IMPLEMENTACIÓN DE LA METAHEURISTICA *PARTICLE SWARM OPTIMIZATION* (PSO) EN DIFERENTES MODELOS DE PROGRAMACIÓN (C, CUDA-C Y PTHREADS)

Obregon Ivan[1]
Universidad Industrial Santander
Cra 27 Calle 9 Ciudad Universitaria
Versión 1, 9 de Enero de 2016

Resumen

Los algoritmos de optimización en los sistemas lineales ha sido una dificultad hablando en términos de tiempos de procesado ya que este es un problema iterativo lineal. Las metaheurísticas tales como PSO (*Particle Swarm Optimization*) son una herramienta clave para la reducción de tiempos de cómputo y con una fiabilidad buena a la hora de dar sus resultados. Una mejora que se puede adaptar a esta metaheurística es el enfoque en la forma de la implementación. Existen varios modelos de programación tales como la serial tradicional en código C, o sus formas paralelas usando su librería Pthreads y cómputo externo en GPU CUDA-C. Se obtuvo un factor aceleración hasta de 44.16 veces usando la implementación en paralelo CUDA-C.

1. INTRODUCCIÓN

Los problemas de optimización global en sistemas no lineales pueden ser divididos en varias categorías dependiendo del campo en que se esté trabajando, como puede ser el continuo o el discreto, uni ó multi-objetivo, estático o dinámico etc [1], haciendo el tiempo de procesamiento para la solución de estos sea muy alto y su viabilidad se disminuya.

[1] ivan.fe639@gmail.com

La metaheurística PSO se caracteriza por resolver de manera aproximada los problemas ya antes mencionados, y además poseen la característica de poder ser aplicables en distintos problemas sin ninguna adaptación compleja [1]. Estas metaheurísticas son inspiradas en el comportamiento de la naturaleza, ya sea en su física, biología o comportamiento social.

Algunos modelos de programación secuenciales que son más utilizados para la realización de algoritmos de computación son C, C++, Fortran, Matlab, etc. El uso de C por encima de las demás opciones trae buenas ventajas ya que al ser un lenguaje de bajo-medio nivel se puede utilizar y/o realizar herramientas que aporte una ventaja valiosa en términos de tiempo de cómputo. El aporte de desarrolladoras de hardware para el cómputo en paralelo tal como lo ha hecho NVIDIA, ha producido una migración de los códigos secuenciales a los paralelos, ofreciendo una mayor mejora en los tiempos de procesado [2].

El documento está distribuido en secciones donde se hablará acerca de la metaheurística, seguido de los modelos de programación usados. Un aparte para los resultados y finalmente las conclusiones obtenidas.

2. PARTICLE SWARM OPTIMIZATION (PSO)

2.1. Orígenes Naturales

La metaheurística *PSO* fue inspirada en el fenómeno natural descrito por las aves al volar en bandada o grupo, describiendo una acción tanto individual como global haciendo que este fenómeno natural sea vistoso y ordenado. De estas premisas varios investigadores se enfocaron en obtener las reglas de comportamiento que cada ave debe cumplir para que el fenómeno se dé [3]. A partir de estas reglas se comenzó la aplicación en diferentes problemas simulando el mismo fenómeno pero enfocado a dar respuesta a problemas específicos.

Figura 1 Bandada de aves volando en el cielo.

2.2. Algoritmo

El algoritmo consiste en una implementación iterativa estocástica donde las partículas ("aves") se distribuyen de manera aleatoria sobre un rango y dominio acotados o también llamado espacio de búsqueda, y que poseerán un valor de imagen en aquella posición, el cual será dada por una función costo (función a minimizar) y que producirá un cambio de posición teniendo en cuenta un criterio de posición local y uno de posición global, haciendo que el comportamiento final sea únicamente global y dando como resultado el valor óptimo de la función costo.

El punto de partida del algoritmo es la parametrización inicial que consiste en la creación de una población aleatoria dentro de un rango acotado el cual será definido por el tipo de problema de optimización que se esté realizando. Esta población será constituida por las posiciones iniciales para las partículas. Después se obtendrán las imágenes de esta población de manera individual usando la función costo y se calculará su mínimo para así obtener el criterio inicial global. Una vez realizado la parametrización inicial se procede al proceso iterativo, definido previamente, donde se aplicaran las reglas de la metaheurística.

Las reglas que se aplican son unas series de comparaciones de cada partícula de la población con respecto a su componente local (mejor anterior), su valor modificado (actual) y su componente global (mejor global). El valor modificado será calculado usando la ecuación 1 [4], donde *w*, *c1*,

c2 son los pesos que se darán a cada termino y *r1* y *r2* son las componentes de pesos aleatorios que oscila de 0 a 1.

$$\boldsymbol{vx[i] = w * vx[i-1] + c1r1(bestLocal - Presente) + c2r2(bestGlobal - Presente)} \qquad (1)$$

A partir de la ecuación 1 se calcula el valor $\boldsymbol{vx}$ el cual será el valor del delta que afectará a las posiciones de cada partícula regida por la ecuación 2. Después se realiza una comparación de este valor de posición nueva con las fronteras acotadas del espacio de búsqueda, para conocer si está adentro de los límites acotados por el problema y así mantener estable la dinámica de las partículas. Finalmente se evalúa la posición nueva con la función costo y así obtener su imagen la cual será comparada con su mejor local y posteriormente con su global, en cualquier de las dos comparaciones si se obtuvo una mejora se sobrescribirá el valor de posición.

El ciclo se replica para cada partícula de la población y a su vez para el número de iteraciones produciendo que todas las partículas tiendan a converger a un mismo punto el cual será el resultado de la optimización, en la figura 2 se puede observar el diagrama de funcionamiento del algoritmo.

