



Instituto Tecnológico de Ciudad Victoria

División de Estudios de Posgrado e Investigación

TESIS

Una versión paralela del algoritmo JADE mediante el uso de GPU's

presentada por

Ing. Perla Marlen Bocanegra Castillo

como requisito para la obtención del grado de
Maestro en Sistemas Computacionales

Directora de tesis
Dra. Adriana Mexicano Santoyo

Codirector de tesis
Dr. Salvador Cervantes Alvarez



TECNOLÓGICO NACIONAL DE MÉXICO
Instituto Tecnológico de Cd. Victoria

AUTORIZACIÓN DE IMPRESIÓN DE TESIS

Ciudad Victoria, Tam., a 22 de junio de 2016

C. PERLA MARLÉN BOCANEGRA CASTILLO
CANDIDATO AL GRADO DE MAESTRO EN
SISTEMAS COMPUTACIONALES
PRESENTE.

Después de haber atendido las indicaciones sugeridas por la Comisión Revisora de la Academia de la Maestría en Sistemas Computacionales y en relación a su trabajo de tesis cuyo título es: **"Una versión paralela del algoritmo JADE mediante el uso de GPU's"**, me es grato comunicarle que se le concede la autorización para que proceda con la impresión de su tesis.

ATENTAMENTE

"EXCELENCIA EN EDUCACIÓN TECNOLÓGICA"

DR. PEDRO ALMAGUER SIERRA
JEFE DE LA DIVISIÓN DE ESTUDIOS
DE POSGRADO E INVESTIGACIÓN



INSTITUTO TECNOLÓGICO
DE CD. VICTORIA
DEPARTAMENTO DE
DIVISIÓN DE ESTUDIOS DE
POSGRADO E INVESTIGACIÓN

c.c.p. Subdirección Académica
Presidente del Consejo del Posgrado
Departamento de Servicios Escolares
Expediente



Dedicatoria

Dedico este trabajo a mis padres Alejandro Bocanegra y Ma. Venancia Castillo, a mis hermanas Alejandra, Sonia y Zulema quienes me han dado todo su amor y apoyo, por su tolerancia y motivación. A ustedes les debo todo, les doy todo mi cariño y respeto.

Agradecimientos

Mis sinceros agradecimientos a mis padres por todo el esfuerzo y apoyo brindado, por el amor y la motivación de seguir adelante.

Agradezco al Instituto Tecnológico de Cd. Victoria en especial a la División de Estudios de Posgrado e Investigación por darme la oportunidad para realizar mis estudios de posgrado.

También quiero agradecer a la Coordinación Nacional de Becas de Educación Superior (CNBES), por brindarme los recursos necesarios para permanecer en el programa de posgrado del ITCV.

Agradezco a mi directora de tesis, la Dra. Adriana Mexicano Santoyo, por su apoyo durante el desarrollo de esta tesis, mi agradecimiento también a mi co-director de tesis, el Dr. Salvador Cervantes Álvarez por sus observaciones y aportaciones que ayudaron a este trabajo y al Maestro Antonio Abrego por sus sugerencias y críticas.

Finalmente, quiero dar las gracias a mis compañeros de maestría, Karla, Willie y mi novio Rodrigo por todo el apoyo y afecto que me han dado y los buenos momentos compartidos. Muchas gracias.

RESUMEN

El uso de las Unidad de Procesamiento Gráfico (GPU's) están cobrando cada vez más importancia por su gran potencia de cálculo, escalabilidad y bajo costo, actualmente la corporación internacional NVIDIA es la precursora de esta tecnología y ha creado un nuevo paradigma de desarrollo sobre GPU llamado CUDA (Arquitectura Unificada de Dispositivos de Cómputo), el cual es un modelo de programación y una arquitectura de cálculo.

En el caso de los algoritmos evolutivos, que son inherentemente paralelos, esta tecnología puede ser ventajosa para el funcionamiento de los experimentos que exigen un alto costo computacional ya que los algoritmos evolutivos aplican los operadores evolutivos un gran número de veces, lo que lleva, en algunas ocasiones, a extender su tiempo de ejecución; por ello la necesidad de aplicar procesos altamente paralelos para reducir el tiempo de cálculo.

En este trabajo se presenta una implementación en paralelo del algoritmo evolutivo diferencial adaptativo con archivo externo opcional denominado JADE, utilizando la Arquitectura Unificada de Dispositivos de Cómputo (CUDA, *Compute Unified Device Architecture*), para reducir el tiempo de ejecución del algoritmo. El algoritmo fue probado con tres funciones de evaluación: *Sphere*, *Rosenbrock* y *Rastrigin*, el tiempo de cálculo se ha comparado con el mismo algoritmo implementado en su versión secuencial utilizando el lenguaje de programación C. Los resultados medidos en términos de "*Speed-up*" demuestran que el tiempo de cálculo se puede reducir de manera significativa con el uso de CUDA, este comportamiento se puede observar mejor al trabajar con grandes cantidades de datos. Sin embargo, no necesariamente la población con mayor cantidad de datos alcanza el mejor desempeño.

ABSTRACT

The use of Graphics Processor Unit (GPUs) is increasing due to its high computational throughput, scalability and low cost, currently NVIDIA Corporation is the pioneer of this breakthrough technology and has developed a new paradigm on GPU called CUDA (Compute Unified Device Architecture), which is a programming model and computation architecture.

In the case of evolutionary algorithms that are inherently parallel, the use of GPUs can be convenient in order to run experiments that require a high computational cost as evolutionary algorithms, which applied evolutionary operators many times, leading occasionally to extend its execution run time; therefore the need for highly parallel processes to reduce the computation time.

This work presents a parallel implementation of JADE: Adaptive Differential Evolution With Optional External Archive, using the Compute Unified Device Architecture (CUDA), in order to reduce the algorithm's runtime. The algorithm was tested with three benchmark functions: *Sphere*, *Rosenbrock* and *Rastrigin*, the execution run time was compared against its sequential version. The results were measured in terms of "Speed-up" and they show that the execution run time can be reduced significantly by the use of C-CUDA, this benefit can be observed better when working with large amounts of data. However, not necessarily the population with more data reaches the best performance.

TABLA DE CONTENIDO

1 INTRODUCCIÓN	1
1.1 DESCRIPCIÓN DEL PROBLEMA	2
1.2 OBJETIVO GENERAL	3
1.2.1 <i>Objetivos específicos.....</i>	<i>3</i>
1.3 JUSTIFICACIÓN	4
1.4 ORGANIZACIÓN DEL DOCUMENTO	5
2 MARCO TEÓRICO.....	6
2.1 ALGORITMOS EVOLUTIVOS	6
2.1.1 <i>Algoritmo de Evolución Diferencial.....</i>	<i>7</i>
2.1.2 <i>JADE: Evolución diferencial adaptativa con archivo externo opcional</i>	<i>9</i>
2.2 UNIDADES DE PROCESAMIENTO GRÁFICO	11
2.2.1 <i>Arquitecturas de paralelización</i>	<i>14</i>
2.3 PARALELIZACIÓN DE ALGORITMOS DE EVOLUCIÓN DIFERENCIAL UTILIZANDO GPU'S	17
2.3.1 <i>Análisis de los trabajos relacionados.....</i>	<i>22</i>
3 IMPLEMENTACIÓN EN PARALELO DEL ALGORITMO JADE.....	25
3.1 CARACTERÍSTICAS DE LOS ALGORITMOS ADAPTATIVOS DE EVOLUCIÓN DIFERENCIAL.....	25
3.1.1 <i>Evaluación de los algoritmos adaptativos de Evolución Diferencial</i>	<i>27</i>
3.1.2 <i>Análisis de las características de los dispositivos de hardware.....</i>	<i>31</i>
3.2 ANÁLISIS DE LAS FASES DEL ALGORITMO JADE.....	33
3.2.1 <i>Inicialización</i>	<i>34</i>
3.2.2 <i>Evaluación de la función.....</i>	<i>35</i>
3.2.3 <i>Mutación.....</i>	<i>37</i>
3.2.4 <i>Cruce.....</i>	<i>39</i>
3.2.5 <i>Selección.....</i>	<i>40</i>
3.3 FUNCIONES DE EVALUACIÓN	41
3.3.1 <i>Función de evaluación “Sphere”.....</i>	<i>42</i>

3.3.2 Función de evaluación “Rosenbrock”	43
3.3.2 Función de evaluación “Rastrigin”	44
3.4 IMPLEMENTACIÓN PARALELA DE JADE UTILIZANDO CUDA	45
4 EVALUACIÓN DE LA VERSIÓN PARALELA DEL ALGORITMO JADE	51
4.1 AMBIENTE DE LAS PRUEBAS	51
4.2 EVALUACIÓN DEL DESEMPEÑO DEL ALGORITMO JADE EN SU VERSIÓN PARALELA.....	52
4.2.1 Evaluación de JADE secuencial contra paralelo utilizando la función Sphere.....	53
4.2.2 Evaluación de JADE secuencial contra paralelo utilizando la función Rosenbrock	55
4.2.3 Evaluación de JADE secuencial contra paralelo utilizando la función Rastrigin	57
4.3 RENDIMIENTO JADE SECUENCIAL CONTRA JADE PARALELO.....	58
5 CONCLUSIONES Y TRABAJOS FUTUROS	61
5.1 CONCLUSIONES	61
5.2 TRABAJOS FUTUROS.....	62
5.3 LISTA DE PUBLICACIONES	63
REFERENCIAS.....	64

LISTA DE FIGURAS

Figura 1 Esquema del planteamiento del problema.	3
Figura 2 Estructura de CUDA.	16
Figura 3 Tiempo de cálculo de las funciones de prueba utilizando C-CUDA y C. .	18
Figura 4 Diagrama de flujo de la aplicación en CUDA.	20
Figura 5 Tiempo promedio obtenido por los algoritmos sDE,cudaDEi,cudaDE y FPDE utilizando tamaños de población igual a 50,100,500,1000 para 25 iteraciones	21
Figura 6 Comparación entre el tiempo de ejecución de CUDA y la CPU.	22
Figura 7 Esquema general de las fases del algoritmo. Los recuadros rojos indican las fases que fueron paralelizadas.	34
Figura 8 Inicialización de la población	35
Figura 9 Evaluación de la función	37
Figura 10 Mutación.....	38
Figura 11 Cruce.....	40
Figura 12 Selección.....	41
Figura 13 Función “ <i>Sphere</i> ” evaluada en un cubo de [-5.12, 5.12].	43
Figura 14 Función “ <i>Rosenbrock</i> ” evaluada en un cubo de [-2.048, 2.048].....	44
Figura 15 Función “ <i>Rastrigin</i> ” evaluada en un cubo de [-5.12, 5.12].....	45
Figura 16 Fase de Inicialización.	46
Figura 17 Fase de Evaluación de la Función.	47
Figura 18 Fase de Mutación.....	48
Figura 19 Fase de Cruce.....	49
Figura 20 Fase de Selección.....	50
Figura 21 Comparación Tiempo CPU vs Tiempo GPU usando la función <i>Sphere</i> .55	
Figura 22 Comparación Tiempo CPU vs Tiempo GPU usando la función <i>Rosenbrock</i>	56
Figura 23 Comparación Tiempo CPU vs Tiempo GPU usando la función <i>Rastrigin</i>	58
Figura 24 Comportamiento del “Speed-up” a través del aumento de población. ..	60

LISTA DE TABLAS

Tabla 1 Comparativa de trabajos relacionados.	24
Tabla 2 Principales características de los algoritmos basados en aleatorización- adaptación.....	26
Tabla 3 Resultados experimentales con una población de 100 y una dimensión del problema de 30, los resultados son el promedio de 50 ejecuciones.	28
Tabla 4 Resultados experimentales con una población de 400 y una dimensión del problema de 100, los resultados son el promedio de 50 ejecuciones.	29
Tabla 5 Configuraciones utilizadas en la literatura especializada para diferentes versiones del algoritmo DE en CUDA.	32
Tabla 6 Comparación Tiempo CPU vs Tiempo GPU usando la función Sphere. ...	54
Tabla 7 Comparación Tiempo CPU vs Tiempo GPU usando la función Rosenbrock.	55
Tabla 8 Comparación Tiempo CPU vs Tiempo GPU usando la función Rastrigin.	57

Capítulo 1

INTRODUCCIÓN

Los Algoritmos Evolutivos (AE) son procedimientos de búsqueda y optimización inspirados en el mundo biológico, se caracterizan por imitar procesos adaptativos de los sistemas naturales y se basan en la supervivencia del mejor individuo. Trabajan sobre poblaciones de soluciones que evolucionan de generación en generación mediante operadores genéticos adaptados al problema. Uno de ellos es el algoritmo de evolución diferencial propuesto por Storn y Price en 1995, el cual 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 [1]. Como ejemplo de la aplicación de éstos métodos para resolver problemas reales podemos mencionar la optimización de los ciclos de duración de los semáforos [2] y el control del flujo de agua que pasa por una turbina para generar energía [3] entre otros.

Los AE aplican los operadores evolutivos un gran número de veces lo que lleva en algunas ocasiones a extender su tiempo de ejecución por ello, la necesidad de aplicar procesos altamente paralelos para reducir el tiempo de cálculo. En este sentido el uso de las GPU's están cobrando cada vez más importancia por su gran

potencia de cálculo, escalabilidad y bajo costo, actualmente Nvidia es la precursora de esta tecnología y ha creado un nuevo paradigma de desarrollo sobre GPU llamado CUDA, el cual es un modelo de programación y una arquitectura de cálculo [4].

En la literatura especializada existen algunas versiones del algoritmo evolutivo paralelizadas, cuya finalidad consiste en reducir el tiempo de ejecución en comparación con las versiones secuenciales, sin embargo, no se encontró evidencia de que el algoritmo evolutivo diferencial adaptativo con archivo externo opcional (JADE) haya sido previamente paralelizada usando CUDA.

En este trabajo de tesis, se propone desarrollar la versión en paralelo del algoritmo JADE propuesto por Zhang y Sanderson en 2009 [5] con el objetivo de reducir el tiempo de ejecución del algoritmo.

1.1 DESCRIPCIÓN DEL PROBLEMA

Una limitante que se encuentra presente al utilizar versiones secuenciales de los algoritmos de evolución diferencial cuando se requiere resolver problemas reales, es el tiempo de ejecución que tardan dichos algoritmos al procesar grandes cantidades de información. Para atenuar el inconveniente causado por el procesamiento de altas cantidades de información, se ha recurrido a la aplicación de técnicas de paralelización sobre dichos algoritmos. De acuerdo a la literatura especializada, el algoritmo JADE es un algoritmo de evolución diferencial ampliamente utilizado en la solución de problemas reales y a la fecha no hemos encontrado ninguna versión paralelizada utilizando la tecnología de los dispositivos GPU's. En esta tesis se propone implementar la versión paralelizada del algoritmo JADE para reducir su tiempo de ejecución utilizando la arquitectura CUDA.

La Figura 1 muestra un ejemplo de cómo trabaja la forma secuencial y la forma paralelizada, con los mismos parámetros de entrada un tamaño de población (NP)

de 100 individuos con una dimensión del problema (D) de 30, se puede apreciar que la forma paralelizada tiende a ser más rápida que la secuencial ya que en ella las tareas independientes se distribuyen y ejecutan de forma simultánea, con lo cual se obtiene una reducción en el tiempo de ejecución a diferencia de la forma secuencial que ejecuta una tarea tras otra haciendo más lento el proceso.

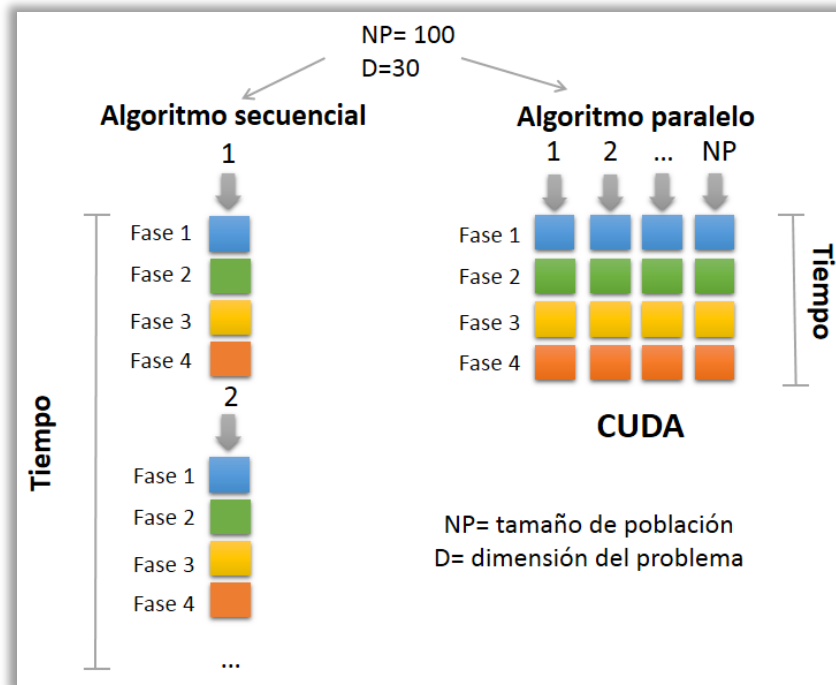


Figura 1 Esquema del planteamiento del problema.

1.2 OBJETIVO GENERAL

El objetivo general de esta tesis es desarrollar una versión paralela del algoritmo de evolución diferencial JADE utilizando la arquitectura CUDA.

1.2.1 Objetivos específicos

Los objetivos específicos para el desarrollo del trabajo de tesis fueron los siguientes:

- Analizar el algoritmo JADE.
- Implementar el algoritmo JADE en su forma secuencial.
- Identificar las fases que serán paralelizadas.
- Implementar el algoritmo JADE con la arquitectura de paralelización CUDA.
- Comparar la versión secuencial con la implementada en la arquitectura CUDA.
- Analizar y documentar resultados.

1.3 JUSTIFICACIÓN

Los Algoritmos Evolutivos 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, finanzas, bioinformática, reconocimiento de patrones entre otros , un AE es utilizado dada la enorme cantidad de posibles configuraciones y la necesidad de una búsqueda eficaz.

Una limitante que se encuentra inmersa al utilizar versiones secuenciales de algoritmos de evolución diferencial cuando se requieren resolver problemas reales de optimización, es su tiempo de ejecución, para lo cual en trabajos como [6], [7], [8], [9] se han implementado versiones paralelas de los algoritmos que permiten menguar dicho inconveniente. En este sentido, el uso de las Unidades de Procesamiento Gráfico (GPU's) utilizadas como medio de paralelización están cobrando cada vez más importancia por su gran potencialidad de cálculo, escalabilidad y bajo costo. Existen diferentes arquitecturas que muestran las ventajas de la paralelización de evolución diferencial (MPI, CUDA, etc.), de entre las cuales destaca la arquitectura CUDA debido a su relación costo/beneficio. No obstante no se ha reportado ninguna implementación del algoritmo JADE en arquitectura CUDA.

1.4 ORGANIZACIÓN DEL DOCUMENTO

Lo que resta de este trabajo se encuentra organizado de la siguiente manera:

El Capítulo 2 aborda el marco teórico de esta investigación donde se presentan algunos conceptos relacionados con los AEs y el análisis de estos, después, se muestra la revisión de los trabajos relacionados y por último se expone una tabla comparativa donde se resumen las características del algoritmo JADE.

En el Capítulo 3 se encuentra la metodología e implementación del algoritmo JADE en su forma paralela. En este Capítulo se analizan las características de los Algoritmos Evolutivos y posteriormente se realiza la descripción de las fases de un Algoritmo Evolutivo, por último se muestra la implementación del algoritmo usando la programación paralela.

