias) necesarios en un kernel exclusivo para ello antes de comenzar a generar los números aleatorios.

El estado de un generador se puede almacenar en memoria global entre las invocaciones de distintos kernels de cómputo que requieran hacer uso de él. Si se requiere la generación rápida de más de un número aleatorio dentro del kernel de cómputo, es posible almacenar el estado del generador en memoria local y una vez que se haya dejado de utilizar el generador, guardar de nuevo el estado actual en memoria global. En el listado 3.1 se muestra un código resumido de como puede realizarse lo mencionado previamente.

```
#include curand_kernel.h
⋮
/* Kernel que recibe un arreglo de curandStates (uno por cada thread)
     almacenados en memoria global */
__global__ void RNG_Kernel(curandState *global_state){
        // ID del thread
        int i = threadIdx.x;

        unsigned int x;

        // Se almacena el estado en memoria local
        curandState local_state;
        local_state = global_state[i];

        for(int j = 0; j < 10000; j++) {
                /* los numeros generados pueden almacenarse en cualquier
                    tipo de variable: local o global */
                x = curand(&local_state);
                ⋮
        }
```

```
        // Se almacena el estado actual en memoria global
        global_state[i] = local_state;
}
```

Listado 3.1: Ejemplo del manejo del estado de un generador de números aleatorios.

El establecimiento del estado inicial de un generador de números aleatorios puede requerir hasta 16KB de memoria local asignada en la pila por cada *thread*. El tamaño de pila por defecto es de 1KB por *thread*. Para poder hacer la inicialización de los generadores de números aleatorios requeridos, se debe establecer explícitamente el tamaño de la pila para cada *thread*. Esto se realiza a través de la función,

```
cudaThreadSetLimit(cudaLimitStackSize, size)
```

la cual debe de ser llamada como una instrucción previa a la invocación del kernel que realizará la definición de los estados iniciales de los generadores. El parámetro *size* especifica el tamaño de la pila en Bytes. Una vez que el kernel que define los estados iniciales de los generadores de números aleatorios ha terminado su ejecución, es posible regresar el tamaño de pila por *thread* a su valor por defecto.

**Ejemplo del uso de la librería CURAND**

Para mostrar como se realiza el establecimiento de los estados iniciales de $N$ generadores de números aleatorios y el proceso de generación, se presenta en el Listado 3.2 un código resumido que muestra el proceso de generación de números aleatorios empleando la librería CURAND. Este código se divide en cuatro partes:

1. **Una función principal**. En la cual se declaran la mayoría de las variables requeridas, se reservan recursos de memoria para estas variables y se instauran las condiciones necesarias para realizar la creación de $N$ generadores (o secuencias) de números aleatorios.

2. **Una función de inicialización**. La cual establece los valores de todos los parámetros requeridos en la definición de los estados iniciales de los $N$ generadores de números

aleatorios y que además se encarga de invocar el kernel de cómputo que realizará esta tarea.

3. **Un kernel de establecimiento de estados iniciales**. El cual crea los estados iniciales, disponiendo de los parámetros necesarios para llamar a la función `curand_init()`.

4. **Un kernel de generación de números aleatorios.** El cual muestra el procedimiento para hacer uso de las secuencias de números aleatorios, inicializadas previamente en un kernel de cómputo dedicado exclusivamente a esta tarea.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include curand_kernel.h
/* Declaracion del kernel para establecer los estados
   iniciales de las secuencias de numeros aleatorios,
   el cual recibe un arreglo de estados y las semillas
   que utilizara para definirlos */
__global__ void setupRNG_kernel(curandState *states, unsigned long *seeds){
        int i = threadIdx.x;    // Se obtiene el ID del thread
        /* Se usa la semilla correspondiente del arreglo,
           para inicializar el estado de la secuencia,
           el estado inicial se almacena en memoria global */
        curand_init(seeds[i], i, 0, &state[i]);
}
/* Se declara la funcion que invocará el kernel para el establecimiento
   de los estados iniciales de las N secuencias de numeros aleatorios */
void SetupRNG(int N, curandState *d_states){
        /* Se declaran los arreglos para almacenar las semillas
           utilizadas para crear los estados iniciales de las
           secuencias de numeros aleatorios */
        unsigned long  *d_seeds, *h_seeds;
        /* Se inicializa el generador de numeros aleatorios de la
           de la libreria estandar de C */
        srand((unsigned) time(0));

        // Se reserva memoria en el host para N semillas
```

```
        h_seeds = (unsigned long *) malloc(N * sizeof(unsigned long));
        for(i=0; i<N; i++){
                // Se establece cada semilla con un numero aleatorio en C
                h_seeds[i] = (unsigned long) rand();
        }
        ⋮
```

Listado 3.2: Ejemplo del uso de la librería CURAND.

```
        ⋮
        // Se reserva memoria en el device para N semillas
        cudaMalloc((void **)&d_seeds, N * sizeof(unsigned long));
        // Se copian las semillas almacenadas en el host en el device
        cudaMemcpy(d_seeds, h_seeds,  N * sizeof(unsigned long),
                                        cudaMemcpyHostToDevice);
        // Se establece el tamano de la pila de cada thread en 16 KB
        cudaThreadSetLimit(cudaLimitStackSize, 16*1024);

        /* Se invoca el kernel que inicializara los estados de las N
           secuencias de numeros aleatorios */
        setupRNG_kernel<<<1, N>>>(d_states, d_seeds);

        // Se restablece el tamano de la pila de cada thread en 1 KB
        cudaThreadSetLimit(cudaLimitStackSize, 1024);
        ⋮
        // Se libera la memoria reservada
        cudaFree(d_seeds);
        free(h_seeds);
}

/* Funcion principal en el ejemplo de uso de la libreria CURAND */
int main(int argc, char **argv){
        ⋮
        // Se declara un arreglo de estados iniciales
        curandState *d_States;
        ⋮
        // Se reserva memoria en la GPU para almacenar N estados iniciales
        cudaMalloc((void **)&d_States, N * sizeof(curandState));
        /* Se invoca a la funcion que establecerá los N estados
           iniciales para N secuencias de numeros aleatorios */
        setupRNG(N, d_States);
        ⋮
```

Listado 3.2: Ejemplo del uso de la librería CURAND (continuación).

```
        // Invocacion del kernel declarado en el listado 3.1
        RNG_Kernel<<<1, N>>>(d_States);
        ⋮
        // Liberamos la memoria ocupada por los estados
        cudaFree(d_States);
}
```

Listado 3.2: Ejemplo del uso de la librería CURAND (continuación).

Tal como se puede apreciar en la función principal, una vez que se inicializaron los estados de los generadores, es posible hacer uso de ellos. En este caso, al final de la función principal se invoca al kernel declarado en el Listado 3.1, el cual realiza simplemente la generación de 10,000 números aleatorios en paralelo por cada *thread*.

## 3.6. Sumario

Los AEs son métodos estocásticos de optimización que representan modelos reducidos de fenómenos manifestados en la naturaleza. Todos estos modelos son ejemplos de procesos con un éxito notable de algún tipo de optimización en el mundo natural. El cálculo proporciona herramientas suficientes para optimizar muchas funciones de costo de una manera elegante y eficiente. Sin embargo, este proceso no nos puede asegurar si un óptimo encontrado es el óptimo global o existen mejores soluciones. Los métodos determinísticos de optimización pueden encontrar rápidamente la solución superando el rendimiento de un algortitmo evolutivo, encontrando rápidamente el mínimo mientras un AE se encontraría realizando las primeras iteraciones. De cualquier manera, la principal desventaja de los enfoques determinísticos de optimización es que no se ajustan para la solución de la mayoría de los problemas en el mundo real, comúnmente encuentran el óptimo incorrecto y no funcionan correctamente con variables discretas. Los AEs son capaces de producir resultados adecuados cuando los métodos tradicionales de optimización fallan.

El enorme poder de búsqueda de los AEs está dado por las grandes poblaciones. Sin embargo, los tiempos de ejecución crecen enormemente dependiendo del tamaño de

las poblaciones y el número de iteraciones que debe realizar el algoritmo. No obstante, gracias a la naturaleza de los algoritmos evolutivos es posible reducir los tiempos asociados a su ejecución a través del uso de técnicas de procesamiento en paralelo. La evolución diferencial es un algoritmo evolutivo relativamente nuevo el cual ha atraído la atención de una amplia variedad de aplicaciones en la ingeniería. El método de ED fue propuesto como una alternativa robusta y fácil de usar a los algoritmos de optimización adaptativos clásicos. Como parte de la familia de los AEs, este método ED es susceptible de ser paralelizado. Las operaciones de cruza, mutación, selección e incluso la creación de la población inicial pueden realizarse empleando esquemas de procesamiento en paralelo.

En este trabajo de investigación, se propone la implementación de un algoritmo de evolución diferencial implementado en la GPU, con algunas variantes al algoritmo original. Tales variantes involucran la propuesta de un nuevo esquema de mutación y la omisión de la operación de recombinación, con el objeto de introducir cierto nivel de voracidad a la estrategia de búsqueda. Esta implementación propone la realización de todas las operaciones evolutivas en paralelo dentro de una GPU, haciendo uso del entorno de programación CUDA. Con lo que se pretende lograr una aceleración considerable en la ejecución de este método.

Capítulo 4

# Implementación de la Evaluación de individuos en la GPU

Con el objetivo de establecer una referencia rápida para futuras implementaciones en GPUs, en este capítulo se ejemplifica el modelo de paralelización en las GPUs NVIDIA a través de la descripción del código para realizar de manera paralela la operación de evaluación en el algoritmo de ED en CUDA. El código completo del algoritmo de ED paralelo en CUDA se presenta en el apéndice B de esta tesis. La paralelización de la evaluación de los individuos y en general todas las operaciones evolutivas fueron realizadas empleando programación modular la cual se describe en tres etapas. La primera de estas etapas se refiere al llamado de las funciones encargadas de realizar cada una de estas operaciones desde la función principal, la segunda involucra a las funciones encargadas de establecer las condiciones para que los kérneles de cómputo puedan ser invocados y la invocación de los mismos; por último se tiene la implementación de las funciones asociadas a los kérneles de cómputo.

## 4.1. Función principal

La función principal o *main* es la función que controla el flujo de ejecución de la implementación, es decir, representa al módulo principal en la programación del algoritmo de

ED. Junto con esta función se definen los parámetros de ejecución del algoritmo, mientras que dentro de ella se establecen se declaran las variables y se reservan los recursos de memoria necesarios para la ejecución del algoritmo. La función principal del algoritmo de ED se ilustra esquemáticamente en la Figura 4.1.

Como se puede observar en la Figura 4.1, es necesario definir los parámetros de ejecución del algoritmo, estos son: el tamaño de la población $N_p$, el número de dimensiones del problema de optimización $D$, los límites del espacio de búsqueda $[u_j, l_j]$ y el número

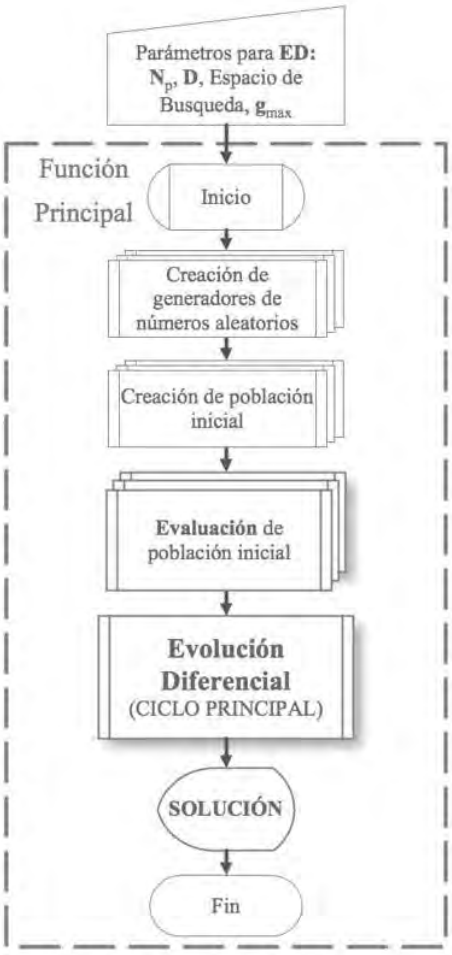



Figura 4.1: Diagrama de flujo de la función principal del algoritmo de ED.

máximo de generaciones $g_{max}$. El espacio de búsqieda se limita a través de un arreglo bidimensional de $D \times 2$, en donde la primer columna representa los límites inferiores de cada una de las $D$ dimensiones mientras que la segunda columna representa los límites superiores, respectivamente. Una vez que se han definido los parámetros de ejecución, se realiza de manera paralela la creación de $N_p$ generadores de números aleatorios, la creación de la población inicial y finalmente la evaluación de ésta población. Una vez terminada la evaluación de la población inicial se inicia el ciclo principal del algoritmo de ED, el cual desplegará en pantalla la solución obtenida una vez cumplida su condición de terminación.

El Listado 4.1 muestra el código en CUDA correspondiente a la definición de los parámetros de ejecución del algoritmo de ED,

```
/* PARÁMETROS DEL ALGORITMO DE EVOLUCIÓN DIFERENCIAL */
#define N_p 8192 // Tamaño de la población
#define D 128 // Número de dimensiones del problema de optimización
#define G_MAX 3000 // Número máximo de generaciones
__constant__ float Restrictions[D][2] = {{-600, 600},{-600, 600},{-600, 600},
                                         {-600, 600},{-600, 600},{-600, 600},
                                         ⋮
                                         {-600, 600},{-600, 600},{-600, 600}};
/* FUNCIÓN PRINCIPAL */
int main(int argc, char **argv){ // Inicio
        size_t size = D * sizeof(float); // Tamaño de un individuo en Bytes
        ⋮
        // Arreglos para almacenar las poblaciones principal y de prueba
        float *Population, aux_Population;
        /* Arreglos para almacenar la aptitud de los individuos */
        float *Fitness, aux_Fitness;
        ⋮
```

Listado 4.1: Versión simplificada de la función principal.

```
⋮
/* Arreglo para almacenar los estados de los generadores
  de números aleatorios */
curandState *d_States;
/* Se reserva memoria en el device para almacenar los estados
  para N_p generadores de números aleatorios */
cudaMalloc((void **)&d_States, N_p * sizeof(curandState));
/* CREACIÓN DE N_P GENERADORES DE NÚMEROS ALEATORIOS
setupRNG(d_States);

/* Se reserva memoria en el device para almacenar la población*/
cudaMalloc((void**)&Population, Np * size));
/* CREACIÓN DE LA POBLACIÓN INICIAL */
CreateInitPop(Population, d_States);
⋮
/* Se reserva memoria en el device para almacenar
 el valor de aptitud de los individuos */
cudaMalloc((void**)&Fitness, Np * sizeof(float));
/* EVALUACIÓN DE LA POBLACIÓN INICIAL */
Evaluation(Fitness, Population);
⋮
// Número de generación
g = 1;
/ * CICLO PRINCIPAL DEL ALGORITMO DE EVOLUCIÓN DIFERENCIAL */
while ((g < G_MAX) && (fitness_best_solution > 1e-10)) {
        // Mutación
        Mutation(aux_Population, Population, d_States);

        //Evaluación
        Evaluation(aux_Fitness, aux_Population);
        ⋮
```

Listado 4.1: Versión simplificada de la función principal (continuación).

```
                ⋮
                // Selección
                Selection(Population, Fitness, aux_Population, aux_Fitness);

                // Se obtiene y actualiza la mejor solución encontrada
                UpdateBest(Best_Solution, Fitness_Best_Solution,
                                        Population, Fitness, N_p);
                g++;
        }
        ⋮
        // Liberación de la memoria empleada por los datos
        cudaFree(Population); cudaFree(aux_Population);
        cudaFree(Fitness); cudaFree(aux_Fitness);
} // Fin
```

Listado 4.1: Versión simplificada de la función principal (continuación).

Aunque este código es autodescriptivo en gran parte, es necesario aclarar lo que representan algunas de sus variables y puntualizar sobre el manejo de los datos en algunas de ellas. En este sentido, la Tabla 4.1 describe los datos representados por las variables en el código principal.

Las poblaciones en los AEs comúnmente son representadas como matrices, sin embargo, en esta ocasión se justifica el uso de arreglos unidimensionales en términos de eficiencia. Si las poblaciones son representadas como arreglos unidimensionales, es posible realizar la asignación de memoria, las copias de los datos entre la CPU y la GPU y la liberación de los espacios de memoria en una sóla instrucción. De otro modo, si se utilizaran arreglos en 2 dimensiones (dobles apuntadores) estas operaciones tendrían que realizarse renglón por renglón. La representación unidimensional de las poblaciones no implica mayores complicaciones, por ejemplo, considere el arreglo $\mathbf{x}$ que representa a la población $\mathbf{X}$. Es posible identificar donde comienza el individuo $i$ dentro de este arreglo a través de una operación sencilla, esto es,

$$x_{i,0} = \mathbf{x}[i \times D] \tag{4.1}$$

Tabla 4.1: Parámetros representados por las variables en la función principal.

| Variable | Parámetro |
|---|---|
| N_p | Constante que define el tamaño de la población. |
| D | Constante que define las dimensiones del problema. |
| G_MAX | Número máximo de generaciones. |
| Restrictions | Arreglo bidimensional que almacena los límites inferiores y superiores $[u_j, l_j]$ en cada dimensión del espacio de búsqueda. |
| size | Tamaño en bytes del vector representante de un individuo. |
| Population | Arreglo que almacena la población de individuos. |
| Fitness | Arreglo que almacena el valor de aptitud de los individuos en la población. |
| aux_Population | Arreglo auxiliar que almacena los individuos generados en la mutación. |
| aux_Fitness | Arreglo auxiliar que almacena la aptitud de los individuos generados en la mutación. |
| d_States | Arreglo que almacena los estados de los N_p generadores de números aleatorios necesarios por el algoritmo de ED. |
| Best_Solution | Arreglo que almacena la mejor solución encontrada hasta el momento. |
| Fitness_Best_Solution | Variable que almacena el valor de aptitud de la mejor solución encontrada hasta el momento. |

mientras que si se quiere hacer referencia a la componente $j$ del individuo $i$, se realiza:

$$x_{ij} = \mathbf{x}[i \times D + j] \tag{4.2}$$

## 4.2. Función para invocar el kernel de evaluación.

Siguiendo el esquema modular, se requiere una función que defina los datos necesarios y que establezca las condiciones necesarias para invocar un kernel de cómputo dentro del entorno de programación CUDA. El objetivo principal de esta función es la invocación del kernel de evaluación. La ejecución de la función kernel es realizada bajo un esquema de paralelismo en el cual cada *thread* va a realizar la evaluación de un único individuo en la población. Por lo tanto, es necesario realizar un diseño lógico en la disposición de los *threads* de manera que se se puedan crear tantos *threads* como individuos existan en la población.

El diagrama de flujo de la ejecución de esta función se ilustra en la Figura 4.2

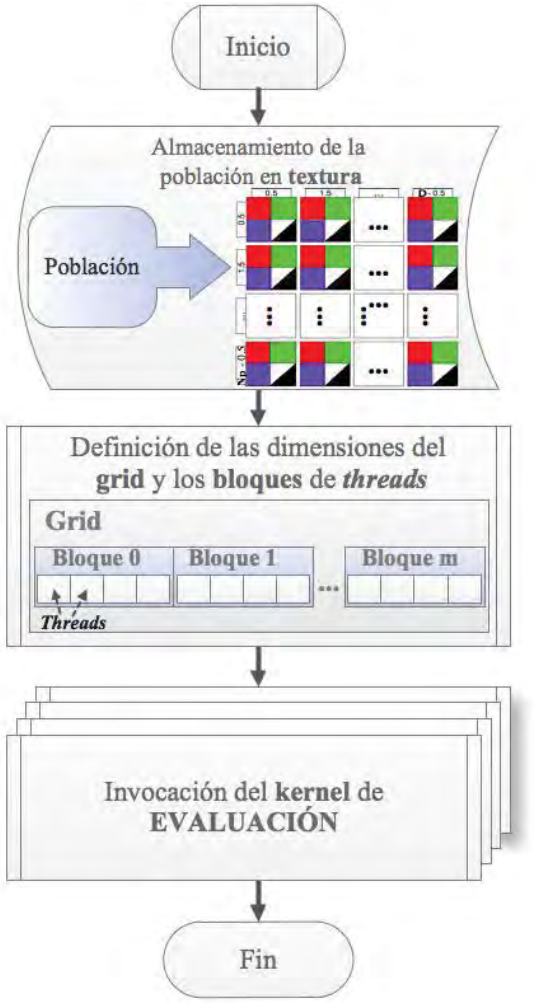



Figura 4.2: Diagrama de flujo de la función encargada de invocar el kernel de evaluación.

La función representada por el diagrama en la Figura 4.2 invocará el kernel que evaluará cada individuo en paralelo. Esta función requiere solamente de dos argumentos: el vector donde almacenará los valores de aptitud para cada individuo (`Fitness`) y el vector donde se encuentra almacenada la población (`Population`). Una vez que se ha recibido el argumento de la población, ésta es almacenada en un espacio de memoria de textura con la finalidad de agilizar la lectura de estos datos desde el kernel de cómputo. En el listado

4.2 se presenta el código referente a la función que invoca el kernel responsable de realizar la operación de la evaluación de la población.

```
// Textura en donde será almacenada la población
texture<float, 2, cudaReadModeElementType> texPopulation;
/* FUNCIÓN QUE INVOCA EL KERNEL DE EVALUACIÓN DE LA POBLACIÓN */
void Evaluation(float *Fitness, float *Population){ // Inicio
        int GRID_SIZE;

        // cudaArray para almacenar la población
        cudaArray *Population_array;
        // Descrinpción de los canales RGBA y los datos empleados por la textura

        cudaChannelFormatDesc channelDesc =
                cudaCreateChannelDesc(32, 0, 0, 0,cudaChannelFormatKindFloat);
        /* Se reserva memoria para un cudaArray
          con la descripción chanelDesc y de dimensiones D×N_p */
        cudaMallocArray( &Population_array, &channelDesc, D, N_p);
        // Se copia la población al cudaArray
        cudaMemcpyToArray(Population_array, 0, 0, Population,
                                N_p * D * sizeof(float),
                                cudaMemcpyDeviceToDevice);
        // Se vincula la textura al cudaArray
        cudaBindTextureToArray( texPopulation, Population_array, channelDesc);

        /*Se calcula el tamaño de los bloques de threads en función de
          el tamaño de población y el tamaño del grid */
        GRID_SIZE =(int) ceil((float) N_p/BLOCK_SIZE);

        /* INVOCACIÓN DEL KERNEL DE EVALUACIÓN */
        Evaluation_Kernel<<<GRID_SIZE, BLOCK_SIZE>>>(Fitness);
} // Fin
```

Listado 4.2: Función encargada de invocar el kernel de cómputo para efectuar la evaluación de la población.

Los datos representados por las variables declaradas en esta función son descritos en la Tabla 4.2.

Tabla 4.2: Parámetros representados por las variables en la función encargada de invocar el kernel para efectuar la evaluación de la población.

| **Variable** | **Parámetro** |
|---|---|
| `Fitness` | Arreglo que almacena el valor de aptitud de los individuos en la población. |
| `Population` | Arreglo que almacena la población de individuos. |
| `texPopulation` | Textura formada a partir de la población. |
| `channelDesc` | Variable que describe el formato de los datos que se van a manejar por la textura `texPopulation`. |
| `Population_array` | cudaArray que almacena la población para formar la textura `texPopulation`. |
| `GRID_SIZE` | Tamaño del grid que contendrá los bloques de *threads*. |
| `BLOCK_SIZE` | Tamaño de los bloques de *threads*. |

De acuerdo al Listado 4.2, en primera instancia la población se encuentra almacenada en un espacio de memoria global. Las lecturas desde los espacios de memoria global de la GPU son los más costosos computacionalmente. Por otra parte, los espacios de memoria asignados para las texturas se encuentran en caché, por lo que leer los datos de la población desde una referencia a textura aumenta considerablemente la eficiencia al realizar esta operación. Las texturas sólo pueden ser vinculadas a un cudaArray, es por eso que se requiere primeramente copiar los datos almacenados en el vector `Population` al *cudaArray* `Population_array`.

Las texturas son capaces de manejar cuatro datos simultáneamente correspondientes a los cuatro canales del sistema RGBA. Por lo tanto, es necesario establecer la descripción del formato del uso de los cuatro canales que será utilizada por la textura `texPopulation`. Esta descripción se establece en la variable `channelDesc`, en donde se especifica el número de bits que se empleará para representar cada canal además del formato de los datos que se van a manejar. Sin embargo, debido a que para los fines de la presente implementación no se requiere más que del manejo de un dato en la textura, sólo se activa el canal R con una resolución de 32 bits y se especifica que los datos serán de punto flotante. A continuación, se procede a vincular la textura al *cudaArray*, quedando disponibles los datos en la textura para su lectura desde el kernel de cómputo.

Después de la vinculación de la textura al *cudaArray* lo que resta es establecer

la disposición lógica de los *threads* que van a ejecutar el kernel e invocar el kernel de cómputo. En este caso los *threads* fueron dispuestos en bloques unidimensionales de tamaño `BLOCK_SIZE`, mientras que los bloques de *threads* se encuentran también dentro de un grid unidimensional de tamaño `GRID_SIZE`. Es necesario considerar que el tamaño de los bloques está dado por la relación entre el tamaño de la población y el tamaño del grid, por lo que se debe prestar especial cuidado de que el número de *threads* dentro de cada bloque no supere al máximo número permitido por la arquitectura de la GPU. En caso de que el kernel sea ejecutado con una configuración en la cual el tamaño de los bloques de *threads* sea mayor al máximo permitido por la arquitectura, el kernel se ejecutará aparentemente sin problemas. Sin embargo, la integridad de los datos no se conserva y se generan datos con valores inconsistentes en los resultados.

Finalmente, sólo queda invocar el kernel de cómputo `Evaluation_Kernel`, el cual requiere únicamente de un argumento: un apuntador al vector `Fitness`, en el cual se almacenará el valor de aptitud de los individuos en la posición correspondiente al índice de cada individuo dentro de la población.

## 4.3. Definición del kernel de evaluación

El kernel de evaluación es la función que será ejecutada en paralelo por todos los *threads*. Esta función sólo recibe como argumento el arreglo `Fitness`, en el cual se colocará el valor de aptitud correspondiente a cada individuo de acuerdo a su índice dentro de la población. El kernel de evaluación es presentado en el listado 4.3.

Debido a que la implementación presentada en esta tesis se planteó desde un principio como una herramienta general, la función de aptitud `fitness_function` se define de manera genérica para que de esta manera pueda ser personalizada para los requerimientos del problema de optimización que se esté abordando. La función `fitness_function` recibe dos argumentos, el primero es un apuntador a la posición del arreglo `Fitness` en cuya referencia se almacenará el resultado de la evaluación. El segundo corresponde a la coordenada en $y$ del sistema de coordenadas de las texturas, para el cual se debe considerar que dentro de las texturas existe un desplazamiento de 0.5 en cada coordenada, tal como

```
/* KERNEL DE EVALUACIÓN */
__global__ void Evaluation_Kernel(float *Fitness){ // In icio
        /* Se obtiene el identificador del thread, el cual a su vez
          representa el índice de cada individuo dentro de la población */
        int y = blockIdx.x * blockDim.x + threadIdx.x;

