TESIS DOCTORAL

Algoritmos de Optimización sobre Unidades de Proceso Gráfico

Autor

Pablo Javier Vidal

Directores

Dr. Enrique Alba Torres y Dr. Francisco Luna Valero

Departamento de Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

2015

Publicaciones y
Divulgación Científica

AUTOR: Pablo Javier Vidal

http://orcid.org/0000-0001-6502-8010

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga

Los Drs. **Enrique Alba Torres y Francisco Luna Valero**, Catedrático de Universidad y Profesor Contratado Doctor del Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga,

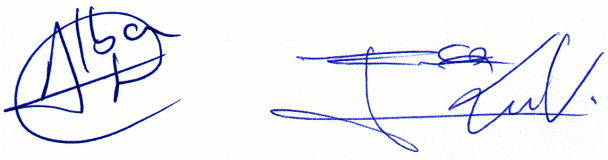
Certifican QUE

D. **Pablo Javier Vidal**, Magister en Ingeniería de Software e Inteligencia Artificial por la Universidad de Málaga, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo su dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada

Algoritmos de Optimización sobre Unidades de Proceso Gráfico.

Revisado el presente trabajo, estimamos que puede ser presentado al tribunal que ha de juzgarlo, y autorizamos la presentación de esta Tesis Doctoral en la Universidad de Málaga.

En Málaga, Diciembre de 2015.



Firmado: Dr. Enrique Alba Torres y Dr. Francisco Luna Valero
Catedrático de Universidad y Profesor Contratado Doctor
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga

Agradecimientos

En primer lugar deseo expresar mi agradecimiento a los directores de esta tesis doctoral. Al Dr. Enrique Alba Torres, por haberme recibido y compartido parte de su inmenso conocimiento conmigo. Al Dr. Francisco Luna Valero, por la dedicación y apoyo que ha brindado a este trabajo y por haberme soportado en los diferentes momentos que han transcurrido durante el transcurso de este doctorado. A ambos, por el respeto a mis sugerencias e ideas y por la dirección y el rigor que ha facilitado a las mismas. Gracias por la confianza ofrecida desde que comencé este camino.

Asimismo, agradezco a mis compañeros y amigos del LabTEm y NEO por su apoyo personal y humano. En particular a Marta Lasso y a Daniel Pandolfi, ya que sin ellos no podría haber concretado el doctorado en la Universidad de Málaga.

También debo agradecer a la Universidad Nacional de la Patagonia Austral, por su continuo apoyo durante mi estudio doctoral. También agradezco a la Universidad de Málaga y al proyecto TIN2014-57341-R (moveON).

Gracias a mis amigos del alma, que siempre me han prestado un gran apoyo moral y humano, necesarios en los momentos difíciles de este trabajo y esta profesión.

Gracias a mis padres, por apoyarme en todas las decisiones que he tomado a lo largo de la vida, hayan sido buenas o malas, y especialmente por enseñarme a luchar por lo que quiero y a terminar lo que he empezado.

Pero mi mayor agradecimiento se lo debo, a mi esposa, Ana Carolina Olivera, por su paciencia, comprensión y solidaridad con este proyecto, por el tiempo que me ha concedido y soportado, un tiempo robado a nuestra vida familiar. Sin su apoyo este trabajo nunca se habría escrito y, por eso, este trabajo es también suyo.

Con amor y agradecimientos infinitos solo queda por decir,
Gracias Totales.

En mi cabeza tengo cuatro botoncitos.
El botón de no escuchar,
el de callarse la boca, el de desaparecer
y el botón de ataque. Con eso resuelvo todas las situaciones.
Pappo

Tabla de Contenidos

Página

Capítulo 1

Introducción

1.1. Planteamiento

El diseño de algoritmos cada vez más eficientes para resolver problemas complejos ha sido tradicionalmente uno de los aspectos más importantes de la investigación en Informática. El objetivo perseguido en este campo es, fundamentalmente, el desarrollo de nuevos métodos capaces de resolver problemas complejos con el menor esfuerzo computacional posible, mejorando así a los algoritmos existentes. En consecuencia, esto no sólo permite afrontar problemas actuales de forma más eficiente, sino también procesos vedados en el pasado debido a su alto coste computacional. En este contexto, la actividad investigadora tanto en algoritmos exactos [103, 137] como en heurísticos *ad hoc* y metaheurísticos [48, 99, 119] para resolver problemas de optimización está creciendo de forma evidente en estos días debido a la al progreso en el contexto tecnológico [63, 128].

La evolución de la tecnología *hardware* y *software* en los últimos años ha logrado demostrar la importancia de la paralelización de los algoritmos [113]. Este crecimiento ha encontrado por parte de los fabricantes y diseñadores límites físicos que han sido resueltos, entre otras formas, implementando *Unidades de Proceso de Cálculo* (CPU) con varios núcleos en un mismo chip [13].

En la actualidad, han emergido nuevas plataformas de cómputo paralelo tales como las *Unidades de Procesamiento Gráfico* (GPUs, por sus siglas en inglés). Las GPUs están especialmente diseñadas para proporcionar un entorno donde es posible explotar su modelo de paralelismo y desarrollar diferentes propuestas algorítmicas paralelas para este tipo de plataformas. En este sentido, las GPUs se han convertido en una alternativa interesante de cómputo a las tradicionales CPU [57, 67], dada su capacidad para realizar cálculos en paralelo y también diferenciadas de

otras formas de usar múltiples procesadores, como los clusters de máquinas [7], los sistemas de memoria compartida *manycore* [100], o las *Field Programmable Gate Arrays* (FPGAs) [29]. Por ello, las GPUs han dejado de ser procesadores exclusivos para el procesamiento de gráficos y se han convertido en sofisticados co-procesadores de bajo costo y de alto rendimiento. Esto ha permitido a una computadora de escritorio o portátil aumentar su capacidad de cómputo debido a las mejoras de las tarjetas a nivel hardware y software, lo que las convierte en una opción para realizar cómputo de alto rendimiento, *High Performance Computing* (HPC, por sus siglas en inglés).

El desarrollo de estrategias metaheurísticas paralelas en GPU para la resolución de problemas complejos se presenta como una línea de investigación muy importante en la actualidad, ya que por una parte, las citadas estrategias (potencialmente no exactas) proporcionan una solución de alta calidad en un tiempo razonable, mientras que por otro, es bien sabido que tanto la eficiencia de dichos algoritmos heurísticos como los resultados que éstos obtienen mejoran de manera significativa cuando se aplican estrategias de paralelismo [2, 6].

El contenido de este trabajo se centra en evaluar técnicas metaheurísticas en arquitecturas GPU buscando la mejor técnica que se adapte a la GPU. De la misma forma, se busca diseñar y analizar nuevos modelos que puedan proveer: sencillez en la implementación y potencia computacional, equilibrio necesario entre la eficiencia y la eficacia suficiente para la supervivencia en diferentes problemas. Todo esto por varias razones: en primer lugar, las ventajas de estas nuevas tecnologías ofrece la posibilidad de cubrir la demanda de eficiencia tanto en tiempo real como numérica, lo cual nos lleva a la segunda razón. Las metaheurísticas son técnicas ampliamente aceptadas y utilizadas en diferentes áreas de aplicación. Cabe la posibilidad entonces de brindar al investigador una solución de cierta calidad en un breve periodo de tiempo: un compromiso entre calidad de la solución y la rapidez. Finalmente, muchas de las contribuciones que se materialicen pueden ser extensibles a otras técnicas metaheurísticas o ser usadas como base para otros modelos algorítmicos en futuros estudios posteriores a esta tesis.

Para empezar, un estudio canónico del comportamiento paralelo sobre GPU presenta múltiples ventajas, tanto en el terreno teórico como en el práctico. La utilización de modelos paralelos está notablemente avalada por un conjunto muy numeroso de aplicaciones a problemas reales y estudios sobre su comportamiento [2]. Sin embargo, es manifiesta la falta de uniformidad de estos modelos paralelos que, de existir, ayudaría a crear una metodología y extracción de características comunes que permitieran estimar su comportamiento en otros problemas.

En consecuencia, es importante la presentación de los distintos modelos paralelos como subclases de una formalización común de algoritmo paralelo que permita una comparación de sus componentes, una identificación de similitudes/diferencias, y una transferencia de conceptos y a veces de resultados teóricos entre ellos. Esta visión permite explicar, clasificar y estudiar

algoritmos existentes así como derivar nuevos algoritmos de comportamiento más predecibles.

En particular, la transformación parcial o completa de modelos secuenciales a paralelos a veces no implica mejoras significativas por el simple hecho de paralelizar. Algunas de ellas simplemente no pueden sacarle provecho a una GPU [32, 109]. Este es un vacío de conocimiento que pretendemos cubrir mediante la implementación de una nueva técnica de optimización basada en conceptos arquitectónicos del pasado que han probado su valía anteriormente en investigación. En este sentido, las características de la *computación sistólica* [76, 77] se presenta como una buena base para modelar y crear un modelo de optimización que se beneficie de las cualidades de una GPU. Éste es uno de los objetivos de la presente tesis, donde se intenta dar un marco de trabajo para derivar algoritmos especialmente buenos cuando se ejecutan en GPU, con un posterior uso de este marco general para presentar algoritmos concretos que resuelven problemas concretos.

Para terminar este planteamiento inicial resta añadir algunas consideraciones sobre la implementación de algoritmos paralelos sobre GPUs. En este caso, y debido a la naturaleza estocástica de los sistemas que manejamos, los algoritmos deben ejecutarse hasta alcanzar soluciones de adecuación similares teniendo en cuenta el esfuerzo y el tiempo consumido tomado como cota de ejecución. Estos son criterios justos y realistas para la comparación entre modelos secuenciales y paralelos [2]. En consecuencia, resulta imperativo un estudio sobre problemas de complejidad no trivial y tan complejos como sea posible para poner de manifiesto los puntos fuertes y débiles de cada modelo y obtener conclusiones correctas con una seguridad y rigurosidad razonable.

En concreto, este trabajo se basa en explorar nuevas propuestas de metaheurísticas puras, híbridas, paralelas sobre GPU, con diferente distribución de población, para mejorar los resultados existentes en problemas pertenecientes a los dominios académico e industrial. Todos los nuevos algoritmos propuestos en este trabajo son comparados con diferentes algoritmos canónicos y con aquellos pertenecientes al estado del arte para una plétora de problemas complejos de optimización que pertenecen a los campos de optimización combinatoria y continua.

1.2. Objetivos

El objetivo global de esta tesis doctoral es proporcionar un estudio significativo de nuevos modelos metaheurísticos útiles para cubrir las necesidades de un procesamiento paralelo eficiente y genéricamente útil en optimización/búsqueda sobre Unidades de Proceso Gráfico. Asimismo, estos modelos deben servir como base para proponer nuevos que presenten una mayor eficiencia y que hagan uso pleno de los recursos de la GPU. Queremos igualmente analizar los resultados para comprender el comportamiento de estos algoritmos y proponer nuevos métodos capaces de resolver los problemas de manera más eficaz y eficiente. Este objetivo global guía nuestras

decisiones en la tesis y nos permite hacer un uso del Método Científico [65, 135] para su desarrollo. Este desarrollo nos lleva de forma natural a considerar los siguiente sub-objetivos particulares:

1. Descripción unificada de las clases de metaheurísticas paralelas existentes e implementadas sobre una arquitectura de tipo GPU.
2. Construcción y evaluación de un modelo paralelo basado en un Algoritmo Genético Celular (como caso de algoritmo de inteligencia de enjambre, con particularidades cercanas a la arquitectura GPU) para una arquitectura de una y múltiples GPUs.
3. Estudio de las prestaciones de estos modelos con población única y con estructura subdividida. Análisis de desempeño mediante la evaluación de problemas simples incrementando progresivamente la complejidad en los dominios discreto y continuo.
4. Diseño de un modelo de optimización basado en conceptos de *Computación Sistólica* para GPUs. Estudio de variaciones en su modelo canónico evaluando estructuras de comunicación y perturbación sobre las soluciones, conducentes a mejoras sobre el modelo original.
5. Hibridación mediante un paradigma de optimización basado en la explotación sistemática de los recursos existentes en las dos arquitecturas, CPU y GPU respectivamente. Determinación de las estrategias y combinación de metaheurísticas (o componentes) para maximizar el esfuerzo y aprovechamiento de las arquitecturas anteriormente mencionadas.

1.3. Contribuciones

En este apartado se listan las principales contribuciones que se han incluido a lo largo de esta memoria; las mismas se pueden resumir como sigue:

1. *Estudio y clasificación de los modelos paralelos de metaheurísticas existentes sobre Unidades de Proceso Gráfico.* Para ello partimos de la clasificación proporcionada en [72, 114, 116] e introducimos algunos nuevos modelos y técnicas híbridas incluyendo un mayor número de detalles. Como aportaciones originales se incluyen los resúmenes ordenados por año y los detalles de los modelos paralelos, de forma similar a [9, 72].
2. *Estudio unificado de los algoritmos Genéticos Celulares (cGAs, por sus siglas en inglés) y sus operaciones en GPUs.* Para ello pretendemos extender algunos estudios previos [2, 3] sobre cGAs paralelos al campo de las GPUs.
3. *Propuesta de un modelo genético celular para una arquitectura con una sola y múltiples GPUs.* Análisis de modelado asíncrono y síncrono de acuerdo a lo expresado en [4]. Estudio sobre un conjunto de casos representativos de dominio discreto y continuo.

4. *Estudio práctico de la influencia de características intrínsecas de la plataforma y modelo de programación de la GPU sobre un cGA*. Las aportaciones se basan en considerar algunas funciones de cómputo propias del modelo y estudiar el impacto de almacenar en distintas unidades de memoria de la GPU y cómo afecta esto al modelo canónico del cGA.

5. *Caracterización formal unificada de la computación sistólica y su relación con modelos algorítmicos de optimización paralelos.* Se trata de brindar una descripción detallada del uso de este tipo de arquitectura computacional [76, 77], su utilidad en el pasado [15, 59] y someros acercamientos en el campo de la optimización [20, 86].

6. *Propuesta y estudio detallado de un modelo de búsqueda sistólica para problemas de optimización.* Realizamos una aportación nueva en este sentido y derivamos nuevos modelos paralelos trabajando en diversos aspectos del modelo canónico de una *búsqueda sistólica por vecindario* basado en el funcionamiento de una arquitectura sistólica tomando las bondades de la misma como base: la organización de los datos, la estructura de comunicación, las operaciones de perturbación sobre cada elemento perteneciente a esa organización tanto a nivel espacial como a nivel local [76].

7. *Análisis sobre la influencia de decisiones de implementación en sistemas paralelos de algoritmos de optimización.* Este análisis se enfocará sobre todo en aquellos modelos que utilizan múltiples GPUs o variación en el uso de los tipos de memoria existentes en la GPU. De la misma forma, la comunicación de los elementos en un algoritmo y entre las diversas arquitecturas (CPU y/o múltiples GPUs) no son características que sean a menudo presentadas conjuntamente en la literatura [6].

8. *Estudio de hibridación paralela mediante la ejecución de una búsqueda sistólica sobre GPU y otras metaheurísticas sobre CPU* buscando maximizar el uso de los recursos hardware. Demostración de mejoras de eficiencia en término de esfuerzo de evaluación y en tiempo real. Asimismo, la capacidad heterogénea paralela híbrida presentada a nivel de búsqueda es distintiva y nueva. En [84, 127] podemos encontrar aproximaciones similares usando de forma muy simple los recursos de la CPU.

1.4. Organización de la tesis

Esta memoria de tesis se estructura en cuatro partes, cada una con varios capítulos. En la primera se presentan los conceptos generales sobre las metaheurísticas, la arquitectura de la GPU y los aspectos relacionados con la optimización sobre GPUs. Finalmente, se detalla un extenso estado del arte sobre las principales aportaciones que existen en la literatura en dicho campo, y

una crítica a dicho estado del arte, poniendo de manifiesto los problemas con los que cuentan las técnicas actuales.

En la segunda parte se describen las propuestas metodológicas originales de esta tesis y los detalles de las mismas, que han sido desarrolladas con el fin de solventar parte de los aspectos criticados en la primer parte.

La tercera parte contiene los resultados de los estudios realizados para las diversas aproximaciones presentadas anteriormente ante problemas progresivamente más complejos. Finalmente, en la cuarta parte mostramos las principales conclusiones alcanzadas, así como las líneas de trabajo futuro inmediatas que surgen de esta memoria. A continuación describimos detalladamente el contenido de los capítulos que podemos encontrar agrupados en estas cuatro partes.

- **Parte I: Fundamentos**

  El Capítulo 2 da una visión global sobre optimización utilizando metaheurísticas. Posteriormente, se dan detalles sobre los paradigmas existentes, enfocándonos finalmente en los modelos paralelos de metaheurísticas.

  El Capítulo 3 presenta los aspectos generales de la arquitectura de una GPU, el modelo de flujo de control y de control de memoria. Después se muestran algunas cuestiones concernientes con los diferentes desafíos a la hora de diseñar y utilizar metaheurísticas sobre una arquitectura de GPU. Finalmente, nos enfocamos en el estudio del estado del arte de los algoritmos evolutivos sobre plataformas GPU, presentando un amplio resumen de las principales publicaciones previas que han aparecido en la literatura estudiando este tipo de aproximaciones.

  El Capítulo 4 proporciona una introducción genérica sobre los problemas abordados durante todo el desarrollo de la tesis, así como una justificación de su elección. Posteriormente describe la evaluación de los resultados, los indicadores de rendimiento utilizados y finalmente la descripción de los elementos hardware usados en la presente tesis.

- **Parte II: Propuestas algorítmicas**

  El Capítulo 5 presenta las propuestas metodológicas: la primera propuesta se basa en el algoritmo Genético Celular enfocándose en el control eficiente de este modelo utilizando paralelismo. Asimismo, se presenta una aproximación para el caso de una arquitectura trabajando sobre múltiples GPUs (multi-GPU). A continuación se expone un nuevo modelo de optimización llamado *Búsqueda Sistólica de Vecindario* (*SNS*, por sus siglas en inglés *Systolic Neighborhood Search*) así como sucesivas extensiones desarrolladas especialmente para obtener una mejora en el rendimiento del algoritmo canónico. Ofrecemos extensiones

avanzadas en dos sentidos: primero, para resolver problemas de optimización y, segundo, para que se puedan ejecutar especialmente bien en una arquitectura GPU. Finalmente, este capítulo discute las características de un algoritmo de optimización híbrido buscando maximizar el uso del paradigma paralelo de cooperación entre CPU y GPU. En este sentido, estudiamos la mejor manera de transferir información entre las dos arquitecturas y el impacto que esto tiene sobre el rendimiento de la nueva técnica.

El Capítulo 6 presenta los resultados obtenidos por la implementación en GPU y multi-GPU del algoritmo Genético Celular. Se resuelven distintos problemas e instancias referentes a los dominios discretos y continuos. Se realiza una discusión de los resultados para llegar a una comprensión profunda del diseño.

El Capítulo 7 muestra todos aquellos resultados obtenidos con respecto a la propuesta sistólica SNS y sus múltiples variaciones diseñadas ilustradas con el problema de la mochila multidimensional. Se resuelven distintas instancias del problema con diferentes características y se discuten los resultados enfatizando aquellas características de los modelos desarrollados que puedan proporcionar mejoras significativas.

El Capítulo 8 estudia los resultados obtenidos tras la aplicación de una nueva técnica híbrida de optimización la cual utiliza como base el comportamiento del SNS y busca aprovechar la CPU mediante el uso de una algoritmo micro-Genético (μGA). Esta técnica híbrida es aplicada a tres problemas de optimización combinatoria; el Problema de la Mochila, el Problema Deceptivo Masivamente Multimodal y el Problema de Suma de Subconjuntos. Se analizan los componentes en su aproximación y por separado y se comparan los resultados entre ellos y con la literatura.

- **Parte III: Conclusiones y trabajo futuro**

  Terminamos esta tesis con dos capítulos. El Capítulo 9 presenta las conclusiones sobre todo lo expuesto en el resto del documento. El Capítulo 10 describe las principales líneas de trabajo futuro que surgen del presente estudio.

- **Parte IV: Apéndices**

  El Apéndice A ofrece una justificación del interés y del respaldo internacional al trabajo contenido en esta tesis a través de una justificación de los contenidos y las publicaciones científicas de impacto del doctorando realizadas durante la elaboración de la tesis.

Parte

1

Fundamentos de las metaheurísticas y arquitecturas paralelas sobre Unidades de Proceso Gráfico

Capítulo 2

Metaheurísticas

Este capítulo está dedicado a establecer los fundamentos necesarios sobre el contexto de optimización y los algoritmos utilizados en este ámbito como base de las aproximaciones algorítmicas realizadas en GPU en este trabajo de tesis. Partiremos de un planteamiento clásico de problema de optimización para definir luego el concepto de metaheurística y discutir una taxonomía de las diferentes familias de estos algoritmos. A continuación se introducirán conceptos de optimización paralela como un medio para reducir los tiempos de ejecución, mostrando una visión general de los modelos paralelos existentes en metaheurísticas para la acelerar los distintos procesos de cómputo que se pueden encontrar dentro de cada uno de ellos.

2.1. Contexto de optimización

Un problema de optimización se conoce como la optimización (maximizar o minimizar), una función de *fitness* (monoobjetivo), o de un conjunto de funciones de *fitness* (multiobjetivo). Podemos definir un *problema de optimización* como sigue:

Definición 1 (Problema de optimización)**.** *Un problema de optimización se formaliza como un par* (S, f)*, donde* $S \neq \emptyset$ *representa el espacio de soluciones (o de búsqueda) del problema, mientras que* f *es una función denominada* función objetivo *o* función de fitness*, que se define como:*

$$f : S \rightarrow \mathbb{R} \ . \tag{2.1}$$

Así, resolver un problema de optimización consiste en encontrar una solución, $i^* \in S$*, que satisfaga la siguiente desigualdad:*

$$f(i^*) \leq f(i), \qquad \forall\ i \in S\ . \tag{2.2}$$

Asumir el caso de maximización o minimización no restringe la generalidad de los resultados, puesto que se puede establecer una igualdad entre tipos de problemas de maximización y minimización de la siguiente forma [10, 51]:

$$\text{máx}\{f(i)|i \in S\} \equiv \text{mín}\{-f(i)|i \in S\}\ . \tag{2.3}$$

En función del dominio al que pertenezca S, podemos definir problemas de *optimización binaria* ($S \subseteq \mathbb{B}^*$), *entera* ($S \subseteq \mathbb{N}^*$), *continua*($S \subseteq \mathbb{R}^*$), o *mixta* ($S \subseteq (\mathbb{B} \cup \mathbb{N} \cup \mathbb{R})^*$). La codificación debe ser adecuada y relevante para el problema de optimización en cuestión. Por otra parte, la calidad de una representación tiene una influencia considerable en la eficiencia de las técnicas aplicadas en esta representación.

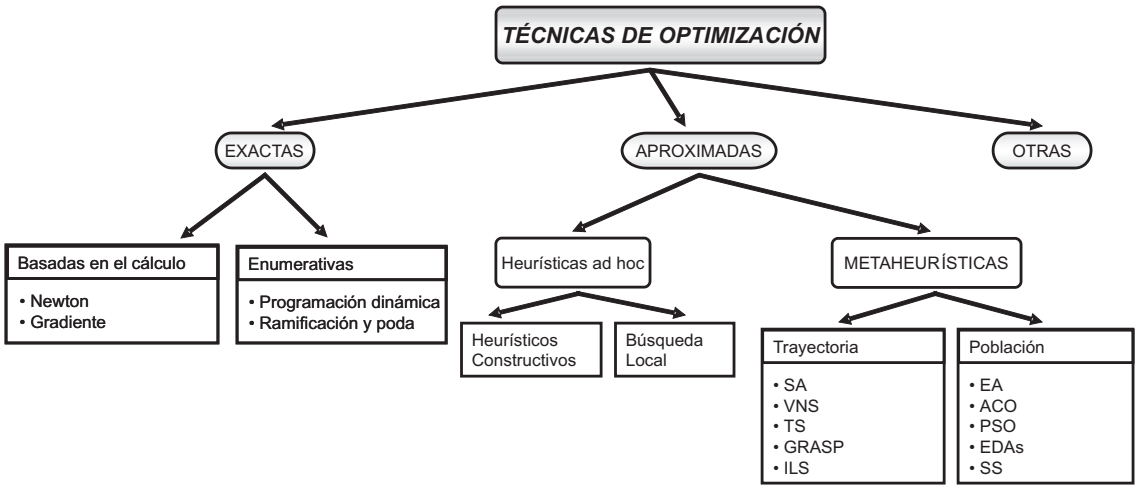


Figura 2.1: Clasificación de las técnicas de optimización.

Debido a la gran importancia de los problemas de optimización, a lo largo de la historia de la Informática se han desarrollado múltiples métodos para tratar de resolverlos. Una clasificación muy simple de estos métodos se muestra en la Figura 2.1. Inicialmente, las técnicas las podemos clasificar en exactas (o enumerativas, exhaustivas, etc.) y aproximadas. Las técnicas exactas garantizan encontrar la solución óptima para cualquier instancia de cualquier problema en un tiempo acotado. El inconveniente de estos métodos es que el tiempo y/o memoria que se necesitan, aunque acotados, crecen exponencialmente con el tamaño del problema, ya que la mayoría de éstos son NP-duros. Esto supone en muchos casos que el uso de estas técnicas sea inviable, ya que

se requiere mucho tiempo (posiblemente miles de años) y/o una cantidad desorbitada de memoria para la resolución del problema. Por lo tanto, los algoritmos aproximados para resolver estos problemas están recibiendo una atención cada vez mayor por parte de la comunidad internacional desde hace unas décadas. Estos métodos sacrifican la garantía de encontrar el óptimo a cambio de encontrar una solución satisfactoria en un tiempo razonable.

Dentro de los algoritmos aproximados se pueden encontrar dos tipos: los heurísticos *ad hoc* y las metaheurísticas (en las que nos centramos en este capítulo). Los heurísticos *ad hoc* permiten encontrar soluciones de una calidad "buena" para problemas de gran tamaño, permitiendo obtener un rendimiento aceptable a costos aceptables en una amplia gama de problemas. No obstante, este tipo de algoritmos no tienen una garantía de aproximación de las soluciones óptimas, si bien están adaptados y diseñados para resolver un problema específico o / y de la instancia. Por otro lado, las metaheurísticas son algoritmos de propósito general que pueden ser aplicadas para resolver casi cualquier problema de optimización.

Las técnicas denominadas *metaheurísticas* surgieron en los años setenta como una nueva clase de algoritmos aproximados, cuya idea básica era combinar diferentes métodos heurísticos a un nivel más alto para conseguir una exploración del espacio de búsqueda de forma eficiente y efectiva. Este término fue introducido por primera vez por Glover [46]. Esta clase de algoritmos incluye técnicas como colonias de hormigas, algoritmos evolutivos, búsqueda local iterada, enfriamiento simulado y búsqueda tabú, entre otras. Se pueden encontrar revisiones de metaheurísticas en [14, 50].

Una metaheurística es una estrategia de alto nivel que usa diferentes métodos para explorar el espacio de búsqueda [21, 85]. En otras palabras, una metaheurística es una plantilla general no determinista que debe ser rellenada con datos específicos del problema (representación de las soluciones, operadores para manipularlas, etc.) y que permiten abordar problemas con espacios de búsqueda de gran tamaño. En este tipo de técnicas es especialmente importante el correcto equilibrio (generalmente dinámico) que haya entre *diversificación* e *intensificación*. El término diversificación se refiere a la evaluación de soluciones en regiones distantes del espacio de búsqueda (de acuerdo a una distancia previamente definida entre soluciones); también se conoce como *exploración* del espacio de búsqueda. El término intensificación, por otro lado, se refiere a la evaluación de soluciones en regiones acotadas y pequeñas con respecto al espacio de búsqueda centradas en el vecindario de soluciones concretas (*explotación* del espacio de búsqueda). El equilibrio entre estos dos aspectos contrapuestos es de gran importancia, ya que por un lado deben identificarse rápidamente las regiones prometedoras del espacio de búsqueda global y por otro lado no se debe malgastar tiempo en las regiones que ya han sido exploradas o que no contienen soluciones de alta calidad.

2.2. Paradigmas principales de metaheurísticas

Hay diferentes formas de clasificar y describir las metaheurísticas [14]. Dependiendo de las características que se seleccionen se pueden obtener diferentes taxonomías: basadas en la naturaleza y no basadas en la naturaleza, con memoria o sin ella, con una o varias estructuras de vecindario, etc. Una de las clasificaciones más populares las divide en metaheurísticas *basadas en trayectoria* y *basadas en población*. Las primeras manipulan en cada paso un único elemento del espacio de búsqueda, mientras que las segundas trabajan sobre un conjunto de ellos (población). Esta taxonomía se muestra de forma gráfica en la Figura 2.1, que además incluye las técnicas más representativas. Estas clasificación es descrita en las dos secciones siguientes.

2.2.1. Metaheurísticas basadas en trayectoria

La mayoría de estos algoritmos parten de una solución y, mediante la exploración del vecindario, van actualizando la solución actual, formando una trayectoria. El vecindario de una solución s, que denotamos con $N(s)$, es el conjunto de soluciones que se pueden construir a partir de s aplicando un operador específico de modificación (generalmente denominado *movimiento*). Estos algoritmos surgen como extensiones de los métodos simples *búsqueda local* (*Local Search* o LS) a los que se les añade algún mecanismo para escapar de los mínimos locales. Esto implica la necesidad de una condición de parada más elaborada que la de encontrar un mínimo local. Normalmente la búsqueda termina cuando se alcanza un número máximo predefinido de iteraciones, se encuentra una solución con una calidad aceptable, o se detecta un estancamiento del proceso. Por otra parte, las características más importantes en esta clase de metaheurísticas radica en la definición de la función utilizada para seleccionar las soluciones que serán manipuladas en la siguiente iteración y en la actualización de las variables de estado.

2.2.2. Metaheurísticas basadas en población

Los métodos basados en población se caracterizan por trabajar con un conjunto de soluciones, usualmente denominado población, en cada iteración, a diferencia de los métodos basados en trayectoria, que únicamente manipulan una solución del espacio de búsqueda por iteración. El resultado final proporcionado por este tipo de algoritmos depende fuertemente de la forma en que manipula la población

2.2.3. Aproximaciones cooperativas híbridas

En los últimos años, el interés en las metaheurísticas que cooperen de forma híbridas ha aumentado considerablemente en el área de optimización. Los mejores resultados encontrados

en muchos problemas de optimización clásicos o de la vida real son obtenidos por algoritmos híbridos [121]. Cuatro diferentes tipos de combinaciones se pueden considerar:

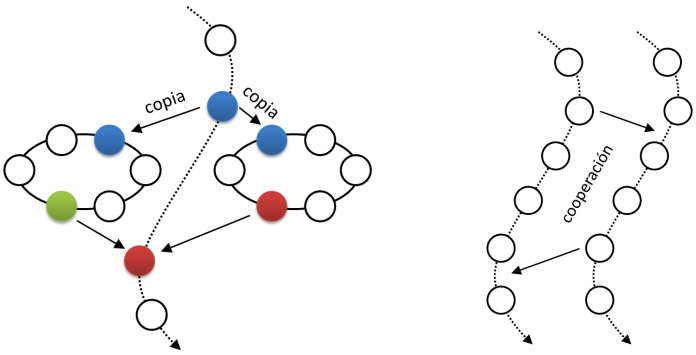
- Combinar dos o más metaheurísticas diferentes.
- Combinar metaheurísticas con métodos exactos procedentes de la programación matemática, los cuales son ampliamente usados en investigación operativa.
- Combinar metaheurísticas con métodos provenientes de la programación con restricciones desarrollados en la comunidad de la inteligencia artificial.
- Combinar metaheurísticas con técnicas de aprendizaje automático y de minería de datos.

Combinaciones de algoritmos tales como metaheurísticas basadas en población o/y en trayectoria, programación matemática, programación con restricciones y técnicas de aprendizaje automático (*machine learning*) proveen algoritmos de búsqueda muy poderosos.

2.3. Metaheurísticas paralelas

Aunque el uso de metaheurísticas permite reducir significativamente la complejidad temporal del proceso de búsqueda, este tiempo puede seguir siendo muy elevado en algunos problemas de interés real (alta dimensionalidad, funciones objetivo computacionalmente costosas, etc.). Con la proliferación de plataformas paralelas de cómputo eficientes, la implementación paralela de estas metaheurísticas surge de forma natural como una alternativa para acelerar la obtención de soluciones precisas a estos problemas. La literatura es muy extensa en cuanto a la paralelización de metaheurísticas se refiere [2, 28, 30, 82] ya que se trata de una aproximación que puede ayudar no sólo a reducir el tiempo de cómputo, sino a producir también una mejora en la calidad de las soluciones encontradas. Esta mejora está basada en un nuevo modelo de búsqueda que alcanza un mejor balance entre intensificación y diversificación. Tanto para las metaheurísticas basadas en trayectoria como para las basadas en población se han propuesto modelos paralelos acorde a sus características. Las siguientes secciones 2.3.1 y 2.3.2 presentan, respectivamente, generalidades relacionadas con la paralelización de cada tipo de metaheurística.

En nuestro contexto, donde vamos a considerar problemas del mundo real, la utilización no sólo de modelos paralelos, sino también de plataformas paralelas es casi obligatorio, ya que una simple evaluación del problema puede llevar minutos, horas, días e incluso meses si se consideran problemas en los que intervienen, por ejemplo, complejas simulaciones. La Sección 2.3.3 está dedicada a definir las diferentes plataformas existentes hasta el momento, así como la introducción de aspectos de diseño claves para el despliegue de metaheurísticas en este tipo de plataformas de cómputo.



a Modelo paralelo de evaluación sobre múltiples copias.

b Modelo paralelo con cooperación.

Figura 2.2: A la izquierda se muestra el modelo de movimientos paralelos, donde se hace una exploración paralela del vecindario. A la derecha, se detalla el modelo de ejecuciones múltiples con cooperación, donde hay varios métodos ejecutándose en paralelo y cooperando entre ellos.

2.3.1. Modelos paralelos para métodos basados en trayectoria

Los modelos paralelos de metaheurísticas basadas en trayectoria encontrados en la literatura se pueden clasificar, generalmente, dentro de tres posibles esquemas: ejecución en paralelo de varios métodos (*modelo de múltiples ejecuciones*), exploración en paralelo del vecindario (*modelo de movimientos paralelos*), y cálculo en paralelo de la función de *fitness* (*modelo de aceleración del movimiento*). A continuación detallamos cada uno de ellos.

- **Modelo de múltiples ejecuciones:** consiste en ejecutar en paralelo varios subalgoritmos ya sean homogéneos o heterogéneos [5, 81]. En general, cada subalgoritmo comienza con una solución inicial diferente. Se pueden distinguir diferentes casos dependiendo de si los subalgoritmos colaboran entre sí o no. El caso donde las ejecuciones son totalmente independientes se usa ampliamente porque es simple de utilizar y muy natural (como aparece en la Figura 2.2a). En este caso, la semántica del modelo es la misma que la de la ejecución secuencial, ya que no existe cooperación. El único beneficio al utilizar este modelo respecto a realizar las ejecuciones en una única máquina es la reducción del tiempo de ejecución total.

  Por otro lado, en el caso cooperativo (véase el ejemplo de la derecha de la Figura 2.2b), los diferentes subalgoritmos intercambian información durante la ejecución. En este caso el comportamiento global del algoritmo paralelo es diferente al secuencial y su rendimiento se ve afectado por cómo esté configurado este intercambio. El usuario debe fijar ciertos parámetros para completar el modelo: qué información se intercambian, cada cuánto se pasan la información y cómo se realiza este intercambio. La información intercambiada suele ser la mejor solución en-

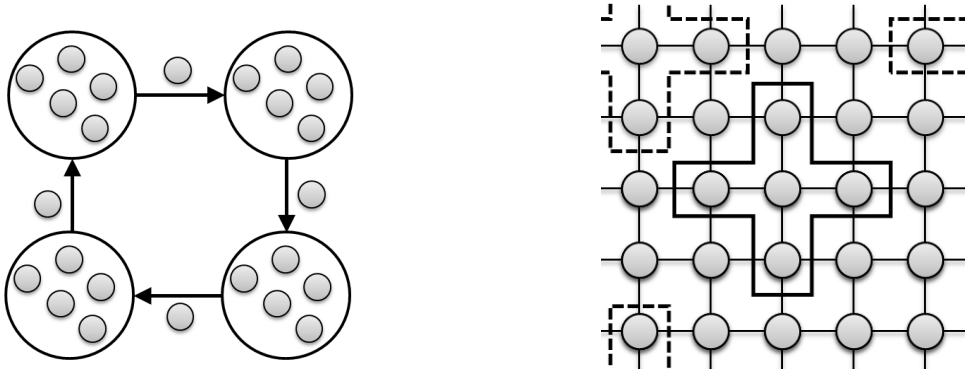
contrada, los movimientos realizados o algún tipo de información sobre la trayectoria realizada. En cualquier caso, esta información no debe ser abundante para que el coste de la comunicación no sea excesivo e influya negativamente en la eficiencia. También se debe fijar cada cuántos pasos del algoritmo se intercambia la información. Para elegir este valor hay que tener en cuenta que el intercambio no sea muy frecuente, para que el coste de la comunicación no sea perjudicial, ni muy poco frecuente, para que el intercambio tenga algún efecto en el comportamiento global. Por último, se debe indicar si las comunicaciones se realizarán de forma asíncrona o síncrona. En el caso síncrono, los subalgoritmos, cuando llegan a la fase de comunicación, se detienen hasta que todos ellos llegan a este paso y sólo entonces se realiza la comunicación. En el caso asíncrono, que es el más usado, cada subalgoritmo realiza el intercambio sin esperar al resto cuando llega al paso de comunicación.