El Capítulo 4 muestra las pruebas realizadas de la implementación del algoritmo JADE en su forma secuencial y paralela. Se experimentó con tres funciones de prueba reconocidas en la literatura especializada: *Sphere*, *Rosenbrock* y *Rastrigin*, los datos obtenidos fueron utilizados para calcular el “*Speed-up*” de JADE paralelizado.

El Capítulo 5 presenta las conclusiones en base a los resultados obtenidos de las pruebas realizadas con las versiones del algoritmo y se muestran posibles trabajos futuros que pueden dar continuidad a esta investigación.

Capítulo 2

MARCO TEÓRICO

2.1 ALGORITMOS EVOLUTIVOS

Los Algoritmos Evolutivos (AEs) son métodos de optimización y búsqueda de soluciones basados en los postulados de la evolución biológica, tales como la reproducción, mutación, recombinación y selección. Son utilizados principalmente en problemas con espacios de búsqueda extensos y no lineales, en donde otros métodos no son capaces de encontrar soluciones en un tiempo razonable [10].

Los Algoritmos Evolutivos tuvieron su origen en el los años 60's, fueron introducidos por John Holland [11], quien incorporó los métodos de selección natural y supervivencia a la resolución de problemas de Inteligencia Artificial. Los algoritmos evolutivos surgieron por la simulación de procesos de la evolución natural dando resultado a lo que hoy se conoce como optimización estocástica (Algoritmos Evolutivos). Debido a la larga historia del cómputo evolutivo, existen diferentes variaciones de los AEs; pero, los mecanismos dentro de estas técnicas son los mismos. En base a una población de individuos las condiciones del ambiente provocan una selección y por lo tanto sobrevive la población más apta [12].

2.1.1 Algoritmo de Evolución Diferencial

El algoritmo de Evolución Diferencial (ED) fue desarrollado por Rainer Storn y Kenneth Price [13] para optimización en espacios continuos.

El algoritmo de Evolución Diferencial se utiliza como método de optimización perteneciente a la categoría de computación evolutiva, aplicado en la resolución de problemas complejos. Al igual que otros algoritmos de esta categoría, la ED mantiene una población de soluciones candidatas, las cuales se recombinan y mutan para producir nuevos individuos que podrán ser elegidos de acuerdo al valor de su función de desempeño [14]. Lo que caracteriza a la ED es el uso de vectores de prueba, que compiten con los individuos de la población actual a fin de sobrevivir.

Este algoritmo se basa en la evolución de una población de individuos (vectores de valores reales que representan las soluciones en el espacio de búsqueda), donde la generación de nuevos individuos se lleva a cabo mediante los operadores de mutación y cruce. Después de la inicialización, el algoritmo DE repite las operaciones mutación, cruza, y selección para producir un vector de prueba y seleccionar uno de esos vectores con el mejor valor en cada vector de prueba hasta satisfacer algún criterio de terminación [15].

A continuación se describe la función que realiza cada una de las operaciones del algoritmo DE iniciando con la creación de la población hasta la selección de la población que cumpla con el criterio de terminación:

Convencionalmente, generaciones subsecuentes en el algoritmo DE son denotadas por $G = 0, 1, \dots, G_{max}$. Se denota al i -ésimo vector de la población en la generación actual como se muestra en la ecuación 1:

$$\vec{X}_i^G = (x_{i,1}^G, x_{i,2}^G, \dots, x_{i,j}^G), \quad j = 1, 2, \dots, D. \quad (1)$$

Inicialización. En primer lugar, NP individuos son generados aleatoriamente dentro de un espacio de búsqueda de D -dimensiones. La población inicial debe cubrir completamente el espacio de búsqueda restringido por los límites mínimo y máximo: $\vec{X}_{min} = (x_{min,1}, x_{min,2}, \dots, x_{min,D})$, $\vec{X}_{max} = (x_{max,1}, x_{max,2}, \dots, x_{max,D})$. Por consiguiente, la inicialización del j -ésimo elemento en el i -ésimo vector se realiza como se indica en la ecuación 2:

$$x_{i,j}^0 = x_{min,j} + rand(0,1) \cdot (x_{max,j} - x_{min,j}), \quad j = 1, 2, \dots, D, \quad (2)$$

donde $rand(0,1)$ representa una variable aleatoria distribuida uniformemente dentro de los rangos $[0,1]$, y es evento independientemente para cada componente del i -ésimo vector.

Operación Mutación. En la literatura de los algoritmos DE, el vector de mutación V_i , o vector donante, es obtenido a través de la operación llamada mutación diferencial con respecto a cada individuo $\vec{X}_i$, conocido como vector objetivo, en la población actual. El vector donante es creado con una estrategia de mutación. Varias estrategias de mutación han sido propuestas, en la ecuación 3 se muestra una de las más populares y simples formas de mutaciones:

$$\vec{V}_i^G = \vec{X}_{r_1}^G + F \cdot (\vec{X}_{r_2}^G - \vec{X}_{r_3}^G). \quad (3)$$

Los índices r_1 , r_2 , y r_3 son números enteros mutuamente excluyentes generados aleatoriamente dentro del rango $[1, NP]$, los cuales son diferentes al índice del vector base i . Estos índices son generados aleatoriamente en cada vector de mutación. Ahora, la diferencia de los dos vectores es escalada por un factor de mutación F el cual típicamente se encuentra en el intervalo $[0.4, 1]$ acorde a la literatura de los algoritmos DE, la diferencia escalada es añadida al tercer vector, obteniendo así el vector donante v .

Operación Cruza. Después de la operación mutación, acorde al vector objetivo $\vec{X}_i^G$ y su correspondiente vector donante $\vec{V}_i^G$, un vector de prueba $\vec{U}_i^G$ es producido por la operación de cruza. En la versión tradicional, el algoritmo DE aplica la cruza binaria como se muestra en la ecuación (4):

$$u_{i,j}^G = \begin{cases} v_{i,j}^G, & \text{si } rand_{i,j} [0,1] \leq C_r \text{ or } j = j_{rand}, \\ x_{i,j}^G, & \text{de otra forma,} \end{cases} \quad (4)$$

donde C_r es una tasa de cruce dentro del rango $[0,1]$, definido por el usuario como una constante, la cual controla la probabilidad de los valores empleados del vector donante. j_{rand} es un entero elegido aleatoriamente en el rango $[1, NP]$ el cual es introducido para asegurar que el vector de prueba contiene al menos un parámetro del vector donante.

Operación Selección. La aptitud de cada vector de prueba es evaluada y comparada con su correspondiente vector objetivo en la población actual. Para el problema de minimización, si el valor de aptitud del vector de prueba es menor que el vector objetivo, el vector objetivo será remplazado por el vector de prueba en la población de la siguiente generación. De lo contrario, el vector objetivo se mantendrá en la población de la siguiente generación. La operación selección es expresada en la ecuación (5) de la siguiente manera:

$$\vec{X}_i^{G+1} = \begin{cases} \vec{U}_i^G, & \text{si } f(\vec{U}_i^G) \leq f(\vec{X}_i^G), \\ \vec{X}_i^G, & \text{de otra forma} \end{cases} \quad (5)$$

2.1.2 JADE: Evolución diferencial adaptativa con archivo externo opcional

JADE es un algoritmo evolutivo diferencial adaptativo con archivo externo opcional propuesto por Zhang y Sanderson [5] [16], el cual implementa una estrategia de mutación basada en el vecindario y un archivo opcional externo empleado para mejorar el desempeño del DE.

Los pasos del algoritmo son los siguientes:

Paso 1. Inicialización

JADE sigue el procedimiento básico de la evolución diferencial. La población inicial $\{x_{i,0} = (x_{1,i,0}, x_{2,i,0}, \dots, x_{D,i,0}) | i = 1, 2, \dots, NP\}$ se genera aleatoriamente según una distribución uniforme $x_j^{low} \leq x_{j,i,0} \leq x_j^{up} | j = 1, 2, \dots, D$, donde D es la dimensión del problema, NP es el tamaño de la población, x_j^{low} y x_j^{up} definen los

límites superior e inferior de la variable de decisión j -ésimo. Después de la inicialización, JADE entra en un bucle de operaciones evolutivas (mutación, cruce y selección) y las operaciones de adaptación de parámetros.

Paso 2. Mutación

En la ecuación (6) se muestra el vector de mutación generado por DE/current-to- p best, con archivo:

$$V_i^G = X_i^G + F_i \cdot (X_{best,p}^G - X_i^G) + F_i \cdot (X_{r1}^G - \tilde{X}_{r2}^G), \quad (6)$$

donde $X_{best,p}^G$ es seleccionado aleatoriamente como uno de los mejores 100p individuos de la población actual con $p \in (0,1]$. Mientras, X_i^G y X_{r1}^G son individuos aleatorios en la población actual P , respectivamente. $\tilde{X}_{r2}^G$ es seleccionado aleatoriamente de la unión de P y el archivo A . En lo particular, A es un conjunto de soluciones de las generaciones anteriores y su número de individuos no es mayor al tamaño de la población. En cada generación, el factor de mutación F_i y la probabilidad de cruce C_{r_i} de cada individuo X_i son, respectivamente, actualizados dinámicamente acorde a una distribución Cauchy con media μ_F expresada en la ecuación (7) y una distribución normal con media μ_C , descrita en la ecuación (8):

$$F_i = rand\ c_i(\mu_F, 0.1), \quad (7)$$

$$C_{r_i} = rand\ n_i(\mu_C, 0.1). \quad (8)$$

Los dos parámetros propuestos se inicializan como 0.5 y al final de cada generación son generados con las ecuaciones (9) y (10):

$$\mu_F = (1 - c) \cdot \mu_F + c \cdot mean_L(S_F), \quad (9)$$

$$\mu_C = (1 - c) \cdot \mu_C + c \cdot mean_A(S_{C_r}), \quad (10)$$

donde c está en un rango $(0,1)$; S_F y S_{C_r} indican el conjunto de todos los factores de mutación y probabilidades de cruce exitosos en las generaciones; $mean_A(\cdot)$ denota la usual media aritmética, y $mean_L(\cdot)$ es la media de Lehmer, la cual se define en la ecuación (11):

$$mean_L(S_F) = \frac{\sum_{i=1}^{|S_F|} F_i^2}{\sum_{i=1}^{|S_F|} F_i}. \quad (11)$$

Paso 3. Cruza

Después de la mutación, la operación binomio de cruza $u_{i,g} = (u_{1,i,g}, u_{2,i,g}, \dots, u_{D,i,g})$ forma el vector de prueba como se muestra en la ecuación (12):

$$u_{j,i,g} = \begin{cases} v_{j,i,g} & \text{si } rand_j(0,1) \leq CR_i \text{ o } j = j_{rand} \\ x_{j,i,g} & \text{en otro caso} \end{cases} \quad (12)$$

donde $rand_j(0,1)$ es un numero aleatorio en el intervalo (0,1) y es generado para cada j , $j_{rand} = randint_i(1, D)$ es un numero entero elegido al azar entre 1 y D y es generado para cada i , y la probabilidad de cruce $CR_i \in [0,1]$, corresponde aproximadamente a la fracción promedio de los componentes del vector que se heredan a partir del vector de mutación.

Paso 4. Selección

La operación de selección elige el mejor entre el vector padre $x_{i,g}$ y el vector de prueba $u_{i,g}$ acorde a los valores de la función de evaluación. Por ejemplo, si tenemos un problema de minimización, el vector seleccionado se define como muestra la ecuación (13):

$$x_{i,g+1} = \begin{cases} u_{i,g}, & \text{si } f(u_{i,g}) < f(x_{i,g}) \\ x_{i,g}, & \text{en otro caso} \end{cases} \quad (13)$$

La operación es llamada actualización correcta si el vector de prueba $u_{i,g}$ es mejor que el de los padres $x_{i,g}$, es decir, la mejora o progreso de la evolución $\Delta_{i,g} = f(x_{i,g}) - f(u_{i,g})$ es positivo. En consecuencia, los parámetros de control F_i y CR_i usados para generar $u_{i,g}$ son llamados factor de mutación exitoso y probabilidad de cruce exitoso, respectivamente.

2.2 UNIDADES DE PROCESAMIENTO GRÁFICO

Impulsados por la 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

altamente paralelas con una gran cantidad de procesadores y un gran poder de cómputo [17].

Los GPUs son microprocesadores especializados que permiten la aceleración de la representación de los gráficos desde la Unidad Central de Proceso (CPU) para aligerar su carga de trabajo, es decir mientras el GPU se encarga de la representación de los gráficos, la CPU se puede encargar de otros cálculos. Los GPUs actuales son más eficientes en la manipulación de los gráficos en una computadora, cuentan con una arquitectura altamente paralela que ha mostrado ser más eficiente que las CPUs para la implementación y ejecución de ciertos algoritmos [18]. Un GPU puede estar presente en una PC en forma de una tarjeta de video o en la actualidad puede incluso ser la tarjeta base.

En el 2006 NVIDIA introdujo una nueva arquitectura de cómputo paralelo de propósito general denominada CUDA (del inglés *Compute Unified DEvice Architecture*) capaz de aprovechar al máximo las capacidades de los dispositivos GPUs con el objetivo de resolver problemas computacionales complejos de una manera más eficiente. CUDA permite a los desarrolladores utilizar el lenguaje C como lenguaje de implementación de alto nivel.

Algunos de los términos directamente relacionados con las GPUs se describen a continuación:

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

CUDA: es una arquitectura de cómputo paralelo de propósito general la cual permite aprovechar el poder de los GPUs, y se basa en la utilización de un elevado

número de nodos de procesamiento para realizar operaciones sobre un gran volumen de datos en paralelo, consiguiendo reducir el tiempo de procesado y obteniendo altas prestaciones.

Thread: son los hilos de ejecución que se encargan de realizar una operación de manera paralela sobre determinado conjunto de datos. Los *thread* tienen un identificador único al cual se puede tener acceso a través de la variable integrada *threadIdx*.

Bloque de *threads*: son conjuntos de *threads* organizados en arreglos unidimensionales, bidimensionales o tridimensionales. El tamaño de los bloques 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 dentro de un arreglo en una o dos dimensiones denominado *grid*. El número y organización de los bloques en un *grid* son determinados por el tamaño de los datos a procesar y también por la cantidad de procesadores disponibles en el sistema.

Host: en CUDA se denomina *host* a la parte del Sistema de cómputo que involucre la unidad central de proceso CPU y los recursos de cómputo de que dispone directamente.

Device: conjunto de recursos contenidos dentro de la GPU, además especifica una función que será ejecuta en la GPU y podrá ser llamada únicamente por funciones que hayan sido declaradas para ejecutarse en la GPU.

Kernel: se refiere a una función que se va a 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*.

2.2.1 Arquitecturas de paralelización

La programación paralela es el uso de varios procesadores trabajando en conjunto para dar solución a una tarea en común, dividen el trabajo y cada procesador hace una porción del problema al poder intercambiar datos por una red de interconexión o a través de memoria [19]. La programación paralela permite resolver problemas que no caben en un solo procesador y que no se resuelven en un tiempo razonable, se pueden ejecutar problemas con mayor complejidad más rápidamente. Una alternativa que se ha vuelto muy popular es el uso de las arquitecturas de paralelización:

PVM

Es el acrónimo de *Parallel Virtual Machine*. PVM es un sistema que permite a una red heterogénea de ordenadores comportarse como un único ordenador paralelo. La paralelización se realiza mediante paso de mensaje. Debe existir un proceso demonio que se encargue del control de los diversos procesos en las distintas maquinas, realizando de manera transparente al usuario toda la comunicación efectiva entre los distintos procesos, así como toda la inicialización, sincronización, terminación, etc. de los procesos. La gran ventaja que plantea PVM es su heterogeneidad, sin embargo, esto se garantiza, en general, a costa de menores rendimientos [20].

MPI

Es el acrónimo de *Message-Passing Interface*. MPI se definió como un estándar portable que trabaja bajo el supuesto de memoria distribuida y comunicación explícita por paso de mensajes. MPI define la sintaxis y la semántica de un conjunto de rutinas de librería para la creación de aplicaciones paralelas en C y Fortran. Existen muchas implementaciones de MPI, tanto públicas como comerciales. No

requiere la existencia de ningún proceso de dominio que controle la ejecución en diversas maquinas. La funcionalidad que ofrece MPI es similar a la de PVM, aunque con un mayor número de funciones. Se puede ejecutar en sistemas heterogéneos siempre y cuando el propio programador se encargue de manejar las diferencias entre los distintos sistemas [21].

CUDA

En el 2007, NVIDIA introdujo los GPU (Unidades de Procesamiento Gráfico) a la comunidad científica por medio de una interfaz de programación, la cual denomino CUDA. Abriendo así la posibilidad de programar los GPUs sin tener que aprender lenguajes, CUDA permite utilizarlos programando en C. El código está orientado en funciones que se ejecutan en el CPU o en el procesador del dispositivo (GPU) [22]. Entre los beneficios del uso de C-CUDA destaca el hecho de que al ejecutar aplicaciones sobre procesadores gráficos de NVIDIA tienen un rendimiento superior por dólar (costo) y por *watt* que implementaciones sobre tecnologías centrales de procesamiento [23].

CUDA permite traducir casi directamente código C al dispositivo. Este modelo unifica el CPU (*host*) y el GPU (*device*) en un sistema de cómputo heterogéneo, para sacar el mejor provecho de ambos dispositivos, contiene tres tipos de funciones la *Host Function* las cuales son llamadas y ejecutadas en el CPU, la *Kernel Function* las cuales son llamadas en el *host* y ejecutadas en el *device*; y la tercera es la *Device Function* las cuales son llamadas y ejecutadas en el GPU. CUDA presenta en su estructura los conceptos de “hilo”, “bloque” y “*grid*” como se muestra en la Figura 2 los hilos están organizados en los bloques y pueden ser hasta de 3 dimensiones, y los bloques están organizados en los *grid* los cuales también pueden ser de 3 dimensiones. Cada bloque de hilos es ejecutado en un *Streaming Multiprocesor* (SM). Cada SM permite ejecutar un número de bloques, una vez que se alcanza la capacidad máxima de bloque por SM, los bloques restantes tienen que esperar para ser ejecutados en el SM.

Las dimensiones del *grid* y los bloques dependen de las características del problema y algoritmo.