$$\boldsymbol{posParticulaNueva = posParticulaAnterior + vx} \qquad (2)$$

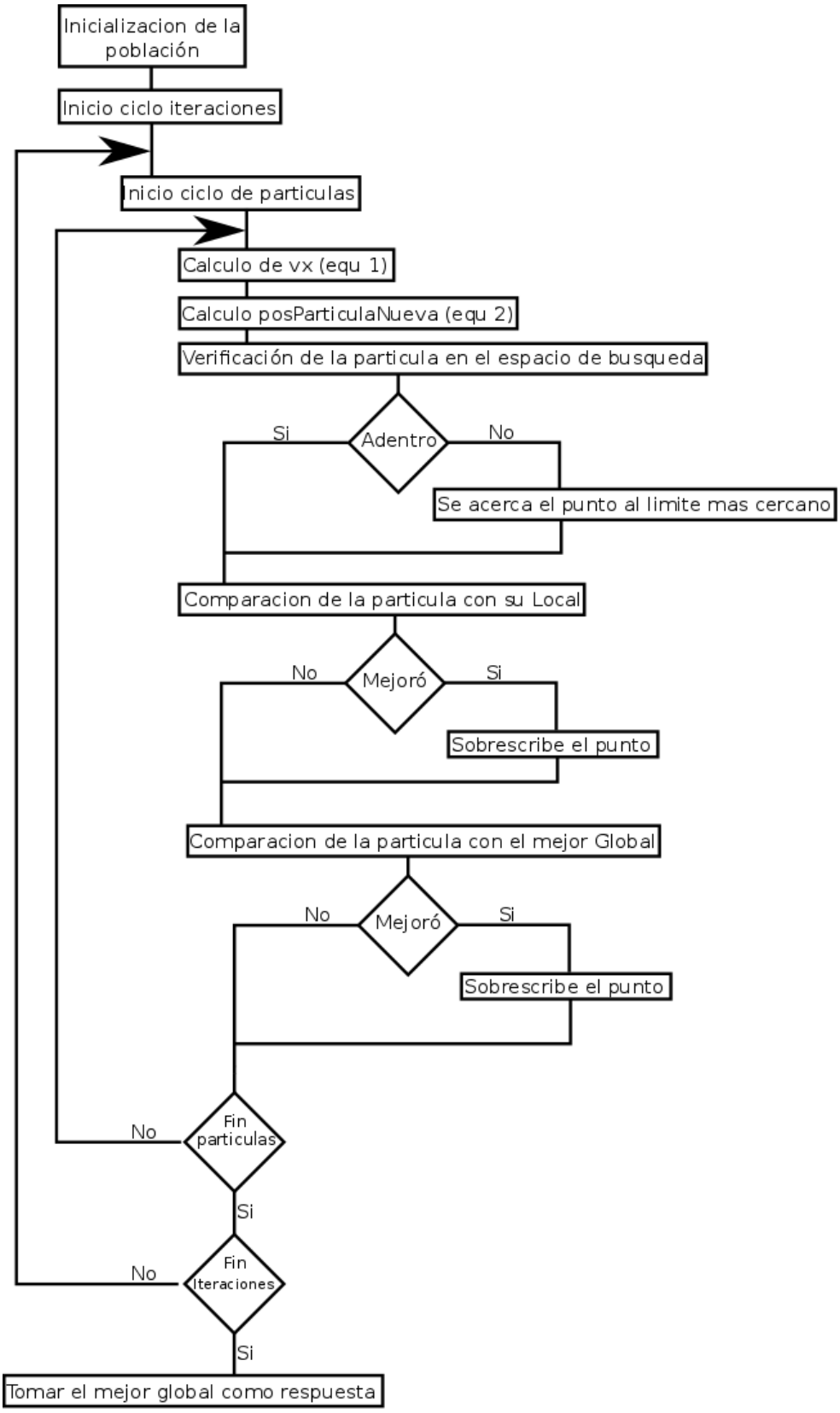


Figura 2 Diagrama de flujo del algoritmo PSO.

3. MODELOS DE PROGRAMACIÓN

Para la implementación del algoritmo se optó por utilizar tres modelos de programación: ANSI C, C-Pthreads la cual usa una librería de cómputo paralelo de núcleos CPU y CUDA-C. Todos los modelos usados son basados en C para obtener una comparación más concluyente acerca de la eficiencia de cada uno. Además estos códigos fueron realizados usando el compilador GCC y NVCC para los códigos en C y CUDA respectivamente, ambos incorporados en el *kernel*[2] de Linux.

3.1. Implementación Usando lenguaje C

El lenguaje de programación C fue creado en la década de los años 70 por Dennis Ritchie y Brian Kernighan y fue implementado por primera vez en un sistema operativo UNIX [5]. Una de sus características es que es un lenguaje estructurado sin subdivisiones de bloques, esto quiere decir que no es posible crear subrutinas dentro de otras subrutinas [5]. Además este lenguaje C es uno de los más utilizados por programadores para realizar todo tipos de programas [6] ya que posee las ventajas de ser flexible, portable y sencillo [6]

La estrategia que se realizó para este lenguaje fue la de programación serial, usando cabeceros (*headers*) y llamadas a funciones. El algoritmo está basado en el diagrama de la figura 2.

3.2. Implementación usando C-Pthreads

La biblioteca Pthreads creada para el lenguaje C permite trabajar con distintos hilos de ejecución llamados *threads* al mismo tiempo cumpliendo con el estándar POSIX[3]. Su principal característica es la compartir su memoria entre sí (ver figura 3) a diferencia de un proceso común el cual no lo hace [7].

La estrategia de programación que se utilizó fue el de dividir la carga computacional de las partículas en partes iguales al número de *threads* implementados, produciendo un tiempo de

[2]Un kernel es el componente central de un sistema operativo GNU/Linux y es el encargado de manejar los recursos hardware del computador.
[3] Acrónimo de *Portable Operating System Interface* del sistema operativo Linux.

cálculo menor al realizar el bucle de partículas teóricamente hablando. El diagrama de esta implementación puede verse en la figura 4.

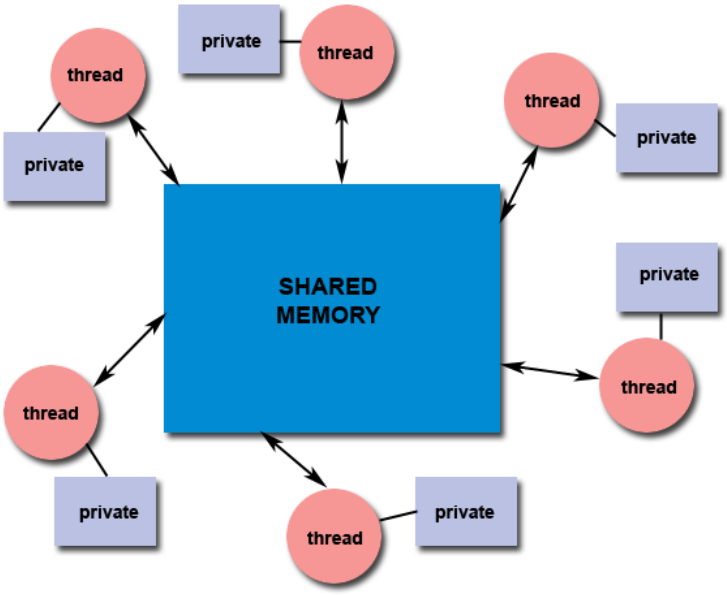


Figura 3 Interacción de las memorias con los hilos. Tomado de [8]

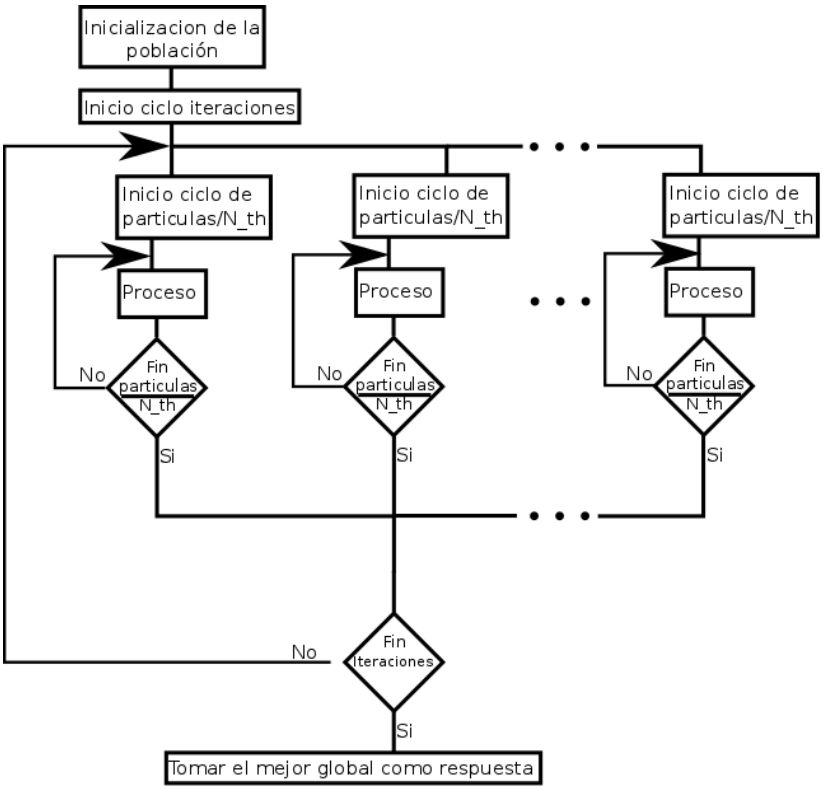


Figura 4 Diagrama de la implementación usando Pthreads.

