Algoritmos de Inteligencia de Emjambres sobre GPUs: una revisión exhaustiva

S. Molina, F. Piccoli and G. Leguizamón

Laboratorio de Investigación y Desarrollo en Inteligencia Computacional (LIDIC)
Universidad Nacional de San Luis,
Ejército de los Andes 950, (5700) San Luis, Argentina
{smolina,mpiccoli,legui}@unsl.edu.ar

Resumen La *Inteligencia de Enjambres* involucra acciones de grupos de individuos descentralizados y auto-organizados, las cuales pueden realizarse en paralelo. Las GPUs han mostrado ser arquitecturas capaces de proporcionar paralelismo masivo, las cuales están en pleno desarrollo. En este trabajo se proporciona una breve revisión de metaherurísticas basadas en Inteligencia de Enjambres orientadas a GPUs, propuestas desde el año 2012. Abordamos dos de las metaheurísticas más estudiadas como la *Optimización de Colonias de Hormigas* y el *Algoritmo Colonia de Abejas*, además de otras metaheurísticas más recientes y menos estudiadas como los son: la *Búsqueda Cuco* y el *Algoritmo Fuegos Artificiales*.

1. Introducción

La *Inteligencia de Enjambres* (*Swarm Intelligence, SI*) estudia el comportamiento colectivo de sistemas compuestos por muchos individuos (el *swarm*) interactuando localmente y con su entorno. Los *swarms* inherentemente usan formas de control descentralizadas y de auto-organización para alcanzar sus objetivos (Martens et al. [14]). Ejemplos de metaheurísticas *SI* son: *Optimización de Colonias de Hormigas* (*Ant Colony Optimization*, ACO) (Dorigo and Blum [8]); *Búsqueda Cuco* (*Cuckoo Search*, CS) (Xi et al. [20]), *Algoritmo Fuegos Artificiales* (*Fireworks Algorithm*, FA) (Tan and Zhu [19]) y *Algoritmo de Colonia de Abejas*(*Bees Colony Algorithm*, BCA) (Pham et al. [16]).

Varias razones hacen a los *SI* aptos para aplicar técnicas paralelas en su solución computacional, por un lado el comportamiento de cualquier *swarm* implica acciones independientes de sus individuos. Por otro lado, cuando un algoritmo *SI* trabaja con grandes tamaños de *swarms* y muchas repeticiones del proceso, su desempeño es mejor. Además, frente a problemas de gran escala (por ejemplo, *BigData*), las técnicas *SI* deben manipular espacios grandes de búsqueda, inclusive con soluciones de alta dimensionalidad. Manipular representaciones de soluciones de estos espacios de búsqueda y evaluar la calidad de las soluciones, puede requerir mucho tiempo de procesamiento.

El objetivo de este trabajo es realizar un cuidadoso análisis del estado del arte de algoritmos *SI* para GPUs. Para ello consideramos dos de las metaheurísticas más estudiadas: *ACO* y *BCA*, y aquellas propuestas más recientemente: *CS* y

FA. El trabajo se basa en algoritmos propuestos desde el año 2012, haciendo especial énfasis en cómo se asignan los recursos del hardware subyacente a las GPUs a las distintas partes algorítmicas de manera tal de explotar el paralelismo masivo de datos para alcanzar un buen desempeño.

El trabajo está organizado de la siguiente manera, en la sección 2 se realiza una breve introducción a la arquitectura GPU y al modelo de programación CUDA describiendo sus principales características. En la sección 3 se realiza una descripción de las metheurísticas *SI*, para luego focalizarnos a partir de la sección 4 en el análisis de los algoritmos y las estrategias propuestas para *SI* sobre GPUs. Finalmente en la sección 5 se presentan las conclusiones.

2. Arquitectura GPU y Modelo de Programación CUDA

Las GPUs (*Unidades de Procesamiento Gráfico*) se han covertido en una alternativa válida a las supercomputadoras paralelas debido a su poder computacional, bajo costo y al constante avance asociado a su hardware y software.