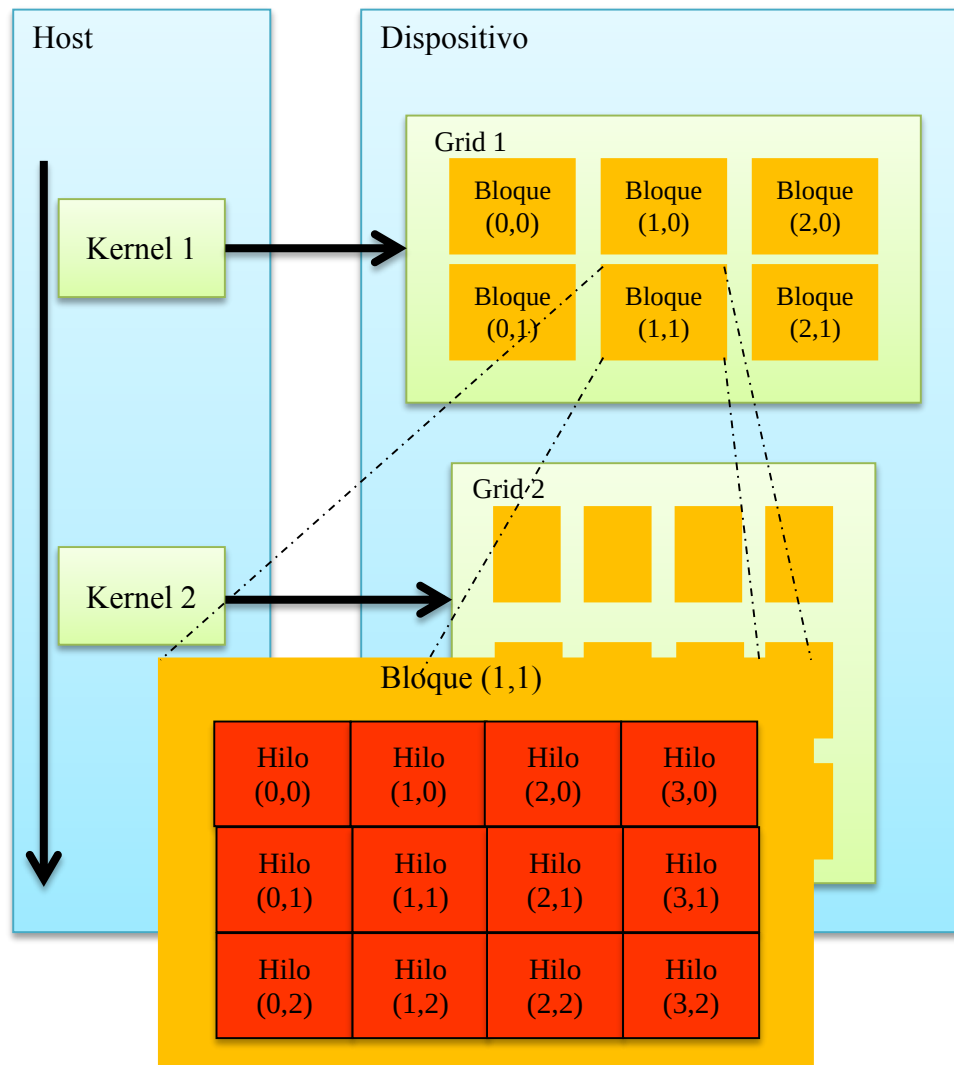


Figura 2 Estructura de CUDA.

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 GPU. 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 cuándo 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 (*Peripheral Component Interconnect Express*). 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.
 - Dado que las transferencias de los 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 *kernels* 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*.

2.3 PARALELIZACIÓN DE ALGORITMOS DE EVOLUCIÓN DIFERENCIAL UTILIZANDO GPU'S

En diferentes trabajos de investigación se ha mostrado el uso de los dispositivos GPU para la paralelización del algoritmo de Evolución Diferencial utilizando la arquitectura CUDA.

A continuación se describen algunos de los trabajos más destacados:

En 2010 Veronése y Krohling [6] implementaron el algoritmo de Evolución Diferencial en C-CUDA, mediante el uso de la tecnología de GPU's y CUDA. El algoritmo fue probado en un conjunto de funciones de referencia [24] y el tiempo de cálculo se ha comparado con el mismo algoritmo implementado en C. Los resultados demuestran que el tiempo de cálculo se puede reducir hasta 34 veces usando C-CUDA. Cada experimento se llevó a cabo alcanzando el número máximo de iteraciones fijadas 100.000. En la Figura 3 se representan los tiempos que arrojó uno de los experimentos con un número máximo de 100.000 iteraciones, los bloques naranja corresponden a la ejecución del algoritmo secuencial, mientras que los azules corresponden a la versión paralela, el mejor rendimiento computacional se logró con un aumento de velocidad de alrededor de 34 veces sobre la implementación secuencial en C. Los autores concluyeron que el potencial del paralelismo proporcionado por la GPU radica en la división de la población y los individuos. Cuanto mayor sea el tamaño de la población y de las dimensiones, más alto es el rendimiento computacional en términos de aumento de velocidad.

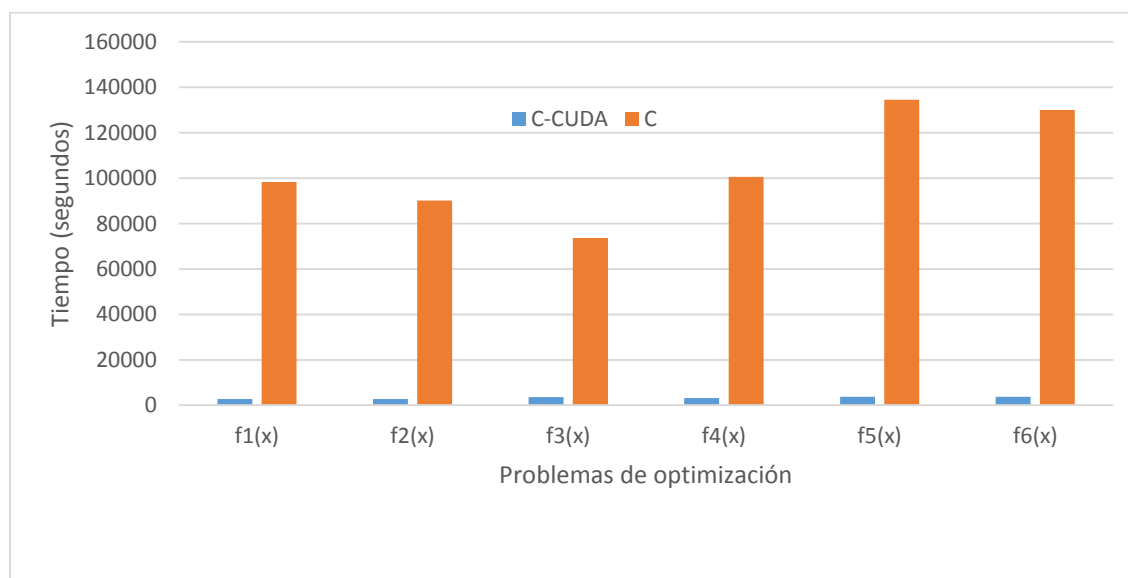


Figura 3 Tiempo de cálculo de las funciones de prueba utilizando C-CUDA y C.

En 2011 Krömer y Platos [7] propusieron una implementación del ED en CUDA para maximizar la utilización de los recursos proporcionados por las GPUs utilizando

los hilos disponibles para procesar cada solución candidata de individuos y utilizando el GPU para generar números aleatorios necesarios para la optimización. La meta de la implementación es alcanzar un alto paralelismo mientras se mantiene la simplicidad del algoritmo. La implementación consiste en un conjunto de funciones *kernel* C-CUDA la primera genera la población inicial, la segunda hace la implementación de la evaluación de la función, la tercera hace una selección aleatoria formando los vectores de prueba, la cuarta crea los vectores de mutación y cruza utilizando los vectores de prueba, la quinta evalúa la nueva población con la función y por ultimo ejecuta la selección de las poblaciones padre e hijo como se muestra en la Figura 4. Al efectuar una serie de experimentos, los autores concluyeron que su implementación es aproximadamente 3 veces más rápida que la versión en paralelo del algoritmo DE propuesta por Veronése y Krohling en 2010. Esta propuesta es capaz de encontrar un buen mínimo en funciones continuas y puede ser competitiva con las implementaciones anteriores de C-CUDA.

En 2012 Qin, Raimondo y Forbes [8] propusieron mejorar el algoritmo de Evolución Diferencial basado en CUDA para optimizar la memoria y la utilización de los dispositivos GPU's; varios *kernels* relacionados lógicamente se combinaron en uno compuesto, para reducir el acceso de memoria global, los parámetros de configuración de ejecución del *kernel* se determinaron automáticamente para maximizar la ocupación del dispositivo GPU; en el trabajo se emplearon secuencias para permitir la ejecución concurrente de *kernels* para maximizar la utilización de los GPU's, es decir, se ejecutaron dos *kernel* simultáneamente.

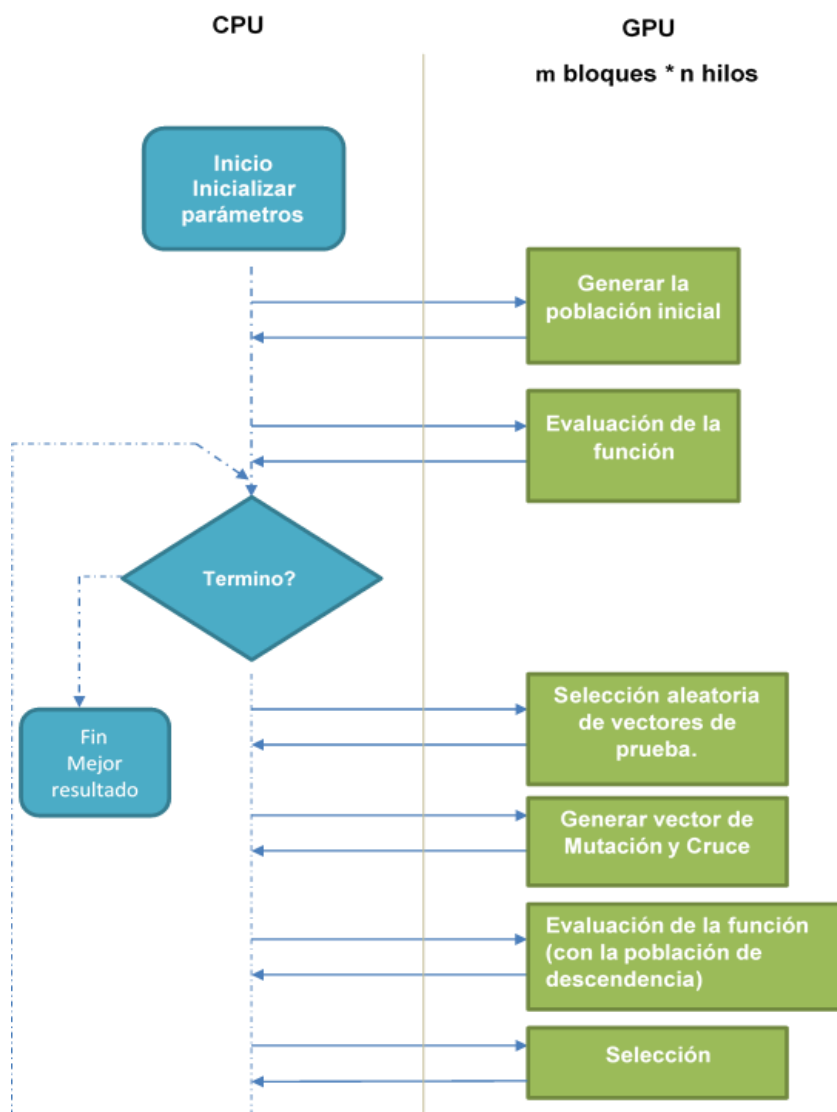


Figura 4 Diagrama de flujo de la aplicación en CUDA.

Los experimentos se realizaron al comparar las versiones paralelizadas del algoritmo de evolución diferencial; la propuesta por Veronése y Krohling [6], la cual fue llamada CudaDE, la propuesta por Domínguez y García [25] FPDE y la implementación secuencial; para todos los problemas de prueba. La mejora denominada cudaDEi muestra consistentemente la eficiencia del tiempo de computo, por ejemplo en la Figura 5 para la función *sphere* [26] con una dimensión de 10 y una población de 50 el tiempo requerido para su ejecución fue de 1.159

segundos, con una población de 100 el tiempo fue 0.633 segundos, cuando se utilizó una población de 500 el tiempo requerido fue de 0.228 segundos; finalmente , con una población de 1,000 se requirió de un tiempo de 0.912 segundos, En la Figura 5 las barras de color azul muestran el tiempo de ejecución del algoritmo CudaDE, el naranja representa el algoritmo FPDE, mientras que el gris son los tiempos del sDE y por último el amarillo corresponde a la mejora cudaDEi:

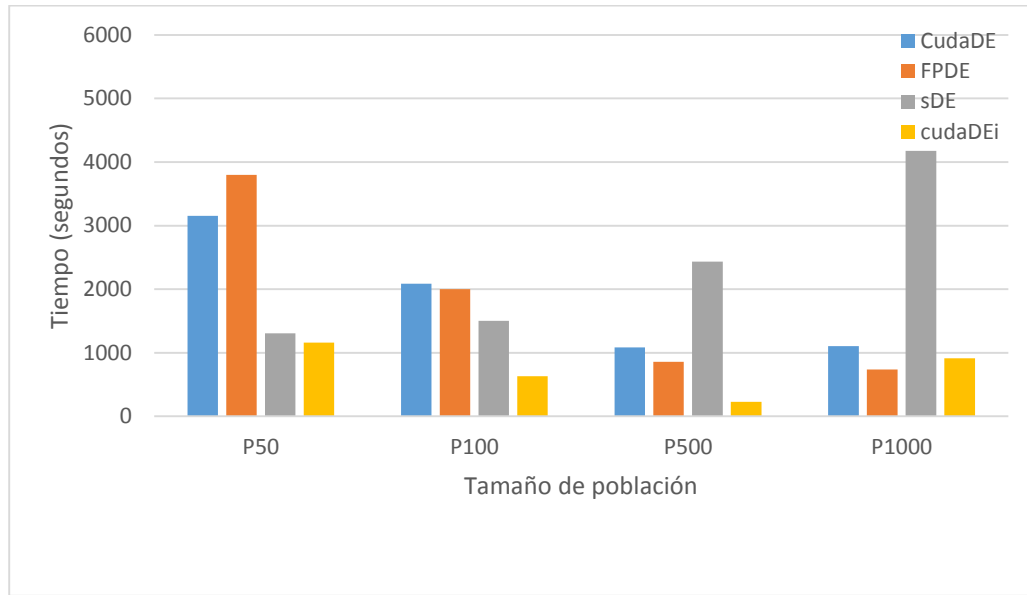


Figura 5 Tiempo promedio obtenido por los algoritmos sDE,cudaDEi,CudaDE y FPDE utilizando tamaños de población igual a 50,100,500,1000 para 25 iteraciones

En 2012 Davedra, Gaura y Bialic-Davendra [9] propusieron validar la aplicación del uso de una GPU de programación de propósito general, en cuanto a la medición del desempeño de cualquier algoritmo, tomando como objetivo la reducción del tiempo de ejecución. Este diseño aplicó el código “*Enhanced Differential Evolution (EDE)*” propuesto en [27] utilizando la búsqueda local en cada evaluación de la función. En base a los resultados se observa fácilmente que el EDE basado en CUDA, representado por el color naranja, es un algoritmo de ejecución rápido en comparación con la versión secuencial representada en color azul, como se muestra en la Figura 6. En general para cada conjunto de problemas, hay una mejora significativa para el tiempo de ejecución de la GPU. La mejor media total de la GPU

es 2,345.34 sobre la CPU. El tiempo medio de ejecución es 1,247.56 seg. en la GPU en comparación con 9,109.69 seg. en la CPU. Por lo tanto, los autores concluyen que la ejecución del algoritmo está en una magnitud de ocho veces más rápida en la GPU.

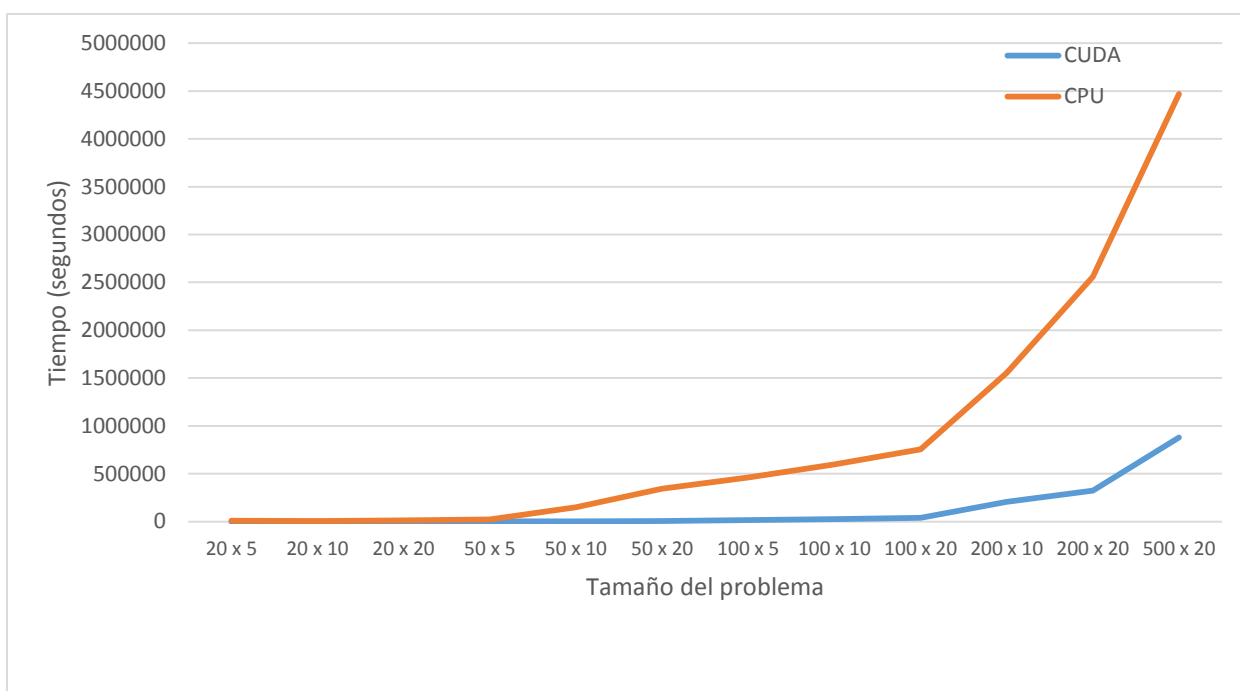


Figura 6 Comparación entre el tiempo de ejecución de CUDA y la CPU.

2.3.1 Análisis de los trabajos relacionados

A continuación se presenta una tabla comparativa de los trabajos relacionados, está organizada de la siguiente manera, en la primera columna se encuentran los nombres de dichos trabajos, en la segunda se mencionan las fases del algoritmo las cuales se paralelizan en cada trabajo, en la tercera columna se menciona la estrategia de mutación que se utiliza y en la cuarta el número de funciones *kernels* utilizadas para la paralelización. Todos los artículos mencionan mejoras para el algoritmo de evolución diferencial, la mayoría paralelizan todas las fases obteniendo

un mejor tiempo en sus ejecuciones y utilizan la misma estrategia de mutación a excepción del trabajo “*Cuda Based Enhanced Differential Evolution: A Computational analysis*” [9], que solo paraleliza una de las fases y utiliza una estrategia de mutación diferente; y el trabajo Implementación en paralelo del algoritmo JADE en el cual solo se paralelizan cuatro funciones *kernels* y utiliza una estrategia de mutación diferente a los demás trabajos. Las implementaciones varían en base al número de funciones *kernels* que se utilizan, esto es lo que marca la diferencia entre los resultados que se obtienen en cada uno de los trabajos analizados.

Es importante destacar que a la fecha no se ha encontrado evidencia alguna de la implementación del algoritmo JADE en paralelo, esto puede ser debido a que la mayoría de los artículos estudiados se enfocan en proponer mejoras al algoritmo de Evolución Diferencial, por lo anterior se podría decir que en el actual trabajo de tesis se realiza dicha paralelización.

Tabla 1 Comparativa de trabajos relacionados.