        // Si el índice no está más allá del tamaño de la población
        if (y < N_p){
                /* Se llama la función de aptitud, especificando el índice
                   del individuo del cual va ser evaluado */
                fitness_function(&Fitness[y], y);
        }
} // Fin
```

Listado 4.3: Definición del kernel de cómputo para la evaluación.

se observa en la Figura 4.3. El valor de $y$ representa además el identificador del *thread* que está ejecutando el kernel, al mismo tiempo que denota el índice del individuo que se está evaluando dentro de la población

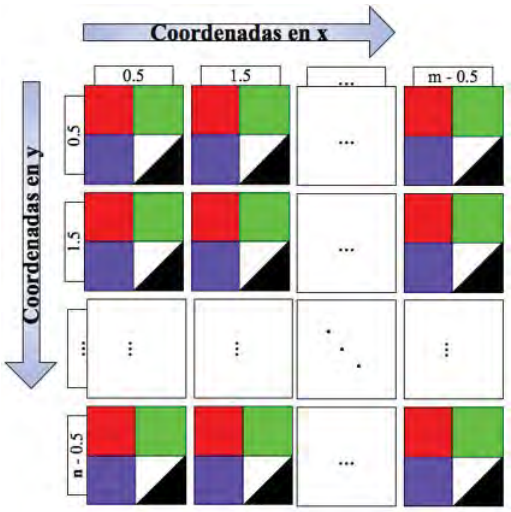



Figura 4.3: Sistema de coordenadas en una textura.

Dado que los datos necesarios para realizar la evaluación se encuentran en el espacio de memoria de una textura, sus valores pueden ser extraídos desde la función `fitness_function` simplemente conociendo las coordenadas a las que va a hacer referencia. Los datos en este caso se pueden extraer mediante la instrucción:

```
tex2D(texPopulation, x+0.5f, y+0.5f)
```

en donde `texPopulation` corresponde a la textura que almacena la población (ver Listado 4.2) y $x$ y $y$ representan las coordenadas. Ees necesario aplicar el desplazamiento de 0.5 en cada coordenada $(x, y)$ para de este modo poder hacer referencia a cada una de las componentes de un individuo determinado.

En el apéndice B se presenta el código completo en CUDA para realizar la operación de evaluación en paralelo.

## 4.4. Conclusiones

El ejemplo de implementación en la GPU presentado en este capítulo se describe con la intención de que pueda servir de referencia en implementaciones futuras basadas en GPUs. Se describe además el esquema de programación modular empleado para realizar las operaciones paralelizadas dentro del algoritmo paralelo de evolución diferencial. Este esquema ha sido descrito en el caso particular de la operación asociada a la evaluación de los individuos, en la cual se identifican tres etapas. La primera de estas etapas se refiere a la ejecución de la función principal (`main`), la cual representa el módulo principal en la programación del algoritmo de ED. En la segunda etapa se definieron los datos necesarios y se establecieron las condiciones necesarias para invocar el kernel de evaluación. Por último se tiene la implementación del kernel de cómputo correspondiente a la operación de evaluación. El kernel de evaluación se refiere a la función que es declarada para realizar la evaluación de la población de individuos al ser ejecutada en paralelo por todos los *threads*. La explotación de los recursos de memoria disponibles en los espacios de textura permiten agilizar la lectura de los datos, al encontrarse almacenados en caché, logrando de esta manera realizar los cálculos de manera más rápida y eficiente. El beneficio en el uso de los espacios de memoria de texturas en comparación con los espacios de memoria global aumenta con la latencia de

los accesos a memoria, es decir, entre más accesos se requiera para determinado conjunto de datos, el beneficio en el uso de texturas será mayor.

Capítulo 5

# Casos de estudio

Para evaluar la implementación del algoritmo de evolución diferencial en CUDA, se llevó acabo una serie de experimentos sobre cuatro casos de estudio de optimización sin restricciones. Estos casos involucran la optimización de cuatro funciones ampliamente utilizadas como pruebas de desempeño para algoritmos de optimización. Tres de estas funciones son continuas y diferenciables: la función de Griewank, la función de Rosenbrock y la función de Ackley [Price05]. Además, se incluye una función no diferenciable empleada como prueba en [Lee04, Veronese10, Zhu09], la cual se denominará como $f_4$ en el presente trabajo.

El desempeño de la implementación en la GPU desarrollada en este trabajo, la cual denominaremos como EDCP (Evolución Diferencial Completamente Paralela), es comparado con el desempeño de una variante en la cual se paraleliza exclusivamente la evaluación en la GPU. Además, se compara la EDCP con las versiones secuenciales y paralelas del mismo algoritmo, programadas en C en la CPU. Por último se hace la comparación con el esquema original de ED propuesto en [Storn95]. Los resultados están expresados, de acuerdo a la comparación, en términos de tiempos de ejecución, factores de aceleración y precisión de las soluciones. El factor de aceleración es obtenido al dividir un tiempo empleado como referencia $t_1$ entre un tiempo $t_2$ contra el cual se pretende comparar. Si el factor de aceleración resulta $\frac{t_1}{t_2} \leq 1$ significa que no se está obteniendo beneficio alguno puesto que $t_2 \geq t_1$.

Todos los experimentos han sido ejecutados 35 veces de acuerdo a la teoría ele-

mental del muestreo [Spiegel92], con el objetivo de validar la eficacia y eficiencia de las implementaciones reportando su desempeño promedio. El porcentaje de efectividad y la precisión promedio de las soluciones se obtuvieron estableciendo un error máximo permitido de $1 \times 10^{-10}$. Los tiempos de ejecución han sido obtenidos variando el tamaño de las poblaciones $N_p$ desde 2048 hasta 8192, estableciendo la dimensionalidad de los problemas en $D = 128$ y dominio de búsqueda en primera instancia como $\{j \in \mathbb{R}^D \mid -600 \leq j \leq 600\}$ a excepción de la función de Ackley, para la cual el dominio de búsqueda es $\{j \in \mathbb{R}^D \mid -32 \leq j \leq 32\}$. El número máximo de generaciones $g_{max}$ fue definido empíricamente para cada una de las 4 funciones de prueba empleadas.

Los dominios de búsqueda son establecidos únicamente en primera instancia, es decir, para generar la población inicial, y no se realizan procedimientos para limitar este espacio de búsqueda en futuras generaciones. Por esta razón, si se presentara el caso en que el óptimo global no se encontrara en estos dominios de búsqueda, el método puede conducir la búsqueda fuera de ellos y regresar el mínimo valor encontrado después de un número máximo de generaciones. Lo anterior dependerá de la topología de la función objetivo. Todos los parámetros fueron definidos basándose en los parámetros empleados en [Price05] [Veronese10] y [Zhu09], estableciendo una dimensionalidad mayor a la máxima empleada en estas referencias e igualando o incrementando el espacio de búsqueda.

Todos los experimentos han sido ejecutados en un CPU AMD Phenom II, 3.0 GHz quad-core con 3.9 GB de memoria física. La GPU utilizada para este trabajo de investigación es una tarjeta gráfica NVIDIA Tesla C2050, arquitectura Fermi con 3 GB de memoria. Esta GPU contiene 14 SMs, cada una con 32 SPs resultando en un total de 448 procesadores de flujo. La Tabla 5.1 resume las características y beneficios más importantes de la tarjeta gráfica usada en este trabajo.

Tabla 5.1: Características de la tarjeta GPU NVIDIA Tesla C2050.

| | |
|---|---|
| NÚMERO DE NÚCLEOS CUDA | 448 |
| FRECUENCIA DE LOS NÚCLEOS CUDA | 1.15 GHz. |
| PICO MÁXIMO DE RENDIMIENTO (doble precisión) | 515 Gflops/s |
| PICO MÁXIMO DE RENDIMIENTO (precisión sencilla) | 1.03 Tflops/s |
| MEMORIA TOTAL DEDICADA | 3 GB GDDR5. |
| VELOCIDAD DE MEMORIA | 1.5 GHz. |
| ANCHO DE BANDA DE LA MEMORIA | 144 GB/sec. |
| MEMORIA ECC | Protección de los datos en la memoria, manteniendo su integridad y aumentando su confiabilidad. Registros, caché L1/L2, memoria compartida y DRAM protegidos por códigos de corrección de errores. |
| CONSUMO MÁXIMO DE POTENCIA | 283 W. |
| *NVIDIA PARALLEL DATACACHE*$^{TM}$ | Caché L1 configurable por SM y caché L2 unificado para todos los núcleos de procesamiento. |
| TRANSFERENCIA ASÍNCRONA DE DATOS | El rendimiento del sistema es acelerado mediante la transferencia de datos a través del bus PCIe, mientras los núcleos de procesamiento realizan el cómputo de otros datos. |
| LENGUAJES Y APIs SOPORTADOS | C, C++, OpenCL, DirectCompute y Fortran. Herramienta *NVIDIA Parallel Nsight* disponible para desarrolladores de *Microsoft Visual Studio*. |
| INTERFAZ DEL SISTEMA | PCIe x16 2.0. |

## 5.1. Comparación entre la paralelización completa del algoritmo de ED y la paralelización exclusiva de la operación de evaluación en la GPU

La principal propuesta de este trabajo es la paralelización completa de todas las operaciones evolutivas en el algoritmo de ED, a través de la arquitectura masivamente paralela de las GPUs. En la presente sección se desarrolla la comparación en cuanto a desempeño entre el la versión denominada como EDCP y una variante, la cual implementa únicamente

la operación de la evaluación de manera paralela en la GPU. Los resultados obtenidos de esta comparación se describen a continuación en términos de tiempos de ejecución y factores de aceleración obtenidos por parte de la versión más rápida para las cuatro funciones establecidas como prueba de rendimiento.

### 5.1.1. La función de Griewank

La función de Griewank está definida por

$$f(\mathbf{x}) = \sum_{j=1}^{D} \frac{x_j^2}{4000} - \prod_{j=1}^{D} \cos\left(\frac{x_j}{\sqrt{j}}\right) + 1 \tag{5.1}$$

la cual se trata de una función continua y diferenciable.

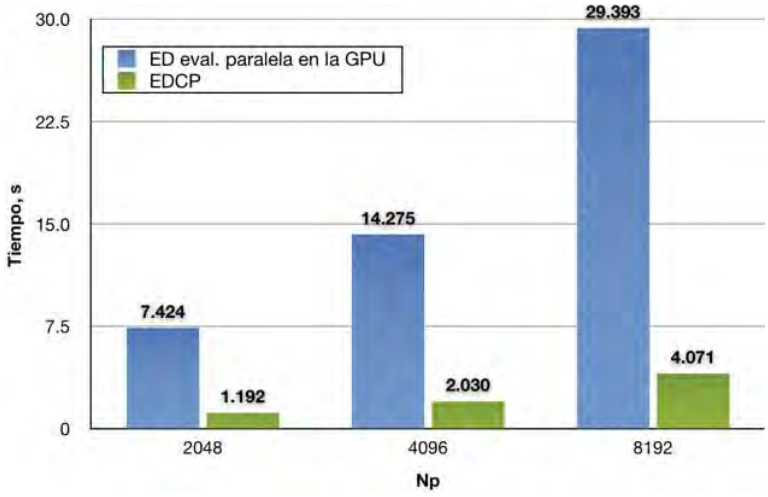
La comparación entre las dos versiones del método de ED propuesto, involucradas en la presente sección, se realiza estableciendo un número de generaciones $g_{max} = 1000$ y tamaños de población que van desde $N_p = 2048$ hasta $N_p = 8192$. La Figura 5.1(a) muestra los tiempos de cómputo promedio demandados por parte de ambas versiones, al emplear la función de Griewank como función objetivo. De esta figura se puede observar que la versión completamente paralelizada presenta un mejor desempeño, obteniendo tiempos de ejecución de tan sólo 1.192, 2.030 y 4.071 segundos, mientras que para la versión en la que sólo se paraleliza la evaluación se obtiene 7.424, 14.275 y 29.393 segundos para $N_p$ =2048, 4096 y 8192, respectivamente.

Por su parte, la Figura 5.1(b) muestra los factores de aceleración obtenidos por parte de versión EDCP sobre la versión en la que sólo se paraleliza la evaluación en la GPU. De los datos reportados en esta gráfica se puede observar un factor de aceleración creciente al aumentar el valor de $N_p$ obteniendo un factor de aceleración máximo igual a 7.22 cuando el tamaño de población es 8192.

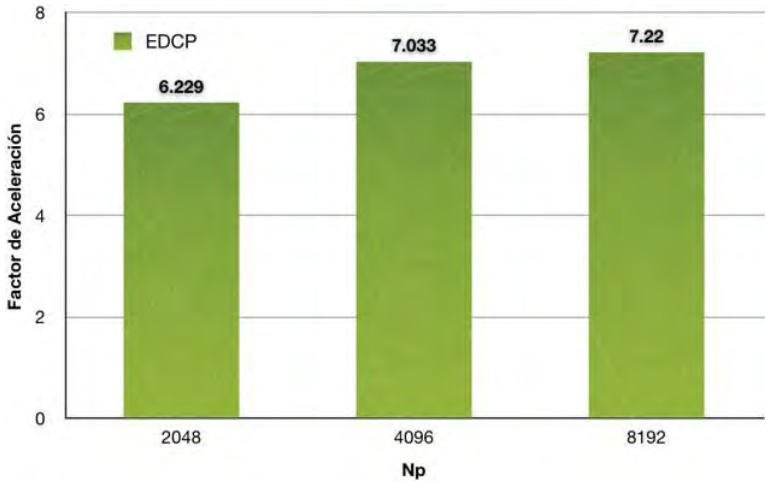
### 5.1.2. La función de Rosenbrock

La función de Rosenbrock es una función continua, diferenciable y no convexa, la cual está definida por

$$f(X) = \sum_{j=1}^{D-1} (1 - x_j)^2 + 100(x_{j+1} - x_j^2)^2 \tag{5.2}$$



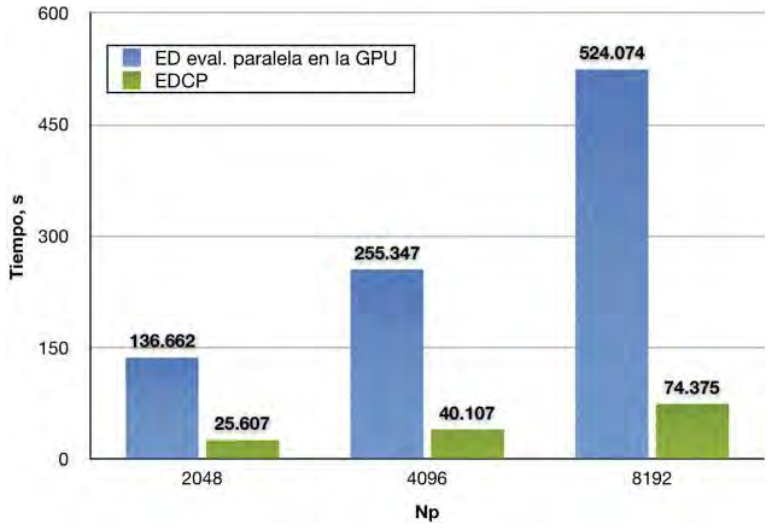

(a)



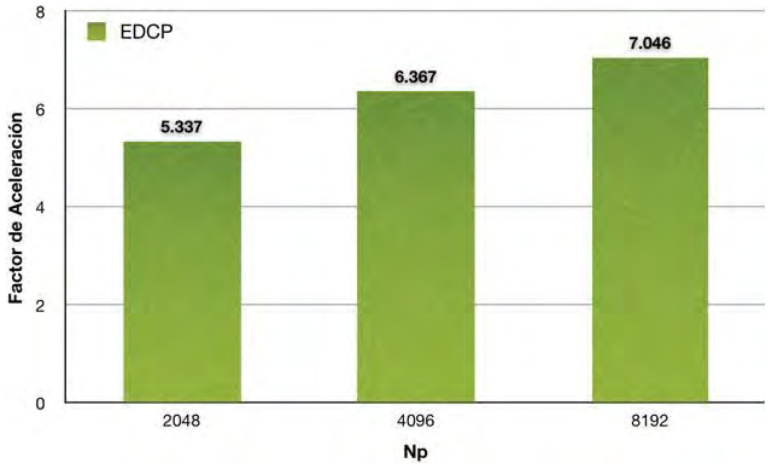

(b)

Figura 5.1: Esfuerzo computacional total en términos de (a).- tiempos de cómputo y (b).- factores de aceleración para la función de Griewank.

Al emplear la función de Rosenbrock para comparar el rendimiento de ambas versiones del algoritmo de ED propuesto, el número máximo de generaciones se estableció como $g_{max} = 18000$. El tiempo total promedio requerido para la ejecución de la versión denominada EDCP y la versión del algoritmo en la cual sólo se paraleliza la evaluación se muestra




**(a)**




**(b)**

Figura 5.2: Esfuerzo computacional total en términos de (a).- tiempos de cómputo y (b).- factores de aceleración para la función de Rosenbrock.

en la Figura 5.2(a). Los tiempos de ejecución al paralelizar sólo la evaluación son 136.662, 255.347 y 524.074 segundos para tamaños de población $N_p$ =2048, 4096 y 8192, respectivamente. Por su parte, la versión EDCP requiere tiempos de ejecución menores los cuales son 25.607, 40.107 y 74.375 segundos para los mismos tamaños de población.

En lo que respecta a factores de aceleración, la Figura 5.2(b) presenta los resultados obtenidos por la versión EDCP. Estos factores de aceleración corresponden a 5.337, 6.367 y 7.046 para $N_p$ =2048, 4096, y 8192, respectivamente.

### 5.1.3. La función de Ackley

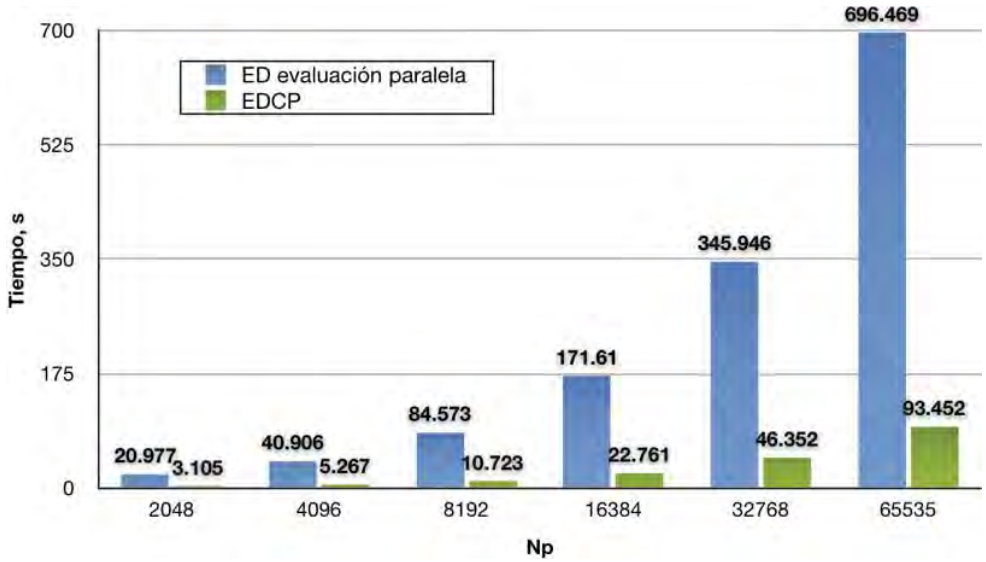
La función de Ackley está definida por

$$f(\mathbf{x}) = 20 + \mathbf{e}^{-0.2\sqrt{\frac{1}{D}\cdot\sum_{j=1}^{D} x_j^2}} - \mathbf{e}^{\frac{1}{D}\cdot\sum_{j=1}^{D}\cos(2\pi\cdot x_j)} + 20 + \mathbf{e} \tag{5.3}$$

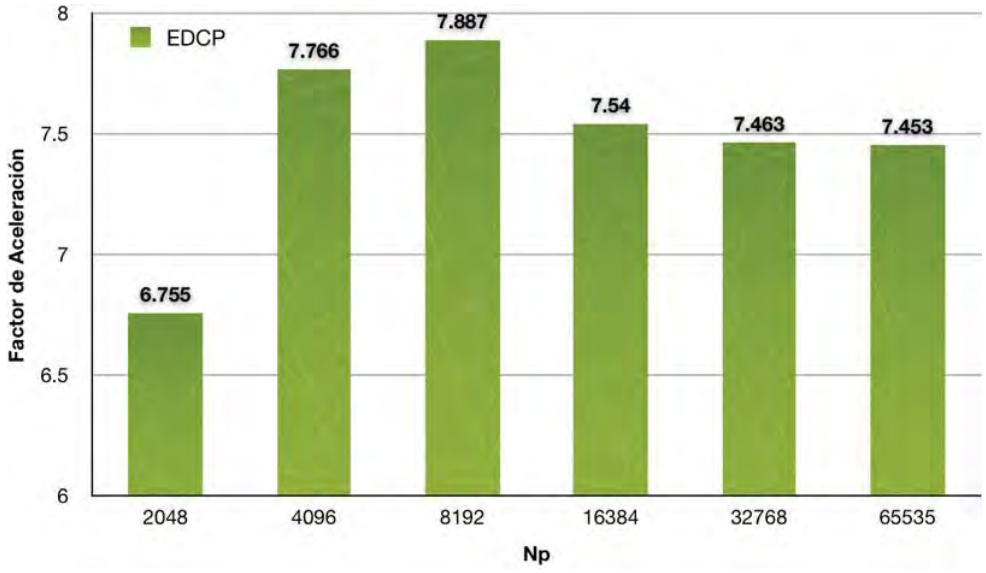
en donde $\boldsymbol{e}$ representa la constante de Euler y D representa el numero de dimensiones en el que se expresa la función. Para el caso de la función de Ackley el espacio de búsqueda fue reducido a $\{j \in \mathbb{R}^D \mid -32 \leq j \leq 32\}$ de acuerdo a [Price05]. Esta función resulta particularmente difícil de minimizar aplicando la presente propuesta de ED, considerando una tolerancia máxima al error de $1\times10^{-10}$. Este análisis se presenta en la siguiente sección y a partir del mismo se decide realizar las pruebas de rendimiento, empleando esta función, con tamaños de población iguales a 16384, 32768 y 65535, además de los tamaños de población establecidos previamente para el resto de las funciones.

El tiempo promedio requerido para la ejecución del algoritmo de ED propuesto por parte de la versión EDCP así como la versión en la cual sólo se paraleliza la evaluación, se muestra en la Figura 5.3(a), al emplear la función de Ackley como función de prueba y un número máximo de generaciones $g_{max} = 3000$. En esta figura se puede apreciar que la versión EDCP presenta tiempos de ejecución menores, los cuales son iguales a 3.105, 5.267, 10.723, 22.761, 46.352 y 93.452 segundos, mientras que paralelizando solamente la operación de la evaluación en la GPU se obtienen 20.977, 40.906, 84.573, 171.61, 345.946 y 696.469 segundos para tamaños de población que van desde $N_p$ =2048 hasta $N_p$= 65535, respectivamente.

En lo que respecta a los factores de aceleración, la Figura 5.3(b) muestra el beneficio obtenido por parte de la versión EDCP. Es posible apreciar que se obtienen factores de aceleración mayores o iguales a 6.755. De esta figura se puede observar que el mayor beneficio se obtiene cuando $N_p = 8192$.




**(a)**




**(b)**

Figura 5.3: Esfuerzo computacional total en términos de (a).- tiempos de cómputo y (b).- factores de aceleración para la función de Ackley.

Para explicar el comportamiento del beneficio obtenido es necesario dar un vistazo más de cerca a la arquitectura de las GPUs. El sistema de cómputo de una GPU incluye varias unidades SM, las cuales comparten la memoria global que se destina para texturas y la memoria caché de nivel 2 (L2 *cache*). A su vez, una unidad SM tiene varias unidades (SPs) las cuales comparten acceso a una parte de la memoria caché de nivel 1 (L1 *caché*). De este modo, los recursos de memoria siempre son compartidos por parte de las unidades de procesamiento a distintos niveles. Es decir, tanto los SPs dentro de un SM como los SM dentro de la GPU tienen acceso a los recursos de memoria utilizando los mismos medios a través de un controlador de dichos recursos compartidos [Soveiko10].

La arquitectura de la GPU proporciona entonces un acceso muy sencillo a cantidades considerables de memoria. Sin embargo, esto a su vez puede convertirse en una gran desventaja. Por ejemplo, cuando todas las unidades SM intentan acceder a los espacios de memoria que comparten a través del mismo bus/controlador y, de manera similar, cuando todas las unidades SP intentan acceder a los espacios de memoria compartida dentro de las unidades SM se pueden producir cuellos de botella.

El problema de los cuellos de botella puede ser mitigado cuando las unidades de procesamiento poseen grandes cachés. Sin embargo, esto se reduce sólo hasta cierto punto para un problema como el que se está abordando en esta tesis, considerando que aún la memoría caché de más bajo nivel es un recurso compartido. En el caso de la versión del algoritmo de ED propuesto denominada EDCP, esto se observa en una disminución en la eficiencia para tamaños de poblaciones grandes en donde los fallos de caché se vuelven más frecuentes y el ancho de banda del bus de memoria se convierte en un factor limitante. El cuello de botella del ancho de banda de memoria se vuelve más pronunciado en alta concurrencia aumentando la latencia de los recursos de memoria, debido a que tanto las unidades SM como las unidades SP pierden tiempo compitiendo unos con otros al intentar acceder a estos recursos.

En [Soveiko10] se presenta un análisis más detallado sobre el desempeño de una arquitectura como la de las GPUs, al establecer un número fijo de unidades de procesamiento y variando la dimensionalidad del problema. Los resultados en términos del beneficio obtenido en la referencia mencionada presentan un comportamiento similar al mostrado en la

Figura 5.3(b).

### 5.1.4. La función $f_4$

La función denominada en este trabajo como $f_4$ (correspondiente a la función $f_6$ en [Veronese10] y a la función $f_9$ en [Lee04]) está definida por,

$$f(\mathbf{x}) = 0.1 \left\{ \sin^2(3\pi x_1) + \sum_{j=1}^{D-1} (x_i - 1^2) \left[1 + 10 \sin^2(3\pi x_{i+1})\right] + (x_D - 1)^2 \left[1 + \sin^2(2\pi x_D)\right] \right\} + \sum_{j=1}^{D} u(x_i, 5, 100, 4) \tag{5.4}$$

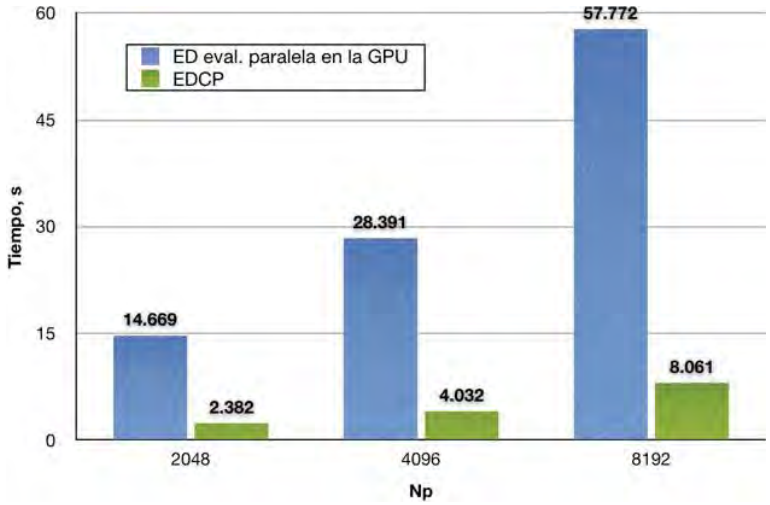
en donde,

$$u(x, a, k, m) = \begin{cases} k \cdot (x - a)^m, & x > a \\ 0, & -a \leq x \leq a \\ k \cdot (-x - a)^m, & x < -a \end{cases} \tag{5.5}$$

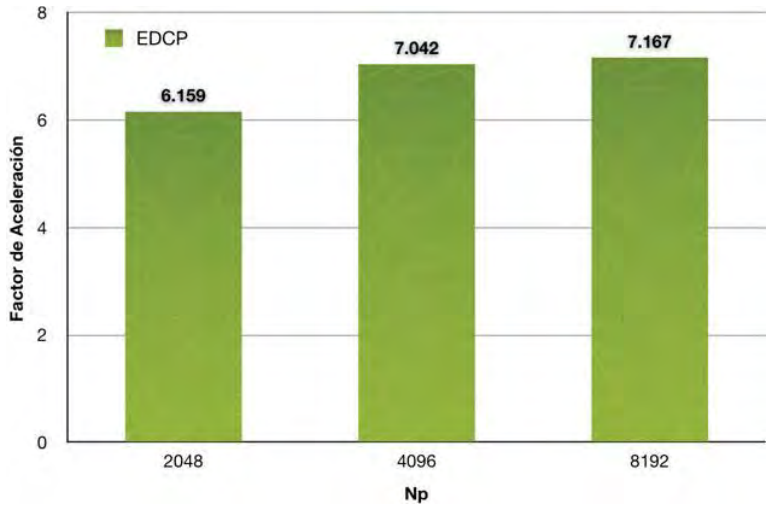
Para el caso de la función $f_4$ el número máximo de generaciones se estableció como $g_{max} = 2000$.

En la Figura 5.4(a) es posible apreciar un mejor desempeño por parte de la versión EDCP en todo momento, para los parámetros de ejecución establecidos. Los tiempos de ejecución obtenidos por EDCP son iguales a 2.382, 4.032 y 8.061 segundos en comparació con resultados de 14.669, 28.391 y 57.772 segundos para $N_p$ =2048, 4096 y 8192, respectivamente.

Por su parte, la Figura 5.4(b) muestra los factores de aceleración obtenidos por parte de versión EDCP. De los datos presentados en esta gráfica se puede observar un factor de aceleración máximo obtenido de 7.167 para $N_p = 8192$, justificando de esta manera el esquema de completa paralelización del algoritmo de ED.




(a)




(b)

Figura 5.4: Esfuerzo computacional total en términos de (a).- tiempos de cómputo y (b).- factores de aceleración, para la función $f4$.

## 5.2. Comparación en el desempeño entre CPU y GPU en la ejecución del algoritmo de ED propuesto

La razón principal detrás de una implementación paralela de un AE es reducir el esfuerzo computacional requerido para encontrar una solución. Sin embargo, los AEs parale-

los se han enfocado principalmente en paralelizar la operación de la evaluación, lo cual afecta directamente el desempeño del algoritmo en términos de eficiencia y consecuentemente se ve afectada la velocidad de ejecución. En esta sección se presentan los resultados obtenidos a partir de la comparación en cuanto al desempeño de la versión EDCP y dos versiones implementadas para ejecutarse en la CPU del algoritmo de ED propuesto en diferentes etapas. La primera de las implementaciones en la CPU corresponde a la versión secuencial del algoritmo, la cual además servirá como referencia para evaluar el desempeño de las otras versiones. La segunda versión en la CPU corresponde a una versión paralela del algoritmo en la cual, debido a las limitantes del lenguaje C de programación, se paraleliza únicamente la operación de evaluación, utilizando para esto cuatro núcleos de procesamiento.

### 5.2.1. Implementación de la tarea de creación de población inicial y mutación en la GPU

La primer etapa del algoritmo que se analiza es la creación de la población inicial. A continuación se describen los resultados obtenidos en términos de tiempos de cómputo y factores de aceleración para realizar las operaciones de la creación de la poblacion inicial y la mutación de manera paralelizada en la GPU en comparación con las mismas operaciones implementadas secuencialmente en la CPU. Los resultados reportados en las Tablas 5.2 y 5.3 representan el desempeño general de las implementaciones durante las etapas de la creación de la población inicial y la mutación para los cuatro casos de estudo: la función de Griewank, la función de Rosenbrock, la función de Ackley y la función $f_4$. El desempeño es reportado una sola vez debido a que estas operaciones son realizadas bajo las mismas circunstancias en todos los casos.

#### Creación de la población inicial

A pesar de no tratarse de una tarea tan demandante como lo es la evaluación de los individuos, la creación de la población inicial es una etapa importante en todo algoritmo evolutivo. Es importante garantizar que dentro de la población inicial, se tenga la diversidad estructural de estas soluciones para tener una representación de la mayor parte de la posible población. Una generación eficaz de la población inicial deberá proporcionar la diversidad

Tabla 5.2: Tiempos de ejecución y factores de aceleración durante la creación de la población inicial

| $N_p$ | Versión | Tiempo de cómputo, ms | Factor de aceleración |
|---|---|---|---|
| 2048 | C-secuencial | 4.7957 | |
| | C-paralelo | 4.7812 | 1.0030 |
| | EDCP | 2.1519 | 2.2285 |
| 4096 | C-secuencial | 9.5986 | |
| | C-paralelo | 9.5814 | 1.0018 |
| | EDCP | 3.0754 | 3.1210 |
| 8192 | C-sequencial | 19.7551 | |
| | C-paralelo | 19.7490 | 1.0003 |
| | EDCP | 4.8760 | 4.0515 |
| 16384 | C-secuencial | 40.1561 | |
| | C-paralelo | 39.7606 | 1.0099 |
| | EDCP | 9.1651 | **4.3814** |
| 32768 | C-secuencial | 80.4419 | |
| | C-paralelo | 79.5416 | 1.0113 |
| | EDCP | 21.2692 | 3.7821 |
| 65535 | C-sequencial | 154.7046 | |
| | C-paralelo | 155.0261 | 0.9979 |
| | EDCP | 49.2569 | 3.1408 |

requerida por estos métodos estocásticos.