La programación paralela de aplicaciones en la GPU, GPGPU (*General Propourse en GPU*), debe considerar varias diferencias con la programación paralela en computadoras paralelas típicas, las más relevantes son: el número de unidades de procesamiento, la estructura de la memoria CPU-GPU y el número de hilos (threads) paralelos. Respecto al número de unidades de procesamiento, a diferencia de las soluciones para computadoras masivamente paralelas donde el número de unidades de proceso coincide con el número de unidades de procesamiento (Procesadores o cores), en GPGPU esto no es considerado (es posible la ejecución de cientos o miles de unidades de proceso en un número inferior de cores). En el caso de la estructura de la memoria del sistema CPU-GPU, existen dos espacios de memoria diferentes: la memoria del host (CPU) y la memoria de la GPU. Como los hilos de la GPU ejecutan en un espacio de memoria separado a los hilos del host en la aplicación, las transferencias de código y datos son necesarios entre el host y la GPU. Finalmente, la programación de las GPU tiene la posibilidad de iniciar un gran número de hilos con poca sobrecarga. Toda GPU tiene mecanismos transparentes y de bajo costo para la creación y administración de hilos. Además poseen mecanismos de planificación propios, los hilos son planificados a *warps*. Un *warp* es la unidad de planificación y está formado por 32 hilos.

Si bien existen diferentes alternativas para procesamiento en GPU, la más ampliamente utilizada es la tarjeta *nVidia*, con un ambiente de desarrollo denominado *Compute Unified Device Architecture* (CUDA), el cual ha sido diseñado para simplificar el trabajo de sincronización de hilos y la comunicación con la GPU. CUDA propone un modelo de programación SIMD (*Single Instruction Multiple Data*) [11,18], y presenta a la arquitectura de la GPU como un conjunto de multi-procesadores (*Streaming Multiprocessors, SM*) MIMD (Múltiple Instrucciones-Múltiple Datos). Cada SM posee un conjunto de procesadores (*Streaming Processors*, SP) SIMD. Respecto a la jerarquía de memoria, cada SP posee una serie de registros a modo de memoria local (sólo accesible por él), a

su vez cada SM tiene una memoria compartida o *shared* (accesible por todos los SP del SM) y finalmente la memoria global, a la cual acceden todos los SM.

El programador desarrolla un único programa CUDA, el cual contiene el código de la CPU (fase de poco o nulo paralelismo) y del dispositivo GPU (fase con paralelismo de datos y de gránulo fino). El código para la GPU se denomina *kernel*, es ejecutado por cada hilo. Cada vez que un *kernel* es activado, se genera una estructura en la GPU denominada *grid*, el cual está formado por *bloques* de hilos. Cada bloque se ejecuta en un SM. La cantidad de hilos por bloque y los recursos para cada uno de ellos es determianda por el programador, pudiendo realizar ajustes a fin de lograr mejoras en el desempeño.

3. Inteligencia de Enjambres

El campo vinculado a la inteligencia de enjambres estudia el comportamiento colectivo de sistemas compuestos de muchos individuos (*swarm*) interactuando localmente y con su entorno. Los *swarms* inherentemente usan formas de control descentralizadas y auto-organización para alcanzar sus objetivos (Martens et al. [14]). Como dijimos antes, en este trabajo nos focalizamos en el estudio de algoritmos para GPUs de metaherurísticas *SI*, tal como ACO, BCA, CS y FA. Las características de cada una de ellas son:

- *ACO* (Dorigo and Blum [8])
  Es una metaheurística poblacional inspirada en el comportamiento de la búsqueda de alimento de una especie de hormigas. Las hormigas inicialmente exploran el área alrededor del nido en una manera aleatoria. Cuando una hormiga encuentra una fuente de comida, ésta evalúa la calidad y la cantidad de la misma y la lleva al nido, depositando en el entorno una sustancia química denominada *feromona*, dependiendo de la cantidad y la calidad de la comida encontrada.
  Un algoritmo *ACO* puede ser visto como una interacción de tres procedimientos: *Construcción de Soluciones* (el cual maneja una colonia de hormigas, donde concurrente y asincrónicamente visitan los estados del problema considerado, mediante movimientos a través de los nodos vecinos del grafo de construcción, representante del espacio de búsqueda de la solución), *Modificación de Feromona* (encargado del proceso de depósito y evaporación de la feromona); y *Acciones Demonios* (lleva a cabo las acciones centralizadas, las cuales no pueden ser realizadas por una única hormiga).
  En [1] y [15] se muestran variantes de algoritmos ACO, donde se proponen versiones secuenciales y paralelas.
- *BCA* (Pham et al. [16])
  Se inspira en el comportamiento de búsqueda de alimento de las abejas productoras de miel, realizando un tipo de búsqueda de vecindad combinada con una búsqueda aleatoria.
- *CS* (Xi et al. [20])
  Se basa en el comportamiento parásito de algunas especies de cucos a la hora

de depositar sus huevos en nidos ajenos. Los huevos del cuco han evolucionado en forma y color hasta imitar a los huevos del pájaro dueño del nido. Si dicho pájaro descubre que el huevo no es suyo, lo tira o abandona el nido, construyendo un nido en otro lugar. Si el huevo del cuco eclosiona el ciclo se repite.