Trabajo	Fases paralelizadas	Estrategias de Mutación	Implementación	Algoritmo paralelizado
Implementación en paralelo del algoritmo JADE, 2016.	Evaluación de la función Mutación Cruce Selección	De/ current-to-pbest/1	Cuatro funciones de <i>Kernels</i> : utilizar un <i>kernel</i> para cada fase del algoritmo.	JADE: Evolución diferencial adaptativa con archivo externo opcional.
<i>Cuda Based Enhanced Differential Evolution: A Computational Analysis</i> , 2012.	Evaluación de la función.	DE/rand/2/bin	Una función de <i>kernel</i> .	Evolución diferencial.
<i>An Improved CUDA-Based Implementation of Differential Evolution on GPU</i> , 2012	Inicialización Evaluación de la función Preparación Mutación Mutación Cruce Selección	DE/rand/1/bin	Tres funciones de <i>kernels</i> : el primero para generar la inicialización y evaluación de la función, el segundo genera la preparación para la mutación, y el tercero que hace la mutación, cruce y selección.	Evolución diferencial.
<i>Many-threaded Differential Evolution on the GPU</i> , 2011.	Inicialización Evaluación de la función Valores de prueba Mutación Cruce Selección	DE/rand/1/bin	Seis funciones de <i>kernels</i> : utiliza un <i>kernel</i> para cada fase dentro de ellas agrega la paralelización de los valores de prueba.	Evolución diferencial.
<i>Differential Evolution Algorithm on the GPU with C-CUDA</i> , 2010.	Inicialización Evaluación de la función Mutación Cruce Selección	DE/rand/1/bin	Cinco funciones de <i>kernels</i> : utiliza un <i>kernel</i> para cada fase	Evolución diferencial.

Capítulo 3

IMPLEMENTACIÓN EN PARALELO DEL ALGORITMO JADE

En la literatura especializada existen algunas versiones del Algoritmo Evolutivo paralelizadas, cuya finalidad consiste en realizar versiones más rápidas en comparación con las versiones secuenciales, sin embargo, no se encontró evidencia de que la versión de JADE haya sido previamente paralelizada usando CUDA. De aquí el interés en estudiar e implementar dicho algoritmo utilizando los GPU's.

Este capítulo está estructurado de la siguiente forma, la Sección 3.1 presenta las características del algoritmo JADE, en la Sección 3.2 se muestra el análisis realizado para la identificación de las fases a paralelizar y su justificación así como el análisis de las características de los dispositivos de hardware a utilizar, en la Sección 3.3 se describen las funciones de evaluación a utilizar y finalmente en la Sección 3.4 se muestra la evidencia de cada una de las fases paralelizadas del algoritmo JADE.

3.1 CARACTERÍSTICAS DE LOS ALGORITMOS ADAPTATIVOS DE EVOLUCIÓN DIFERENCIAL

JADE es un algoritmo evolutivo diferencial adaptativo propuesto por Zhang y Sanderson en [5] [16], donde se prueba el algoritmo contra otros algoritmos clásicos o adaptativos en términos de rendimiento, tomando como medida la convergencia de un conjunto de funciones [28] [29].

La mayoría de los algoritmos adaptativos DE (como Sade, JDE, SaNSDE y JADE) siguen un procedimiento de aleatorización-adaptación, es decir, estos algoritmos generan aleatoriamente diferentes valores de los parámetros de control (factor de mutación F y probabilidad de cruce CR) según una cierta distribución de probabilidad, donde se propaga a la siguiente generación los valores que llevan una descendencia exitosa.

Las características principales de los diferentes algoritmos DE basados en aleatorización-adaptación se resumen en la Tabla 2 donde la columna *Algoritmos* muestra los nombres de los algoritmos, en la columna *Estrategia* de mutación se mencionan las estrategias de mutación que utiliza cada algoritmo, en la columna *Factor de mutación F* se mencionan las distribuciones que utiliza cada algoritmo para generar el factor de mutación, de la misma manera en la columna *Probabilidad de cruce CR* aparecen las distribuciones que son utilizadas para generar la probabilidad de cruce.

Tabla 2 Principales características de los algoritmos basados en aleatorización-adaptación.

Algoritmos	Estrategia de mutación	Factor de mutación F	Probabilidad de cruce CR
JADE	DE/current-to-pbest/	Distribución Cauchy y adaptación.	Distribución gaussiana y adaptación de parámetros.
DESAP		Distribución uniforme.	Distribución uniforme.
jDE	DE/rand/1	Distribución uniforme y adaptación.	Distribución uniforme y adaptación de parámetros.

SaDE	DE/rand/1 DE/current-to- best/2	Distribución gaussiana y adaptación.	Distribución gaussiana y adaptación de parámetros.
SaNSDE	DE/rand/1 DE/current-to- best/2	Distribución Cauchy y gaussiana.	Distribución gaussiana y adaptación de parámetros.

3.1.1 Evaluación de los algoritmos adaptativos de Evolución Diferencial

En esta sección se analizan los experimentos realizados por los autores Zhang y Sanderson donde JADE se compara con los algoritmos de evolución diferencial adaptativos jDE y SaDE, el clásico DE/rand/1/bin y un algoritmo de PSO canónico. Con la finalidad de realizar una comparación equitativa, en todas las simulaciones se establecieron los parámetros de JADE fijos, $p = 0.05$ y $c = 0.1$. Los parámetros de jDE y SaDE fueron tomados de sus documentos originales así como los parámetros para PSO. Los parámetros de DE/rand/1/bin se fijaron para $F=0.5$ y $CR=0.9$.

En todas las simulaciones, se establece el tamaño de la población NP de 100, y 400; dimensiones del problema D de 30 y 100, respectivamente. Todos los resultados reportados se obtienen sobre la base de 50 ejecuciones. Para mayor claridad los mejores resultados están marcados en negrita respectivamente. La media y la desviación estándar de los resultados obtenidos por cada algoritmo para las funciones F1-F13 se muestran en las Tabla 3 y Tabla 4, las cuales están organizadas de la siguiente manera: la primer columna tiene las funciones, la segunda columna contiene el número de generaciones para cada algoritmo, en la siguiente columna se encuentran los resultados del algoritmo JADE sin archivo tanto la media como la desviación estándar, en las siguientes columnas se encuentran los algoritmos JADE con archivo, jDE, SaDE, DE/rand/1/bin y PSO, en todos los casos se presenta la media y desviación estándar respectivamente.

Tabla 3 Resultados experimentales con una población de 100 y una dimensión del problema de 30, los resultados son el promedio de 50 ejecuciones.

	G	JADE sin archivo Media (Desviación estándar)	JADE con archivo Media (Desviación estándar)	jDE Media (Desviación estándar)	SaDE Media (Desviación estándar)	DE/rand/1/bin Media (Desviación estándar)	PSO Media (Desviación estándar)
<i>f1</i>	1500	1.8E-60 (8.4E-60)	<i>1.3E-54 (9.2E-54)</i>	2.5E-28 (3.5E-28)	4.5E-20 (6.9E-20)	9.8E-14 (8.4E-14)	9.6E-42 (2.7E-41)
<i>f2</i>	2000	1.8E-25 (8.8E-25)	3.9E-22 (2.7E-21)	<i>1.5E-23 (1.0E-23)</i>	1.9E-14 (1.05E-14)	1.6E-09 (1.1E-09)	9.3E-21 (6.3E-20)
<i>f3</i>	5000	<i>5.7E-61 (2.7E-60)</i>	6.0E-87 (1.9E-86)	5.2E-14 (1.1E-13)	9.0E-37 (5.43E-36)	6.6E-11 (8.8E-11)	2.5E-19 (3.9E-19)
<i>f4</i>	5000	<i>8.2E-24 (4.0E-23)</i>	4.3E-66 (1.2E-65)	1.4E-15 (1.0E-15)	7.4E-11 (1.82E-10)	4.2E-01 (1.1E+00)	4.4E-14 (9.3E-14)
<i>f5</i>	3000	8.0E-02 (5.6E-01)	<i>3.2E-01 (1.1E+00)</i>	1.3E+01 (1.4E+01)	2.1E+01 (7.8E+00)	2.1E+00 (1.5E+00)	2.5E+01 (3.2E+01)
	20000	8.0E-02 (5.6E-01)	3.2E-01 (1.1E+00)	8.0E-02 (5.6E-01)	1.8E+01 (6.7E+00)	8.0E-02 (5.6E-01)	1.7E+01 (2.3E+01)
<i>f6</i>	100	2.9E+00 (1.2E+00)	<i>5.6E+00 (1.6E+00)</i>	1.0E+03 (2.2E+02)	9.3E+02 (1.8E+02)	4.7E+03 (1.1E+03)	4.5E+01 (2.4E+01)
	1500	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	8.0E-02 (2.7E-01)
<i>f7</i>	3000	6.4E-04 (2.5E-04)	<i>6.8E-04 (2.5E-04)</i>	3.3E-03 (8.5E-04)	4.8E-03 (1.2E-03)	4.7E-03 (1.2E-03)	2.5E-03 (1.4E-03)
<i>f8</i>	1000	<i>3.3E-05 (2.3E-05)</i>	7.1E+00 (2.8E+01)	7.9E-11 (1.3E-10)	4.7E+00 (3.3E+01)	5.9E+03 (1.1E+03)	2.4E+03 (6.7E+02)
	9000	0.0E+00 (0.0E+00)	7.1E+00 (2.8E+01)	0.0E+00 (0.0E+00)	4.7E+00 (3.3E+01)	5.7E+01 (7.6E+01)	2.4E+03 (6.7E+02)
<i>f9</i>	1000	1.0E-04 (6.0E-05)	<i>1.4E-04 (6.5E-05)</i>	1.5E-04 (2.0E-04)	1.2E-03 (6.5E-04)	1.8E+02 (1.3E+01)	5.2E+01 (1.6E+01)
	5000	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	7.1E+01 (2.1E+01)	5.2E+01 (1.6E+01)
<i>f10</i>	500	8.2E-10 (6.9E-10)	<i>3.0E-09 (2.2E-09)</i>	3.5E-04 (1.0E-04)	2.7E-03 (5.1E-04)	1.1E-01 (3.9E-02)	4.6E-01 (6.6E-01)
	2000	4.4E-15 (0.0E+00)	4.4E-15 (0.0E+00)	4.7E-15 (9.6E-16)	4.3E-14 (2.6E-14)	9.7E-11 (5.0E-11)	4.6E-01 (6.6E-01)
<i>f11</i>	500	9.9E-08 (6.0E-07)	2.0E-04 (1.4E-03)	<i>1.9E-05 (5.8E-05)</i>	7.8E-04 (1.2E-03)	2.0E-01 (1.1E-01)	1.3E-02 (1.7E-02)
	3000	0.0E+00 (0.0E+00)	2.0E-04 (1.4E-03)	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	1.1E-02 (1.6E-02)
<i>f12</i>	500	4.6E-17 (1.9E-16)	<i>3.8E-16 (8.3E-16)</i>	1.6E-07 (1.5E-07)	1.9E-05 (9.2E-06)	1.2E-02 (1.0E-02)	1.9E-01 (3.9E-01)
	1500	1.6E-32 (5.5E-48)	1.6E-32 (5.5E-48)	2.6E-29 (7.5E-29)	1.2E-19 (2.0E-19)	1.1E-14 (1.0E-14)	1.9E-01 (3.9E-01)
<i>f13</i>	500	2.0E-16 (6.5E-16)	<i>1.2E-15 (2.8E-15)</i>	1.5E-06 (9.8E-07)	6.1E-05 (2.0E-05)	7.5E-02 (3.8E-02)	2.9E-03 (4.8E-03)
	1500	1.4E-32 (1.1E-47)	1.4E-32 (1.1E-47)	1.9E-28 (2.2E-28)	1.7E-19 (2.4E-19)	7.5E-14 (4.8E-14)	2.9E-03 (4.8E-03)

Tabla 4 Resultados experimentales con una población de 400 y una dimensión del problema de 100, los resultados son el promedio de 50 ejecuciones.

	G	JADE sin archivo Media (Desviación estándar)	JADE con archivo Media (Desviación estándar)	jDE Media (Desviación estándar)	SaDE Media (Desviación estándar)	DE/rand/1/bin Media (Desviación estándar)	PSO Media (Desviación estándar)
<i>f1</i>	2000	<i>1.2E-48 (1.5E-48)</i>	5.4E-67 (1.6E-66)	5.0E-15 (1.7E-15)	2.9E-08 (3.2E-08)	3.5E+01 (9.6E+00)	6.0E-11 (2.5E-10)
<i>f2</i>	3000	<i>1.1E-41 (5.1E-41)</i>	9.2E-51 (2.2E-50)	4.1E-15 (1.1E-15)	1.7E-05 (3.8E-06)	2.3E+00 (5.6E-01)	2.8E-04 (1.3E-03)
<i>f3</i>	8000	<i>1.2E-26 (2.0E-26)</i>	2.2E-37 (2.5E-37)	5.4E-02 (2.7E-02)	2.4E-13 (5.2E-13)	2.1E+05 (3.1E+04)	1.2E+02 (6.7E+01)
<i>f4</i>	15000	1.9E-02 (1.5E-02)	3.2E-71 (8.3E-71)	<i>3.1E-09 (5.9E-10)</i>	1.1E+00 (4.0E-01)	9.3E+01 (2.8E+00)	4.9E+01 (2.5E+01)
<i>f5</i>	6000	<i>5.6E-01 (1.4E+00)</i>	4.0E-01 (1.2E+00)	7.2E+01 (1.1E+01)	9.4E+01 (4.0E-01)	9.5E+01 (1.4E+01)	1.3E+02 (4.8E+01)
	20000	5.6E-01 (1.4E+00)	4.0E-01 (1.2E+00)	9.1E-03 (2.5E-02)	9.1E+01 (3.1E-01)	4.3E+01 (1.2E+01)	6.3E+01 (4.2E+01)
<i>f6</i>	100	1.1E+02 (1.5E+01)	<i>1.2E+02 (1.3E+01)</i>	7.1E+04 (6.0E+03)	3.3E+04 (2.1E+03)	1.8E+05 (1.8E+04)	1.9E+04 (4.5E+03)
	1500	1.6E-01 (3.7E-01)	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	3.2E+02 (6.3E+01)	4.7E+01 (7.9E+01)
<i>f7</i>	6000	<i>1.1E-03 (2.1E-04)</i>	7.8E-04 (1.4E-04)	8.1E-03 (9.0E-04)	1.0E-02 (4.9E-03)	2.9E-02 (5.7E-03)	9.2E-03 (2.6E-03)
<i>f8</i>	1000	8.9E+03 (3.0E+02)	8.6E+03 (4.2E+02)	4.9E+03 (4.1E+02)	<i>5.4E+03 (3.7E+02)</i>	3.2E+04 (4.7E+02)	9.5E+03 (1.3E+03)
	9000	1.1E-10 (0.0E+00)	1.1E-10 (0.0E+00)	1.1E-10 (0.0E+00)	1.1E-10 (0.0E+00)	3.0E+04 (9.0E+02)	9.4E+03 (1.2E+03)
<i>f9</i>	3000	1.9E-01 (3.8E-02)	2.0E-01 (3.7E-02)	2.1E-04 (2.1E-04)	<i>9.1E-03 (1.8E-03)</i>	8.6E+02 (2.2E+01)	3.4E+02 (4.4E+01)
	9000	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	0.0E+00 (0.0E+00)	8.1E+02 (1.8E+01)	3.4E+02 (4.4E+01)
<i>f10</i>	500	<i>7.9E-06 (2.6E-06)</i>	4.2E-07 (1.2E-07)	8.5E-01 (1.2E-01)	1.6E+00 (1.2E-01)	1.5E+01 (5.8E-01)	3.6E+00 (9.3E-01)
	3000	8.9E-15 (2.1E-15)	8.0E-15 (0.0E+00)	9.9E-14 (2.0E-14)	2.1E-07 (1.0E-07)	1.3E-01 (2.4E-02)	2.6E+00 (6.8E-01)
<i>f11</i>	500	<i>3.9E-04 (2.0E-03)</i>	1.5E-04 (1.0E-03)	1.1E+00 (2.0E-02)	1.1E+00 (1.8E-02)	2.7E+02 (4.4E+01)	1.0E+00 (5.6E-01)
	3000	3.9E-04 (2.0E-03)	1.5E-04 (1.0E-03)	0.0E+00 (0.0E+00)	8.6E-13 (8.2E-13)	2.0E-01 (5.8E-02)	8.8E-02 (2.5E-01)
<i>f12</i>	500	2.2E-11 (1.2E-11)	2.8E-13 (9.8E-13)	4.0E+00 (6.8E-01)	2.4E+00 (3.9E-01)	1.8E+09 (5.1E+08)	1.1E+01 (3.4E+00)
	3000	4.7E-33 (2.2E-34)	4.7E-33 (6.8E-49)	1.7E-25 (7.7E-26)	1.4E-11 (9.1E-12)	1.7E+00 (1.5E+00)	1.3E-01 (1.6E-01)
<i>f13</i>	500	<i>1.9E-09 (1.5E-09)</i>	5.8E-12 (5.5E-12)	3.1E+01 (7.8E+00)	1.2E+01 (1.8E+00)	2.4E+09 (1.1E+09)	9.8E+01 (2.4E+01)
	3000	1.4E-32 (1.1E-47)	1.4E-32 (1.1E-47)	2.1E-24 (1.5E-24)	1.3E-11 (9.6E-12)	5.1E+00 (3.2E+00)	1.4E-01 (5.6E-01)

Analizando los resultados presentados en la Tabla 3 y Tabla 4, se puede realizar las siguientes observaciones en relación a la tasa de convergencia y la fiabilidad de los diferentes algoritmos.

En primer lugar, los resultados sugieren que la tasa de convergencia global de JADE con o sin archivo es mejor para el conjunto de problemas f1-f13: JADE sin archivo funciona mejor con una baja dimensión del problema ($D = 30$), mientras que JADE con archivo alcanza el mejor rendimiento en el caso de problemas de alta dimensión del problema ($D = 100$). En ambos casos JADE (con y sin archivo) converge más rápido para la optimización de f1-f7 y f10-f13. En los casos de f8 y f9, jDE se comporta mejor y JADE logra resultados muy competitivos. Las tasas de convergencia de SaDE y PSO son generalmente peores que JADE y jDE, pero son mejores que DE/rand/1/bin.

En segundo lugar, es importante comparar la fiabilidad de diferentes algoritmos, como se muestra en las Tablas 4 y 5. Los algoritmos PSO y DE/rand/1/bin muestran limitantes en relación a que el primer algoritmo sufre de convergencia prematura frecuente mientras que el segundo algoritmo converge muy lentamente dentro de un número limitado de evaluaciones de la función. SaDE es mejor que el algoritmo PSO y DE/rand/1/bin pero es claramente menos satisfactorio para algunos problemas, especialmente cuando la dimensión del problema es alta. JDE muestra que funciona muy bien, ya que su estrategia de mutación subyacente "DE/rand/1" es más robusta que las estrategias voraces. Es interesante que la fiabilidad de JADE sea similar a la de JDE para todas las 13 funciones escalables. La alta fiabilidad y la rápida convergencia de JADE se derivan tanto del control adaptativo de los parámetros, la diversidad proporcionada por el archivo y la estrategia de mutación "DE/current-to- pbest / 1".

En base a los resultados de los experimentos diferentes autores (Tabla 3 y 4), se puede observar que JADE presenta un mejor desempeño en el manejo de grandes dimensiones, lo cual, nos llevó a realizar su versión paralelizada haciendo

uso de los GPUs para ver que tanto podría reducir sus tiempos en relación a su forma secuencial.

3.1.2 Análisis de las características de los dispositivos de hardware

Para determinar la configuración adecuada en los dispositivos GPUs se revisaron diferentes trabajos del estado del arte orientados a la paralelización en GPU de algoritmos Evolutivos Diferenciales.