La creación de la población inicial es la primera de las operaciones que son susceptibles de ser paralelizadas dentro de un AE, considerando que cada individuo puede ser creado de manera independiente. Sin embargo, debido a las limitantes en la generación de números aleatorios por parte del lenguaje C, esta operación ha tenido que ser implementada de manera secuencial en la CPU. Los tiempos de cómputo y factores de aceleración obtenidos en la realización de esta operación por las diferentes versiones (C-secuencial, C-paralelo y C-CUDA) implementadas del algoritmo de ED se resumen en la Tabla 5.2. Se puede apreciar que el programa en CUDA proporciona el mayor beneficio para $N_p = 16384$, obteniendo un factor de aceleración igual a 4.3814 con respecto a las versiones en C. Las dos versiones en la CPU muestran un comportamiento similar debido a que ambas utilizan la misma función para realizar la generación de la población inicial.

A pesar de que la creación de la población inicial representa sólo una mínima parte

del esfuerzo de cómputo total requerido por el algoritmo de ED, la paralelización de esta operación en la GPU mostró tener un beneficio en todo momento. La realización en paralelo de esta operación resulta de utilidad para evaluar el desempeño de la librería CURAND, empleada en la versión EDCP para la generación de números aleatorios bajo un esquema multi-hilo en la GPU. Cabe señalar que se trata de una librería relativamente nueva de la cual aún no se encuentra reportado su desempeño en la literatura.

**Mutación**

Debido a que las variaciones que sufren los individuos durante el proceso de mutación tienen una componente aleatoria, esta operación está limitada a realizarse de manera secuencial para las versiones en la CPU. Por su parte, la versión EDCP es capaz de realizar dicha operación en paralelo para cada individuo debido a las facilidades provistas por la librería CURAND. Los resultados en términos de tiempos de ejecución y factores de aceleración promedio obtenidos por generación se describen en la Tabla 5.3, para tamaños de población que van desde 2048 hasta 65535.

La Tabla 5.3 presenta resultados interesantes obtenidos por la versión EDCP en relación a las versiones en código C. Se puede observar que la versión C-paralela resulta ser más lenta que la versión secuencial. La razón detrás de este hecho se asocia a la mayor complejidad en las ejecución de un programa multi-hilo. La programación multi-hilo implica un concepto denominado *context switch*, el cual se refiere a cuando dos *threads* son intercambiados para realizar la ejecución del código secuencial de una aplicación. Estos cambios de contexto resultan ser lentos y costosos [NVIDIA10b]. Por su parte, la versión denominada EDCP mostró un desempeño mejor en términos de tiempos de ejecución, obteniendo el mayor beneficio cuando $N_p = 4092$ con un factor de aceleración igual a 22.6947 sobre la versión secuencial.

De los datos presentados en la Tabla 5.3, se puede observar un comportamiento decreciente en el beneficio para $N_p > 4096$, para explicar dicho comportamiento es necesario considerar el esquema de mutación empleado. Durante la etapa de mutación son requeridos los datos asociados a la mejor solución encontrada hasta el momento. La mejor solución es almacenados en la memoria constante de la GPU y ésta necesita ser leída concurrentemente

Tabla 5.3: Tiempos de ejecución y factores de aceleración durante la mutación

| $N_p$ | Versión | Tiempo de cómputo (ms) | Factor de aceleración |
|---|---|---|---|
| 2048 | C-secuencial | 5.6852 | |
| | C-paralelo | 7.7693 | 0.7317 |
| | EDCP | 0.3031 | 18.7563 |
| 4096 | C-sequencial | 11.6173 | |
| | C-paralela | 20.5160 | 0.5663 |
| | EDCP | 0.5119 | **22.6947** |
| 8192 | C-sequencial | 24.6343 | |
| | C-paralela | 40.4510 | 0.6090 |
| | EDCP | 1.2953 | 19.0177 |
| 16384 | C-secuencial | 50.5893 | |
| | C-paralelo | 74.3736 | 0.6802 |
| | EDCP | 3.2385 | 15.6213 |
| 32768 | C-sequencial | 103.4750 | |
| | C-paralela | 134.7452 | 0.7679 |
| | EDCP | 6.9686 | 14.8488 |
| 65535 | C-sequencial | 212.5464 | |
| | C-paralela | 260.3565 | 0.816366681 |
| | EDCP | 14.4672 | 14.6916 |

por todos los *threads*. Dado que se trata de un recurso compartido y que la concurrencia crece al incrementarse el tamaño de la poblaciones, se producen cuellos de botella cada vez más mayores [Soveiko10], produciendo un aumento en la latencia de la memoria constante y disminuyendo el beneficio de la implementación en al GPU.

**Selección**

De acuerdo con el esquema propuesto de ED, el cual involucra cierto nivel de voracidad, la operación de selección es llevada a cabo sin considerar una fase previa de cruza. De esta manera, el $i$-ésimo mutante se compara directamente con el $i$-ésimo miembro de la población en cada generación, selecionando la mejor de estas dos solucines para formar parte de la siguiente generación. Considerando que se trata del proceso de minimización de $f$, esto es,

$$\mathbf{x}_i^{g+1} = \begin{cases} \mathbf{v}_i^{(g)} & \text{if } f(\mathbf{v}_i^{(g)}) < f(\mathbf{x}_i^{(g)}) \\ \mathbf{x}_i^{(g)} & otherwise \end{cases} \tag{5.6}$$

La operación de selección es una tarea relativamente sencilla en comparación con el resto de las operaciones evolutivas. En la Tabla 5.4 se presentan los resultados asociados a los tiempos de ejecución y factores de aceleración obtenidos durante una generación, en esta etapa del algoritmo de ED.

Tabla 5.4: Tiempos de ejecución y factores de aceleración durante la selección

| $N_p$ | Versión | Tiempo de cómputo (ms) | Factor de aceleración |
|---|---|---|---|
| 2048 | C-secuencial | 0.003172 | |
| | C-paralelo | 0.051424 | 0.072184194 |
| | EDCP | 0.0442632 | 0.08386198919 |
| 4096 | C-sequencial | 0.005824 | |
| | C-paralela | 0.092320 | 0.063084922 |
| | EDCP | 0.079036 | 0.07368793967 |
| 8192 | C-sequencial | 0.017472 | |
| | C-paralela | 0.100992 | 0.173003802 |
| | EDCP | 0.1498672 | 0.11658321501 |
| 16384 | C-secuencial | 0.0357288 | |
| | C-paralelo | 0.133248 | 0.268137608 |
| | EDCP | 0.2903336 | 0.12306119581 |
| 32768 | C-sequencial | 0.050720 | |
| | C-paralela | 0.145600 | 0.348351648 |
| | EDCP | 0.5429928 | 0.09340823672 |
| 65535 | C-sequencial | 0.096640 | |
| | C-paralela | 0.157344 | 0.614195648 |
| | EDCP | 1.0645952 | 0.09077628755 |

Como se puede observar en la Tabla 5.4, la versión secuencial presenta los mejores tiempos de ejecución. Este hecho sugiere que la poca complejidad de la operación de selección permite que esta operación se realice de manera secuencial en un tiempo menor al requerido para establecer las condiciones necesarias para realizarla de manera paralela tanto en la CPU como en la GPU.

### 5.2.2. La funcion de Griewank

En el caso de la función de Griewank, Eq. (5.1), el término sumatorio de la expresión crea una cuenca parabólica mientras que el producto de los cosenos genera una enorme cantidad de óptimos locales a medida que aumenta la dimensionalidad y el dominio de búsqueda. El mínimo global de esta función está localizado en el origen el cual es $f(\mathbf{x}^*) = 0$. Minimizar esta función empleando un método determinístico de optimización, tal como un método basado en gradiente, requeriría un enorme esfuerzo computacional, mientras que para un AE es común que se requiera una población relativamente grande para evitar la convergencia prematura [Price05]. El espacio de búsqueda para este caso de estudio se ha establecido como $\{j \in \mathbb{R}^D \mid -600 \leq j \leq 600\}$. La Figura 5.5 muestra la representación en 3D de esta función de Griewank, para distintos dominios.

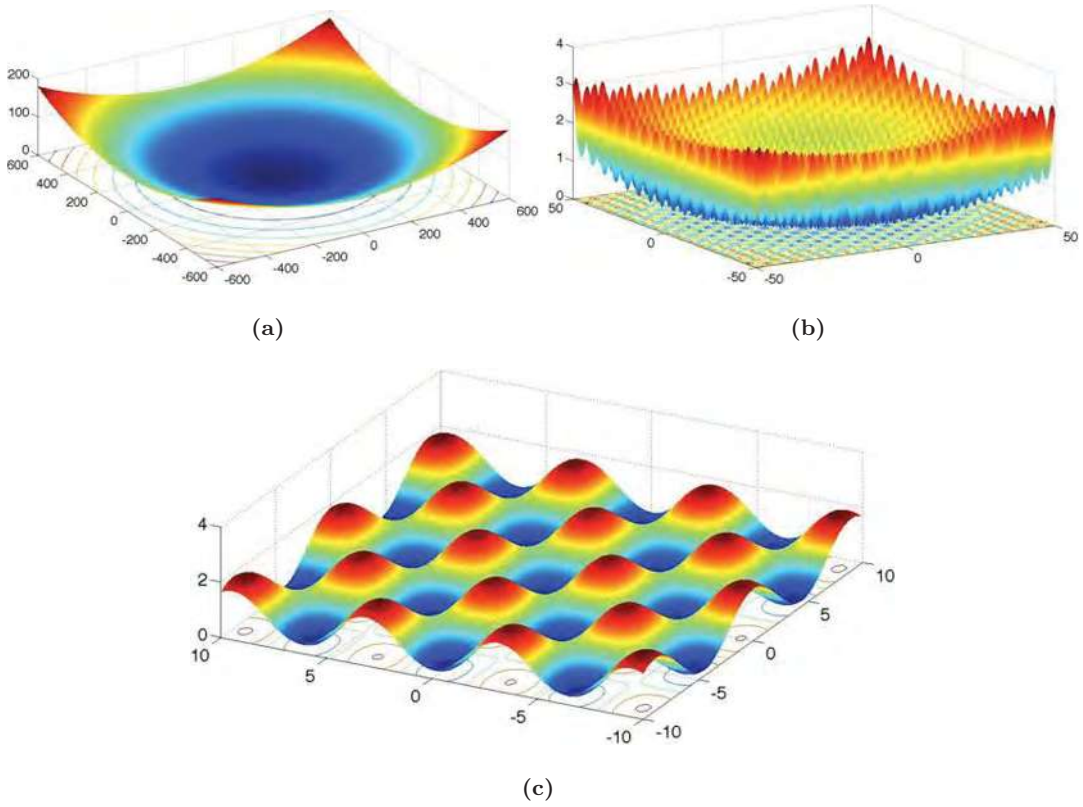
Los resultados obtenidos tras minimizar la función de Griewank aplicando el algoritmo de evolución diferencial paralelo implementado en esta tesis son presentados en términos de eficacia, precisión, tiempos de ejecución y factores de aceleración.

#### Tiempos de ejecución y factores de aceleración

La Figura 5.6(a) muestra los tiempos de cómputo promedio demandados para realizar la etapa de evaluación de la función de Griewank durante una generación, para tamaños de población desde $N_p = 2048$ hasta $N_p = 8192$. En esta figura se puede ver que la versión secuencial muestra un crecimiento más pronunciado en los tiempos de cómputo, en comparación con las versiones paralelas al incrementar $N_p$ desde 2048 hasta 8192. La versión paralela en C proporciona una importante reducción en el esfuerzo computacional, sin embargo, la versión EDCP desarrollada en CUDA prueba ser la solución más eficiente con tan sólo 0.600, 1.160 y 2.285 millisegundos para $N_p$ =2048, 4096 y 8192, respectivamente.

Por su parte, la Figura 5.6(b) muestra los factores de aceleración medidos en base al desempeño del código secuencial en C. Resulta evidente que la versión EDCP basada en GPU proporciona un rendimiento sobresaliente con un factor de aceleración máximo de 40.232 para $N_p = 8192$, comparado con sólo 2.604 obtenido por la versión paralela en C.

El esfuerzo de cómputo total necesario para la minimización de la función de




**(a)** **(b)**

**(c)**

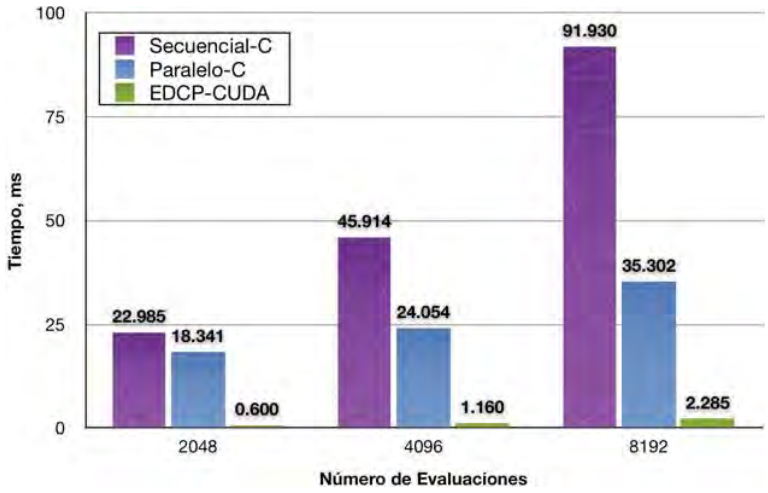
Figura 5.5: Función de Griewank en 3D en los dominios (a).- $\{j \in \mathbb{R}^2 \mid -600 \leq j \leq 600\}$, (b).- $\{j \in \mathbb{R}^2 \mid -50 \leq j \leq 50\}$ y (c).- $\{j \in \mathbb{R}^2 \mid -10 \leq j \leq 10\}$

.

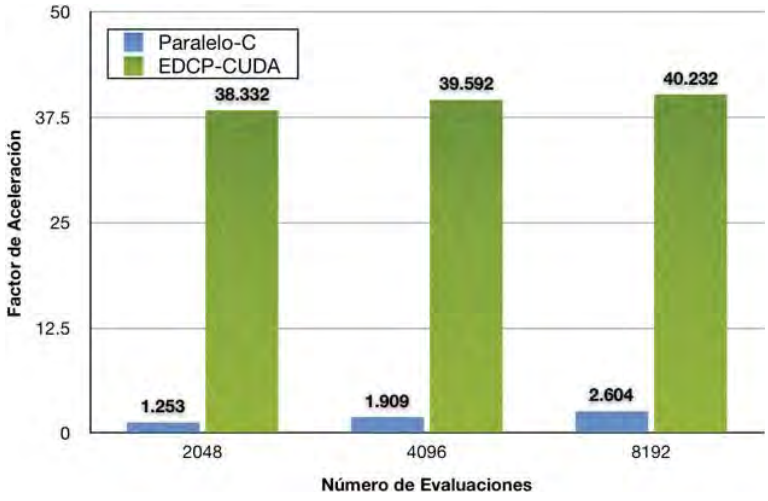
Griewank es mostrado en la Figura 5.7(a). De esta figura se puede apreciar que el esfuerzo computacional se reduce considerablemente mediante la implementación basada en GPU. El tiempo de cómputo de la versión en CUDA muestra un crecimiento lento y lineal al incrementarse el número de evaluaciones totales. El número de evaluaciones totales $N_{eval}$ se calcula como

$$N_{eval} = N_p + N_{mut} * g_{max} \tag{5.7}$$

en donde $N_{mut}$ representa el número de mutantes generados en cada generación. Por su parte, las versiones en C (secuencial y paralela) muestran un crecimiento con una pendiente mayor, siendo la mayor de estas pendientes la correspondiente a la versión secuencial.



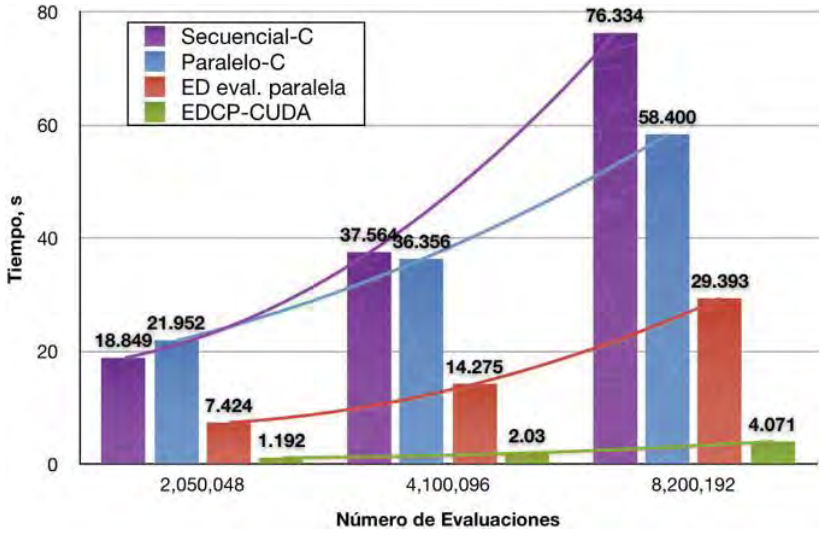

**(a)**



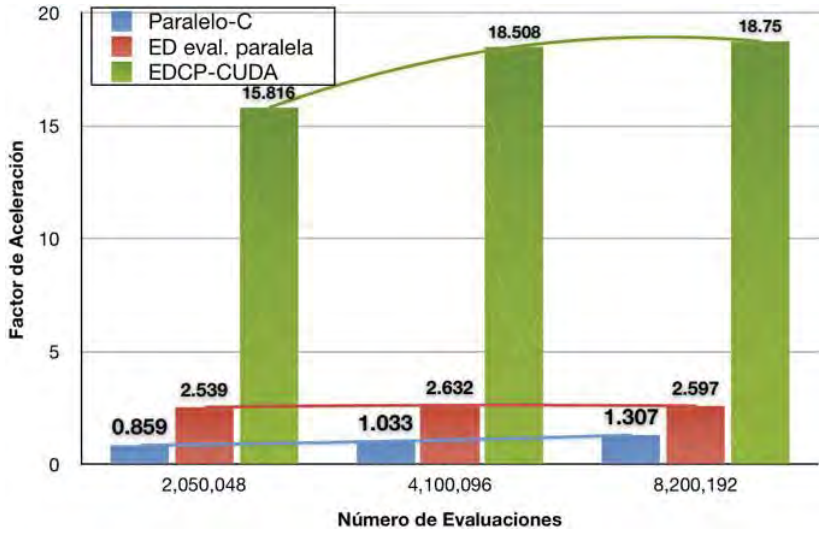

**(b)**

Figura 5.6: Resultados en la etapa de evaluación en términos de (a).- tiempos de cómputo y (b).- factores de aceleración, para la función de Griewank.

En lo que respecta al factor de aceleración total del algoritmo de ED paralelizado, la Figura 5.7(b) muestra los resultados producidos por los códigos paralelos en C y CUDA, con respecto del código secuencial en C. Se puede apreciar que la versión EDCP muestra el mejor rendimiento con factores de aceleración iguales a 15.816, 18.508 y 18.750 para números




**(a)**




**(b)**

Figura 5.7: Esfuerzo computacional total en términos de (a).- tiempos de cómputo y (b).- factores de aceleración para la función de Griewank.

de evaluaciones totales iguales a 2,050,048, 4,100,096 y 8,200,192, respectivamente.

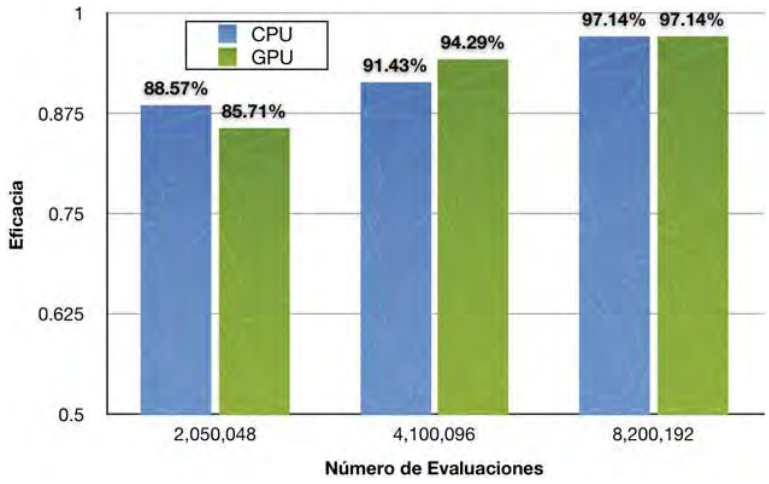



Figura 5.8: Eficacia en la minimización de la función de Griewank para números de evaluaciones correspondientes a $N_p = \{2048, 4096, 8192\}$.

**Eficacia**

La eficacia en un AE es la capacidad de alcanzar su objetivo después de cierto número de generaciones o evaluaciones. Es posible mejorar la eficacia en un AE, en primera instancia, a través del incremento en el tamaño de la población o estableciendo un número máximo de generaciones de evaluaciónes mayor. En el algoritmo de ED propuesto se implementa una variación en la estrategia de búsqueda. Esta variación involucra cierto nivel de voracidad, incorporando a la etapa de mutación información útil acerca de la mejor solución encontrada en cada iteración. La variación implementada se basa en el segundo esquema de ED presentado en [Storn95].

Para obtener la eficacia del algoritmo de ED propuesto, expresado en porciento, se divide la cantidad de veces que el algoritmo cumple su objetivo entre el número de veces que éste es ejecutado, siendo 35 veces en total para cada caso de estudio. El objetivo predefinido del algoritmo de ED es obtener un error máximo de $1 \times 10^{-10}$. La Figura 5.8 muestra la eficacia del algoritmo de ED para lograr dicho objetivo en el caso de la función de Griewank, en cuyo caso se establece un número máximo de generaciones $g_{max} = 1000$.

De la Figura 5.8 se puede apreciar que la eficacia del método aumenta al incrementar el número de evaluaciones, tanto para la versión en la CPU como en la GPU. En

el caso de la CPU, se obtienen eficacias de 88.57, 91.43 y 97.14 %, mientras que para las versiones en la GPU el 85.71, 94.29 y 97.14 % de las veces se cumple con el objetivo. Estos resultados se obtuvieron para números de evaluaciones totales iguales a 2050048, 4100096 y 8299192, respectivamente.

**Precisión**

Una de las desventajas más mencionadas de algunos AEs es la falta de precisión en las soluciones, especialmente cuando se trata de optimizar funciones continuas. Para el caso de la minimización de funciones continuas y diferenciables los métodos determinísticos presentan un nivel más alto de exactitud que los AEs, además de que los métodos evolutivos en general han probado requerir mucho más tiempo para converger que un método determinístico. El método de evolución diferencial, representa una alternativa a los AEs comunes. La ED ha demostrado ser un procedimiento heurístico con un alto nivel de precisión en las soluciones [Price05] y requrimientos de tamaños de población pequeños en proporción al tamaño del espacio de búsqueda, en comparación con AGs.

Los errores promedio de las 35 ejecuciones del algoritmo de ED propuesto para la minimización de la función de Griewank son presentados en la Figura 5.9 para las versiones en la CPU y GPU, considerando $g_{max} = 1000$ y $N_p$ =2048, 4096 y 8192. El error *err* se calcula a partir del hecho de que se conoce el mínimo global de las cuatro funciones de prueba utilizadas. Para estas cuatro funciones se tiene que el mínimo global es $f(\mathbf{x}^*) = 0$. Tal como se menciona previamente, se espera que las posibilidades de encontrar el mínimo global se incrementen al aumentar el tamaño de la población. Consistentemente con esto, se puede apreciar una mejora en la precisión de las soluciones obtenidas a medida que aumenta el número de evaluaciones totales, obteniendo errores promedio que van desde 0.000986 hasta 0.000352 para la versión en la CPU. Por su parte, la versión en la GPU presenta errores promedio que van desde 0.000986 hasta 0.000211 al aumentar $N_p$ desde 2048 hasta 8192.

De acuerdo a la Figura 5.9, la versión en la GPU presenta, en general, una precisión superior a la obtenida por el mismo algoritmo ejecutado en la CPU. La diferencia de precisión obtenida por parte ambas arquitecturas de cómputo plantea la hipótesis de que el compilador de CUDA presenta un mejor desempeño asociado a los errores de redondeo

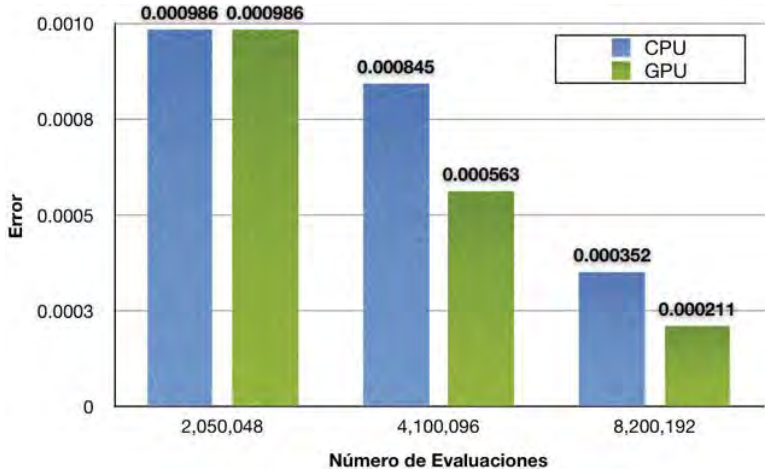
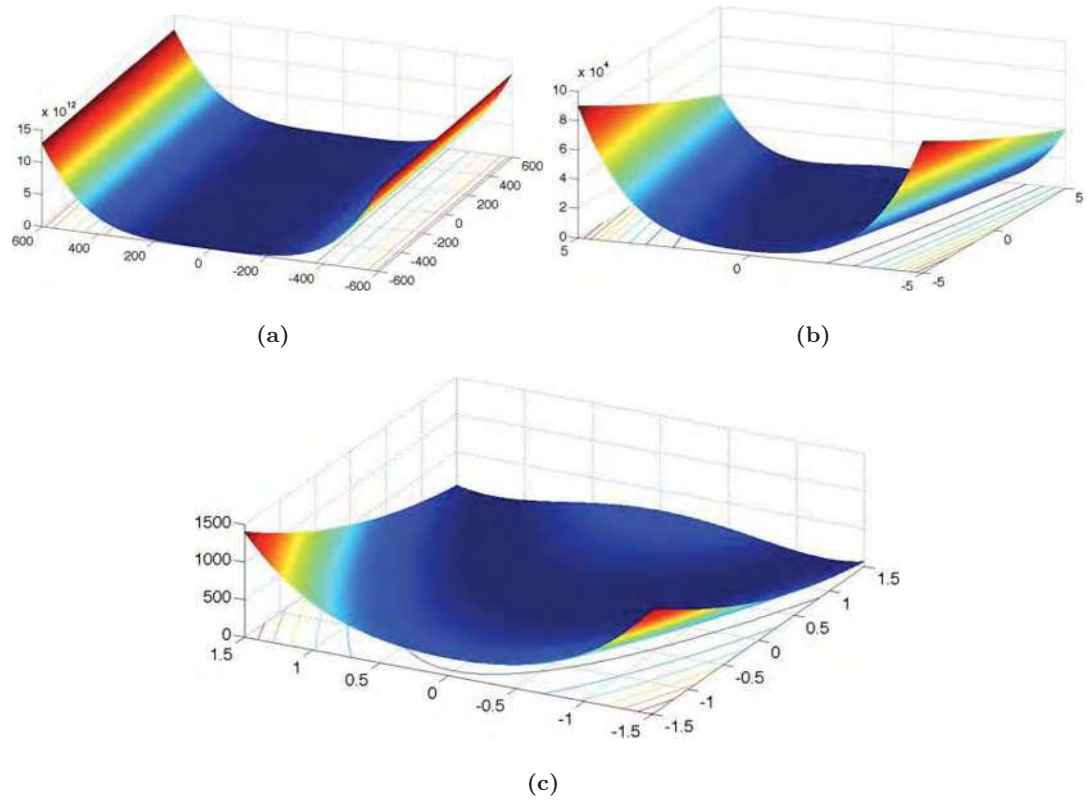



Figura 5.9: Error promedio en la minimización de la función de Griewank para números de evaluaciones correspondientes a $N_p = \{2048, 4096, 8192\}$.

implícitos en la aritmética de punto flotante, en comparación con el compilador del lenguaje C. Con el objetivo de probar dicha hipótesis sería necesaria la ejecución de experimentos específicos que puedan brindar resultados relevantes en tétminos de precisión por parte de ambos compiladores.

### 5.2.3. La función de Rosenbrock

La función de Rosenbrock es una función continua y diferenciable, no convexa y con un mínimo global $f(\mathbf{x}^*) = 0$ con $\mathbf{x}^* = [1, 1, ..., 1]$. La función original de Rosenbrock se estableció en primera instancia como una función bidimensional. Sin embargo, esta función ha sido generalizada a versiones de mayor dimensionalidad. Limitar el espacio de búsqueda para minimizar esta función ha representado un problema en sí mismo. Algunos estudios plantean limitar el dominio de búsqueda a $\{j \in \mathbb{R}^D \mid -2.048 \leq j \leq 2.048\}$, mientras otros lo definen en el intervalo $\{j \in \mathbb{R}^D \mid -5.12 \leq j \leq 5.12\}$, dominios para los cuales los métodos de optimización analítica suelen funcionar de manera muy adecuada [Price05]. Se han analizado además casos de estudios en el intervalo $\{j \in \mathbb{R}^D \mid -32 \leq j \leq 32\}$ y $\{j \in \mathbb{R}^D \mid -30 \leq j \leq 30\}$, el estudio de esta función para dichos dominios es en [Price05]. No obstante, en esta tesis el estudio se realizó estableciendo el espacio de búsqueda dentro




**(a)** **(b)**

**(c)**

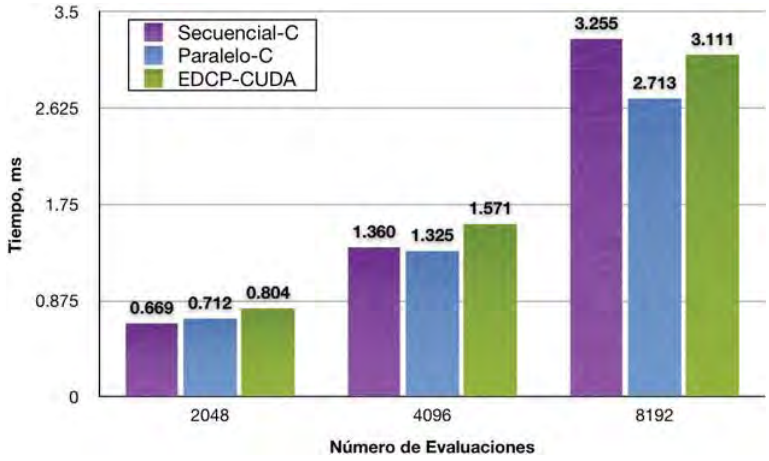
Figura 5.10: Función de Rosenbrock en 3D en los dominios (a).- $\{j \in \mathbb{R}^2 \mid -600 \leq j \leq 600\}$, (b).- $\{j \in \mathbb{R}^2 \mid -5 \leq j \leq 5\}$ y (c).- $\{j \in \mathbb{R}^2 \mid -1.5 \leq j \leq 1.5\}$.

de los límites $\{j \in \mathbb{R}^D \mid -600 \leq j \leq 600\}$ con la intención de evaluar el desempeño del algoritmo propuesto más allá de los límites establecidos en la literatura.

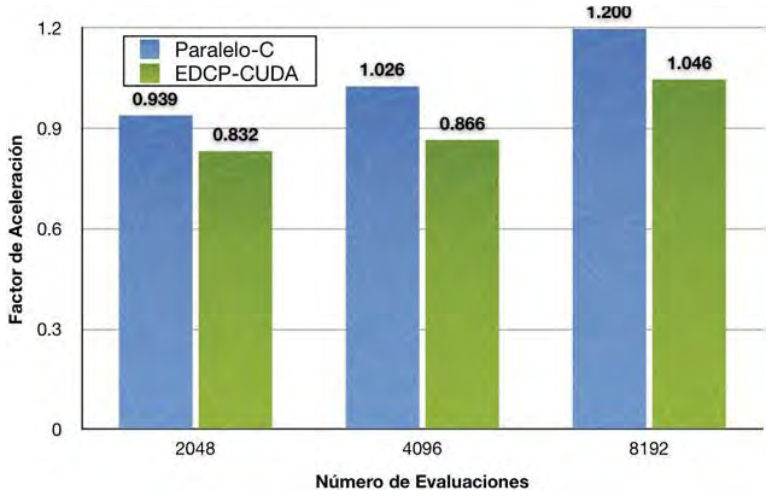
La Figura 5.10 muestra la representación en 3D de esta función. El mínimo global se encuentra dentro de un valle plano y estrecho con forma de parábola, el cual resulta trivial de ser encontrado. Sin embargo, aunque la evaluación de esta función no implica operaciones complejas, lograr converger al mínimo global resulta un problema que aumenta en dificultad dependiendo de los límites establecidos en el espacio de búsqueda. Los resultados obtenidos al minimizar la función de Rosenbrock aplicando el algoritmo de evolución diferencial propuesto en esta tesis se presentana continuación en términos de eficacia, presición, tiempos de ejecución y factores de aceleración.

### Tiempos de ejecución y factores de aceleración

En la Figura 5.11(a) se muestran los tiempos promedio de cómputo requeridos para desarrollar la etapa de evaluación de la función de Rosenbrock durante una generación y tamaños de población $N_p$ =2048, 4096 y 8192. En esta figura es posible observar que la versión secuencial presenta el crecimiento más pronunciado en los tiempos de cómputo al




**(a)**



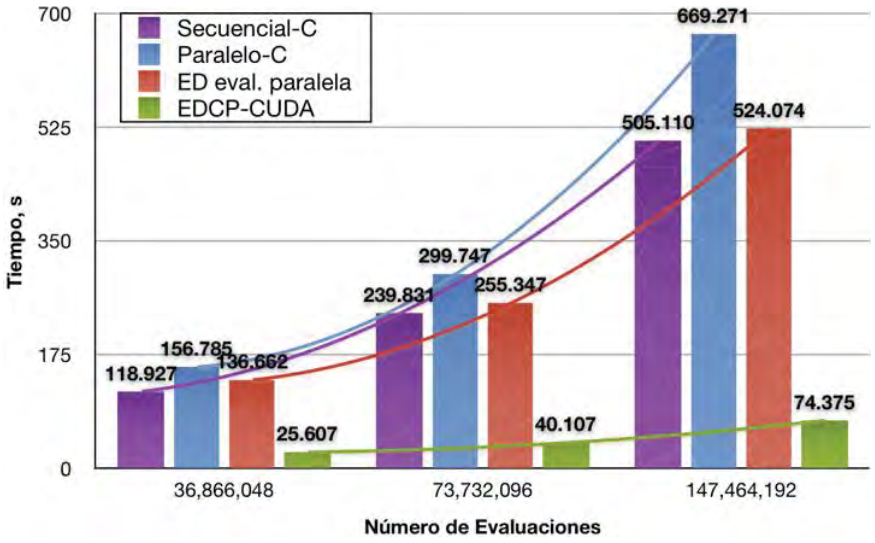

**(b)**

Figura 5.11: Resultados en la etapa de evaluación en términos de (a).- tiempos de cómputo y (b).- factores de aceleración para la función de Rosenbrock.

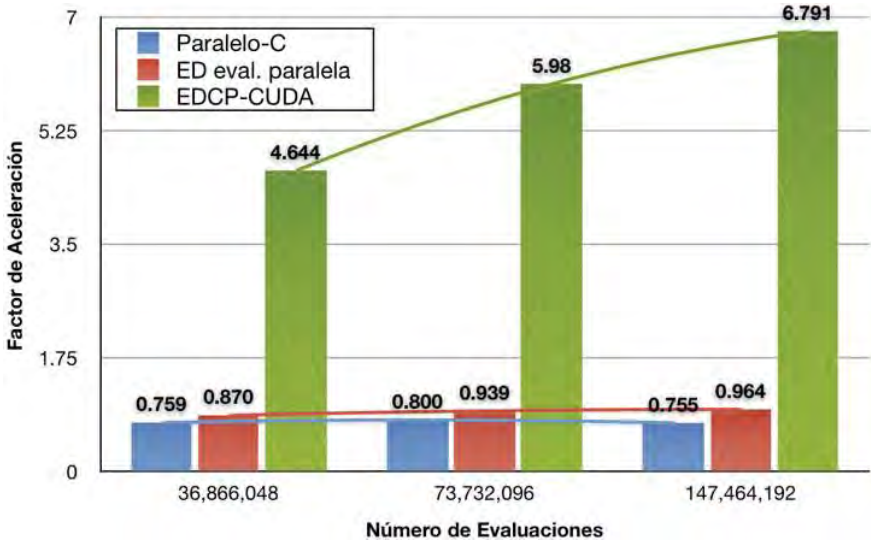
aumentar $N_{eval}$. La versión paralela en C sólo muestra beneficio cuando $N_p > 2048$ aunque la reducción en el esfuerzo computacional no es muy significativo. En lo que respecta al algoritmo en la GPU, se puede observar que se obtiene una reducción marginal de tan sólo 0.144 milisegundos cuando $N_p = 8192$. lo cual no representa un beneficio sustancial durante la etapa de evaluación.

Por su parte, la Figura 5.11(b) muestra los resultados obtenidos en términos de factores de aceleración. La version paralela en C logra un factor de aceleración máximo de 1.200 para $N_p = 8102$, a pesar de contar con 4 núcleos de procesamiento en la CPU. Sin embargo, ésta resulta ser la mejor solución debido a que el algoritmo programado en CUDA mostró un beneficio casi nulo para este mismo tamaño de población, en cuyo caso se obtuvo un factor de aceleración de 1.046. Los pobres beneficios por parte de las versiones paralelas están relacionados al esfuerzo computacional requerido por la función de Rosenbrock. Su definición no involucra operaciones complejas, por lo que la ejecución de la evaluación de esta función de manera secuencial, para tamaños de población pequeños, tarda menos tiempo que el requerido para establecer las condiciones que permitan ejecutarla de manera paralela.

El tiempo total promedio requerido en la minimización de la función de Rosenbrock se muestra en la Figura 5.12(a). Los tiempos de ejecución de la versión secuencial son 119.47, 189.26 y 324.98 para $g_{max} = 18000$ y tamaños de población $N_p = 2048, 4096, 8192$, lo cual se traduce en números de evaluaciones totales $N_{eval}$ =36866048, 73732096 y 147464192, respectivamente. Se puede apreciar que la versión paralela en C no muestra beneficio alguno para este caso particular, ya que se requiere tiempos de ejecución mayores que en la versión secuencial. Los tiempos de ejecución mayores por parte de la versión paralela en C están asociados a los problemas en los resultados presentados en la subsección B.7, referente a la mutación en la CPU. Los cambios de contexto producidos por los diferentes *threads* que participan en su la ejecución de la versión paralela producen tiempos de ejecución mayores durante la etapa de mutación por parte de la versión paralela en C. Estos problemas durante la mutación nulifican el poco beneficio obtenido durante la evaluación de individuos lo cual resulta en un tiempo de ejecución total mayor. Por su parte, versión EDCP basada en GPU requirió tiempos de ejecución de 25.607, 40.107 y 74.375 segundos para $N_{eval}$ =36866048, 73732096 y 147464192, respectivamente.




(a)




(b)

Figura 5.12: Esfuerzo computacional total en términos de (a).- tiempos de cómputo y (b).- factores de aceleración para la función de Rosenbrock.

Respecto al factor de aceleración total, la Figura 5.12(b) muestra los resultados producidos por las versiones paralelos en relación al código secuencial. Se puede apreciar