3.3. Implementación Usando CUDA-C

El lenguaje de programación CUDA-C (por sus siglas en ingles *Compute Unified Device Architecture-C*) fue creado en el 2007 por la compañía Nvidia para las plataformas de Windows y Linux permitiendo a los programadores una variación del lenguaje C para la codificación de algoritmos usando GPUs. Estas GPUs las cuales también son producidas por Nvidia poseen una arquitectura basada en múltiples núcleos de procesamiento el cual permite realizar operaciones rutinas simultáneamente (paralelo) mejorando los tiempos de cómputo [9].

Figura 5 GPU Tesla K40. Tomada de [9].

Para la realización del código en CUDA-C se tuvo en cuenta los parámetros de la GPU que son: El número máximo de hilos por bloques, la máxima dimensión de los bloques, el tamaño de la malla, su memoria total, mencionando sus más importantes. El diseño del algoritmo es similar al usado utilizando Pthreads solo que esta vez no se agrupó partículas por hilo sino que esta vez cada hilo era el encargado de realizar el proceso para cada partícula. En la figura 6 se puede observar el diagrama del algoritmo que se realizó.

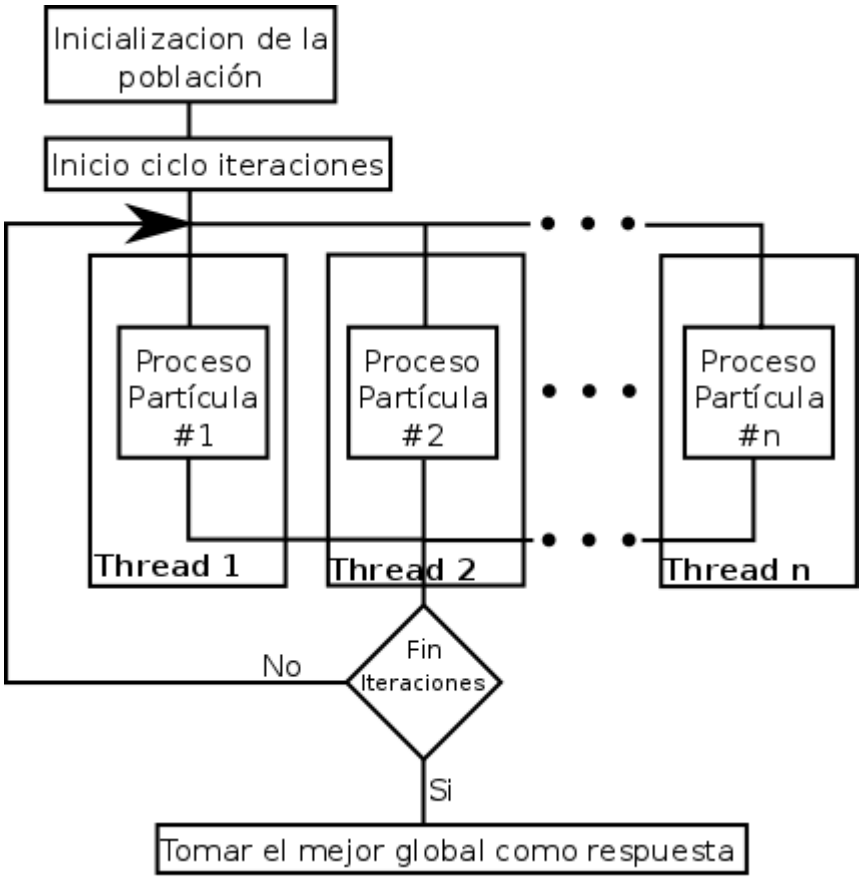


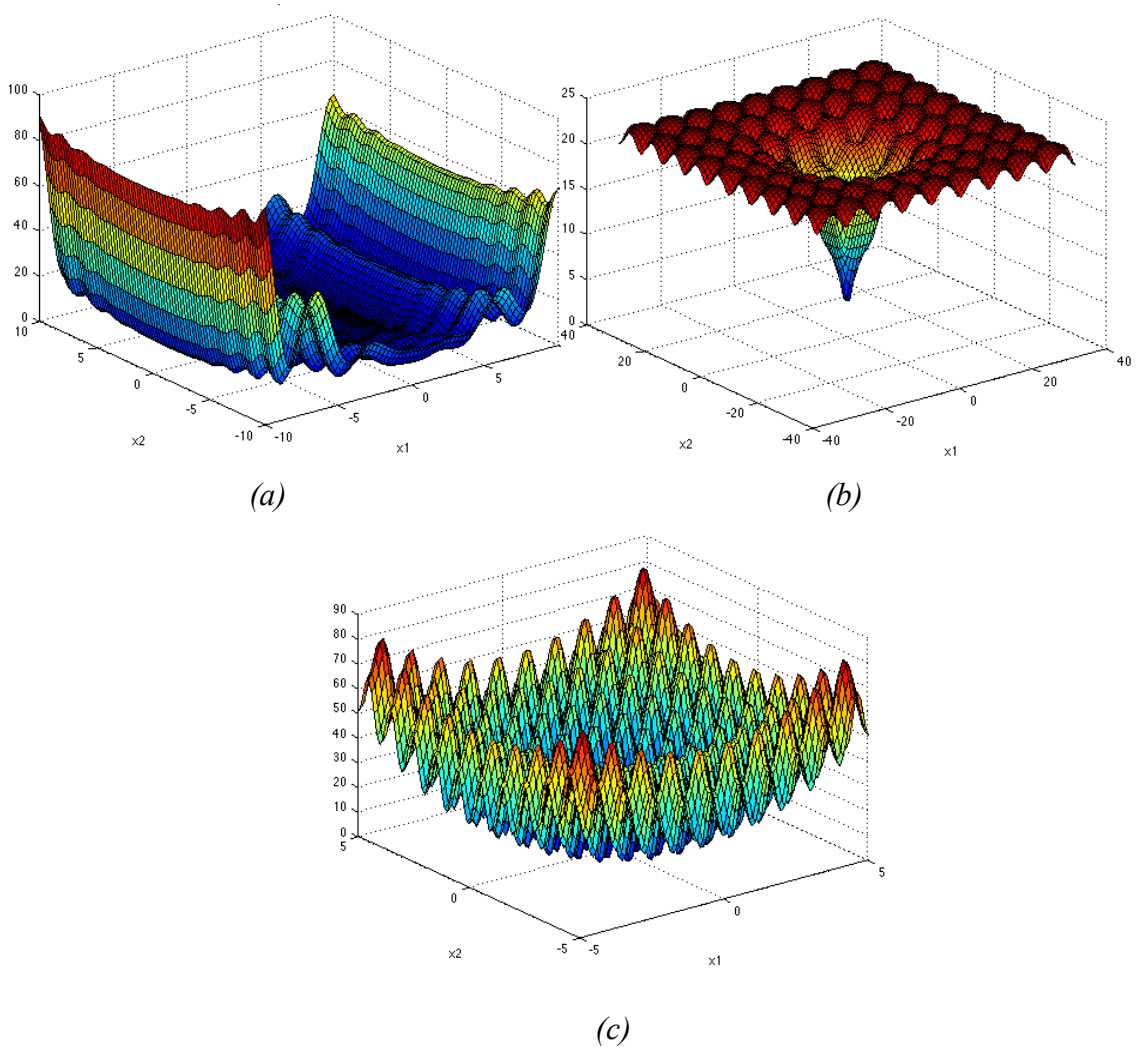
Figura 6 Diagrama implementación en CUDA-C.

4. EXPERIMENTO

El tipo de experimento que se realizó fue la medición de los tiempos de ejecución de cada uno de los tres códigos escritos bajo los mismos parámetros de funcionamiento, que en los tres casos fueron el número de iteraciones y el número de partículas, con la excepción para la implementación usando Pthreads el cual también se varió el número de threads y para CUDA-C el número de bloques, obteniendo graficas de tiempos de procesado para poder con estas realizar una comparación de eficiencia comparadas siempre con la implementación tradicional serial C.