En la Tabla 5 se muestran las diferentes configuraciones que fueron utilizadas en los artículos revisados en el estado del arte, la tabla está organizada de la siguiente forma: en la primer columna se presenta el nombre de los artículos estudiados, en la segunda columna se encuentra el algoritmo utilizado, la columna Configuración muestra la configuración que utilizan los autores, es decir el número de hilos por bloque dentro de los dispositivos, la columna Hardware describe el equipo que se utilizó para realizar las experimentaciones, en la columna Justificación se explica brevemente el porqué de las configuraciones. Es importante destacar que en algunos artículos los autores definen sus configuraciones empíricamente, mientras que otros autores buscan, de forma experimental, la configuración que les permita aprovechar al máximo la capacidad de sus dispositivos tomando en cuenta las características de hardware. Por último, la columna Parámetros muestra los diferentes tamaños de población, número de generaciones y las dimensiones que se utilizaron para la experimentación.

Tabla 5 Configuraciones utilizadas en la literatura especializada para diferentes versiones del algoritmo DE en CUDA.

TRABAJO	ALGORITMO	CONFIGURACIÓN		HARDWARE	JUSTIFICACIÓN	PARAMETROS
		Bloques	Hilos			
<i>Differential Evolution Algorithm on the GPU with C-CUDA</i> , 2010.	ED	32	64	NVIDIA GTX 285 con 1GB de GDDR3		NP= 100, 1000 D= 100 G= 1000
<i>Many-threaded Differential Evolution on the GPU</i> , 2011.	ED		512	NVIDIA Tesla C2050 con 448 núcleos a 1.15GHz	Para maximizar la utilización de los recursos proporcionados por las GPUs.	NP= 100 D= 128
<i>Fully Parallel Differential Evolution 2011</i>	FPDE	N/16	16	NVIDIA Tesla C2050	Configuración definida empíricamente.	NP= 8192 D= 128 G= 3000
<i>An Improved CUDA-Based Implementation of Differential Evolution on GPU</i> , 2012	ED (2010)	240	64	NVIDIA GTX560 con 1 GB de memoria GDDR5	Los parámetros de configuración se generan de forma automática para maximizar la ocupación del dispositivo.	NP= 50, 100, 500, 1000 D= 10, 50, 100 G= 25
	FPDE (2011)	N/16	16			
	cudaED		512			
<i>Cuda Based Enhanced Differential Evolution: A Computational Analysis</i> , 2012.	ED		512	NVIDIA Tesla C2050	Maximiza la utilización del dispositivo.	NP= 100 D= 30 G= 100

NP= tamaño de la población D= dimensión del problema G= número de generaciones de la población.

En la Tabla 5 se observa que la mayoría de los trabajos maximizan el uso del dispositivo tomando en cuenta sus capacidades así como también el tipo de estrategia al momento de analizar el algoritmo para identificar las fases que puedan paralelizarse y así hacer el algoritmo más rápido. Dado que en el estado del arte se observa que encontrar la configuración adecuada es una labor empírica, se optó por realizar experimentación para determinar la configuración más adecuada tomando como base las configuraciones de las versiones del algoritmo de Evolución Diferencial en CUDA propuesto en los artículos mencionados.

La experimentación se llevó a cabo en una computadora con las siguientes características: procesador Intel i7 Ghz, sistema Operativo Windows 7, tarjeta gráfica NVIDIA GT 650M con 2GB de GDDR5 de memoria global y con 384 cores.

3.2 ANÁLISIS DE LAS FASES DEL ALGORITMO JADE

Al analizar el algoritmo JADE en su versión secuencial se identificaron las fases que pueden paralelizarse. En la Figura 7 se muestra un esquema donde los recuadros rojos indican los procesos que se pueden paralelizar, los incisos representan el número de *kernel* (en este trabajo se usó un *kernel* para cada fase) y los recuadros amarillos indican los procesos que se ejecutan en cada generación.

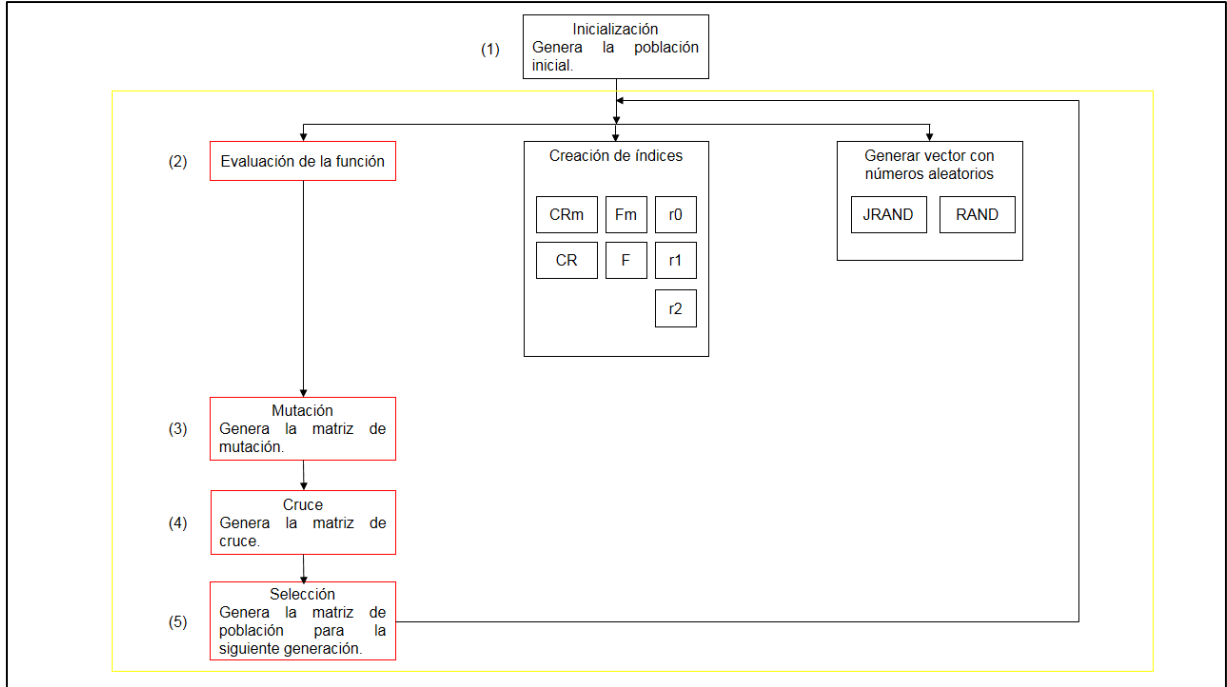


Figura 7 Esquema general de las fases del algoritmo. Los recuadros rojos indican las fases que fueron paralelizadas.

A continuación se muestra de forma gráfica cómo se llevó a cabo la paralelización dentro de los *Kernels*, es decir cómo va cambiando la población en cada fase:

3.2.1 Inicialización

Se genera la población de manera aleatoria dentro del host para cada generación (g), la población tiene un tamaño NP ($i=1,2,3,\dots, NP$), y cada individuo en una población tiene D características ($j=1,2,3,\dots, D$); por tanto una población es representada por una matriz como la que se muestra en la Figura 8 donde las filas representan a cada individuo y las columnas sus características, es importante destacar que esta representación (Figura 8) es la manera en que se guardan los datos de la población en el host, lo cual facilita la manipulación de los datos a través de las fases. Por ejemplo al crear una población de $NP=2$ con $D=5$ características, con valores en $x \in [-100,100]$ como se muestra en la matriz x :

x	-45	72	38	-64	-54
	1	-82	12	89	96

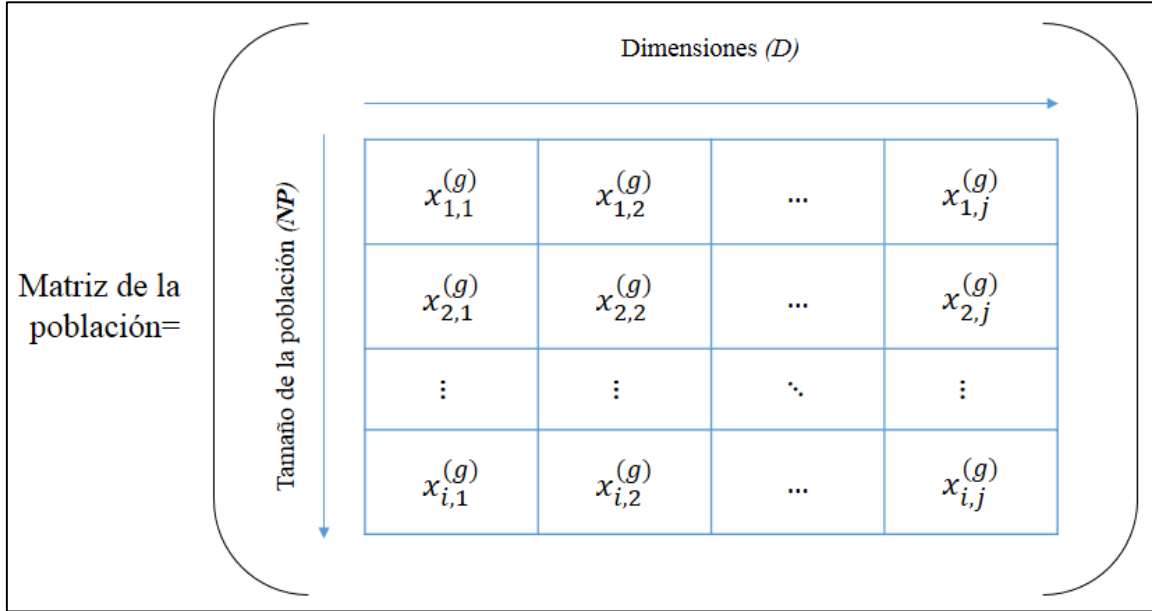


Figura 8 Inicialización de la población

3.2.2 Evaluación de la función

Una vez generada la población inicial o primera generación (g), ésta es mandada al *kernel* para llevar a cabo la segunda fase correspondiente a la evaluación de la función, Tomando el ejemplo de la matriz de población de la fase anterior al evaluar la función *Sphere* (14):

$$f(x) = \sum_{i=1}^D z_i^2$$

$$z = x - o, \quad x = [x_1, x_2, \dots, x_D], \quad o = [o_1, o_2, \dots, o_D] \quad (14)$$

Donde x es la matriz de población, o son valores en un rango de $o \in [-100, 100]$, dichos valores son proporcionados en el CEC [26], z es el resultado de la diferencia entre la matriz x menos la matriz o :

o	-39.3119	58.8999	-46.3224	-74.6515	-16.7997
	-39.3119	58.8999	-46.3224	-74.6515	-16.7997

z	-6	13	84	11	-37
	40	-141	58	164	113

Aplicando la función *Sphere* descrita en la ecuación 14 se obtiene el vector con los valores óptimos $F(x)$:

f(X)	8811.55
	64384.92

Se toma el valor más óptimo y en base a su índice dentro del vector se toma el individuo de la matriz de población x en este caso es el individuo 1 el cual es el que se acerca más a la función objetivo creando el vector x_{best} :

xbest	-45	72	38	-64	-54
-------	-----	----	----	-----	-----

En la Figura 9 se muestra cómo se acomodan los datos de la población dentro del *grid*, en forma de matriz, donde cada dato es evaluado de manera independiente. El tamaño del grid se calcula aplicando la ecuación 15:

$$tamaño_{grid} = \frac{NP-1}{anchoBloque+1}, \frac{D-1}{anchoBloque+1} \quad (15)$$

donde NP es el tamaño de la población, D indica el número de las dimensiones o características y *anchoBloque* es la longitud del bloque. Por ejemplo, al utilizar una población de $NP=100$, una dimensión con $D=10$ y una longitud del bloque *anchoBloque* =32, ya que la capacidad máxima del hardware es de 1024 hilos se utilizaron bloques con dimensiones de 32x32 hilos; sustituyendo los valores en la ecuación 14, se obtiene el siguiente resultado:

$$tamaño_{grid} = \frac{100-1}{32+1}, \frac{10-1}{32+1}$$

$$\text{tamaño}_{\text{grid}} = 3,1$$

Cada uno de los datos de la población es evaluado por un hilo, en esta fase se aplica la función objetivo para generar un vector con los valores objetivos:

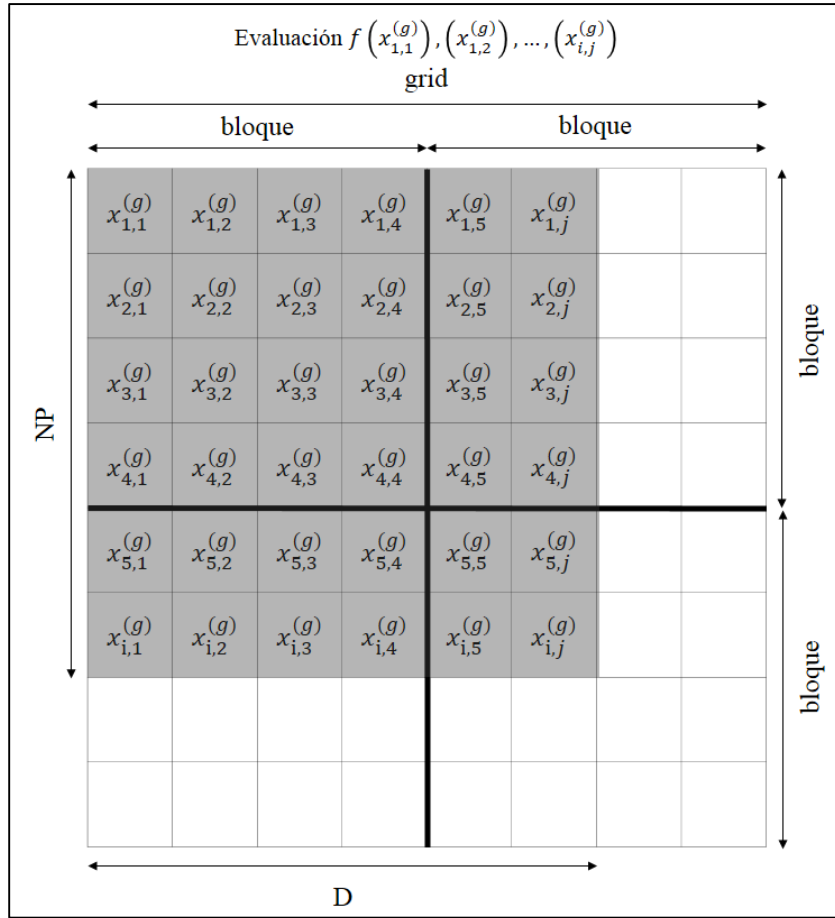


Figura 9 Evaluación de la función

3.2.3 Mutación

La Figura 10 muestra cómo va cambiando la población al aplicar el proceso de la mutación, una vez evaluados los individuos en la fase 2 se les implementa la estrategia de mutación "DE/current-to-pbest" la cual consiste en evaluar a cada

individuo para determinar su grado de aptitud, es decir si se acerca o no a la solución, aplicando la estrategia de mutación de la ecuación (16) al ejemplo se tiene la matriz de mutación v :

$$v = x + F \cdot (xbest - x) + F \cdot (r1 - r2) \quad (16)$$

donde x es la matriz de población, el factor de mutación $F=0.5$, el vector con el mejor individuo $xbest$, $r1$ y $r2$ son individuos aleatorios de la población actual.

v	-68	149	51	-140.5	-129
	-45	72	38	-64	-54

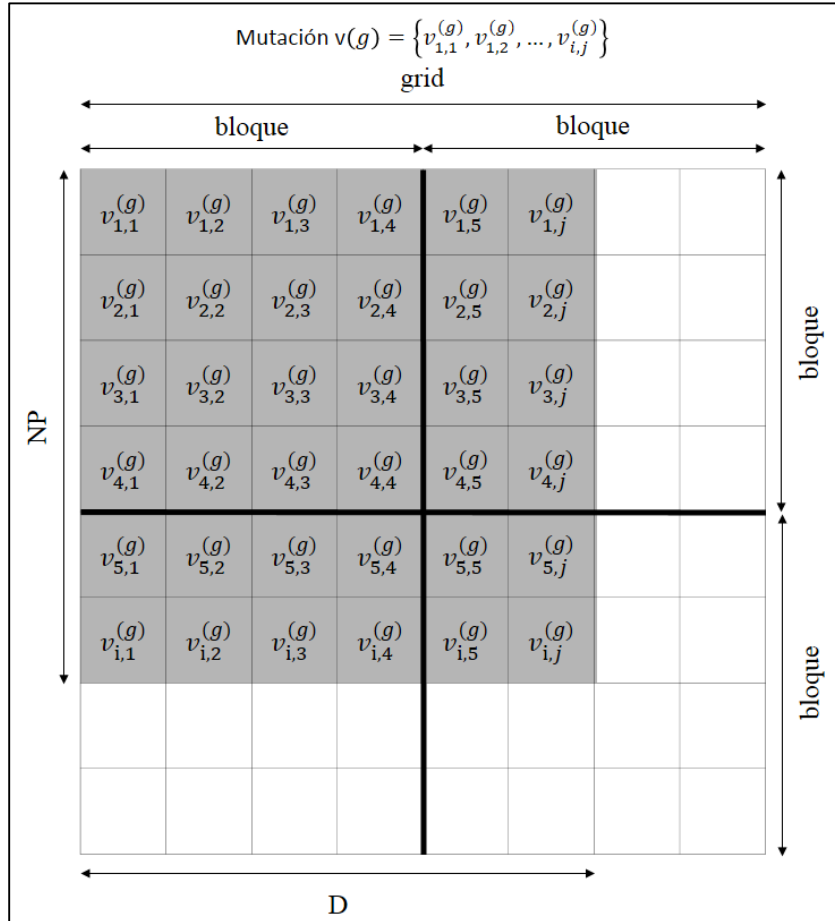


Figura 10 Mutación

3.2.4 Cruce

Una vez aplicada la fase de mutación se realiza una recombinación de la población generada en la matriz x y la matriz de mutación v , formando una matriz de prueba u , como se muestra en la ecuación 16:

$$u = \begin{cases} v & \text{si } rand(0,1) \leq CR \text{ o } j = jrand \\ x & \text{en otro caso} \end{cases} \quad (16)$$

donde $rand(0,1)$ es un vector de números aleatorios entre 0 y 1, la probabilidad de cruce $CR=0.5$ y el vector j son los valores de los índices de la matriz de población; la función de cruce va cargando las características de los individuos conforme las condiciones de la ecuación 16 donde si los valores del vector $rand$ es mayor o igual a la probabilidad de cruce CR o si los índices del vector j son iguales a índices aleatorios del vector $jrand$ entonces se toma el valor de la matriz de mutación v de lo contrario se carga la característica de la matriz de población x :

u	-68	72	51	-64	-54
	1	72	38	-64	96

En la Figura 11 se muestra el cambio dentro del *grid* al realizar la fase de cruce:

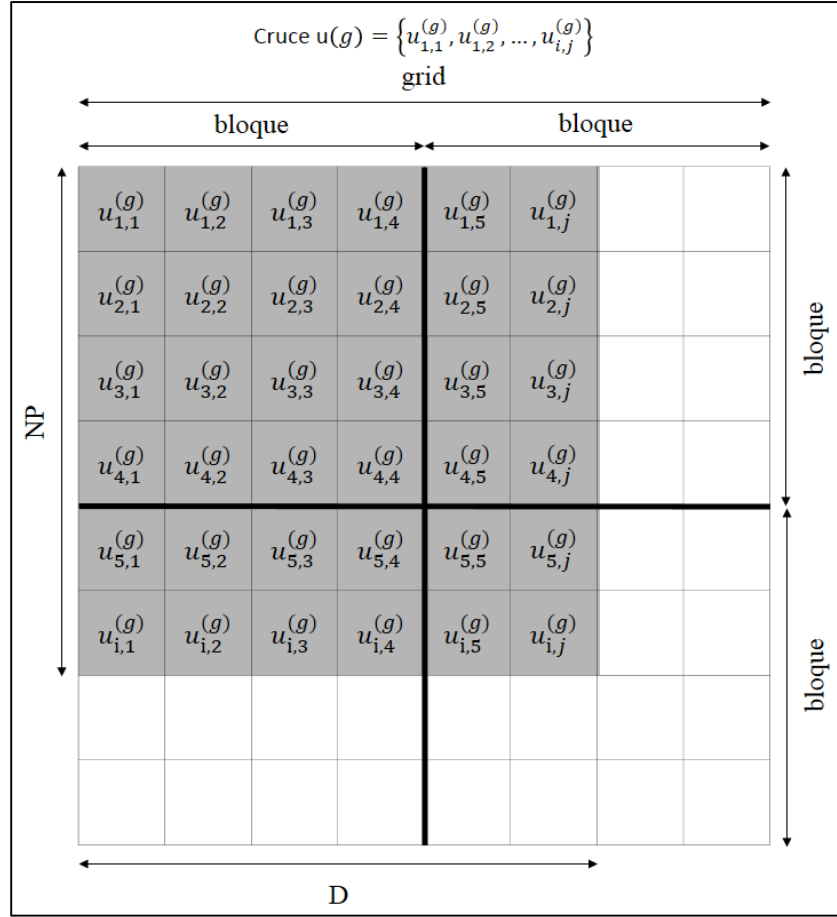


Figura 11 Cruce

3.2.5 Selección

La operación de selección elige el mejor individuo entre el vector padre x y el vector de prueba u generado en la operación de cruce; acorde a los valores de la función de evaluación, como se muestra en la ecuación 17; para generar la población de la nueva generación dando como resultado la matriz $x2$:

$$x2 = \begin{cases} u & \text{si } f(u) < f(x) \\ x & \text{en otro caso} \end{cases} \quad (17)$$

donde toma el primer individuo de la matriz de población y el segundo es obtenido de la matriz de cruce:

x2	-45	72	38	-64	-54
	1	72	38	-64	96

La Figura 12 muestra el proceso de selección dentro del *grid*.