- *FWA* (Tan and Zhu [19], Ding et al. [7])
  Es un nuevo algoritmo *SI* inspirado en el fenómeno de explosión de los fuegos artificiales. El proceso de optimización es guiado por las explosiones de un swarm de fuegos artificiales. El fuego con mejor fitness genera más chispas dentro de un rango pequeño. Para mejorar la búsqueda son necesarios más recursos de computación en las mejores zonas del espacio de búsqueda y menos en las peores zonas de dicho espacio de búsqueda. También se introduce la posibilidad de incrementar la diversidad del swam aplicando *Mutación Gaussiana*.

En la siguiente sección analizamos las características de algoritmos *SI* propuestos para un sistema de computación CPU-GPU.

4. Algoritmos y Estrategias SI sobre GPUs

La propuesta de este trabajo surge a partir de la observación de que se han realizado varios trabajos abocados al estudio del estado del arte de metaheurísticas paralelas en general y sus tendencias (Alba et al. [1]) otros más abocados a *SI* específicas (Krömer et al. [12] y Pedemonte et al. [15]), pero de los tres trabajos mencionados sólo en el último se describe más detalladamente cómo se paralelizan las partes de un determinado algoritmo. En dicho trabajo, presentan una revisión sobre metaheurísticas ACO paralelas publicadas desde el año 1993 hasta el año 2010, muestran el interés creciente de este campo de estudio siendo más notable en los años 2009 y 2010, sólo hace referencia a 4 artículos relacionados a GPUs. Además, proponen una nueva taxonomía de estrategias para implementaciones ACO paralelas, la cual incluye ideas generales de las taxonomías propuestas por Randall and Lewis [17] y Janson et al. [9]. Por esto y debido a que hasta el momento no se han publicado recientemente trabajos con las características que proponemos, es que representa una primera revisión, aunque breve pero interesante aportación, que puede ser de utilidad a los investigadores de esta área y les permita profundizar en esta línea de investigación.

El cuadro 1 muestra los algoritmos *SI* y/o estrategias orientadas a GPUs propuestas en las publicaciones revisadas. Dichas publicaciones corresponden a metaheurísticas que han sido más ampliamente estudiadas en general como: *ACO* y *BCA*, pero que respecto al uso de GPUs están todavía en una etapa de maduración como es el caso de *ACO* y en una etapa de exploración como es el caso de *BCA*. Otras publicaciones, corresponden a *SI* recientemente propuestas: *CS* y *FA* y con mucho por investigar en lo que se refiere a algoritmos sobre GPUs.

A través de las implementaciónes sobre GPUs, los investigadores, buscan explotar el paralelismo masivo intentando de esta manera mejorar el tiempo

SI	**Algoritmo/ Estrategia (e)**	**Problema**	**Año**	**Referencia**
ACO	[6]-ACO	*VRP*	2012	Diego et al. [6]
ACO-MMAS	ANT_{thread} (e) ANT_{block} (e) $COLONY_{block}$ (e) $COLONY_{GPU}$ (e)	*TSP*	2013	Delévacq et al. [5]
ACO-AS	[2]-AS	*TSP*	2013	Cecilia et al. [2]
ACO-AS	[4]-AS	*TSP*	2013	Dawson and Stewart [4]
ACO-AS	[3]-AS	*TSP*	2013	Dawson and Stewart [3]
FA	*GPU-FWA*	Benchmark	2013	Ding et al. [7]
CS	[10]-CS	Benchmark	2013	Jovanovic and Tuba [10]
BCA	*CUBA*	Benchmark	2014	Luo et al. [13]

Cuadro 1. Algoritmos *SI* y estrategias (e) desarrolladas sobre GPUs. En donde *TSP* se refiere a *Traveling Salesman Problem*) y *VRP* a *Vehicle Routing Problem*. Con Benchmark se hace referencia a un conjunto de funciones.

de cómputo. Dicho tiempo, se logra disminuir por un buen aprovechamiento del hardware y del software subyacente a las GPUs. Por ejemplo: evitando las transferencias costosas en tiempo entre la CPU y la GPU, evitando utilizar la memoria global debido a que ésta tiene mayor latencia, utilizando funciones eficientes de la librería CUDA, balanceando la carga de trabajo entre los hilos de un mismo bloque para aprovechar al máximo el paralelismo, evitando las divergencias warp para que no se produzcan serializaciones en las ejecuciones, etc.