Las funciones costo que se utilizaron para la prueba de funcionamiento del algoritmo fueron extraídas de [10] y son funciones n-dimensionales y que la comunidad científica usa para la verificación del correcto funcionamiento de los algoritmos de optimización. Las tres funciones

que se tuvieron en cuenta fueron: la función Levy, Ackley y Rastrigin (ver figura 7), ya que sus formas intrincadas y complejas aportaban un grado de dificultad alto para la optimización usando el algoritmo PSO.



(a) *(b)*

(c)

Figura 7 Funciones de prueba tomadas de *[10]* (a) Funcion Levy (b) Funcion Ackley (c) Funcion Rastrigin.

El rango de los parámetros que se variaron fue teniendo en cuenta el error del valor arrojado por el algoritmo y el valor teórico de la función, tomando una tolerancia del 1%.

4.1. Equipo de Computo Utilizado

Para las pruebas se utilizó un equipo de cómputo CPU y GPU y cuyas características se puede ver en las tablas 1 y 2 respectivamente.

Tabla 1 Especificaciones equipo de cómputo CPU

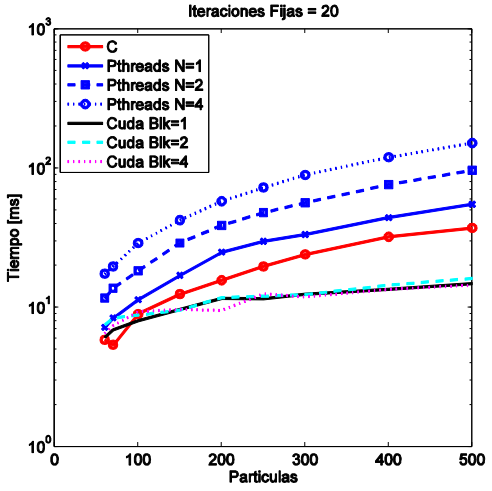
Modelo	Intel® Xeon® CPU E5-2609 v2 @ 2.50 GHz
Arquitectura	x86_64
Número Cores	8
L1d cache	32k
L2 cache	256k
L3 cache	1024k

Tabla 2 Especificaciones e equipo de cómputo GPU

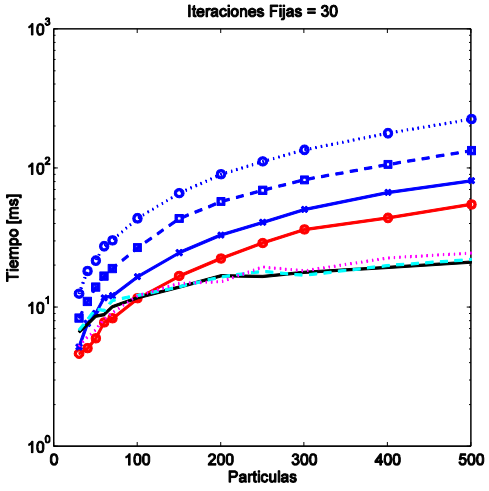
Nombre	Tesla K40c
Velocidad del reloj	745 MHz
Cantidad de SM's	15
Hilos por bloque max	1024
Memoria compartida por bloque	49.152 Mb
Dimensión máxima por bloque	1024x1024x64

5. RESULTADOS

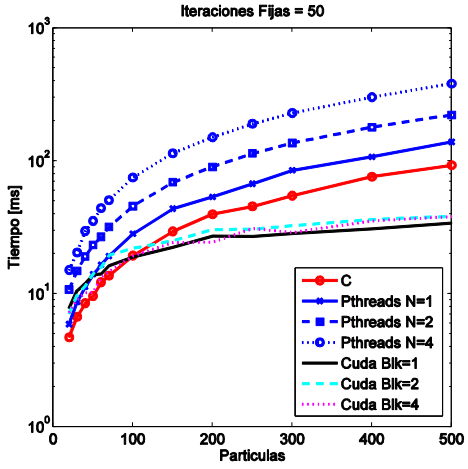
Se hizo la toma de tiempos para los diferentes modelos realizados obteniendo como resultado las siguientes gráficas:


Iteraciones Fijas = 20
C
Pthreads N=1
Pthreads N=2
Pthreads N=4
Cuda Blk=1
Cuda Blk=2
Cuda Blk=4
Tiempo [ms]
Particulas

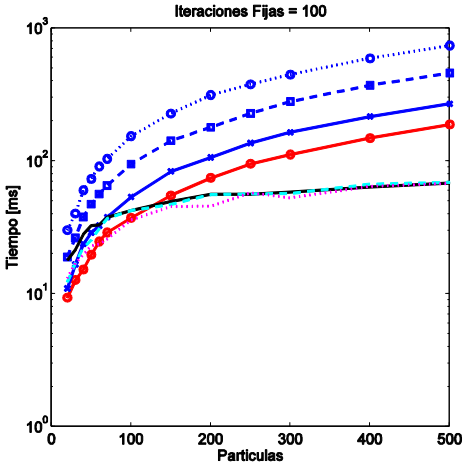
(a)


Iteraciones Fijas = 30
Tiempo [ms]
Particulas

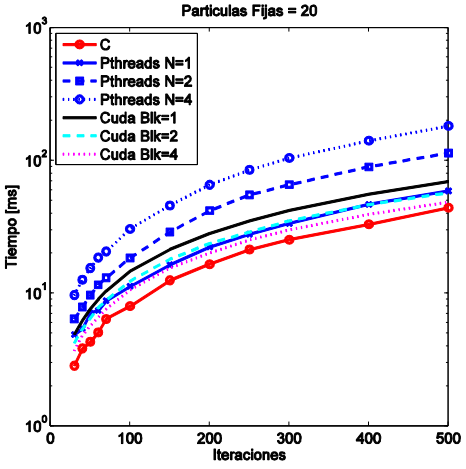
(b)


Iteraciones Fijas = 50
C
Pthreads N=1
Pthreads N=2
Pthreads N=4
Cuda Blk=1
Cuda Blk=2
Cuda Blk=4
Tiempo [ms]
Particulas

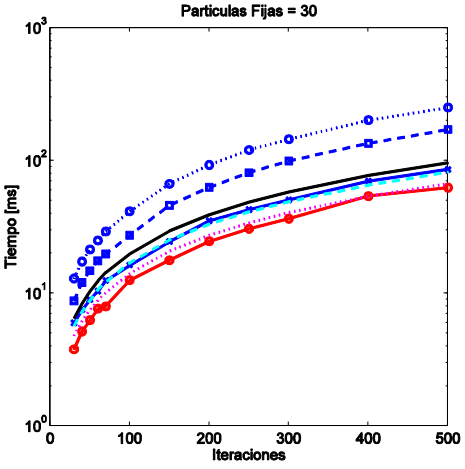
(c)


Iteraciones Fijas = 100
Tiempo [ms]
Particulas

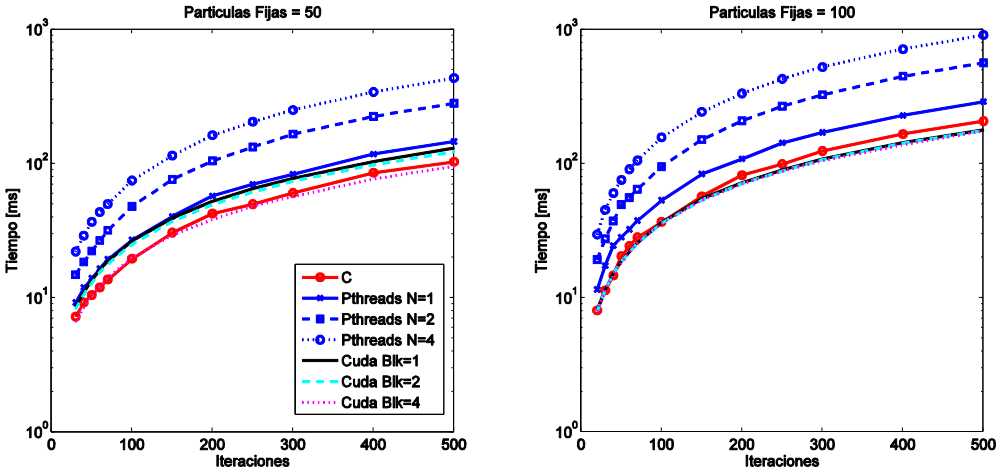
(d)