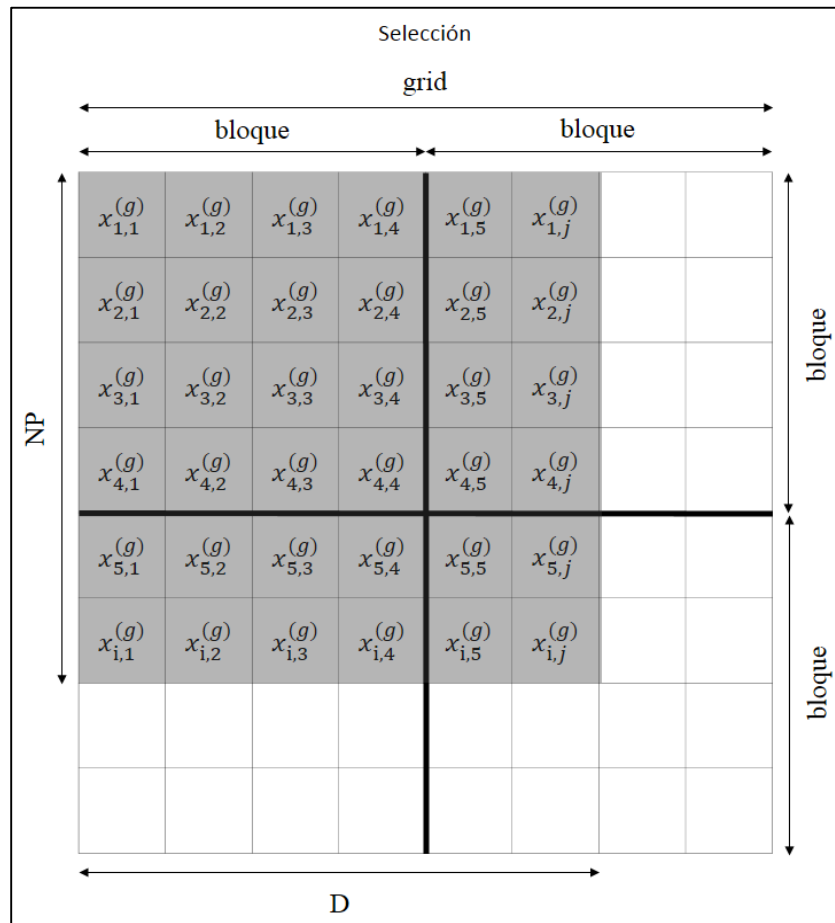


Figura 12 Selección

3.3 FUNCIONES DE EVALUACIÓN

La optimización consiste encontrar valores mínimos o máximos que resuelvan una determinada problemática, por lo general representada por medio de una función. En este contexto, la función de prueba también llamada función objetivo, aporta un problema artificial que sirve para evaluar la efectividad de un algoritmo de optimización [13].

En las secciones 3.3.1- 3.1.3 se presentan las funciones de prueba para evaluar la versión paralela del algoritmo JADE.

3.3.1 Función de evaluación “Sphere”

En la Figura 13 se puede apreciar la función de prueba “**Sphere**”, la cual, es continua, convexa y unimodal, esto quiere decir que sólo se encuentra un óptimo global en todo el espacio de búsqueda que define la función, La función “**Sphere**” se representa matemáticamente como lo indica la expresión 16.

$$f(x) = \sum_{i=1}^D z_i^2$$

$$z = x - o, \quad x = [x_1, x_2, \dots, x_D], \quad o = [o_1, o_2, \dots, o_D] \quad (18)$$

En donde, D corresponde al número de características del problema, es decir el número de variables en las que se realiza la optimización, x es un vector cuyos valores oscilan en $x \in [-100, 100]$, el vector x es presentado como individuo en los algoritmos de evolución diferencial, o es un vector con valores en un rango de $o \in [-100, 100]$, dichos valores son proporcionados en el CEC 2005 [26], z es un vector resultado de la diferencia del vector x menos el vector o .

La Figura 13 muestra la representación gráfica de la función de evaluación “**Sphere**”, la función es graficada en un cubo de dimensiones $[-5.12, 5.12]$, donde eje y y eje x de la gráfica se les asignan los valores de $[-5.12, 5.12]$ para tener una imagen 3D de la función.

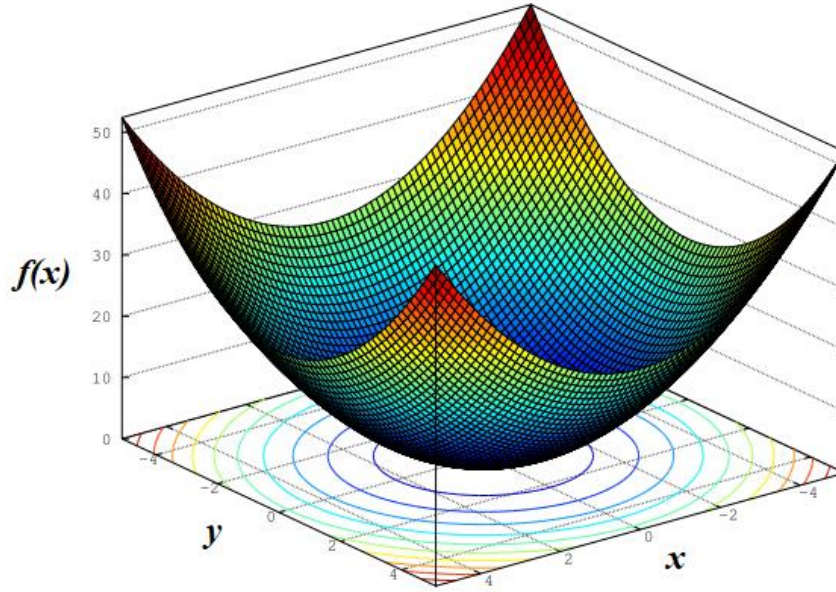


Figura 13 Función “Sphere” evaluada en un cubo de [-5.12, 5.12].

3.3.2 Función de evaluación “Rosenbrock”

La función “**Rosenbrock**” también conocida como Valle o función banana, es una función de prueba popular en algoritmos de optimización, es unimodal y su mínimo global reside en un estrecho y parabólico valle, a pesar de que el valle es fácil de encontrar, en general los algoritmos de búsqueda y optimización tardan en descubrir el mínimo global. La función “**Rosenmbrock**” se representa matemáticamente por la expresión 17.

$$f(x) = \sum_{i=1}^{D-1} (100(z_i^2 - z_{i+1})^2 + (z_i - 1)^2)$$

$$z = x - o, \quad x = [x_1, x_2, \dots, x_D], \quad o = [o_1, o_2, \dots, o_D] \quad (19)$$

Donde la variable D representa las dimensiones del problema, x es un individuo con D características, o es un vector con D valores obtenidos del CEC 2005 [26], el vector z es la diferencia de x y o ; además, la función se caracteriza por tomar hasta $D-1$ características debido a que en el primer paréntesis $(z_i^2 - z_{i+1})$ se calcula una resta con la característica siguiente en el vector z .

La Figura 14 muestra de manera gráfica la función de prueba “**Rosenbrock**”, para generar una imagen 3D de la función se utilizó un cubo con el rango $[-2.048, 2.048]$. Como se puede observar en la Figura 14, la mayor parte del espacio es ocupado por un valle prolongado, dentro de esta zona recta el algoritmo debe encontrar el mínimo global, es por esta razón que por lo general los algoritmos de búsqueda y optimización toman más tiempo en converger [30].

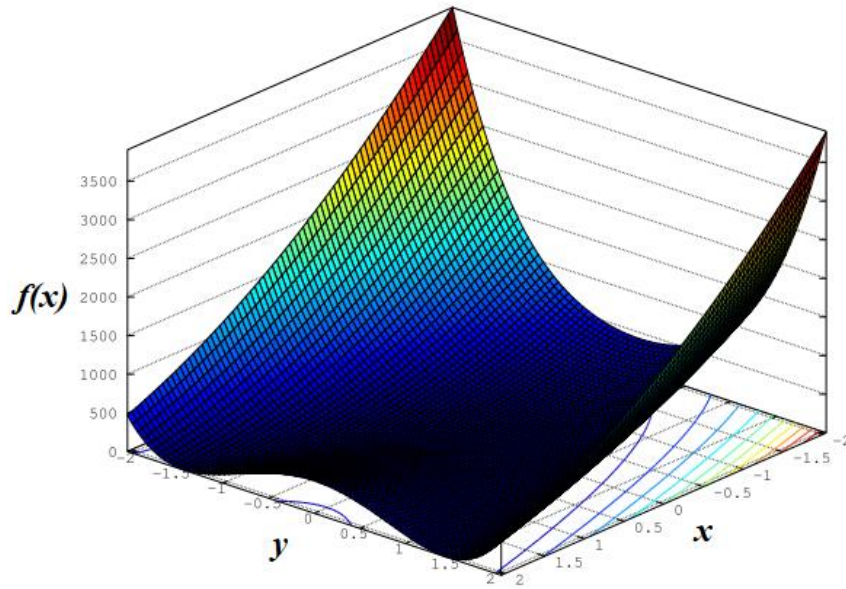


Figura 14 Función “*Rosenbrock*” evaluada en un cubo de $[-2.048, 2.048]$.

3.3.2 Función de evaluación “**Rastrigin**”

La función de prueba “**Rastrigin**” tiene varios mínimos locales lo que significa que es una función multimodal, como se observa en la Figura 15, la posición de los mínimos locales está distribuida uniformemente, cabe mencionar que encontrar el mínimo global de esta función es difícil debido a su extenso espacio de búsqueda y su gran número de mínimos locales. La definición matemática de la función “**Rastrigin**” se muestra a continuación:

$$f(x) = \sum_{i=1}^D (z_i^2 - 10 \cos(2\pi z_i) + 10)$$

$$z = x - o, \quad x = [x_1, x_2, \dots, x_D], \quad o = [o_1, o_2, \dots, o_D] \quad (20)$$

Donde D es el número de características, x es un vector que representa a un individuo, o es un vector con valores numéricos extraídos del CEC 2005 [26] y z es un vector con la diferencia de x y o . La representación 3D de la función de evaluación “*Rastrigin*” se muestra en la Figura 15, se puede observar que la gráfica está compuesta por ondas sinusoidales que envuelven la curvatura de la función “*Sphere*”, lo cual, genera su forma característica y la aparición de mínimos locales.

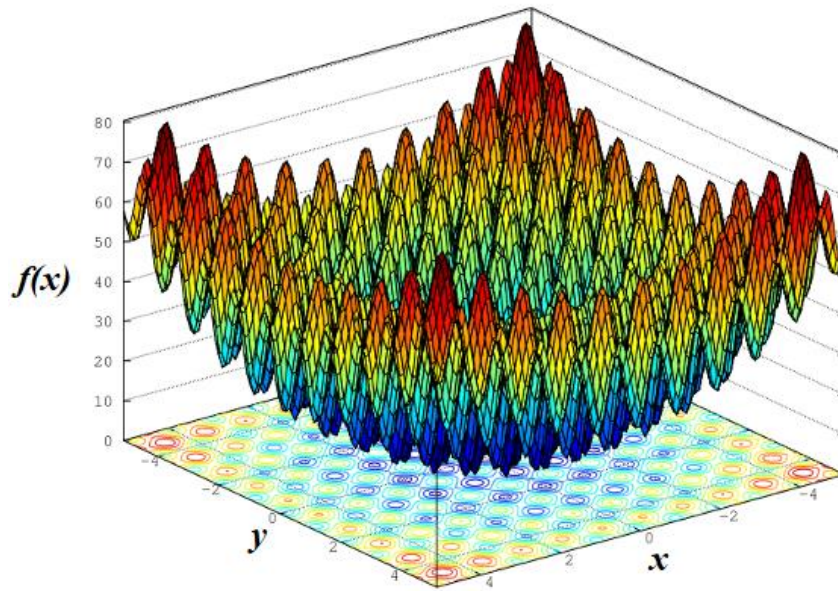


Figura 15 Función “*Rastrigin*” evaluada en un cubo de $[-5.12, 5.12]$.

3.4 IMPLEMENTACIÓN PARALELA DE JADE UTILIZANDO CUDA

En esta sección se muestran algunos de los segmentos de código implementados en CUDA que corresponden a la versión paralela del algoritmo JADE. Es importante mencionar que inicialmente se reserva memoria en el host para almacenar los datos que se usarán durante la ejecución del algoritmo, también se asigna memoria en el GPU para los vectores/matriz que almacenarán los datos a procesar; asimismo, se calcula el tamaño del *grid* de acuerdo al tamaño de la población. Una vez asignada la memoria y generado el tamaño del *grid* se manda llamar las funciones *kernel*,

cada fase del algoritmo tiene su propia función según se muestra en las Figuras 13-17.

Paso 1.- Inicialización: En este paso se genera la población inicial en la cual los individuos son generados aleatoriamente dentro de una matriz en el *host* (Figura 16), la línea 2 muestra la declaración de la función la cual recibe los parámetros: referencia a la matriz de la población (**pop**), tamaño de la población (*psize*), las dimensiones o características *d*, los límites mínimo y máximo que definen el rango de los valores de las características (*boundmin* y *boundmax*) los cuales se fijaron de -100 a 100, en la línea 3 se define un ciclo que recorre la matriz **pop** en filas, en la línea 4 se crea un ciclo anidado que recorre la matriz **pop** por columnas, la línea 5 establece aleatoriamente un valor dentro de los límites definidos -100 a 100 en cada celda de la matriz:

```

1 //Inicialización
2 void initialization(float *pop,int psize,int d,int boudmax, int boundmin){
3     for (int row = 0; row < psize; row++)
4         for(int col = 0; col < d; col++)
5             pop[row * d + col] = boundmin + ((float)rand() / RAND_MAX) * (boudmax - boundmin);
6 }

```

Figura 16 Fase de Inicialización.

Paso 2.- Evaluación de la Función: En esta función se determina la aptitud de los individuos como se muestra en la Figura 17, la línea 2 es la declaración de la función *kernel* la cual recibe como parámetros: la matriz de población **devPopold** con dimensiones de $NP \times D$, el vector donde se guardan los resultados **devvalParents**, la matriz donde se almacena un resultado temporal **devx**, la matriz con los valores de la función de evaluación **devRepmatO**; la línea 3 y 4 definen los índices (coordenadas x,y) dentro del *grid*; la línea 5 muestra la condición que evita que los índices sobrepasen las dimensiones de la matriz de la población; la línea 6 genera la matriz temporal **devx** restando a la matriz de población **devPopold** la matriz con los valores de la función **devRepmatO**; en la línea 7 se manda llamar la función *_syncthread*s(), dicha función define una barrera de sincronización la cual no permite seguir con los cálculos hasta que la instrucción anterior haya terminado su ejecución; la línea 8 crea una variable acumuladora *sum* inicializada en cero; en

la línea 9 se recorren las características por individuo; en la línea 10 se establece la variable *sum* como la suma de las características elevadas al cuadrado; en la línea 12 se verifica que la variable acumulada *sum* no haya excedido el máximo valor permitido de un valor tipo *float*, si la condición es verdadera, en la línea 13 se fija como resultado el valor máximo permitido de un tipo *float*; en caso contrario se establece el valor de la variable acumulada *sum* en el vector resultante ***devvalParents*** como se muestra en la línea 15.

```

1 //Función de evaluación Sphere
2 global void sphere(float *devPopold, float *devvalParents, float *devx, float *devRepmat0, int psize, int d){
3     int idx = blockIdx.x * blockDim.x + threadIdx.x;
4     int idy = blockIdx.y * blockDim.y + threadIdx.y;
5     if(idy < psize && idx < d){
6         devx[idy * d + idx] = devPopold[idy * d + idx] - devRepmat0[idy * d + idx];
7         __syncthreads();
8         float sum = 0;
9         for(int k=0; k<d; k++){
10             sum+= powf(devx[idy*d + k], 2.0);
11         }
12         if (isinf(sum)){
13             devvalParents[idy]= FLT_MAX;
14         }else{
15             devvalParents[idy]= sum;
16         }
17     }
18 }

```

Figura 17 Fase de Evaluación de la Función.