Las métricas del tiempo de cómputo más utilizadas son el *Speedup* y la *Eficiencia*. En los trabajos revisados se alcanzan siempre valores superlineales. Cabe destacar que algunos trabajos han logrado inclusive mejorar la calidad de las soluciones.

Respecto a la plataforma computacional utilizada en la experimentación, en la mayoría de los casos se utilizan sistemas con GPUs *nVidia* con arquitecturas *Fermi* y se utiliza CUDA.

A continuación resaltamos las particularidades más relevantes de los algoritmos de cada una de las publicaciones revisadas.

4.1. Revisión de Publicaciones Relacionadas a *SI* para *GPU*

En esta sección, mostramos cómo los investigadores logran obtener algoritmos con un buen desempeño con el uso de GPUs. Nos focalizamos en describir cómo las partes algorítmicas son mapeadas en los distintos elementos de procesamiento, tales como: warps, hilos, bloques, grids y procesadores; además de mencionar cómo se administra la jerarquía de memoria para el almacenamiento de las principales estructuras de datos de cada tipo de *SI*. Respecto a las características de la arquitectura de la GPU utilizadas y las métricas consideradas en cada caso, sólo se mencionarán aquellas que no se incluyen en la introducción de la sección 4.

Metaheurística ACO sobre GPU. En la mayoría de las publicaciones revisadas se proponen algoritmos para el problema *TSP* (Delévacq et al. [5], Cecilia et al. [2], y Dawson and Stewart [4,3]), un caso excepcional es el presentado en Diego et al. [6], en donde se aborda el problema VRP.

- Delévacq et al. [5] describen dos algoritmos *Hormigas Paralelas* y *Múltiples Colonias de Hormigas Paralelas* basados en el *Max-Min Ant System* con *Búsqueda Local 3-opt*.
  En *Hormigas Paralelas*, las hormigas construyen un tour en forma paralela. Se proponen dos estrategias: (i) ANT_{thread}, en donde las hormigas son distribuidas entre los *SP*, cada una en un hilo. Cada hilo, computa la regla de transición de estado de cada hormiga en un modo SIMD. Las estructuras de datos se almacenan en la memoria global y (ii) ANT_{block}, aquí cada hormiga se corresponde a un bloque, un único hilo de un bloque se encarga de la construcción de un tour, el resto de los hilos del bloque calculan en paralelo la regla de transición de estado y la evaluación de las ciudades candidatas a ser la próxima componente de la solución corriente. Para evaluar los beneficios y limitaciones respecto a la memoria que se utiliza (compartida o global) para los datos necesarios para el cómputo de la regla de transición de estado se analizan dos variantes de esta estrategia: ANT_{Block}^{Shared} y ANT_{Block}^{Global} con esta última se obtiene un mayor *Speedup*.
  Para *Múltiples Colonias de Hormigas*, se proponen dos estrategias: (i) $COLONY_{block}$, en donde el número de bloques y hilos se corresponde al número de colonias y hormigas respectivamente y (ii) $COLONY_{GPU}$, en donde las colonias de hormigas completas se asignan a varias GPUs interconectadas. La estrategia ANT_{Block}^{Global} es integrada a cada colonia.
  Para evitar la transferencia entre la CPU y la GPU, se utiliza el generador de números aleatorios *Congruencial Lineal* (Yu et al. [21]) sobre la GPU.
- Cecilia et al. [2] presentan el primer algoritmo ACO para el TSP desarrollado completamente sobre la GPU. Proponen varios diseños del algoritmo *Ant System* (*AS*) para GPU en busca de explotar el paralelismo de dato y la jerarquía de memoria.
  Para mejorar el bandwith de las memorias, se utiliza la memoria compartida para las estructuras de datos. Además, se propone una técnica de *Tiling*, en donde todos los hilos cooperan para cargar el dato desde la memoria global a la memoria compartida.
  Para la construcción del tour, se distinguen dos tipos de hormigas, las *Reinas* y las *Obreras*, estas últimas ayudan a las primeras en la selección de la próxima ciudad a visitar. Se asocia a cada reina un bloque en donde cada hilo representa una de las ciudades que las obreras deben visitar. Con una lista *Tabú* (las ciudades que no pueden ser visitadas) se evitan las divergencias *Warps*, esta lista es almacenada en la memoria compartida o en un registro de un archivo, según se use o no la técnica *Tiling* para distribuir las ciudades entre los hilos de un bloque.
  A partir de mejoras al método *La Rueda de la Ruleta* (*RR*) surge el método *Independent Roulette* (*I-Roulette*) el cual es más paralelizable que el primero.