Particulas Fijas = 20
C
Pthreads N=1
Pthreads N=2
Pthreads N=4
Cuda Blk=1
Cuda Blk=2
Cuda Blk=4
Tiempo [ms]
Iteraciones

(e)


Particulas Fijas = 30
Tiempo [ms]
Iteraciones

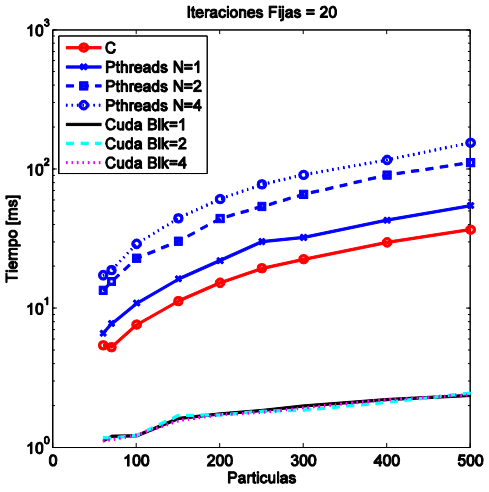
(f)



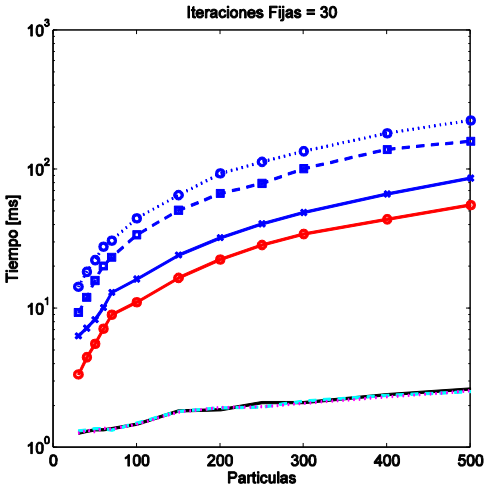
(g) *(h)*

Figura 8 Función de prueba Levy (a), (b), (c) y (d) Resultados de tiempo fijando el parámetro iteraciones. (e), (f), (g) y (h) Resultados de tiempo fijando el parámetro Partículas.

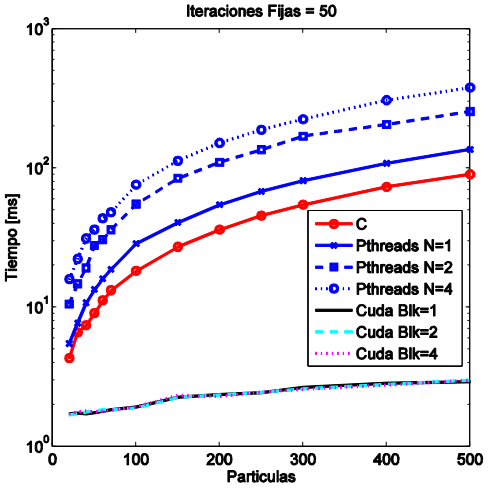
De la figura 8 se puede observar que el modelo de programación usando Pthreads no brinda ninguna mejora en ninguna de los dos tipos de prueba que se realizaron comparándola con el código serial en C. Además en la figura 8 (a)-(d) se puede extraer que las implementaciones en CUDA-C posee una condición mínima del número de *threads* (partículas) utilizados para que la estrategia tenga una aceleración en comparación a la forma tradicional. Cabe resaltar que los resultados que arrojó la prueba manteniendo fijas las partículas, el comportamiento de las implementaciones en CUDA-C se observa una aceleración usando 4 bloques en comparación a sus otras 2, esto es debido a que la GPU utiliza muy pocos recursos físicos dificultando la ejecución en comparación cuando se utilizan 1 y 2 ya que este equipo de cómputo está diseñado para el cálculo masivo de datos, y esto se puede evidenciar en la gráfica 8 (h) donde aumentando sus números de threads el comportamiento de las tres pruebas en CUDA-C se comportan de la misma manera.


Iteraciones Fijas = 20
C
Pthreads N=1
Pthreads N=2
Pthreads N=4
Cuda Blk=1
Cuda Blk=2
Cuda Blk=4
Tiempo [ms]
Particulas

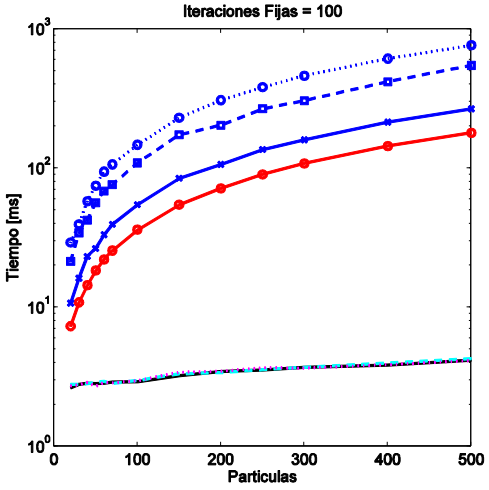
(a)


Iteraciones Fijas = 30
Tiempo [ms]
Particulas

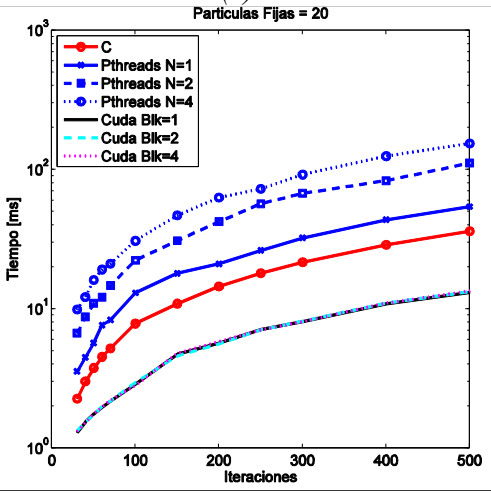
(b)


Iteraciones Fijas = 50
C
Pthreads N=1
Pthreads N=2
Pthreads N=4
Cuda Blk=1
Cuda Blk=2
Cuda Blk=4
Tiempo [ms]
Particulas

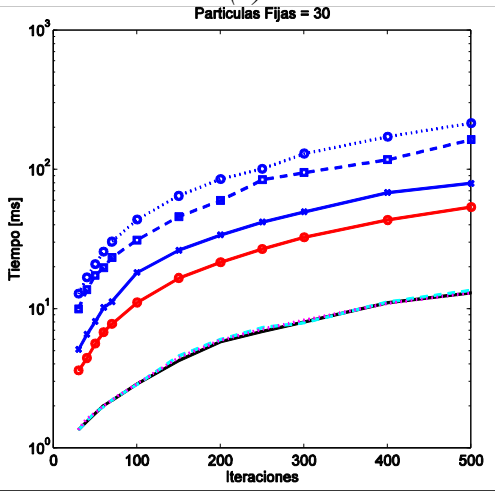
(c)