- **Modelo de movimientos paralelos:** los métodos basados en trayectoria en cada paso examinan parte de su vecindario y, de él, eligen la siguiente solución a considerar. Este paso suele ser computacionalmente costoso, ya que examinar el vecindario implica múltiples cálculos de la función de *fitness*. El modelo de movimientos paralelos tiene como objetivo acelerar dicho proceso mediante la exploración en paralelo del vecindario (véase el esquema de la izquierda de la Figura 2.2a). Siguiendo un modelo maestro-esclavo, el maestro (el que ejecuta el algoritmo) pasa a cada esclavo la solución actual. Cada esclavo explora parte del vecindario de esta solución devolviendo la más prometedora. Entre todas estas soluciones devueltas el maestro elige una para continuar el proceso. Este modelo no cambia la semántica del algoritmo, sino que simplemente acelera su ejecución en caso de ser lanzado en una plataforma paralela. Este modelo es bastante popular debido a su simplicidad.

- **Modelo de aceleración del movimiento:** en muchos casos, el proceso más costoso del algoritmo es el cálculo de la función de *fitness*. Este cálculo, en muchos problemas, se puede descomponer en varios cómputos independientes más simples que, una vez llevados a cabo, se pueden combinar para obtener el valor final de la función de *fitness*. En este modelo, cada uno de esos cálculos más simples se asignan a los diferentes procesadores y se realizan en paralelo, acelerando el cálculo total. Al igual que el anterior, este modelo tampoco modifica la semántica del algoritmo respecto a su ejecución secuencial.

2.3.2. Modelos paralelos para métodos basados en población

El paralelismo surge de manera natural cuando se trabaja con poblaciones, ya que cada individuo puede manejarse de forma independiente. Debido a esto, el rendimiento de los algoritmos basados en población suele mejorar bastante cuando se ejecutan en paralelo. A alto nivel podemos



a Modelo paralelo distribuido. b Modelo paralelo celular.

Figura 2.3: Los dos modelos más populares para estructurar la población: a la izquierda el modelo distribuido y a la derecha el modelo celular.

dividir las estrategias de paralelización de este tipo de métodos en dos categorías: (1) paralelización del cómputo, donde las operaciones que se llevan a cabo sobre los individuos son ejecutadas en paralelo, y (2) paralelización de la población, donde se procede a la estructuración de la población.

- **Modelo maestro-esclavo:** en este esquema, un proceso central realiza las operaciones que afectan a toda la población (como, por ejemplo, la selección en los algoritmos evolutivos) mientras que los procesos esclavos se encargan de las operaciones que afectan a las soluciones independientemente (como la evaluación de la función de *fitness*, u operadores genéticos). Con este modelo, la semántica del algoritmo paralelo no cambia respecto al secuencial pero el tiempo global de cómputo es reducido. Este tipo de estrategias son muy utilizadas en las situaciones donde el cálculo de la función de *fitness* es un proceso muy costoso en tiempo. Otra estrategia muy popular es la de acelerar el cómputo mediante la realización de múltiples ejecuciones independientes (sin ninguna interacción entre ellas) usando múltiples máquinas para, finalmente, quedarse con la mejor solución encontrada entre todas las ejecuciones. Al igual que ocurría con el modelo de múltiples ejecuciones sin cooperación de las metaheurísticas paralelas basadas en trayectoria, este esquema no cambia el comportamiento del algoritmo, pero reduce de forma importante el tiempo total de cómputo.

- **Población estructurada:** la mayoría de los algoritmos paralelos basados en población encontrados en la literatura utilizan alguna clase de estructuración de los individuos de la población. Este esquema es ampliamente utilizado especialmente en el campo de los algoritmos evolutivos y es el que mejor ilustra esta categorización. Entre los esquemas más populares para estructurar la población encontramos el modelo *distribuido* (o de grano grueso) y el modelo *celular* (o de grano fino) [8].

En el caso de los algoritmos distribuidos [1] (véase el esquema de en la Figura 2.3a), la población se divide entre un conjunto de islas que ejecutan una metaheurística secuencial. Las islas cooperan entre sí mediante el intercambio de información (generalmente soluciones, aunque nada impide intercambiar otro tipo de información). Esta cooperación permite introducir diversidad en las subpoblaciones, evitando caer así en los óptimos locales. El usuario debe determinar una serie de parámetros tales como: la topología, que indica a dónde se envían los individuos de cada isla y de dónde se pueden recibir; el periodo de migración, que es el número de iteraciones entre dos intercambios de información; la tasa de migración, que es el número de individuos emigrados; el criterio de selección de los individuos a migrar y criterio de reemplazo, que indica si se reemplazan algunos individuos de la población actual para introducir a los inmigrantes y determina qué individuos se reemplazarán. Finalmente, se debe decidir si estos intercambios se realizan de forma síncrona o asíncrona.

Por otro lado, las metaheurísticas celulares [40] (véase el esquema en la Figura 2.3b) se basan en el concepto de vecindario (conjunto de individuos que son vecinos de uno dado)[1]. Cada individuo tiene a su alrededor un conjunto de individuos vecinos donde se lleva a cabo la explotación de las soluciones. La exploración y la difusión de las soluciones al resto de la población se produce debido a que los vecindarios están solapados, lo que produce que las buenas soluciones se extiendan lentamente por toda la población.

Aparte de estos modelos básicos, en la literatura también se han propuesto modelos híbridos donde se implementan esquemas de dos niveles. Por ejemplo, una estrategia bastante común en la literatura es aquella donde en el nivel más alto tenemos un esquema de grano grueso, mientras que cada subpoblación se organiza siguiendo un esquema celular.

2.3.3. Plataformas paralelas

La búsqueda de un mayor rendimiento en los modelos algorítmicos ha impulsado la utilización de la computación paralela en muchas áreas. Durante mucho tiempo, el surgimiento de arquitecturas de computación paralela, tal como los ambientes grid [12, 42], cluster de máquinas [7] o los procesadores multicore [57, 67] y manycore [100], proveen nuevas oportunidades para desarrollar técnicas de computación paralelas para mejorar la solución de problemas y disminuir los tiempos de procesamiento requeridos. No obstante, todos los modelos paralelos de metaheurísticas no se adecuan a este tipo de plataformas y explotan toda su capacidad de cómputo.

En particular, las plataformas multiprocesador son herramientas útiles para el desarrollo e implementación de aplicaciones paralelas de metaheurísticas. La disponibilidad de múltiples

[1] No debe confundirse con el concepto de vecindario en el espacio de soluciones que se usa en la búsqueda local

recursos integrados en un único dispositivo permite tratar con problemas de optimización que requieren una solución rápida, o en tiempo real, en una plataforma fácil de programar.

Todos estos sistemas han posibilitado la mejora en la búsqueda de soluciones (casi) óptimas mediante metaheurísticas. Sin embargo, hoy en día, diversos factores han generado que el tamaño de algunos problemas reales sea muy complicado de procesar, ni siquiera las arquitecturas paralelas más usuales como los clusters de máquinas o las arquitecturas many-core tienen la capacidad suficiente para abordar, en un tiempo razonable, estos problemas de optimización que implican tareas tan costosas computacionalmente. En los últimos tiempos, la aparición de una nueva arquitectura denominadas Unidades de Proceso Gráfico [17] han permitido ir solventando inconvenientes de computo para una gran cantidad de datos, ya que son capaces de usar la potencia computacional de una gran cantidad de procesadores concentrados en un componente de pequeño tamaño.

2.4. Conclusiones

En este capítulo se han introducido los conceptos fundamentales sobre metaheurísticas. Posteriormente, se ha mostrado una clasificación general de las metaheurísticas realizando una división en dos clases, atendiendo al número de soluciones tentativas con la que trabajan en cada iteración: metaheurísticas basadas en trayectoria y en población. Tras este repaso se han introducido las metaheurísticas paralelas indicando las distintas formas de paralelismo consideradas en la literatura. Cada uno de estos modelos ha tenido una descripción de sus características propias cuando se trabaja con metaheurísticas. Finalmente, se ha dado explicado las diversas plataformas paralelas que han ido emergiendo con el correr del tiempo y la necesidad de acelerar o mejorar los resultados de las metaheurísticas.

Capítulo 3

Computación en GPU utilizando metaheurísticas

Hoy en día, uno de las mayores exponentes de las arquitecturas paralelas son las Unidades de Proceso Gráfico (GPUs, del inglés, Graphical Processing Units), que han experimentado un gran éxito estos últimos años gracias a su capacidad para procesar datos en paralelo de forma masiva, convirtiéndose actualmente en plataformas con una excelente relación coste/rendimiento para la computación de propósito general. En este capítulo se da una noción inicial de la GPU, su arquitectura y modelo de programación. Posteriormente, presentaremos una revisión de los principales algoritmos metaheurísticos ejecutados sobre GPUs encontrados en la bibliografía asociada a este campo. Pretendemos con ello dar un cauce lógico que plantee la necesidad de resolver problemas complejos usando algoritmos paralelos sobre GPUs. Igualmente, se discuten las principales ventajas de usar estos algoritmos y las falencias que presentan, ayudando a justificar la utilidad de nuestro trabajo en este campo.

3.1. Unidades de Proceso Gráfico

Durante muchos años los fabricantes de *hardware* han basado las mejoras de los computadores en el aumento de los ciclos de reloj del procesador. Este modelo de mejora se estancó a mediados del 2003 debido a los altos consumos de energía que producían las altas frecuencias de reloj. Desde ese momento los fabricantes de microprocesadores optaron por modelos de diseños *multi-core* y *many-core* [100, 113], en el que existen varias unidades de procesos en un mismo chip, de esta forma se aumenta la capacidad de proceso sin aumentar el consumo de energía. En la actualidad las plataformas multicore lideran el mercado de los computadores.

Por otro lado, el gran auge de los videojuegos y el constante crecimiento de este mercado a llevado a que las empresas pueda realizar una evolución en sus tarjetas gráficas de tal manera que dejaron de ser meros renderizadores de *pixeles* y se han convertido en procesadores con una gran capacidad de cómputo. Toda este desarrollo trajo de la mano la mejora en lenguajes de programación para trabajar sobre las mismas tarjetas. La programación ha sido, hasta hace bien poco, una tarea de expertos puesto que sólo se disponía de APIs gráficas, como OpenGL y DirectX. En este sentido, los avances realizados en el modelo de programación y las herramientas de programación de los procesadores gráficos han resultado determinantes para su éxito. Desde hace un tiempo, tanto AMD como NVIDIA han puesto a disposición nuevos modelos de programación a los desarrolladores de algoritmos para propósito general.

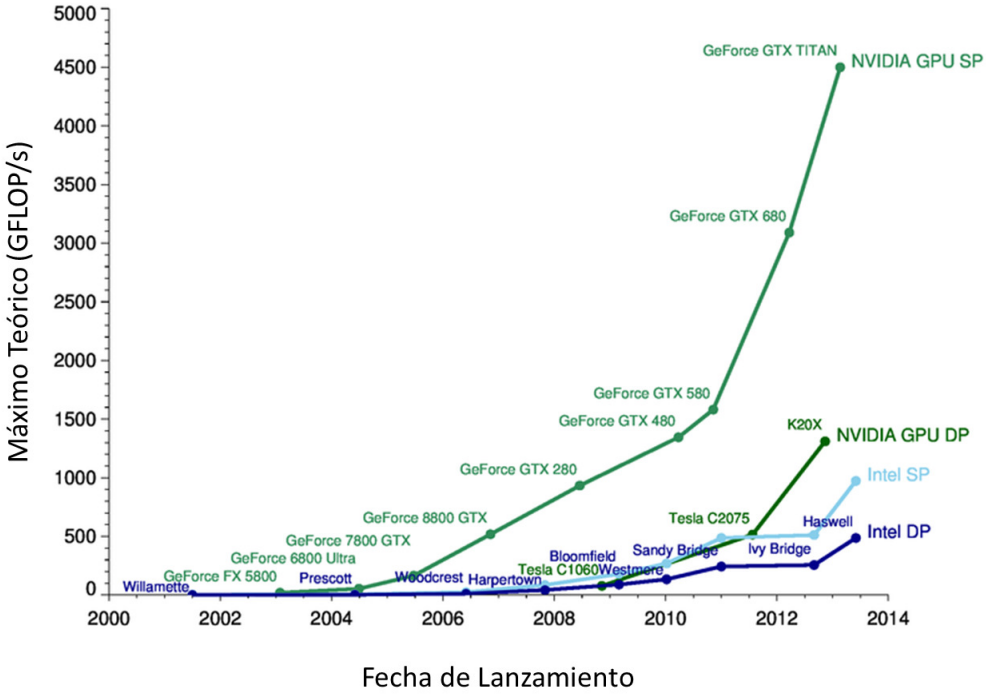


Figura 3.1: Comparativa de rendimiento entre tecnologías CPU y GPU mediante la medición de operaciones de coma flotante por segundo (FLOPS).

Por todos estos motivos las GPUs han evolucionado notablemente en los últimos años, superando en rendimiento a las CPUs (ver Figura 3.1). La figura muestra la evolución a través de los años de diversas arquitecturas (CPU o GPU) y su rendimiento medido en operaciones de coma flotante por segundo (FLOPS, por el inglés floating point operations per second). Se puede observar claramente que la GPU sobrepasa considerablemente a la CPU en cantidad de operaciones por segundo conforme el tiempo avanza.

Las GPUs, en un principio, fueron diseñadas para el procesamiento de imágenes gráficas. En sus orígenes, estaban formadas por un flujo segmentado cuyas etapas realizaban tareas fijas, explotando así el paralelismo a nivel de tareas y también el paralelismo a nivel de datos, ya que en cada etapa se trabajaba sobre varios datos a la vez. El rápido avance, sobre todo en la flexibilidad de programación de los procesadores gráficos, ha permitido utilizarlos para resolver un amplio rango de problemas complejos con altas necesidades computacionales no solo de índole gráfica. Es lo que se conoce como GPGPU (General-Purpose Computing on the GPU) [98].

En términos generales, la CPU utiliza a la GPU como un co-procesador el cual puede aligerar la carga de información mediante el uso de fuerza bruta de procesamiento. De esta manera, la GPU libera de la ejecución de algunas tareas a la CPU y esta puede hacer su trabajo de manera más eficiente. Esta descripción es visible en la Figura 3.2.

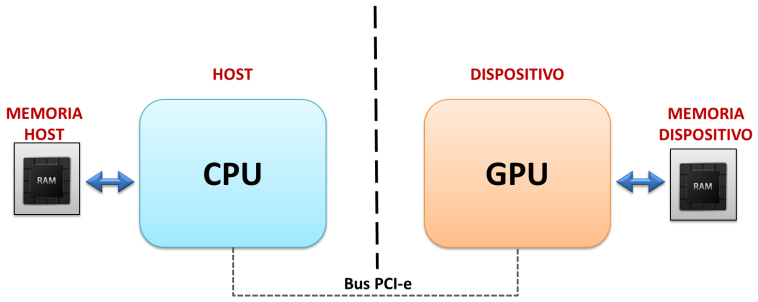


Figura 3.2: Ilustración de la comunicación entre la CPU y la GPU.

3.2. Arquitectura de una GPU

La Corporación NVIDIA presentó a finales del 2006 una nueva línea de *hardware* orientado a la computación de propósito general (modelo G80). Ofrecía un *hardware* de altas prestaciones sin ningún tipo de orientación especifica a aplicaciones gráficas. A pesar de que no es la implementación de NVIDIA más actual, en este apartado nos vamos a centrar en la arquitectura G80, ya que representó un hito tecnológico importante por el hecho de implementar una arquitectura unificada y, tal como vamos a ver en el siguiente apartado, trajo consigo la aparición del modelo de programación CUDA [96]. NVIDIA ha presentado un conjunto de nuevas facilidades a nivel de arquitectura y de modelo de programación para usuarios que no tenían un vasto conocimiento del mundo gráfico y deseaban sacar provecho del enorme poder de procesamiento paralelo inherente en las GPUs.

La Figura 3.3 muestra la arquitectura típica de una GPU actual. Está compuesta por un número escalable de multiprocesadores paralelos (SMs). El número total de SMs (definido por el valor de N) varía desde las arquitecturas más antiguas, hasta las más modernas y de mayor gama. El

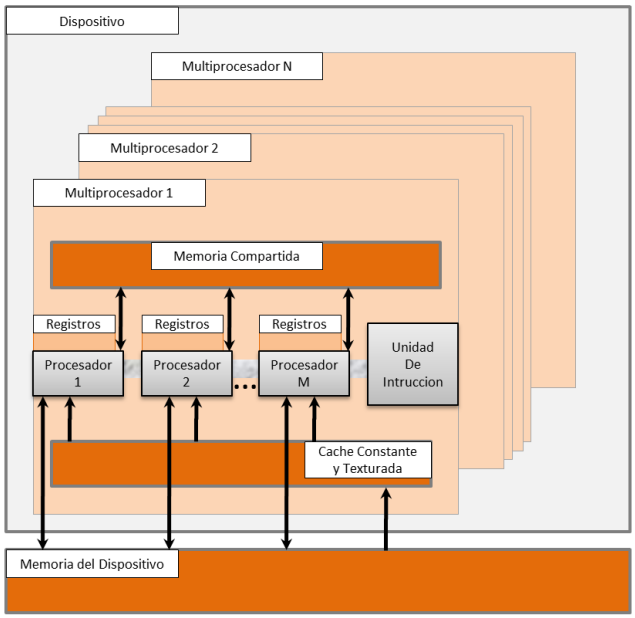


Figura 3.3: Arquitectura de una GPU actual.

diseño interno de cada SM es similar para todas las versiones, cada SM cuenta con un numero igual de procesadores escalares (SPs) (dado por el valor M), lo que hace un total de $M \times N$ núcleos y 2 Unidades Especiales (SFUs), capaces de realizar operaciones en punto flotante. También cuenta con una unidad de multiplicación y suma (MAD) y una unidad adicional de multiplicación (MUL). Los M procesadores de un multiprocesador comparten la unidad de instrucción y búsqueda distribuyendo y ejecutando la misma para todos los procesadores. Cada SM cuenta con un diverso grupo de memorias que son usadas de acuerdo al contexto de trabajo y al tamaño de información a ser asignada. Todos estos componentes son descritos en la siguiente sección.

3.3. Modelo general de trabajo en CUDA

Las GPUs trabajan de acuerdo al paradigma SIMD (del inglés *Single Instruction-Multiple Data*). La clase SIMD engloba a los procesadores vectoriales. Todos los elementos de procesamiento reciben, de una unidad de control, la orden de ejecutar la misma porción de código sobre diferentes conjuntos de datos procedentes de distintos flujos de datos.

Para el caso de NVIDIA, el modelo de programación de CUDA [96] supone que las unidades de procesamiento básica, los *hilos* CUDA, se ejecutan en el dispositivo GPU que actúa como co-procesador a la CPU (el *host* o anfitrión) donde esta funcionando el programa principal. Algunas de sus secciones de código se ejecutan en la CPU (código secuencial), mientras que otras secciones se ejecutan en la GPU como se puede apreciar en la Figura 3.4.

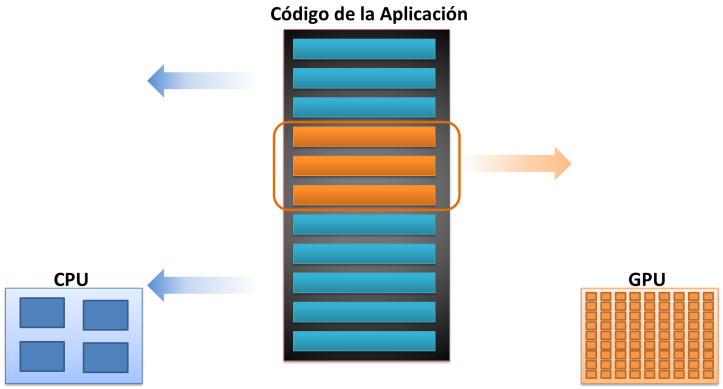


Figura 3.4: Distribución del código entre la CPU (anfitrión) y la GPU (dispositivo).

La programación en paralelo sobre GPUs permite implementar programas en lenguaje C con extensiones predefinidas CUDA que permiten definir cierto tipo de funciones. Éstas se llaman *kernels*, y son invocadas desde el *host*. El *kernel* es la porción de código que ejecutan los hilos durante la fase paralela.

Código 3.1: Invocación de un *kernel* en un programa C.

```
// Definicion del Kernel
__global__ void SumaVec(float* A, float* B, float* C) {
   int i = threadIdx.x;
   C[i] = A[i] + B[i];
 }

 void main(void) {
 ...//Codigo Secuencial
 // Invocacion del Kernel con N hilos
 SumaVec <<< 1, N >>> (A, B, C);
 ...//Codigo Secuencial
 }
```

El Código 3.1 muestra la invocación del *kernel* llamado *SumaVec* dentro de un código secuencial y arriba el cuerpo del mismo *kernel*, el cual será ejecutado en la GPU. Los valores entre <<< ... >>> se conocen como la configuración del *kernel*, y definen la distribución de hilos según la estructura de trabajo de CUDA. Cada invocación de un *kernel* genera un nuevo contexto con una estructura definida y su distribución de hilos correspondientes.

Las diferencias principales entre el *host* y un dispositivo GPU residen en el trabajo de los hilos y la memoria [96]

- *Hilos*: Los sistemas *host* pueden soportar un número limitado de hilos concurrentes. Los hilos de la CPU son generalmente entidades pesadas, el sistema operativo tiene que estar arrancando y desactivando hilos para proveer una ejecución multi-hilo, es muy costoso pasar de un estado a otro. Por otro lado los hilos de la GPU son muy ligeros, en un sistema típico miles de hilos son encolados a razón de 32 hilos por *warp*. El *warp* es la agrupación de hilos que es enviado a cada procesador para ser ejecutado. Si la GPU debe esperar a un *warp*, simplemente empieza a ejecutar otro, como los registros se asignan a hilos activos, ningún intercambio de registros y de estado ocurre entre hilos de la GPU, estando los recursos disponibles hasta que se complete la ejecución de los hilos.
- *RAM*: Ambos (host y GPU) tienen su propia RAM (Figura 3.2. En el *host* generalmente la RAM es igualmente accesible para todo el código (con limitaciones establecidas por el sistema operativo). En el dispositivo la RAM se divide virtual y físicamente en diferentes tipos (ver Tabla 3.1), cada una de ellas para satisfacer diferentes necesidades.

Existe una jerarquía perfectamente definida sobre los hilos desplegados con CUDA. Normalmente, el conjunto de todos los hilos que se generan cuando se invoca un *kernel* se denominan *grids* (en torno a miles o incluso millones de hilos). Los hilos de un *grid* están organizados en una jerarquía de dos niveles, tal como se puede ver en la Figura 3.5.

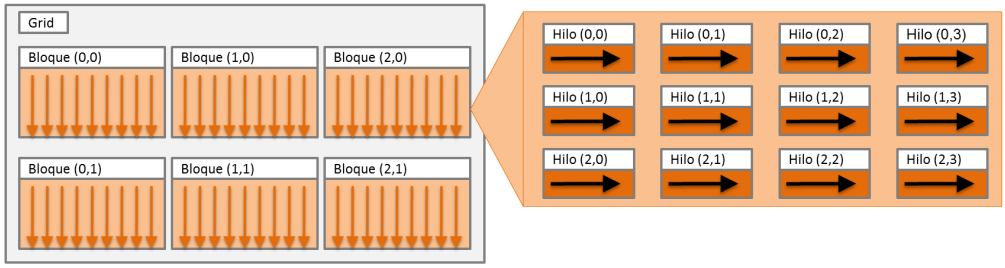


Figura 3.5: Distribución de los hilos a través de la jerarquía grid, bloque, hilo.

El nivel superior de un grid consiste en uno o más bloques de hilos. Todos los bloques de un grid tienen el mismo número de hilos y deben estar organizados del mismo modo. En la Figura 3.5, el grid está compuesto por seis bloques y está organizado en una matriz de 2×3 bloques. Los bloques de un grid se pueden organizar en una, dos o tres dimensiones con coordenadas únicas en el espacio que se encuentren. Los hilos del mismo bloque pueden cooperar entre sí, compartiendo datos y sincronizando sus ejecuciones. Sin embargo, hilos de distintos bloques no pueden cooperar entre sí. Cada hilo queda perfectamente identificado por un *ID* identificador del bloque y un *ID* para el propio hilo dentro del bloque los cuales suelen usarse como índices para definir que porciones de los datos procesa cada hilo.

La ejecución de los hilos en la GPU no se lleva a cabo de forma independiente. En primer lugar, cada SM recibe un bloque de hilos para ser ejecutados. El multiprocesador divide los bloques de hilos en warps. A la hora de lanzar una nueva instrucción, la unidad de planificación, selecciona un *warp* disponible y lanza la misma instrucción para todos los hilos de ese *warp*.

Las instrucciones de salto suponen un problema ya que hilos de un mismo *warp* pueden tomar caminos de ejecución distintos. En este caso, la ejecución se serializa, ejecutando primero los hilos de un camino y después los hilos del otro. Asimismo, existen instrucciones para sincronizar todos los hilos de un mismo bloque, haciendo que *warps* enteros detengan su ejecución hasta que todos los hilos del bloque alcancen el mismo punto de ejecución.

En la Figura 3.3 se puede observar los diferentes modelos de memoria que ofrece y brinda CUDA al programador en términos de asignación, transferencia y utilización de espacio. Cada hilo tiene acceso a distintos tipos de memoria. En primer lugar una memoria local accesible solo por el propio hilo, esta memoria es parte de la memoria principal de la GPU. De la misma forma, todos los hilos de un mismo bloque comparten una región denominada memoria compartida, la cual sirve para almacenar información accesible desde cualquier hilo del bloque. La memoria compartida tiene el mismo tiempo de vida que el bloque de hilos, es decir, cuando se destruye la estructura de bloques, la información desaparece. Por último, todos los hilos tienen acceso a la misma memoria global (memoria del dispositivo). En ella se encontramos espacios reservados para memorias de tipo constante y para almacenar texturas gráficas. La Tabla 3.1 presenta un resumen de cada memoria y los diferentes accesos de acuerdo al tipo de alcance de los hilos.

Tabla 3.1: Jerarquía de memorias dentro de CUDA.

Tipo de Memoria	**Ubicación**	**Cacheada**	**Acceso**	**Alcance**
Registro	En el Chip	No	Lectura/Escritura	Hilo Único
Local	Fuera del Chip	No	Lectura/Escritura	Hilo Único
Compartida	En el Chip	No	Lectura/Escritura	Hilos de un Bloque
Global	En el Chip	No	Lectura/Escritura	Todos
Constante	En el Chip	Si	Lectura	Todos
Texturada	En el Chip	Si	Lectura	Todos

Las nuevas arquitecturas de NVIDIA llamadas Fermi [94] y luego Kepler [95], están implementadas con más 3 millones de transistores, con un promedio de 512 núcleos CUDA o más, organizados en muchos SM de 32 núcleos cada uno. Fermi y Kepler proporcionan la capacidad de un superordenador por tan sólo una décima parte del coste y una vigésima parte del consumo de energía de los tradicionales servidores basados en CPUs. Para una mayor revisión de este tipo de arquitecturas se recomienda mirar la literatura [17, 107]

3.4. Cómputo evolutivo sobre GPU

El uso de GPUs para cómputo evolutivo ha empezado a gozar de cierta popularidad debido a que las metaheurísticas son altamente paralelizables en modelos SIMD. Sin embargo, la optimización en GPU no es una metodología trivial de trabajo, es un proceso que requiere esfuerzo, capacidad de diseño y de implementación. Son muchas las características de estas arquitecturas a tener en cuenta a la hora de modelar nuevos desafíos. El mayor desafío que podemos encontrar es la eficiente distribución de tareas de procesamiento de datos a través de las diferentes tipos de memoria, la correcta utilización de la organización estilo top-down de los hilos y la transferencia de información entre cada una de las memorias de la GPU, y entre el dispositivo GPU y el *host*. A continuación, se revisan aquellos trabajos previos enfocados a metaheurísticas sobre GPUs. Se presenta un análisis de las características de cada modelo implementado, las estrategias utilizadas para abordar el diseño e implementación sobre las GPUs y los resultados obtenidos sobre los problemas evaluados.

3.4.1. Estado del arte de metaheurísticas sobre GPU

En esta sección presentamos una revisión de las principales metaheurísticas y nuevas aproximaciones implementadas sobre plataformas GPU encontradas en la bibliografía. El objetivo de este apartado es dar una dirección organizada que ejemplifique la necesidad de utilizar algoritmos conocidos por su eficiencia para resolver problemas complejos del tipo NP-duro.

Se explican los principales trabajos existentes en los que se han aplicado diferentes modelos paralelos siguiendo la metodología Maestro-Esclavo, Grano Grueso y Grano Fino. Para terminar esta sección, se brinda un resumen de algunos dominios en los cuales estas técnicas han posibilitado una resolución numérica de manera eficaz y eficiente, lo cual permitirá justificar la utilidad de nuestro trabajo en el campo de optimización.

3.4.1.1. Aproximación a nivel de solución

Las aproximaciones paralelas pueden trabajar con una única solución, sean independientes o cooperativas entre sí. Para estos casos ha habido varios progresos en este contexto debido a la simplicidad de algunos operadores y a la posibilidad de implementar múltiples aproximaciones paralelas. La Tabla 3.2 reporta los trabajos más importantes en GPU a nivel de solución de forma ordenados en el tiempo. Los trabajos son presentados teniendo en cuenta la clasificación propuesta en la Sección 2.3. Se brinda una breve descripción de las características básicas de cada implementación con lo cual es posible analizar la heterogeneidad existente y las tendencias actuales.

Tabla 3.2: Resumen ordenado por año de algunas implementaciones sobre GPU.

Referencia	Año	Algoritmo	Modelo Tarjeta	Ganancia de Tiempo de Tiempo
[33]	2010	DE	NVIDIA GTX 285	36×
[143]	2011	DE	NVIDIA GTX 280	321×
[73]	2011	DE	Tesla C2050	4×
[75]	2011	DE	Tesla C2050	2× a 216×
[60]	2011	DE	GeForce GTS 250 G92	6× a 160×
[110]	2012	SA	NVIDIA GTX 580	160× a 420×
[11]	2012	SA	GeForce G 103M	1.1×
[133]	2013	DE	NVIDIA GTX 285	75×
[41]	2013	SA	NVIDIA GTX 470	76× a 269×

La Evolución Diferencial (DE, por sus siglas en inglés) es adecuada para la ejecución paralela en la GPU debido a la simplicidad de sus operaciones, la amplia adopción del algoritmo, y verdadera codificación de soluciones candidatas. En una DE, cada solución candidata se representa por un vector de números reales (parámetros) y la población nueva que se crea puede ser vista como una matriz de valor real. Además, tanto el operador de mutación y el operador de cruce pueden ser implementados fácilmente como operaciones vectoriales sencillos. Las primeras implementaciones de un DE en las GPU fueron introducidos en 2010 por [33, 143]. El trabajo presentado en [33] logra una aceleración entre 19× a 34× en el tiempo de ejecución en comparación con una implementación basada en la CPU en un conjunto de funciones de evaluación comparativa. Zhu [143] desarrolla un DE utilizando un patrón de evolución para problemas de optimización con restricciones y como parte de una simulación de Monte Carlo. En ambos casos, el rendimiento de los algoritmos fue probado sobre un dominio de problemas continuos y con una ganancia de tiempo entre 4× a 408×. Estos estudios son ampliados de forma más detallada en [73].

En [75] se presentó una aplicación de grano fino para DE. La misma está dirigida a aumentar la utilización y ocupación de las GPUs usando todos los recursos disponibles para cada hilo. Para este trabajo se puede observar el uso de herramientas de simulación propias de nVidia, tales como el generador de números aleatorios sobre GPU llamado cuRAND [93]. Estas implementaciones [73, 75] fueron aplicadas sobre problemas e optimización combinatorios. Los autores han informado de un aumento de velocidad entre 1.4× a 9.7× para el problema de ordenación lineal y una aceleración de 2.2× a 12.5× para el problema de programación de tareas.

En 2013, los autores de [133] implementaron un algoritmo DE paralelo con parámetros auto-adaptables teniendo en cuenta los principios del aprendizaje general basados en oposición para resolver problemas de optimización de gran dimensión. Los resultados demostraron un aumento de velocidad desde 1.29× hasta 7.84× para una población con 128 soluciones procesadas por

128 hilos paralelos. Con el tamaño de la población de 4096 y 4096 hilos paralelos, la aceleración reportada esta entre 12.23× y 75.03×.

Otro algoritmo ampliamente estudiado ha sido el Recocido Simulado (SA, por sus siglas en inglés). En [60] se ha estudiado la implementación de un SA para resolver la planificación en circuitos integrados VLSI (del inglés Very Large Scale Integration) y se analizaron varias propiedades que afectan al rendimiento del método en la GPU como el uso de memoria local, compartida y global y por otro lado la texturada y constante. Asimismo, ellos han trabajado sobre la descomposición de todo el problema entre la CPU y la GPU para reducir las transferencias de memoria entre el *host* y la GPU. Los experimentos con dos plataformas de hardware mostraron que la aceleración de la versión en GPU rondaba entre 6× a 160× dependiendo de las propiedades y parámetros analizados. En [110] se publico otro SA paralelo el cual es utilizado para simulaciones numéricas. La aplicación obtuvo una aceleración entre 160× a 420×.

Una breve comparación del desempeño de SA entre las CPUs y GPUs ha sido publicada en [11]. El estudio presenta una implementación más en GPU, no obstante, no ha sido lo suficientemente amplio para incluir todas las implementaciones de tipo actualizadas en GPU e incluso el problema de prueba no fue bien definido. Finalmente, un estudio bien detallado de la ejecución de un SA paralelo esta explicado en [41]. Los autores han diseñado primeramente un SA para un solo hilo y posteriormente la versión en GPU. Luego, desarrollaron una version más eficiente síncrona del algoritmo capaz de ir intercambiando soluciones en cada cambio de nivel de temperatura. La ganancia de tiempo para ambas aproximaciones de GPU es de 76× y 269×, respectivamente. Las pruebas fueron realizadas evaluando la variación del número de variables del problema (8 a 512) relacionado con el número de hilos ejecutados por bloque y el número de bloques. La comparación indica que las versiones de GPU del SA puede ser muy eficiente para ambas aproximaciones.

3.4.1.2. Aproximación a nivel de población

Con respecto al trabajo de metaheurísticas basadas en poblaciones existen muchas aproximaciones las cuales han tratado de paralelizar la mayoría de algoritmos existentes en la actualidad. Primeramente, la evaluación de soluciones candidatas es en muchos casos la operación que consume el mayor tiempo de toda la ejecución de algoritmos de este tipo. La forma de aliviar la ejecución es mediante la paralelización o distribución de este cómputo. Trabajos previos se han enfocado sobre dos tipos de evaluaciones: paralelizar la evaluación de la función de fitness o trabajar sobre cada una de las soluciones de la población. En la Tabla 3.3 se muestra un resumen de estas implementaciones. La misma agrupa aquellas importantes aproximaciones relacionadas con los modelos de Maestro-Esclavo, Grano Grueso y Grano Fino.

Una de las metaheurísticas más analizadas dentro del contexto de optimización han sido los

Algoritmos Genéticos (GAs, por sus siglas en inglés). En [43] se introduce un método paralelo para el cálculo de fitness en un GA. Su propuesta alcanza ganancia de tiempo de hasta 52×. En general, los primeros trabajos [43, 101, 102, 132] se han centrado en la paralelización de la evaluación de múltiples individuos o casos de entrenamiento y muchas de estas propuestas se limitaron a pequeños conjuntos de datos debido a las limitaciones de memoria impuestas por la arquitectura de la GPU en los comienzos de las investigaciones. Una comparación entre modernas CPUs y GPUs es presentado en [64]. El objetivo del trabajo ha logrado mostrar la relación de eficiencia entre las dos arquitecturas buscando erradicar algunos mitos que rondan a la ejecución de algoritmos GAs sobre GPU. El trabajo demuestra que las implementaciones en CPU bien realizadas pueden reducir en un 50 % la diferencia de ganancia de tiempo entre un algoritmo CPU y el mismo en GPU. En 2013, los autores de [19] presentan un GA orientado a trabajar en problemas de minería de datos obteniendo una ganancia de tiempo de 6× a 200×. Estas implementaciones de GA demuestra el inherente paralelismo que presenta esta técnica.

Tabla 3.3: Resumen ordenado por año de algunas implementaciones sobre GPU.

Referencia	Año	Algoritmo	Modelo de Tarjeta	Ganancia de Tiempo
[140]	2005	PSO, GA, DE	GeForce 670x	10×
[101]	2009	GA	GeForce GTX 8800 / GTX 285	0× a 8000×
[122]	2009	GA	NVIDIA GTX 285	3× a 12×
[90]	2009	GA + LS	Tesla C1060	25×
[138]	2009	GA	GeForce FX 6800	1.17× a 5.30×
[43]	2010	GA	Tesla C1060	765×
[37]	2010	EA	-	76×
[37]	2010	EA	NVIDIA GTX 280	8.5× a 14.6×
[34]	2010	Sistema de Hormigas Max-Min	Tesla Serie-10	0.42× a 5.52×
[124]	2010	ACO + TS	NVIDIA GTX 480	1.2× a 4.9×
[132]	2011	GA	Tesla C1060	3.7× a 7.8×
[64]	2012	GA	NVIDIA GTX 580	1× a 375×
[22]	2012	GP	GeForce 460	5× a 395×
[18]	2012	GP	GeForce GTX 285 / GeForce 480 GTX (2)	1× a 820×
[91]	2012	PSO, SS, DE	GeForce GTS 450	-
[126]	2013	PSO + DE	GeForce GTS 450	1.3×
[19]	2013	GA	GeForce GT540M	6× a 200×

Una variante de un GA, la Programación Genética (ingles, Genetic Programming (GP)) ha sido estudiado. Chitty [22] ha realizado un estudio comparativo de una implementación CPU multicore frente a una versión en GPU. Asimismo, en [18] los autores han estudiado diferentes aproximaciones de paralelización, tres variantes de GP usando una aproximación CPU multi-core y dos GPU.

En el 2010, una comparación entre varios EAs fueron implementados por [37]. Se probaron tres algoritmos (GA, DE y Optimización mediante Enjambre de Partículas (PSO, por sus siglas en inglés) utilizando el entorno de computo distribuido Milkyway@home con diversas funciones de

prueba. Los experimentos revelaron que todos los algoritmos investigados fueron adecuados para tal arquitectura y el aumento de velocidad logrado prometían ganancias de tiempo. Asimismo, un algoritmo de cooperación basado en el modelo de PSO y DE para la detección de objetos fue propuesto en [126]. El algoritmo utiliza dos métodos de optimización para la localización de objetos en imágenes 2D con éxito acelerando ambos algoritmos con la GPU. Finalmente, una aproximación sin mucho detalle indica la implementación de tres algoritmos: PSO, Búsqueda Dispersa (del inglés Scatter Search (SS)), y un DE). Los detalles de implementación son escasos, sobre todo para el caso del SS.