Paso 3.- Mutación: En la Figura 18 se muestra la función *kernel* que realiza la mutación o perturbación de la población; en la línea 2 se define la función tomando como parámetros: la matriz de mutación ***devVI*** en donde se almacena el resultado de aplicar la estrategia de mutación “DE/current-to-*pbest*” a la población, la matriz de población ***devPopold***, la matriz ***devArchive*** que contiene individuos de generaciones anteriores, el vector ***devR1*** que contiene índices aleatorios diferentes al índice consecutivo de la matriz de población, el vector ***devR2*** con índices aleatorios considerando también los índices de la matriz ***devArchive*** y diferentes a los índices ***devR1***, el vector ***devFs*** con los factores de mutación que se aplica a cada individuo, el vector ***devindexRand*** con índices aleatorios con valores que van desde 0 hasta *p* (*p* indica el porcentaje de los que con mejor aptitud se tomaran en la fórmula de mutación), el vector ***devBestIndex*** el cual posee los índices de la población ordenados en base a la aptitud; en la línea 4 se definen los índices de la coordenada “x” del *grid*; en la línea 5 se establece la coordenada “y” del *grid*; la línea

6 condiciona que las operaciones se realicen sólo dentro de las dimensiones de la matriz de población; en la línea 7 se evalúa si el índice contenido en **devR2** es mayor al tamaño de la población, si esta condición se cumple se utiliza la fórmula de mutación usando la matriz del archivo; en las líneas 8-10 se almacena en la matriz **devVI** el resultado de otro modo aplica la fórmula usando sólo los individuos de la población, líneas 12-14. En la línea 16 se llama la función `_syncthread()` para que todos los procesos lleguen a ese punto, la sentencia en la línea 17 evalúa si el valor generado en la matriz **devVI** es menor al límite inferior, si la condición es verdadera, pasa a la línea 18 en la cual se reemplaza el valor que sobrepasa el límite inferior por el promedio del valor y el límite inferior, de otro modo pasa a la línea 20 que evalúa si el valor en la matriz **devVI** supera el límite superior, si la condición es cierta pasa a la línea 21 en donde se establece el nuevo valor promediando el valor actual y el límite superior.

```

1 //Mutación
2 _global void mutacion(float *devVI, float *devPopold, float *devArchive, int *devR1, int *devR2, float *devFs,
3 int *devindexRand, int *devbestIndex, int psize, int d){
4     int idx = blockIdx.x * blockDim.x + threadIdx.x;
5     int idy = blockIdx.y * blockDim.y + threadIdx.y;
6     if(idy < psize && idx < d){
7         if(devR2[idy] >= psize){
8             devVI[idy * d + idx] = devPopold[idy * d + idx] + devFs[idy] *
9                 (devPopold[devbestIndex[devindexRand[idy]] * d + idx] - devPopold[idy * d + idx]) +
10                 devFs[idy] * (devPopold[devR1[idy] * d + idx] - devArchive[(devR2[idy] - psize) * d + idx]);
11         }else{
12             devVI[idy * d + idx] = devPopold[idy * d + idx] + devFs[idy] *
13                 (devPopold[devbestIndex[devindexRand[idy]] * d + idx] - devPopold[idy * d + idx]) +
14                 devFs[idy] * (devPopold[devR1[idy] * d + idx] - devPopold[devR2[idy] * d + idx]);
15         }
16         _syncthread();
17         if (devVI[idy * d + idx] < minBound){
18             devVI[idy * d + idx] = (float)(devPopold[idy * d + idx] + (minBound))/2;
19         } else {
20             if (devVI[idy * d + idx] > maxBound){
21                 devVI[idy * d + idx] = (float)(devPopold[idy * d + idx] + (maxBound))/2;
22             }
23         }
24     }
25 }

```

Figura 18 Fase de Mutación.

Paso 4.- Cruce: En la *Figura 19* se muestra el código de la etapa de cruce o recombinación; en la línea 2 se declara la función recibiendo como parámetros: la matriz **devUI** donde se guarda el resultado de combinar la matriz de mutación **devVI** y la matriz de la población **devPopold**; el vector **devCR** que alberga las probabilidad de cruce por individuo; la matriz **devjrand** que contiene números enteros aleatorios por cada característica de los individuos y la matriz

devrandomCR que guarda valores de punto flotante aleatorios por cada característica de la población; la línea 4 y 5 definen los índices (coordenadas x,y) dentro del grid, la línea 6 muestra la condición que evita que los índices sobrepasen las dimensiones de la matriz de la población; la condición en la línea 7 evalúa primero si el índice “x” en el *grid* es igual al valor de la matriz **devjrand**, es decir, $j = j_{rand}$; la segunda condición evalúa si la matriz **devrandomCR** contiene valores que sobrepasen o sean igual a la probabilidad de cruce definida por individuo, es decir, $rand_{i,j} [0,1] \leq C_r$; si alguna de estas dos condiciones se cumple entonces la matriz **devUI** es cargada con los valores de la matriz de mutación **devVI** como se muestra en la línea 8, de otro modo la matriz **devUI** se llena con la matriz de población **devPopold** como se presenta en la línea 10.

```

1 //Cruce
2 global void crossover(float *devUI, float *devVI, float *devPopold, float *devCR,
3 int *devjrand, float *devrandomCR, int psize, int d){
4     int idx = blockIdx.x * blockDim.x + threadIdx.x;
5     int idy = blockIdx.y * blockDim.y + threadIdx.y;
6     if (idy < psize && idx < d){
7         if (idx == devjrand[idy * d + idx] || devrandomCR[idy * d + idx] <= devCR[idy]){
8             devUI[idy * d + idx] = devVI[idy * d + idx];
9         }else{
10             devUI[idy * d + idx] = devPopold[idy * d + idx];
11         }
12     }
13 }

```

Figura 19 Fase de Cruce

Paso 5.- Selección: En la *Figura 20* se puede apreciar la implementación del proceso de selección; la línea 2 declara la función recibiendo como parámetros: la matriz de población **devPopold**; la matriz de cruce **devUI**; el vector **devI** el cual indica con un cero si el individuo de la población actual tiene una mejor aptitud que el nuevo individuo generado y un uno si el individuo resultante de las fases de mutación y cruce tiene una mejor aptitud que el individuo actual; la línea 3 y 4 definen los índices (coordenadas x,y) dentro del *grid*; la línea 5 muestra la condición que evita que los índices sobrepasen las dimensiones de la matriz de la población; en la línea 6 se evalúa si el vector **devI** contiene uno, si es cierto quiere decir que el nuevo individuo generado tiene una mejor aptitud, por lo tanto este individuo

reemplaza al anterior como se indica en la línea 7 insertando el individuo de la matriz ***devUI*** en la matriz ***devPopold***.

```
1 //Selección
2 __global__ void selection(float *devPopold, float *devUI, int *devI, int psize, int d){
3     int idx = blockIdx.x * blockDim.x + threadIdx.x;
4     int idy = blockIdx.y * blockDim.y + threadIdx.y;
5     if (idy < psize && idx < d){
6         if ( devI[idy] == 1){
7             devPopold[ idy * d + idx] = devUI[ idy * d + idx];
8         }
9     }
10 }
```

Figura 20 Fase de Selección

Capítulo 4

EVALUACIÓN DE LA VERSIÓN PARALELA DEL ALGORITMO JADE

En esta sección se presentan los resultados de la evaluación del rendimiento de la versión paralela del algoritmo JADE contra su versión secuencial. Para realizar la comparación entre versiones, se llevaron a cabo varios experimentos utilizando tres funciones de prueba: ***Sphere***, ***Rosenbrock*** y ***Rastrigin***; las cuales son ampliamente usadas en la literatura para probar algoritmos de optimización.

4.1 AMBIENTE DE LAS PRUEBAS

El ambiente de las pruebas fue establecido bajo las siguientes condiciones: Se utilizó una computadora portátil con procesador Intel core i7, 6 GB de memoria RAM, disco duro de 1 TB, sistema operativo Windows 7 Professional. Una tarjeta gráfica GeForce GT 650M con 384 núcleos CUDA, reloj de procesador 900 MHz.

Las funciones de prueba utilizadas durante la evaluación de los algoritmos fueron extraídas del CEC 2005 (*Congress on Evolutionary Computation*) [26] y son listadas a continuación: ***Sphere*** una función simple bien conocida con un solo mínimo global, ***Rosenbrock*** o función Banana posee un valle que desciende

lentamente por origen de la función lo que hace difícil encontrar su mínimo global y **Rastrigin** esta función tiene un mínimo global bastante difícil de encontrar, debido a que la función contiene muchos mínimos locales:

La configuración utilizada para el algoritmo JADE en su versión paralela y secuencial fue la siguiente:

Factor de mutación (F) = 0.5

Probabilidad de cruza (Cr) = 0.5

Generaciones (G) = 200

En cuanto al tamaño de población utilizado durante la experimentación, se utilizaron cinco tamaños de poblaciones **30 D x 100 NP**, **100 D x 100 NP**, **100 D x 1,000 NP**, **100 D x 10,000 NP** y **100 D x 20,000 NP**. Estos cinco tamaños de población fueron definidos de acuerdo a las pruebas realizadas en el estado del arte [24]. Cada una de las cinco configuraciones fueron ejecutados en la versión paralela y secuencial; y cada versión fue ejecutada con las tres funciones de evaluación mencionadas previamente.

Es importante mencionar que todos los experimentos fueron ejecutados 30 veces, de acuerdo al teorema del límite central, el cual establece que un tamaño de muestra de 30 es suficientemente grande para proveer estimaciones estadísticas confiables [31]. Se realizaron 30 experimentos ejecutados 30 veces cada uno, dando un total 900 de pruebas.

4.2 EVALUACIÓN DEL DESEMPEÑO DEL ALGORITMO JADE EN SU VERSIÓN PARALELA

En esta sección se muestra una comparativa entre el desempeño del algoritmo JADE secuencial y la versión paralela propuesta en este trabajo. Los resultados son expresados, en términos de tiempos de ejecución (segundos) y factor de aceleración conocido como “**Speed-up**”.

El “**Speed-up**” [32] representa la razón de crecimiento del rendimiento entre el tiempo ejecución secuencial y el tiempo de ejecución paralelo. La expresión

$$S = \frac{\text{Tiempo CPU}}{\text{Tiempo GPU}}$$

corresponde al cálculo del “**Speed-up**”, donde S representa el “**Speed-up**”, mientras que *Tiempo CPU* y *Tiempo GPU* corresponden a los tiempos ejecución de acuerdo al hardware donde se lleva a cabo el experimento. Si el factor de aceleración obtenido es menor o igual a uno ($S \leq 1$), significa que no se obtiene beneficio de la versión paralela.

En las Secciones 4.2.1 – 4.2.3 se reportan los tiempos de ejecución del algoritmo JADE paralelizado contra su versión secuencial, dichas ejecuciones se llevaron a cabo en entorno mencionado en la Sección 4.1.

4.2.1 Evaluación de JADE secuencial contra paralelo utilizando la función **Sphere**

En la tabla Tabla 6 se presentan los resultados de la experimentación utilizando la función de evaluación “**Sphere**”, en donde la primera columna muestra el tamaño de la población usada en el experimento, en la columna **Tiempo CPU** y **Tiempo GPU** se muestran los tiempos promedios de 30 ejecuciones realizados en sus respectivas versiones, la columna “**Speed-up**” presenta la aceleración que presentó la versión paralela contra la secuencial. A manera de ejemplo podemos interpretar la penúltima fila de la tabla de la siguiente manera, para una población de **100 D x 10,000 NP** la versión secuencial del algoritmo tardó 542.262100s, mientras que la versión paralela tardó 8.6684s, por lo tanto la versión paralela es 62.5561 veces más rápida. Cabe mencionar que se realizaron un total de 300 pruebas utilizando la función de prueba “**Spehre**” producto de ejecutar 30 veces los 10 experimentos presentados en la Tabla 6.

Tabla 6 Comparación Tiempo CPU vs Tiempo GPU usando la función Sphere.

D x NP	Tiempo CPU (s)	Tiempo GPU (s)	Speed-up (S)
30 x 100	0.079067	0.2366	0.3341
100 x 100	0.197600	0.3973	0.4973
100 x 1,000	1.634867	13.5976	0.6277
100 x 10,000	542.2621	8.6684	62.5561
100 x 20,000	1866.1677	50.5901	36.8879

En la Figura 21 se presenta la información reportada en la Tabla 6, en donde el eje “x” representa los diferentes tamaños de población utilizados en la experimentación, el eje “y” presenta los tiempos promedio de latencia en los experimentos realizados, debido particularmente a que el experimento con la población **100 D x 20,000 NP** arroja un tiempo muy elevado en el CPU (1866.1677s) fue necesario escalar el eje “y” de la gráfica, de manera que se puedan visualizar valores pequeños en los primeros experimentos y un valor elevado en el último experimento, como se observa en el eje “y” de la Figura 21 el tiempo de los experimentos abarca desde 0.01s hasta 1000s; las barras color azul corresponden a los tiempos de ejecución de la versión paralela y las barras grises corresponden a los tiempos de ejecución de la versión secuencial.

Es destacable que en el experimento **100 D x 10,000 NP** usando la función de prueba “**Sphere**” la versión paralela del algoritmo JADE es 62 veces más veloz, como puede observar en la Figura 21, para interpretar la gráfica es necesario tomar en cuenta el escalamiento del eje “y”.

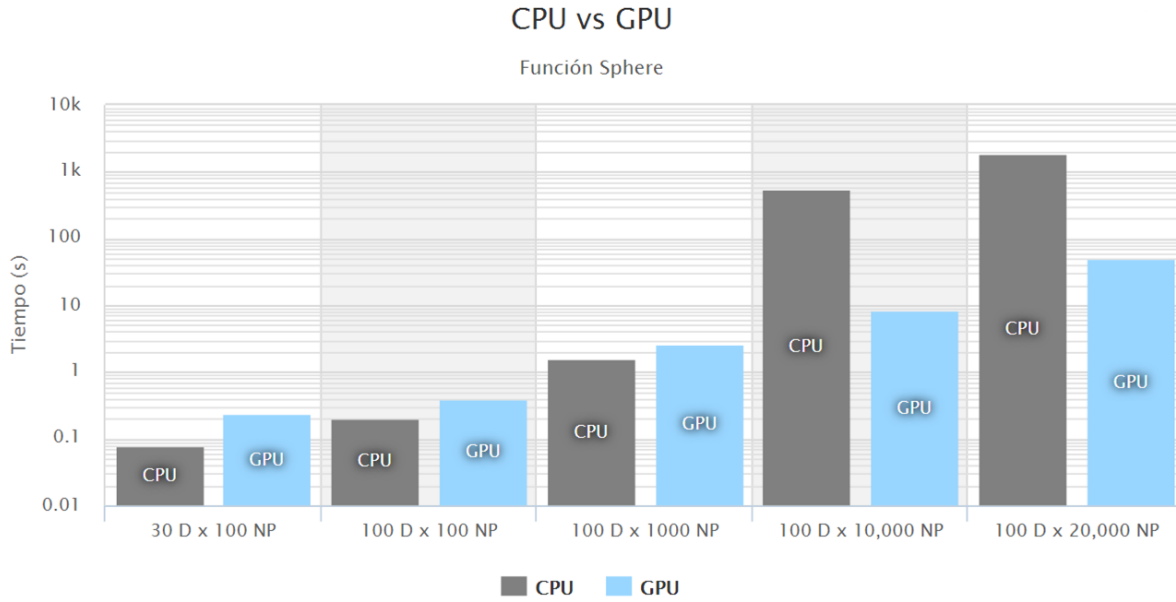


Figura 21 Comparación Tiempo CPU vs Tiempo GPU usando la función Sphere.

4.2.2 Evaluación de JADE secuencial contra paralelo utilizando la función *Rosenbrock*

La Tabla 7 muestra el desempeño del algoritmo JADE paralelizado y su versión secuencial usando la función de prueba “*Rosenbrock*”. Como se observa en la Tabla 7 los experimentos utilizando poblaciones de **30 D x 100 NP** y **100 D x 100 NP** arrojaron tiempos aproximados del CPU y GPU, por lo tanto no aumenta el rendimiento en la versión paralela en dichos experimentos; en el experimento **100 D x 1,000 NP** se aprecia un ligero aumento en el “*Speed-up*” y en el experimento **100 D x 10,000 NP** el tiempo del CPU se eleva considerablemente dando como resultado que la versión paralela es 19.4537 veces más rápida que la versión secuencial en dicho experimento mientras que en el experimento **100 D x 20,000 NP** el desempeño es 6.02 veces más rápido que el secuencial. Bajo la función de prueba “*Rosenbrock*” se realizaron 300 pruebas resultado de ejecutar 30 veces los 10 experimentos mostrados en la Tabla 7.

Tabla 7 Comparación Tiempo CPU vs Tiempo GPU usando la función Rosenbrock.

D x NP	Tiempo CPU (s)	Tiempo GPU (s)	Speed-up (S)
--------	----------------	----------------	--------------

30 x 100	0.373866	0.3645	1.0256
100 x 100	0.700433	0.6916	1.0127
100 x 1,000	7.325766	4.7965	1.5273
100 x 10,000	177.060633	9.101633	19.4537
100 x 20,000	332.645633	55.180166	6.0283

La Figura 22 muestra de manera gráfica los tiempos de ejecución del algoritmo JADE usando la función de evaluación **Rosenbrock** correspondiente a la Tabla 7. Se puede observar que en este problema con una población de **30 D x 100 NP** y **100 D x 100 NP** el rendimiento de las dos versiones es aproximadamente equivalente, para una población de **100 D x 1000 NP** se aprecia una ligera aceleración, en la configuración **100 D x 10,000 NP** se observa que la versión paralela es 19.45 veces más rápida que su versión secuencial y para una población **100 D x 20,000 NP** es 6.02 veces más rápida la versión paralela sobre la versión secuencial, para analizar correctamente la gráfica se debe considerar el escalamiento del eje “y”.

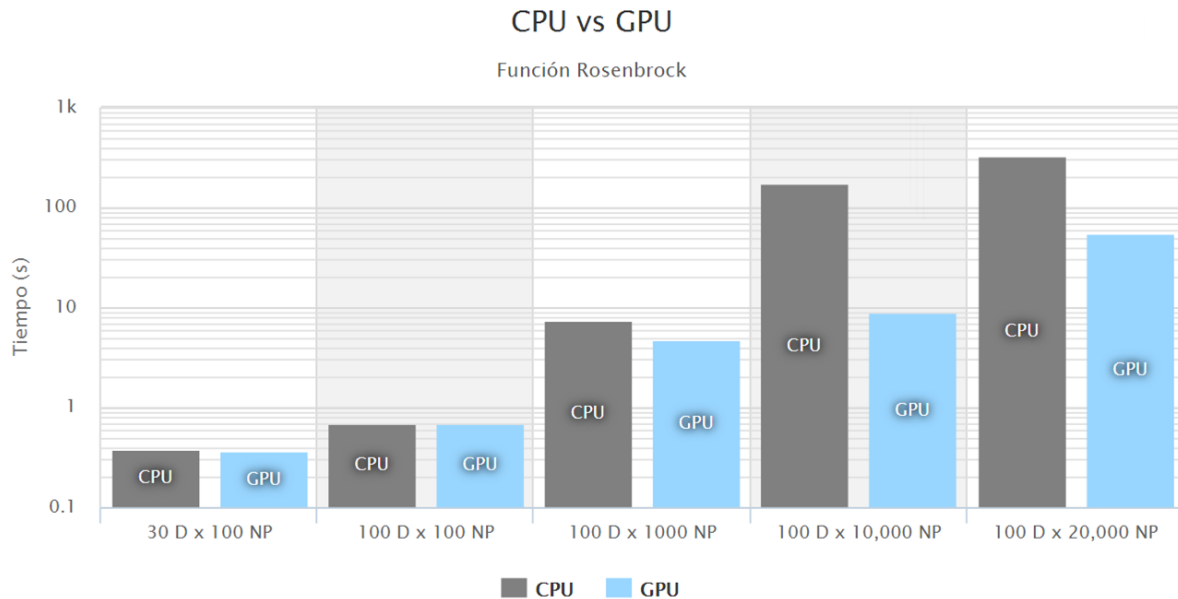


Figura 22 Comparación Tiempo CPU vs Tiempo GPU usando la función Rosenbrock.

4.2.3 Evaluación de JADE secuencial contra paralelo utilizando la función *Rastrigin*

La Tabla 8 expone los tiempos de ejecución del algoritmo JADE en su versión paralela contra su versión secuencial utilizando la función de prueba *Rastrigin*. En la experimentación realizada con los tamaños de población de **30 D x 100 NP** y **100 D x 100** el tiempo de la versión secuencial es menor al de la versión paralela, por esta razón el “*Speed-up*” no sobrepasa de uno; en el experimento **100 D x 1,000 NP** los tiempos del CPU y GPU son cercanos; se puede observar que el tiempo de la versión paralela superar al tiempo de la versión secuencial en los experimentos con una población de **100 D x 10,000 NP**, siendo 37.43 veces más rápida la versión paralela y una población de **100 D x 20,000 NP**, siendo 32.65 veces más rápida la versión paralela. Cabe señalar que se realizaron 300 pruebas usando la función de prueba “*Rastrigin*”, resultado de los 10 experimentos presentados en la Tabla 8 fue ejecutado 30 veces.