un desempeño superior por parte de la versión EDCP en CUDA, en relación a la versión paralela en C, la cual presenta factores de aceleración menores a 1. De tal manera que el beneficio que se pudiera obtener al realizar en paralelo la evaluación de esta operación relativamente sencilla, se ve degradado por el tiempo requerido para establecer la infraestructura requerida para poder realizar de manera paralela esta operación. La versión en la GPU presentó factores de aceleración de 4.644, 5.980 y 6.791 para $N_{eval}$ =36866048, 73732096 y 147464192. Se debe llamar la atención al hecho de que los pobres resultados obtenidos con la versión paralela en C y el beneficio reportado con la versión en la GPU está relacionado con el poco esfuerzo de cómputo que demanda la etapa de evaluación de este caso de estudio. Para el caso de la función de Rosenbrock se demuestra el beneficio de la completa paralelización del algoritmo de ED. A pesar de que durante la etapa de evaluación se obtuvo poco o nulo beneficio por parte de la versión en la GPU, el realizar el resto de las operaciones de manera paralela contribuyó a obtener factores de aceleración significativos.

**Eficacia**

El objetivo del algoritmo de ED propuesto es obtener un error máximo de $1 \times 10^{-10}$ en la minimización de la función de Rosenbrock, estableciendo $g_{max} = 18,000$. La Figura 5.13 muestra los resultados obtenidos en términos de la eficacia al minimizar dicha función. En esta figura se puede observar una mejora sólo cuando $N_p > 4096$, es decir, cuando $N_{eval} > 73,732,096$, tanto para la versión en la CPU como para la versión en la GPU, logrando el objetivo predefinido el 88.57 % y 91.43 % de las veces, respectivamente. Para $N_{eval} \leq 73,732,096$, sólo se logra el obtetivo el 82.86 % de las veces por cualquiera de las dos versiones.

**Precisión**

Los errores promedio de las 35 ejecuciones del algoritmo de ED para la minimización de la función de Rosenbrock usando diferentes tamaños de población se describen en la Figura 5.14. Los resultados obtenidos son errores promedio iguales a 0.683432, 0.569520 y 0.455615 en la CPU, mientras que en la GPU se obtienen 0.569521, 0.569520 y 0.341711, para $N_{eval}$ =36,866,048, 73,732,096 y 147,464,192, respectivamente. La precisión promedio

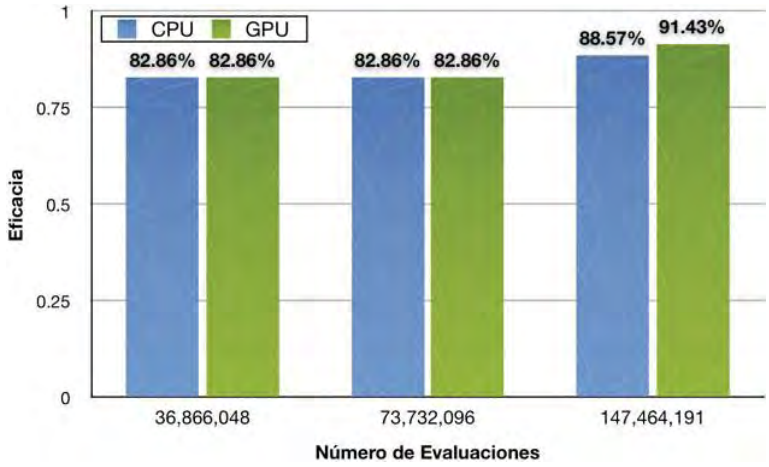



Figura 5.13: Eficacia en la minimización de la función de Rosenbrock para números de evaluaciones correspondientes a $N_p = \{2048, 4096, 8192\}$.

para el caso de esta función es muy baja, a pesar de que la eficacia llega hasta un 91.43 % cuando $N_p = 8192$. La obtención de estos grandes errores a pesar de la gran eficiencia depende de la topología de la función objetivo. En el caso de la función de Rosenbrock los rangos de las soluciones que no cumplen con la tolerancia son mucho mayores que los rangos de las soluciones que si la cumplen. La enorme diferencia que existe entre las soluciones que cumplen con la tolerancia y las que no (a pesar de encontrarse relativamente cerca unas de otras) produce un incremento considerable en el error promedio.

De la Figura 5.14 se pude apreciar nuevamente que la precisión obtenida por la versión en la GPU es mayor o igual que la obtenida en la CPU, lo cual plantea la hipótesis de que el compilador de CUDA presente un mejor rendimiento, que el compilador del lenguaje C, asociado a los errores de redondeo en la aritmética de punto flotante.

### 5.2.4. La función de Ackley

A excepción de los puntos cercanos al óptimo global, la función de Ackley se comporta como una llanura. En la Figura 5.15 se presenta la representación en 3D de esta función para diferentes intervalos en su dominio.

La función de Ackley es una de las funciones de prueba más comúnmente citadas

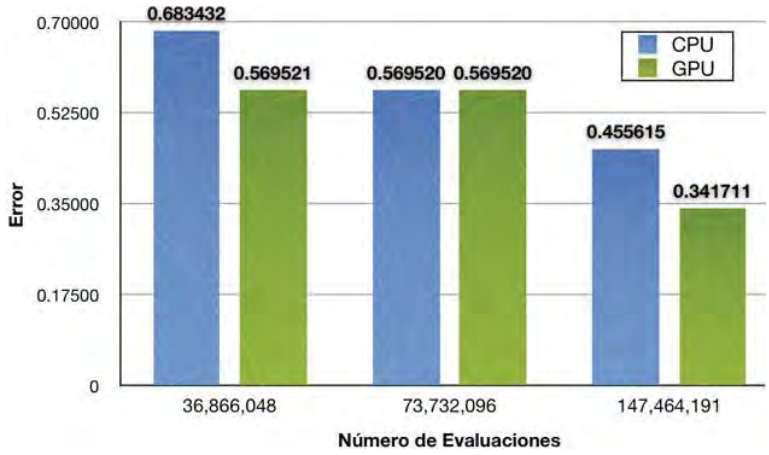
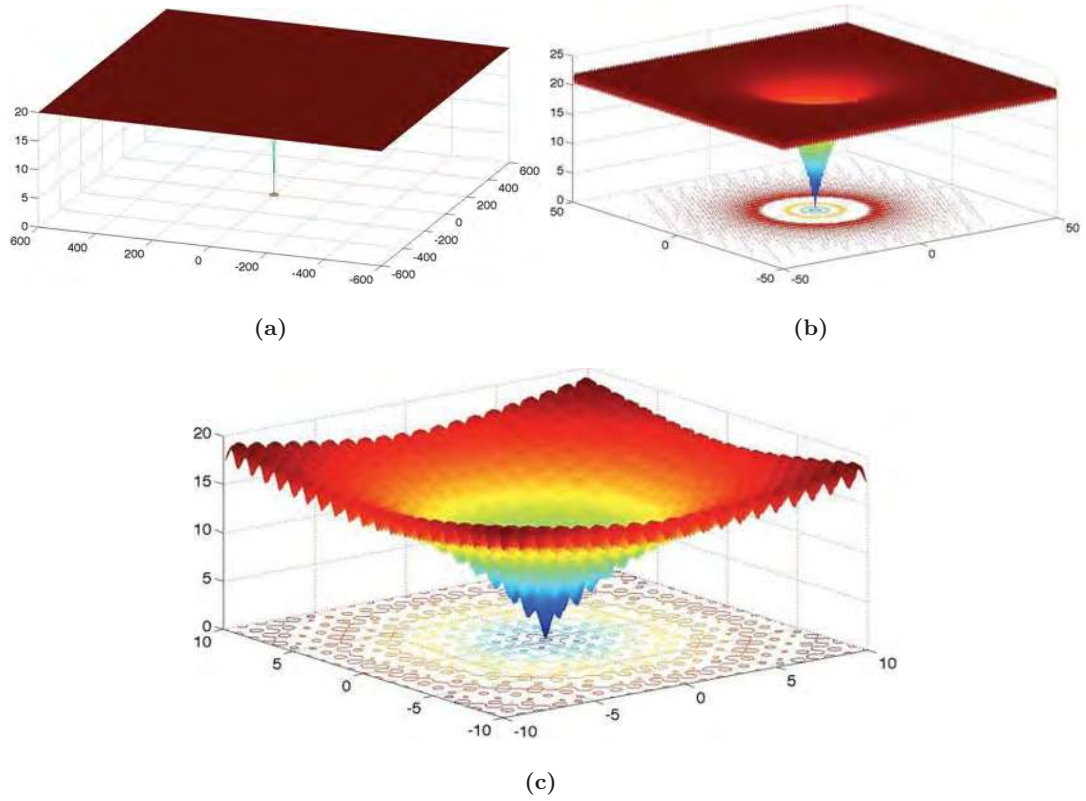



Figura 5.14: Error promedio en la minimización de la función de Rosenbrock para números de evaluaciones correspondientes a $N_p = \{2048, 4096, 8192\}$.

para evaluar el desempeño de algoritmos de optimización. Se trata de una función interesante con una enorme cantidad de óptimos locales y con un óptimo global localizado en el origen $f(\mathbf{x}^*) = 0$. En estudios realizados para grandes dimensionalidades ($D \geq 30$), es necesario tener cuidado en los códigos de las implementaciones para asegurar una evaluación precisa de esta función. Los límites en el espacio de búsqueda normalmente se establecen en $\{j \in \mathbb{R}^D \mid -32 \leq j \leq 32\}$ [Price05], sin embargo, el análisis de esta función en la referencia citada se establecen en $\{j \in \mathbb{R}^D \mid -30 \leq j \leq 30\}$. Para el caso de esta función, se establecieron dos diferencias en los experimentos con respecto al resto de los casos de estudio, dado que resultó más dificil lograr la convergencia del algoritmo implementado. La primer diferencia radica en que el espacio de búsqueda fue reducido a $\{j \in \mathbb{R}^D \mid -32 \leq j \leq 32\}$, en comparación con el resto de los casos de estudio. La segunda diferencia es que se llevaron a cabo experimentos multiplicando por 2, 4 y 6 la población más grande que se había manejado.

### Tiempos de ejecución y factores de aceleración

En la Figura 5.16(a) se muestran los tiempos de ejecución promedio requeridos para ejecutar la evaluación de la función de Ackley durante una generación, para poblaciones que




**(a)** **(b)**

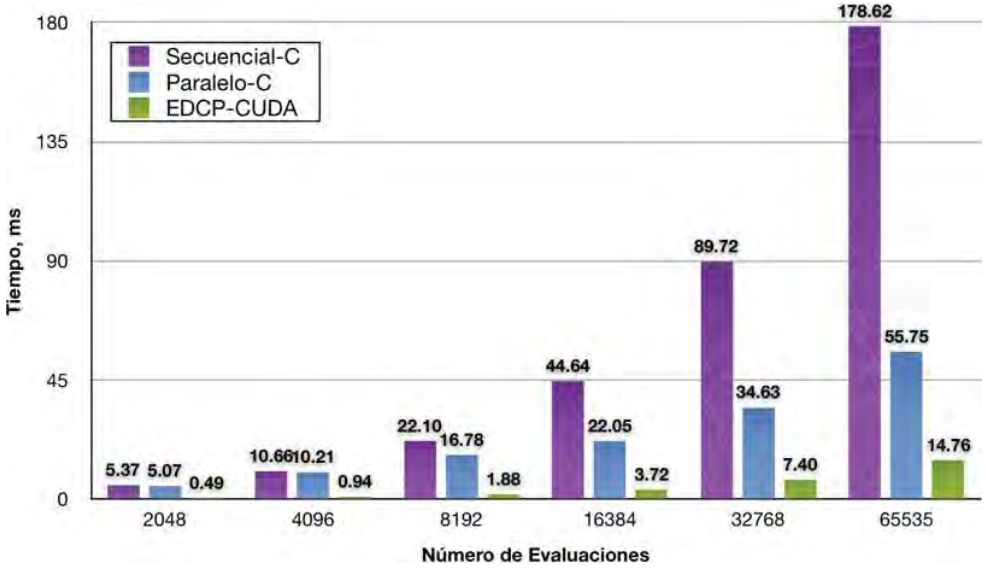
**(c)**

Figura 5.15: Función de Ackley en 3D en los dominios (a).- $\{j \in \mathbb{R}^2 \mid -600 \leq j \leq 600\}$, (b).- $\{j \in \mathbb{R}^2 \mid -50 \leq j \leq 50\}$ y (c).- $\{j \in \mathbb{R}^2 \mid -10 \leq j \leq 10\}$

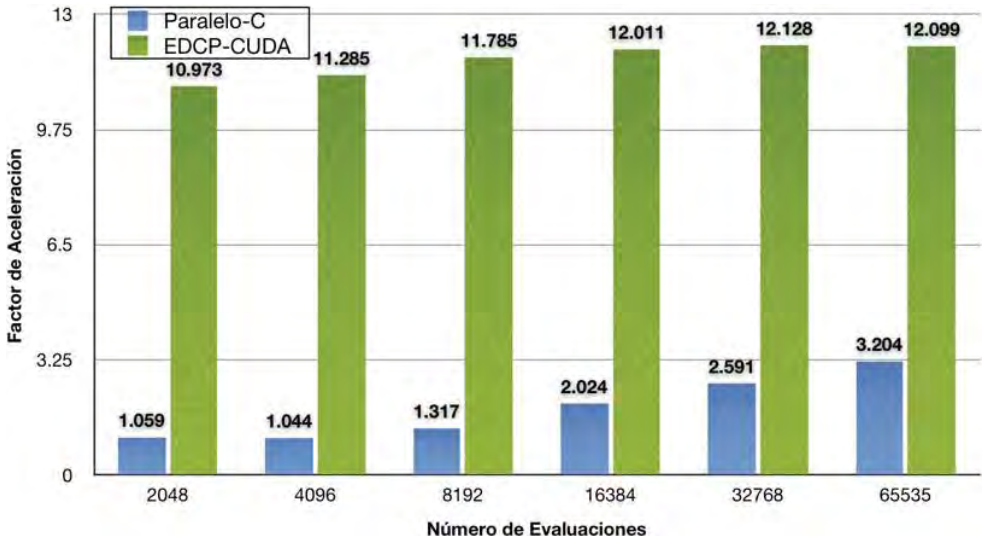
.

van desde $N_p = 2048$ hasta $N_p = 65535$. En esta figura es posible observar que la versión secuencial muestra un comportamiento lineal con un crecimiento muy pronunciado en los tiempos de cómputo. Un comportamiento similar es descrito por la versión paralela en C, sin embargo el crecimiento de los tiempos de cómputo al incrementar la población es menos pronunciado. Como es de esperarse, la versión paralela en C presenta mejores resultados a medida que aumenta $N_p$. Por su parte, la versión en la GPU denota un desempeño mucho mayor al mostrado por las versiones anteriores, obteniendo tiempos de ejecución que van desde 0.49 hasta 14.76 para $N_{eval}$ desde 2048 hasta 65535, respectivamente.

Por otra parte, la Figura 5.16(b) muestra los factores de aceleración obtenidos con las versiones paralelas del algoritmo de ED. Un factor de aceleración cercano al ideal se




**(a)**




**(b)**

Figura 5.16: Resultados en la etapa de evaluación en términos de (a).- tiempos de cómputo y (b).- factores de aceleración para la función de Ackley.

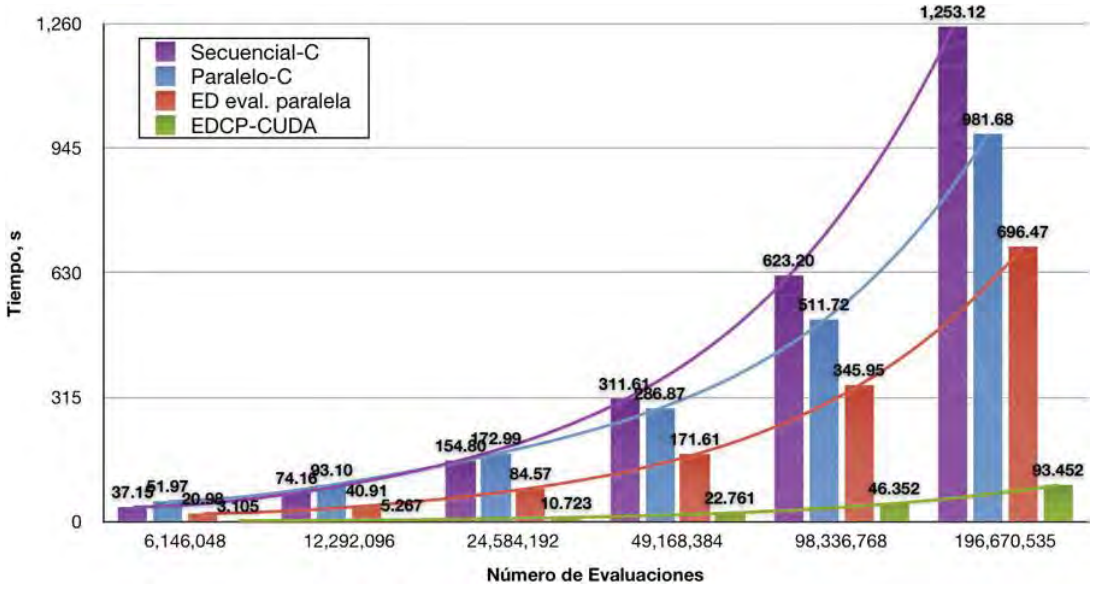
logró por parte de la versión en C para $N_{eval} = 65535$, considerando que se dispone de cuatro núcleos de procesamiento. La versión en CUDA mostró un rendimiento superior al obtener

factores de aceleración en esta etapa que van desde 10.973 hasta llegar a un máximo de 12.128 para $N_{eval}$ que va desde 2048 hasta 32768, respectivamente. Para el número máximo de evaluaciones $N_{eval} = 65535$ se presenta un pequeño decremento en el beneficio, el cual se justifica considerando que los espacios de memoria para texturas (donde se encuentra almacenada la población) son recursos compartidos y que para $N_{eval} = 65535$ se presenta la mayor concurrencia en el acceso a estos recursos. Es necesario recordar que entre mayor sea la concurrencia para el uso de un recurso compartido, existe mayor probabilidad de que se puedan generar cuellos de botella en el medio de acceso o el controlador de dicho recurso [Soveiko10].

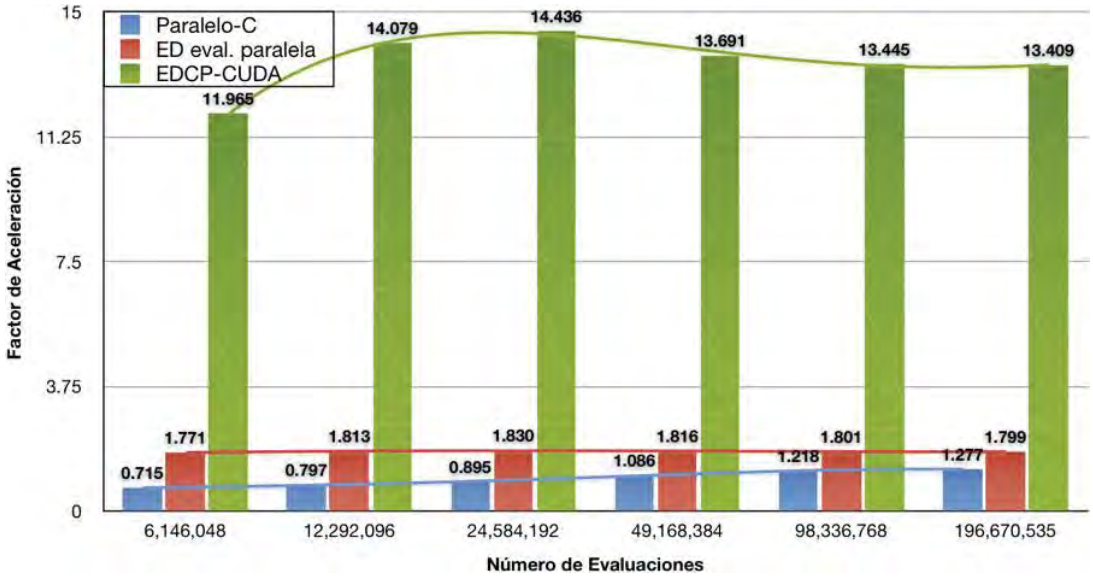
El tiempo total promedio requerido en la minimización de la función de Ackley se muestra en la Figura 5.17(a), para los tamaños de población que van desde $N_p = 2048$ hasta $N_p = 65535$ y $g_{max} = 3000$. En esta figura es posible apreciar que la versión paralela en C sólo muestra un beneficio moderado. La versión en la que se paraleliza exclusivamente la evaluación en la GPU reduce un poco más el esfuerzo de cómputo. Por otra parte, a través de la versión EDCP basada en GPU se logra un gran beneficio en todo momento.

Respecto al factor de aceleración total del proceso de optimización, en la Figura 5.17(b) se describen los resultados en términos de factores de aceleración producidos por las versiones que implican procesamiento paralelo, en referencia al código secuencial en C. Como se puede apreciar en esta figura, la versión paralela multi-hilo en C alcanza un factor máximo de aceleración de tan sólo 1.277 cuando $N_{eval} = 196670535$, mientras que la versión en la que se paraleliza la evaluación en la GPU obtiene su máximo beneficio cuando $N_{eval} = 24584192$, el cual es de 1.830. Por su parte, la versión EDCP en CUDA presenta un desempeño superior en todo momento, obteniendo el mayor factor de aceleración igual a 14.436 cuando $N_{eval} = 24584192$.

La disminución en el beneficio por parte de esta versión mostrada en la Figura 5.17(b) para $N_{eval} > 24584192$ y $N_p > 8192$ se encuentra relacionada una vez más al alto grado de concurrencia que se produce en el acceso a los recursos de cómputo de que se dispone, lo cual incrementa considerablemente la latencia de los recursos de memoria. Además, es necesario considerar que los bloques de *threads* en las implementación basadas en GPU son configurados para contener un número de *threads* igual a 16. Esto implica que




(a)




(b)

Figura 5.17: Esfuerzo computacional total en términos de (a).- tiempos de cómputo y (b).- factores de aceleración para la función de Ackley.

para un tamaño de población de $N_p = 8192$ se requieren 512 bloques de *threads* y este número aumenta al incrementarse el tamaño de la población. Cada bloque de *threads* debe ser ejecutado en una misma unidad SM por lo que se requiere un trabajo de calendarización más intenso y colas de espera más grandes al aumentar el número de bloques, considerando que en el dispositivo utilizado sólo se dispone de 14 SMs. La disminución en el beneficio por parte de las porciones paralelas de código y el aumento constante de las porciones secuenciales producen como resultado una disminución en el desempeño global del algoritmo en la GPU.

### Eficacia

Los cambios del experimento que se establecieron para este caso de estudio fueron necesarios ya que la eficacia en los experimentos resultó ser de 0 % para la tolerancia al error establecida. A pesar de tratar con tamaños de población mayores a los empleados para el resto de las funciones, la eficacia del método no se logró mojorar. Sin embargo, aún con los malos resultados en términos de eficacia, el error promedio sí se logra disminuir, como se presenta en la siguiente subsección.

### Precisión

En la Tabla 5.5 se presentan los resultados obtenidos durante las 35 ejecuciones del algoritmo para un tamaño de población $N_p = 65535$ y $N_{eval} = 196670535$.

Los errores promedio obtenidos por el método de ED en la minimización de la función de Ackley se muestran en la Figura 5.18, para los tamaños de población que van desde $N_p = 2048$ hasta $N_p = 65535$ y sus correspondientes números de evaluaciones. De esta figura es posible apreciar que el error promedio para el caso de esta función va desde 0.000016 hasta 0.000014 para el caso de las versiones en la CPU, mientras que para la GPU los errores promedio van desde 0.000017 hasta 0.000007, para tamaños de población desde 2048 hasta 65535, respectivamente. Una vez más se puede observar , tanto de la Tabla 5.5 como de la Figura 5.18, que la versión ejecutada en la GPU presenta en general mayor precisión que el código ejecutado en la CPU.

Tabla 5.5: Errores obtenidos para $N_p = 65535$ en la minimización de la función de Ackley

| Ejecución | Secuencial-C | Paralelo-C | C-CUDA |
|---|---|---|---|
| 1 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 2 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 3 | 1.62E-05 | 1.24E-05 | 6.68E-06 |
| 4 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 5 | 1.62E-05 | 1.24E-05 | 6.68E-06 |
| 6 | 1.62E-05 | 1.24E-05 | 8.58E-06 |
| 7 | 1.24E-05 | 1.62E-05 | 6.68E-06 |
| 8 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 9 | 8.58E-06 | 1.24E-05 | 8.58E-06 |
| 10 | 1.24E-05 | 1.24E-05 | 8.58E-06 |
| 11 | 1.24E-05 | 1.62E-05 | 6.68E-06 |
| 12 | 1.24E-05 | 1.62E-05 | 6.68E-06 |
| 13 | 1.62E-05 | 1.24E-05 | 6.68E-06 |
| 14 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 15 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 16 | 1.24E-05 | 1.62E-05 | 6.68E-06 |
| 17 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 18 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 19 | 1.24E-05 | 1.62E-05 | 1.05E-05 |
| 20 | 1.62E-05 | 1.24E-05 | 6.68E-06 |
| 21 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 22 | 1.24E-05 | 1.62E-05 | 8.58E-06 |
| 23 | 1.24E-05 | 1.62E-05 | 6.68E-06 |
| 24 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 25 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 26 | 1.62E-05 | 1.62E-05 | 6.68E-06 |
| 27 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 28 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 29 | 1.62E-05 | 1.24E-05 | 6.68E-06 |
| 30 | 1.24E-05 | 1.24E-05 | 8.58E-06 |
| 31 | 1.24E-05 | 1.62E-05 | 6.68E-06 |
| 32 | 1.24E-05 | 1.62E-05 | 6.68E-06 |
| 33 | 1.24E-05 | 1.24E-05 | 6.68E-06 |
| 34 | 1.24E-05 | 1.62E-05 | 6.68E-06 |
| 35 | 1.24E-05 | 1.24E-05 | 6.68E-06 |

### 5.2.5. Caso de estudio 4: La función $f_4$

La función denominada en este trabajo como $f_4$ está definida por,

$$f(\mathbf{x}) = 0.1 \left\{ \sin^2(3\pi x_1) + \sum_{j=1}^{D-1} (x_i - 1^2) \left[1 + 10 \sin^2(3\pi x_{i+1})\right] + (x_D - 1)^2 \left[1 + \sin^2(2\pi x_D)\right] \right\} + \sum_{j=1}^{D} u(x_i, 5, 100, 4) \tag{5.8}$$

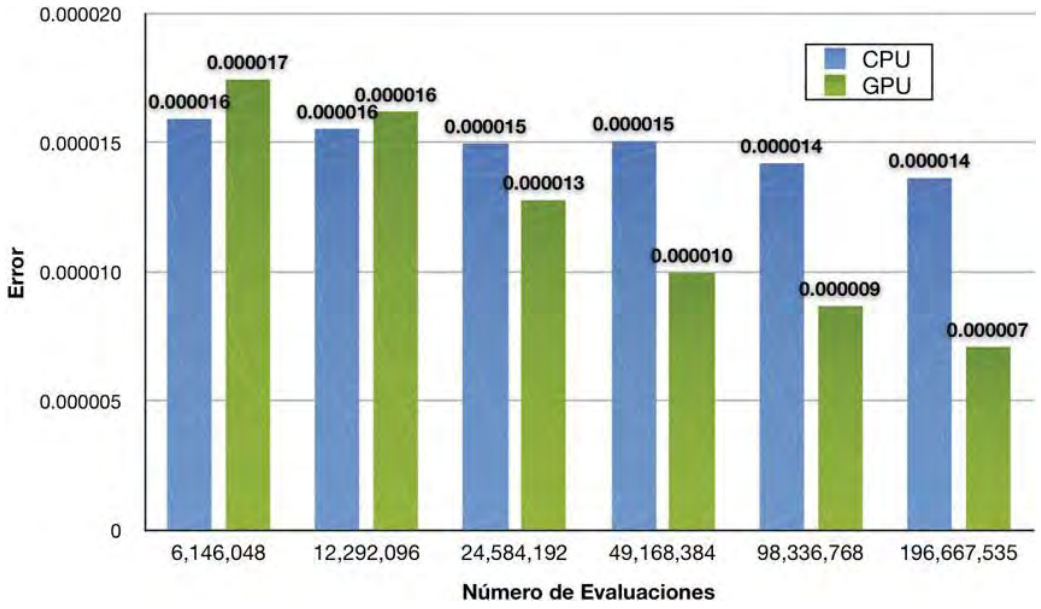



Figura 5.18: Error promedio en la minimización de la función de Ackley para $g_{max} = 3000$, $N_p = \{2048, 4096, 8192, 16384, 32768, 65535\}$.

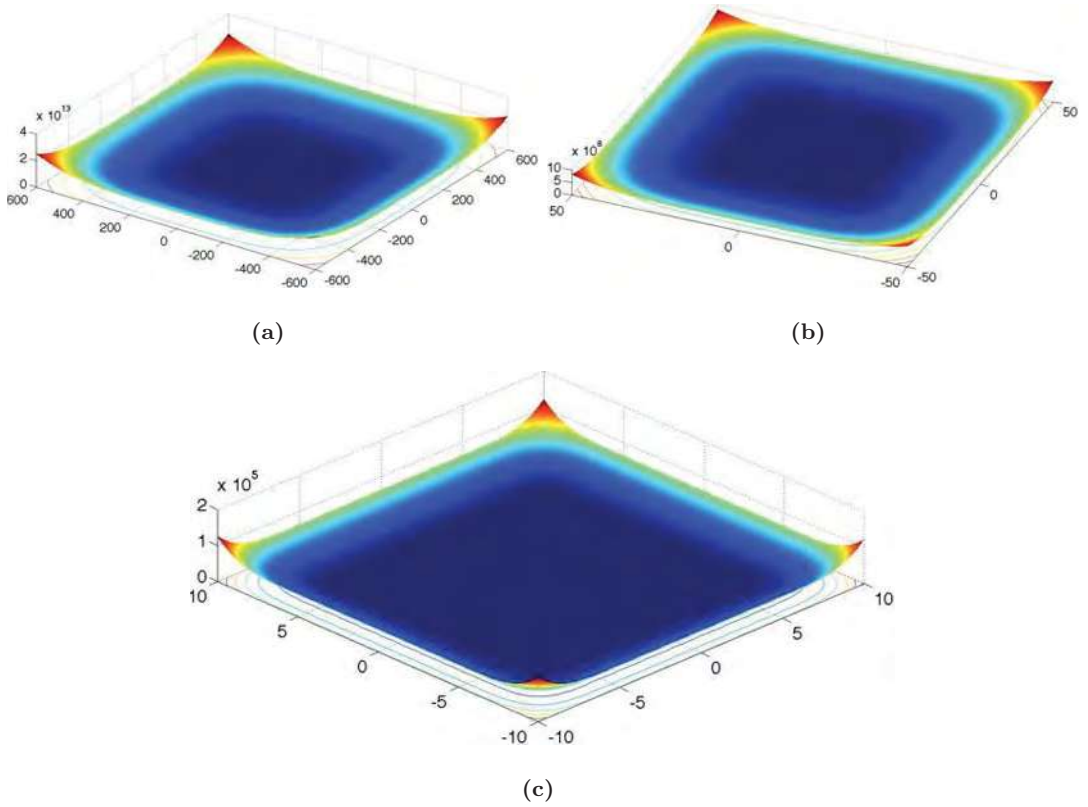
en donde,

$$u(x, a, k, m) = \begin{cases} k \cdot (x-a)^m, & x > a \\ 0, & -a \leq x \leq a \\ k \cdot (-x-a)^m, & x < -a \end{cases} \tag{5.9}$$

se trata de una función continua pero no diferenciable cuya representación en 3D se muestra en la Figura 5.19 para diferentes intervalos en su dominio. Los resultados en términos de eficacia, presición, número de generaciones y tiempos de ejecución asociados a la aplicación del algoritmo de ED en la minimización de esta función se describen a continuación.

### Tiempos de ejecución y factores de aceleración

En la Figura 5.20(a) se muestran los tiempos promedio de cómputo requeridos para ejecutar la operación de evaluación emplando la función $f_4$ como función de prueba durante una generación, para los tamaños de población $N_p = \{2048, 4096, 8192\}$. En esta figura es posible observar que la versión secuencial muestra un crecimiento lineal muy pronunciado



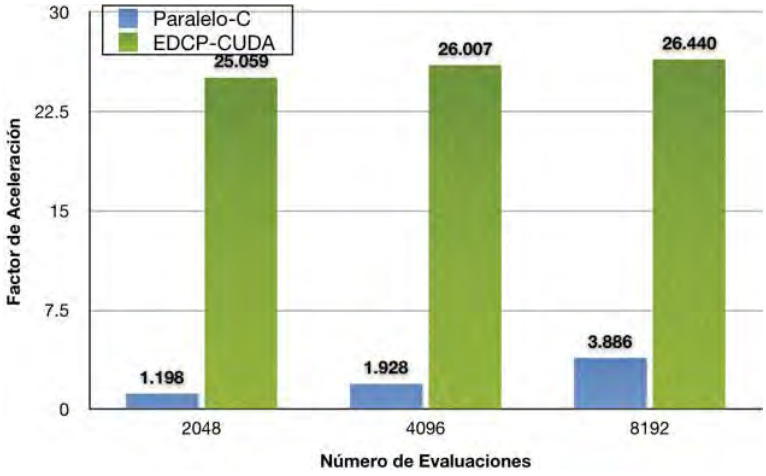

**(a)** **(b)**

**(c)**

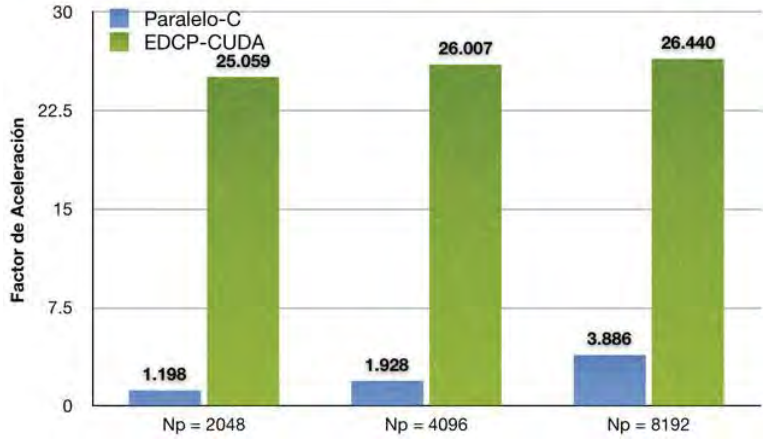
Figura 5.19: Función de $f_4$ en 3D en los dominios (a).- $\{j \in \mathbb{R}^2 \mid -600 \leq j \leq 600\}$, (b).- $\{j \in \mathbb{R}^2 \mid -50 \leq j \leq 50\}$ y (c).- $\{j \in \mathbb{R}^2 \mid -10 \leq j \leq 10\}$.

en cuanto a los tiempos de cómputo, al igual que la versión paralela en C. Sin embargo, esta última presenta un crecimiento más lento, mostrando una reducción importante en el esfuerzo computacional. Por su parte, el algoritmo programado para su ejecución en la GPU prueba ser la mejor solución obteniendo una reducción considerable en los tiempos de cómputo.

En la Figura 5.20(b) se muestran los factores de aceleración logrados por las versiones paralelas. La versión paralela en código C muestra un factor de aceleración máximo ligeramente superior a 3.886 cuando $N_p = 8192$, lo cual implica un valor cercano al ideal, considerando que la CPU cuenta con 4 núcleos de procesamiento. La versión en CUDA muestra un rendimiento mucho mayor obteniendo factores de aceleración de 25.059, 26.007




**(a)**




**(b)**

Figura 5.20: Resultados durante la etapa de evaluación en términos de (a).- tiempos de cómputo y (b).- factores de aceleración para la función $f_4$.

y 26.440 para $N_p = \{2048, 4096, 8192\}$

El tiempo total promedio requerido por el proceso de minimización se muestra en la Figura 5.21(a), con $g_{max} = 2,000$. En esta figura es posible apreciar que en todo momento se logra una importante reducción del esfuerzo de cómputo total a través de la versión EDCP basada en GPU, a diferencia de la versión paralela en la CPU en la cual la reducción de los tiempos de cómputo son a lo mucho de un poco más de la mitad cuando

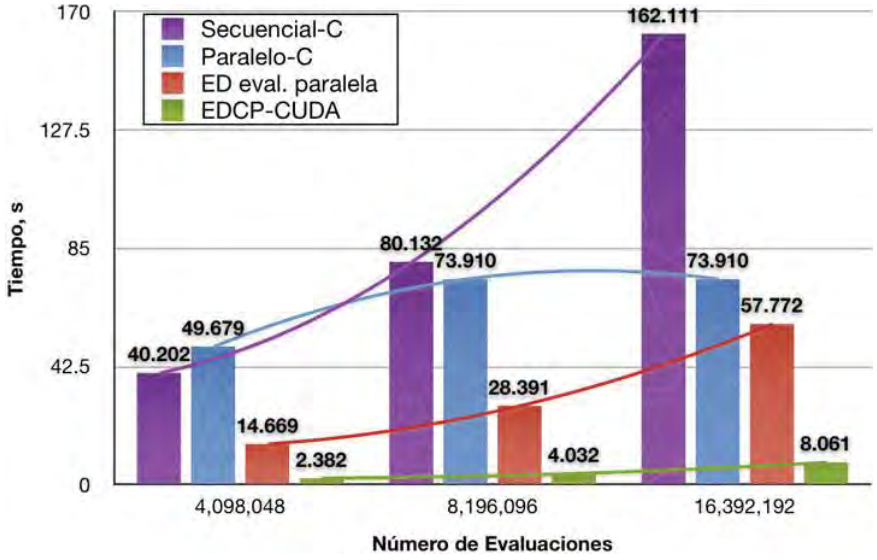
$N_{eval} = 16,392,192,$. La versión en la cual sólo se paraleliza la evaluación de los individuos en la GPU brinda una ligera reducción en el esfuerzo computacional en comparación con la versión paralela en C. Sin embargo, el beneficio es inferior al obtenido por EDCP.

En lo que respecta a los factores de aceleración totales, la Figura 5.21(b) describe los resultados producidos por las versiones que implican procesamiento paralelo. En esta figura se puede apreciar el desempeño superior por parte de la versión ejecutada en la GPU, obteniendo factores de aceleración de 16.879, 19.874 y 20.110 para los números de evaluaciones $N_{eval}$ =4098048, 8196096 y 16392192, respectivamente. Por otra parte, la versión multi-hilo en C muestra un beneficio máximo de 2.193 cuando $N_{eval}$ =16,392,192 y éste ya no mejora al aumentar el número de evaluaciones. Finalmente, la versión en la cual se paraleliza la operación de evaluación en la GPU presenta su mayor beneficio cuando $N_{eval}$ =8,196,096 el cual es igual a 2.822. Este factor de aceleración es superior al obtenido por la versión en C, pero mucho menor que el obtenido por la versión EDCP.

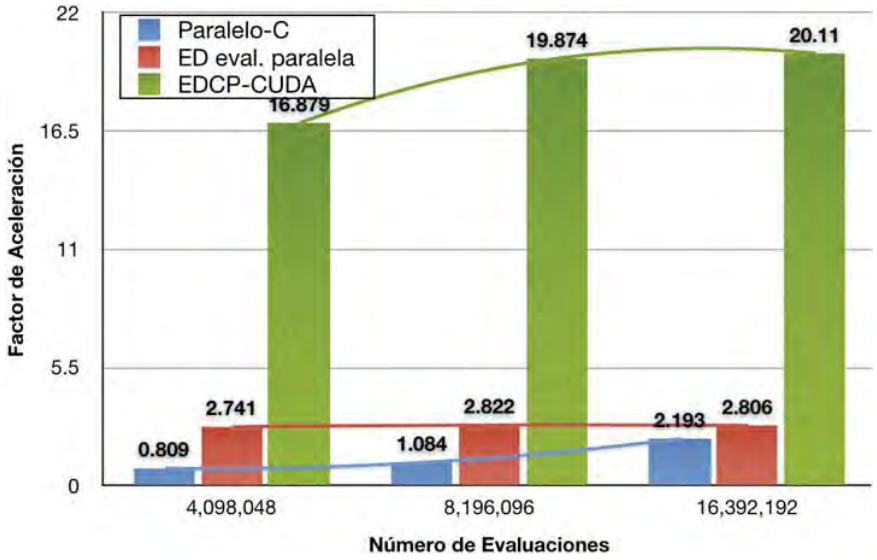
**Eficacia**

La Figura 5.22 muestra los resultados obtenidos en términos de la eficacia al minimizar la función $f_4$ para distinto número de evaluaciones $N_{eval}$ = 4098048, 8196096 y 16392192, considerando un error máximo permitido de $1 \times 10^{-10}$ y $g_{max}$=2,000. En la Figura 5.22 es posible apreciar los valores correspondientes al porcentaje de eficacia para los distintos números de evaluaciones, los cuales corresponden al 91.11 %, 93.33 % y 97.78 % para $N_{eval}$ = 4098048, 8196096 y 16392192, respectivamente, por parte de las versiónes en la CPU. De estos tres resultados se observa una alta eficacia en la minimización de esta función, superando el 90 % en todos los casos. Por otra parte, la versión en la GPU presenta excelentes resultados, consiguiendo en todos los casos una eficacia del 100 %, es decir, siempre se logra el objetivo predeterminado, el cual consiste en lograr una tolerancia máxima al error permitido de $1 \times 10^{-10}$.

El mejor desempeño por parte de la versión EDCP basada en GPU, en términos de eficacia, está relacionado estrechamente a los resultados presentados a continuación, los cuales describen la precisión obtenida por las versiones en la CPU y en la GPU.




(a)




(b)

Figura 5.21: Esfuerzo computacional total en términos de (a).- tiempos de cómputo y (b).- factores de aceleración para la función $f4$.

### Precisión

Los errores promedio para la función $f_4$, para diferentes números de evaluaciones son descritos en la Figura 5.23. De esta figura se puede observar que la precisión promedio

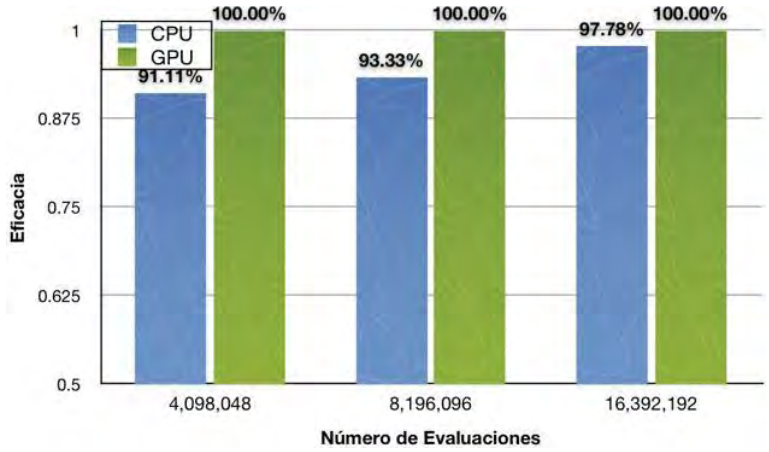



Figura 5.22: Eficacia en la minimización de la función de $f_4$ para números de evaluaciones correspondientes a $N_p = \{2048, 4096, 8192\}$.

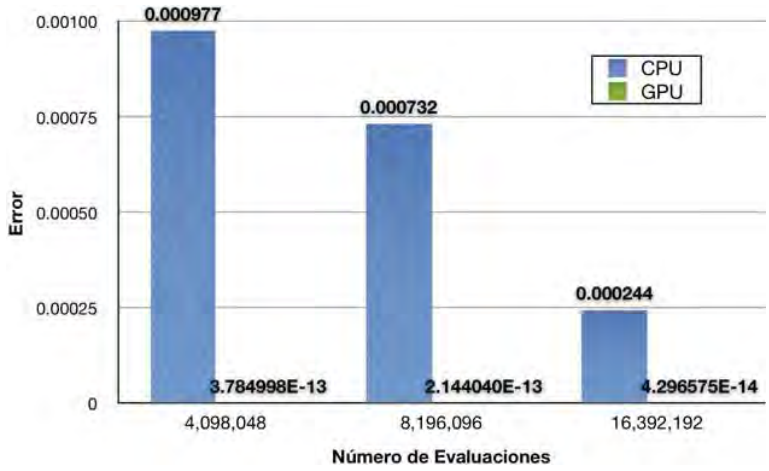



Figura 5.23: Error promedio en la minimización de la función $f_4$ para números de evaluaciones correspondientes a $N_p = \{2048, 4096, 8192\}$.

para el caso de esta función en la CPU logra valores de 0.000977, 0.000732 y 0.000244 para $N_{eval} = 4098048$, 8196096 y 16392192, respectivamente. En el caso de la versión en la GPU estos errores son muchos menores e iguales a 3.784998E-13, 2.144040E-13 y 2.144040E-13 para los correspondientes números de evaluaciones.

Tal como se planteó previamente, la diferencia en la precisión obtenida por la

implementación del algoritmo propuesto en la CPU y la versión en la GPU puede deberse a la manera en que ambos compiladores manejan la aritmética de punto flotante. De acuerdo a los resultados obtenidos a través de los casos de estudio, este efecto se puede asociar a que los errores de redondeo durante las conversiones de binario a flotante pudieran ser más significativos en la CPU que en la GPU lo cual lleva a las operaciones ejecutadas en este dispositivo a obtener una mayor precisión. Sin embargo, es necesario realizar un análisis más específico de ambos compiladores y la ejecución de experimentos particulares para poder comprobar dicha hipótesis.