Aproximación maestro-esclavo

En [142] los autores plantean la adaptación de diferentes EAs a una GPU usando CUDA. Los autores aplican un algoritmo GA a través de un esquema maestro-esclavo. En esta aplicación, la CPU inicializa las poblaciones y las distribuye en las memorias compartidas de cada bloque de hilos. Posteriormente, dentro de cada bloque de la GPU se ejecuta de forma independiente un GA, y seguidamente se migran los individuos a otros bloques del vecindario. En este caso, no se reportaron resultados de ganancia de tiempo.

El trabajo presentado en [132] implementa una aproximación GA maestro-esclavo y realiza una comparación entre diferentes arquitecturas (multi-core, cluster de máquinas y una GPU). Los resultados muestran una ganancia de tiempo hasta 7.8× con respecto al procesador multi-core y de 5× con respecto al cluster de máquinas. Es una de las más atractivas comparaciones existentes dado que implica una comparación entre diversos modelos de computación paralela.

Más adelante otros trabajos relacionados con este enfoque han aparecido. En particular, el Algoritmo de Optimización de Colonia de Hormigas (ACO, por sus siglas en inglés) ha demostrado dos interesantes aproximaciones. Uno de ellos se puede apreciar en [34] donde se utiliza un Sistema de Hormigas Max-Min, variación del ACO. Esta aproximación trabaja sobre el problema TSP, donde el maestro (la CPU) se encarga de casi todo el proceso salvo de la fase de construcción de los caminos de TSP para cada solución (ejecutado sobre la GPU). Alcanza ganancias de tiempo entre 0.52× y 5.52×. En [124] se utiliza un enfoque de maestro-esclavo con un algoritmo ACO y una Búsqueda Tabú [49]. Para este trabajo se utiliza una sola GPU NVIDIA GeForce GTX480. Se compara entre las implementaciones de CPU y GPU obteniendo resultados que mostraron que la computación de la GPU con MATA (del inglés Move-Cost Adjusted Thread Assignment), un método eficiente para la asignación de costos de uso de hilos mostrando una aceleración de hasta 5× en comparación con el cálculo de la CPU.

Aproximación de grano grueso

Con el fin de ejecutar una metaheurística de grano grueso utilizando una GPU. La principal limitación esta en controlar cada sub-población y la migración de información hacia otras sub-poblaciones. Esta limitación significaría que las mecánicas convencionales de EAs de grano grueso debería verse afectada al utilizar una o varias GPUs.

Uno de los primeros modelos de la isla sobre GPU se publicó en la competencia de optimización en GPU de GECCO de 2009 [101]. La aproximación hace foco en el trabajo sobre una topología de anillo de sub-poblaciones, las cuales son controladas por cada procesador de la GPU. No obstante, los operadores evolutivos implementados en GPU son sólo específicos para la competencia GECCO, y la validez de los experimentos sólo funciona en un pequeño número de problemas.

En Tsutsui et al. [122] proponen ejecutar un GA de grano grueso en la GPU para resolver el problema de asignación cuadrática (QAP, del inglés Quadratic Assignment Problem) usando CUDA. Este es uno de los problemas más difíciles de optimización en dominios de permutación. El modelo genera la población inicial en la CPU y se copia a la memoria global de la GPU; a continuación, cada sub-población es controlada y evolucionada usando un multiprocesador de la GPU (NVIDIA GeForce GTX285). En algunas generaciones, las soluciones de las sub-poblaciones se intercambian a través de la memoria global de la GPU. Los resultados mostraron una ganancia de tiempo entre 3× a 12× (probado sobre instancias de QAP), en comparación de la versión de CPU con un procesador Intel i7 965.

El modelo propuesto por Luong et al. [83] se basa en un re-diseño del modelo de isla. Se proponen tres esquemas diferentes: El primero de ellos implementa un EA de grano grueso con un modelo de maestro-esclavo para ejecutar la etapa de evaluación en la GPU. El segundo distribuye la población de EA en las múltiples GPUs, mientras que la tercera propuesta se extiende a la segunda versión utilizando la memoria compartida. El segundo y tercer modelo logran reducir los tiempos de latencia de memoria CPU/GPU, aun cuando varios de los parámetros tales como números de islas, frecuencia de migración, entre otros tuvieron que ser adaptados a las características de la arquitectura GPU. La comparación entre las versiones secuenciales y paralelas fueron comparada obteniendo una aceleración máxima de 1757× usando el tercer enfoque.

Aproximación de grano fino

En el caso de aproximaciones con grano fino, Munawar ofrece en 2009 un GA utilizando una malla toroidal 2D donde se localizan las soluciones [90]. El algoritmo está orientado a resolver el problema MAX-SAT hibridizado con una LS obteniendo una ganancia de 25× con respecto a la versión secuencial. Wong et al. [138, 139] propuso un GA híbrido paralelo (HGA), donde todo el proceso evolutivo se ejecuta en la GPU, y sólo la generación de números aleatorios se realiza en

la CPU. Utilizando un nuevo método de selección pseudo-determinista, la cantidad de números aleatorios es transferida de la CPU a la GPU de una forma bastante reducida. Para trabajar en la GPU utiliza una malla de dos dimensiones toroidal para trabajar (al estilo algoritmo Genético Celular). Los autores comparan HGA con un GA secuencial en una CPU y el realizado en [140]. El HGA alcanza aceleración de 5.30 cuando se compara con la versión secuencial.

3.4.1.3. Aproximaciones paralelas híbridas

El término híbrido se refiere a diseñar algoritmos que puedan explotar y combinar las ventajas de las estrategias individuales puras. Estas metaheurísticas híbridas han demostrado ser potentes algoritmos de búsqueda [115]. A veces, no solo se puede referir a combinar diferentes componentes algorítmicos, sino también a la combinación de plataformas de computo heterogéneas involucradas en un determinado cálculo. Para nuestro caso, la GPU puede manejar fácilmente los datos que implican un tareas masivamente paralelas al beneficiarse de su extenso número de núcleos e hilos ligeros, mientras que la CPU ha sido especialmente diseñada para las tareas que necesitan de un trabajo con varias operaciones implicadas no necesariamente paralelas, tales como las inicializaciones, rutinas de arranque, las transferencias de datos y la gestión de resultado final. Siguiendo la secuencia anterior, lo natural sería una combinación de ambos contextos (hardware y software). Esto es complicado debido a las diferentes características de cada uno y, sobre todo, porque la comunicación entre la CPU y la GPU consume un tiempo excesivo en comparación con la velocidad de procesamiento real. Todo esto puede conducir a una mala utilización de la propia CPU, restringir la creatividad del diseñador cuando se desear crear nuevos algoritmos, y se podría perder una gran cantidad de tiempo en la transferencia de datos. Existen muy pocas aproximaciones relacionadas con este tópico las cuales merecen un análisis y un planteamiento con respecto a lo expresado .

En [90], los autores introdujeron una contribución que incorpora una LS en un GA, usando la GPU como procesador principal. Por lo tanto, se lleva a cabo la comunicación entre ellos dentro de la GPU. Este enfoque obtuvo un aumento de velocidad máximo de 25× para las instancias seleccionadas del problema MAXSAT. Una investigación en esta misma línea es presentado por [24], donde introdujeron una distribución de tareas de una LS y un EA entre CPU y GPU. El aumento de velocidad oscila entre 10× a 27 ×.

En una línea similar, un nuevo enfoque fue presentado por Vang Loung [127] presenta pautas interesantes relativas a la distribución del proceso de búsqueda implementado entre la CPU y la GPU, minimizando la transferencia de datos entre las mismas. La propuesta algorítmica indica que la CPU administra todo el proceso evolutivo híbrido y permite que la GPU sea utilizada como un co-procesador dedicado a cálculos intensivos. El modelo de hibridación utilizado en su

trabajo incluye un Algoritmo Evolutivo en la CPU y una LS en la GPU. Este enfoque obtiene una aceleración máxima de 4× para el problema de permutación. La propuesta demuestra claramente que la hibridación es una poderosa manera de lograr una alta eficiencia computacional mediante ambas arquitecturas. En [84] los autores utilizan la GPU para resolver una variedad de problemas usando la GPU para evaluar un LS por GPU, mientras que el CPU estaba a cargo del control del algoritmo (un algoritmo híbrido con múltiples GPUs). Mientras la GPU ejecuta un LS, la CPU se encarga de manejar los hilos y generar nuevos procesamientos de datos. Los algoritmos informan de un aumento de velocidad de hasta 50 × tiempos para las instancias utilizadas con respecto a la versión secuencial.

Después de analizar todos estos trabajos, parece claro que muy pocas aproximaciones utilizan un planteamiento de optimización sobre un modelo de cooperación CPU-GPU para buscar numéricamente a lo largo del espacio del problema. Las pocas aproximaciones relacionadas con el uso de ambos dispositivos de manera eficiente en optimización han reportado muy buenos resultados. La metaheurísticas híbridas en sistemas paralelos cooperativos no ha sido explotadas, quizás por la complejidad de la implementación de las mismas o porque simplemente no es necesario para algunos casos. Lo cierto es que es posible trabajar con hibridación usando la cooperación entre la CPU y la GPU como una plataforma para la implementación de una variedad de técnicas que hagan un uso intensivo de cada arquitectura. Esto pone de manifiesto que una combinación CPU-GPU merece una investigación más exhaustiva.

3.4.1.4. Dominios de aplicación

Este apartado se centra en examinar y describir aquellos desarrollos algorítmicos que trabajan sobre problemas relevantes de la industria o que estén estrechamente relacionados entre el ámbito académico y la vida real. Existen numerosos dominios entre los que destacamos los contenidos en la Tabla 3.4. En esta tabla puede observarse una multiplicidad de trabajos en dominios de optimización funcional, optimización combinatoria, inteligencia artificial, bioinformatica e ingeniería, entre otros. Existen una gran variedad de interesantes aplicaciones en una multitud de dominios que justifican el valor del trabajo de modelar e implementar algoritmos sobre GPU.

En la Tabla 3.4 podemos advertir la variedad de modelos algorítmicos que pueden ejecutarse perfectamente desde entornos caseros (modelo GT, GeForce o Quadro) hasta aquellos altamente especializados para computo científico (modelo Tesla). De acuerdo a la complejidad del problema o del tipo de metaheurística es posible obtener ganancias relativas de tiempo.

Estos trabajos demuestran la flexibilidad de las metaheurísticas, debido a su capacidad para atacar muchas aplicaciones diferentes, sobre una arquitectura paralela tal como la GPU. Esto se logra mediante el uso de una o la combinación de varios métodos para explorar el espacio de

Tabla 3.4: Resumen ordenado por año de algunas implementaciones sobre GPU.

Referencia	Dominio de Aplicación	Algoritmos	Modelo de Tarjeta Gráfica	Ganancia de Tiempo (Speedup)
[118]	Sistemas algebraicos	microGA	Tesla C2050	20× a 42×
[79]	BioInformática	GP	GeForce 8800 GTX	7× a 12.6×
[134]	Minería de Datos	ACO	Tesla C1060	7× a 12.6×
[87]	Permutaciones	LS	GeForce 8800 / GTX 280	4.2× a 573×
[102]	Problema de Mochila	GA	GeForce GTX 260	0.18× a 1339.13×
[38]	Ruteo de Vehículos	ACO	GeForce GTX 460	0.25× a 12.33×
[25]	Ruteo de Vehículos	VNS	GeForce GTX 560 Ti	2.63× a 16.23×
[35]	Viajante de Comercio (TSP)	LS	Tesla C2050	0.42× a 6.02×
[105]	Viajante de Comercio (TSP)	LS	GeForce GTX 680x	0.6× a 59×
[41]	Programación de trabajos	GA / TS / SA	GeForce 670x	10×
[117]	Optimización Funcional	PSO	GeForce GT 330M	2.26× a 5.02×
[80]	Predicción Estructuras Proteínas	SA	-	3×
[74]	Clustering	GA	Tesla C2050	2.64× a 15.20×
[123]	Problemas Combinatoriales	GA	GeForce GTX 285	3× a 12×
[125]	Análisis de Señales de Trafico	PSO, DE	Quadro FX 5800	0.4× a 3.12×
[16]	Problema de Asignación	GA	GeForce 670x	10×

búsqueda, escapar de óptimos locales, y determinar cuándo se han encontrado buenas soluciones para el problema en cuestión. Asimismo, el modelado de metaheurísticas sobre GPU proporciona los medios para gestionar el equilibrio entre el rendimiento y la calidad de las soluciones.

3.5. Conclusiones

Hemos explorado en este capítulo la mayoría de los trabajos existentes en el campo de los algoritmos metaheurísticos sobre GPUscelulares. Los trabajos analizados comprenden tanto las principales publicaciones del campo como las tendencias más recientes que están apareciendo en los últimos años.

La elección de una metaheurística no es trivial en problemas complejos y es necesario un análisis integral teniendo en cuenta cuestiones tales como: (1) para un problema dado la selección de un modelo algorítmico a implementar, evaluando si la búsqueda que puede realizar el mismo es eficaz y eficiente. Finalmente, pero no menos importante el análisis de las características del entorno de ejecución y el emparejamiento de ciertas características entre la arquitectura GPU y la técnica con la cual se va a trabajar.

Este estudio nos permite adquirir un conocimiento del dominio tanto a nivel de implementación como a nivel de complejidad abordada que será necesario para proponer y justificar las investigaciones desarrolladas en este trabajo de tesis.

Capítulo 4

Problemas abordados y diseño experimental

En este capítulo se introducirán las características de los problemas evaluados durante todo el trabajo de tesis. A continuación, se presenta el plan experimental y el procedimiento estadístico para evaluar las técnicas implementadas, donde se muestran los principales indicadores de calidad utilizados. Finalmente, se detalla cada entorno de ejecución utilizado describiendo el hardware y software con el cual se ha trabajado.

4.1. Problemas analizados

El objeto de esta sección no es únicamente dar una colección de ejemplos reales o de problemas difíciles de resolver sino, el de establecer modelos que han sido muy estudiados como base para la evaluación de nuestras propuestas algorítmicas. Así, se tratara de reconocer las estructuras especiales que aparecen en estos modelos y de esta forma se podrá aprovechar la extensa literatura y experiencia computacional al respecto. Además, no debemos olvidar la limitada, pero significativa, importancia práctica de alguno de estos modelos de problemas.

Problema Masivamente Multimodal

El Problema Deceptivo Masivamente Multimodal [54] (MMDP, por sus siglas en inglés) está compuesto de k subproblemas, cada uno compuesto por 6 bits que pueden ser 0 o 1. La suma de estos 6 bits puede tomar diferentes valores teniendo en cuenta la cantidad de unos (desde 0 a 6) y son apreciables en la Tabla 4.1)). Esto se conoce como una función de unitación, donde el valor

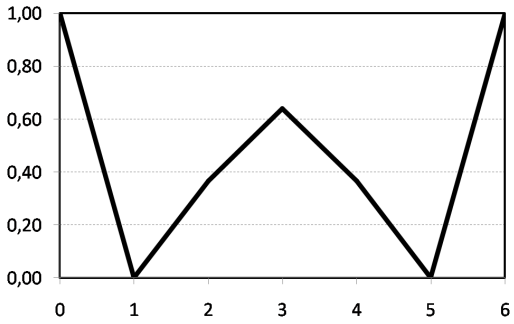


Figura 4.1: Representación gráfica de los valores de unitación con respecto a los valores posibles de un sub-problema.

del problema está dado por las variables componentes con valor uno, independientemente de la posición de cada una de ellas. Esta función de unitación es importante ya que permite el estudio del comportamiento de los algoritmos de optimización ante la presencia de múltiples óptimos locales y globales. Los resultados de cada subproblema pueden tener dos máximos globales y un punto de atracción medio con un valor resultado intermedio para un subproblema. La función de *fitness* a ser maximizada se muestra en la Ecuación 4.1.

Tabla 4.1: Valor resultante de cada subproblema y su valor de unitación correspondiente

Valor resultante de cada subproblema	Resultado de unitación
0	1.000000
1	0.000000
2	0.360384
3	0.640576
4	0.360384
5	0.000000
6	1.000000

$$f(\vec{x}) = \sum_{i=1}^{k} fitness_{subp_i} \tag{4.1}$$

Problema de Corrector de Códigos

El problema de diseño de corrector de errores de códigos (ECC) [45] consiste en la asignación de palabras de código para un alfabeto que reduce al mínimo la longitud de los mensajes transmitidos y que proporciona la corrección máxima de errores de bit no correlacionados individuales, cuando los mensajes se transmiten a través de canales ruidosos.

Un código puede representarse formalmente tupla ternaria (n, M, d), donde n es la longitud (número de bits) de cada palabra de código, M es el número de palabras de código y d es la distancia de Hamming mínima entre cualquier par de palabras de código. Un código óptimo consiste en m palabras de código binarios, cada uno de longitud n, tal que d, la distancia mínima de Hamming entre cada palabra de código y todas las otras palabras de código, se maximiza. El criterio a optimizar es medir qué tan bien se colocan las palabras M en las esquinas de un espacio n-dimensional teniendo en cuenta la configuración de mínima energía de partículas M (donde d_{ij} es la distancia de Hamming entre las palabras i y j). La función de *fitness* a ser maximizada se muestra en la Ecuación 4.2.

$$F(C) = \frac{1}{\sum_{i=1}^{M} \sum_{j=1, i \neq j}^{M} \frac{1}{d_{ij}^2}} \tag{4.2}$$

Problema de Colville

La función de Colville [92] ha sido utilizada desde hace mucho tiempo en el campo de la optimización como función multimodal, permitiendo evaluar los resultados de diversas técnicas de optimización. Los detalles de este problema se encuentran descritos en la Tabla 4.2.

Tabla 4.2: Detalles del problema de Colville.

Característica	Dato
Tipo de función	Minimización
Función de *fitness*	$f(x_1 \cdots x_n) = 100(x_2 - x_1)^2 + (1 - x_1)^2 + 90(x_4 - x_3)^2$ $+(1 - x_3)^2 + 10.1((x_2 - 1)^2 + (x_4 - 1)^2) + 19.8(x_2 - 1)(x_4 - 1)$
Dominio de búsqueda	$-10 \leq x_i \leq 10$
Mínimo global	mínimo en $f(1, 1, \cdots, 1) = 0$

Problema de la Mochila Multi-Dimensional

El problema de la mochila multidimensional (Multidimensional Knapsack Problem (MKP)), es un problema el cual se compone de un número arbitrario de mochilas (objetivos) cada una con capacidad (restricción) y un conjunto de elementos que tienen asociado un peso y un beneficio. La tarea consiste en encontrar un subconjunto de estos elementos que colocados dentro de las mochilas maximice el beneficio total teniendo en cuenta las capacidades de estas [44]. Formalmente el problema MKP se define en la Ecuación 4.3 y 4.4 de la siguiente manera:

$$\text{Maximize} \quad \sum_{j=1}^{n} c_j x_j \tag{4.3}$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij}x_j \leq b_i \qquad i = 1,...,m \tag{4.4}$$
$$x_j \in \{0,1\} \qquad j = 1,...,n$$

tal que n es el número de elementos, m es el número de mochilas, cada una con capacidad b_i ($i = 1,2,...,m$), teniendo en cuenta que $c \in \mathbb{N}^n$, $A \in \mathbb{N}^{m \times n}$ y $b \in \mathbb{N}^m$. Cada item j ($j = 1,2,...,n$) rinde c_j unidades de ganancia y consume una cantidad dada de recursos a_{ij} para cada mochila i. El objetivo es encontrar un vector $x = (x_1, x_2, ..., x_n) \in \{0.1\}^n$, tal que las restricciones de capacidad descritas en la Ecuación 4.4 son satisfechas y para la cual $f(x) = (f_1(x), f_2(x), ..., f_n(x))$ respecto de la Ecuación 4.3.

Problema de Suma de Subconjuntos

Los problemas de suma de subconjuntos (en inglés Subset Sum Problem, SSP) son una clase especial de problemas de las mochilas binarias. Para el trabajo durante esta tesis se consideró la siguiente función de optimización:

Dados un conjunto $W = \{w_1, w_2, w_3, ..., w_n\}$ de n números positivos y otro número positivo C. Se trata de encontrar un vector $X = (x_1, x_2, ..., x_n)$ donde $x_i \, \epsilon \, \{0,1\}$, de forma que cumpla con la Ecuación 4.5. Se usa la misma codificación y función objetivo sugerida en [68]. Para garantizar que todas las soluciones no factibles tengan un valor de *fitness* más grande (peor) que aquellas factible se usa una función penalidad definida en la Ecuación 4.6, donde $s = 1$ cuando $\vec{x}$ es factible ($C - \vec{x} \geq 0$), y $s = 0$ cuando $\vec{x}$ es no factible.

$$f(\vec{x}) = \sum_{i=1}^{n} w_i x_i \leq C \text{ con } i = 1,2,...,n \tag{4.5}$$

$$f(\vec{x}) = s * (C - P(\vec{x})) + (1 - s) * P(\vec{x}) \tag{4.6}$$

Problema del MAX-CUT

El problema del MAX-CUT consiste en encontrar una partición de los vértices de un grafo ponderado en dos subconjuntos disjuntos, de tal forma que se maximiza la suma de los pesos de las aristas que van de un subconjunto al otro.

Sea $G = (V, E)$ un grafo con conjunto de vértices $V(|V| = n)$ y conjunto de aristas $E(|E| = m)$. Sea w_{ij} el peso asociado a cada arista $(i, j) \in E$. En este contexto, un corte $(S, S")$ es una partición de V en dos subconjuntos $S, S' = VS$ el valor de dicho corte $cut(S, S')$ viene dado por la Ecuación 4.7. El problema del MAX-CUT consiste en encontrar un corte con valor máximo sobre G.

$$cut(S, S') = \sum_{\substack{u \in S \\ v \in S'}} w_{uv} \tag{4.7}$$

Problema de Grienwak

El problema de Griewak define un problema de minimización sobre un espacio multidimensional. El parámetro d controla la profundidad de los mínimos locales; cuanto más grande sea éste más se parecerán los mínimos locales al mínimo global. Este problema se encuentra descrito en la Tabla4.3.

Tabla 4.3: Detalles del problema de Grienwak.

Característica	Dato
Tipo de función	Minimización
Función de *fitness*	$f(x_1 \cdots x_n) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}})$
Dominio de búsqueda	$-512 \leq x_i \leq 512$
Mínimo global	mínimo at $f(0, \cdots, 0) = 0$

Problema de Rosenbrock

El mínimo global se encuentra dentro de un largo y estrecho valle parabólico donde encontrar dicho valle es trivial, pero no así el mínimo global. Los detalles de este problema se encuentran descritos en la Tabla 4.5.

Tabla 4.4: Detalles del problema de Rosenbrock.

Característica	Dato
Tipo de función	Minimización
Función de *fitness*	$f(x_1 \cdots x_n) = \sum_{i=1}^{n-1}(100(x_i^2 - x_{i+1})^2 + (1 - x_i)^2)$
Dominio de búsqueda	$-2.048 \leq x_i \leq 2.048$
Minino global	mínimo at $f(1, 1, \cdots, 1) = 0$

Problema de Rastrigin

El problema generalizado de Rastrigin es un ejemplo típico de función multimodal no lineal. Inicialmente fue propuesto por Rastrigin [31] como una función bidimensional, siendo posteriormente generalizado por Mühlenbein [88]. Esta función supone un problema difícil de optimizar debido a su amplio espacio de búsqueda así como por el gran número de mínimos locales. Los detalles de este problema se encuentran descritos en la Tabla 4.5.

Tabla 4.5: Detalles del problema de Rastrigin.

Característica	Dato
Tipo de función	Minimización
Función de *fitness*	$f(x_1 \cdots x_n) = 10n + \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i))$
Dominio de búsqueda	$-5.12 \leq x_i \leq 5.12$
Minino global	minimum at $f(0, \cdots, 0) = 0$

4.2. Análisis y evaluación de los resultados

Como ya se ha comentado en varias ocasiones a lo largo de este documento, las metaheurísticas son técnicas no deterministas. Esto implica que diferentes ejecuciones del mismo algoritmo sobre un problema dado no tienen por qué encontrar la misma solución. Esta propiedad característica de las metaheurísticas supone un problema importante para los investigadores a la hora de evaluar sus resultados y, por tanto, a la hora de comparar su algoritmo con otros existentes.

Existen algunos trabajos que abordan el análisis teórico para un gran número de heurísticas y problemas [55, 66], pero dada la dificultad que entraña este tipo de análisis teórico, tradicionalmente se analiza el comportamiento de los algoritmos mediante comparaciones empíricas. Para ello es necesario definir indicadores que permitan estas comparaciones. Por un lado, tenemos aquellos que miden la calidad de las soluciones obtenidas. Dado que a lo largo del desarrollo de esta tesis se abordaran diferentes problemas de optimización, hay que considerar distintos indicadores de calidad. Por otro lado, están los indicadores que miden el rendimiento de los algoritmos y que hacen referencia a los tiempos de ejecución o a la cantidad de recursos computacionales utilizados. Aunque la discusión que incluimos en los párrafos siguientes trata los dos tipos de indicadores por separado, ambos están íntimamente ligados y suelen utilizarse conjuntamente para la evaluación de metaheurísticas, ya que el objetivo de este tipo de algoritmos es encontrar soluciones de alta calidad en un tiempo razonable.

Una vez definidos los indicadores, hay que realizar un mínimo de ejecuciones independientes del algoritmo para obtener resultados estadísticamente consistentes. Un valor de 30 suele considerarse el mínimo aceptable, aunque valores de 100 son recomendables.

4.2.1. Indicadores de calidad y rendimiento

Estos indicadores son los más importantes a la hora de evaluar una metaheurística. Son distintos dependiendo de si se conoce o no la solución óptima del problema en cuestión (algo común para problema clásicos de la literatura, pero poco usual en problemas del mundo real).

Para instancias de problemas donde la solución óptima es conocida, es fácil definir un indicador de calidad para controlar la calidad de la metaheurística: el número de veces que aquella se alcanza (*hit rate*). Esta medida generalmente se define como el porcentaje de veces que se alcanza la solución óptima respecto al total de ejecuciones realizadas. Desgraciadamente, conocer la solución óptima no es un caso habitual para problemas realistas o, aunque se conozca, su cálculo puede ser tan pesado computacionalmente que lo que interesa es encontrar una buena aproximación en un tiempo menor. De hecho, es común que los experimentos con metaheurísticas estén limitados a realizar a lo sumo un esfuerzo computacional definido de antemano (visitar un máximo número de puntos del espacio de búsqueda o un tiempo máximo de ejecución). En estos casos, cuando el

óptimo no es conocido, se suelen usar medidas estadísticas del indicador correspondiente. Las más populares son la media y la mediana del mejor valor de *fitness* encontrado en cada ejecución independiente. En general, es necesario ofrecer otros datos estadísticos como la varianza o la desviación estándar, además del correspondiente análisis estadístico, para dar confianza estadística a los resultados.

En los problemas donde el óptimo es conocido se pueden usar ambas métricas, tanto el número de éxitos como la media/mediana del *fitness* final (o también del esfuerzo). Es más, al usar ambas se obtiene más información: por ejemplo, un bajo número de éxitos pero una alta precisión indica que raramente encuentra el óptimo pero que es un método robusto.

Entendemos por medidas de rendimiento aquellas que hacen referencia al tiempo o la cantidad de recursos computacionales utilizados por las metaheurísticas, que se suelen medir en base al número de soluciones visitadas del espacio de búsqueda (esfuerzo computacional) o al tiempo de ejecución. Muchos investigadores prefieren el número de evaluaciones como manera de medir el esfuerzo computacional, ya que elimina los efectos particulares de la implementación, del software y del hardware, haciendo así que las comparaciones sean independientes de esos factores. Pero esta medida puede ser engañosa en algunos casos, ya que puede ocurrir que algunas evaluaciones tarden más que otras (algo muy común en programación genética [70]) o incluso que los operadores que manipulan las soluciones sean más costosos en una técnica u otra. En general, es recomendable usar las dos métricas (evaluaciones y tiempo) para obtener una medida realista del esfuerzo computacional.

Dado que vamos a evaluar algoritmos que se pueden ejecutar sobre plataformas de cómputo paralela, a continuación presentamos los indicadores que se han utilizado. En este sentido, el indicador más importante para los algoritmos paralelos es sin ninguna duda el *speedup*, que compara el tiempo de la ejecución secuencial con el tiempo correspondiente para el caso paralelo a la hora de resolver un problema. Si notamos por T_m el tiempo de ejecución para un algoritmo usando m procesadores, el *speedup* es el ratio entre la ejecución más rápida en un sistema mono-procesador T_1 y el tiempo de ejecución en m procesadores T_m:

$$s_m = \frac{T_1}{T_m} \tag{4.8}$$

4.3. Análisis estadístico de los resultados

Una vez definidos los indicadores de calidad y rendimiento, y realizadas las, al menos, 30 ejecuciones independientes, tenemos un conjunto de datos por cada indicador. Desde el punto de vista estadístico, estos datos se pueden considerar como muestras de una función de densidad de probabilidad y, para poder sacar conclusiones correctas, se ha realizado el siguiente análisis estadístico [36, 111].

Primero aplicamos un test de Kolmogorov-Smirnov para determinar si los valores obtenidos siguen una distribución normal (de Gauss). Si la siguen, utilizamos el test de Levene para comprobar la homocedasticidad de las muestras (igualdad de las varianzas). Si este test es positivo (las varianzas son iguales), se realiza un test ANOVA; en caso contrario, usamos el test de Welch. Para distribuciones no gaussianas, empleamos el test no paramétrico de Kruskal-Wallis para comparar las medianas de los resultados. El nivel de confianza en todos los tests es de 95 % (i.e., un nivel de significancia de 5 % o p-value por debajo de 0,05). La Figura 4.2 muestra gráficamente el procedimiento seguido. Cuando la comparación de resultados para un problema concreto involucra a más de dos algoritmos, también utilizamos un test de comparaciones múltiples [62].

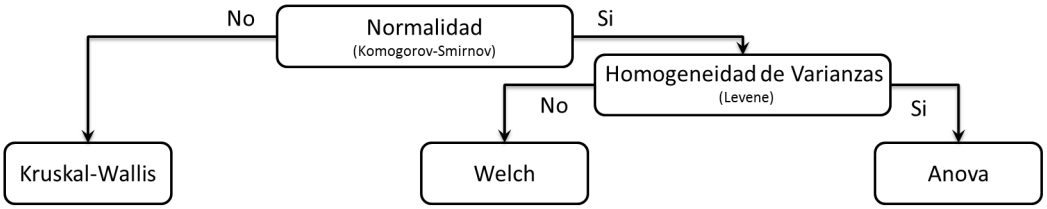


Figura 4.2: Análisis estadístico de los experimentos.

4.4. Entorno de ejecución

Las diferentes aproximaciones algorítmicas propuestas durante el transcurso de esta tesis han sido evaluadas mediante diferentes experimentos configuraciones hardware particulares. Cada una de ellas se describe en la Tabla 4.6

Tabla 4.6: Entornos de ejecución utilizados en nuestra experimentación.

Configuración	CPU/GPU	Núcleos	Memoria	Disco Duro	Sistema Operativo
Configuración 1	Intel(R) i7 CPU 920 @ 2.67GHz GeForce 8600m GT	4 Núcleos 240 Núcleos	8 GB 1 GB	Seagate 500 GB 7200 RPM)	Ubuntu 11.04 (natty) 64bits
Configuración 2	Amd Bulldozer Fx 8150 GeForce GTX 680	8 Núcleos 1536 Núcleos	8 GB 2 GB	WD 500gb Sata3 Caviar Black 64mb	Ubuntu 12.04 (Precise Pangolin) 64bits
Configuración 3	AMD FX(tm)- 8320 GeForce GTX 780 Ti	8 Núcleos 2880 Núcleos	8 GB 3 GB	SSD Seagate 256 GB)	Ubuntu 14.04 (Trusty Tahr) 64bits

4.5. Conclusiones

En este capítulo hemos analizado los componentes que intervienen durante la evaluación y análisis de los resultados obtenidos durante esta tesis. Se detallan los los problemas que se pretende resolver en esta tesis. Después, se ha explicado la evaluación de los resultados obtenidos, que indicadores son utilizados y su forma de implementación. Posteriormente, se detalla procedimiento utilizado para comparar de forma estadísticamente correcta los resultados obtenidos. Para terminar, se da un breve resumen de las plataformas hardware utilizadas y las diferentes características existentes entre las mismas.

Parte

2

Propuestas algorítmicas

Capítulo 5

Propuestas

En este capítulo formalizaremos nuestras propuestas de algoritmos paralelos sobre GPUs cuyos fundamentos, necesidad, características globales y pseudocódigo básico son explicados.

En la Sección 5.1 se presentan un conjunto de consideraciones iniciales sobre los algoritmo Genético Celulares (cGAs, por sus siglas en inglés, Cellular Genetic Algorithms). A continuación introducimos la adaptación de los principales procesos de un cGA a la GPU. Esto nos permitirá describir el alcance de la propuesta y exponer los beneficios y problemas que cabe esperar. Posteriormente, se muestra una aproximación cGA mediante una arquitectura de múltiples GPUs. Se indican las diferencias más importantes que se pueden encontrar con respecto a la primera implementación presentada.

La Sección 5.2 contiene el estudio sobre una aproximación original de búsqueda diseñada especialmente para funcionar sobre una plataforma GPU basada en las características de la Computación Sistólica. Luego, se provee un descripción de los diferentes modelos derivables del modelo canónico.

A partir del algoritmo propuesto en la Sección 5.2 se presenta un nuevo modelo que utiliza la idea de paralelismo híbrido como concepto de partida.

En la Sección 5.3 se discute el diseño de una técnica de paralelismo híbrido la cual utiliza algoritmos capaces de sacar provecho de las cualidades intrínsecas de cada arquitectura (CPU y GPU). Se discute detalles sobre el sistema de mejora de la búsqueda a través de la cooperación entre los algoritmos. Posteriormente, se analizan consideraciones importantes sobre la implementación. Finalmente, las conclusiones del capítulo son resumidas en la Sección 5.4.

5.1. Propuesta de paralelización basada en una metaheurística de población

La disposición/distribución de las soluciones es un factor clave a la hora de paralelizar metaheurísticas poblacionales. Si analizamos la distribución del cómputo de una GPU, el modelo de cGA se adapta perfectamente a las necesidades de la arquitectura GPU sin tener que distribuir físicamente cada solución y evitar la pérdida de tiempo por comunicaciones entre los diferentes nodos. En esta sección se presenta una introducción sobre el cGA y sus características, para posteriormente exponer nuestra aproximación celular sobre GPUs.

5.1.1. Algoritmo Genético Celular (cGA)

Un cGA es un tipo de Algoritmo Genético, donde la población está descentralizada y estructurada de acuerdo a una topología definida usualmente como un grafo de soluciones interconectadas. Habitualmente las soluciones están situadas en una malla toroidal (aunque el número de dimensiones puede ser extendido fácilmente a tres o más dimensiones). Cada solución interactúa sólo con sus vecinos más cercanos resultando en la conformación de un vecindario. A medida que el algoritmo itera, los vecindarios se solapan permitiendo la difusión de las mejores soluciones entre los vecindarios. La explotación tiene lugar dentro de cada vecindario mediante la utilización de operadores genéticos. La Figura 5.1 nos permite apreciar la selección de un vecindario, en este caso el vecindario seleccionado está compuesto por la solución actual en el centro sumado a los vecinos localizados al Norte-Sur-Este-Oeste.

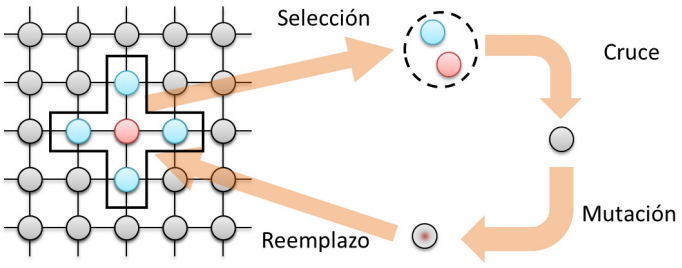


Figura 5.1: Proceso de evolución dentro de una población celular. Una solución actual y su vecindario a los cuales se aplican operadores genéticos.

El proceso de iteración implica evolucionar la población de soluciones siguiendo los pasos descritos a continuación:

- **Inicialización**. La población inicial de soluciones candidatas se genera usualmente de forma uniformemente aleatoria, aunque se puede incorporar fácilmente conocimiento del problema u otro tipo de información en esta fase.

- **Evaluación**. Una vez inicializada la población, o cuando se crea una nueva solución descendiente, es necesario calcular el valor de *fitness* de las soluciones candidatas.
- **Selección**. Mediante la selección se trata de dar preferencia en la evolución a aquellas soluciones con un mayor valor de *fitness*, imponiéndose así un mecanismo que permite la supervivencia de los mejores individuos. Existen muchos procedimientos de selección propuestos para cumplir con esta idea [26, 53, 108].
- **Cruce o Recombinación**. Combina partes de dos o más soluciones padres para crear nuevas soluciones (descendientes), que son posiblemente mejores y no serán idénticas a ninguno de los padres, sino que contendrá información combinada de los ellos.
- **Mutación**. La mutación modifica una única solución de forma aleatoria. En otras palabras, la mutación realiza un camino aleatorio en la vecindad de una solución candidata.
- **Reemplazo**. La población descendiente que creamos mediante la selección, recombinación y mutación reemplaza a la población de padres siguiendo algún criterio determinado.