Tabla 8 Comparación Tiempo CPU vs Tiempo GPU usando la función *Rastrigin*.

D x NP	Tiempo CPU (s)	Tiempo GPU (s)	<i>Speed-up</i> (S)
30 x 100	0.1175	0.2761	0.4255
100 x 100	0.2464	0.5756	0.4281
100 x 1,000	5.0596	3.8933	1.2995
100 x 10,000	329.2183	8.7938	37.4372
100 x 20,000	1960.0433	60.0194	32.6568

En la Figura 23 se muestran los tiempos de ejecución de la experimentación del algoritmo JADE usando la función de evaluación *Rastrigin*, el eje “x” representa las dimensiones de la población, el eje “y” simboliza el tiempo promedio de ejecución del algoritmo, los tiempos de la versión paralela se visualizan en las barras color azul, mientras el tiempo de la versión secuencial en las barras color gris; cabe

señalar que el eje “*y*” (Tiempo) de la gráfica está escalado para permitir representar los tiempos reducidos y extensos de la experimentación.

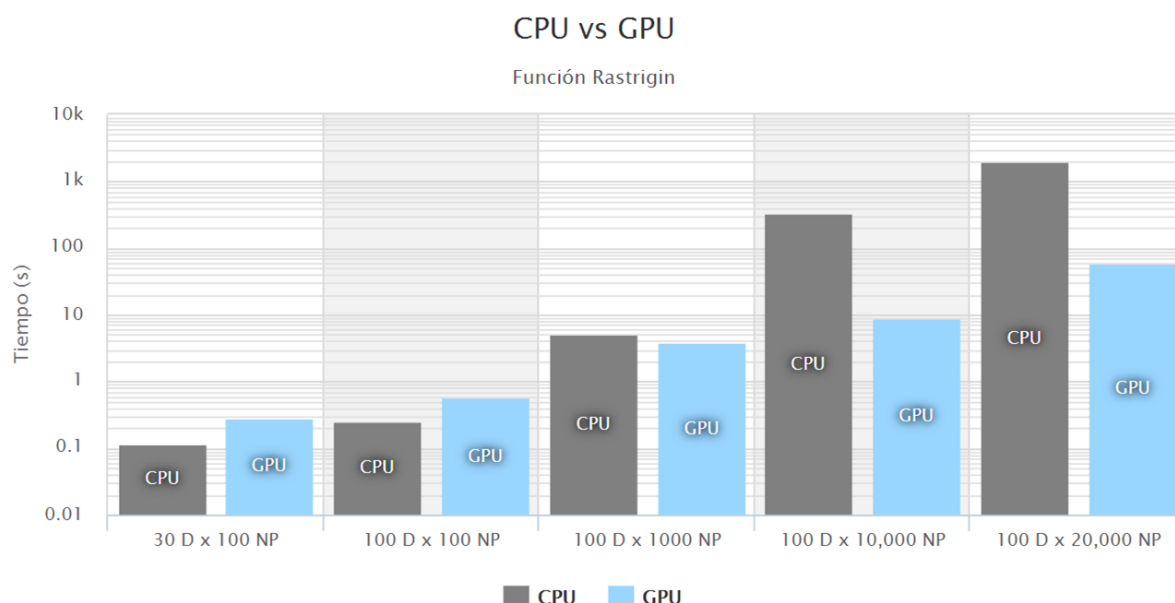


Figura 23 Comparación Tiempo CPU vs Tiempo GPU usando la función Rastrigin.

De manera general se observa que el comportamiento de *S* (“*Speed-up*”) aumenta cuando se incrementa la cantidad de dimensiones (*D*) y el número de individuos (*NP*).

4.3 RENDIMIENTO JADE SECUENCIAL CONTRA JADE PARALELO

En esta sección se presenta el comportamiento del rendimiento del algoritmo JADE en su versión paralela a través de los diferentes tamaños de población usados durante la experimentación. En la Figura 24 se muestra de manera gráfica el comportamiento del “*Speed-up*” a lo largo de las configuraciones de los experimentos; en donde el eje “*y*” presenta la razón en que la versión paralela del algoritmo JADE es más veloz a su versión secuencial (razón “*Speed-up*”), por motivos de estética fue necesario escalar el eje “*y*” de la gráfica, de tal forma que dicho eje abarca desde .01 hasta 100; el eje “*x*” muestra las configuraciones del

tamaño de la población, la línea horizontal roja indica si el valor “**Speed-up**” está por encima de uno, si esta línea no es rebasada, no existe un beneficio de la versión paralela; la línea azul de la Figura 24 representa el “**Speed-up**” de la experimentación realizada con la función **Sphere**, en donde se observa ganancia en las configuraciones **100 D x 10,000 NP** y **100 D x 20,000 NP**. La línea negra corresponde al valor “**Speed-up**” usando la función **Rosenbrock**, en dicha experimentación el “**Speed-up**” es aproximadamente equivalente a la versión secuencial en las configuraciones **30 D x 100 NP** y **100 D x 100 NP** y a partir de la configuración **100 D x 1,000 NP** se obtiene un rendimiento considerable de la versión paralela. La línea verde corresponde a la experimentación usando la función **Rastrigin** en la cual para las configuraciones **30 D x 100 NP** y **100 D x 100 NP** no se obtiene una mejora notable de la versión paralela es a partir de la configuración **100 D x 1,000 NP** que se observa una mejora en el rendimiento de la versión paralela.

Como se observa en la Figura 24 el rendimiento de la versión paralela incrementa a medida que se aumenta la cantidad de datos que deben ser procesados. Además, los valores de “**Speed-up**” varían de acuerdo a la función de prueba usada. Debido a que la sobrecarga que existe al hacer una llamada a una función *kernel*, la versión paralela no presenta una ventaja sobre la versión secuencial al procesar cantidades de datos pequeñas [33].

Es destacable el rendimiento del algoritmo JADE paralelizado propuesto en este trabajo, en poblaciones grandes. De acuerdo a [34] es ventajoso tener una población grande debido a que existe una relación positiva entre el tamaño de la población y la variación de la población. Si el tamaño de la población es pequeña, el algoritmo tiene una probabilidad de converger prematuramente, es decir, la variación de la población ocurre abruptamente y se alcanzan mínimos locales; o tener un estancamiento, esto ocurre cuando la población se mantiene diversa, pero no converge por lo tanto el proceso de optimización no progresa. En la mayoría de los casos, las poblaciones grandes son más estables.

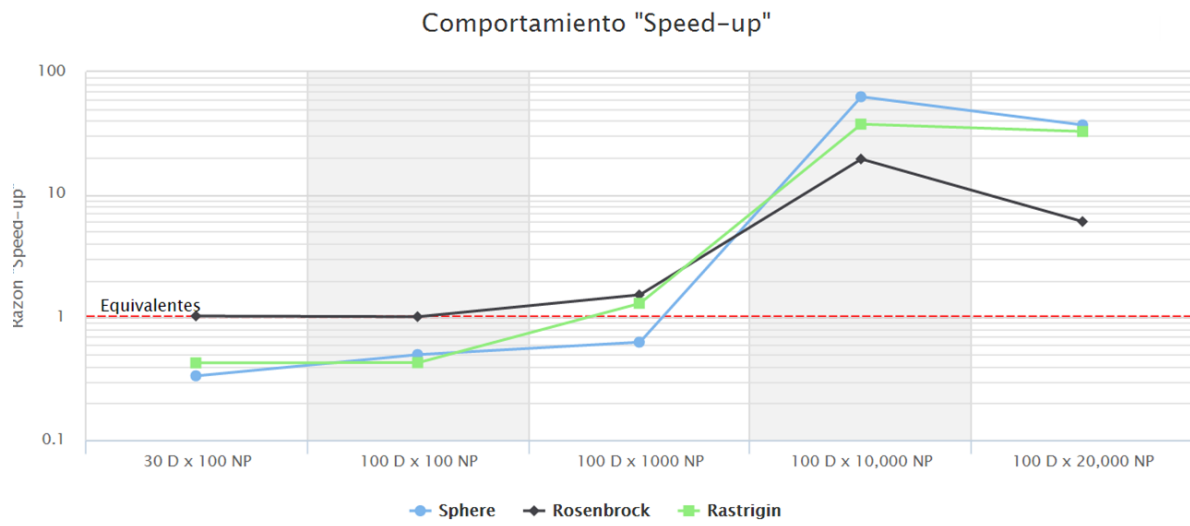


Figura 24 Comportamiento del "Speed-up" a través del aumento de población.

Capítulo 5

CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se presentan las conclusiones obtenidas al finalizar este trabajo de tesis. En este trabajo se propuso la implementación en paralelo del algoritmo de evolución diferencial JADE, con el objetivo de mejorar los tiempos de ejecución del algoritmo mediante una plataforma GPU. Se presentan posibles mejoras a la implementación del algoritmo JADE en su versión paralela, dichas mejoras fueron detectadas durante el desarrollo de esta investigación.

5.1 CONCLUSIONES

La arquitectura CUDA de las tarjetas NVIDIA permite resolver eficientemente problemas que pueden ser expresados bajo el esquema “una instrucción, múltiples datos” (*SIMD, Single Instruction, Multiple Data*). El algoritmo de evolución diferencial se adapta al esquema SIMD, ya que las etapas de dicho algoritmo (mutación, cruza y selección) ejecutan la misma instrucción sobre un conjunto de datos (población).

En particular, la versión en paralelo del algoritmo JADE, propuesta en este trabajo, mostró un desempeño que superó ampliamente, en términos de tiempos de ejecución, a la versión secuencial del algoritmo JADE, dando como resultado el

aumento del factor de aceleración “**Speed-up**”. La evaluación del desempeño de la versión paralela del algoritmo JADE se llevó a cabo de la siguiente manera, se utilizaron tres funciones de prueba **Sphere**, **Rosenbrock** y **Rastrigin** y cinco tamaños de poblaciones **30 D x 100 NP**, **100 D x 100 NP**, **100 D x 1,000 NP**, **100 D x 10,000 NP** y **100 D x 20,000 NP**, debido a la características de aleatoriedad de los algoritmos, cada uno (secuencial y paralelo) fue ejecutado 30 veces, utilizando cada uno de los tamaños de las poblaciones previamente mencionados.

Los resultados muestran que la versión paralela del algoritmo JADE alcanza los mejores tiempos de cómputo y los mejores valores para el factor de aceleración al utilizar la configuración **100 D x 10,000 NP**. Para la función de prueba **Sphere** se observó que la versión paralela fue 62.5 veces más rápida que la versión secuencial; al evaluar la función **Rosenbrock** la versión paralela resultó 19.4 veces más rápida en comparación con la secuencia y finalmente, al evaluar la función **Rastrigin** se obtuvo una aceleración de 37.4 en relación a la versión secuencial.

Se puede concluir que el rendimiento de la versión paralela aumenta en relación a la cantidad de datos procesados, mientras mayor sea el tamaño de las dimensiones (D) y el tamaño de la población (NP) se obtendrá un beneficio considerable de la versión paralela.

5.2 TRABAJOS FUTUROS

Para mejorar el rendimiento de la implementación del algoritmo JADE paralelizado en CUDA y darle seguimiento a este trabajo, se plantean las siguientes mejoras:

Utilizar la librería CURAND de CUDA para generar los números aleatorios necesarios en las operaciones del algoritmo JADE, la finalidad es aprovechar el rendimiento del GPU y evitar la sobrecarga por copiar memoria del host al dispositivo y viceversa.

Hacer uso de los *streams* de CUDA para solapar los tiempos de ejecución de las funciones *kernels* con los tiempos de transferencia de memoria, esto resultaría en la mejora de la eficiencia del uso del GPU en la implementación del algoritmo JADE.

5.3 LISTA DE PUBLICACIONES

Durante el desarrollo de esta tesis se publicaron resultados preliminares en los siguientes artículos:

- Mexicano, P. Bocanegra, S. Cervantes, R. Rodríguez, “C-Jade: Una versión Paralela del algoritmo Jade”, *Congreso Universitario 2014, Universidad Tecnológica de Cd. Juárez, México*, pp. 212-217.
- También, se realizó la ponencia “*Implementación en paralelo del algoritmo JADE*” en el 4° Foro Universitario “Investigación, Sociedad y Desarrollo. Avances y Perspectivas”, en el marco de la 3ERA Edición Expociencia e Innovación Tamaulipas. Cd. Victoria, Tam, el 8 de abril de 2014.

REFERENCIAS

- [1] K. Lee y M. El-Sharkawi, *Modern Heuristic Optimization Techniques: Theory and Applications to Power Systems*, Wiley-IEEE Press, 2008.
- [2] J. Sanchez, M. Galán y E. Rubio, "Genetic Algorithms and Cellular Automata: A New Architecture for Traffic Light Cycles," *Congress on Evolutionary Computation CEC2004*, Portland, OR, USA, 2004, vol. 2, pp. 1668-1674.
- [3] D. Regulwar, S. Choudhari y P. Raj, "Differential Evolution Algorithm with Application to Optimal Operation of Multipurpose Reservoir," *Journal of Water Resource and Protection*, vol. 2, no 6, pp. 560-568, 2010.
- [4] I. Rivera y M. Vargas-Lombardo, "Principios y Campos de Aplicación en CUDA Programación paralela y sus potencialidades," *Nexo Revista Científica*, vol. 25, no. 02, pp. 39-46, 2012.
- [5] J. Zhang y A. C. Sanderson, "JADE: adaptive differential evolution with optional external archive," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 5, pp. 945–958, 2009.
- [6] L. d. P. Veronese y R. A. Krohling, "Differential Evolution Algorithm on the GPU with C-CUDA," *IEEE Congress on Evolutionary Computation CEC 2010*, Barcelona, España, 2010, pp. 1-7.
- [7] P. Krömer, J. Platos, V. Snásel y A. Abraham, *Many-threaded Differential Evolution on the GPU*, New York, NY, USA: Natalio Krasnogor, 2011.
- [8] A. K. Qin, F. Raimondo, F. Forbes y Y. S. Ong, *Proceedings of the 14th annual conference on Genetic and evolutionary computation (GECCO'12)*, Philadelphia, USA, pp. 991-998, 2012.
- [9] D. Davendra, J. Gaura, M. Bialic-Davendra y R. Senkerik, "CUDA Based Enhanced Differential Evolution: A Computational Analysis," *Proceedings 26th European Conference on Modelling and Simulation ECMS 2012*, Koblenz, Alemania, pp. 399-404, 2012.

- [10] C. C. R. Jose Ignacio Hidalgo, "Una revisión de los algoritmos evolutivos y sus aplicaciones," *CES Felipe II*, ISSN 1695-8543, no. 2, 2004.
- [11] J. H. Holland, *Adaptation in Natural and Artificial Systems*, Cambridge, MA, USA: MIT Press, 1975.
- [12] N. E. A. Khalid, N. M. Ariff, S. Yahya y N. M. Noor, "A Review of Bio-inspired Algorithms as Image Processing Techniques," *Communications in Computer and Information Science*, vol. 179, pp. 660-673, 2011.
- [13] K. V. Price, R. M. Storn y J. A. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization*, New York, USA: Springer-Verlag, 2005.
- [14] E. A. S. G.-C. Javier Arellano-Verdejo, "Efficiently finding the optimum number of clusters in a dataset with a new hybrid differential evolution algorithm: DELA," *Soft Computing*, vol. 20, no. 3, pp. 895-905, 2016.
- [15] J. T. Pert Bujok, "New variants of adaptive differential evolution algorithm with competing strategies," *Acta Electrotechnica et Informatica*, vol. 15, no. 2, pp. 49-56, 2015.
- [16] A. C. Sanderson y J. Zhang, *Adaptive Differential Evolution: A Robust Approach to Multimodal Problem Optimization*, NY: Springer Publishing Company, Incorporated, 2009.
- [17] C. O. F. Jiménez, "A GPU-based parallel Object kinetic Monte Carlo algorithm for the evolution of defects in irradiated materials," *Computational Materials Science*, vol. 113, no. 4, pp. 178-186, 2016.
- [18] NVIDIA CUDA C Programming Guide (Junio 2011). NVIDIA Corporation [En línea]. Disponible: <http://docs.nvidia.com/cuda/cuda-c-programming-guide>.
- [19] P. Werstein, M. Pethick y Z. Huang, "A performance comparison of DSM, PVM, and MPI," *IEEE Parallel and Distributed Computing, Applications and Technologies (PDCAT'2003)*, Chengdu, China, pp. 476 - 482, 2003.
- [20] A. B. A. D. J. J. W. M. R. a. S. V. Geist, *Pvm 3 User's Guide And Reference Manual*, U.S. Department of Energy, 1994.

- [21] P. S. Pacheco, *Parallel Programming with MPI*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.
- [22] S. Cook, *CUDA programming A developer's guide to parallel computing with GPUs*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.
- [23] J. Sanders y E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, 2010.
- [24] C.-Y. Lee y X. Yao, "Evolutionary programming using mutations based on the Levy probability distribution," *IEEE Trans. Evolutionary Computation*, vol. 8, no 1, pp. 1-13, 2004.
- [25] S. J. Domínguez González y N. García Barriga, "Fully Parallel Differential Evolution," *Genetic and Evolutionary Computation Conference (GECCO'2011)*, Dublin, Ireland, 2011.
- [26] P. N. Suganthan, N. Hansen, J. Liang, K. Deb, Y. -P. Chen, A. Auger y S. Tiwari, "Problem Definitions and Evaluation Criteria for the CEC 2005 Special Session on Real-Parameter Optimization," Nanyang Technological University, Singapore, 2005.
- [27] D. Davendra y G. Onwubolu, "Flow shop scheduling using enhanced differential evolution," *European Conference on Modeling and Simulation*, vol. 50, no. 10, República Checa, pp. 2757-2773, 2007.
- [28] X. Yao, Y. Liu y G. Lin, "Evolutionary programming made faster," *IEEE Trans. Evolutionary Computation*, vol. 3, no. 2, pp. 82-102, 1999.
- [29] X. Yao, Y. Liu, K.-H. Liang y G. Lin, "Fast Evolutionary Algorithms," *Advances in Evolutionary Computing*, New York, NY, USA, Springer-Verlag, 2003, pp. 45-95.
- [30] V. Picheny, T. Wagner y D. Ginsbourger, "A benchmark of kriging-based infill criteria for noisy optimization," *Structural and Multidisciplinary Optimization*, vol. 48, no. No. 3, pp. pp 607-626, 2013.
- [31] M. L. Samuels, J. A. Witmer y A. A. Schaffner, *Statistics for the Life Sciences* (4th Edition), Boston, MA: Prentice Hall, 2012.

- [32] D. Conte, R. D'ambrosio y B. Paternoster, "GPU-acceleration of waveform relaxation methods for large differential systems," *Numerical Algorithms*, vol. 71, no. 2, pp. 293-310, 2016.
- [33] LAVA Lab CUDA Support (Mayo 2016). T. U. o. Virginia [En línea]. Disponible: https://www.cs.virginia.edu/~mwb7w/cuda_support/kernel_overhead.html.
- [34] T. Hu, S. Harding y W. Banzhaf, "Variable population size and evolution acceleration: a case study with a parallel evolutionary algorithm," *Genetic Programming and Evolvable Machines*, vol. 11, no 2, pp 205-225, 2010.