Un único hilo realiza la evaporación de la matriz de feromonas. Para el depósito de feromona, se utilizan transformaciones *Scatter-to-gather* para unificar los valores que deposita cada hormiga, evitando así realizar operaciones atómicas costosas. Se crea un hilo en los distintos bloques para las celdas de la matriz.

- Dawson and Stewart [4] proponen una implementación paralela de un algoritmo *AS*. Extienden investigaciones recientes como la de Cecilia et al. [2]. Ejecuta en las GPUs: La inicialización de las estructuras, la construcción de la solución y la actualización de la matriz de feromonas.
  La selección de la próxima componente de la solución, la cual involucra al método *RR*, es compleja implementarla en la GPU debido a su naturaleza lineal, las estructuras de control divergentes, la necesidad de números aleatorios y la sincronización de hilos constante por lo que se propone el método alternativo *Double-Spin Roulette (DS-Roulette)*, el cual es altamente paralelo y explota el paralelismo a nivel de warps, reduce las dependencias de memoria compartida, reduce la cantidad de instrucciones totales y el trabajo realizado por la GPU. Para la construcción del tour, cada hormiga es mapeada a un bloque de hilos. Otra propuesta es dividir los hilos de los bloques dedicados a un número de ciudades válidas, aplicando así, paralelismo a nivel de warps, lo cual implícitamente garantizan que todos los hilos ejecuten sin divergencias y disminuya el tiempo de ejecución significativamente.
- Dawson and Stewart [3] es una extensión de Dawson and Stewart [4] destinado a resolver instancias grandes del problema *TSP*. Se presentan estrategias paralelas utilizando el conjunto de ciudades candidatas para la ejecución sobre GPU. Este conjunto es esencial para limitar el espacio de búsqueda y reducir el tiempo de ejecución total. Los mejores resultados obtenidos se alcanzan cuando se asocia una hormiga por bloque. Además, utilizan primitivas a nivel de warps y evitan branchs (o divergencias warps) para disminuir el *Speedup*.
- Diego et al. [6] proponen una estrategia de paralelización para resolver el *VRP*. Se implementan varios kernels destinados a: (1) la inicialización de los datos de las soluciones, (2) la construcción de las soluciones, (3) la elección de la mejor solución, (4) grabar la mejor solución, (5) la evaporación de la matriz de feromonas y (6) el depósito de feromona en dicha matriz. Cada kernel se corresponde a un único bloque. Se utiliza un único hilo para los kernels (4) y (6), una cantidad de hilos igual al número de hormigas para (2); un número de hilos igual a la mitad del número de hormigas para el kernel (3) y un número de hilos igual al número de soluciones por el número de hormigas para el kernel (1). En el trabajo no especifica en detalle porque utilizan las cantidades de hilos elegidas.

Metaheurística FWA sobre GPU. Ding et al. [7] adecuan el algoritmo original *FA* para la arquitectura GPU, resultando el algoritmo *GPU-FWA*. El algoritmo muestra un muy buen desempeño mostrando un *Speedup* superlineal cuando se lo compara a versiones secuenciales de *FA* y *PSO*; además de ser simple y escalable.

Se asigna a cada fuego un warp, en donde se genera cada chispa del mismo, de esta manera las chispas se sincronizan implícitamente reduciendo el overhead de la comunicación entre ellas.

Una explosión se implementa en un bloque, esto permite utilizar la memoria compartida. Se accede únicamente a la memoria global para obtener la posición del fuego y el valor de fitness, disminuyendo así notablemente la latencia de accesos.