## 5.3. Comparación entre el algoritmo original de evolución diferencial y el algoritmo de ED propuesto

En el esquema propuesto de ED la constante $F$ es sustituida por un vector aleatorio con componentes $\lambda_j \in (0, 1]$ durante la mutación, lo cual elimina la necesidad de tener que ajustar $F$. Cada una de las componentes $\lambda_j$ se usa para escalar la diferencia entre las componentes $j$ de dos vectores miembros de la población elegidos aleatoriamente. Como complemento, un factor $(1 - \lambda_j)$ es empleado para escalar la diferencia entre la componente $x_{i,j}$ y la componente $x_{best,j}$ del vector que representa la mejor solución encontrada en la historia. Esta combinación lineal normalizada de diferencias se suma finalmente a cada una de las componentes $x_{i,j}^{(g)}$ del vector base $\mathbf{x}_i^{(g)}$ produciendo un mutante $\mathbf{v}_i^{(g)}$. De acuerdo a lo anterior, para cada vector en la población en la generación $g$ denotado como $\mathbf{x}_i^{(g)}$, un vector mutante $\mathbf{v}_i^{(g)}$ es creado a través del cálculo de sus componentes por medio de la siguiente definición,

$$v_{i,j}^{(g)} = x_{i,j}^{(g)} + \lambda_j \cdot (x_{r1,j}^{(g)} - x_{r2,j}^{(g)}) + (1 - \lambda_j) \cdot (x_{best,j}^{(g)} - x_{i,j}^{(g)}), \quad \forall \;\; i = 1, 2, ..., N_p, \;\; j = i, 2, ..., D \tag{5.10}$$

en donde $r1$, $r2 \in [1, N_p]$ son enteros aleatorios diferentes entre sí.

Con el objetivo de evaluar el desempeño de la propuesta implementada en esta tesis, se presenta la comparación de la misma con el esquema básico de ED representado esquemáticamente en la Figura 3.2. Los códigos del método de ED utilizados para realizar

esta comparación son únicamente las versiones basadas en GPU y cada experimento fue ejecutado 35 veces.

Los parámetros de ejecución empleados en esta etapa de análisis son los mismos que fueron empleados en las secciones anteriores para cada uno de los casos de estudio. El número de dimensiones es $D = 128$ y los parámetro $N_p$ y $g_{max}$ fueron establecidos con los valores máximos empleados en cada caso y a través de los cuales de calcula el número de evaluaciones totales realizadas por el algoritmo. Estos valores en los parámetros de ejecución pretenden elevar las posibilidades de encontrar el valor mínimo y de esta forma obtener la mayor eficacia y precisión posibles. La constante de mutación $F$ y la probabilidad de cruza $Cr$ fueron definidas como 0.3 y 0.5, respectivamente, para el caso del esquema original de ED. Tanto la constante $Cr$ como $F$ se establecieron de acuerdo a los valores manejados en [Storn95].

Tabla 5.6: Comparación del desempeño del algoritmo de ED original y la propuesta de esta tesis para minimización de la función de Griewank, Rosenbrock, Ackley y $f_4$.

| Función de prueba | Esquema de ED | Número de evaluaciones | Tiempo de cómputo (s) | Error promedio |
|---|---|---|---|---|
| Griewank | Original | 8,299,192 | 5.43684 | 0.04316 |
| | Propuesto | | 5.14613 | 0.00041 |
| Rosenbrock | Original | 147,464,192 | 92.017630 | 114.25053 |
| | Propuesto | | 79.20554 | 0.35437 |
| Ackley | Original | 196,670,535 | 152.13696 | 21.11168 |
| | Propuesto | | 139.83214 | 0.000007 |
| $f_4$ | Original | 16,392,192 | 11.29543 | 4.00148 |
| | Propuesto | | 9.53449 | 0.00024 |

La Tabla 5.6 muestra la comparación de los resultados en desempeño del esquema básico de ED con la propuesta implementada en esta tesis. Estos resultados están expresados en términos de tiempos de cómputo y los errores promedio obtenidos por ambas versiones. De esta tabla se puede observar que para un mismo número de evaluaciones, en general, el algoritmo de ED propuesto logra converger hacia una mejor solución, de acuerdo al error promedio obtenido por ambas versiones. La diferencia en cuanto a los errores obtenidos radica en las diferencias que existen en las estrategias de búsqueda implícitas en ambos esquemas. El esquema propuesto de ED define una búsqueda más directa que lo lleva a

converger hacia una mejor solución en un número menor de evaluaciones. Por su parte, el esquema original de ED requiere un número mayor de evaluaciones para poder obtener resultados similares, los cuales, como se ha reportado en la literatura, es capaz de obtener. Los tiempos de cómputo resultan ser menos relevantes, considerando que el algoritmo de ED propuesto omite la operación de cruza, por lo cual se esoera que su tiempo de ejecución sea menor que en el caso del algoritmo de ED original.

En el caso de la función de Griewank el error promedio obtenido por ambas versiones el cual es de 0.04316 por parte del esquema original y 0.00041 por parte de la actual propuesta. Para el caso de la función de Rosenbrock se puede observar un error promedio muy grande igual a 114.25053 por parte de la implementación que involucra el esquema original de ED, mientras que la versión del esquema propuesto en esta tesis obtiene un error de 0.35437. Considerando los resultados obtenidos para esta función durante la etapa de evaluación, se puede observar que la operación de recombinación tiene una contribución importante en los tiempos de cómputo de este caso particular usando el esquema de ED original. En el caso de la función de Ackley se observa un desempeño deficiente en términos de precisión al obtener un error promedio de 21.11168 por parte del algoritmo original de ED. Por su parte, la versión que se propone en esta tesis logró obtener un error promedio igual a 0.000007 para esta misma función. En el caso de la función $f_4$ es posible observar un error promedio igual a 4.00148 con esquema original de ED, en comparación con el esquema que se propone en donde resultó sólo de 0.00024.

## 5.4. Conclusiones

El análisis desarrollado empleando cuatro funciones de prueba, las cuales han sido ampliamente utilizadas en la literatura especializada, ha permitido realizar la evaluación del desempeño del algoritmo de evolución diferencial paralelo implementado en la GPU, en el cual se realizan todas las operaciones evolutivas empleando un esquema de procesamiento paralelo. La versión EDCP del algoritmo propuesto, programada en CUDA, mostró un desempeño superior en términos de tiempo de cómputo, en comparación con las versiones secuencial y paralela del mismo algoritmo programadas en la CPU, así como una versión

más en la cual sólo se paraleliza la operación de evaluación en la GPU. El esquema de paralelización empleado y la explotación eficiente de los recursos de cómputo de la GPU, como la utilización de espacios de memoria de rápido acceso tales como la memoria constante y los espacios de memoria de texturas, resultaron ser factores clave en la obtención de los beneficios reportados por parte de la versión EDCP.

A pesar de establecer una tolerancia máxima al error de $tol = 1 \times 10^{-10}$, los resultados en términos de presición y eficacia mostraron un alto desempeño por parte de la estrategia de búsqueda empleada por el esquema de ED propuesto. En general, la eficacia del método aumentó a medida que el error disminuye cuando aumenta el parámetro $N_p$ y consecuentemente el número de evaluaciones. Los mejores resultados en cuanto a eficacia y errores promedio fueron obtenidos para tamaños de población grandes. Una excepción resultó ser el caso de la función de Ackley para la cual la eficacia resulto de 0 %. Esto significa que no se cumplió nunca con la tolerancia establecida a pesar de realizar la experimentación hasta con $N_p = 65535$, sin embargo, el error promedio fue efectivamente reducido hasta llegar a 0.000007. Para el caso de la función de Griewank se obtuvo una eficacia máxima de 95.56 % y un error 0.00041. La función de Rosenbrock reportó resultados de hasta 90 % en la efiacia para un error promedio de 0.354. Por último, la función $f_4$ presentó una eficacia máxima de 97.78 % para un error promedio correspondiente de 0.00024.

De acuerdo a los resultados presentados, es posible concluir que el beneficio global que se puede lograr a través de la paralelización de este algoritmo depende no sólo de la complejidad de las operaciones involucradas en la función objetivo del problema que se esté abordando, sino que también influye el beneficio parcial que se puede obtener de la paralelización del resto de las operaciones evolutivas, gracias a las ventajas en el esquema de procesamiento en paralelo que brinda una arquitectura de hardware como la de las GPUs. De esta manera resulta conveniente el uso de estos dispositivos en la paralelización de algoritmos evolutivos, al ser capaces de obtener un beneficio global sustancial incluso para los casos en donde el beneficio durante la etapa de evaluación no resultó ser tan relevante.

Para el caso de la función de Griewank el mayor beneficio global obtenido fue para un número de evaluaciones $N_{eval} = 8,200,192$ en donde se obtuvo un factor de aceleración de 18.75, en comparación con la versión secuencial. En el caso de la función de Rosenbrock,

en donde no se logró un beneficio significativo durante la etapa de evaluación, se obtuvo un beneficio global máximo de 6.791 en el factor de aceleración cuando $N_{eval} = 147,464,192$, el cual fue obtenido básicamente gracias a la paralelización del resto de las operaciones evolutivas. Para la función de Ackley, en la cual se obtuvieron los factores de aceleración más altos durante la etapa de evaluación, el máximo beneficio se logró cuando $N_{eval} = 24,584,192$ al registrar un factor de aceleración de 14.436. Por último, para el caso de la función $f_4$ el mayor beneficio resultó para $N_{eval} = 16,392,192$ obteniendo un factor de aceleración global de 20.11.

Durante los experimentos realizados, fué posible observar una mayor precisión en las soluciónes obtenidas por parte de las versiones basadas en GPU. Esta diferencia en cuanto a precisión, pudiera estar asociada a la diferencia que debe existir entre los compiladores g++ (compilador de C) y nvcc (compilador de CUDA) en cuanto al manejo de los errores de redondeo durante la conversión de binario a flotante al realizar la aritmética de punto flotante.

En comparación con la implementación del esquema original de ED, la propuesta implementada en esta tesis mostró en general un desempeño superior, obteniendo en todos los casos un error promedio menor para un número igual de evaluaciones. Esto demuestra que a pesar de que la evolución diferencial es un método sencillo y rápido, en ciertas ocaciones es necesario un gran número de generaciones para lograr la convergencia, dependiendo en todo momento de los parámetros de ejecución que sean definidos. Para los casos de estudio analizados, la omisión de la operación de recombinación influyó en el esquema de busqueda para realizar una búsqueda más directa tratando de mantener suficiente diversidad en la población para evitar converger prematuramente. De esta manera, el algoritmo propuesto es adecuado para la solución de problemas de optimización global.

Capítulo 6

# Conclusiones y trabajos futuros

En este trabajo de tesis se ha presentado la implementación de un algoritmo de Evolución Diferencial basado en procesamiento paralelo y una plataforma GPU. La arquitectura masivamente paralela de estas tarjetas gráficas permite resolver eficientemente problemas que pueden ser expresados como cálculos de datos en paralelo en un esquema SIMD. El algoritmo de evolución diferencial se ajusta a este esquema de procesamiento paralelo ya que las operaciones involucradas en su desarrollo se refieren a la ejecución de la misma instrucción sobre múltiples datos. De tal modo que importantes operaciones evolutivas tales como la creación de la población inicial, la evaluación, la mutación y la selección de individuos fueron realizadas en paralelo empleando la arquitectura masivamente paralela de una GPU. Una variante del algoritmo original de ED es propuesta en este trabajo, la cual emplea un nuevo esquema durante la etapa de mutación y elimina la operación de recombinación. El objetivo de la variante al método de ED se realiza con dos objetivos: a).- reducir los parámetros de control del algoritmo, evitando la necesidad realizar extenuantes experimentos con la finalidad de ajustar los valores para lograr el mejor desempeño y b).- agregar un cierto nivel de voracidad a la estrategia de búsqueda, incluyendo información relevante durante la etapa de mutación y afectando la diversidad de la población al eliminar la constante de cruza.

**Conclusiones**

La implementación del algoritmo paralelo de ED basado en GPU presentado en este trabajo mostró un desempeño que superó ampliamente a versiones del mismo algoritmo basadas en CPU, en términos de tiempos de ejecución y factores de aceleración por parte de las versiones paralelas. Del mismo modo, la variante propuesta del algoritmo de ED presentó un desempeño sobresaliente en términos de velocidad de convergencia, precisión y eficacia, en comparación con el esquema original de ED. Las implementaciónes poseen una gran flexibilidad, en el sentido de que es relativamente sencillo cambiar el esquema de ED empleado. En este sentido, es posible regresar a los esquemas comunes de ED modificando unas cuantas líneas de código, en caso de que el esquema propuesto no satisfaga a los usuarios de la aplicación.

La evaluación de las implementaciones desarrolladas durante este trabajo de tesis se realizó empleando cuatro casos de estudio, los cuales involucran la optimización global de cuatro funciones continuas ampliamente utilizadas en la literatura especializada, tres de estas diferenciables. La dimensionalidad de los problemas fue establecida con $D = 128$ y fueron obtenidos resultados significativos, a pesar de que los tamaños de población empleados son relativamente pequeños en proporción al tamaño del espacio de búsqueda. El principal objetivo de esta implementación fue logrado, obteniendo en todos los casos un beneficio sustancial en cuanto a los tiempo de cómputo requeridos, los cuales varían en función de la complejidad de los cálculos involucrados en las evaluaciones de las funciones de aptitud y el total de evaluaciones realizadas.

Se reportaron resultados importantes en términos de tiempos de cómputo y factores de aceleración con la propuesta EDCP presentada en esta tesis. Los resultados obtenidos con la versión EDCP son superiores a las versiones paralelas, una basada en CPU y otra en GPU, ya que éstas últimas realizan únicamente de manera paralela la operación de evaluación. Los beneficios obtenidos en términos de factores de aceleración por parte de la propuesta desarrollada son de 6.791 para el caso de la función de Rosenbrock cuando el número de evaluaciones es $N_{eval}$ =147464192, un factor de aceleración máximo de 18.75 al establecer $N_{eval}$ =8200192 con la función de Griewank; un beneficio máximo de 14.436 para

la función de Ackley cuando $N_{eval} = 24584192$ y finalmente, el beneficio fue de 20.110 para el caso de la función $f_4$ cuando $N_{eval}$ =16392192.

Aún cuando la tolerancia máxima al error se definió como $tol = 1 \times 10^{-10}$, los resultados asociados a la presición y eficacia son relevantes, empleando la estrategia de búsqueda implícita en el esquema de ED propuesto. Se observó que el error disminuye mientras se eleva la eficacia del método al aumentar el número de evaluaciones realizadas por el algoritmo. Para los casos de las funciones de Griewank, Rosenbrock y la función $f_4$ los mejores resultados en términos de precisión con $N_p = 8192$ fueron los errores promedio 0.000211, 0.341711 y 2.144040E-13, mientras que para las eficacias se obtuvieron 97.14 %, 91.43 % y 100 %, respectivamente. La excepción resultó para el caso de la función de Ackley, para la cual la eficacia final fue de 0 % para el valor de tolerancia establecido, incluso al aumentar el parámetro del tamaño de población hasta $N_p = 65535$. Sin embargo el error promedio se logró disminuir hasta 0.000007.

En comparación con reportes previos acerca del algoritmo de ED basado en GPU, en la propuesta implementada en esta tesis se emplea un esquema más eficiente en la generación de números aleatorios. A diferencia de las propuestas anteriores, en donde la generación de números aleatorios se realiza de manera secuencial en la CPU o se realiza en la GPU a través de un esquema poco eficiente, la implementación propuesta hace uso de la biblioteca CURAND, recientemente incluida al entorno de programación CUDA. A través de esta herramienta, los números aleatorios pueden ser generados de manera paralela en la medida que se requieran.

A partir de los resultados obtenidos en términos de precisión, se tiene que las implementaciones basadas en GPU presentan en general un mejor desempeño que las versiones compiladas y ejecutadas en la CPU. Estos resultados sugieren que el compilador de CUDA es menos vulnerable a errores por redondeo y por lo tanto la aritmética de punto flotante es realizada de manera más eficiente en la GPU que en la CPU. Sin embargo, para comprobar esta hipótesis es necesario realizar experimentos adicionales enfocados a la evaluación de la precisión para poder establecer las diferencias en precisión que poseen los compiladores de ambas plataformas de cómputo.

**Trabajos futuros**

La presente tesis ha sido desarrollada con la finalidad de establecer las bases y una referencia para la solución de problemas que demanden un gran poder de cómputo y que puedan ser solucionados empleando técnicas de procesamiento en paralelo, particularmente a través del aprovechamiento de la arquitectura de las GPUs.

En lo que respecta al área de computación evolutiva, el trabajo futuro inmediato es el desarrollo, a partir de la la herramienta de optimización implementada, de otros algoritmos evolutivos, además de implementar las adecuaciones necesarias para poder realizar optimización con variables discretas, optimización con restricciones, optimización multimodal y optimización multiobjetivo. La evaluación del desempeño de dichas implementaciones debería realizarse a través de su aplicación para la solución de problemas en el mundo real. Una posibilidad de evaluación son los problemas de optimización en sistemas eléctricos de potencia tales como lo es la programación de generación y mantenimiento, despacho económico, reconfiguración y expansión de redes de distribución, ubicación de dispositivos de compensación de potencia reactiva y observadores para el diagnóstico de fallas dentro de las redes de distribución, predicción de carga y velocidades de viento.

Explorar la posibilidad de realizar paralelismo a dos niveles en un sistema de cómputo integrado por más de una CPU y una o más GPUs. Bajo este esquema cada *thread* en el CPU puede hacer uso simultánea e independientemente de la GPU, aprovechando así, los recursos de cómputo de que se dispone de una manera más eficiente. Un ejemplo del uso de este esquema de procesamiento puede ser el diseño y entrenamiento evolutivo de redes neuronales artificiales, en donde se requieren dos ciclos evolutivos anidados. Las arquitecturas de la red neuronal artificial podrían ser propuestas por un algoritmo evolutivo paralelo en la CPU mientras que el entrenamiento de las arquitecturas podría llevarse a cabo a través de un algoritmo evolutivo en la GPU, similar al propuesto en esta tesis.

Establecer un esquema de manejo de datos que permita hacer uso y explotar el tipo de dato `float4`. Este tipo de dato es una estructura que encapsula hasta cuatro valores de punto flotante y es ámpliamente utilizado en las GPUs en aplicaciónes gráficas ya que permite almacenar simultáneamente los valores de intensidad de los cuatro canales

del sistema RGBA. De esta manera cualquier operación realizada sobre un `float4` tiene el mismo costo que realizar la misma operación sobre un `float` sencillo dentro de la GPU. El uso de este tipo de datos permitiría, por ejemplo, la ejecución de un algoritmo evolutivo capaz de manejar hasta cuatro poblaciones o la ejecución de cuatro algoritmos evolutivos simultáneamente.

Finalmente, se plantea la realización de experimentos especializados que permitan realizar, por un lado, una evaluación específica de la calidad de los generadores de números aleatorios provistos por la biblioteca CURAND, a través del análisis de las propiedades estadísticas que estos generadores presenten durante la producción masivamente paralela de números aleatorios. Por otro lado, es necesario considerar la hipótesis de que la aritmética de punto flotante se lleva a cabo más eficientemente en la GPU que en la CPU. Dicha hipótesis debe ser probada a través de experimentos que permitan comparar de una manera directa ambos compiladores.

Apéndice A

# Códigos para realizar la suma de dos vectores y la suma de dos matrices de manera paralela en la GPU empleando CUDA

## A.1. Suma paralelizada de dos vectores en la GPU