Algoritmo 5.1 Pseudocódigo de un cGA.

```
1:  P ← generarPoblaciónInicial();
2:  evaluar(P);
3:  Mientras no se cumpla la condición de parada hacer
4:     Para cada solución i de P hacer
5:        vecindario ← seleccionarVecindario(i,P);
6:        padres ← seleccionarPadres(vecindario);
7:        hijos ← aplicarOperadoresGenéticos(padres);
8:        evaluar(hijos);
9:        P' ← seleccionarMejorHijo(hijos);
10:    Fin Para
11:    P ← remplazarPoblación(P, P');
12: Fin Mientras
13: seleccionar la mejor solución encontrada en P
```

En el Algoritmo 5.1 presentamos el pseudocódigo de un cGA canónico. Se puede observar que tras la generación y evaluación de la población inicial, los operadores genéticos anteriormente mencionados (selección de los padres,su recombinación, mutación del (los) descendiente(s) y el reemplazo de la solución actual por un descendiente) son aplicados a cada una de las soluciones dentro del entorno de sus vecindarios iterativamente hasta que se alcanza la condición de terminación. El pseudocódigo presentado en el Algoritmo 5.1 se corresponde con un cGA síncrono, puesto que las soluciones que formarán parte de la población de la siguiente generación (bien los nuevos descendientes generados o bien las soluciones de la población actual, dependiendo del criterio de

reemplazo) van almacenándose en una población auxiliar que tras cada generación reemplaza a la población actual. En este modelo todas las soluciones son actualizadas simultáneamente en la población. En el caso asíncrono, no sería necesario utilizar la población auxiliar, ya que los descendientes generados reemplazan a las soluciones de la población (según el criterio empleado) conforme van siendo seleccionados.

Los algoritmos celulares presentan características particulares debido a su funcionamiento y tratamiento sobre cada solución lo cual tiene una gran importancia por varias razones:

- Su estructura espacial interna crea muchos subconjuntos generando pequeños solapamientos lo cual permite que la diversidad existente en los valores de fitness y de los cromosomas perdure durante un mayor número de iteraciones.
- Soluciones de alta calidad se extienden gradualmente (difusión).
- Algunos trabajos [3, 4] han podido establecer las ventajas de utilizar algoritmos celulares para tareas complejas de optimización (alta eficacia y un reducido número de pasos) en relación a otras técnicas.
- La similitud entre GAs y autómatas celulares [89], sus aplicaciones potenciales, y su posibilidad de ser implementados en arquitecturas SIMD los hacen dignos de estudio.

Estas características hacen al cGA un algoritmo altamente paralelizable capaz de ser introducido en una arquitectura SIMD. Por todo esto, su implementación en arquitecturas GPU es directa, pudiéndose situar una o más soluciones por hilo dependiendo de la política de administración de cada una de ellas. La distribución de la población del cGA encuentra un mapeo directo con la estructura de hilos de la GPU. Además, la dependencia de soluciones solo se da con respecto al vecindario actual y no por la elección aleatoria de cualquier solución de la población celular. Todo esto nos permite definir un marco teórico de desarrollo para un cGA sobre una GPU.

5.1.2. cGA sobre GPU

Las características que presenta el algoritmo cGA tales como: disposición espacial de las soluciones (de tamaño $n \times m$), modelo de propagación de buenas soluciones y el equilibrio del modelo de búsqueda con respecto a la exploración/explotación, hacen que la implementación en arquitecturas masivamente paralelas sea de interés. Nuestro esfuerzo se ha centrado en el diseño de un modelo celular que permita sacar el mayor provecho de las GPUs. Se ha buscado mejorar el equilibrio entre la exploración y la explotación que se aplica sobre el espacio de búsqueda mediante el estudio de modelos paralelos eficientes.

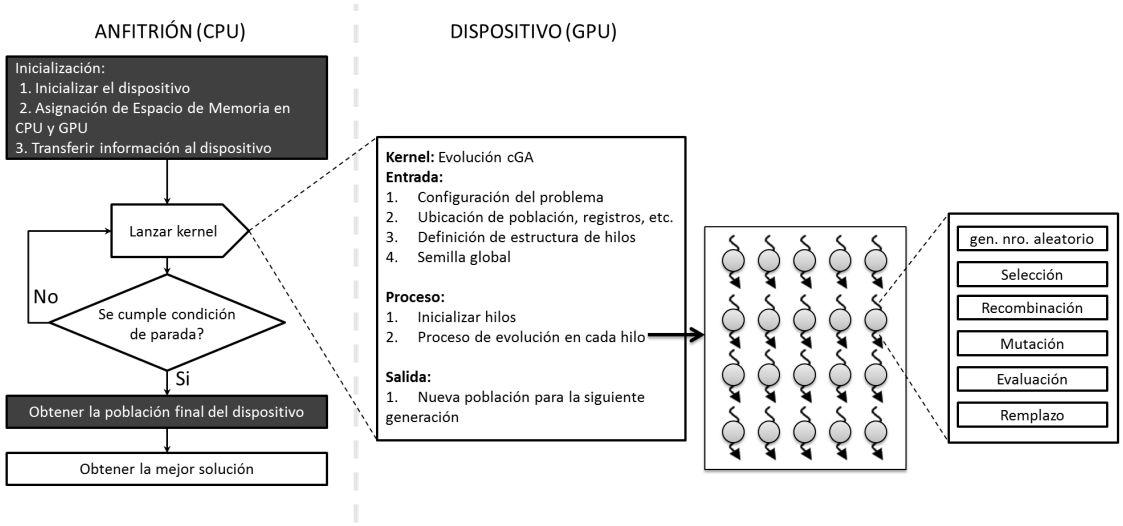


Figura 5.2: Descripción del proceso de ejecución del cGA en GPU.

La Figura 5.2 muestra gráficamente el proceso de funcionamiento para un cGA en una GPU (cGA-GPU) atendiendo a los procesos de evolución en un esquema celular. Primeramente, necesitamos realizar todo el proceso de inicialización en el cual se necesita configurar los espacios de memoria a utilizar en la CPU y la GPU. Seguidamente, se invoca un kernel para largar un conjunto de hilos los cuales controlaran la población en todo el proceso de evolución. De esta forma, con una población de tamaño $x \times y$ necesitaremos lanzar un número igual de hilos en la GPU. Una vez creada la estructura de hilos, cada solución es controlada por un hilo el cual realizara todos los pasos necesarios para la evolución de la solución actual. En la Figura 5.2 se puede observar una población de 5×4, la cual es administrada por una estructura de hilos del mismo tamaño. En cada hilo se produce las operaciones encuadradas que aparecen en la figura. Desde la generación de los números aleatorios, trabajando con los operadores genéticos, hasta el remplazo de la solución actual. El criterio de parada o de terminación del cGA-GPU es el mismo que el cGA secuencial.

Una característica que se evidencia al utilizar este tipo de arquitectura es que de existir una cierta sincronización, sería necesaria sólo al final de cada generación (lo que no es poco en términos de una ejecución en computadores SIMD o MIMD, ya que restringe el paralelismo). Esto supone que una implementación masivamente paralela síncrona en el que cada estructura reside en un procesador o *hilo* de la GPU genera una carga de comunicación muy importante al pedir copias de las estructuras designadas como vecinas (o al menos de sus valores de *fitness*). Esto implica que la decisión de una u otra política de comunicación y almacenamiento de información puede impactar directamente en el rendimiento del algoritmo paralelo.

Con respecto al uso de operadores, el algoritmo genético celular utiliza como base los operadores genéticos comunes existentes en la literatura. Para el caso de una implementación en GPU, los operadores trabajan de forma muy similar, salvo algunas excepciones.

En la Figura 5.2 se puede observar que los operadores buscan trabajar de la misma forma que en el modelo canónico secuencial. Con respecto a la creación de cada solución, se utiliza una semilla aleatoria por cada hilo. La generación de esta semilla se discute en la Sección 5.1.4. La evaluación se ejecuta de igual forma que en la CPU. Tanto para el operador de selección, recombinación y de mutación, cada hilo es responsable de generar el vecindario, seleccionar los padres, cruzarlos y aplicar mutación. El comportamiento de los operadores utilizados no es distinto al trabajado en el modelo secuencial, con lo cual, la comparación de eficiencia es factible y válida.

5.1.3. Modelo multi-GPU

Nuestro objetivo en esta sección es poder analizar y ampliar nuestra aproximación cGA de una sola GPU hacia un contexto de múltiples GPUs (cGA-multiGPU). Con el fin de hacer frente a las dificultades asociadas a un modo multi-GPU, un modelo multi-hilo en la CPU es utilizado para gestionar cada dispositivo GPU, con lo cual se plantea una interacción entre CPU y GPUs al estilo maestro-esclavo. En este modo, cada hilo de la CPU es responsable de un dispositivo GPU. Esta nueva aproximación se denomina cGA-multiGPU.

En primer lugar, se asocia cada hilo de CPU a una GPU en particular. Dado que el cGA trabaja con una malla toroidal, su representación permite dividir la población en sub-poblaciones. Cada una de ellas es almacenada en una GPU (ver Figura 5.3). Las GPUs almacenan sus respectivas sub-poblaciones en la memoria global. Con esta descomposición, aquellas soluciones ubicadas en la frontera de cada sub-población van a necesitar soluciones ubicadas en otras sub-poblaciones. Estas soluciones las denominamos vecinos frontera y están en el límite de la malla 2D de las sub-poblaciones de la GPU contigua. Por cuestiones propias de la arquitectura donde se trabaja (modelo de tarjeta) se debe copiar estas fronteras de vuelta al host cada vez que necesiten ser usadas, lo que en un principio podría añadir una sobrecarga adicional de comunicación para el trabajo de cómputo global del algoritmo. Una vez obtenido los vecinos frontera, el funcionamiento se da como una cGA en una sola GPU. Cuando termina la iteración debe utilizarse una barrera de sincronización para asegurarse de que cada GPU termina su trabajo y, finalmente, la recopilación de todos los datos y la transferencia a las otras GPUs se produce de forma síncrona. El proceso se repite hasta que un criterio de parada es satisfecho. Esto es lo que se puede observar en la Figura 5.3 donde las fronteras se intercambian entre las poblaciones y posteriormente la ejecución en cada GPU se comporta como un cGA paralelo definido en la sección anterior.

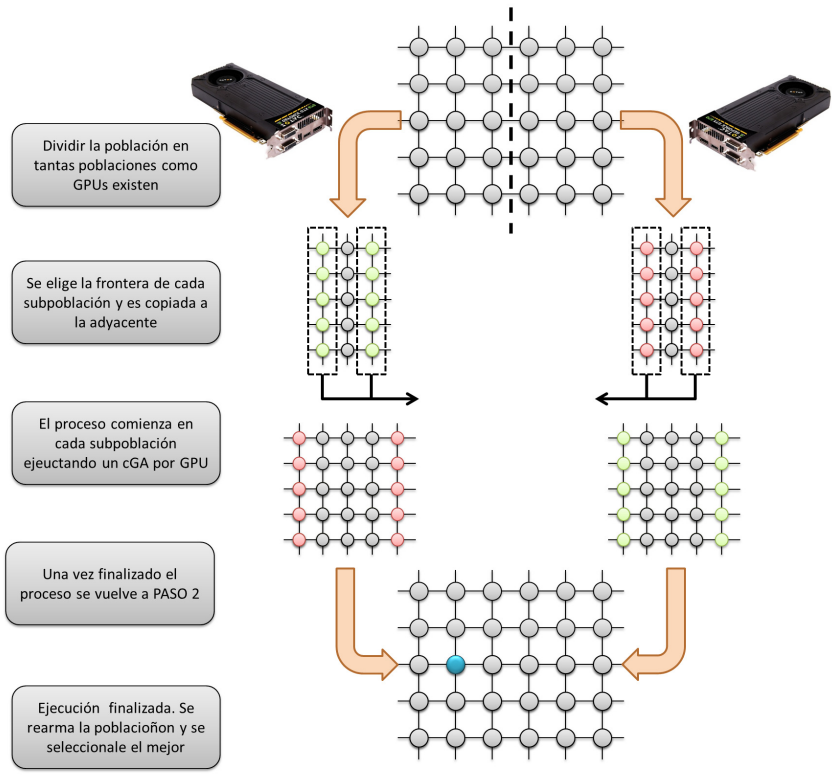


Figura 5.3: Descripción del proceso de ejecución del cGA para multiples GPUs.

El enfoque presentado proporciona una división de la malla-2D en sub-poblaciones y otorga al cGA una simplicidad de funcionamiento lo cual permite beneficiarnos de la integración de múltiples plataformas GPU de bajo costo y que aportan un gran poder de cómputo.

5.1.4. Detalles de implementación

Una vez formalizado los dos modelos (cGA-GPU y cGA-multiGPU) como familia de algoritmos evolutivos celulares sobre GPUs, hemos de abordar las principales decisiones de implementación que más pueden llegar a influir y caracterizar estos modelos. Respecto a la implementación en sí, cada uno de los algoritmos está desarrollado siguiendo un esquema de programación funcional. Este aspecto es fundamental dado que CUDA ofrece un conjunto muy limitado de opciones para trabajar con programación orientada a objetos.

El primer paso para trabajar con la GPU es reservar la memoria necesaria donde se alojarán la estructura utilizada para la población. Es importante resaltar que la GPU no posee un generador de números aleatorios, por lo tanto existe una serie de técnicas que pueden ser utilizadas para

generar estos números. Se ha utilizado un Generador Linear Congruencial (LCG, por sus siglas en inglés) [69] dada su simplicidad de uso y la calidad de los números generados. Su funcionamiento está basado en enviar una semilla global desde la CPU a la GPU y cada hilo al recibirla generara una semilla propia mediante modificaciones de desplazamientos de bits.

En cuanto a la implementación de los modelos paralelos, es evidente que el sistema de comunicación entre la CPU y la GPU es de vital importancia. Su eficiencia y flexibilidad deben ser muy elevadas para poder ser usado con garantías de que no restringe por él mismo la aplicación del algoritmo resultante. Igualmente importantes son tanto las estructuras de datos intercambiadas (para configuración o durante la ejecución) como el conjunto de reglas que regulan dicha comunicación (protocolo). En este caso, está determinado por el ancho de banda que presenta el bus de conexión de la tarjeta con la placa madre. Por ello, las soluciones se trabajan a nivel de bit en el caso de problemas combinatorios, y para problemas de tipo continuo se ha buscado trabajar con representaciones que no puedan exceder el volumen de memoria global de la GPU.

5.2. Propuesta de nuevo modelo algorítmico especialmente diseñado para GPU

En esta sección proponemos un nuevo algoritmo de optimización especialmente desarrollado para ser ejecutado sobre una plataforma GPU. Tomamos como concepto de partida la idea de *Computación Sistólica* y las ventajas que esta ofrece al trabajar en arquitecturas SIMD, generando un nuevo modelo algorítmico denominado *Algoritmo de Búsqueda Sistólica por Vecindario* (Systolic Neighborhood Search, SNS).

La Computación Sistólica fue propuesta inicialmente en la Universidad Carnegie-Mellon por Kung [76, 77]. Es un modelo real que fusiona los conceptos de *pipelining*, paralelismo y conexión entre elementos o unidades de procesamiento. Un sistema sistólico (como el que puede apreciarse en la Figura 5.4) utiliza un vector de celdas (elementos que pueden ser de tipo *hardware* o *software*) interconectados entre sí llamado vector sistólico. Un sistema sistólico puede estar conformado por uno o más vectores sistólicos. El trabajo que realiza cada celda es el mismo y se centra en ejecutar operaciones que no impliquen demasiada complejidad, por ejemplo suma o multiplicación. Finalizada cada operación, la información se traslada a la celda contigua para volver a realizar la misma operación.

El trabajo de este tipo de sistemas se basan en tres puntos básicos: la topología de la estructura sistólica, el cálculo en cada componente (celda) y la dirección del flujo de datos. En primer lugar, cada algoritmo sistólico debe definir la naturaleza exacta de la red, que indica la propagación de la información a través de toda la estructura permitiendo una comunicación regular y fluida. En

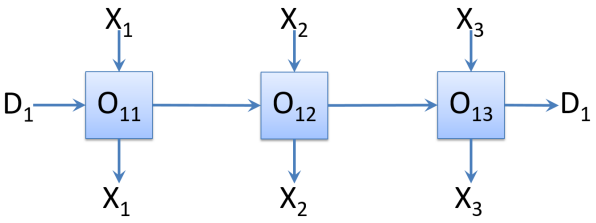


Figura 5.4: Estructura de un sistema sistólico, con la definición de cada celda y el flujo de información.

segundo lugar, para cada componente se realiza el mismo cálculo con la información entrante de una manera repetitiva, regular y simple. Por último, cada celda puede compartir la información con sus vecinos inmediatamente después de procesar de manera síncrona, explotando concurrencia y reduciendo en gran medida el tiempo de ejecución global del algoritmo.

La aplicación de la Computación Sistólica a los problemas de optimización mediante metaheurísticas ha sido escasa durante las últimas décadas. En la literatura se han registrado apenas algunas implementaciones en el campo de optimización [20, 86] sin mayor notoriedad ni continuación del trabajo debido a los inconvenientes de programar sobre *hardware*. De la misma forma, por el alto coste de implementación de esta aproximación, el actual estado del arte y la gran explosión de las GPUs con su arquitectura de trabajo SIMD hace factible trabajar con una aproximación de computación sistólica para la resolución de problemas complejos del tipo NP-duro.

La principal contribución de esta sección es un novedoso algoritmo que utiliza las ventajas de la computación sistólica para optimizar la búsqueda en un determinado espacio de soluciones. La búsqueda se realiza por medio del movimiento de soluciones a través de una malla toroidal, en la cual cada celda realiza una simple operación de perturbación siguiendo unos criterios predefinidos. Se pretende así explotar sobre las soluciones todas aquellas más prometedoras de la población, mientras que la diversidad tiende a mantenerse mediante el movimiento de estas soluciones a través de la celdas.

5.2.1. Esquema de la Búsqueda Sistólica por Vecindario (SNS)

La idea esencial es trabajar de la misma forma que una red de vectores sistólicos, a través de una malla toroidal donde las soluciones se encuentran localizadas en cada celda. Cada una de ellas trabaja realizando una simple operación de perturbación sobre una solución y posteriormente la mejor entre la original y la modificada es trasladada a la celda contigua para realizar el mismo proceso. Las soluciones se mueven a través de toda la fila sufriendo múltiples cambios en cada celda de acuerdo a un patrón predefinido.

El potencial del algoritmo se basa en la exploración del espacio de búsqueda de manera eficiente y estructurada mediante la ejecución de continuas modificaciones selectivas en las soluciones de manera sistemática.

La Figura 5.5 muestra una población de $(m \times n)$ rectángulos pequeños distribuidos en una malla toroidal de $m \times n$ celdas (cuadrados grandes). Esta estructura se entiende mejor como dos mallas: una dinámica que consistiría en las soluciones moviéndose a través de una segunda, un conjunto de celdas estático que forman un grupo de vectores sistólicos. Cada celda se encarga de recibir, modificar, evaluar y enviar una solución ($sol_{(x,y)}$) a la siguiente celda. La malla toroidal permite a aquellas soluciones que han llegado al final de la fila seguir siendo modificadas según el patrón de perturbación en la primera celda de la misma fila. En la Figura 5.5 podemos observar todo este proceso mediante la celda aumentada ubicada a la izquierda de la figura.

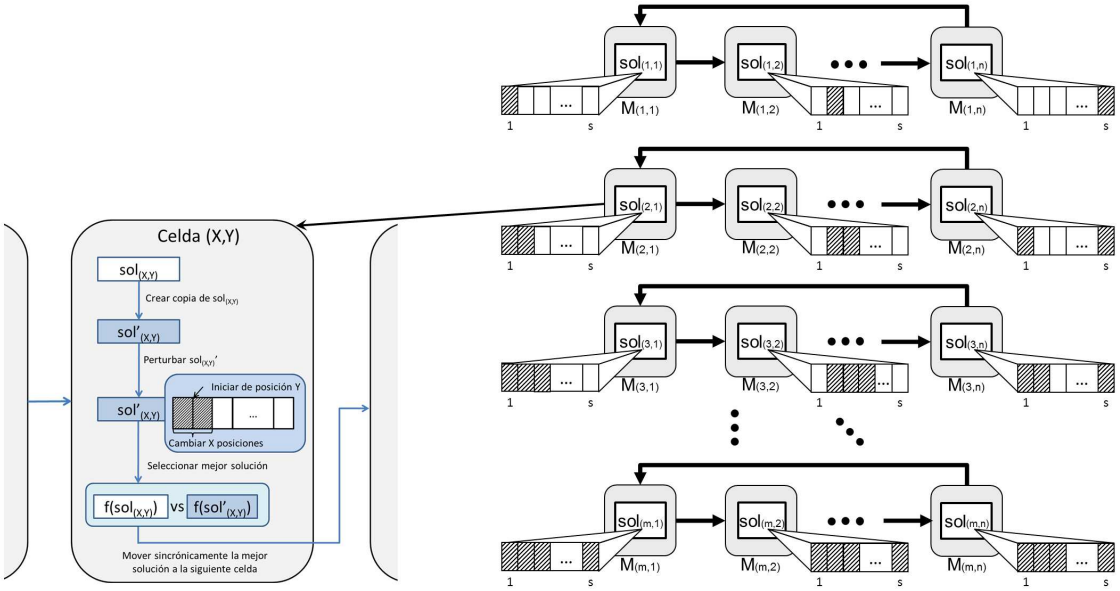


Figura 5.5: Distribución de soluciones dentro de la malla del SNS. Proceso de ejecución de cada solución dentro de cada celda.

En el Algoritmo 5.2 se muestra el pseudocódigo de un SNS canónico. El algoritmo SNS comienza definiendo el tamaño de la malla en concordancia al valor del tamaño del problema a resolver (definido como s). Una vez definido la malla M se procede a inicializar las soluciones que estarían localizadas inicialmente en cada una de las celdas de M. Cada solución $sol_{(x,y)}$ es generada y evaluada en paralelo (linea 3-7). Seguidamente, el algoritmo comienza a iterar hasta cumplir un criterio de parada. Cada iteración de un SNS consiste exactamente en ejecutar una serie de procesos.

Algoritmo 5.2 Pseudocódigo del SNS canónico.

```
1: Definir variables m y n dependiente del tamaño s del problema
2: Definir malla M de tamaño n × m
3: Para todo sol_(x,y) ∈ M hacer en paralelo
4:    sol_(x,y) ←generarSolucion(sol_(x,y));
5:    sol_(x,y) ←evaluarSolucion(sol_(x,y));
6: Fin Para todo
7: Mientras (no se cumpla criterio de parada) hacer
8:    Para todo sol_(x,y) ∈ M hacer en paralelo
9:       Calcular la posición (x, y) para cada sol_i
10:      sol'_i ← sol_i
11:      Para init = 0 a (x − 1) hacer
12:         sol'_i ← aplicar operaciones de cambio sobre la posición (y+posInicio) en el vector de valores
13:      Fin Para
14:      sol'_i ← parcial_eval(sol'_i)
15:      Si sol'_i mejor que sol_i Entonces
16:         sol_i ← sol'_i
17:      Fin Si
18:      esperar la terminación de todas las operaciones en todas las celdas (entre lineas 8-17)
19:      calcular sigPosicion = (y + 1) mód n
20:      sol_(x,sigPosicion) = sol_(x,y)
21:   Fin Para todo
22: Fin Mientras
23: obtenerMejorSolucionDe(M)
```

El primer paso del Algoritmo 5.2 es crear una copia ($sol'_{(x,y)}$) con respecto a la solución actual (linea 10). Luego, se aplica el proceso de perturbación (linea 11-13). Finalmente se realiza una evaluación parcial de la copia. Finalmente, en la etapa de reemplazo, si la solución actual es mejor que la copia quedara la actual, caso contrario esta sera reemplazada por la copia (linea 15-17). Para esta aproximación hemos propuesto el uso de una política de actualización de celdas síncrona (linea 18), aunque también podríamos haber utilizado alguna política asíncrona. El aspecto asíncrono no lo vamos a estudiar aquí por las muchas implicaciones y que no tiene injerencia directa con el modelo de computación sistólica. Como paso final, definimos la celda a la cual transmitiremos la solución (linea 19) y realizamos el movimiento (linea 20). Pretendemos destacar mediante este pseudocódigo todos los componentes que intervienen. En particular, el SNS trabaja en base a la definición de tres componentes principales: una topología definida, la computación en las celdas y el flujo de la información. Cada uno de ellos es explicado a continuación.

Topología El SNS utiliza una malla toroidal para ubicar las soluciones. La Figura 5.5 muestra la disposición de las celdas y soluciones. La malla está definida por un tamaño $m \times n$. Las dimensiones se definen en base al número s de elementos dado por un problema que se va a resolver.

Cómputo en Celdas Una vez definida la malla, es necesario explicar la configuración de las operaciones realizadas en cada una de las celdas. Cada solución $sol_{(x,y)}$ esta conformada por un vector de elementos y en cada celda se ejecuta una operación simple de perturbación sobre un cierto grupo de elementos ese vector.

El proceso de perturbación se inicia creando la copia $sol'_{(x,y)}$ de la solución original $sol_{(x,y)}$. Los cambios que se producen sobre $sol'_{(x,y)}$ son definidos a partir de los parámetros (x, y) (es el índice para cada celda dentro de la malla M). El parámetro x define el número de valores a cambiar en cada vector, de esta forma, a medida que el número de la fila va aumentando, también lo hace el número de elementos que va a variar dentro de cada solución. El parámetro y indica la posición a partir de la cual se producirán los cambios. Estas variaciones pueden ser observadas de forma más clara en la Figura 5.5. Tenemos una solución ubicado en la posición $(x, y) = (2, 1)$ con lo cual se modificarán dos valores a partir de la posición 1. Este patrón de perturbación nos permite puntualizar la importancia de tener una malla con dimensiones iguales al tamaño del problema, que busque evaluar todos los posibles grupos de perturbaciones sobre el vector solución, partiendo de cada posición dentro del mismo.

Una vez realizados los cambios, procedemos a re-evaluar $sol'_{(x,y)}$, usando una función de evaluación parcial para no consumir tiempo de cómputo haciendo una re-evaluación completa de toda la solución. De esta manera, solo se evalúan aquellos sectores de la solución que han sufrido modificaciones. Obviamente, esto depende en gran medida de la complejidad de la función de evaluación y la dependencia entre los elementos del vector solución

Flujo de Información El SNS mueve cada solución a la siguiente celda de manera síncrona al mismo tiempo. De esta forma, al tener una malla con n columnas se pueden perturbar todos los grupos posibles de elementos dentro de las soluciones. Si bien la estructura de manera casi obligatoria define un modelo síncrono, la simplicidad del operador de perturbación permite que una fluidez de las soluciones.

5.2.2. Variaciones del modelo canónico

Si bien el modelo canónico es un excelente modelo de optimización para ser ejecutado sobre GPU, a partir de este se pueden derivar nuevos modelos y/o estrategias para mejorar la base del SNS. A continuación se exponen una serie de ellos que pueden ser usados en diferentes momentos de ejecución del SNS.

5.2.2.1. SNS con perturbaciones de variabilidad exponencial (SNS^{exp})

Todas las modificaciones que realiza el SNS puede llegar a no ser suficiente para alcanzar o garantizar una cierta calidad de soluciones. Además ciertas GPUs pueden no lograr guardar y trabajar al mismo tiempo sobre un número elevado de soluciones lo cual puede afectar a la escalabilidad del SNS. Esta nueva aproximación denominada SNS^{exp} tiene la intención de reducir el números de filas con respecto a la original de SNS y así reducir el número de soluciones a ser

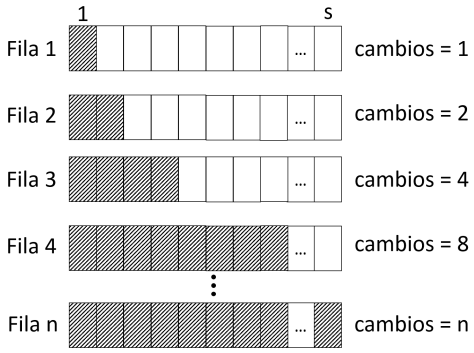


Figura 5.6: Número de perturbaciones por filas para la aproximación SNS^{exp}.

modificadas. Al reducir el número de filas podemos permitir un mayor número de iteraciones de las soluciones a través de las celdas. Sin embargo, no podemos usar el mismo mecanismo de los cambios incrementales por fila en las soluciones, por lo que, es necesario definir dos cuestiones: el número de filas que tendrá M y el número de elementos que van a ser perturbados por fila. La primer cuestión se calcula utilizando el tamaño del problema: utilizaremos un número de filas basados en la función $\log_2 k$. En segundo lugar, como SNS^{exp} no puede utilizar los cambios incrementales en las soluciones por fila, simplemente porque no hay suficientes filas para utilizar este mecanismo, utilizaremos la operación 2^x como número prefijado de cambios por filas de acuerdo con el cálculo previo de números de filas. El exponente x indica el valor de la fila actual donde se encuentra una solución dada. Sumado a esto, la primera fila siempre modificara un solo elemento con lo cual tenemos en total $((\log_2 k) + 1)$ filas. Con este enfoque, el número de valores a ser cambiado aumenta en múltiplos de dos. Estos cambios se pueden observar de mejor manera en la Figura 5.6 donde las filas van aumentando sus perturbaciones en potencias de a dos.

5.2.2.2. SNS con perturbaciones aleatorias ((r)SNS)

Hasta ahora, los modelos SNS y SNS^{exp} han sido diseñados para perturbar un grupo de elementos contiguos en cada solución. Cuando el algoritmo examina modificaciones contiguas, el espacio de búsqueda se reduce y los posibles cambios sobre otros elementos de la solución está restringido. Además, en algunos casos, los cambios en esos grupos pueden afectar a parte de la solución que proporciona un gran beneficio y que más tarde se habrán perdido. Por lo tanto, una variación en las reglas de selección de los elementos a perturbar es ser seleccionados al azar.

Por lo tanto, proponemos una nueva extensión llamada (r)SNS, la cual funciona de la misma forma que el SNS pero este método se centrará en hacer una simple selección aleatoria sobre los elementos de la solución, cada uno de ellos con la misma probabilidad.

La modificación utilizada en el (r)SNS puede ser relacionada con la línea 12 del Algoritmo 5.2, en lugar de cambiar los bits de forma secuencial ($y + posInicio$), (r)SNS modifica x elementos de una solución en forma aleatoria.

5.2.2.3. SNS utilizando menor cantidad de filas

Si bien el modelo canónico presenta un modelo de búsqueda exhaustivo, claro y eficiente, el hecho de que algunas soluciones tengan que modificar demasiada cantidad de información puede ser contraproducente para algunos problemas. Por ello, es necesario comprobar si la supresión de ciertas filas de M puede mejorar el rendimiento del algoritmo SNS. Para ellos definimos una estrategia de supresión: s

- **Eliminación de la última fila**. Dado que la última fila del modelo realiza cambios completos, esas soluciones van cambian demasiado lo cual en algunos problemas puede ocasionar perdida de sectores de la solución extremadamente buenos. Asimismo, para algunos problemas, el tiempo en cambiar todos los elementos de la solución y volver a evaluar significa una re-evaluación completa y puede generar una pérdida de tiempo excesivamente grande para ciertos contextos. Las Figura 5.7 muestra la aproximacion anteriormente explicada.

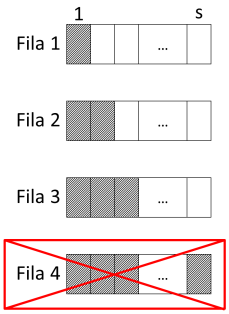


Figura 5.7: Eliminación de la ultima fila.

5.2.2.4. SNS con movimientos de filas

El flujo de información en el SNS se basa en el movimiento de soluciones a través de las celdas de una misma fila. Dependiendo del contexto de trabajo, puede existir momentos en que los continuos cambios utilizando el mismo patrón no genera saltos de calidad en las soluciones. Además, como hemos explicado en una sección anterior, la zona inferior de la estructura del SNS puede realizar cambios demasiados drásticos. Estas perturbaciones abarcan un enorme número de elementos en cada solución sin ninguna contribución de manera significativa al contexto global de optimización, provocando una convergencia prematura del *fitness* en estas filas.

Nuestro objetivo es promover el movimiento del grupo de soluciones localizadas en una fila a la siguiente situada por debajo de ella. Con este movimiento la red se transforma en una malla toroidal de abajo hacia arriba y de atrás hacia adelante. Al principio del proceso, el mecanismo es el mismo que sucede con el SNS canónico, pero cuando las soluciones han recorrido un número determinado de movimientos a través de las celdas de una misma fila, todas las soluciones se desplazan a la siguiente de abajo para continuar el proceso de perturbación con el patrón de esa nueva fila. Este método se define como SNS con movimiento de filas ((mr)SNS).

5.2.2.5. Combinación de modelos

La diversidad de variaciones permite una combinación basados en los modelos anteriormente descritos. Esto nos ofrece una gama de situaciones para analizar y mejorar. Los modelos a combinar pueden formarse de combinar una o más técnicas. En concreto, buscamos analizar si las variaciones puede responder a cierto tipo de problemas o si la combinación de ellas pueden resolver problemas de una manera rápida sin perder resolución numérica.

5.2.3. Detalles de implementación

Una vez definida la familia de algoritmos SNS, algunos detalles de implementación deben ser tenidos en cuenta para trabajar sobre GPU.

La ejecución es realizada enteramente sobre la GPU, la CPU es utilizada como mera comunicadora de parámetros y como recepcionista de la mejor solución final encontrada con lo cual estamos maximizando el uso de la GPU.

Para inicializar las soluciones y trabajar posteriormente en el proceso de evolución de las soluciones, sobre todo para el caso de $(r)SNS$, utilizamos un LCG usado en trabajos anteriores comparativos de generación de números aleatorios sobre diferentes arquitecturas [78, 120]. Esta aproximación permite a cada hilo generar sus propias semillas sin necesidad de recibir bloques de números aleatorios o estar constantemente recibiendo información desde la CPU.

5.3. Propuesta de modelo híbrido de optimización mediante cooperación entre plataforma CPU-GPU

En un sistema heterogéneo CPU-GPU orientado a resolver problemas complejos a gran escala, usualmente sólo un hilo de la CPU es asignado para controlar la ejecución en GPU. El resto de los núcleos de la CPU están en estado de reposo, mientras que la GPU lleva a cabo la ejecución de las tareas asignadas. Todo esto lleva a desperdiciar una gran cantidad de recursos disponibles en la CPU y de la misma forma a implementar métodos que no se adaptan de manera integral a la

arquitectura de la GPU. Por lo tanto, es muy importante encontrar un método de optimización eficiente numéricamente que haga pleno uso de todos los recursos computacionales disponibles de arquitecturas heterogéneas con CPUs y GPUs, lo cual abarca una gran cantidad de retos.

En el campo de la computación evolutiva los últimos avances muestran claramente que las metaheurísticas pueden obtener buenos resultados para diferentes problemas cuando son hibridados con otras técnicas, como por ejemplo funciones de fitness adaptativas, operadores específicos, u optimización local. El uso de hibridación en un algoritmo de GPU supone el acoplamiento de otra técnica en esta misma arquitectura. Dicho lo anterior, parece sensato realizar una técnica híbrida en la cual cada uno de los componentes o métodos sea ejecutado en plataformas diferentes (CPU y GPU). En este sentido, el diseño de un algoritmo de optimización híbrido eficiente en una plataforma CPU-GPU abarca muchos desafíos a nivel *software* y *hardware*.

La motivación de esta sección es presentar el comportamiento de una técnica híbrida que tenga como método de búsqueda base en la GPU al SNS. Asimismo, incorporamos información del problema mediante el uso de operadores de recombinación, mutación y búsqueda local, los cuales son ejecutados en la CPU. El uso de estos operadores en la CPU es debido a su complejidad de implementación en la GPU y para no afectar el comportamiento de búsqueda del SNS. También, para beneficiarnos de la utilización de los núcleos de la CPU para tareas no paralelas. Esta técnica se denomina *Búsqueda Sistólica Híbrida* (HySyS, del inglés Hybrid Systolic Search).

En la Sección 5.3.1 presentamos nuestra nueva propuesta HySyS. La Sección 5.3.2 describe detalladamente de las técnicas usadas para la hibridación. Los detalles de implementación se presentan en la Sección 5.3.3.

5.3.1. Modelo de cooperación de Búsqueda Sistólica Híbrida (HySyS)

Proponemos una implementación híbrida en una arquitectura heterogénea CPU-GPU que es capaz de proporcionar un comportamiento robusto y cooperativo utilizando dos algoritmos de optimización complementarios.

Si bien las arquitecturas *hardware* son indispensables para el trabajo de esta técnica híbrida, queremos ir un paso más allá y proporcionar un modelo de partida, el cual demuestre la importancia de trabajar en ambas plataformas de una manera cooperativa y demostrar la eficiencia de cada una de ellas. Nuestro objetivo es mostrar una técnica de optimización híbrida que pueda utilizar los recursos computacionales de la CPU y GPU y que además ofrezca calidad en las soluciones generadas. Nuestras contribuciones en este apartado se pueden definir de la siguiente manera:

- Proponemos una implementación híbrida cooperativa mediante el modelo CPU-GPU para resolver de manera eficiente problemas de optimización en un sistema heterogéneo.

- Presentamos dos algoritmos de optimización complementándose entre sí seleccionados de acuerdo a sus mecanismos de trabajo en cada arquitectura con el fin de crear un modelo cooperativo que puede ser utilizado en las máquinas de uso personal.