Para acelerar el proceso de búsqueda se introduce el mecanismo de mutación *Attract-repulse* para alcanzar un balance entre explotación y exploración.

Para la generación de los números aleatorios de alta calidad se utiliza la librería *CURAND* de GPU.

Metaheurística CS sobre GPU. Jovanovic and Tuba [10] proponen un algoritmo paralelo sobre GPU, el cual muestra cambios significativos respecto a su versión secuencial debido a que esta última presenta varias partes que no son naturalemente paralelas. Se utiliza la paralelización en tres niveles: la reducción paralela para el cálculo de la función de fitness de cada individuo; el cálculo de los nidos del swarm realizado en un bloque; finalmente varias colonias son ejecutadas en paralelo en bloques separados.

Se asume que el algoritmo es utilizado para optimizar funciones que incluyen varias sumatorias como las funciones: *Sphere*, *Rosenbrocks valley*, *Schwefels*, etc. Los resultados muestran que el algoritmo *CS* paralelo tiene mejor desempeño respecto al secuencial en cuanto al tiempo de ejecución y a la calidad de las soluciones para algunas evaluaciones.

Metaheurística BCA sobre GPU. Luo et al. [13] presentan a CUBA, el que sería el primer algoritmo *BCA* sobre GPUs según lo mencionan sus autores. Diseñan un algoritmo *Bees Multi-Colonias*, se ejecutan varios algoritmos *Bees* en paralelo. Cada abeja es asignada a un hilo la cual se encarga de buscar una solución para su colonia. Un bloque es dividido en diferentes colonias las cuales se ejecutan independientemente. Cuando una iteración ha terminado, el algoritmo *CUBA* intercambia información entre las colonias del mismo bloque a través de la memoria compartida.

El número de abejas y de colonias depende del número de hilos por bloques y del número de bloques por grid respectivamente.

A diferencia del algoritmo secuencial que agrupa a la mayoría de las abejas en las zonas más prometedoras y las restantes en las otras zonas, el algoritmo paralelo agrupa a las abejas a lo largo de las zonas en función de la carga de trabajo de los hilos.

En el algoritmo se realizan numerosas búsquedas paralelas en diferentes zonas realizadas por numerosas colonias.

Al final de cada iteración: se utiliza el mecanismo *Odd–Even* para el ordenamiento por bloques de las soluciones en forma paralela y para proporcionar la mejor solución, las colonias utilizan un mecanismo de comunicación de dos

fases con la cual se decrementa el tiempo de convergencia. La mejor de todas las soluciones reemplazará a la peor solución de un bloque.

El algoritmo propuesto muestra un muy buen desempeño. Se analizan una amplia experimentación entre las que se consideran diferentes combinaciones paramétricas y se realiza el cálculo del *Speedup* y la *Robustez* del algoritmo paralelo propuesto.

5. Conclusiones

En este trabajo se logra un buen material de lectura para quienes están interesados en los avances del campo de investigación relacionados a la inteligencia de enjambres, ya que se presenta una incipiente revisión de algoritmos *SI* orientados a GPUs, propuestos en trabajos publicados a partir del año 2012. Abordamos una de las metaheurísticas más estudiadas en este campo como lo es *ACO*, además de otras más nuevas y/o poco exploradas como *CS, FA y BCA*.

Observamos que las *SI* abordadas pueden explotar el paralelismo masivo: aplicando estrategias para minimizar el tiempo de cómputo y de acceso a las memorias, utilizando funciones de CUDA eficientes, etc.

En general, es necesario un estudio exhaustivo de la arquitectura CPU-GPU subyacente para administrar eficientemente los recursos que ésta ofrece en función del problema a tratar y las características de la metaheurística en sí.

Finalmente, una de las líneas de investigación a profundizar, es el diseño de modelos *SI* orientados a GPUs que contemplen la minimización de la latencia de memoria, ya que la administración de la memoria es uno de los problemas centrales a la hora de diseñar algoritmos eficientes y escalables. Dicho problema se acrecenta cuando se requieren cantidades variables de memoria durante la ejecución, en función del comportamiento no determinístico y/o dinámico del algoritmo. Por ejemplo, en *ACO* la matriz de feromonas es una de las estructuras centrales a administrar, su tamaño depende del tamaño del problema, a ella tienen acceso todas las hormigas en cada iteración; en FWA, en cambio, es necesario considerar la generación dinámica y aleatoria de las chispas a partir de las explosiones del fuego.