Iteraciones Fijas = 100
Tiempo [ms]
Particulas

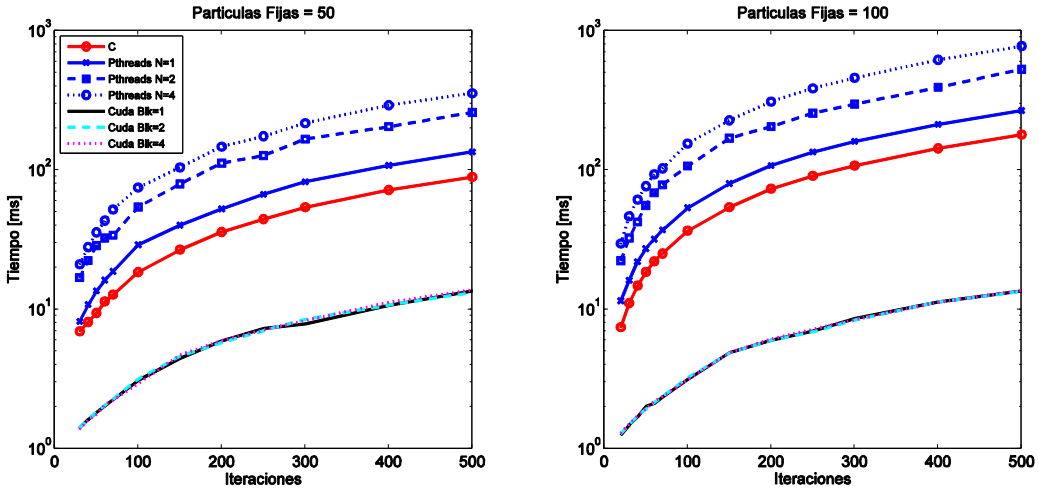
(d)


Particulas Fijas = 20
C
Pthreads N=1
Pthreads N=2
Pthreads N=4
Cuda Blk=1
Cuda Blk=2
Cuda Blk=4
Tiempo [ms]
Iteraciones

(e)

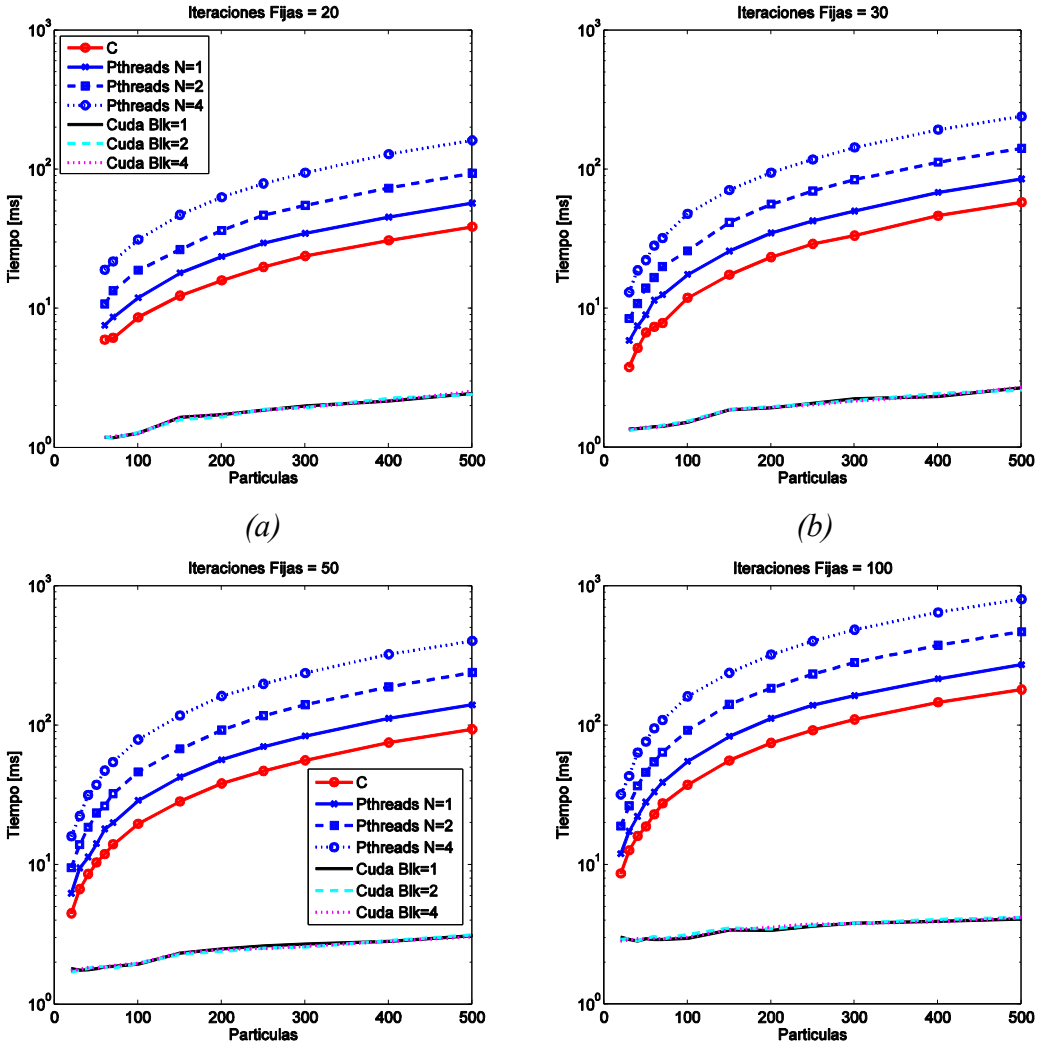

Particulas Fijas = 30
Tiempo [ms]
Iteraciones

(f)



(g) *(h)*

Figura 9 Función de prueba Ackley (a), (b), (c) y (d) Resultados de tiempo fijando el parámetro iteraciones. (e), (f), (g) y (h) Resultados de tiempo fijando el parámetro Partículas.