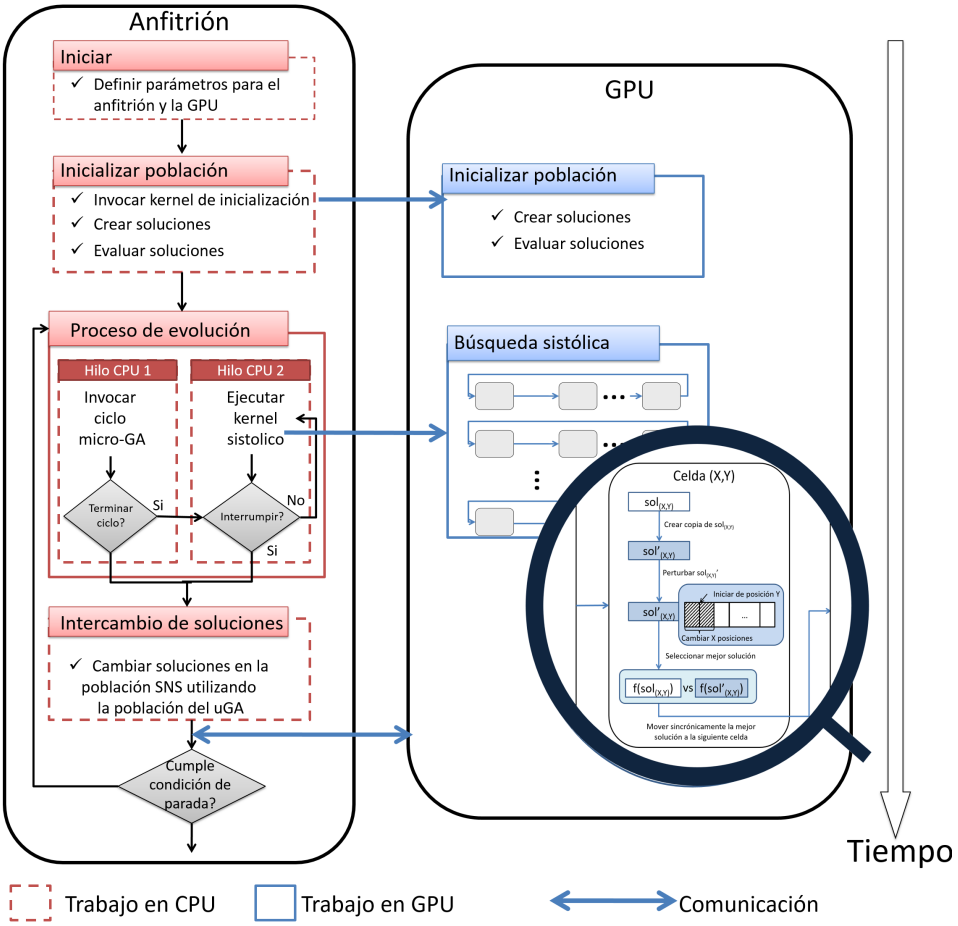


Figura 5.8: Proceso de ejecución del modelo HySyS.

En resumen, un hilo de CPU se dedica en primer lugar a la operación de transferencia de datos, configuraciones, y esta información es guardada en la memoria global de la GPU. Posteriormente, el segundo hilo CPU invoca el *kernel* CUDA y todos los hilos de GPU comienzan el proceso de búsqueda sistólica. El primer hilo de CPU arranca un algoritmo en la CPU en paralelo con la búsqueda sistólica. Esto permitirá trabajar con operadores genéticos sobre una población y que no demande un tiempo considerablemente de ejecución. Este proceso de ejecución se puede apreciar en la Figura 5.8. En nuestra búsqueda de un algoritmo para ejecutar en la CPU, primero debemos definir una lista de requisitos primordiales que debe cumplirse para ser utilizado dentro del marco de este enfoque híbrido:

- Debe ser un algoritmo rápido y de un proceso de ejecución liviano, capaz de interactuar con la GPU varias veces durante la computación sistólica.
- Precisa, en el cálculo de la óptima o una solución cuasi-óptima.
- Suficientemente flexible, para introducir a los mejores operadores conocidos para resolver los problemas de optimización presentados en la literatura.

Con estos requerimientos en mente, un algoritmo parece particularmente adecuado. El componente seleccionado es el algoritmo μGA. El término "micro"(mu) se refiere a que este algoritmo trabaja con una pequeña población que opera con los mismos operadores genéticos como los GAs estándar. Como los tamaños de población más grandes requieren más tiempo de cómputo para encontrar la solución óptima, Krishnakumar propuso un micro Algoritmo Genético (μGA) [71], basado en el trabajo teórico por [52].

Dentro de HySyS, el μGA funciona realizando ciclos, cada uno aplica todos los operadores genéticos sobre una población y como resultado se obtiene la siguiente población. Esta población sera usada como soluciones bases para ser insertadas en la población del SNS. Utilizamos las soluciones del μGA para buscar una mejora general de la calidad de la población sistólica y para ayudar en la búsqueda.

El proceso μGA comienza creando de forma aleatoria una población (definida como P) formada por P_i soluciones con $i = \{1,2,\ldots,g\}$. Luego, el μGA elige a los padres, que son sometidos a operadores genéticos (cruce, mutación). Finalmente, se evalúa los hijos resultantes de los operadores. Estos pasos se repiten hasta que una población auxiliar (llamada $auxP$) se completa con las soluciones tentativas recién generadas. Este proceso podemos definirlo como un ciclo del μGA.

Con el fin de controlar mejor las capacidades de intensificación de μGA, un procedimiento de búsqueda local se ha incluido a nivel local mejorar las soluciones encontradas. En nuestro caso, hemos utilizado un Hill Climbing (HC) [106]. El HC es una búsqueda iterativa que funciona a partir de una solución arbitraria inicial a un problema.

Se busca cualquier movimiento que suponga una mejora respecto a la solución padre y el proceso se repite hasta que luego de un número determinado de pasos no se pueda mejorar la solución.

5.3.2. Descripción detallada del modelo HySyS

En la presente sección, se describe en detalle nuestra propuesta HySyS. El Algoritmo 5.3 muestra la distribución de los componentes algorítmicos utilizados y como es su funcionamiento a lo largo de la ejecución de HySyS. Básicamente, podemos observar tres pasos fundamentales:

- Establecer todas las variables de configuración y determinar el número de hilos de CPU requeridos y los de la GPU respectivamente, de acuerdo con las variables definidas anteriormente. Además, creamos y evaluamos la población utilizada en cada componente de la CPU y la GPU, respectivamente, de acuerdo con el esquema algorítmico determinado.
- Ejecutar cada proceso teniendo en cuenta la plataforma utilizada. Para el caso de la CPU, un ciclo μGA se ejecuta y para la GPU se corre un enfoque SNS^{exp}.
- Una vez completado un ciclo μGA, dejamos de todos los procesos y reemplazamos algunas soluciones tentativas de las poblaciones M con las existentes en P. Por último, HySyS reinicia todo el proceso de nuevo ejecutando el ciclo μGA y continúa el proceso SNS^{exp}

Al comienzo del proceso evolutivo, $HySyS$ comienza con la inicialización en los lados anfitrión (CPU) y el dispositivo (GPU). Durante la fase de inicialización, primero establecemos parámetros de configuración para ambos componentes, μGA y SNS^{exp} (líneas 1-3). A continuación, lanzamos el proceso de μGA para la creación y la evaluación de P (líneas 4-6). También, se pone en marcha el *kernel* de la creación del lado del dispositivo y la evaluación de la población en el de malla M (líneas 7-9).

En el lado del anfitrión el proceso de evolución comienza con una población P compuesta de cinco soluciones, donde $P = \{P_1, \ldots, p_5\}$. En cada ciclo las soluciones son sometidas a procedimientos específicos para generar nuevas soluciones. En primer lugar, p_1 y p_2 se someten a un HC. Cada solución generada se asigna a $auxP_1$ y $auxP_2$. Seguidamente, p_3 y p_4 se cruzan y se generan 2 hijos. La mejor solución de los dos se almacena en $auxP_3$ y el otro se somete a mutación y luego se almacena en $auxP_4$. El objetivo de todas estas operaciones es generar material genético muy diverso (diversificación). Por último, en $auxP_5$ la mejor solución de P se almacena para preservarla (línea 17).

Mientras que μGA se está ejecutando en la CPU, HySyS procede a evolucionar las soluciones localizadas en M mediante el modelo de búsqueda SNS^{exp} en la GPU (línea 21 del Algoritmo 5.3). El algoritmo SNS^{exp} se ejecuta sin interrupción hasta que haya finalizado un ciclo del μGA. En este punto, HySyS detiene la ejecución de ambos componentes y reemplaza un número de u soluciones en M, con las soluciones que fueron almacenadas en P al final del ciclo. La sustitución se lleva a cabo si la copia (solución al azar seleccionado de P) es mejor que la solución actual al azar seleccionado de M, caso contrario la solución actual permanece intacta. El valor para el parámetro u suele ser mayor que el tamaño de P, es decir, $u > 5$. El valor de u debe ser cuidadosamente elegido porque un valor alto (es decir, $u = m \times n$) podría generar un considerable sesgo en la población del SNS^{exp} . Una vez realizado el reemplazo, HySyS relanza tanto el ciclo μGA como el SNS^{exp}.

Algoritmo 5.3 Pseudocodigo de un HySyS canónico.

Entrada: $m, n, s, g, M \neq \emptyset, P \neq \emptyset$

1: $global\_flag \leftarrow false$
2: Configuración kernel creación y evaluación:($grid_1$, $threads_1$)
3: Configuración kernel búsqueda sistólica:($grid_2$, $threads_2$)
4: **Si** ($cpu\_thread\_id = 1$) **Entonces**
5: CrearPoblación(P, g, s)
6: EvaluarPoblación(P, g, s)
7: **Si no Si** ($cpu\_thread\_id = 2$) **Entonces**
8: **invocar** kernel_creation_Y_evaluation($M, m, n, s, grid_1, threads_1$)
9: **Fin Si**
10: {Inicio Proceso de Evolución}
11: **Mientras** Hasta no completar el criterio de parada **hacer**
12: {Sector que controla el componente CPU}
13: **Si** ($cpu\_thread\_id = 1$) **Entonces**
14: **Mientras** (no se complete $auxP$) **hacer**
15: ejecutar ciclo μGA
16: **Fin Mientras**
17: Guardar el mejor de P en $auxP$
18: $global\_flag \leftarrow true$ {Sector que controla el componente GPU}
19: **Si no Si** ($cpu\_thread\_id = 2$) **Entonces**
20: **Mientras** ($global\_flag \neq true$) **hacer**
21: **invocar** kernel_busqueda_sistolica($M, m, n, s, grid_2, threads_2$)
22: **Fin Mientras**
23: **Fin Si**
24: {Movimiento de soluciones entre componente}
25: **Si** ($cpu\_thread\_id = 3$ and $global\_flag = true$) **Entonces**
26: Reemplazar u soluciones en M utilizando P como base
27: $global\_flag \leftarrow false$
28: **Fin Si**
29: **Fin Mientras**
30: **obtener El Mejor de** M

Por un lado, una clara consecuencia del movimiento de soluciones entre la CPU y la GPU es la saturación del bus de comunicación entre ambas arquitecturas. Con nuestra aproximación buscamos evitar esto mediante la utilización del parámetro u el cual además de definir el número de cambios a realizar, permite controlar el flujo de información entre la CPU y la GPU. Por otro lado, tiene la ventaja de eliminar la necesidad de hacer un algoritmo eficiente en la GPU con las principales características de un μGA (eso puede implicar momentos de divergencia y serialización en una gran cantidad de hilos). Se ha buscado crear una carga de trabajo equilibrada entre las dos arquitecturas (CPU y GPU).

Nuestro enfoque tiene como objetivo mejorar la calidad de las soluciones sin aumentar *el costo en términos de tiempo de ejecución, proporcionar mejores soluciones y un algoritmo robusto* utilizando métodos de colaboración que pueden hacer pleno uso de modernas arquitecturas heterogéneas.

5.3.3. Detalles de implementación

En esta sección, nos centramos en los detalles de implementación de bajo nivel. Estas características en realidad tienen un profundo impacto en el rendimiento de HySyS. Tres temas principales se discuten: la generación de números aleatorios, el despliegue de la memoria, y el control del hilo sobre los componentes HySyS.

El rendimiento de cualquier algoritmo estocástico, tal como un algoritmo evolutivo, altamente depende de la calidad de su generación de números aleatorios. Durante la ejecución, SNS^{exp} genera soluciones iniciales de manera aleatoria. Con el fin de evitar la transferencia de datos entre la CPU y la GPU, SNS ^{exp} pasa una semilla global (*seed*) como un parámetro al *kernel* sólo una vez al principio de la ejecución, la cual sera utilizada por cada hilo local de GPU. Cada uno de ellos generara semillas locales y posteriormente podrán generar números aleatorios individualmente. El generador utilizado es una versión del Mersenne Twister que fue implementada dentro de las herramientas de desarrollo de CUDA ha sido utilizado en otra aproximaciones de optimización tales como [39, 104]. Una discusión mucho más amplia se puede ver en [27].

Con el fin de evitar la continua comunicación entre la CPU y la GPU, HySyS usa la memoria global del dispositivo GPU para almacenar toda la población del algoritmo SNS^{exp}. El tamaño de los problemas abordados nos inhabilita a trabajar o implementar algún modelo con memoria compartida.

Con respecto a la gestión de hilos de HySyS, utilizamos Open-MP (del inglés, Open Multi-Processing) [97]. Open-MP se ha establecido como una herramienta importante para modelos de computadoras paralelas de memoria compartida. Open-MP ofrece varias ventajas como un paradigma de programación para GPGPUs.

- Open-MP es eficiente y presenta una escala de aprendizaje simple al trabajar con paralelismo a nivel de bucles.
- Presenta un modelo de paralelización incremental de aplicaciones que permite integrar a la CPU y a la GPU como modelo de trabajo conjunto.

De esta forma, Open-MP nos ha permitido tener una manejo eficiente de los recursos computacionales mediante el uso de un hilo de la CPU para manejar el cálculo del SNS^{exp} y sus recursos en la GPU,y otro hilo de CPU para gestionar los operadores del μGA.

5.4. Conclusiones finales

Resumiendo, las principales consideraciones hechas en este capítulo son las siguientes:

- Se presenta un enfoque en el campo de los cGAs sobre GPUs. El primera de ellos es un potente algoritmo basado en el modelo canónico del cGA, llamado cGA-GPU. Su estructura de búsqueda y distribución de la población lo hace altamente adaptable al modelo de trabajo de una GPU. Se ha buscado explotar el inherente paralelismo de una GPU utilizando un mapeo directo entre la estructura poblacional de un cGA y la distribución de hilos.
- El segundo enfoque denominado cGA-multiGPU divide la población en tantas sub-poblaciones como dispositivos GPU se tengan activados. En cada GPU se ejecuta un cGA independiente con el modelo de ejecución descrito en el párrafo anterior. Al principio de cada iteración se envía la frontera de cada sub-población para respetar el formato de búsqueda del cGA y al final de cada evaluación, cada GPU envía a las otras vecinas las respectivas soluciones fronteras provocando una necesaria actualización síncrona. Se busca dividir la población, sobre todo para aquellas de gran dimensión, favoreciendo la aceleración algorítmica a partir del uso de múltiples GPUs, tratando de distribuir una mayor carga de procesamiento.
- Se presenta un modelo denominado SNS el cual esta basado en el concepto de Computación Sistólica. El modelo presentado esta diseñado para trabajar y aprovechar el modelo de trabajo de la GPU. El SNS exhibe un comportamiento de búsqueda simple realizando sencillas perturbaciones sobre las soluciones que permiten explorar de manera eficaz el espacio de búsqueda.
- A partir del modelo SNS canónico hemos presentado distintas derivaciones, las cuales buscan mejorar o ampliar el abanico de opciones que presenta en un principio el SNS. Cada una de estas variaciones puede ser utilizadas por separado o en conjunto de acuerdo a las características de los problemas, lo cual también hace que el modelo presentado sea altamente flexible y escalable según se presenten las necesidades.
- Se desarrolla una estrategia de paralelismo híbrido llamada HySyS. Hemos utilizado dos algoritmos, un μGA para la CPU y el SNS^{exp} para la GPU. Nuestro objetivo ha sido proporcionar mejoras que afecten a la eficiencia de búsqueda del algoritmo SNS^{exp} a través de la colaboración de soluciones con el μGA. De la misma forma, hemos buscado utilizar recursos que hasta el momento no eran aprovechados para formar parte del proceso de ejecución.

Capítulo 6

Análisis experimental del cGA en GPU

En este capítulo se evalúan y analizan los resultados a partir de la ejecución de nuestras dos aproximaciones celulares sobre GPU. En primer lugar, se utiliza una configuración de grano fino enteramente implementada sobre una GPU llamada cGA-GPU. Para el segundo caso, con el fin de mejorar la capacidad de búsqueda y la eficiencia, se ha extendido la aproximación a una arquitectura con múltiples GPUs denominada cGA-multiGPU.

Para la evaluación de las técnicas anteriormente mencionadas se han estudiado diversos problemas del dominio discreto y continuo (Capítulo 4). Asimismo, se evalúa el comportamiento de nuestras aproximaciones contra un modelo secuencial (cGA-GPU).

En la Sección 6.1 se presentan las instancias seleccionadas y la parametrización empleada en los algoritmos evaluados. Seguidamente,en la Sección 6.2 se muestran los resultados y el análisis de los mismos para las dos aproximaciones implementadas en GPUs. Finalmente, en la Sección 6.3 se incluyen las conclusiones de este capítulo.

6.1. Diseño experimental

El diseño experimental se ha definido para evaluar el rendimiento de las paralelizaciones propuestas, pero si descartar su eficiencia numérica. A continuación se describen las instancias y parámetros utilizados.

Instancias seleccionadas

Se ha seleccionado un conjunto de problemas representativos con diversas características, tanto del dominio discreto como continuo, que podemos encontrar en diferentes contextos de optimización. Los problemas combinatorios son la Minimización de Colville, ECC, y MMDP, y del dominio continuo, son Rastrigin, Rosenbrock y Griewank. La dimensión de los problemas continuos es 50. La elección de esta batería de problemas se justifica tanto por la dificultad que entrañan los problemas que la componen como por los dominios de aplicación a los que pertenecen (optimización combinatoria, optimización continua, telecomunicaciones, planificación, etc.). Estos son ingredientes importantes en cualquier trabajo que trate de evaluar aproximaciones algorítmicas con el objetivo de obtener resultados fiables, como se especifica en [136].

Aunque está fuera de nuestro objetivo realizar un estudio completo de los problemas seleccionados, en este capítulo se presentan y analizan los resultados obtenidos al resolver las instancias de los problemas con las variantes de cGA (cGA-CPU, cGA-GPU y cGA-multiGPU).

Tabla 6.1: Parámetros utilizados para evaluar el cGA-GPU.

Parámetro	**Valor**
Máximo número de generaciones	500
Tamaño de Población	8×8, 16×16, 32×32 64×64, 128×128, 192×192, 256×256, 512×512
Selección de vecindario	Norte-Sur-Este-Oeste
Selección de padres	La solución actual + Torneo Binario/Ruleta
Prob. de recombinación	DPX, p_c = 1.0
Prob. de mutación	bit-flip, p_m=0.05
Reemplazo	Reemplazar si la nueva solución es mejor

Parametrización

En esta apartado se proporciona una descripción de los parámetros empleados basándonos en estudios previos de la literatura [3]. La Tabla 6.1 describe la parametrización utilizada para el conjunto de algoritmos de prueba y nuestra aproximación. Los tamaños de población son mallas cuadradas de 8×8, 16×16, 32×32 64×64, 128×128, 192×192, 256×256, 512×512 soluciones. El operador de selección para el primer análisis con el cGA-GPU es un torneo binario y para el caso del cGA-multiGPU es una selección por ruleta. Todos los algoritmos utilizan el operador de recombinación de dos puntos –DPX– (con probabilidad 1.0) y el operador de mutación es un bit-flip (con probabilidad 0.05). La política de reemplazo es elitista, es decir, una nueva solución generada (descendiente) reemplaza a la actual sólo si tiene un valor de *fitness* mayor (asumiendo que queremos maximizar). La condición de terminación es computar 500 generaciones.

Los experimentos se han desarrollado utilizando el entorno de Configuración 1 (Capítulo 4). Debido a la naturaleza estocástica de los algoritmos, realizamos 30 ejecuciones independientes para cada prueba con el fin de producir suficientes datos experimentales y poder obtener resultados estadísticamente significativos siguiendo el modelo de análisis propuesto en la Capítulo 4.3.

6.2. Presentación y análisis de los resultados

Se presenta a continuación los resultados obtenidos para los dos estudios. Primeramente evaluamos un cGA-GPU frente el cGA-CPU. En primer lugar estamos interesados en comprobar si un cGA puede resolver diferentes problemas sobre la GPU. Posteriormente, ampliamos este análisis para una arquitectura multi-GPU. Al comparar dos algoritmos (secuencial vs. paralelo) podemos obtener una idea clara de la eficiencia numérica del proceso de búsqueda atendiendo al esfuerzo computacional necesario para alcanzar una solución de igual adecuación en cada algoritmo. Asimismo, se analizan los tiempos de ejecución para evaluar la competitividad de nuestra implementación con respecto a la versión secuencial. De la misma forma, es evidente que la ganancia en velocidad únicamente tiene sentido cuando hablamos de dos algoritmos evolutivos de idéntico comportamiento.

La Tabla 6.3 incluye los tiempos de ejecución promedio para 30 ejecuciones. La primera columna indica el tamaño de población y luego las siguientes columnas indican los tiempos de ejecución (en segundos) del cGA-CPU y del cGA-GPU, respectivamente.

6.2.0.1. Aproximación cGA para una sola GPU

La Tabla 6.2 informa los valores de *fitness* promedio para 30 ejecuciones del cGA-GPU en cada problema. La columna uno indica el tamaño de población y las siguientes columnas muestran los valores de *fitness* y la desviación estándar promedio.

Tabla 6.2: Valores de *fitness* promedio del cGA-GPU para 30 ejecuciones.

Tamaño de Población	Colville	ECC	MMDP	Rastrigin	Rosenbrock	Grienwak
8×8	0.001±5.176e-5	0.066±2.065e-3	39.800±6.554e-6	1.223e-6±2.003e-6	2.336e-6±2.018e-6	5.366e-6±9.817e-5
16×16	0.013±9.625e-6	0.067±0.008e-3	39.900±3.258e-6	2.771e-6±3.159e-6	1.989e-6±1.255e-6	2.256e-6±7.12e-5
32×32	0.133±3.176e-6	0.066±1.065e-6	39.896±9.146e-6	4.637e-5±5.512e-6	4.637e-5±3.075e-6	8.258e-5±7.142e-5
64×64	0.111±3.684e-6	0.066±2.667e-6	39.900±9.145e-6	2.978e-5±1.136e-6	2.978e-5±4.919e-6	2.687e-5±5.171e-5
128×128	0.011±0.586e-6	0.066±2.685e-6	39.000±3.644e-6	8.658e-6±2.055e-6	5.366e-5±2.881e-5	8.281e-6±3.881e-5
192×192	0.110±2.009e-5	0.066±1.258e-6	37.333±0.582e-6	8.554e-6±0.2515-6	9.363e-5±2.669e-5	3.582e-5±4.761e-5
256×256	0.100±1.033e-6	0.067±0.361e-6	38.333±1.365e-6	1.218e-5±7.822e-6	1.639e-5±5.761e-5	2.350e-5±3.691e-5
512×512	0.010±0.310e-6	0.067±0.003e-6	39.966±2.713e-6	1.749e-6±4.350e-6	3.311e-5±1.771e-5	1.356e-6±8.323e-5

Para los problemas discretos, se puede observar que los resultados obtenidos alcanzan el óptimo o quedan muy cerca de alcanzarlo utilizando una configuración general de ejecución. En

el caso de los problemas continuos sucede algo similar: los valores alcanzados son los que se esperaban en comparación con los obtenidos en la literatura [3]. En ambos dominios el tamaño de malla parece influenciar minimamente la diversidad media de la población, tanto si es pequeño o grande. Esto es posible que se deba a la alta velocidad con que se encuentra y propaga las soluciones de calidad, y en el caso de las grandes poblaciones se debe a la gran diversidad existente.

Tabla 6.3: Tiempos (s) de ejecución promedio en 30 ejecuciones para el cGA-CPU y cGA-GPU.

Tamaño de	Colville		ECC		MMDP		Rastrigin		Rosenbrock		Grienwak	
Población	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
8×8	0.01	0.02	0.01	0.01	0.02	0.02	0.02	0.02	0.03	0.02	0.03	0.03
16×16	0.05	0.09	0.02	0.03	0.05	0.06	0.04	0.09	0.03	0.07	0.05	0.07
32×32	0.10	0.18	0.11	0.17	0.11	0.14	0.12	0.22	0.11	0.19	0.12	0.15
64×64	1.19	0.21	1.25	0.23	1.21	0.21	1.17	0.21	1.20	0.21	0.21	0.20
128×128	2.81	0.22	3.19	0.25	3.09	0.22	2.97	0.22	2.86	0.22	2.97	0.23
192×192	3.25	0.24	4.03	0.26	3.95	0.26	3.73	0.24	3.69	0.25	3.89	0.25
256×256	4.16	0.25	4.63	0.27	4.09	0.28	4.31	0.27	4.66	0.26	4.53	0.26
512×512	7.46	0.32	7.89	0.35	7.66	0.33	7.21	0.34	7.12	0.33	7.35	0.35

Los resultados de la Tabla 6.3 indican tiempos de ejecución menores para el cGA-GPU con respecto al cGA-CPU a partir de la población 64×64. En el caso de las poblaciones 8×8, 16×16, 32×32 los tiempos son similares entre las dos aproximaciones e incluso para algunos casos la CPU supera a la GPU. A medida que los tamaños de poblaciones van aumentando se ve un incremento de los tiempos de ejecución en GPU mientras que para la CPU aumentan de forma más rápida.

Tabla 6.4: *Speedup* entre cGA-CPU vs cGA-GPU.

Tamaño de Población	Colville	ECC	MMDP	Rastrigin	Rosenbrock	Grienwak
8×8	0.500	1.000	1.000	1.000	1.500	1.000
16×16	0.556	0.067	0.833	0.444	0.428	0.714
32×32	0.556	0.660	0.784	0.494	0.539	0.826
64×64	5.667	5.450	5.645	5.417	5.688	5.783
128×128	12.773	12.760	14.045	13.500	13.000	12.913
192×192	13.542	15.500	15.192	15.541	14.760	15.560
256×256	16.640	16.830	14.485	15.463	17.830	16.964
512×512	23.312	22.419	22.789	20.810	20.982	20.421

Numéricamente, el modelo cGA-GPU permite resolver los problemas sin ninguna perdida de eficacia con respecto al modelo secuencial. En este sentido, la utilización de una cantidad creciente de hilos de GPU a medida que los tamaños de población aumentan no afecta a la resolución numérica del cGA directamente. Estos resultados indican que la ejecución del cGA sobre una GPU presenta un comportamiento estable y robusto.

La ganancia de tiempos (*speedup*) se incluyen en la Tabla 6.4. Se puede observar que, en las poblaciones de tamaño 16×16 y 32×32, los valores de *speedup* se encuentran en su mayoría igual

o por debajo del valor 1.0, lo que nos indica que el cGA en CPU tiene tiempos iguales o menores de ejecución que el cGA-GPU. Para poblaciones iguales o mayores que 64×64 el *speedup* sube a valores entre 5× y 24×.

6.2.0.2. Aproximación cGA para una múltiples GPUs

Habiendo evaluado el comportamiento del cGA en una sola GPU, ahora pasaremos al siguiente estudio: verificar si la ejecución de un cGA sobre múltiples GPUs y la correspondiente distribución de la población es la más adecuada para resolver diversos problemas. Se comparan las versiones cGA-CPU, cGA-GPU, cGA-multiGPU. Para este experimento la única variación que hacemos es que nuestro operador de selección es la ruleta.

Las Tablas 6.5, 6.6 y 6.7 detallan los valores de *fitness* promedio y su desviación estándar para 30 ejecuciones de los algoritmos cGA-CPU, cGA-GPU y cGA-multiGPU, respectivamente.

Tabla 6.5: Valores de *fitness* promedio del cGA-CPU para 30 ejecuciones.

Tamaño de Población	Colville	ECC	MMDP	Rastrigin	Rosenbrock	Grienwak
8×8	0.001±1.792e-5	0.066±3.918e-3	39.933±0.789e-6	2.902e-6±8.291e-6	2.938e-6±2.018e-6	3.988e-6±1.812e-5
16×16	0.011±8.661e-6	0.067±0.001e-3	39.000±2.312e-6	3.967e-6±4.677e-6	1.989e-6±1.255e-6	1.666e-6±4.788e-5
32×32	0.111±3.886e-6	0.066±1.918e-6	39.333±5.392e-6	3.718e-5±1.817e-6	4.637e-5±4.903e-6	1.922e-5±4.918e-5
64×64	0.101±7.696e-6	0.066±3.871e-6	39.333±9.929e-6	3.812e-5±7.133e-6	2.988e-5±2.008e-6	2.901e-5±9.721e-5
128×128	0.010±4.661e-6	0.067±2.986e-6	38.000±7.817e-6	3.787e-6±2.113e-6	6.993e-5±1.082e-5	7.966e-6±3.336e-5
192×192	0.010±4.663e-5	0.067±5,817e-6	39.000±4.761e-6	4.881e-6±3.091e-6	9.363e-5±3.901e-5	9.817e-5±6.643e-5
256×256	0.011±1.996e-6	0.066±7.669e-6	38.000±5.771e-6	0.443e-5±4.810e-6	1.639e-5±4.296e-5	6.881e-5±0.556e-5
512×512	0.001±1.901e-6	0.066±0.099e-6	39.000±2.733e-6	4.991e-6±0.821e-6	3.311e-5±8.109e-5	3.767e-6±3.542e-5

Las Tablas 6.5, 6.6 y 6.7 indican que los valores de *fitness* promedio alcanzan los valores óptimos. Al analizar los resultados de cada modelo paralelo se puede observar que el comportamiento de los algoritmos cGA-GPU y cGA-multiGPU es similar a la versión secuencial ya que obtienen valores óptimos en el mismo número de pasos y pueden muestrear un número similar de soluciones en el espacio de búsqueda del problema.

Tabla 6.6: Valores de *fitness* promedio del cGA-GPU para 30 ejecuciones.

Tamaño de Población	Colville	ECC	MMDP	Rastrigin	Rosenbrock	Grienwak
8×8	0.001±1.792e-5	0.066±2.821e-3	39.333±0.789e-6	1.029e-6±2.003e-6	3.233e-6±9.910e-6	3.991e-6±7.163e-5
16×16	0.010±9.213e-6	0.067±0.001e-3	39.666±4.045e-6	2.771e-6±3.159e-6	4.661e-6±3.331e-6	1.229e-6±6.333e-5
32×32	0.130±5.541e-6	0.067±0.001e-6	39.000±3.509e-6	4.637e-5±5.122e-6	6.116e-5±3.991e-6	3.977e-5±1.230e-5
64×64	0.110±3.412e-6	0.066±2.833e-6	39.900±3.312e-6	2.978e-5±4.187e-6	3.912e-5±2.008e-6	2.981e-5±1.233e-5
128×128	0.010±2.612e-6	0.067±6.919e-6	38.000±3.881e-6	4.871e-6±2.055e-6	3.661e-5±2.918e-5	2.891e-6±1.011e-5
192×192	0.010±2.312e-5	0.067±4.171e-6	39.000±2.982e-6	8.554e-6±6.562e-6	3.661e-5±3.778e-5	3.881e-5±8.011e-5
256×256	0.010±2.883e-6	0.066±3.886e-6	38.000±0.539e-6	1.218e-5±7.822e-6	8.615e-5±5.981e-5	4.668e-5±4.096e-5
512×512	0.001±0.310e-6	0.066±5.813e-6	39.000±1.928e-6	1.749e-6±4.303e-6	4.981e-5±4.871e-5	2.933e-6±4.100e-5

La utilización del operador ruleta ha demostrado ser claramente una mejor elección que el operador Torneo Binario utilizado en el estudio de la Sección 6.2.0.1. Por otro lado, el cGA-multiGPU consigue obtener valores óptimos para todos los problemas, lo cual indica que la distribución de la población en múltiples GPUs es factible desde el punto de vista numérico. Este comportamiento se mantiene a medida que las poblaciones van aumentando su tamaño.

Tabla 6.7: Valores de *fitness* promedio del cGA-multiPU para 30 ejecuciones.

Tamaño de Población	Colville	ECC	MMDP	Rastrigin	Rosenbrock	Grienwak
8×8	0.001±1.792e-5	0.066±2.065e-3	39.933±1.192e-6	1.223e-6±2.003e-6	2.336e-6±2.018e-6	5.366e-6±7.163e-5
16×16	0.010±9.213e-6	0.067±0.001e-3	39.000±2.466e-6	2.771e-6±3.159e-6	1.989e-6±1.255e-6	1.333e-6±6.333e-5
32×32	0.130±5.541e-6	0.066±1.065e-6	39.000±1.881e-6	4.637e-5±5.122e-6	4.637e-5±4.903e-6	8.258e-5±1.230e-5
64×64	0.110±3.412e-6	0.066±2.667e-6	39.900±1.881e-6	2.978e-5±4.187e-6	2.978e-5±2.008e-6	2.687e-5±9.331e-5
128×128	0.010±2.612e-6	0.066±2.685e-6	39.000±1.864e-6	8.658e-6±2.055e-6	8.233e-5±3.109e-5	8.281e-6±1.011e-5
192×192	0.010±2.312e-5	0.066±1.258e-6	37.333±1.881e-6	8.554e-6±6.562e-6	9.363e-5±3.901e-5	3.582e-5±8.011e-5
256×256	0.010±2.883e-6	0.066±0.361e-6	37.866±5.991e-6	1.218e-5±7.822e-6	1.639e-5±5.306e-5	1.981e-5±4.096e-5
512×512	0.001±0.310e-6	0.066±0.003e-6	38.966±4.316e-6	1.749e-6±4.303e-6	3.311e-5±8.109e-5	1.356e-6±4.100e-5

Continuamos analizando la relación entre el tiempo de ejecución del cGA-multiGPU con respecto a las versiones cGA-CPU y cGA-GPU. Los tiempos para el cGA secuencial se presentan en la Tabla 6.8, mientras que los de los algoritmos cGA-GPU y cGA-multiGPU se incluyen en la Tabla 6.9 y Tabla 6.10, respectivamente.

Tabla 6.8: Tiempos (s) de ejecución promedio en 30 ejecuciones para el cGA-CPU.

Tamaño de Población	Colville	ECC	MMDP	Rastrigin	Rosenbrock	Grienwak
8×8	0.168	0.223	0.242	0.254	0.266	0.260
16×16	2.295	2.258	2.923	2.824	3.010	2.703
32×32	8.903	10.468	10.956	10.183	10.860	10.633
64×64	44.770	47.484	46.961	49.362	47.613	46.619
128×128	147.135	158.922	161.984	153.631	158.105	165.174
192×192	286.021	235.831	350.126	309.363	361.069	301.121
256×256	485.754	552.221	507.957	525.413	501.290	505.603
512×512	1120.386	1328.011	1372.319	1450.415	1385.043	1402.827

El análisis de los resultados de las Tablas 6.8, 6.9 y 6.10 permite realizar varias observaciones. En primer lugar el modelo cGA-GPU invierte un menor tiempo (más rápido) en encontrar el óptimo con respecto al cGA-multiGPU, pero el beneficio no es muy significativo para cualquier tamaño de población. Esto es debido a que el tiempo de ejecución en subpoblaciones no compensa la sobrecarga en las comunicaciones al mover las soluciones fronteras entre GPUs. Aun así, la tendencia en ambas aproximaciones es una clara reducción del tiempo de ejecución, estas diferencias mínimas indican que es posible utilizar el modelo cGA-multiGPU para evaluar problemas que requieran un mayor esfuerzo computacional. Finalmente, se presentan los valores de *speedup*

Tabla 6.9: Tiempos (s) de ejecución promedio en 30 ejecuciones para el cGA-GPU

Tamaño de Población	Colville	ECC	MMDP	Rastrigin	Rosenbrock	Grienwak
8×8	0.012	0.014	0.018	0.016	0.020	0.021
16×16	0.056	0.059	0.063	0.057	0.053	0.048
32×32	0.111	0.126	0.122	0.119	0.112	0.116
64×64	0.251	0.241	0.268	0.251	0.247	0.221
128×128	0.508	0.612	0.630	0.597	0.582	0.609
192×192	0.750	0.847	0.861	0.766	0.785	0.765
256×256	0.960	1.146	1.040	1.078	1.106	1.024
512×512	1.468	1.499	1.589	1.673	1.591	1.563

Tabla 6.10: Tiempos (s) de ejecución promedio en 30 ejecuciones para el cGA-multiGPU.

Tamaño de Población	Colville	ECC	MMDP	Rastrigin	Rosenbrock	Grienwak
8×8	0.021	0.023	0.021	0.021	0.028	0.026
16×16	0.078	0.081	0.090	0.087	0.089	0.082
32×32	0.140	0.153	0.157	0.148	0.153	0.151
64×64	0.367	0.382	0.379	0.393	0.389	0.379
128×128	0.667	0.721	0.737	0.692	0.722	0.749
192×192	0.881	0.858	0.952	1.025	0.977	0.896
256×256	1.181	1.227	1.227	1.279	1.214	1.226
512×512	1.621	1.721	1.781	1.891	1.795	1.821

Tabla 6.11: *Speedup* entre cGA-CPU y cGA-multiGPU.

Tamaño de Población	Colville	ECC	MMDP	Rastrigin	Rosenbrock	Grienwak
8×8	8.992	9.730	10.084	10.591	9.509	10.020
16×16	29.423	31.858	32.485	32.463	33.831	32.964
32×32	63.593	68.419	69.789	68.810	70.982	70.421
64×64	121.991	124.306	123.909	125.604	122.400	123.007
128×128	220.593	220.419	219.789	218.810	218.982	220.527
192×192	324,654	274.861	367.779	352.252	369.569	378.292
256×256	411.308	412.413	413.983	410.800	412.925	412.401
512×512	752.860	771.651	767.166	767.010	771.612	770.910

Tabla 6.12: *Speedup* entre cGA-GPU y cGA-multiGPU.

Tamaño de Población	Colville	ECC	MMDP	Rastrigin	Rosenbrock	Grienwak
8×8	0.570	0.666	0.850	0.760	0.714	0.807
16×16	0.717	0.737	0.700	0.655	0.595	0.585
32×32	0.718	0.728	0.777	0.804	0.732	0.768
64×64	0.683	0.818	0.707	0.638	0.634	0.583
128×128	0.761	0.630	0.854	0.862	0.806	0.813
192×192	0.851	0.987	0.904	0.747	0.803	0.853
256×256	0.810	0.855	0.840	1.078	0.911	0.835
512×512	0.871	0.871	0.884	1.673	0.886	0.858

obtenidos entre la aproximación cGA-CPU y cGA-multiGPU (Tabla 6.11) y entre el cGA-GPU y el cGA-multiGPU (Tabla 6.12). Las dos tablas incluyen los tamaños de población usados en la primera columna, mientras que el resto muestran el *speedup* para cada problema trabajado.