```
#include <stdio.h>
#include <stdlib.h>

#define N 1024

/* Definición del Kernel */
__global__ void VecAdd(float *a, float *b, float *c){
        int i = threadIdx.x;

        c[i] = a[i] + b[i];
}
```

Listado A.1: Suma paralelizada de dos vectores de orden $N$ en la GPU.

```
/* Función Main */
int main (int argc, char **argv){
        int i;

        // Vectores en el host
        float *h_a, *h_b, *h_c;
        // Variables en el device
        float *d_a, *d_b, *d_c;

        // Vectores en el host
        h_a = (float *) malloc(N * sizeof(float));
        h_b = (float *) malloc(N * sizeof(float));
        h_c = (float *) malloc(N * sizeof(float));

        // Se reserva memoria en el device;
        cudaMalloc((void**) &d_a, N * sizeof(float));
        cudaMalloc((void**) &d_b, N * sizeof(float));
        cudaMalloc((void**) &d_c, N * sizeof(float));

        // Se llenan los vectores en el host con datos
        for(i=0; i<N; i++){
                h_a[i] = 1.0;
                h_b[i] = 2.0;
        }

        // Se copian los datos al device
        cudaMemcpy(d_a, h_a, N * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, h_b, N * sizeof(float), cudaMemcpyHostToDevice);

        /* Invocación del kernel de cómputo para realizar la suma */
        VecAdd<<<1, N>>>(d_a, d_b, d_c);

        // Se copia el resultado al host
        cudaMemcpy(h_c, d_a, N * sizeof(float), cudaMemcpyDeviceToHost);
```

Listado A.1: Suma paralelizada de dos vectores de orden $N$ en la GPU (continuación).

```
        /* Se imprime el resultado en pantalla */
        for(i=0; i<N; i++){
                printf("%f ", h_c[i]);
        }
        printf("\n");

        // Se libera la memoria en el device
        cudaFree(d_a);
        cudaFree(d_b);
        cudaFree(d_c);

        // Se libera la memoria en el host
        free(h_a);
        free(h_b);
        free(h_c);
}
```

Listado A.1: Suma paralelizada de dos vectores de orden $N$ en la GPU (continuación).

## A.2. Suma paralelizada de dos matrices en la GPU

```
#include <stdio.h>
#include <stdlib.h>

#define N 32

__global__ void MatAdd(float *A, float *B, float *C){
        int i = threadIdx.y;
        int j = threadIdx.x;

        C[i * N + j] = A[i * N + j] + B[i * N + j];
}

int main (int argc, char **argv){
        int i, j;

        // Matrices en el host
        float *h_A, *h_B, *h_C;
        // Matrices en el device
        float *d_A, *d_B, *d_C;

        // Se reserva memoria en el host
        h_A = (float *) malloc(N * N * sizeof(float));
        h_B = (float *) malloc(N * N * sizeof(float));
        h_C = (float *) malloc(N * N * sizeof(float));

        // Se reserva memoria en el device
        cudaMalloc((void**) &d_A, N * N * sizeof(float *));
        cudaMalloc((void**) &d_B, N * N * sizeof(float *));
        cudaMalloc((void**) &d_C, N * N * sizeof(float *));
```

Listado A.2: Suma paralelizada de dos matrices de orden $N \times N$ en la GPU.

```
        // Llenamos las matrices con datos
        for(i=0; i<N; i++)
                for(j=0; j<N; j++){
                        h_A[i * N + j] = 1.0;
                        h_B[i * N + j] = 2.0;
                }

        // Se copian los datos al device
        cudaMemcpy(d_A, h_A, N * N * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, N * N * sizeof(float), cudaMemcpyHostToDevice);

        // Se establece un grid unidimensional de orden 1
        dim3 GridDim(1);
        // Se establece un bloque de threads de orden N×N
        dim3 BlockDim(N, N);

        /* Invocación del kernel de cómputo para realizar la suma */
        MatAdd<<<GridDim, BlockDim>>>(d_A, d_B, d_C);

        // Se copian el resultado al host
        cudaMemcpy(h_C, d_C, N * N * sizeof(float), cudaMemcpyDeviceToHost);

        // Se imprime el resultado en pantalla
        for(i=0; i<N; i++){
                for(j=0; j<N; j++){
                        printf("%f ", h_C[i * N + j]);
                }
                printf("\n");
        }
```

Listado A.2: Suma de dos matrices de orden $N \times N$ en la GPU (continuación).

```
        // Se libera la memoria en el device
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);
        // Se libera la memoria en el host
        free(h_A);
        free(h_B);
        free(h_C);
}
```

Listado A.2: Suma de dos matrices de orden $N \times N$ en la GPU (continuación).

Apéndice B

# Código en CUDA del método de Evolución Diferencial Paralelo basado en GPU

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <string.h>
#include <curand_kernel.h>

#define N_p 8192 // Tamaño de la población
#define D 18 // Número de dimensiones
#define G 10000 // Número máximo de generaciones
//#define F 0.3
//#define Cr 0.5
#define BLOCK_SIZE 16 // Tamaño de los bloques de threads
#define BLOCK_SIZE_X 16 // Tamaño de los bloques de threads en la dimensión x
#define BLOCK_SIZE_Y 1 // Tamaño de los bloques de threads en la dimensión y
⋮
```

Listado B.1: Establecimiento de parámetros globales de ejecución.

```
using namespace std;

/* Se establecen los límites inferior y superior del
   espacio de búsqueda en cada dimensión */
__constant__ float d_Restrictions[D][2] = {{-600,600},{-600,600},{-600,600},
                                           {-600,600},{-600,600},{-600,600},
                                           {-600,600},{-600,600},{-600,600},
                                           {-600,600},{-600,600},{-600,600},
                                           {-600,600},{-600,600},{-600,600},
                                           {-600,600},{-600,600},{-600,600}};

// Se declara la textura bidimensional donde se almacenará la población
texture<float, 2, cudaReadModeElementType> texPopulation;

// Se declara el cudaArray para almacenar la población en la textura
cudaArray *Population_array;
/* Descripción del formato de canales que
 se manejará en la textura y el cudaArray */
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0, 0,
                                                  cudaChannelFormatKindFloat);

// Función que establece attributos y dimensiones del cudaArray y la textura
void Set_Attributes(){
        // Se establecen los atributos de la textura
        texPopulation.addressMode[0] = cudaAddressModeWrap;
        texPopulation.addressMode[1] = cudaAddressModeWrap;
        texPopulation.filterMode = cudaFilterModeLinear;
        texPopulation.normalized = false;

        // cudaArray  y textura de (D×N_p)
        cudaMallocArray( &Population_array, &channelDesc, D, N_p);
}
⋮
```

Listado B.2: Variables globales y funciones auxiliares.

```
⋮
/* Función que libera los recursos de memoria
 empleados por la textura y el cudaArray */
void Free_Memory(){
        // Se libera la memoria utilizada por la textura
        cudaUnbindTexture(texPopulation);

        // Se libera la memoria utilizada por el cudaArray
        cudaFreeArray(Population_array);
}

// Función que muestra la mejor solución y su valor de aptitud
void Print_BestSolution(float *BestSolution, float *Fitness_BestSolution,
                                float *Population, int Index_BestSolution){
        int j;
        cudaMemcpy(BestSolution, &Population[Index_BestSolution * D],
                            D * sizeof(float), cudaMemcpyDeviceToHost);

        printf("Best--------->{");
        for(j=0; j<D; j++)
                        cout<< BestSolution[j] << "␣";
        cout<< "}\t|\t" << *Fitness_BestSolution << "\n";
}

void Get_Index_BestSolution(int *Index_BestSolution, float *Fitness_BestSolution,
                                                                float *devFitness){
        int i;
        float *h_Fitness = (float *) malloc(N_p * sizeof(float));

        cudaMemcpy(h_Fitness, devFitness, N_p * sizeof(float),
                                            cudaMemcpyDeviceToHost);
        ⋮
```

Listado B.2: Variables globales y funciones auxiliares (continuación).

```
        ⋮
        // ...continúa Get_Index_BestSolution
        *Fitness_BestSolution = h_Fitness[0];
        *Index_BestSolution = 0;

        for (i=1; i<N_p; i++)
                if (h_Fitness[i] < *Fitness_BestSolution){
                        *Index_BestSolution = i;
                        *Fitness_BestSolution = h_Fitness[i];
                }

        free(h_Fitness);
}

Update_BestSolution(int *Index_BestSolution, float *Fitness_BestSolution,
                                        float *Population, float *Fitness){
        float fb;

        Get_Index_BestSolution(Index_BestSolution, &fb, Fitness);

        if (fb < *Fitness_BestSolution){
                *Fitness_BestSolution = fb;
                cudaMemcpyToSymbol("d_BestSolution",
                                &Population[Index_BestSolution[0] * D],
                                D * sizeof(float), 0, cudaMemcpyDeviceToDevice);
        }
}
⋮
```

Listado B.2: Variables globales y funciones auxiliares (continuación).

```
// Función que genera números aleatorios enteros en el rango [lowest,highest]
__device__ void RandInt(int *rnd, int lowest, int highest, curandState *state){
        unsigned int aux = curand(state);
        *rnd = (int) ((aux %((highest+1) - lowest)) + lowest);
}
/* Función que genera números aleatorios flotantes
 con distribución uniforme en (0.0f, 1.0f] */
__device__ void Random(float *rnd, curandState *state){
        *rnd = curand_uniform(state);
}

/* Kernel para la creación de los generadores
   de números aleatorios */
__global__ void SetupRNGs_Kernel(curandState *state, unsigned long *seeds){
        // Identificador único del generador que se va crear
        int i = blockIdx.x * blockDim.x + threadIdx.x;

        /* Si el índice no está más allá del tamaño
          del arreglo de generadores requeridos */
        if (i < N_p){
                /* Se inicializa el estado del generador
                   empleando la semilla correspondiente */
                curand_init(seeds[i], i, 0, &state[i]);
        }
}
/* Función que invoca el kernel para crear
   los generadores de números aleatorios */
void SetupRNGs(curandState *devStates){
        int i, GRID_SIZE;
        float aux, aux2, aux3;

        unsigned long  *d_seeds, *h_seeds;
        ⋮
```

Listado B.3: Funciones para la generación de números aleatorios.

```
        ⋮
        // ...continúa SetupRNGs
        /* Se generan N_p semillas de manera aleatoria
           empleando la librería estándar del lenguaje C */
        srand((unsigned) time(0));
        hostseeds = (unsigned long *) malloc(N_p * sizeof(unsigned long));
        for(i=0; i<N_p; i++)
                hostseeds[i] = (unsigned long) rand();

        // Se copian las semillas al device
        cudaMalloc((void **)&d_seeds, N * sizeof(unsigned long);
        cudaMemcpy(devseeds, h_seeds,  N * sizeof(unsigned long),
                                        cudaMemcpyHostToDevice);

        // Se calcula el tamaño del grid
        aux2 = N_p;
        aux3 = BLOCK_SIZE;
        aux = aux2/aux3;
        GRID_SIZE =(int) ceil(aux);
        dim3 dimGrid(GRID_SIZE);
        dim3 dimBlock(BLOCK_SIZE);

        /* Se invoca el kernel para crear los generadores */
        cudaThreadSetLimit(cudaLimitStackSize, 17*1024);
                SetupRNGs_Kernel<<<dimGrid, dimBlock>>>(devStates, devseeds);
        cudaThreadSetLimit(cudaLimitStackSize, 1024);

        // Se libera la memoria en el host y el device
        cudaFree(devseeds);
        free(hostseeds);
}
```

Listado B.3: Funciones para la generación de números aleatorios (continuación).

```
// Función para crear un individuo o agente
__device__ void Create_Ind(float *Ind, curandState *state){
        int j;
        float aux;

        // Se almacena el estado del generador en memoria local
        curandState localstate = *state;

        /* Se genera un valor aleatorio para cada componente del individuo
           de acuerdo a los límites establecidos en cada dimensión */
        for(j=0; j<L_IND; j++){
                Random(&aux, &localstate);
                Ind[j] = aux * (d_Restrictions[j][1] - d_Restrictions[j][0])
                                                    + d_Restrictions[j][0];
        }
        // Se almacena el estado del generador en memoria global
        *state = localstate;
}
/* Kernel para la creación de la población inicial */
__global__ void InitPop_Kernel(float *pop, curandState *states){
        // Índice del individuo o agente que se va crear
        int i =  blockIdx.x * blockDim.x + threadIdx.x;

        /* Si el índice no está más allá
           del tamaño de la poblaci'on */
        if (i < N_p){
                // Se crea el individuo
                CreateIndf(&pop[i * L_IND], &states[i]);
        }
}
```

Listado B.4: Funciones para la creación de la población inicial.

```
void CreateInitPop(float *Population, curandState *States){
        int GRID_SIZE;

        float aux, aux2, aux3;

        aux = N;
        aux2 = BLOCK_SIZE;
        aux3 = aux/aux2;

        // Se calcula el tamaño del grid
        GRID_SIZE =(int) ceil(aux3);

        // Se establecen las dimensiones del grid y los bloques
        dim3 dimGrid(GRID_SIZE);
        dim3 dimBlock(BLOCK_SIZE);

        /*Se invoca el kernel para crear la población inicial*/
        InitPop_Kernel<<<dimGrid, dimBlock>>>(Population, States);
}
```

Listado B.4: Funciones para la creación de la población inicial (continuación).

```
__device__ void Ackley(float *fitness, float y){
        int x, i;
        float s1, s2, a, b;
        float c = 2.0f * M_PI;

        __shared__ float sum1[BLOCK_Y][BLOCK_X];
        __shared__ float sum2[BLOCK_Y][BLOCK_X];

        sum1[threadIdx.y][threadIdx.x] = 0.0;
        sum2[threadIdx.y][threadIdx.x] = 0.0;
        ⋮
```

Listado B.5: Funciones de prueba para la aplicación.

```
        ⋮
        // ...continúa Ackley
        for(x=threadIdx.x; x<D; x+=blockDim.x){
                sum1[threadIdx.y][threadIdx.x] +=
                                powf(tex2D(texPopulation, x+0.5f, y+0.5f), 2.0f);
                sum2[threadIdx.y][threadIdx.x] +=
                                cosf(c * tex2D(texPopulation, x+0.5f, y+0.5f));
        }
        __syncthreads();

        if(threadIdx.x == 0){
                s1 = 0.0f;
                s2 = 0.0f;
                a = 20.0f;
                b = 0.2f;
                for(i=0; i<BLOCK_X; i++){
                        s1 += sum1[threadIdx.y][i];
                        s2 += sum2[threadIdx.y][i];
                }
                *fitness = -a*expf(-b * sqrtf((1.0f / D) * s1)) -
                                expf((1.0f / D) * s2) + a + expf(1.0f);
        }
}

__device__ void Rosenbrock(float *fitness, float y){
        int x, i;
        float s;

        __shared__ float sum[BLOCK_Y][BLOCK_X];

        sum[threadIdx.y][threadIdx.x] = 0.0;
        ⋮
```

Listado B.5: Funciones de prueba para la aplicación (continuación).

```
        ⋮
        // ...continúa Rosenbrock
        for(x=(threadIdx.x+1); x<D; x+=blockDim.x){
                sum[threadIdx.y][threadIdx.x] +=
                        100*powf(tex2D(texPopulation, x+0.5f, y+0.5f) -
                        powf(tex2D(texPopulation, (x-1)+0.5f, y+0.5f),2),2) +
                        powf((tex2D(texPopulation, (x-1)+0.5f, y+0.5f)-1),2);
        }
        __syncthreads();

        if(threadIdx.x == 0){
                s=0.0;
                for(i=0; i<BLOCK_X; i++){
                        s += sum[threadIdx.y][i];
                }
                *fitness = s;
        }
}
__device__ void Griewank(float *fitness, float y){
        int x, i;
        float s, fr, p;
        __shared__ float sum[BLOCK_Y][BLOCK_X];
        __shared__ float pro[BLOCK_Y][BLOCK_X];
        sum[threadIdx.y][threadIdx.x] = 0.0;
        pro[threadIdx.y][threadIdx.x] = 1.0;

        for(x=threadIdx.x; x<D; x+=blockDim.x){
                sum[threadIdx.y][threadIdx.x] +=
                        powf(tex2D(texPopulation, x+0.5f, y+0.5f), 2);
                pro[threadIdx.y][threadIdx.x] *=
                        cosf(tex2D(texPopulation, x+0.5f, y+0.5f)/sqrtf(x+1.0f));
        }
        ⋮
```

Listado B.5: Funciones de prueba para la aplicación (continuación).

```
        ⋮
        // ...continúa Griewank
        __syncthreads();

        if(threadIdx.x == 0){
                fr = 4000.0;
                s = 0.0;
                p = 1.0;
                for(i=0; i<BLOCK_X; i++){
                        s += sum[threadIdx.y][i];
                        p *= pro[threadIdx.y][i];
                }
                *fitness = ((s/fr) - p + 1.0f);
        }
}

__device__ float U(float x, int a, int k, int m){
        if (x>a)
                return k*powf((x-a),m);
        else if (x<-a)
                return k*powf((-x-a),m);
        else
                return 0.0f;
}
__device__ void F4(float *fitness, float y){
        int x, i;
        float s1, s2;
        __shared__ float sum1[BLOCK_Y][BLOCK_X];
        __shared__ float sum2[BLOCK_Y][BLOCK_X];

        sum1[threadIdx.y][threadIdx.x] = 0.0;
        sum2[threadIdx.y][threadIdx.x] = 0.0;
        ⋮
```

Listado B.5: Funciones de prueba para la aplicación (continuación).

```
        ⋮
        // ...continúa F4
        for(x=threadIdx.x; x<(D-1); x+=blockDim.x){
                sum1[threadIdx.y][threadIdx.x] +=
                 (tex2D(texPopulation, x+0.5f, y+0.5f)-1) *
                 (tex2D(texPopulation, x+0.5f, y+0.5f)-1) *
                 (1+sinf(3.0f*M_PI*(tex2D(texPopulation,(x+1)+0.5f, y+0.5f)-1)) *
                 sinf(3.0f*M_PI*(tex2D(texPopulation,(x+1)+0.5f, y+0.5f)-1)));
        }
        for(x=threadIdx.x; x<D; x+=blockDim.x){
                sum2[threadIdx.y][threadIdx.x] +=
                        U(tex2D(texPopulation, x+0.5f, y+0.5f), 5, 100, 4);
        }
        __syncthreads();

        if(threadIdx.x == 0){
                s1 = 0.0;
                s2 = 0.0;
                for(i=0; i<BLOCK_X; i++){
                        s1 += sum1[threadIdx.y][i];
                        s2 += sum2[threadIdx.y][i];
                }
                *fitness= 0.1*(sinf(3.0f*M_PI*tex2D(texPopulation, 0.5, y+0.5f))*
                  sinf(3.0f*M_PI*tex2D(texPopulation, 0.5, y+0.5f))) + s1 +
                  (tex2D(texPopulation, D-0.5f, y+0.5f)-1) *
                  (tex2D(texPopulation, D-0.5f, y+0.5f)-1) *
                  (1.0f+sinf(2.0f*M_PI*tex2D(texPopulation, D-0.5f, y+0.5f))) +
                  s2;
        }
}
```

Listado B.5: Funciones de prueba para la aplicación (continuación).

```
/* KERNEL DE EVALUACIÓN */
__global__ void Evaluation_Kernel(float *Fit){
        // Se obtiene el índice del individuo que se evaluará
        int i = blockIdx.x * blockDim.y + threadIdx.y;

        // Si el índice ino se encuentra más allá del tamaño de la población
        if (i < N_p){
                /* Se llama la función de aptitud, especificando el índice
                   del individuo que va ser evaluado */
                Griewank(&Fit[i], i);

                // Rosenbrock(&Fit[i], i);

                // Ackley(&Fit[i], i);

                // F4(&Fit[i], i);
        }
}

/* FUNCIÓN QUE INVOCA EL KERNEL DE EVALUACIÓN */
void Evaluation(float *Fitness, float *Population){
        int GRID_SIZE;
        float aux, aux2, aux3;

        // Se copia la población al cudaArray
        cudaMemcpyToArray(Population_array, 0, 0, Population,
                                                N_p * D * sizeof(float),
                                                cudaMemcpyDeviceToDevice);
        // Se liga el cudaArray a la textura
        cudaBindTextureToArray( texPopulation, Population_array, channelDesc);
        ⋮
```

Listado B.6: Funciones para realizar la evaluación.

```
        ⋮
        // ... continúa Evaluation

        aux = N_p;
        aux2 = BLOCK_SIZE;
        aux3 = aux/aux2;

        /*Se calcula el tamaño del grid en función de  el tamaño
         de la población y el tamaño de los bloques de threads */
        GRID_SIZE =(int) ceil(aux3);

        // Se establecen las dimensiones del grid y los bloques de threads
        dim3 dimGrid(GRID_SIZE);
        dim3 dimBlock(BLOCK_SIZE);

        /* INVOCACIÓN DEL KERNEL DE EVALUACIÓN */
        Evaluation_Kernel<<<dimGrid, dimBlock>>>(Fitness);
}
```

Listado B.6: Funciones para realizar la evaluación (continuación).

```
__global__ void Mutation_Kernel(float *TestPopulation, curandState *States){
        // Se obtiene el índice del individuo que se mutará
        int i = blockIdx.x * blockDim.x + threadIdx.x;

        int j,
        int r1, r2;
        //int r3;

        float lambda;

        curandState localstate = States[i];
        ⋮
```

Listado B.7: Funciones para realizar la mutación.

```
        ⋮
        // Si el índice i no se encuentra más allá del tamaño de la población
        if (i < N_p){
          r1 = r2 = i;
          //r3 = i;

          while (r1 == i)
            RandInt(&r1, 0, N_p-1, &localstate);
          while ((r2 == i) || (r2 == r1))
            RandInt(&r2, 0, N_p-1, &localstate);
          /*
          while ((r3 == i) || (r3 == r1) || (r3 == r2))
            RandInt(&r3, 0, N-1, &localstate);
          */

          for(j=0; j<D; j++){
            /***** New MUTATION Scheme for DIFFERENTIAL EVOLUTION *****/
              Random(&lambda, &localstate);
              TestPopulation[i * D + j] =  tex2D(texPopulation, j+0.5f, i+0.5f)
                  + lambda*(tex2D(texPopulation, j+0.5f, r1+0.5f)
                    - tex2D(texPopulation, j+0.5f, r2+0.5f))
                  + (1-lambda)*(d_BestSolution[j]
                        - tex2D(texPopulation, j+0.5f, i+0.5f));

            /*** Original MUTATION Scheme for DIFFERENTIAL EVOLUTION ***/
            /* TestPopulation[i * D + j] = tex2D(texPopulation, j+0.5f, r3+0.5f)
                + F*(tex2D(texPopulation, j+0.5f, r1+0.5f)
                - tex2D(texPopulation, j+0.5f, r2+0.5f)); */
          }

          States[i] = localstate;
        }
}
```

Listado B.7: Funciones para realizar la mutación (continuación).

```
extern "C" void Mutation(float *TestPopulation, float *Population,
                                                curandState *devStates){
        int GRID_SIZE;

        float aux, aux2, aux3;

        // Se copia la población al cudaArray
        cudaMemcpyToArray(Population_array, 0, 0, Population,
                                                 N_p * D * sizeof(float),
                                                 cudaMemcpyDeviceToDevice);
        // Se liga el cudaArray a la textura
        cudaBindTextureToArray( texPopulation, Population_array, channelDesc);

        aux = N_p;
        aux2 = BLOCK_SIZE;
        aux3 = aux/aux2;

        // Se calcula el tamaño del GRID
        GRID_SIZE =(int) ceil(aux3);

        // Se establecen las dimensiones del grid y los bloques de threads
        dim3 dimGrid(GRID_SIZE);
        dim3 dimBlock(BLOCK_SIZE);

        /* INVOCACIÓN DEL KERNEL DE MUTACÓN */
        Mutation_Kernel<<<dimGrid, dimBlock>>>(TestPopulation, devStates);

}
```

Listado B.7: Funciones para realizar la mutación (continuación).

```
/* KERNEL DE RECOMBINACIÓN */
__global__ void CrossOver_Kernel(float *TestPopulation, curandState *states){
        // Se obtiene el índice de los individuos que se cruzarán
        int j, i = blockIdx.x * blockDim.x + threadIdx.x;

        float aux;
        curandState localstate;

        // Si el índice ino se encuentra más allá del tamaño de la población
        if (i < N_p){
          localstate = states[i];
          for(j=0; j<D;j++){
            Random(&aux, &localstate);
              if(aux > CR)
                TestPopulation[i*D + j] = tex2D(texPopulation, j+0.5f, i+0.5f);
          }

          states[i] = localstate;
        }
}

/* FUNCIÓN QUE INVOCA EL KERNEL DE RECOMBINACIÓN */
extern "C" void CrossOver(float *TestPopulation, float *Population,
                                                 curandState *devStates){
        int GRID_SIZE;
        float aux, aux2, aux3;
        // Se copia la población al cudaArray
        cudaMemcpyToArray(Population_array, 0, 0, Population,
                                                 N_p * D * sizeof(float),
                                                 cudaMemcpyDeviceToDevice);
        // Se liga el cudaArray a la textura
        cudaBindTextureToArray( texPopulation, Population_array, channelDesc);
        ⋮
```

Listado B.8: Funciones para realizar la recombinación.

```
        ⋮
        aux = N_p;
        aux2 = BLOCK_SIZE;
        aux3 = aux/aux2;

        /*Se calcula el tamaño del grid en función de  el tamaño
         de la población y el tamaño de los bloques de threads */
        GRID_SIZE =(int) ceil(aux3);

        // Se establecen las dimensiones del grid y los bloques de threads
        dim3 dimGrid(GRID_SIZE);
        dim3 dimBlock(BLOCK_SIZE);

        /* INVOCACIÓN DEL KERNEL DE RECOMBINACIÓN */
        CrossOver_Kernel<<<dimGrid, dimBlock>>>(TestPopulation, devStates);
}
```

Listado B.8: Funciones para realizar la recombinación (continuación).

```
/* KERNEL DE SELECCIÓN */
__global__ void Selection_Kernel(float *Population, float *Fitness,
                                                 float *Fitness_TestPopulation){
        // Se obtiene el índice del individuo que se evaluará
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int j;

        if (j < N_p){
          if (Fitness_TestPopulation[y] < Fitness[y]) {
            for(j=0; j<D; j++)
              Population[i * D + j] = tex2D(texPopulation, j+0.5f, i+0.5f);
            Fitness[i] = Fitness_TestPopulation[i];
          }
        }
}
```