(a) *(b)*

(c) *(d)*

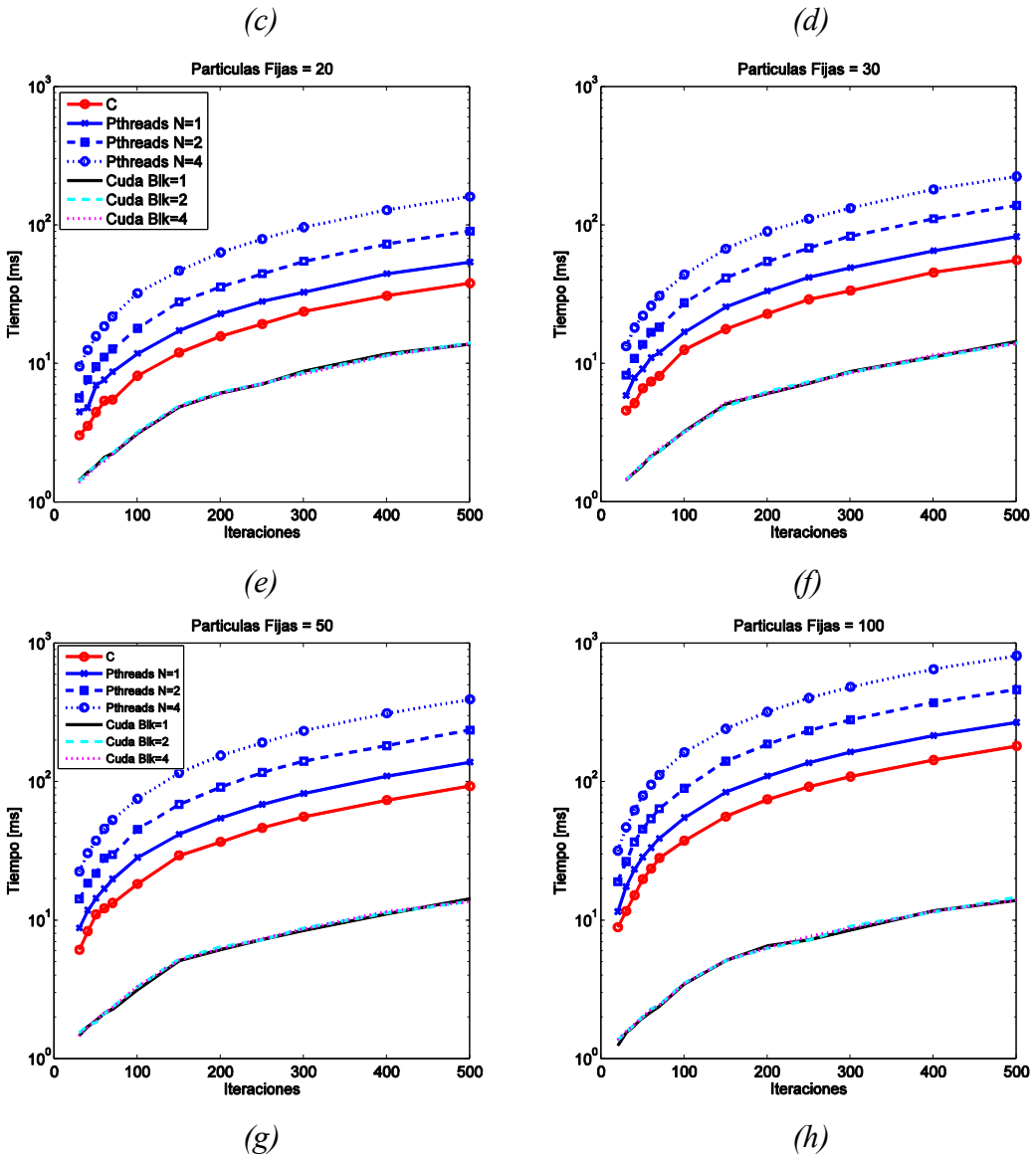


Figura 10 Función de prueba Rastrigin (a), (b), (c) y (d) Resultados de tiempo fijando el parámetro iteraciones. (e), (f), (g) y (h) Resultados de tiempo fijando el parámetro Partículas.

En las figuras 9 y 10 se puede observar que las implementaciones en CUDA-C, en las pruebas donde se dejó el valor de iteraciones fijas, una mejora de 2.51 a 42.96 veces para la función Ackley y de 2.49 a 44.16 para la función Rastrigin. Y para la segunda prueba la cual fue dejar las partículas fijas, se obtuvo una mejora de 1.74 a 13.31 y de 2.11 a 13.02 para las funciones Ackley y Rastrigin respectivamente. Este comportamiento variable de la aceleración es debido a que la curva de tiempo de las implementaciones en CUDA-C no son lineales porque los recursos utilizados por la GPU siempre es constante ya sea para poca o mucha información,

es decir, que para el primer caso se estaría desaprovechando el potencial de la tarjeta y para la el segundo se aprovecharía al máximo. La implementación usando Pthreads se comporta de la misma manera como la anterior función de prueba obteniendo un mayor tiempo de ejecución.

A continuación se muestran los valores de cada una de las gráficas en forma de tablas unificando las tres funciones de prueba utilizadas:

Tabla 3 Prueba dejando iteraciones fijas = 20

Iteraciones Fijas = 20									
Partículas	Levy			Ackley			Rastrigin		
	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]
60	5.869	7.209	6.106	5.457	6.657	1.125	5.98	7.59	1.19
70	5.410	8.433	6.940	5.287	7.810	1.215	6.17	8.69	1.18
100	8.979	11.395	8.066	7.676	10.949	1.218	8.65	11.99	1.27
150	12.508	17.033	9.687	11.322	16.360	1.626	12.38	18.05	1.66
200	15.723	25.017	11.631	15.321	22.101	1.751	15.94	23.64	1.73
250	19.749	29.903	11.590	19.496	30.278	1.847	19.93	29.61	1.86
300	24.065	33.438	12.467	22.635	32.446	1.992	23.92	34.77	1.99
400	32.262	44.259	13.485	29.893	43.144	2.225	30.92	45.45	2.16
500	37.335	55.203	14.805	36.909	55.024	2.381	38.69	57.26	2.45

Tabla 4 Prueba dejando iteraciones fijas = 30

Iteraciones Fijas = 30									
Partículas	Levy			Ackley			Rastrigin		
	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]
30	4.671	5.209	6.648	3.358	6.358	1.272	3.80	5.90	1.33
40	5.106	7.697	7.537	4.464	7.238	1.333	5.20	7.53	1.37
50	6.021	9.101	8.704	5.580	8.344	1.306	6.73	9.03	1.38
60	7.811	11.744	8.879	7.155	10.190	1.377	7.39	11.48	1.40
70	8.373	12.148	10.124	9.029	13.059	1.363	7.88	12.57	1.45
100	11.667	16.688	11.761	11.111	16.340	1.477	11.94	17.62	1.55
150	16.861	24.817	14.039	16.638	24.241	1.805	17.53	25.92	1.88
200	22.536	33.178	16.935	22.539	32.286	1.946	23.42	34.92	1.97
250	29.105	40.980	16.758	28.639	40.621	1.971	29.14	42.67	2.06
300	36.405	50.746	17.891	34.279	49.085	2.083	33.44	50.35	2.18
400	44.072	67.202	19.446	43.822	66.597	2.325	46.54	68.41	2.46

500	55.043	81.622	21.272	55.469	86.497	2.556	58.19	85.47	2.60

Tabla 5 Prueba dejando iteraciones fijas = 50

Iteraciones Fijas = 50									
Partículas	Levy			Ackley			Rastrigin		
	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]
20	4.719	5.922	7.795	4.311	5.506	1.682	4.50	6.26	1.71
30	6.745	8.709	10.570	6.605	7.714	1.715	6.72	9.53	1.76
40	8.507	11.332	12.034	7.453	10.798	1.768	8.63	11.43	1.82
50	9.643	14.069	13.951	9.112	13.458	1.580	10.41	14.26	1.89
60	12.240	16.586	14.157	11.240	16.093	1.834	11.99	18.12	1.86
70	13.760	19.271	16.280	13.289	18.780	1.839	14.07	20.15	1.83
100	19.418	28.312	18.956	18.299	28.692	1.888	19.76	29.06	1.97
150	29.552	43.857	22.455	27.258	40.714	2.261	28.59	42.69	2.29
200	39.787	53.745	27.238	36.119	54.616	2.347	38.37	56.87	2.40
250	45.552	67.520	27.136	45.646	68.181	2.459	47.16	70.62	2.53
300	54.904	85.226	28.505	54.582	81.290	2.595	56.33	84.12	2.59
400	76.232	107.689	30.915	73.340	108.562	2.805	75.45	112.64	2.87
500	92.945	139.377	34.061	90.561	136.369	2.996	93.99	140.77	3.15

Tabla 6 Prueba dejando iteraciones fijas = 100

Iteraciones Fijas = 100									
Partículas	Levy			Ackley			Rastrigin		
	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]
20	9.368	11.013	13.216	7.329	10.730	2.641	8.69	12.03	2.85
30	12.714	16.673	17.097	10.864	16.182	2.797	12.77	17.46	2.89
40	15.283	23.783	19.721	14.428	23.252	2.848	16.15	22.31	2.96
50	19.775	28.963	22.997	18.400	26.434	2.825	18.89	28.28	2.95
60	24.817	32.668	23.802	22.139	33.234	2.858	23.12	33.41	2.95
70	28.979	37.910	25.885	25.597	39.545	2.893	27.61	39.17	2.95
100	37.404	53.684	36.210	36.215	54.754	2.932	37.65	55.43	3.03
150	55.116	83.690	45.422	54.629	84.714	3.228	56.12	83.69	3.42
200	74.630	106.631	45.756	71.667	106.678	3.456	75.09	112.70	3.60
250	95.221	136.904	57.715	90.261	136.112	3.543	92.60	139.81	3.78
300	111.970	164.644	52.892	108.181	160.157	3.709	110.57	163.76	3.80
400	148.923	216.228	64.612	144.108	214.645	3.843	146.78	216.45	3.93
500	188.426	269.686	68.043	179.816	266.677	4.186	181.10	273.11	4.19