En general, el modelo cGA-multiGPU nos permite reducir el tiempo de búsqueda con respecto al cGA-CPU y obtiene un *speedup* que alcanza valores de hasta 771× (Tabla 6.11). En el caso del *speedup* entre los modelos cGA-GPU y cGA-multiGPU mostrado en la Tabla 6.12 se puede observar que para todos los problemas el modelo cGA-multiGPU no puede superar los tiempos de ejecución del cGA-GPU (valores menores a 1.0). Igualmente, podemos concluir que al no existir una diferencia considerable de tiempos entre ambas aproximaciones, es factible la utilización de un modelo multiGPU para acelerar el proceso de búsqueda en problemas computacionalmente más costosos.

6.3. Conclusiones

Como principal contribución de este capítulo, se ha evaluado un nuevo cGA ejecutándose sobre una y múltiples GPUs. Las dos aproximaciones evaluadas han logrado ser altamente competitivas con respecto al estado del arte de los cGAs. Se ha utilizado para evaluar algoritmos diversos problemas de los ámbitos continuo y discreto, y se han conseguido alcanzar soluciones óptimas en la mayoría de las instancias estudiadas. El uso de los modelos celulares en GPU nos ha llevado a un excelente porcentaje de soluciones encontradas y tiempos de ejecución eficientes. La versión cGA-GPU necesita menos tiempo que el cGA-multiGPU ya que para esta aproximación existe una sobrecarga inducida por las continuas comunicaciones entre las diversas GPUs.

Los resultados demuestran que el cGA-multiGPU obtiene tiempos de ejecución un poco mayores que el cGA-GPU en todas las instancias. No obstante, ambos demuestran la misma capacidad de resolución numérica en tiempos mucho menores (mejores) que la versión secuencial del cGA.

El modelo de cGA ha permitido encontrar una correspondencia entre la estructura de hilos de la GPU y la malla de soluciones con la consecuente construcción de vecindarios, no obstante el manejo de estas estructuras no es trivial dada la complejidad de memorias y el uso que se les puede dar. De este capítulo podemos concluir que una de las cuestiones cruciales es la minimización de la transferencia de datos entre la CPU y la GPU.

Finalmente, podemos decir que la exploración de poblaciones o vecindarios de un gran tamaño pueden ser factibles. En este sentido, la computación en GPU permite no sólo acelerar el proceso de búsqueda, sino también poder explotar el paralelismo para mejorar la calidad de las soluciones obtenidas.

Capítulo 7

Análisis experimental del SNS

La principal contribución de este capítulo es un estudio comparativo del algoritmo SNS, y sus diferentes versiones mejoradas, sobre los problemas SSP y MKP. El estudio se completa contrastando los resultados de estas aproximaciones frente a otros algoritmos usados como base comparativa. Como una segunda contribución, extendemos el estudio con un análisis de eficiencia para los algoritmos canónicos (SNS y SNS^{exp}) incluyendo una comparación de sus comportamientos y escalabilidad. Este estudio constituye una buena aproximación empírica para estudiar el comportamiento de un algoritmo en GPU.

El capítulo se estructura de la siguiente manera. En la Sección 7.1 se presenta una breve introducción al diseño experimental utilizado, y se describen los problemas y algoritmos empleados como base comparativa. En la Sección 7.2 se presentan los resultados así como un análisis de los mismos. Finalmente, en la Sección 7.3 se incluyen las conclusiones finales de este capítulo.

7.1. Diseño experimental

Esta sección presenta una descripción de los experimentos realizados en este trabajo. En la Sección 7.1.1 se detallan las características de las instancias seleccionadas. En la Sección 7.1.2 se describen los algoritmos utilizados como base comparativa en lo diversos estudios.

7.1.1. Instancias seleccionadas

Para el problema SSP se han seleccionado tres instancias denominadas SSP_1, SSP_2 y SSP_3. Todas estas instancias tienen un valor $m = 256$. La variable w_i toma valores de 1 a 10000 para SSP_1, de 1 a 100000 para SSP_2 y de 1 a 1000000 para SSP_3.

Las instancias MKP se han dividido en tres grupos buscando evaluar la eficiencia y escalabilidad numérica de los algoritmos. Los grupos se denominan instancias *pequeñas*, *medias* y *grandes*. El primer y segundo grupo corresponden al conocido conjunto de instancias MKP propuestas en [23]. Para las instancias pequeñas hemos seleccionado 18 instancias representativas que varían con $n = \{6, ..., 105\}$ elementos y el número de mochilas $m = \{2, ..., 30\}$. El segundo grupo de instancias (*instancias medianas*) está compuesta por aquellas de mayor tamaño de la librería *OR-library*. A partir de los archivos *mknapcb5.txt*, *mknapcb6.txt*, *mknapcb8.txt* y *mknapcb9.txt* se han utilizado las primeras cinco instancias de cada archivo que contienen la configuración $n = \{250, 500\}$ y $m = \{10, 30\}$. Las instancias grandes han sido seleccionadas de entre las definidas por Glover y Kochenberger en [47]. Particularmente, las instancias GK08 $\{n = 500, m = 50\}$, GK09 $\{n = 1500, m = 25\}$, GK10 $\{n = 1500, m = 50\}$ y GK11 $\{n = 3500, m = 100\}$.

7.1.2. Algoritmos de base comparativa

Los algoritmos empleados para la comparación se describen a continuación:

- ***Búsqueda Aleatoria en CPU***, (RS, por sus siglas en ingles). La RS pura consiste en generar de forma secuencial soluciones aleatorias e ir quedándose con la mejor [112]. La RS es aplicable a cualquier problema de optimización, debido a que no existen precondiciones para su uso. El algoritmo en concreto se ha denominado RS_{CPU} y en nuestro contexto de trabajo se ha aplicado como prueba de verificación para asegurar que SNS y sus diferentes versiones tienen un comportamiento no trivial (no aleatorio).
- ***Búsqueda Aleatoria en GPU***. La RS en GPU trabaja con un conjunto de soluciones de tamaño *n*, siendo n el número de columnas de SNS canónico (dimensión de la instancia). Básicamente, cada solución se gestiona en un hilo de la GPU. Cada uno de ellos aplica el mismo proceso que un RS_{CPU} de forma que, cuando se alcanza la condición de parada, se selecciona la mejor solución del conjunto. Este algoritmo se ha denominado RS_{GPU}.
- ***Algoritmo Genético en CPU***. Los operadores genéticos que utiliza el GA están definidos en la Sección 5.1.1. Para evaluar los algoritmos utilizamos una versión generacional del GA (llamada GA_{CPU}), que se detalla en el Algoritmo 7.1. Hemos seleccionado el GA como método de optimización parte de la base de comparación por su amplio uso en diversos contextos académicos y de ingeniería [31, 56, 131].

Algoritmo 7.1 Pseudocódigo del GA canónico en CPU.

```
1:  Para sol_i ∈ pop hacer
2:     sol_i ← generarSolución();
3:     sol_i ← evaluarSolución();
4:  Fin Para
5:  Mientras (not stop_criterion) hacer
6:     Para sol_i ∈ pop hacer
7:        padres ← seleccionarPadres(pop);
8:        hijo_i ← recombinación(padres);
9:        hijo_i ← mutación();
10:       hijo_i ← evaluarSolución();
11:       aux_pop_i ← reemplazo(hijo_i, i);
12:    Fin Para
13: Fin Mientras
14: findBetter(pop);
```

- ***Algoritmo Genético en GPU***, (GA_{GPU}). Este algoritmo tiene por objeto probar la eficiencia del algoritmo canónico GA en el mismo hardware utilizado para nuestro enfoque SNS. La principal diferencia con el GA ejecutándose en la CPU es que creamos tantos hilos como individuos existen en la población, y todo el proceso se lleva a cabo en paralelo. Cada uno de estos hilos es responsable de las operaciones genéticas (linea 7-11 del Algoritmo 7.1). Cuando todos los hilos han completado estas operaciones y se genera una población nueva, reemplaza la anterior (que se encuentra en la memoria global del dispositivo).

7.1.3. Parametrización

Los parámetros de las versiones CPU y GPU del GA se presentan la Tabla 7.1. La configuración de los operadores de GA es el resultado de un estudio previo de las configuraciones utilizadas en la literatura [8, 99, 141]. El operador de recombinación usado en estos algoritmos es la recombinación de dos puntos (DPX) con probabilidad 1.0. A partir de los dos hijos creados por el operador DPX sólo se elige uno: aquel con la mayor parte del mejor padre. Para mutar, se ha utilizado una mutación *bit-flip*.

Tabla 7.1: Configuración para el Algoritmo Genético.

Operador	Valor
Selección de padres	Selección de torneo binario ($k = 2$)
Recombinación	Dos puntos $Pc = 1.0$
Mutación	$Pm = 0.01$
Reemplazo	Remplazar el peor

El entorno de ejecución utilizado para los experimentos es la Configuración 1 detallada en el Capítulo 4.4. Realizamos 30 ejecuciones independientes para cada algoritmo. Los mejores valores obtenidos para cada problema se resaltan en **negrita**. El tamaño de población para cada algoritmo evaluado esta indicado en la Tabla 7.2

Tabla 7.2: Tamaño de población utilizadas para cada algoritmo.

	Algoritmo						
	RS_{GPU}	**RS_{CPU}**	**GA_{GPU}**	**GA_{CPU}**	**SNS**	**SNS^{exp}**	**SNS^{exp}-1**
Tamaño	256	1	256	256	256×256	256×9	256×8

7.2. Presentación y análisis de resultados

En esta sección analizamos los resultados de nuestra aproximación SNS y diferentes versiones frente a los algoritmos utilizados como base comparativa. El objetivo de estos experimentos es comprender el funcionamiento del SNS para proporcionar guías generales de optimización sobre GPU y clarificar algunas buenas prácticas.

En la Sección 7.2.1 se analiza el comportamiento del SNS y dos versiones iniciales (SNS^{exp} y SNS^{exp}-1) mediante tres estudios que usan diferentes parámetros de parada. Una vez evaluado el problema SSP y obtenido conclusiones iniciales, la Sección 7.2.2 analiza de forma profunda todas las variaciones del SNS con de instancias del MKP. Las mismas servirán para evaluar el rendimiento, robustez y escalabilidad de nuestro modelo de búsqueda sistólica sobre la GPU.

7.2.1. Análisis del problema SSP

Para este primer estudio se evalua el SNS canónico, SNS^{exp} y SNS^{exp}-1. El primer estudio utiliza como valor de parada un número fijo de evaluaciones (8421376 evaluaciones). Este valor de parada se ha fijado para asegurar que las soluciones de SNS puedan recorrer todas las celdas de una misma fila al menos una vez y comparar los métodos de búsqueda entre algoritmos. El segundo estudio evalúa el comportamiento de los algoritmos en un tiempo predefinido de 5 segundos. Finalmente, el tercer estudio selecciona como condición de parada que el algoritmo alcance el óptimo del problema considerado por nosotros (en este caso 0.0).

Estudio 1

La Tabla 7.3 presenta los resultados con respecto al primer estudio. En la primera columna se muestra el nombre del algoritmo usado. Las columnas que siguen (dos a la siete) muestran la relación de error en porcentaje (RE%), que indica la diferencia entre el mejor valor de *fitness* promedio para 30 ejecuciones y el óptimo, junto con el tiempo promedio de ejecución (en segundos).

Para la instancia SSP_1, el GA implementado en CPU obtiene la relación de error más baja con respecto a los otros algoritmos y las tres versiones SNS. El algoritmo SNS ha obtenido valores muy cercanos del óptimo. Para las instancias SSP_2 y SSP_3, el SNS obtiene la proporción más baja de error mientras que los algoritmos SNS^{exp} y SNS^{exp}-1 en las instancias SSP_2 y SSP_3 han obtenido una tasa de error más pequeña que las aproximaciones no sistólicas. Los resultados de tiempo demuestran que los tres modelos SNS obtienen el tiempo más de ejecución mas corto, y en particular SNS es el algoritmo de ejecución más rápido de todos los evaluados.

Tabla 7.3: Resultados de *fitness* para las instancias SSP en 30 ejecuciones utilizando como condición de parada 8421376 evaluaciones.

	SSP_1		SSP_2		SSP_3	
Algoritmo	**RE%**	**Tiempo**	**RE%**	**Tiempo**	**RE%**	**Tiempo**
RS$_{GPU}$	33.54	9.56	4.54	7.56	8.54	6.89
RS$_{CPU}$	6.54	72.93	3.54	74.93	7.59	734.99
GA$_{GPU}$	1.81	723.94	3.81	743.94	4.82	77.09
GA$_{CPU}$	**1.57**	43.56	0.56	43.11	3.59	45.94
SNS	4.36	**0.62**	**0.06**	**2.84**	**0.98**	**3.84**
SNS^{exp}	2.36	21.29	1.41	17.44	3.85	19.26
SNS^{exp}-1	1.83	32.77	1.60	29.79	8.63	32.44
Test	+		+		+	

Los modelos SNS demuestran que para este tipo de problemas la GPU es un excelente contenedor para la ejecución de una técnica de optimización sistólica. El modelo se ajusta a este tipo de arquitectura generando resultados de gran calidad. Se ha pretendido maximizar el número de hilos funcionando en paralelo, y usar un operador de perturbación simple que sea efectivo para realizar exploración/explotación, para así minimizar el tiempo de ejecución. Contrariamente a la intuición, la eliminación de filas (modelos SNS^{exp} y SNS^{exp}-1) significa tener modelos de ejecución más lentos que el SNS canónico, ya que necesitan de llamadas adicionales del kernel de optimización para conseguir el mismo número constante de las evaluaciones. Con respecto a la confianza estadística, podemos afirmar que nuestro algoritmo SNS ha obtenido estadísticamente mejores resultados que los otros algoritmos comparados.

Estudio 2

Los resultados del segundo estudio se detallan en la Tabla 7.4, que muestra el nombre del algoritmo usado en la primera columna, y la segunda columna el tamaño de población utilizada. Posteriormente, se incluye la relación de error y el número promedio de evaluaciones realizadas por cada algoritmo (#Evals.) en las instancias. Se observa que, para un tiempo de ejecución predefinido, SNS es el mejor algoritmo debido a que la relación de error es la más pequeña.

De hecho, las tres versiones de SNS presentan una tasa de error más pequeña que todos los otros algoritmos. Además, encontramos significación estadística entre las versiones del SNS con respecto a los otros cuatro algoritmos.

Tabla 7.4: Resultados de *fitness* para las instancias SSP en 30 ejecuciones utilizando como condición de parada un tiempo de ejecución predefinido de 5 segundos.

	SSP_1		SSP_2		SSP_3	
Algoritmo	**RE%**	**#Evals**	**RE%**	**#Evals.**	**RE%**	**#Evals.**
RS_{GPU}	58.17	4804485	23.61	59754382	19.61	48044805
RS_{CPU}	9.12	576805	1.62	676828	13.62	576805
GA_{GPU}	26.29	58155	3.99	84153	24.99	56796
GA_{CPU}	15.12	956970	1.12	1257513	10.89	956970
SNS	**1.04**	**63160320**	**0.03**	**12981617**	**0.37**	**10867465**
SNS^{exp}	13.66	1913945	0.80	2211688	4.80	1989938
SNS^{exp}-1	14.81	1275965	1.59	1143966	11.18	1445826
Test		+		+		+

Tabla 7.5: Resultados de *fitness* para las instancias SSP en 30 ejecuciones utilizando como condición de parada encontrar el mejor óptimo.

	SSP_1			SSP_2			SSP_3		
Algoritmo	**RE%**	**Tiempo**	**#Evals.**	**RE%**	**Tiempo**	**#Evals.**	**RE%**	**Tiempo**	**#Evals.**
RS_{GPU}	4.92	102.15	490057470	**0.00**	98.31	49904601	**0.00**	98.31	536757470
RS_{CPU}	1.34	90.98	51912450	**0.00**	88.43	48913130	**0.00**	93.52	542143331
GA_{GPU}	**0.00**	985.71	57282675	**0.00**	871.15	56322675	**0.00**	187.15	58461388
GA_{CPU}	**0.00**	52.90	5071941	**0.00**	54.11	5373772	**0.00**	55.17	551238
SNS	**0.00**	**11.44**	**694763520**	**0.00**	**12.46**	**594235672** s	**0.00**	**13.46**	**614236758**
SNS^{exp}	**0.00**	38.08	72729910	**0.00**	35.13	69895398	**0.00**	36.85	68531126
SNS^{exp}-1	**0.00**	43.59	56142460	**0.00**	41.06	57871224	**0.00**	44.06	58748862
Test		+			+			+	

Estudio 3

La Tabla 7.5 muestra la tasa de errores, el tiempo medio en segundos y el número promedio de evaluaciones realizadas por cada algoritmo para las tres instancias. En este caso, el estudio mide la capacidad de los algoritmos en alcanzar el mejor valor óptimo conocido. Para SSP_1, todos los algoritmos excepto RS ha podido encontrar el valor óptimo. Para SSP_2 y SSP_3, todos los algoritmos encontraron el óptimo. Se observa que los valores de la tasa de éxito disminuyen cuando el número de variables aumenta.

En el caso de tiempo, para los tres casos, SNS es el algoritmo que encuentra la solución óptima en todas las ejecuciones en el tiempo más corto para las tres instancias. Los resultados para el análisis estadístico demuestran que hay diferencias estadísticamente significativas entre ellos.

7.2.2. Análisis del problema MKP

En este apartado, analizaremos los resultados obtenidos sobre el MKP. La razón de la elección de este problema está motivada por la necesidad de evaluar nuestra aproximación mediante un problema combinatorios altamente complejo y comparar nuestros resultados a los ya publicados.

Instancias pequeñas

Las Tablas 7.6 y 7.7 incluyen el valor de *fitness* promedio y las Tablas 7.8 y 7.9 el tiempo de ejecución promedio para las instancias pequeñas, sobre 30 ejecuciones independientes.

Tabla 7.6: Valor de *fitness* promedio en 30 ejecuciones para instancias pequeñas del MKP.

Algoritmo	SENTO02	WEISH01	WEISH02	WEISH06	WEISH08	WEISH10	WEISH12	WEISH16
RS_{CPU}	7968.233	4301.400	4319.600	5025.367	5065.533	4677.233	4547.667	5136.367
RS_{GPU}	7923.467	4183.955	4101.000	5047.767	5081.033	4707.767	4694.233	5184.400
GA_{CPU}	**8722.000**	4528.167	**4536.000**	**5557.000**	**5603.800**	6338.933	**6339.000**	**7283.633**
GA_{GPU}	5722.800	3866.900	3994.500	1028.800	1102.900	3693.733	4656.567	6426.267
GA_{SNS}	8721.400	**4554.000**	4531.866	**5557.000**	5592.867	6338.233	6338.500	7282.167
$GA_{SNS^{exp}}$	**8722.000**	**4554.000**	4532.000	**5557.000**	5603.000	6338.900	6336.633	7281.833
SNS	7786.967	4322.787	4283.167	4692.700	4736.100	2324.267	5745.467	5718.267
(*r*)SNS	8335.000	3886.067	3749.886	5281.400	5352.100	5522.367	5441.133	6307.433
(*mr*)SNS	8485.567	**4554.000**	**4536.000**	5504.033	5568.167	6156.800	6115.933	6899.900
(*r.mr*)SNS	8500.533	4486.300	4493.367	5499.833	5564.300	6177.300	6121.567	6874.133
SNS^{exp}	6864.833	**4554.000**	3805.000	4581.133	4556.700	4885.433	4826.033	7239.933
(*r*)SNS^{exp}	8709.133	4538.066	**4536.000**	5550.967	5602.100	**6339.000**	**6339.000**	7275.867
(*mr*)SNS^{exp}	8326.367	4331.000	4281.081	5282.267	5351.100	5507.500	5422.600	6282.333
(*r.mr*)SNS^{exp}	8712.000	4531.000	**4536.000**	5555.633	5604.867	**6339.000**	**6339.000**	7284.233
Valor Óptimo	8722.000	4554.000	4536.000	5557.000	5605.000	6339.000	6339.000	7289.000

Tabla 7.7: Valor de *fitness* promedio en 30 ejecuciones para instancias pequeñas del MKP.

Algoritmo	WEISH18	WEISH22	WEISH30	PB5	PB6	HP1	HP2	WEING7
RS_{CPU}	7771.067	5490.433	8292.733	2117.933	585.167	3348.967	3036.667	933266.800
RS_{GPU}	7855.267	5437.767	8261.567	2117.267	567.533	3354.533	3027.833	931947.967
GA_{CPU}	**9547.667**	8795.767	11035.567	2124.900	758.000	3338.267	3156.000	1089106.000
GA_{GPU}	5075.033	7947.533	6886.233	1726.300	694.767	2893.367	2526.367	786536.333
GA_{SNS}	9554.033	8884.467	11113.533	2117.100	**776.000**	3286.133	3119.767	1090633.333
$GA_{SNS^{exp}}$	9554.767	**8841.800**	11093.333	2105.167	**776.000**	3331.000	3171.000	1091188.333
SNS	7731.500	5532.567	7539.900	1965.400	631.367	3173.900	3008.833	1089370.833
(*r*)SNS	8783.800	6918.800	9267.800	2072.633	639.333	3320.667	3024.433	1082577.333
(*mr*)SNS	8944.700	7459.233	9835.967	2137.300	729.967	3400.067	3140.867	1042101.667
(*r.mr*)SNS	8959.100	7462.367	9842.833	**2138.433**	737.100	3400.867	3141.633	1040627.667
SNS^{exp}	6989.200	7248.100	7659.200	1903.600	477.033	3124.367	2784.067	830926.200
(*r*)SNS^{exp}	9542.967	8829.800	11117.767	2136.733	775.533	**3412.600**	3171.333	1093839.667
(*mr*)SNS^{exp}	8811.367	7379.067	9890.200	2054.400	656.533	3313.467	3021.900	1083721.000
(*r.mr*)SNS^{exp}	9542.667	8831.633	**11132.200**	**2138.433**	775.633	3412.167	**3176.500**	**1093953.000**
Valor Óptimo	9580.000	8947.000	11191.000	2139.000	776.000	3418.000	3186.000	1095445.000

En las Tablas 7.6 y 7.7 se observa claramente que el algoritmo (r, mr)SNS^{exp} obtiene los valores mayores (mejores) en 9 de las 16 instancias evaluadas. Para este modelo, la utilización de perturbaciones aleatorias y movimientos entre filas han posibilitado armar un algoritmo que obtiene los mejores resultados en este grupo. La ventaja de (r, mr)SNS^{exp} se basa principalmente en tres factores: SNS^{exp} trabaja con un menor número de soluciones que el modelo canónico, por lo tanto, el número de iteraciones máximas permitidas aumenta significativamente y esto conlleva que existirán más cambios sobre las soluciones. El segundo y tercer factor están relacionados con las dos variaciones añadidas al comportamiento canónico de las versiones SNS y SNS^{exp}: cambios aleatorios y los movimientos de soluciones entre las filas. La capacidad de cambiar i valores aleatoriamente (siendo i el número de elementos a ser perturbados en cada solución por fila) aumenta la diversidad en las soluciones y evita la convergencia prematura de la población. Además, el flujo de soluciones entre las filas hace que sea posible modificar un grupo más diverso de elementos dentro de las soluciones a través de las iteraciones, reduciendo la posibilidad de caer en un óptimo local (en algunos casos al permanecer en la misma fila para siempre). La combinación de estos factores nos ha permitido mejorar el modelo SNS canónico considerablemente.

Para el caso de versiones exponenciales, la reducción de soluciones no afecta demasiado en los tiempos de ejecución. Las versiones SNS^{exp} obtienen los resultados numéricos más cercanos al valor óptimo, alcanzándolo incluso en algunos casos. En general, las versiones SNS canónicas obtienen los valores de *fitness* más bajos (peores) en comparación con las versiones extendidas.

Los resultados de la Tablas 7.8 y 7.9 demuestran que el algoritmo canónico SNS obtiene los tiempos más bajos en 13 de las 16 instancias. Estos tiempos he han obtenido debido a que el SNS proporciona una estructura simple que puede maximizar la ejecución en la arquitectura GPU. SNS ha buscado aumentar la eficiencia paralela manteniendo ocupados una cantidad considerables de procesadores (e hilos), además de realizando tareas simples que evitan ejecuciones secuenciales. El *speedup* obtenido para el SNS canónico va desde $1.025\times$ a $33\times$. En general, las mayores diferencias en *speedup* de la familia SNS están relacionados con las versiones de GA.

Instancias Medianas

En este apartado el análisis se divide en dos grupos, el primero de ellos consiste en 10 mochilas con 250 y 500 elementos respectivamente. Las Tablas 7.10 y 7.11 presentan el *fitness* promedio de las mejores soluciones obtenidas en cada ejecución y el tiempo de ejecución promedio para este grupo. El segundo grupo de instancias medianas analizadas tiene 30 mochilas y una variación de 250 o 500 elementos y los resultados de*fitness* y tiempo de ejecución se muestran en las Tablas 7.12 y 7.13. La mejor solución para cada caso no se conoce con lo cual se utilizó como valor de referencia aquellos encontrados en publicaciones anteriores relacionadas [23, 61, 130].

Tabla 7.8: Tiempos (s) de ejecución promedio para 30 ejecuciones para instancias pequeñas del MKP.

Algoritmo	SENTO02	WEISH01	WEISH02	WEISH06	WEISH08	WEISH10	WEISH12	WEISH16
RS$_{CPU}$	1.895	0.224	0.233	0.332	0.374	0.405	0.395	0.482
RS$_{GPU}$	7.001	1.188	1.233	1.718	1.718	1.709	1.709	1.702
GA$_{CPU}$	3.473	0.899	0.966	1.097	1.236	1.065	1.063	1.051
GA$_{GPU}$	9.627	2.108	1.988	2.814	2.928	2.755	2.748	2.697
GA$_{SNS}$	2.044	0.317	0.317	0.428	0.429	0.528	0.527	0.635
GA$_{SNS^{exp}}$	2.008	0.315	0.312	0.416	0.416	0.515	0.514	0.613
SNS	0.424	0.098	0.106	0.149	0.148	0.118	0.118	0.104
(*r*)SNS	**0.423**	**0.108**	**0.115**	**0.152**	**0.155**	**0.121**	**0.120**	**0.104**
(*mr*)SNS	0.467	0.113	0.129	0.160	0.163	0.129	0.129	0.122
(*r.mr*)SNS	0.465	0.112	0.131	0.161	0.159	0.130	0.130	0.122
SNS^{exp}	0.850	0.103	0.127	0.324	0.303	0.274	0.276	0.253
(*r*)SNS^{exp}	0.896	0.106	0.139	0.372	0.341	0.312	0.314	0.291
(*mr*)SNS^{exp}	0.851	0.105	0.125	0.330	0.301	0.274	0.277	0.254
(*r.mr*)SNS^{exp}	0.897	0.120	0.148	0.370	0.340	0.311	0.315	0.291

Tabla 7.9: Tiempos (s) de ejecución de 30 ejecuciones para instancias pequeñas del MKP.

Algoritmo	WEISH18	WEISH22	WEISH30	PB5	PB6	HP1	HP2	WEING7
RS$_{CPU}$	0.551	0.611	0.764	0.252	1.325	0.180	0.250	0.668
RS$_{GPU}$	1.707	1.702	1.916	2.848	7.067	1.517	1.510	1.406
GA$_{CPU}$	1.031	1.020	1.133	1.706	3.565	1.058	1.044	0.892
GA$_{GPU}$	2.742	2.748	3.097	4.166	9.565	2.527	2.548	2.640
GA$_{SNS}$	0.739	0.827	1.060	0.317	1.382	0.275	0.345	0.969
GA$_{SNS^{exp}}$	0.721	0.802	1.028	0.312	1.370	0.270	0.339	0.924
SNS	**0.096**	**0.090**	**0.101**	**0.468**	**0.583**	**0.181**	**0.149**	**0.080**
(*r*)SNS	**0.096**	0.091	0.103	0.476	0.593	0.183	0.150	0.082
(*mr*)SNS	0.122	0.126	0.154	0.490	0.599	0.194	0.158	0.144
(*r.mr*)SNS	0.465	0.112	0.131	0.161	0.159	0.130	0.130	0.122
SNS^{exp}	0.182	0.169	0.184	0.761	1.018	0.418	0.290	0.153
(*r*)SNS^{exp}	0.208	0.197	0.218	0.833	1.063	0.485	0.338	0.186
(*mr*)SNS^{exp}	0.176	0.165	0.181	0.765	1.013	0.420	0.290	0.147
(*r.mr*)SNS^{exp}	0.203	0.190	0.212	0.830	1.059	0.489	0.338	0.179

Tabla 7.10 muestra que todos los algoritmos no han alcanzado el valor óptimo. En aquellas instancias con 250 elementos el GA_{CPU} obtuvo resultados competitivos y han superado por muy poca diferencia en los valores de *fitness* a la familia de SNS^{exp}. Es interesante resaltar que las versiones SNS y SNS^{exp} han obtenido valores por debajo del GA_{CPU}, excepto en el caso de la instancia 10.250.3 donde $(r)SNS^{exp}$ obtiene el valor más alto (mejor). Las instancias con 500 elementos muestran que las diferentes versiones SNS^{exp} alcanzan valores cerca del óptimo con respecto al resto de algoritmos, en particular el (r)SNS^{exp} y (r, mr)SNS^{exp}. De acuerdo con estos resultados, ningún algoritmo es el claro ganador. La Tabla 7.11 indica claramente que el conjunto de versiones SNS^{exp} son los algoritmos más rápidos en este grupo instancias. En particular, el (mr)SNS^{exp} obtiene los valores de ejecución más cortos que el resto de los algoritmos. Observamos que a medida que el número de elementos aumenta, los valores de tiempo para los modelos SNS^{exp} no aumentan significativamente, en contraste con los tiempos de las versiones SNS que son 4 veces o más grande a medida que el número de elementos se incrementa.

Tabla 7.10: Valor de *fitness* promedio en 30 ejecuciones para instancias medianas del MKP.

Algoritmo	10.250.0	10.250.1	10.250.2	10.250.3	10.250.4	10.500.0	10.500.1	10.500.2	10.500.3	10.500.4
RS_{CPU}	35505.000	31745.000	31590.000	34566.667	30148.000	90033.333	87566.667	85620.000	84966.667	89533.333
RS_{GPU}	38710.000	34423.333	36030.000	33970.000	37033.333	91933.333	92060.000	861660.000	87340.000	82490.000
GA_{CPU}	**56670.700**	**55528.533**	**57138.600**	56169.567	**55053.433**	96793.933	97281.167	97306.900	97062.033	95584.433
GA_{GPU}	53305.333	52667.600	51083.100	54200.300	51731.000	97462.867	94731.367	92768.367	98765.433	93401.100
GA_{SNS}	49864.267	49698.533	48761.300	51302.333	49220.633	95538.667	96390.367	96059.533	95839.700	95839.700
$GA_{SNS^{exp}}$	50804.333	50421.700	49399.967	52315.433	50012.367	96364.167	96164.833	95965.867	97725.400	96725.400
SNS	35316.667	36836.667	41696.667	36853.333	38663.333	89780.000	87466.667	89833.333	86720.000	81822.000
(*r*)SNS	40116.667	41846.667	44846.667	39736.667	39296.667	91110.000	87566.667	86860.000	93566.667	92466.667
(*mr*)SNS	45343.000	46344.333	40253.333	43443.333	42415.000	90966.667	92466.667	93233.333	94033.333	92833.333
(*r*,*mr*)SNS	46485.000	43030.667	43693.333	47577.667	46639.333	97766.667	96220.000	95029.000	97730.000	92800.000
SNS^{exp}	47213.000	46953.133	46750.200	47759.967	45569.867	94301.233	98732.900	94621.800	95090.867	93399.667
(*r*)SNS^{exp}	55190.767	54544.800	53724.367	**56980.467**	54309.833	103744.500	104950.733	**104923.700**	**104310.200**	102697.033
(*mr*)SNS^{exp}	52324.767	51418.000	50748.333	53592.167	51607.367	102831.300	103514.100	104116.167	103239.233	101480.367
(*r*,*mr*)SNS^{exp}	55231.400	54547.667	53806.133	56886.000	54287.400	104018.733	**105008.567**	104796.433	104279.967	**102792.300**
Valor Óptimo	59187.000	58781.000	58097.000	61000.000	58092.000	117821.000	119249.000	119215.000	118829.000	116530.000

Tabla 7.11: Tiempos (s) de ejecución promedio de 30 ejecuciones para instancias medianas del MKP.

Algoritmo	10.250.0	10.250.1	10.250.2	10.250.3	10.250.4	10.500.0	10.500.1	10.500.2	10.500.3	10.500.4
RS_{CPU}	9.307	9.148	9.307	9.146	8.974	35.466	35.465	35.542	35.442	35.482
RS_{GPU}	10.098	10.097	10.098	10.098	10.097	20.181	20.180	20.181	20.181	20.192
GA_{CPU}	4.256	4.254	4.253	4.400	4.426	8.372	8.375	8.388	8.383	8.371
GA_{GPU}	16.172	16.154	16.117	16.101	16.103	30.050	30.030	30.079	30.064	30.048
GA_{SNS}	11.458	11.437	11.422	11.440	11.431	46.974	46.985	46.982	47.100	47.100
$GA_{SNS^{exp}}$	10.870	10.860	10.852	10.863	10.857	42.571	42.568	42.582	42.650	42.650
SNS	1.084	1.083	1.082	1.086	1.081	4.402	4.400	4.399	4.397	4.398
(*r*)SNS	1.088	1.089	1.089	1.088	1.087	4.381	4.388	4.387	4.383	4.389
(*mr*)SNS	2.301	2.302	2.301	2.301	2.301	9.402	9.404	9.403	9.403	9.405
(*r*,*mr*)SNS	2.300	2.302	2.301	2.301	2.302	9.401	9.412	9.412	9.401	9.408
SNS^{exp}	**0.388**	0.387	0.387	0.389	0.388	0.551	0.553	0.554	0.552	0.553
(*r*)SNS^{exp}	0.482	0.480	0.480	0.482	0.481	0.886	0.888	0.889	0.889	0.888
(*mr*)SNS^{exp}	**0.388**	**0.385**	**0.385**	**0.387**	**0.387**	**0.545**	**0.545**	**0.547**	**0.546**	**0.546**
(*r*,*mr*)SNS^{exp}	0.473	0.470	0.469	0.471	0.471	0.788	0.789	0.790	0.790	0.789

Tabla 7.12: Valor de *fitness* promedio en 30 ejecuciones para instancias medianas del MKP.

Algoritmo	30.250.0	30.250.1	30.250.2	30.250.3	30.250.4	30.500.0	30.500.1	30.500.2	30.500.3	30.500.4
RS_{CPU}	32715.000	33173.000	31912.033	36312.007	33730.166	88912.933	88172.667	85620.000	84966.667	89533.333
RS_{GPU}	31010.337	30153.000	32800.333	31587.500	31215.000	82303.000	82009.233	83850.100	82899.267	80488.900
GA_{CPU}	46187.333	48070.467	45941.367	46340.300	46621.400	96663.133	95872.100	91832.500	92045.233	92561.133
GA_{GPU}	49254.833	48158.111	47211.333	46998.661	46868.000	93154.847	90745.007	88248.100	91702.400	92019.900
GA_{SNS}	47265.100	48831.967	47115.700	47395.233	47479.767	95113.913	94881.667	92566.133	91223.933	91887.600
$GA_{SNS^{exp}}$	45851.467	49617.900	45560.800	45244.200	47203.967	92221.991	91047.100	87811.333	90983.766	91748.167
SNS	30156.100	30812.233	30656.133	31453.000	32122.833	87800.300	87411.113	81811.521	81136.100	79152.233
(r)SNS	32556.033	32181.367	31652.500	31384.033	31115.467	82035.167	81347.833	82988.900	83251.167	92466.267
(mr)SNS	32455.400	33300.333	31248.133	31348.667	32466.333	85018.400	82862.611	83112.400	81485.533	83050.633
(r,mr)SNS	38455.233	35315.467	35167.600	38154.933	36415.100	91414.000	92000.100	90588.933	91545.800	89333.500
SNS^{exp}	40355.050	43486.100	42546.120	40154.100	41886.441	90152.133	95135.100	91311.000	91133.011	89152.000
$(r)SNS^{exp}$	50458.717	**53521.300**	51200.167	51258.117	51009.183	101911.011	101450.133	104323.300	101210.900	99719.053
$(mr)SNS^{exp}$	50113.767	50125.010	49868.991	47998.127	50611.117	100587.100	101384.100	102156.107	100111.241	100447.173
$(r,mr)SNS^{exp}$	**53455.000**	51247.067	**53806.133**	**56886.000**	**54287.400**	**104018.733**	**105008.567**	**104796.433**	**101243.113**	**101032.550**
Valor Óptimo	58842.000	58418.000	56614.000	56930.000	56629.000	115950.000	114810.000	116683.000	115301.000	116435.000

Tabla 7.13: Tiempos (s) de ejecución promedio para 30 ejecuciones para instancias medianas del MKP.