Listado B.9: Funciones para realizar la selección.

```
/* FUNCIÓN QUE INVOCA EL KERNEL DE EVALUACIÓN */
extern "C" void Selection(float *Population, float *Fitness,
                          float *TestPopulation, float *Fitness_TestPopulation){
        int GRID_SIZE;

        float aux, aux2, aux3;

        // Se copia la población al cudaArray
        cudaMemcpyToArray(Population_array, 0, 0, Population,
                                                N_p * D * sizeof(float),
                                                cudaMemcpyDeviceToDevice);

        // Se liga el cudaArray a la textura
        cudaBindTextureToArray( texPopulation, Population_array, channelDesc);

        aux = N_p;
        aux2 = BLOCK_SIZE;
        aux3 = aux/aux2;

        /*Se calcula el tamaño del grid en función de  el tamaño
         de la población y el tamaño de los bloques de threads */
        GRID_SIZE =(int) ceil(aux3);

        // Se establecen las dimensiones del grid y los bloques de threads
        dim3 dimGrid(GRID_SIZE);
        dim3 dimBlock(BLOCK_SIZE);

        /* INVOCACIÓN DEL KERNEL DE SELECCIÓN */
        Selection_Kernel<<<dimGrid, dimBlock>>>(Population, Fitness,
                                                        Fitness_TestPopulation);
}
```

Listado B.9: Funciones para realizar la selección (continuación).

```
/***** FUNCIÓN PRINCIPAL *****/
int main(int argc, char **argv){
        float Time, TotalTime;
        cudaEvent_t startT, start, stopT, stop;
        cudaEventCreate(&startT); cudaEventCreate(&stopT);
        cudaEventCreate(&start); cudaEventCreate(&stop);

        // Se inicia el temporizador global
        cudaEventRecord( startT, 0 );

        // Tamaño en bytes de un individuo
        size_t size = D * sizeof(float);

        float *Population, *TestPopulation;
        float *Fitness, *Fitness_TestPopulation;
        float *BestSolution = (float *) malloc(size);
        float *Fitness_BestSolution = (float *) malloc(sizeof(float));
        int i, Index_BestSolution, g = 0;

        /***** Se establecen los generadores de números aleatorios *****/
         // Reservamos memoria para N_p estados iniciales
        curandState *devStates;
        cudaMalloc((void **)&devStates, N_p * sizeof(curandState));
         // Se inicia temporizador para crear los generadores
        cudaEventRecord( start, 0 );

                SetUpRNG(devStates);

        cudaEventRecord( stop, 0 );
        cudaEventSynchronize( stop );
        cudaEventElapsedTime( &Time, start, stop );
        printf("\nCREATION OF RNGS: %f milliseconds\n", Time);
        ⋮
```

Listado B.10: Función principal.

```
        ⋮
        /***** Se crea la población inicial *****/
         // Resevamos memoria para N_p individuos
        cudaMalloc((void**)&Population, N_p * size);
         // Se inicia temporizador para crear la población inicial
        cudaEventRecord( start, 0 );
                Create_InitialPopulation(Population, devStates);
        cudaEventRecord( stop, 0 );
        cudaEventSynchronize( stop );
        cudaEventElapsedTime( &Time, start, stop );
        printf("\nCREACION POBLACION INICIAL: %f milliseconds\n", Time);

        // Se definen los atributos del cudaArray y la textura empleada
        Set_Attributes();

        /***** Se evalúa la población inicial *****/
         // Resevamos memoria para almacenar la aptitud de N_p individuos
        cudaMalloc((void**)&Fitness, N_p * sizeof(float));
         // Se inicia temporizador para evaluar la población inicial
        cudaEventRecord( start, 0 );
                Evaluation(Fitness, Population);
        cudaEventRecord( stop, 0 );
        cudaEventSynchronize( stop );
        cudaEventElapsedTime( &Time, start, stop );
        printf("\nEVALUACION DE LA POBLACION: %f milliseconds\n", Time);

        /*** Se encuentra el índice de la mejor soluci'on ***/
        Get_Index_BestSolution(&Index_BestSolution,Fitness_BestSolution,Fitness);
                // Se copia la mejor solución a MEMORIA CONSTANTE
        cudaMemcpyToSymbol("d_BestSolution", &Population[Index_BestSolution * D],
                                                size, 0, cudaMemcpyDeviceToDevice);
        ⋮
```

Listado B.10: Función principal (continuación).

```
        ⋮
        printf("\nGen ---------> %d\n", g);
        Print_BestSolution(BestSolution,Fitness_BestSolution,Population,
                                                        Index_BestSolution);
        cudaMalloc((void**)&TestPopulation, N_p * size;
        cudaMalloc((void**)&Fitness_TestPopulation, N_p * sizeof(float));

        /********** DIFERENTIAL EVOLUTION **********/
         // Criterio de terminación
        while ((g<=G) && (Fitness_BestSolution[0] > 1e-10)) {
                /***** Mutación *****/
                cudaEventRecord( start, 0 );
                        Mutation(TestPopulation, Population, devStates);
                cudaEventRecord( stop, 0 );
                cudaEventSynchronize( stop )    ;
                cudaEventElapsedTime( &Time, start, stop );

                printf("\nMUTACIÓN: %f milliseconds\n", Time);

                /***** Recombinación *****/
                /*
                cudaEventRecord( start, 0 );

                        CrossOver(TestPopulation, Population, devStates);

                cudaEventRecord( stop, 0 );
                cudaEventSynchronize( stop );
                cudaEventElapsedTime( &Time, start, stop );
                printf("\nRECOMBINACIÓN: %f milliseconds\n", Time);
                */
                ⋮
```

Listado B.10: Función principal (continuación).

```
        ⋮
        /***** Evaluación *****/
        cudaEventRecord( start, 0 );
                Evaluation(Fitness_TestPopulation, TestPopulation);
        cudaEventRecord( stop, 0 );
        cudaEventSynchronize( stop );
        cudaEventElapsedTime( &Time, start, stop );

        printf("\nEVALUACIÓN: %f milliseconds\n", Time);

        /***** Selección *****/
        cudaEventRecord( start, 0 );
                Selection(Population, Fitness, TestPopulation,
                                                Fitness_TestPopulation);
        cudaEventRecord( stop, 0 );
        cudaEventSynchronize( stop );
        cudaEventElapsedTime( &Time, start, stop );
        printf("\nSELECCIÓN: %f milliseconds\n", Time);

        // Se actualiza la mejor solución
        Update_BestSolution(&Index_BestSolution, Fitness_BestSolution,
                                                  Population, Fitness);

        // Se actualiza el contador de generaciones
        g++;

        printf("\nGen ---------> %d\n", g);
        // Se imprime la mejor solución encontrada
        Print_BestSolution(BestSolution, Fitness_BestSolution,
                                Population, Index_BestSolution);
    }
    ⋮
```

Listado B.10: Función principal (continuación).

```
        ⋮
        /*
        printf("\nGen --------->%d\n", g);
        // Se imprime la mejor solución encontrada
                Print_BestSolution(BestSolution, Fitness_BestSolution,
                                        Population, Index_BestSolution);
        */

        // Se liberan los espacios de memoria empleados por la textura
        Free_Memory();

        // Se libera la memoria reservada
        checkCUDACall(cudaFree(devStates), __LINE__);
        checkCUDACall(cudaFree(Population),__LINE__);
        checkCUDACall(cudaFree(Fitness),__LINE__);
        checkCUDACall(cudaFree(TestPopulation),__LINE__);
        checkCUDACall(cudaFree(Fitness_TestPopulation),__LINE__);
        free(BestSolution);
        free(Fitness_BestSolution);

        // Se detiene el temporizador global
        cudaEventRecord( stopT, 0 );
        cudaEventSynchronize( stopT );
        cudaEventElapsedTime( &TotalTime, startT, stopT );

        // Se eliminan los temporizadores
        cudaEventDestroy( start );
        cudaEventDestroy( stop );
        cudaEventDestroy( startT );
        cudaEventDestroy( stopT );

        printf("\nTIEMPPO TOTAL TRANSCURRIDO: %f milliseconds\n", TotalTime);
}
```

Listado B.10: Función principal (continuación).

# Referencias


[Aarts89] Aarts, E. y Korst, J., "*Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*", Wiley-Interscience series in discrete mathematics and optimization, John Wiley and Sons, 1989.

[Abido00] M.A. Abido y Y.L. Abdel-Magid "*Robust design of multimachine power system stabilisers using tabu search algorithm*", IEE Proceedings-Generation Transmission Distribution, Issue 6, vol. 147, págs 387–394, Noviembre, 2000.

[Amdahl67] Amdahl, G. M., "*Validity of the single processor approach to achieving large scalecomputing capabilities*", AFIPS spring joint computer conference, vol. 30, págs. 483–485, Atlantic City, NJ, USA, Abril 1967.

[Aravind02] Aravind, I., Chandra, C., Guruprasad, M., y Dev, P. "*Implementation of image segmentation and reconstruction using genetic algorithms*", IEEE International Conference on Industrial Technology, IEEE ICIT '02, vol. 2, págs. 970–975, Bankok, Thailand, Diciembre 2002.

[Arora10] Arora, R. T. y R. Deb, K. "*Parallelization of binary and real-coded genetic algorithms on gpu using cuda*", Congress on Evolutionary Computation, págs. 1–7, Barcelona, Spain, Julio 2010.

[Back94] T. Back, "*Selective pressure in evolutionary algorithms: A characteri-*

*zation of selection mechanisms*", 1st IEEE Conference on Evolutionary Computation, vol. 1, págs. 57–62, Orlando, Florida, USA, Agosto 1994.

[Brabazon08] A. Brabazon, M. OÑeill y I. Dempsey, "*An Introduction to Evolutionary Computation in Finance*", Computational Intelligence Magazine, IEEE, Issue 4, vol. 3, págs. 42 – 55, 2008.

[Burke00] Burke E. K. y Smith A. J. "*Hybrid evolutionary techniques for the maintenance scheduling problem*", IEEE Transactions on Power Systems, Issue 1, vol. 15, págs 122–128, Febrero, 2000.

[Cantú98] Cantú Paz, E. "*A survey of parallel genetic algorithms*", Department of Computer Science and Genetic Algorithms Laboratory, University of Illinois, 1998.

[Castillo07] M.G. Castillo Tapia y C.A.C. Coello, "*Applications of multi-objective evolutionary algorithms in economics and finance: A survey*", IEEE Congress on Evolutionary Computation (CEC), págs 532–539, Singapore, Singapore, Septiembre 2007.

[Cervantes10] J. Cervantes, M. Sánchez, P.P. González "*Emerging traits in the application of an evolutionary algorithm to a scalable bioinformatics problem*", IEEE Congress on Evolutionary Computation (CEC), págs. 1–8, Barcelona, España, Julio, 2010.

[Darwin59] Darwin, C., "*On the Origin of Species*", Noviembre 1859.

[Demir10] Demir, V. y A. Z. Elsherbeni, "*Programming finite-difference time-domain for graphics processor units using compute unified device architecture*", IEEE Antennas and Propagation Society International Symposium (APSURSI), págs. 1–4, Toronto, Ontario, Canada, Julio 2010.

[Días09] Días, J. L., "*Aplicación de Algoritmos Genéticos a la Estimación de Estado de Depresiones de Voltaje y la Reconfiguración de Redes Eléctri-*

*cas*", Tesis de Maestría, División de Estudios de Posgrado, Facultad de Ingeniería Eléctrica, U.M.S.N.H., Agosto 2009.

[Dorigo97] Dorigo, M. y Maria, G., "*Ant colony system: a cooperative learning approach to the traveling salesman problem*", IEEE Transactions on Evolutionary Computing, vol. 1, págs. 53–66, Abril 1997.

[Eberhart01] Eberhart, R. C., Shi, Y., y Kennedy, J., "*Swarm intelligence*", The Morgan Kaufmann Series in Evolutionary Computation, San Francisco, CA, USA, 2001.

[Espejo10] P.G. Espejo, S.Ventura y F. Herrera "*A Survey on the Application of Genetic Programming to Classification*", IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, Issue 2, vol. 4, págs 121–144, Marzo 2010.

[Fogel07] G. Fogel, D, Corne, y Y. Pan "*Evolutionary Feature Selection for Bioinformatics*", Computational Intelligence in Bioinformatics, págs 117–139, Wiley-IEEE Press, 2007

[Fonseca09] W.A. dos S. Fonseca, F.G.N. Barros, M.V.A. Nunes, U.H. Bezerra y R.C.L. Oliveira "*Genetic Algorithms and Treatment of Multiple Objectives in the Allocation of Capacitor Banks in an Electric Power Distribution System*", IEEE Bucharest PowerTech, págs 1–8, Bucharest, Romania, Junio 2009.

[Gallego09] L.A. Gallego, M.J. Rider, R. Romero y A.V. Garcia "*A Specialized Genetic Algorithm to Solve the Short Term Transmission Network Expansion Planning*", IEEE Bucharest PowerTech, págs 1–7, Bucharest, Romania, Junio 2009.

[García10] García, N., "*Parallel power flow solutions using a biconjugate gradient algorithm and a newton method: A gpu-based approach*", IEEE Power

and Energy Society General Meeting, págs. 1–4, Minneapolis, MN, USA, Julio 2010.

[Glover97] Glover, F. y Laguna, M., "*Tabu search*", Kluwer Academic Publishers Norwell, MA, USA, University of Colorado, Boulder, 1997.

[Gopal07] Gopal, A., Niebur, D., y S. Venkatasubramanian, "*Dc power flow based contingency analysis using graphics processing units*", IEEE Power Tech, págs. 731–736, Lausanne, Zwitzerland, Julio 2007.

[Gopal10] Gopal, A., "*DC Power Flow Based Contingency Analysis Using Graphics Processing Units*", Tesis de Maestría, Drexel University, Marzo 2010.

[Guoyan06] Guoyan Yu, Zhen He, Chaoan Lai y Bing Lu "*The Application of Interactive Evolutionary Algorithm in Product Design*", The Sixth World Congress on Intelligent Control and Automation (WCICA), vol 2, págs 6758–6762, Dalian, China 2006.

[Haupt04] Haupt, R. L. y Haupt, S. E, "*Practical Genetic Algorithms*", WILEY-INTERSCIENCE, Hoboken, New Jersey, 2004.

[Holland75] Holland, J. H. "*Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*", Ann Arbor, MI: Univeristy of Michigan, 1975.

[Hongying10] Hongying Yuan y Jingsong He "*Evolutionary design of operational amplifier using variable-length differential evolution algorithm*", International Conference on Computer Application and System Modeling (ICCASM), vol 4, págs 610–614, North University of China,Taiyuan, China 2010.

[Huicong10] Huicong Wu y Xuejun Song "*Digital Filter Design Using Evolutionary Algorithms*", 2nd International Workshop on Intelligent Systems and Applications (ISA), págs 1–4, Wuhan, China, 2010.

[Jalili09] Jalili-Marandi, V. y Dinavahi, V., " *Large-scale transient stability simulation on graphics processing units*", IEEE Power and Energy Society General Meeting, págs. 1–6, Calgary, Alberta, Canada, Julio 2009.

[Jin07] Jin Yin, Wei Chen y Yun Li ; "*Evolutionary computation enabled game theory based modelling of electricity market behaviours and applications*", IEEE Congress on Evolutionary Computation (CEC), págs 1896–1903, Singapore, Singapore, Septiembre 2007.

[Kephart94] J.O. Kephart et al, "*A biologically inspired immune system for computers*", Proceedings of Artificial Life IV: The Fourth International Workshop on the Synthesis and Simulation of Living Systems, págs. 130–139, Cambridge, MA, USA, 1994.

[Khamsawang10] Khamsawang, S., Wannakarn, P., y Jiriwibhakorn, S., "*Hybrid pso-de for solving the economic dispatch problem with generator constraints*", The 2nd International Conference onComputer and Automation Engineering (ICCAE), vol. 5, págs. 135–139, Bangkok, Thailand, Febrero 2010.

[Kim97] Kim H., Hayashi Y. y Nara K. "*An algorithm for thermal unit maintenance scheduling through combined use of GA, SA and TS*", IEEE Transactions on Power Systems, Issue 1, vol. 12, págs. 329–335, Febrero, 1997.

[Kircpatrick83] Kircpatrick, S., Jr., C. D. G., y Vecchi, M. P., Optimization by simulated annealing. Inf. Téc. 4598, AAAS, Mayo 1983.

[Kirk10] Kirk, D. B. y mei W. Hwu, W., "*Programming Massively Parallel Processors: A hands-on Approach*", Morgan Kaufmann, Burlington, MA 01803, USA, 2010.

[Koza92] Koza, J. R., "*Genetic Programming: on the programming of computers by means of natural selection*", The MIT Press, 1992.

[Kwedlo06] Kwedlo, W. y Bandurski, K., "*A parallel differential evolution algorithm*", International Symposium on Parallel Computing in Electrical Engineering, págs. 319–324, Bialystok, Poland, Octubre 2006.

[Lang10] Lang Zhang, Haiqing Hu, Daohong Zhang y Liang Chen; "*Research on Small and Medium Enterprises Trade Credit Financing Based on Evolutionary Game*", International Conference on Information Management, Innovation Management and Industrial Engineering (ICIII), págs. 472–476, Kunming, China, Noviembre, 2010.

[Lee04] Lee, C.-Y. y Yao, X., "*Evolutionary programming using mutations based on the lévy probability distribution*", IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, vol. 8, págs. 1–13, February 2004.

[Lee08] Lee, K. Y. y El-Sharkawi, M. A., "*Modern Heuristics Optimization Techniques, Theory and applications to power systems*", WILEY-INTERSCIENCE, Hoes Lane Piscataway, NJ, USA, 2008.

[Li07] Li, J.-M., Wang, X.-J., He, R.-S., y Chi, Z.-X. " *An efficient fine-grained parallel genetic algorithm based on gpu-accelerated*", International Conference on Network and Parallel Computing Workshops, págs. 855–862, Dalian, China, Septiembre 2007.

[Lin10] Chuan Lin, Anyong Qing y Quanyuan Feng, "*A comparative study of crossover in differential evolution*", Journal of Heuristics, págs 1–29, http://dx.doi.org/10.1007/s10732-010-9151-1, 2010.

[Lirui08] Lirui, G., Limin, H., Liguo, Z., Weina, L., y Jie, H., "*Reactive power optimization for distribution systems based on dual population ant colony optimization*", 27th Chinese Control Conference, CCC, págs. 89–93, Kunming, China, Julio 2008.

[Mahdad09] Mahdad, B., Srairi, K., y Bouktir, T., "*Optimal power flow with environmental constraints of the algerian network using decomposed parallel*

*ga*", IEEE Power and Energy Society General Meeting, PES, págs. 1–8, Calgary, Alberta Canada, Julio 2009.

[McGregor92] McGregor, D., Odetayo, M., y Dasgupta, D., "*Adaptive control of a dynamic system using genetic-based methods*", Proceedings of the 1992 IEEE International Symposium on Intelligent Control, págs. 521–525, Glasgow, United Kingdom, Agosto 1992.

[Meng07] Meng, X. y Song, B. "*Fast genetic algorithms used for pid parameter optimization*", IEEE International Conference on Automation and Logistics, págs. 2144–2148, Jinan, China, Agosto 2007.

[Montgomery10] J. Montgomery y S. Chen., "*An analysis of the operation of differential evolution at high and low crossover rates*", IEEE Congress on Evolutionary Computation (CEC), págs. 1327–1334, Barcelona, España, Julio, 2010.

[Nickolls10] Nickolls, J. y Dally, W. J., "*The gpu computing era*", Computing Now Magazine, págs. 56–69, Marzo 2010.

[Niu06] Niu Xiao-shu y Ju Xiao-feng "*The Analyses of International Trade Relationship between China and Other Southeast Asian Countries Based on the Evolutionary Game Theory*", International Conference on Management Science and Engineering ICMSE, págs 1095–1100, Lille, France, Octubre, 2006.

[Nocedal99] Nocedal, J. y Wright, S. J., "*Numerical Optimization*", Springer, Springer-Verlag, New York, NY, USA, 1999.

[NVIDIA09] NVIDIA, "*NVIDIA Fermi Compute Architecture Whitepaper*", NVIDIA Corporation, Santa Clara, CA, 2009.

[NVIDIA10] NVIDIA, "*CUDA CURAND Library*", NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA, Agosto 2010.

[NVIDIA10a] NVIDIA, "*CUDA C Best Practices Guide*", NVIDIA Corporation, Santa Clara, CA, Agosto 2010.

[NVIDIA10b] NVIDIA, "*CUDA C Best Practices Guide*", NVIDIA Corporation, Santa Clara, CA, Agosto 2010.

[NVIDIA10c] NVIDIA, "*NVIDIA CUDA C GETTING STARTED GUIDE FOR MAC OS X*", NVIDIA Corporation, Santa Clara, CA, Agosto 2010.

[NVIDIA10d] NVIDIA, "*NVIDIA CUDA C Programming Guide*", NVIDIA Corporation, Santa Clara, CA , Septiembre 2010.

[NVIDIA11] NVIDIA, "*High performance computing, supercomputing with tesla gpus*", Enero 2011.
URL `http://www.nvidia.com/object/tesla_computing_solutions.html`

[Owens08] Owens, Houston, J., Luebke, M., Green, D., Stone, S., Phillips, J., y J.C., "*Gpu computing*", Proceedings of the IEEE, vol. 96, págs. 879–899, Mayo 2008.

[Pal06] S.K. Pal, S. Bandyopadhyay y S.S. Ray "*Evolutionary computation in bioinformatics: a review*", IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, Issue 5, vól 36, págs. 601–615, Septiembre, 2006.

[Parsopoulos02] Parsopoulos, K. E. y Vrahatis, M. N., "*Recent approaches to global optimization problems through particle swarm optimization*", Natural Computing: an international journal, Kluwer Academic Publishers, vol. 1, págs. 235–306, Netherlands 2002.

[Pei-Fang] Pei-Fang Guo, Bhattacharya, P y Kharma, N.; "*An efficient image pattern recognition system using an evolutionary search strategy*", IEEE International Conference on Systems, Man and Cybernetics (SMC), págs. 599–604, San Antonio, TX, USA, Octubre, 2009.

[Pit95] Pit, L. J., "*Parallel Genetic Algorithms*", Tesis de Maestría, Department of Computer Science, Leiden University, 1995.

[Price96] Price, K., "*Differential evolution: A fast and simple numerical optimizer*", North american Fuzzy Information Processing Society NAFIPS, págs. 524–527, Berkeley, California, USA, 1996.

[Price05] Price, K. V. y Storn, R. M., "*Differential Evolution - A practical approach to global optimization*", Studies in Computational Intelligence, Springer, Polish Academy of Sciences ul. Newelska, Warsaw, Poland, 2005.

[Qing06] Qing, A., "*Dynamic differential evolution strategy and applications in electromagnetic inverse scattering problems*", IEEE Transactions on Geoscience and Remote Sensing, vol. 44, págs. 116–125, Enero 2006.

[Halprin10] Rab Halprin y Moni Naor "*Games for extracting randomness*", XRDS, ACM, Issue 2, vol. 17, págs 44–48, New York, NY, USA, Diciembre, 2010.

[Rudolf99] Rudolf A. y Bayrleithner R. "*A genetic algorithm for solving the unit commitment problem of a hydro-thermal power system*", IEEE Transactions on Power Systems, Issue 4, vol. 14, págs. 1460–1468, Noviembre, 1999.

[Shi10] Shi, P. y Cui, Y., "*Dynamic path planning for mobile robot based on genetic algorithm in unknown environment*", 2010 Chinese Control and Decision Conference (CCDC), págs. 4325 – 4329, Xuzhou, China, Mayo 2010.

[Shinn10] Shinn-Ying Ho y Kual-Zheng Lee "*An efficient evolutionary image segmentation algorithm*", IEEE Congress on Evolutionary Computation (CEC), vol. 2, págs. 1327–1334, Barcelona, España, Julio, 2010.

[Soveiko10] Nick Soveiko, Michel S. Nakhla y Ramachandra Achar "*Comparison Study of Performance of Parallel Steady State Solver on Different Computer Architectures*", IEEE Transactions on Computer-Aided Design of

Integrated Circuits and Systems, Issue 1, vol. 29, págs 65–77, Enero, 2010.

[Spiegel92] Spiegel, M. R., "*Estadística*", McGraw Hill, 1992.

[Storn95] Storn, R. y Price, K., "*Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces*", International Computer Science Institute, Berkeley, CA, USA, Marzo 1995.

[Tasoulis04] Tasoulis, D., Pavlidis, N., Plagianakos, V., y Vrahatis, M., *Parallel differential evolution*", Congress on Evolutionary Computation, vol. 2, págs. 2023–2029, Portland, Oregon, USA, Junio 2004.

[Ting07] Lee, V.C.S., Ting Yean Tan y Kwok, T., *Evolutionary Finance Simulation of Long Term Equity Portfolio Management System*", International Conference on Service Systems and Service Management, págs. 1–6, Chengdu, China, Junio 2007.

[Tsong07] Tsong Yueh Chen y R. Merkel "*Quasi-Random Testing*", IEEE Transactions on Reliability, Issue 3, vol. 56, págs. 562–568, Septiembre, 2007.

[Unno09] Unno, M., Inoue, Y., y Asar, H., "*Gpgpu-fdtd method for 2-dimensional electromagnetic field simulation and its estimation*", IEEE 18th Conference on Electrical Performance of Electronic Packaging and Systems, EPEPS, Portland, OR, USA, págs. 239 – 242, Octubre 2009.

[Veronese10] de P. Veronese, L. y Krohling, R. A., "*Differential evolution algorithm on the gpu with c-cuda*", IEEE Congress on Evolutionary Computation (CEC), págs. 1–7, Barcelona, España, Julio, 2010.

[Villanueva10] Villanueva, M. L. C., "*Herramienta para el Análisis de Sistemas Dinámicos Mediante Diagramas de Bifurcación Basado en Metaheurísticas*", Tesis de Maestría, División de Estudios de Posgrado, Facultad de Ingeniería Eléctrica, U.M.S.N.H., Agosto 2010.

[Vitorino09] Vitorino, R., Neves, L., y Jorge, H., "*Network reconfiguration to improve reliability and efficiency in distribution systems*", IEEE Bucharest PowerTech, págs. 1–7, Bucharest Romania, Junio 2009.

[Wong95] K.P. Wong y Y.W. Wong "*Thermal generator scheduling using hybrid genetic/simulated-annealing approach*", IEE Proceedings-Generation, Transmission and Distribution, Issue 4, vol. 142, págs 372–380, Julio, 1995.

[Wong06] Wong, M.-L. y Wong, T.-T., "*Parallel hybrid genetic algorithms on consumer-level graphics hardware*", IEEE Congress on Evolutionary Computation, págs. 2973 – 2980, Vancouver, British Columbia, Canada, September 2006.

[Yalcinoz01] Yalcinoz, T., Altun, H., y Uzam, M., "*Economic dispatch solution using a genetic algorithm based on arithmetic crossover*", IEEE Porto Power Tech Proceedings, vol. 2, pág. 4, Singapore, Singapore, Septiembre 2001.

[Zainud-Deen09] Zainud-Deen, S., El-Deen, E., Ibrahim, M., y Botros, A., "*Graphical processing units (gpu) acceleration of finite-difference frequency-domain (fdfd) technique*", National Radio Science Conference, NRSC, pág. 1, Charleston, SC, USA, Marzo 2009.

[Zhang10] Zhang, N., shan Chen, Y., y li Wang, J., "*Image parallel processing based on gpu*", 2nd International Conference on Advanced Computer Control, ICACC, vol. 3, págs. 367 – 370, Shenyang, China, Marzo 2010.

[Zhixiang07] Zhixiang, H., "*Parameters identification of continuous system based on hybrid genetic algorithm*", Chinese Control Conference, CCC, págs. 278–281, Hunan, China, Julio 2007.

[Zhu09] Zhu, Weihang, "*Massively parallel differential evolution–pattern search optimization with graphics hardware acceleration: an investigation on*

*bound constrained optimization problems*", Journal of Global Optimization, págs. 1–21, Department of Industrial Engineering, Lamar University, Beaumont, TX, USA, Agosto 2010.