Tabla 7 Prueba dejando Partículas fijas = 20

Particulas Fijas = 20									
Iteraciones	Levy			Ackley			Rastrigin		
	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]
30	2.849	4.895	3.704	2.272	3.568	1.304	3.05	4.50	1.39
40	3.849	5.437	4.812	3.029	4.499	1.550	3.56	4.82	1.58
50	4.315	7.104	5.707	3.770	5.707	1.762	4.47	7.02	1.82
60	5.086	7.438	6.627	4.530	7.667	1.981	5.40	7.65	1.98
70	6.411	8.799	7.660	5.201	8.338	2.188	5.51	8.76	2.22
100	8.009	11.298	10.594	7.845	13.046	2.897	8.20	11.89	3.12
150	12.541	16.423	15.487	10.916	18.031	4.746	12.03	17.34	4.91
200	16.575	22.374	20.136	14.502	21.103	5.692	15.83	23.02	6.08
250	21.401	27.890	25.224	18.080	26.340	7.108	19.38	28.24	7.23
300	25.368	33.743	30.094	21.697	32.381	8.101	23.84	32.92	8.48
400	33.140	46.803	39.434	28.832	43.691	10.863	30.95	44.70	11.52
500	44.215	59.077	48.759	36.214	54.266	13.179	38.18	54.36	13.86

Tabla 8 Prueba dejando Partículas fijas = 30

Particulas Fijas = 30									
Iteraciones	Levy			Ackley			Rastrigin		
	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]
30	3.779	5.992	6.505	3.623	5.132	1.363	4.60	5.90	1.47
40	5.158	7.529	8.426	4.447	6.585	1.560	5.19	7.93	1.66
50	6.293	8.959	10.414	5.649	8.147	1.777	6.64	9.18	1.90
60	7.674	10.395	12.388	6.815	10.236	2.028	7.44	11.08	2.19
70	7.979	12.499	14.365	7.820	11.278	2.209	8.18	12.06	2.42
100	12.576	16.344	19.916	11.134	18.280	2.919	12.63	16.97	3.18
150	17.754	24.575	29.479	16.692	26.378	4.254	17.80	25.80	5.26
200	24.694	35.158	39.123	21.656	34.041	5.820	22.95	33.59	6.11
250	30.676	42.739	48.716	26.962	42.061	6.918	29.13	42.09	7.25
300	36.553	50.660	58.206	32.684	49.791	8.091	33.72	49.44	8.56
400	54.077	70.177	77.214	43.503	68.505	11.118	45.69	65.54	11.54
500	62.582	85.841	96.587	54.003	79.835	13.070	56.00	83.00	13.96

Tabla 9 Prueba dejando Partículas fijas = 50

Particulas Fijas = 50									
Iteraciones	Levy			Ackley			Rastrigin		
	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]
30	7.267	9.224	6.625	6.957	8.205	1.387	6.15	8.82	1.48
40	9.265	11.943	8.522	8.111	10.846	1.612	8.38	11.90	1.70
50	10.509	14.062	10.268	9.417	13.604	1.770	11.08	14.48	1.93
60	11.999	16.593	12.186	11.440	16.262	2.073	12.26	16.97	2.16
70	13.710	19.406	14.233	12.752	18.759	2.244	13.37	19.96	2.36
100	19.620	27.089	19.992	18.550	29.091	2.927	18.39	28.50	3.35
150	30.663	40.599	29.415	26.809	40.204	4.708	29.49	41.95	5.18
200	42.510	57.499	38.344	35.954	52.649	5.969	36.94	54.88	6.22
250	49.973	70.317	48.440	44.476	66.928	7.110	46.72	68.84	7.39
300	60.590	83.702	56.866	54.160	82.358	8.337	56.18	82.59	8.69
400	85.727	117.999	76.814	72.052	108.064	11.210	73.94	110.38	11.62
500	103.320	146.588	95.560	89.276	135.011	13.776	93.34	138.97	13.73

Tabla 10 Prueba dejando Partículas fijas = 100

Particulas Fijas = 100									
Iteraciones	Levy			Ackley			Rastrigin		
	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]	C [ms]	Pthreads [ms]	CUDA-C [ms]
20	8.073	11.583	7.986	7.446	11.516	1.256	8.69	12.03	2.85
30	11.374	17.424	11.429	11.081	16.189	1.482	12.77	17.46	2.89
40	14.697	24.552	14.918	14.816	22.046	1.689	16.15	22.31	2.96
50	20.474	28.342	18.256	18.579	27.296	2.005	18.89	28.28	2.95
60	24.353	32.461	21.774	22.135	31.883	2.111	23.12	33.41	2.95
70	28.301	37.716	25.538	25.158	37.167	2.357	27.61	39.17	2.95
100	36.839	53.572	36.138	36.752	53.570	3.154	37.65	55.43	3.03
150	57.128	84.034	53.345	53.995	80.080	4.879	56.12	83.69	3.42
200	82.272	108.822	70.741	73.539	107.785	6.026	75.09	112.70	3.60
250	99.362	143.180	88.312	90.878	134.840	6.998	92.60	139.81	3.78
300	124.594	171.421	105.251	107.493	160.663	8.575	110.57	163.76	3.80
400	166.708	229.167	138.802	143.113	212.579	11.304	146.78	216.45	3.93
500	208.047	290.465	175.609	179.969	267.358	13.517	181.10	273.11	4.19

6. CONCLUSIONES Y RECOMENDACIONES

La implementación del algoritmo metaheurístico PSO posee la característica de ser altamente paralelable ya que su comportamiento y dinámica de funcionamiento es de tipo enjambre y sus procesos son independientes el uno al otro.

La complejidad de la función de prueba ya sea por su naturaleza o por su cálculo algorítmicamente afecta en los tiempos de procesado ya que requiere un mayor tiempo de procesado por iteración.

La utilización de la estrategia usando el modelo Pthreads arrojó resultados desfavorables de aceleración, ya que la estrategia que se realizó era necesario un uso excesivo de estructuras haciendo que el código se ralentice cuando se hacen escrituras en sus variables internas, por esta razón es necesario realizar una mejor implementación donde se reduca el número de veces que se hace un llamado a memoria.

REFERENCIAS

[1] I. Boussaïd, J. Lepagnot y P. Siarry, «A survey on optimization metaheuristics,» *Information Sciences,* pp. 82-117, 2013.

[2] R. Shams y N. Barnes, «Speeding up Mutual Information Computation Using NVIDIA CUDA Hardware,» *Digital Image Computing Techniques and Applications,* pp. 555-560, 2007.

[3] J. Kennedy y R. Eberhart, «Particle Swarm Optimization,» *IEEE,* 1995.

[4] S. Yuan, S. Wang y N. Tian, «Swarm intelligence optimization and its application in geophysical data inversion,» *APPLIED GEOPHYSICS,* pp. 166-174, 2009.

[5] E. V. Bonet Esteban, Lenguaje C, 2008.

[6] M. Peña Basurto y J. Cela Espín, Introducción a la programación en C, Aula Politécnica/ETSETB.

[7] V. Sosa sosa, Programación en POSIX Threads (Pthreads), Universidad Nacional de La Plata.

[8] «computing.llnl.gov,» [En línea]. Available: https://computing.llnl.gov/tutorials/pthreads/#Abstract. [Último acceso: 27 Abril 2015].

[9] N. D. Zone, «www.nvidia.es,» [En línea]. Available: http://www.nvidia.es/object/gpu-computing-es.html. [Último acceso: 20 Enero 2016].

[10] S. Surjanovic y D. Bingham, «sfu,» 2013. [En línea]. Available: http://www.sfu.ca/~ssurjano/optimization.html. [Último acceso: 25 Abril 2015].