Algoritmo	30.250.0	30.250.1	30.250.2	30.250.3	30.250.4	30.500.0	30.500.1	30.500.2	30.500.3	30.500.4
RS_{CPU}	23.641	23.479	23.430	23.459	23.415	92.481	92.491	92.685	92.486	92.470
RS_{GPU}	26.474	26.470	26.463	26.470	26.474	52.895	52.894	52.891	52.893	52.896
GA_{CPU}	9.838	9.850	10.068	9.848	9.836	19.777	19.782	19.834	19.815	19.829
GA_{GPU}	32.695	32.690	32.706	32.660	32.677	64.685	64.714	64.702	64.686	64.687
GA_{SNS}	26.187	26.169	26.170	26.182	26.168	108.605	108.556	108.652	108.683	108.685
$GA_{SNS^{exp}}$	24.986	24.977	24.970	24.986	24.975	98.773	98.712	98.788	99.114	98.847
SNS	2.710	2.713	2.709	2.713	2.710	11.362	11.350	11.354	11.366	11.307
(r)SNS	2.729	2.736	2.732	2.735	2.731	11.381	11.401	11.400	11.393	11.396
(mr)SNS	5.471	5.473	5.469	5.472	5.471	20.633	20.647	20.636	20.629	20.629
(r,mr)SNS	5.473	5.469	5.472	5.471	5.473	20.633	20.646	20.641	20.635	20.633
SNS^{exp}	0.844	0.846	0.844	0.844	0.844	1.153	1.156	1.157	1.152	1.155
$(r)SNS^{exp}$	1.017	1.018	1.017	1.017	1.016	1.917	1.934	1.934	1.925	1.906
$(mr)SNS^{exp}$	**0.843**	**0.8444**	**0.843**	**0.843**	**0.843**	**1.137**	**1.139**	**1.138**	**1.135**	**1.124**
$(r,mr)SNS^{exp}$	0.981	0.982	0.981	0.981	0.981	1.618	1.626	1.621	1.617	1.598

Entre las versiones SNS y SNS^{exp}, se puede ver claramente que la familia SNS^{exp} obtiene las soluciones más cercanas al valor óptimo. Las extensiones ayudan al SNS canónico y SNS^{exp} a mejorar los resultados. En general, SNS^{exp} ha demostrado un comportamiento estable como el número de elementos aumenta.

En cuanto a las implementaciones RS, estos algoritmos no superan los valores obtenidos de la familia SNS^{exp}. Entre las versiones deñ GA, sólo el GA$_{CPU}$ muestra resultados competitivos contra SNS^{exp}. Estos resultados vuelven a mostrar que el enfoque sistólico encaja con el modelo de trabajo de la arquitectura GPU. Además, esto indica que el aumento del número de elementos no conduce a una degradación sustancial en la calidad de la soluciones obtenidas por las aproximaciones sistólicas. En general, podemos ver una significación estadística en (r) SNS^{exp} y (r, mr)SNS^{exp} con respecto a los RS$_{CPU}$, RS$_{GPU}$, y GA$_{GPU}$ y las diferentes versiones del SNS.

La Tabla 7.12 muestra los valores promedio de *fitness* y Tabla 7.13 indica el tiempo de ejecución en 30 ejecuciones independientes para el segundo grupo de instancias. La Tabla 7.12 muestra claramente que (r, mr)SNS^{exp} supera al resto de algoritmos en 9 de las 10 instancias, pero no logran alcanzar el valor óptimo de *fitness*. Con respecto a los tiempo de ejecución, los valores más bajos se obtienen por el (mr)SNS^{exp} para todo el grupo. Además, los tiempos obtenidos por el algoritmo (r, mr)SNS^{exp} es muy similar a los del (mr)SNS^{exp}, lo cual indica que el (r, mr)SNS^{exp} obtiene soluciones muy cerca al valor óptimo en tiempos de ejecución muy cortos.

Con respecto a los resultados obtenidos por los grupos SNS y SNS^{exp}, sólo la versiones canónicas obtienen valores de *fitness* peores que las del resto de versiones de cada grupo. Claramente, podemos ver que las extensiones propuestas (cambios aleatorios, y el movimiento de soluciones entre las filas) han generado una mejora significativa sobre las versiones SNS y SNS^{exp} canónicas. En particular, el uso de ambas extensiones con el modelo SNS^{exp} ha demostrado una clara robustez del algoritmo para los diversos tamaños de instancia con resultados muy cercanos al valor óptimo.

En cuanto a las implementaciones de RS, el comportamiento de los resultados es el mismo que el observado en el primer grupo de instancias. El RS no excede los resultados numéricos y temporales obtenidos por nuestros enfoques para cualquier instancia.

Como comentario final con respecto a este conjunto de instancias medianas, se observa un comportamiento robusto de las diferentes versiones SNS^{exp}. En particular, aquellas versiones con extensiones han obtenido los mejores resultados que alcanzan valores cercanos del valor óptimo en tiempos de ejecución muy cortos. Asimismo, podemos ver que existe significación estadística de (r)SNS^{exp} y (r, mr)SNS^{exp} con respecto a los algoritmos RS y GA en ambas versiones.

Instancias Grandes

La Tabla 7.14 muestra los valores promedio de *fitness* y la Tabla 7.15 presenta los tiempos promedio en segundos de ejecución de cada algoritmo para el grupo de las instancias grandes.

Tabla 7.14: Valor de *fitness* promedio en 30 ejecuciones para las instancias grandes del MKP.

	GK08	**GK09**	**GK10**	**GK11**
RS$_{CPU}$	16748.667	55445.100	54007.600	85775.900
RS$_{GPU}$	16788.467	55450.967	54016.067	85716.133
GA$_{CPU}$	15967.100	53002.733	52036.767	81326.333
GA$_{GPU}$	15418.500	51589.800	51056.533	65364.900
GA$_{SNS}$	16528.333	54512.133	52080.933	82964.533
GA$_{SNS^{exp}}$	16332.067	54995.333	53478.133	83932.667
SNS	16987.267	NA	NA	NA
(r)SNS	16670.533	NA	NA	NA
(mr)SNS	16907.500	NA	NA	NA
(r,mr)SNS	16915.667	NA	NA	NA
SNS^{exp}	17091.667	55346.067	53915.967	87255.400
(r)SNS^{exp}	17288.333	57952.467	**56383.300**	88604.400
(mr)SNS^{exp}	**18485.900**	57610.967	56235.000	87510.033
(r,mr)SNS^{exp}	18293.833	**57974.230**	56380.133	**88666.833**
Valor Óptimo	18806.000	58087.000	57297.000	95237.000

Tabla 7.15: Tiempos (s) de ejecución en 30 ejecuciones para las instancias grandes del MKP.

Algoritmo	**GK08**	**GK09**	**GK10**	**GK11**
RS$_{CPU}$	146.640	278.974	526.842	1725.676
RS$_{GPU}$	85.378	53.932	102.792	200.818
GA$_{CPU}$	30.865	20.442	36.869	71.005
GA$_{GPU}$	99.997	72.559	129.484	287.455
GA$_{SNS}$	174.156	442.671	818.155	3575.626
GA$_{SNS^{exp}}$	158.309	314.956	569.796	1812.432
SNS	18.629	NA	NA	NA
(r)SNS	18.744	NA	NA	NA
(mr)SNS	31.443	NA	NA	NA
(r,mr)SNS	31.446	NA	NA	NA
SNS^{exp}	1.771	1.970	3.073	8.327
(r)SNS^{exp}	2.834	3.322	5.283	12.812
(mr)SNS^{exp}	**1.737**	**1.942**	**2.970**	**8.160**
(r,mr)SNS^{exp}	2.384	2.762	4.374	11.146

Con respecto a las Tabla 7.14 y Tabla 7.15, tenemos que señalar para las instancias $GK09$, $GK10$ y $GK11$ existen algoritmos que no son capaces de ejecutarse y están marcados con el símbolo NA. Esta marca indica que esas instancia no han podido ser evaluadas por el modelo algorítmico sobre la GPU dada las características propias del algoritmo y del modelo de tarjeta gráfica, dado que se necesita espacio de memoria mayor al disponible en la tarjeta.

En la Tabla 7.14 la mejor solución para cada instancia no se conoce con lo cual se ha utilizado como valor de referencia los existentes en la literatura [129]. Los resultados de la tabla evidencian que ningún algoritmo ha sido capaz de alcanzar el valor óptimo. El SNS^{exp} obtiene los valores más altos de *fitness* (más cercano al óptimo conocido) en esta serie de experimentos. En particular, (r, mr)SNS^{exp} es mejor en la instancia $GK009$ y $GK11$, mientras que los algoritmos (mr)SNS^{exp} y (r)SNS^{exp} obtienen mejores resultados para las instancias $GK08$ y $GK10$. Además, la Tabla 7.14 indica claramente que el (mr)SNS^{exp} es el algoritmo más rápido en todas las instancias. Con respecto a los valores de *speedup*, esta oscila desde $1,01\times$ hasta $439\times$.

Entre las versiones SNS y SNS^{exp}, podemos obtener conclusiones de manera general de acuerdo a los datos de la instancia $GK08$, en los que los resultados de la familia SNS^{exp} tiene valores de *fitness* más altos (mejores) al mejor óptimo conocido que los obtenidos por la familia SNS. Una vez más, vemos que se repite el comportamiento de los resultados observados con los grupos de instancias anteriores. Las versiones SNS^{exp} con diferentes extensiones son aquellas que obtienen los mejores resultados. En cuanto a los tiempos, el grupo SNS^{exp} obtiene los tiempos más bajos en comparación con la familia SNS.

La Tabla 7.15 muestra una diferencia clara entre las implementaciones SNS^{exp} y las versiones GAs. Con respecto a los resultados RS, estos no superan a ningún resultado de nuestros enfoque desde el punto de vista numérico ni en la eficiencia del tiempo.

Se evidencia una significancia estadística para las versiones (r)SNS^{exp} y (r, mr)SNS^{exp} con respecto a ambas implementaciones de GA y RS.

Finalmente, para ilustrar los efectos del comportamiento de la población del SNS y SNS^{exp}, mostramos una ejecución de ejemplo del SNS (Figuras 7.1, 7.2, 7.3, 7.4, 7.5 y 7.6) y del SNS^{exp} (Figuras 7.7, 7.8, 7.9, 7.10, 7.11 y 7.12) para la instancia WEISH16. Las figuras son instantáneas de la población tomadas cada 20 % de evaluaciones hasta completar el 100 %. Cuanto más azul esté coloreado una solución, peor (más bajo) es su valor de *fitness*; el color rojo representa la solución óptima al problema.

Por un lado, el SNS mejora gradualmente las soluciones ubicadas en las filas superiores de la malla generando soluciones prometedoras que alcanzan soluciones óptimas o muy cercanas a la misma. Por otro lado, existen soluciones de mala calidad durante toda la ejecución en las filas inferiores. En el caso del SNS^{exp}, se observa una rápida evolución de las soluciones en casi todas las columnas excepto en la última. Se puede ver que el al existir mayor cantidad de movimientos se generan mayor número perturbaciones ha permitido en muchas de las filas alcanzar el óptimo.

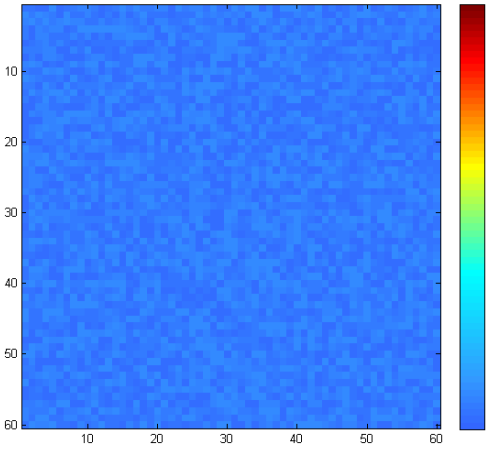


Figura 7.1: 0 % de evals.

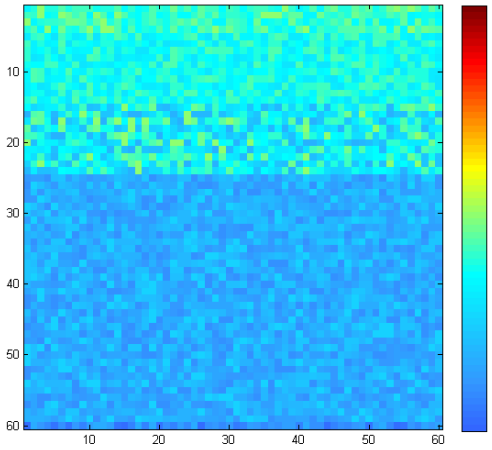


Figura 7.2: 20 % de evals.

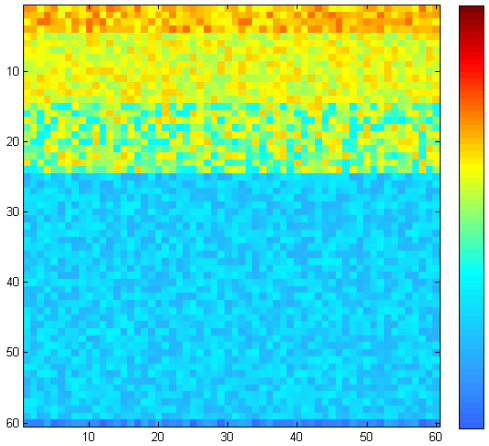


Figura 7.3: 40 % de evals.

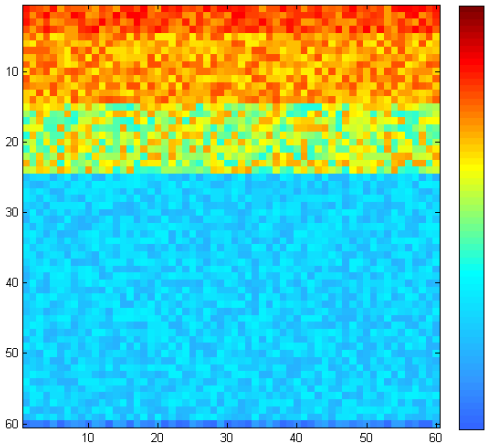


Figura 7.4: 60 % de evals.

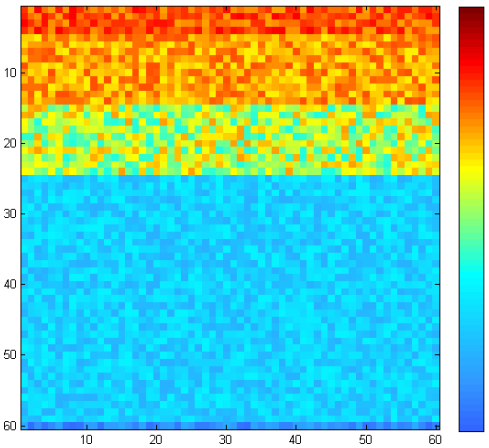


Figura 7.5: 80 % de evals.

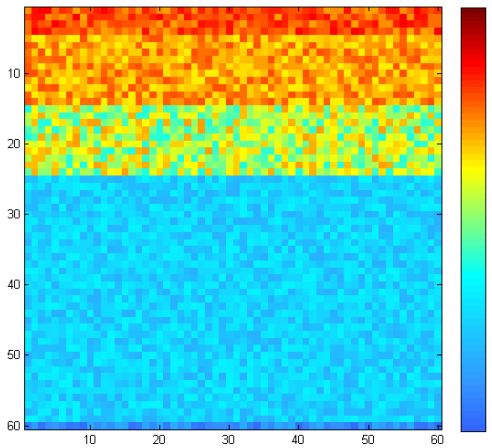


Figura 7.6: 100 % de evals.

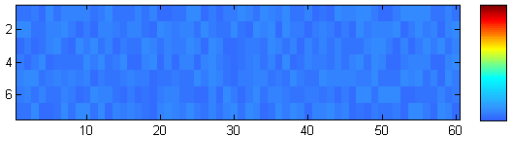


Figura 7.7: 0 % de evals.

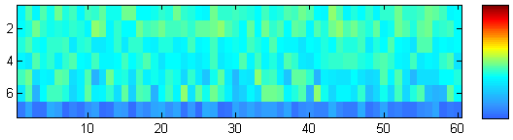


Figura 7.8: 20 % de evals.

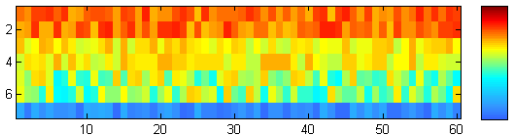


Figura 7.9: 40 % de evals.

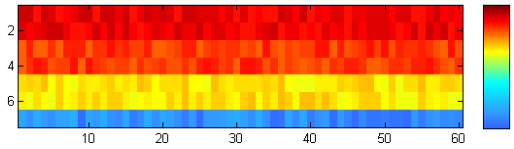


Figura 7.10: 60 % de evals.

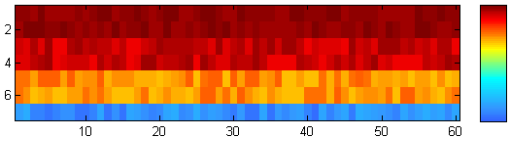


Figura 7.11: 80 % de evals.

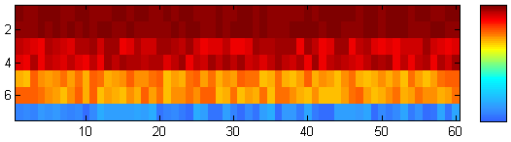


Figura 7.12: 100 % de evals.

7.3. Conclusiones

En este capítulo se ha presentado los resultados de un nuevo modelo algorítmico conocido como SNS y ademas hemos analizado diversas variaciones sobre diferentes conjunto de instancias para el SSP y el MKP. También, hemos comparado estos resultados con los de los diferentes implementaciones de RS y GA en CPU y GPU, respectivamente.

Estos resultados muestran que, con respecto a nuestros planteamientos, (r, mr)SNS^{exp} es el mejor algoritmo en general. Este algoritmo genera una búsqueda sistemática e inteligente a través del espacio de búsqueda, superando el resto de los algoritmos en la mayoría de los casos.

La eficiencia de los modelos canónicos SNS y SNS^{exp} han permitido mostrar que el modelo basado en computación sistólica se adapta perfectamente a la arquitectura GPU.

En general, todos los modelos SNS^{exp} han superado los valores de *fitness* de las versiones RS. Esto nos permite a decir que el modelo sistólico realiza una búsqueda de manera inteligente y competente en comparación con un muestreo estocástico completamente ciego. En cuanto a las diferencias con las versiones GA, las versiones SNS no superan los valores obtenidos por GA_{CPU}. Para el caso de la familia SNS^{exp} y las versiones GA, los resultados han sido similares e incluso las versiones exponenciales han superado ampliamente en algunas instancias. Para el caso de las instancias medianas/grandes, el SNS^{exp} ha mejorado los resultados obtenidos por los GAs.

En cuanto a nuestro enfoque en relación con el número de elementos y mochilas, nos encontramos con que los mejores resultados se obtienen por (r, mr)SNS^{exp}. Esto demuestra que nuestra aproximación soporta trabajar con diferentes tamaños de instancia sin una gran pérdida de tiempo y calidad en la solución. Para el problema SSP observamos resultados similares, donde el comportamiento de las versiones SNS^{exp} es el mejor en comparación al modelo canónico y al resto de algoritmos evaluados (RS y GA).

Se ha observado que la introducción de cambios aleatorios y movimientos paralelos de soluciones puede proporcionar una mejora significativa en nuestros algoritmos. Estos cambios mejoran tanto el tiempo de cómputo como la calidad final de las soluciones obtenidas.

Es importante tener en cuenta que estamos probando nuestro algoritmo contra una base canónica de RS y GA, dado que nuestro algoritmo no busca centrarse en un problema específico, sino para resolver el problema de diseñar un algoritmo para una arquitectura GPU sin operadores que introduzcan métodos específicos para el MKP.

Por otra parte, se ha visto que una de las cuestiones más relevantes es el equilibrio de la comunicación y la computación que existe en nuestro modelo. Por otra parte, el paralelismo proporciona un entorno donde estos cambios son bastante fáciles de realizar y permiten una ejecución de manera rápida. Estas características combinadas con las dos extensiones que mayor rédito han dado para los problemas de optimización han proporcionado una mejoría en nuestro enfoque inicial. La adición de los movimientos de solución a través de las filas y los cambios aleatorios ha permitido superar los resultados del resto de los algoritmos evaluados permitiendo alcanzar el óptimo mejor conocido en la mayoría de las instancias.

Capítulo 8

Análisis experimental del HySyS

En este capítulo se analizan los resultados de la evaluación de la paralelización híbrida denominada HySyS. Para el estudio experimental utilizamos 3 instancias del problema MMDP y SSP, mientras que para el MAX-CUT trabajamos con 4 instancias bien diferenciadas. Con estos estudios queremos analizar la influencia de los parámetros de los componentes algorítmicos, y evaluar si los resultados cambian del modo sugerido por la capacidad de búsqueda inherente de HySyS o bien es por la naturaleza de algunos de sus componentes individualmente.

El capítulo se organiza de la siguiente forma. La sección 8.1 presenta los parámetros utilizados en los experimentos y describe las instancias de los problemas utilizados y la elección de las mismas. La sección 8.2 analiza los resultados obtenidos tras la aplicación de las técnicas comparadas a los 3 grupos de instancias. La Sección 8.2.3 muestra más detalles sobre la dinámica del algoritmo HySyS cuando se resuelven las instancias y el comportamiento del mismo. Por último, la Sección 8.3 presenta las conclusiones finales extraídas del trabajo experimental.

8.1. Diseño experimental

Para este apartado se presentan la configuración experimental utilizada en este estudio. En la Sección 8.1.1 explicamos en detalle las características de las instancias seleccionadas. Posteriormente, en la Sección 8.1.2 se describen los algoritmos han sido usados como base comparativa con nuestro trabajo. Finalmente, se describe la parametrización utilizada.

8.1.1. Instancias seleccionadas

Para la evaluación del HySyS se utilizan tres problemas: MMDP, SSP, y el MAX-CUT (Capítulo 4). El MMDP tiene como distintivo el uso del parámetro k como regulador del grado de multimodalidad. En este caso hemos considerado instancias grandes según la literatura, $k = \{20, 30, 40\}$. Cada instancia se denomina mediante la etiqueta $MMDP_k$.

Para el SSP se han creado un grupo de instancias de manera similar al método usado en [68]. El tamaño de W ha sido definido en $\{1000, 5000, 10000\}$ y los elementos del vector W han sido generados aleatoriamente a partir de una distribución uniforme con un intervalo $[0, 10^4]$. Los tres problemas han sido llamados SSP_i, donde i es un índice que va desde 1 a 3.

Finalmente, el último problema a analizar es el MAX-CUT, del cual hemos utilizado tres instancias previamente empleados en la literatura [68] con tamaños que varían entre 20 y 100 elementos (denominadas 20_01, 20_09, and 100). Además, hemos seleccionado una instancia que se llamara G1[1] que consta de 800 elementos.

8.1.2. Algoritmos de base comparativa

Para evaluar el comportamiento del algoritmo HySyS, se compara respecto la ejecución de cada uno de sus componentes algorítmicos por separado. En la CPU, se ha implementado el μGA y un RS. En la GPU se emplea un SNS^{exp}. Se quiere comprobar si el comportamiento cooperativo emergente de la hibridación hardware supera a cada uno los componentes por separado. Por otra parte, consideramos al μGA como una búsqueda inteligente avanzada clásica que servirá para contextualizar los resultados de HySyS.

Además de la comparación de nuestra propuesta contra otros tres algoritmos canónicos de la literatura, la experimentación ha sido diseñada para demostrar las capacidades de escalabilidad de nuestra aproximación algorítmica híbrida CPU-GPU.

8.1.3. Parametrización

Con respecto al entorno de ejecución, hemos trabajado sobre dos variaciones, Configuración 2 y 3 (Capítulo 4.4) buscando enfatizar nuestro estudio en la ejecución de HySyS sobre dos arquitecturas GPUs diferentes (Fermi y Kepler). Hemos trabajado con la configuración 2 para la ejecución de todos los algoritmos y la configuración 3 ha sido utilizada para realizar una ejecución aparte de HySyS solamente. Los resultados de HySyS en cada modelo de GPU son llamados $HySyS\_n650$ y $HySyS\_n780$, respectivamente. Estos nombres están compuestos por el nombre del algoritmo seguido del modelo de tarjeta gráfica usada.

[1] `http://www.optsicom.es/maxcut/`

8.2. Resultados de HySyS

Para un análisis más detallado de los resultados de nuestro enfoque con respecto a los otros algoritmos, se analizan primero los resultados numéricos y de tiempos de ejecución para, posteriormente, realizar un análisis de rendimiento y escalabilidad únicamente enfocado a HySyS.

8.2.1. Resultados numéricos

En esta sección se presentan y analizan la calidad de las soluciones alcanzadas por los algoritmos evaluados. Las Tablas 8.1, 8.2 y 8.3) incluyen los valores promedio de *fitness* ($Avg.$) y la tasa de éxito ($\%Hits$) para las instancias de los tres problemas de optimización seleccionados.

Con respecto al problema MMDP, la Tabla 8.1 muestra los valores promedio de *fitness* y el hit-rate promedio para 30 ejecuciones independientes. Se puede observar claramente que ambos modelos HySyS obtiene los mejores valores (más altos) para todas las instancias consideradas, las diferencias con respecto a los otros algoritmos empleados se vuelve mayor a medida que el tamaño del problema crece. De hecho, nuestra propuesta encuentra al menos una vez el valor óptimo en todas las instancias. El segundo mejor algoritmo que obtiene resultados es el μGA para este grupo instancias, donde el valor óptimo es encontrado al menos una vez. El RS_{CPU} obtiene los peores resultados con respecto a los otros algoritmos.

Tabla 8.1: Valores de *fitness* promedio ($Avg.$) y tasa de éxito ($\%Hits$) para las instancias MMDP en 30 ejecuciones independientes.

Algoritmo	***MMDP*_20**		***MMDP*_30**		***MMDP*_40**	
	Avg.	**% Hit**	***Avg.***	**% Hit**	***Avg.***	**% Hit**
RS$_{CPU}$	13.399	0.00 %	18.552	0.00 %	23.756	0.00 %
μGA$_{CPU}$	18.810	23,33 %	28.279	13.33 %	28.968	0.00 %
SNS^{exp}	16.348	0.00 %	23.915	0.00 %	26.143	0.00 %
HySyS_*n*650	**19.619**	**90.67 %**	29.221	40.00 %	**36.803**	**6.67 %**
HySy_*n*780	19.601	86.67 %	**29.368**	**56.67** %	36.719	**6.67 %**
Valor Óptimo	20.000		30.000		40.000	

Habiendo evaluado el rendimiento de HySyS contra cada técnica por separado, HySyS puede encontrar con un porcentaje de éxito del 90 % la solución óptima para $MMDP\_20$, 56 % para $MMDP\_30$, 6 % para $MMDP\_40$. Los resultados obtenidos indican claramente que HySyS puede explorar el espacio de búsqueda mejor que sus componentes por separado, mostrando así un comportamiento emergente bastante prometedor.

La Tabla 8.2 muestra los resultados para las instancias del problema SSP. La primera conclusión que se puede extraer de la Tabla 8.2 es que casi todos los algoritmos encontraron la mejor solución conocida al menos una vez (excepto RS_{CPU} para la instancia SSP_3). Para el problema SSP,

también es notable que el SNS^{exp} alcanza el 100% de éxito en todas las instancias SSP_1 y SSP_2. El algoritmo μGA es el segundo algoritmo que demuestra ser efectivo para la resolución del problema SSP. En la Tabla 8.2 se observa que para las instancias SSP el algoritmo HySyS obtiene resultados competitivos y ha superado a todos los otros algoritmos. En el caso de RS_{CPU} y μGA_{CPU}, hay una diferencia estadística con respecto a la calidad de las soluciones encontradas.

Tabla 8.2: Valores de *fitness* promedio ($Avg.$) y tasa de éxito ($\%Hit$) para las instancias SSP en 30 ejecuciones independientes.

Algoritmo	***SSP_1***		***SSP_2***		***SSP_3***	
	Avg.	**% Hits**	***Avg.***	**% Hits**	***Avg.***	**% Hits**
RS$_{CPU}$	2384738.0	26.67%	12006252.0	6,67%	14961670.0	0.00%
μGA$_{CPU}$	2664167.0	90.00%	12193134.0	13.34%	23982636.0	36.67%
SNS^{exp}	**2684659.0**	**100.00%**	**12481273.0**	**100.00%**	24812667.0	60.00%
HySyS_*n*650	**2684659.0**	**100.00%**	**12481273.0**	**100.00%**	**24872631.0**	**100.00%**
HySyS_*n*780	**2684659.0**	**100.00%**	**12481273.0**	**100.00%**	**24872631.0**	**100.00%**
Valor Óptimo	2684659.0		12481273.0		24872631.0	

La Tabla 8.3 muestra los valores de *fitness* medio para las instancias MAX-CUT. El algoritmo HySyS muestra ser el mejor de todos los evaluados en esta comparación, ya que encuentra los mejores valores promedio de *fitness* (más altos) en las cuatro instancias consideradas. HySyS ha podido obtener soluciones de mayor calidad mediante la combinación de ambas técnicas (μGA y SNS^{exp}), en lugar de utilizar cada uno de ellas por separado, las cuales han obtenidos valores cercanos al óptimo pero peores que las de HySyS.

Tabla 8.3: Valores de *fitness* promedio ($Avg.$) y tasa de éxito ($\%Hit$) para las instancias MAX-CUT en 30 ejecuciones independientes.

Algoritmo	**Instancias MAX-CUT**							
	20_01		**20_09**		**100**		***G*1**	
	Avg.	**% Hits**	***Avg.***	**% Hits**	***Avg.***	**% Hits**	***Avg.***	**% Hits**
RS$_{CPU}$	9.119800	6,67%	55.678740	0.00%	571.366	0.00%	13619.733	0.00%
μGA$_{CPU}$	10.110205	26.67%	55.918000	0.00%	807.150	0.00%	16894.000	0.00%
SNS^{exp}	**10.119812**	**100.00%**	56.499010	60.00%	957.266	0.00%	15716.016	0.00%
HySyS_*n*650	**10.119812**	**100.00%**	**55.992004**	83.33%	**1075.483**	**73.33%**	**17887.968**	0.00%
HySyS_*n*780	**10.119812**	**100.00%**	**56.115299**	**90.00%**	1075.101	70.00%	17814.563	0.00%
Valor Óptimo	10.119812		56.740064		1077.000		19176.000	

En resumen, HySyS presenta valores muy cercanos al valor óptimo aun cuando los tamaños de instancia comienzan a crecer. De hecho, a medida que el tamaño de la instancia crece, el μGA y el SNS^{exp} comienzan a enfrentar mayores dificultades para encontrar valores quasi-óptimos. Para

las instancias SSP, hemos observado que HySyS obtiene una tasa de éxito muy elevada en todas las instancias, mientras que el resto de algoritmos queda siempre por detrás según este indicador de calidad. Por lo tanto, se puede concluir HySyS puede resolver eficientemente instancias grandes. Esto se ve apoyado por los resultados de las instancias de los otros dos problemas. El algoritmo HySyS sigue encontrando el valor óptimo al menos una vez. Por lo tanto, nuestra propuesta puede proporcionar una búsqueda más robusta en estas condiciones experimentales.

8.2.2. Resultados de tiempo de ejecución

La Tabla 8.4 presenta los resultados de los tiempos de ejecución promedio medidos en segundos para las instancias MMDP en 30 ejecuciones independientes. Los resultados demuestran que HySyS_n780 obtiene los tiempos más cortos en todas las instancias. El algoritmo HySyS_n650 obtiene tiempos cortos muy cercanos al HySyS_n780, especialmente en aquellas instancias con menor cantidad de elementos. Para la instancia MMDP_20 los tiempos son muy similares entre el algoritmo SNS^{exp} y las dos versiones de HySyS. Esto demuestra que aunque HySyS incorpore los comportamientos de μGA y SNS^{exp}, el número de iteraciones realizadas y el tiempo de ejecución del algoritmo HySyS es inferior a las iteraciones empleadas para cada componente algorítmico por separado.

Tabla 8.4: Tiempos (s) de ejecución para la instancias MMDP en 30 ejecuciones independientes.

Algoritmo	$MMDP\_20$	$MMDP\_30$	$MMDP\_40$
RS$_{CPU}$	3.639	5.281	6.966
μGA$_{CPU}$	0.305	0.431	0.556
SNS^{exp}	0.078	0.272	0.783
HySyS_n650	0.055	0.056	0.065
HySyS_n780	**0.042**	**0.047**	**0.059**

Podemos observar que para las instancias pequeñas, los tiempos de ejecución son similares. En aquellas instancias más grandes se puede ver que, para ambos HySyS, los tiempos de ejecución son menores a los obtenidos por el SNS^{exp}. Esto se puede justificar si vemos que el rendimiento de μGA con HC consume parte de las evaluaciones, finalizando la ejecución de HySyS casi al mismo tiempo como un SNS^{exp}. Si se tiene en cuenta el rendimiento numérico, se puede afirmar que HySyS muestra el comportamiento más robusto, pudiendo alcanzar o estar cerca del óptimo en el menor tiempo de ejecución.

Ahora pasamos a analizar los tiempos de ejecución para el problema SSP. La Tabla 8.5 indica que los tiempos más cortos de ejecución se obtienen con $HySyS\_n780$ en todas las instancias.

Para este tipo de problemas se observa una clara diferencia en el tiempo de ejecución entre los dos modelos HySyS. En general, los resultados revelan un rendimiento más rápido con la arquitectura CPU-GPU. En las instancias más grandes, el algoritmo SNS^{exp} tiene un menor tiempo de cómputo que el μGA. Por último, el RS_{CPU} obtiene los tiempos de ejecución más largos en cada instancia SSP.

Tabla 8.5: Tiempo (s) de ejecución para la instancias SSP en 30 ejecuciones independientes.

Algoritmo	***SSP*_1**	***SSP*_2**	***SSP*_3**
RS$_{CPU}$	25.942	129.756	262.525
μGA$_{CPU}$	1.773	7.808	15.336
SNS^{exp}	0.177	3.409	11.292
HySyS_*n*650	0.085	2.766	9.868
HySyS_*n*780	**0.071**	**2.133**	**6.966**

Los resultados para las instancias MAX-CUT son presentados en la Tabla 8.6. En particular, RS$_{CPU}$ tiene los tiempos de ejecución más largos en comparación con el resto de los algoritmos. La ejecución de HySyS en la plataforma de computo Geforce GTX 780 ha generado el tiempo promedio más corto de ejecución, posteriormente lo sigue el μGA y en tercera posición el HySyS ejecutado sobre la GTX 650. Para este último caso, tanto esta aproximación como el SNS^{exp} obtienen tiempos similares de ejecución. A medida que el tamaño de las instancias va aumentando, HySyS genera menores tiempos de ejecución (con ambos modelos) en comparación con los otros algoritmos.

Tabla 8.6: Tiempos (s) de ejecución para la instancias MAX-CUT en 30 ejecuciones independientes.

Algoritmo	**Nombre de instancias MAX-CUT**			
	20_01	**20_09**	**100**	***G*1**
RS$_{CPU}$	2.781	2.838	51.916	3116.330
μGA$_{CPU}$	0.199	0.206	3.031	172.189
SNS^{exp}	0.336	0.336	0.857	32.142
HySyS_*n*650	0.339	0.337	0.818	24.171
HySyS_*n*780	**0.115**	**0.183**	**0.524**	**18.761**

8.2.3. Análisis de escalabilidad sobre HySyS

Este apartado está dedicado exclusivamente a evaluar el comportamiento de escalabilidad de HySyS y el rendimiento observado. Para este análisis, también hemos tenido en cuenta el tiempo y los resultados numéricos explicados anteriormente.

A continuación se detalla el tiempo de ejecución de los principales procesos considerados para la aproximación HySyS_$n650$. Con este estudio se quiere poner de manifiesto la necesidad de reportar el coste de los diversos procesos que intervienen en una ejecución CPU-GPU en la que se realizan transferencias entre diferentes memorias. Se busca definir una línea base de comparación del rendimiento para el comportamiento colaborativo. Hemos seleccionado el primer modelo de la GPU ya que los tiempos de ejecución entre los dos modelos son similares y también es interesante para evaluar el comportamiento de este enfoque en tarjetas gráficas de uso personal.

Nos hemos centrado en tres operaciones: (1) El tiempo empleado por el proceso principal de evolución. (2) La transferencia de datos entre las dos arquitecturas (no referente a las soluciones). (3) Por último, el tiempo necesario para mover soluciones entre el anfitrión (host) y el dispositivo GPU. La primera operación mide el tiempo que ambas técnicas (es decir, SNS^{exp} y μGA) aplican sus respectivos operadores sobre las soluciones. La segunda operación es un punto importante porque brinda la posibilidad de saber si el HySyS_$n650$ está consumiendo demasiado tiempo al transferir información entre la CPU y la GPU. Es bien sabido que la transferencia de información en el modelo CPU-GPU-CPU puede ser un cuello de botella en el rendimiento de los algoritmos. La última operación se relaciona con el tiempo de ejecución total que HySyS $_n650$ utiliza para mover soluciones entre CPU y GPU. El control de estos tiempos proporciona información sobre el comportamiento de los diferentes procesos implicados en el algoritmo. También nos permite analizar si existe un cuello de botella o un retraso significativo en la ejecución principal.

La Figura 8.1 presenta el porcentaje de tiempo dedicado a las operaciones nombradas en el párrafo anterior. La figura muestra (para cada instancia) el porcentaje de tiempo utilizado con respecto a los tiempos de ejecución totales.

Como primera observación, en la Figura 8.1 se puede apreciar los tiempos largos relacionado con la comunicación y transferencia de datos. Estos valores son mayores en las instancias que presentan un menor número de elementos. Por ejemplo, para las instancias MMDP, se obtiene el porcentaje de tiempo más grande, el que disminuye con el tamaños de la instancia. De hecho, para las instancias MMDP los rangos porcentuales varían entre el 25 % y 30 % del tiempo total de ejecución; para las instancias SSP, del 5 % al 19 % y, finalmente, llega a un 1 % para la instancia más grande del problema MAX-CUT.

Como una segunda observación, el tiempo de intercambio de soluciones en la mayoría de los casos es inferior a 10 %; sólo para los las instancias MMDP el algoritmo llega a un máximo de

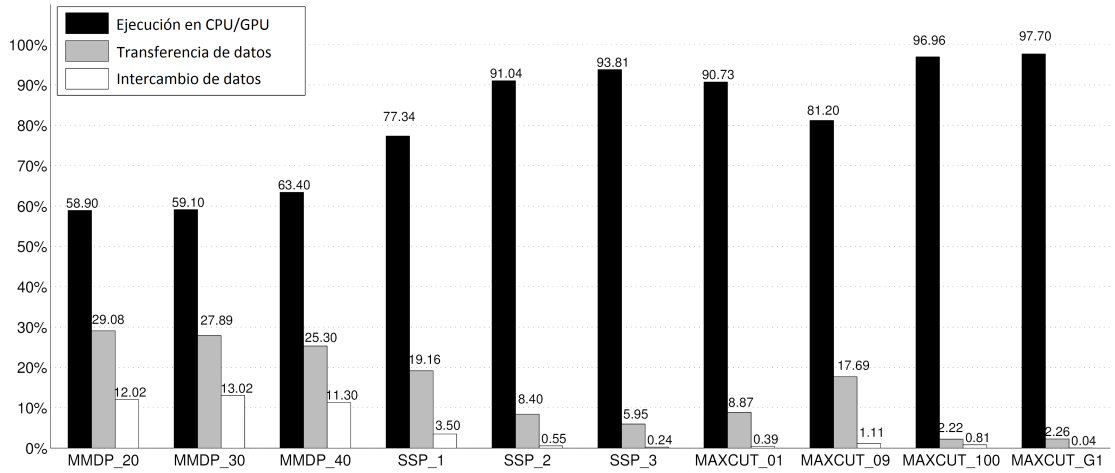


Figura 8.1: Porcentaje promedio del tiempo empleado por el algoritmo HySys_$n650$ para las tres operaciones de mayor consumo.