Referencias

1. E. Alba, G. Luque, and S. Nesmachnow. Parallel Metaheuristics: Recent Advances and New Trends. *International Transactions in Operational Research*, 20(1):1–48, 2013.
2. J. M. Cecilia, A. Nisbet, M. Amos, J. M. García, and M. Ujaldón. Enhancing GPU Parallelism in Nature-inspired Algorithms. *J. Supercomput.*, 63(3):773–789, March 2013.
3. L. Dawson and I. Stewart. Candidate Set Parallelization Strategies for Ant Colony Optimization on the GPU. In Joanna Kołodziej, Beniamino Di Martino, Domenico Talia, and Kaiqi Xiong, editors, *Algorithms and Architectures for Parallel Processing*, volume 8285 of *Lecture Notes in Computer Science*, pages 216–225. Springer International Publishing, 2013.

4. L. Dawson and I. Stewart. Improving Ant Colony Optimization Performance on the GPU Using CUDA. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 1901–1908, June 2013.
5. A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki. Parallel Ant Colony Optimization on Graphics Processing Units. *Journal of Parallel and Distributed Computing*, 73(1):52–61, 2013. Metaheuristics on GPUs.
6. F. J. Diego, E. M. Gómez, M. Ortega-Mier, and Á. García-Sánchez. Parallel CUDA Architecture for Solving de VRP with ACO. In Suresh P. Sethi, Marija Bogataj, and Lorenzo Ros-McDonnell, editors, *Industrial Engineering: Innovative Networks*, pages 385–393. Springer London, 2012.
7. K. Ding, S. Zheng, and Y. Tan. A GPU-based Parallel Fireworks Algorithm for Optimization. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 9–16, New York, NY, USA, 2013. ACM.
8. M. Dorigo and C. Blum. Ant colony Optimization Theory: A Survey. *Theoretical Computer Science*, 344(2-3):243–278, 2005.
9. S. Janson, D. Merkle, and M. Middendorf. *Parallel Ant Colony Algorithms*, pages 171–201. John Wiley & Sons, Inc., 2005.
10. R. Jovanovic and M. Tuba. Parallelization of the Cuckoo Search using CUDA Architecture. *7th International Conference on Applied Mathematics, Simulation, Modelling (ASM '13), 2013*, 2013.
11. D. B. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.
12. P. Krömer, J. Platoš, and V. Snášel. Nature-Inspired Meta-Heuristics on Modern GPUs: State of the Art and Brief Survey of Selected Algorithms. *International Journal of Parallel Programming*, 42(5):681–709, 2014.
13. G. Luo, S. Huang, Y. Chang, and S. Yuan. A Parallel Bees Algorithm Implementation on GPU. *J. Syst. Archit.*, 60(3):271–279, March 2014.
14. D. Martens, T. Fawcett, and B. Baesens. Editorial Survey: Swarm Intelligence for Data Mining. *Machine Learning*, 82(1):1–42, January 2011.
15. M. Pedemonte, S. Nesmachnow, and H. Cancela. A survey on Parallel Ant Colony Optimization. *Applied Soft Computing*, 11(8):5181–5197, 2011.
16. D. T. Pham, A. Ghanbarzadeh, E. Koc, S. Otri, S. Rahim, and M. Zaidi. The Bees Algorithm, A Novel Tool for Complex Optimisation Problems. In *Proceedings of the 2nd International Virtual Conference on Intelligent Production Machines and Systems (IPROMS 2006)*, pages 454–459. Elsevier, 2006.
17. M. Randall and A. Lewis. A Parallel Implementation of Ant Colony Optimization. *J. Parallel Distrib. Comput.*, 62(9):1421–1432, 2002.
18. J. Sanders and Kandrot E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.
19. Y. Tan and Y. Zhu. Fireworks Algorithm for Optimization. In Y. Tan, Y. Shi, and K. Tan, editors, *Advances in Swarm Intelligence*, volume 6145 of *Lecture Notes in Computer Science*, pages 355–364. Springer Berlin Heidelberg, 2010.
20. X. Yang and S. Deb. Cuckoo Search Via Lévy Flights. In *Nature Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pages 210–214, Dec 2009.
21. Q. Yu, C. Chen, and Z. Pan. Parallel Genetic Algorithms on Programmable Graphics Hardware. In *Lecture Notes in Computer Science 3612*, page 1051. Springer, 2005.