13 %. Estos valores indican que el intercambio de soluciones utiliza alrededor del 10 % del tiempo total de ejecución. Estos tiempos pueden ser explicados teniendo en cuenta que el número de iteraciones a realizar disminuye a medida que el número de elementos aumenta. Por lo tanto, la oportunidad de transferir soluciones entre los dispositivos se reduce considerablemente.

Como observación final de la Figura 8.1, podemos afirmar que la ventaja de una distribución completa en las tareas entre la CPU y la GPU demuestra que el algoritmo HySyS sigue siendo rápido y eficaz a pesar de que para instancias pequeñas existe un tiempo bastante grande utilizado debido a las operaciones de transferencia. La transmisión de datos realizada por el modelo algorítmico es eficiente con respecto al tiempo de ejecución. Por otra parte, este proceso es también el que menos tiempo utiliza, lo que nos puede llevar a idear nuevas variantes basados en esta metodología. Como resultado de ello, se puede concluir que el tiempo asociado con la comunicación y/o transferencia afecta directamente el tiempo de ejecución en los casos más pequeños, pero es insignificante para los casos más grandes.

8.3. Conclusiones

Este capítulo ha presentado la implementación de un nuevo modelo algorítmico híbrido paralelo sobre GPU: el algoritmo de Búsqueda Sistólica Híbrida, HySyS. Nuestra idea es utilizar este modelo como base para ser el primero de muchos otros enfoques que se pueden derivar del uso de la misma metodología que HySyS. Hemos tratado de adaptar el concepto de metaheurísticas híbridas que exhiben comportamientos complementarios para mejorar la eficacia y robustez en una arquitectura complementaria/comunicativa del tipo CPU-GPU.

La principal contribución científica de este capítulo es el nuevo modelo híbrido que explota dos métodos de optimización computacionales enfocados cada uno a funcionar en una arquitectura *hardware* en particular. Por un lado, el uso del modelo sistólico sobre una arquitectura GPU es novedoso, y la realización de todos estos experimentos ha reforzado su potencial como técnica de optimización. Por otro lado, la utilización de los recursos de la CPU como parte de un proceso de optimización y no relegado a un simple controlador de entrada/salida también es novedoso. Nuestra aproximación HySyS constituye un modelo original donde es posible introducir nuevas técnicas según la necesidad del problema y reutilizar componentes de optimización diseñados para cada plataforma de cómputo.

Nuestro estudio ha tenido en cuenta tres problemas de optimización combinatoria comúnmente empleados en el campo de los EAs. Los resultados experimentales muestran que la hibridación CPU-GPU tiene rendimientos de ejecución significativos haciendo plena utilización de todos los recursos computacionales disponibles tanto de la CPU y GPU. Estas paralelizaciones proporcionan las ventajas básicas de los procedimientos paralelos. Por otro lado, los operadores del algoritmo μGA son muy importantes debido a su baja complejidad computacional, ya que no demandan un cálculo demasiado largo y costoso logrando un incremento de la capacidad de resolución numérica y la factibilidad del espacio de búsqueda. HySyS ha alcanzando valores de *fitness* óptimos en tiempos muy cortos y demostrando una alta escalabilidad y robustez de acuerdo a las instancias analizadas.

Parte

3

Conclusiones y trabajo futuro

Capítulo 9

Conclusiones

Este capítulo resume las principales conclusiones extraídas del presente trabajo de tesis doctoral dedicada a los algoritmos paralelos para optimización con metaheurísticas utilizando unidades de procesamiento gráfico como plataforma de cómputo [72, 114].

La paralelización propuesta de diversas metaheurísticas, tanto existentes como especialmente diseñadas teniendo en cuenta las características de cómputo de las GPUs, ha permitido mejorar la eficacia y eficiencia de las mismas en diferentes problemas de optimización académicos y del mundo real. No obstante, si se quiere sacar provecho a toda la potencia de cómputo de las GPUs, la implementación ha requerido un trabajo muy minucioso para considerar las características intrínsecas de la nueva plataforma de cómputo. En este trabajo hemos buscado definir líneas y modelos algorítmicos nuevos también, que permitan usar al máximo las plataformas disponibles en un nuevo concepto interesante de colaboración CPU+GPU.

Inicialmente, se ha introducido el concepto de optimización, la necesidad del uso de algoritmos aproximados (no exactos) como las metaheurísticas, a la necesidad también del paralelismo, y a los principales modelos paralelos para metaheurísticas existentes en la actualidad. En esta línea, se introduce la GPU como arquitectura de cómputo de alto rendimiento y se presenta una descripción del modelo de programación CUDA, su distribución de hilos y jerarquía de memorias.

Para poder establecer bien las bases de las contribuciones científicas del trabajo, se ha realizado una profunda revisión de la literatura sobre la paralelización de metaheurísticas en GPU. Se han analizado un gran número de trabajos y se han clasificado atendiendo a los diferentes modelos paralelos que implementan. Esta revisión del estado del arte también ha permitido detallar los dominios de aplicación de los trabajos publicados.

El siguiente paso ha consistido en definir las tres contribuciones novedosas en el dominio de las metaheurísticas paralelas en una arquitectura CPU-GPU: un cGA multi-GPU, un algoritmo de búsqueda sistólica por vecindario (SNS) y un modelo paralelo híbrido que es capaz de aprovechar tanto la CPU como la GPU durante su ejecución. En el primer caso, el punto de partida ha sido un algoritmo cGA sobre GPU y, atendiendo a la estructuración de la población del mismo, se ha propuesto una paralelización que divide dicha población entre diferentes tarjetas para aprovechar la potencia de cómputo de todas ellas. En el caso de SNS, el análisis de la literatura reveló que muchos trabajos consistían únicamente en portar un algoritmo existente a la GPU. Por el contrario, nuestra hipótesis de trabajo ha sido diseñar un algoritmo que se ajuste especialmente a las características de cómputo de las GPUs, y para ello se ha utilizado la base de la *Computación Sistólica*. Partiendo de un modelo canónico inicial, se han ido proponiendo sucesivas mejoras hasta definir un modelo muy eficience a la vez que efectivo. Por último, la línea de trabajo de la tercera propuesta también ha resultado del análisis de la literatura, en el que existen un escaso número de trabajos en los que se propongan paralelizaciones CPU-GPU en las que ambas plataformas colaboren en la búsqueda. De ahí ha surgido nuestra propuesta denominada HySyS, diseñada con dos objetivos en mente:(1) la mejora en las soluciones y (2) reducción de los tiempos de ejecución maximizando el uso de los recursos de cada arquitectura. Se busca aprovechar, desde un punto de vista hardware, la potencia de cada arquitectura y, desde el punto algorítmico, la colaboración entre técnicas de optimización que permitan utilizar obtener resultados de calidad de una forma rápida y aplicable a problemas de diferentes contextos.

Una vez presentadas las propuestas, se ha desarrollado un trabajo experimental minucioso para evaluar cada una de ellas. Así, el estudio realizado con los algoritmos cGA-GPU y cGA-multiGPU utilizando problemas de diversas características ha conducido a la distribución eficiente de la población para cada modelo. Se han obtenido ganancias de 8× a 770× con respecto a la versión del cGA secuencial, aunque en el caso del cGA-multiGPU son algo más reducidas que en una sola GPU. En el caso de SNS, es necesario remarcar aquí que, hasta donde conocemos, esta es la primera aproximación de optimización sistólica sobre GPUs en la literatura. Para analizar este algoritmo, se ha realizado una evaluación pormenorizada tanto del modelo canónico inicial, como de las sucesivas mejoras que se le han ido introduciendo (diferentes patrones de perturbación, tamaño de la malla de soluciones, etc.). Los experimentos realizados han mostrado que el algoritmo SNS^{exp} puede ser una técnica altamente especializada para ser ejecutada sobre GPU, tanto por las ganancias en velocidad como por la capacidad de enfrentarse a problemas de complejidad creciente con recursos limitados, ratificando las ventajas de los modelos sistólicos. La última parte de los experimentos se han dedicado a evaluar la propuesta híbrida HySyS. Los resultados han mostrado que la combinación de técnicas en CPU y GPU es capaz de superar numéricamente a cada componente por separado. Hoy en día, el uso de procesadores multi-core en equipos que

instalan además dispositivos GPU permite paralelizar en un ambiente híbrido heterogéneo, lo que abre una línea de trabajo real muy prometedora. En particular, los resultados han mostrado que es posible obtener ganancias de hasta 80× en términos de aceleración (en comparación con una arquitectura CPU de un solo núcleo) en las arquitecturas CPU-GPU.

Las versiones paralelas de los algoritmos presentados en este trabajo se han implementado sobre varias familias de GPUs de NVidia, como son la GeForce GTX 285, GTX 650 y la GTX 780 ti (familia GT200, Fermi y Kepler). En la actualidad, y a partir de ideas como las presentadas este trabajo, la computación GPU aplicada a resolución de problemas de optimizatión es un área en el que compañías y agencias internacionales están comenzando a realizar desarrollos muy importantes orientados a la aplicación, desde la implementación de algoritmos secuenciales para problemas que manejan una gran cantidad de datos, pasando por otras aplicaciones en los dominios de la bioinformatica, el transporte inteligente, la simulación de estructuras genómicas, o la sismología, entre otras. Por tanto, se espera que los resultados de este trabajo de tesis sirvan como base para otros muchos estudios en otros campos relacionados.

Las conclusiones alcanzadas en cuanto a aceleración y precisión en el análisis reflejan las elevadas prestaciones que las tarjetas gráficas programables ofrecen en el ámbito de la optimización metaheurística.

Capítulo 10

Trabajo Futuro

A partir de este trabajo de tesis doctoral surgen diversas líneas muy prometedoras. A continuación se destacan aquellas que consideramos que gozan de un mayor interés para la comunidad científica.

A nivel algorítmico

En relación a los estudios realizados sobre cGAs, creemos que es interesante estudiar la influencia de modelos asíncronos y variación de vecindarios a nivel de bloques de hilos (islas) como forma de optimizar la búsqueda mediante modelos dinámicos de búsqueda, capaces de auto-ajustarse de acuerdo al nivel de explotación/exploración que existe en el cGA-GPU. Con respecto al estudio de los cGAs en el dominio de optimización numérica, proponemos como trabajo futuro incluir los componentes generales en el SNS para evaluar la relación de exploración/explotación del algoritmo, lo que nos podría llevar a mejorar los resultados actuales. Además, habilitaría la posibilidad de realizar un estudio del comportamiento más detallado durante la ejecución por kernel, grid, por bloques o particularmente hilos, de forma que se puedan identificar posibles cuellos de botella, así como problemas de las divergencias de los hilos para evitar la secuencialización. En el caso de HySyS, sería interesante realizar un profundo estudio de ajuste de la parametrización para tratar de mejorar los resultados obtenidos, Adicionalmente, la hibridación de algunos modelos mejorados existentes en la literatura y el uso de sus componentes más importantes nos puede conducir a obtener aún mejores resultados. También, un trabajo interesante sería resolver instancias de un un gran tamaño con estos algoritmos, puesto que el SNS^{exp} y el HySyS reportaron buenos resultados en los problemas de mayor tamaño.

En cuanto al estudio y evaluación de cGAs, SNS y HySyS sobre problemas multi-objetivo y su aplicación a la resolución de problemas del mundo real, son un asunto de trabajo futuro. En concreto, se propone primeramente una modelización de la búsqueda multi-objetivo sobre GPU y la implementación de medidas de evaluación propias para este tipo de problemas de optimización. Es un campo que recientemente está cobrando notoriedad [58, 144].

A nivel de hardware

Desde el punto de vista hardware, la integración de SNS o particularmente HySyS en cluster de GPUs y/o sistemas multi-núcleo puede ser objeto de estudio. En esta línea, desarrollar versiones paralelas multiplataforma capaces de funcionar en arquitecturas como clusters, sistemas multi-núcleo ó clústers de GPUs. Estos clusters son un tipo de arquitectura que cada vez es más popular dado que permiten una explotación híbrida basada en paralelismo funcional entre nodos CPU complementado con procesamiento local en GPUs. Actualmente forman parte de los clusters de computo en muchos centros de investigación (prueba de ello son aquellos estructuras que ocupan las primeras posiciones en la lista de supercomputadores más rápidos). En este sentido, OpenCL se presenta como una oportunidad para implementar algoritmos paralelos que puedan ser ejecutados sobre múltiples plataformas.

Finalmente, si bien existe una gran heterogeneidad de GPUs en el mercado, creemos que es necesario adaptarse a los avances tan espectaculares que se están produciendo en el campo de las tarjetas gráficas. Así, sería interesante estudiar el ajuste de los algoritmos paralelos para beneficiarse de las nuevas características que instalan las ultimas tarjetas de la familia Maxwell, como son el manejo de la memoria, la ejecución paralela de kernels y la intercomunicación entre los diversos componentes. El objetivo sería mejorar aún más los resultados obtenidos e incluso plantear nuevos modelos algorítmicos para diversos contextos de problemas no trabajados hasta el momento.

A nivel del ámbito aplicación

Por último, se pretende aplicar todo el conocimiento adquirido durante el transcurso del doctorado a problemas propios del contexto local, regional y nacional del lugar de origen del doctorando a partir del trabajo desarrollado aquí. Dentro de la industria del petróleo y gas existen muchos problemas que pueden ser encarados desde el ámbito de optimización, desde disposición de red de tuberías hasta la extracción de estos componentes propiamente dichos, petroleo, gas y shale. De la misma forma, cuando se considera zonas industriales, es de interés abordar el crecimiento desmesurado en algunos casos de la zona urbana, en otros de la estructuración de las calles y las direcciones de las mismas. En este sentido, es posible aplicar lo aprendido para

desarrollar sistemas avanzados de información al viajero, sistemas avanzados de gestión del tráfico y sistemas avanzados de optimización del tráfico. Finalmente, la optimización de los ciclos de los semáforos se encuentra incluida dentro de los problemas a tratar. Se busca conocer las magnitudes de tráfico que circulan por la ciudad, constituyéndose en una herramienta muy útil para plantear soluciones a los problemas detectados o aquellos que se puedan prever, tanto en el corto como en el largo plazo.

Parte

4

Apéndices

Apéndice A

Relación de publicaciones que respaldan la tesis doctoral

Este apéndice resume la producción científica del autor de esta tesis en colaboración con sus directores. Estas publicaciones han sido fruto de las investigaciones originales desarrolladas a lo largo de esta tesis doctoral. Estas publicaciones avalan el interés, la validez, y las aportaciones de este trabajo en la literatura, ya que estos trabajos han aparecido publicados en foros de prestigio y, por tanto, han sido sometidos a procesos de revisión por reconocidos investigadores especializados.

Revistas indexadas en ISI JCR

[1] P. Vidal, E. Alba, and F. Luna. Systolic neighborhood search on graphics processing units. *Soft Computing*, 18(1): pages 125 – 142, 2014. Factor de Impacto ISI: 1.271. Cuartil: Q3

[2] P. Vidal, E. Alba, and F. Luna. Solving optimization problems using a Hybrid Systolic Search on GPU plus CPU. *Soft Computing*, En segunda vuelta-Minor Revision, 2015.

Congresos internacionales:

[3] P. Vidal and E. Alba. A multi-GPU implementation of a Cellular Genetic Algorithm. *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010*, pages 1–7, 2010. Congreso **CORE A** (Año 2010)

[4] P. Vidal and E. Alba. Cellular Genetic Algorithm on Graphic Processing Units. En Juan Ramón González and David A. Pelta and Carlos Cruz and Germán Terrazas and Natalio

Krasnogor, editors, *Proceedings of the Workshop on Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, pages 223–232, 2010.

Congresos internacionales publicados en la serie *Lecture Notes in Computer Science*:

[5] E. Alba and P. Vidal. Systolic Optimization on GPU Platform. En *Computer Aided Systems Theory - EUROCAST 2011 - 13th International Conference, Las Palmas de Gran Canaria, Spain, February 6-11, 2011, Revised Selected Papers, Part I* , pages 375–383, 2011.

Referencias

[1] E. Alba. *Análisis y Diseño de Algoritmos Genéticos Paralelos Distribuidos.* PhD thesis, University of Málaga, 1999.

[2] E. Alba, editor. *Parallel Metaheuristics: A New Class of Algorithms.* Wiley, 2005.

[3] E. Alba and B. Dorronsoro. *Cellular genetic algorithms*, volume 42. Springer Science & Business Media, 2009.

[4] E. Alba, M. Giacobini, M. Tomassini, and S. Romero. Comparing synchronous and asynchronous cellular genetic algorithms. In *Parallel Problem Solving from Nature—PPSN VII*, pages 601–610. Springer, 2002.

[5] E. Alba, F. Luna, and A. J. Nebro. Advances in parallel heterogeneous genetic algorithms for continuous optimization. *International Journal of Applied Mathematics and Computer Science*, 14(3):101 – 117, 2004.

[6] E. Alba, G. Luque, and S. Nesmachnow. Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research*, 20(1):1–48, 2013.

[7] E. Alba and M. Tomassini. Parallelism and evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 6(5):443–462, 2002.

[8] E. Alba and M. Tomassini. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443 – 462, 2002.

[9] M. G. Arenas, A. M. Mora, G. Romero, and P. A. Castillo. Gpu computation in bioinspired algorithms: a review. In *Advances in Computational Intelligence*, pages 433–440. Springer, 2011.

[10] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, 1996.

[11] E. Bajrami, M. Asic, E. Cogo, D. Trnka, and N. Nosovic. Performance comparison of simulated annealing algorithm execution on gpu and cpu. In *MIPRO, 2012 Proceedings of the 35th International Convention*, pages 1785–1788, May 2012.

[12] F. Berman, G. Fox, and A. Hey. *Grid Computing. Making the Global Infrastructure a Reality.* Communications Networking and Distributed Systems. Wiley, 2003.

[13] G. Blake, R. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, November 2009.

[14] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268 – 308, 2003.

[15] V. Boriakoff. Fft computation with systolic arrays, a new architecture. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 41(4):278–284, 1994.

[16] W. Bozejko and M. Uchronski. Distributed tabu search algorithm for the job shop problem. In *1st Australian Conference on the Applications of Systems Engineering*, page 54, 2012.

[17] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[18] A. Cano, A. Zafra, and S. Ventura. Speeding up the evaluation phase of gp classification algorithms on gpus. *Soft Computing*, 16(2):187–202, 2012.

[19] S. Cavuoti, M. Garofalo, M. Brescia, G. Longo, G. Ventre, et al. Genetic Algorithm Modeling with GPU Parallel Computing Technology. In B. Apolloni, S. Bassis, A. Esposito, and F. C. Morabito, editors, *Neural Nets and Surroundings*, volume 19 of *Smart Innovation, Systems and Technologies*, pages 29–39. Springer Berlin Heidelberg, 2013.

[20] H. Chan and P. Mazumder. A systolic architecture for high speed hypergraph partitioning using a genetic algorithm. In X. Yao, editor, *Progress in Evolutionary Computation*, volume 956 of *Lecture Notes in Computer Science*, pages 109–126. Springer Berlin / Heidelberg, 1995.

[21] J. F. Chicano. *Metaheurísticas e Ingeniería del Software*. PhD thesis, University of Málaga, 2007.

[22] D. Chitty. Fast parallel genetic programming: multi-core cpu versus many-core gpu. *Soft Computing*, 16(10):1795–1814, 2012.

[23] P. Chu and J. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.

[24] I. Coelho, M. Haddad, L. Ochi, M. Souza, and R. Farias. A Hybrid CPU-GPU Local Search Heuristic for the Unrelated Parallel Machine Scheduling Psroblem. In *Applications for Multi-Core Architectures (WAMCA), 2012 Third Workshop on*, pages 19–23, 2012.

[25] I. Coelho, L. Ochi, P. Munhoz, M. Souza, R. Farias, and C. Bentes. The single vehicle routing problem with deliveries and selective pickups in a cpu-gpu heterogeneous environment. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 1606–1611, June 2012.

[26] R. J. Collins and D. R. Jefferson. *Selection in massively parallel genetic algorithms*. Citeseer, 1991.

[27] R. Couturier and C. Guyeux. Pseudorandom number generator on GPU. *Designing Scientific Applications on GPUs*, page 441, 2013.

[28] T. G. Crainic and M. Toulouse. Parallel strategies for metaheuristics. In F. W. Glover and G. A. Kochenberger, editors, *Handbook of Metaheuristics*, USA, 2003.

[29] V.-D. Cung, S. L. Martins, C. C. Ribeiro, and C. Roucairol. Strategies for the parallel implementation of metaheuristics. In *Essays and surveys in metaheuristics*, pages 263–308. Springer, 2002.

[30] V.-D. Cung, S. L. Martins, C. C. Ribeiro, and C. Roucairol. Strategies for the Parallel Implementation of Metaheuristics. In C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 263–308, Norwell, MA, USA, 2003. Kluwer Academic Publishers.

[31] D. Dasgupta and Z. Michalewicz. *Evolutionary algorithms in engineering applications*. Springer Science & Business Media, 2013.

[32] N. A. Davis, A. Pandey, and B. A. McKinney. Real-world comparison of cpu and gpu implementations of snprank: a network analysis tool for gwas. *Bioinformatics*, 27(2):284–285, 2011.

[33] L. de Veronese and R. Krohling. Differential evolution algorithm on the gpu with c-cuda. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–7, July 2010.

[34] A. Delévacq, P. Delisle, and M. Krajecki. Max-min ant system on graphics processing units. In *Third International Conference on Metaheuristics and Nature Inspired Computing*, 2010.

[35] A. Delévacq, P. Delisle, and M. Krajecki. Parallel gpu implementation of iterated local search for the travelling salesman problem. In *Learning and Intelligent Optimization*, LNCS, pages 372–377. Springer, 2012.

[36] J. Demšar. Statistical comparison of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1 – 30, 2006.

[37] T. Desell, D. Anderson, M. Magdon-Ismail, H. Newberg, B. Szymanski, and C. Varela. An analysis of massively distributed evolutionary algorithms. In *Evolutionary Computation (CEC)*, pages 1–8, July 2010.

[38] F. J. Diego, E. M. Gómez, M. Ortega-Mier, and A. García-Sánchez. Parallel CUDA Architecture for Solving de VRP with ACO. In S. P. Sethi, M. Bogataj, and L. Ros-McDonnell, editors, *Industrial Engineering: Innovative Networks*, pages 385–393. Springer London, 2012.

[39] K. Ding and Y. Tan. Comparison of random number generators in Particle Swarm Optimization algorithm. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, July 6-11, 2014*, pages 2664–2671, 2014.

[40] B. Dorronsoro. *Diseño e implementación de algoritmos genéticos celulares para problemas complejos*. PhD thesis, University of Málaga, 2007.

[41] A. Ferreiro, J. García, J. López-Salas, and C. Vázquez. An efficient implementation of parallel simulated annealing algorithm in gpus. *Journal of Global Optimization*, 57(3):863–890, 2013.

[42] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[43] M. A. Franco, N. Krasnogor, and J. Bacardit. Speeding up the evaluation of evolutionary learning systems using gpgpus. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 1039–1046, USA, 2010. ACM.

[44] A. Freville. The multidimensional 0–1 knapsack problem: An overview. *European Journal of Operational Research*, 155(1):1–21, 2004.

[45] A. E. Gamal, L. Hemachandra, I. Shperling, and V. Wei. Using Simulated Annealing to Design Good Codes. *IEEE Trans. Information Theory*, 33(1), January 1997.

[46] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13:533 – 549, 1986.

[47] F. Glover and G. A. Kochenberger. Critical event tabu search for multidimensional knapsack problems. metaheuristics: The theory and applications. pages 407–427, 1996.

[48] F. Glover and G. A. Kochenberger. *Handbook of metaheuristics*. Springer Science & Business Media, 2003.

[49] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[50] F. W. Glover and G. A. Kochenberger. *Handbook of Metaheuristics*. Kluwer, 2003.

[51] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[52] D. E. Goldberg. Sizing Populations for Serial and Parallel Genetic Algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 70–79, 1989.

[53] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, 1:69–93, 1991.

[54] D. E. Goldberg, K. Deb, and J. Horn. Massive Multimodality, Deception, and Genetic Algorithms. In *Parallel Problem Solving from Nature*, pages 37–48. Elsevier, 1992.

[55] R. L. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal of Applied Mathematics*, 17:416 – 429, 1969.

[56] J. J. Grefenstette. *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. Psychology Press, 2013.

[57] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144, April 2011.

[58] S. Gupta and G. Tan. A scalable parallel implementation of evolutionary algorithms for multi-objective optimization on gpus. In *Evolutionary Computation (CEC), 2015 IEEE Congress on*, pages 1567–1574. IEEE, 2015.

[59] M. Gušev, D. J. Evans, and J. Tasič. Fast linear systolic matrix vector multiplication. *International journal of computer mathematics*, 43(3-4):231–248, 1992.

[60] Y. Han, S. Roy, and K. Chakraborty. Optimizing simulated annealing on gpu: A case study with ic floorplanning. In *Quality Electronic Design (ISQED), 2011 12th International Symposium on*, pages 1–7, March 2011.

[61] C. W. S. Hanafi. New convergent heuristics for 0-1 mixed integer programming. *European Journal of Operational Research*, 195(1):62 – 74, 2009.

[62] Y. Hochberg and A. C. Tamhane. *Multiple Comparison Procedures*. Wiley, 1987.

[63] T. Ibaraki, K. Nonobe, and M. Yagiura. *Metaheuristics:: Progress as Real Problem Solvers*, volume 32. Springer Science & Business Media, 2006.

[64] J. Jaros and P. Pospichal. A fair comparison of modern cpus and gpus running the genetic algorithm under the knapsack benchmark. In *Applications of Evolutionary Computation*, volume 7248 of *Lecture Notes in Computer Science*, pages 426–435. Springer Berlin Heidelberg, 2012.

[65] H. G. G. Jr. *Scientific Method in Practice*. Cambridge University Press, 2002.

[66] R. M. Karp. Probabilistic analysis of partitioning algorithms for the traveling salesman problem in the plane. *Mathematics of Operations Research*, 2:209 – 224, 1977.

[67] S. Kestur, J. Davis, and O. Williams. Blas comparison on fpga, cpu and gpu. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 288–293, July 2010.

[68] S. Khuri, T. Bäck, and J. Heitkötter. An Evolutionary Approach to Combinatorial Optimization Problems. In *Proceedings of the 22nd annual ACM computer science conference*, pages 66–73, 1994.

[69] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1981.

[70] J. R. Koza. *Genetic Programming. On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts, 1992.

[71] K. Krishnakumar. Micro-Genetic Algorithms for Stationary and Non-Stationary Function Optimization. In *Intelligent Control and Adaptive Systems*, pages 289–296, 1989.

[72] 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.

[73] P. Kromer, J. Platos, and V. Snasel. Differential evolution for the linear ordering problem

implemented on cuda. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 796–802, June 2011.

[74] P. Kromer, J. Platos, and V. Snásel. Genetic algorithm for clustering accelerated by the cuda platform. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pages 1005–1010. IEEE, 2012.

[75] P. Krömer, V. Snášel, J. Platoš, and A. Abraham. Many-threaded implementation of differential evolution for the cuda platform. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1595–1602. ACM, 2011.

[76] H.-T. Kung. Why systolic architectures? *IEEE computer*, 15(1):37–46, 1982.

[77] H. Kungt and C. E. Leisersont. Systolic arrays (for vlsi). *Society for Industrial & Applied*, page 256, 1979.

[78] A. Lagae, S. Lefebvre, G. Drettakis, and P. Dutré. Procedural noise using sparse Gabor convolution. *ACM Transactions on Graphics*, 28(3):54:1–54:10, 2009.

[79] W. B. Langdon. Large scale bioinformatics data mining with parallel genetic programming on graphics processing units. In F. F. de Vega and E. Cantu-Paz, editors, *Parallel and Distributed Computational Intelligence*, volume 279 of *Studies in Computational Intelligence*, chapter 5, pages 113–141. Springer, 2010.

[80] H. Li and C. Liu. Prediction of protein structures using gpu based simulated annealing. In *Machine Learning and Applications (ICMLA)*, volume 1, pages 630–633, Dec 2012.

[81] F. Luna, E. Alba, and A. J. Nebro. Parallel heterogeneous metaheuristics. In E. Alba, editor, *Parallel Metaheuristics*, pages 395 – 422. Wiley, 2005.

[82] F. Luna, A. J. Nebro, and E. Alba. Parallel evolutionary multiobjective optimization. In N. . Nedjah, E. Alba, and L. de Macedo, editors, *Parallel Evolutionary Computations*, volume 22 of *Studies in Computational Intelligence*, chapter 2, pages 33 – 56. Springer, 2006.

[83] T. V. Luong, N. Melab, and E.-G. Talbi. Gpu-based island model for evolutionary algorithms. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 1089–1096. ACM, 2010.

[84] T. V. Luong, N. Melab, and E.-G. Talbi. GPU Computing for Parallel Local Search Metaheuristic Algorithms. *Computers, IEEE Transactions on*, 62(1):173–185, Jan 2013.

[85] G. Luque. *Resolución de Problemas Combinatorios con Aplicación Real en Sistemas Distribuidos*. PhD thesis, University of Málaga, 2006.

[86] G. Megson. Synthesis of a systolic array genetic algorithm. In *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, Washington, DC, USA, 1998. IEEE Computer Society.

[87] N. Melab, E.-G. Talbi, et al. Local search algorithms on graphics processing units. a case

study: the permutation perceptron problem. In *Evolutionary Computation in Combinatorial Optimization*, pages 264–275. Springer, 2010.

[88] H. Mühlenbein, M. Schomisch, and J. Born. The parallel genetic algorithm as function optimizer. *Parallel Computing*, 17(6–7):619–632, 1991.

[89] M. Mitchell, J. P. Crutchfield, and P. T. Hraber. Evolving cellular automata to perform computations: mechanisms and impediments. *Physica D: Nonlinear Phenomena*, 75(1–3):361 – 391, 1994.

[90] A. Munawar, M. Wahib, M. Munetomo, and K. Akama. Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia CUDA framework. *Genetic Programming and Evolvable Machines*, 10(4):391–415, 2009.

[91] Y. Nashed, P. Mesejo, R. Ugolotti, J. Dubois-Lacoste, and S. Cagnoni. A comparative Study of Three GPU-based metaheuristics. In *Parallel Problem Solving from Nature - PPSN XII*, volume 7492 of *Lecture Notes in Computer Science*, pages 398–407. Springer, 2012.

[92] C.-K. Ng, L.-S. Zhang, D. Li, and W.-W. Tian. Discrete filled function method for discrete global optimization. *Comput. Optim. Appl.*, 31(1):87–115, May 2005.

[93] NVIDIA. *CUDA CURAND Library*. NVIDIA Corporation, Santa Clara, CA, USA, Aug. 2010.

[94] NVIDIA. *Fermi Compute Architecture Whitepaper*, 2012.

[95] NVIDIA. Kepler GK110 Whitepaper, 2012.

[96] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[97] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0, May 2008.

[98] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[99] J. Parejo, A. Ruiz-Cortés, S. Lozano, and P. Fernandez. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing*, 16(3):527–561, 2012.

[100] S. Pllana and F. Xhafa. *Programming Multi-core and Many-core Computing Systems*. Wiley Publishing, 1st edition, 2014.

[101] P. Pospíchal and J. Jaros. Gpu-based acceleration of the genetic algorithm. *GECCO competition*, 2009.

[102] P. Pospíchal, J. Schwarz, and J. Jaros. Parallel genetic algorithm solving 0/1 knapsack problem running on the gpu. In *16th International Conference on Soft Computing MENDEL*, volume 2010, pages 64–70, 2010.

[103] J. Puchinger and G. Raidl. Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In J. Mira and J. Álvarez, editors, *Artificial*

Intelligence and Knowledge Engineering Applications: A Bioinspired Approach, volume 3562 of *Lecture Notes in Computer Science*, pages 41–53. Springer Berlin Heidelberg, 2005.

[104] M. Rabinovich, P. Kainga, D. Johnson, B. Shafer, J. Lee, and R. Eberhart. Particle Swarm Optimization on a GPU. In *Electro/Information Technology (EIT), 2012 IEEE International Conference on*, pages 1–6, 2012.

[105] K. Rocki and R. Suda. High performance gpu accelerated local optimization in tsp. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1788–1796. IEEE, 2013.

[106] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[107] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[108] J. Sarma and K. A. De Jong. An analysis of local selection algorithms in a spatially structured evolutionary algorithm. In *ICGA*, pages 181–187, 1997.

[109] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on cpus and gpus: A case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 351–362. ACM, 2010.

[110] M. Schröck and H. Vogt. Gauge fixing using overrelaxation and simulated annealing on gpus. *arXiv preprint arXiv:1209.4008*, 2012.

[111] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. CRC Press, 2003.

[112] L. Shi and S. Ólafsson. Nested partitions method for global optimization. *Operations Research*, 48(3):390–407, 2000.

[113] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: Survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 1:1–1:10, USA, 2013. ACM.

[114] S. Singh, J. Kaur, and R. S. Sinha. A comprehensive survey on various evolutionary algorithms on gpu. In *International Conference on Communication, Computing & Systems (ICCCS2014). hgpu. org*, 2014.

[115] A. Sinha and D. E. Goldberg. A Survey of Hybrid Genetic and Evolutionary Algorithms, July 2003.

[116] R. S. Sinha and S. Singh. Optimization techniques on gpu. In *International Multi Track Conference on Science, Engineering & Technical Innovations*, page 566–568, CT Group of Institutions, Jalandhar, June 2014.

[117] D. L. Souza, G. D. Monteiro, T. C. Martins, V. A. Dmitriev, and O. N. Teixeira. Pso-gpu:

Accelerating particle swarm optimization in cuda-based graphics processing units. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '11, pages 837–838, USA, 2011. ACM.

[118] A. D. Stivala, P. J. Stuckey, and A. I. Wirth. Fast and accurate protein substructure searching with simulated annealing and gpus. *BMC bioinformatics*, 11(1):446, 2010.

[119] E.-G. Talbi. *Metaheuristics : from design to implementation*, volume 10 of *The Sciences Po series in international relations and political economy*. John Wiley & Sons, 2009.

[120] D. B. Thomas, L. Howes, and W. Luk. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Symposium on Field Programmable Gate Arrays*, pages 63–72, 2009.

[121] T. Ting, X.-S. Yang, S. Cheng, and K. Huang. Hybrid metaheuristic algorithms: Past, present, and future. In X.-S. Yang, editor, *Recent Advances in Swarm Intelligence and Evolutionary Computation*, volume 585 of *Studies in Computational Intelligence*. Springer International Publishing, 2015.

[122] S. Tsutsui and N. Fujimoto. Solving quadratic assignment problems by genetic algorithms with gpu computation: A case study. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 2523–2530, USA, 2009. ACM.

[123] S. Tsutsui and N. Fujimoto. An analytical study of gpu computation for solving qaps by parallel evolutionary computation with independent run. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8, July 2010.

[124] S. Tsutsui and N. Fujimoto. Aco with tabu search on a gpu for solving qaps using move-cost adjusted thread assignment. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1547–1554, USA, 2011. ACM.

[125] R. Ugolotti, Y. Nashed, and S. Cagnoni. Real-time gpu based road sign detection and classification. In *Parallel Problem Solving from Nature - PPSN XII*, volume 7491 of *LNCS*, pages 153–162. Springer Berlin Heidelberg, 2012.

[126] R. Ugolotti, Y. S. Nashed, P. Mesejo, Špela Ivekovič, L. Mussi, and S. Cagnoni. Particle swarm optimization and differential evolution for model-based object detection. *Applied Soft Computing*, 13(6):3092 – 3105, 2013.

[127] T. Van Luong, N. Melab, and E. Talbi. Parallel hybrid evolutionary algorithms on gpu. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8, July 2010.

[128] P. M. Vasant. *Meta-heuristics optimization algorithms in engineering, business, economics, and finance*. IGI Global, 2012.

[129] M. Vasquez and J.-K. Hao. A hybrid approach for the 0–1 multidimensional knapsack

problem. In *In Proceedings of the International Joint Conference on Artificial Intelligence 2001*, pages 328–333, 2001.

[130] M. Vasquez and Y. Vimont. Improved results on the 0-1 multidimensional knapsack problem. *European Journal of Operational Research*, 165:70–81, 2005.

[131] T. Vidal, T. G. Crainic, M. Gendreau, N. Lahrichi, and W. Rei. A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. *Operations Research*, 60(3):611–624, 2012.

[132] M. Wahib, A. Munawar, M. Munetomo, and K. Akama. Optimization of parallel genetic algorithms for nVidia GPUs. In *CEC IEEE*, pages 803–811. IEEE, 2011.

[133] H. Wang, S. Rahnamayan, and Z. Wu. Parallel differential evolution with self-adapting control parameters and generalized opposition-based learning for solving high-dimensional optimization problems. *Journal of Parallel and Distributed Computing*, 73(1):62 – 73, 2013.

[134] R. Weiss and M. C. C. S. Dept. *GPU-accelerated Data Mining with Swarm Intelligence*. Honors project Macalester College. Springer, 2010.

[135] W. Whewell and R. E. Butts. *Theory of Scientific Method*. Hackett Pub Co Inc, 1989.

[136] D. Whitley, S. Rana, J. Dzubera, and K. E. Mathias. Evaluating evolutionary algorithms. *Artificial Intelligence*, 85(1–2):245 – 276, 1996.

[137] G. Woeginger. Exact algorithms for np-hard problems: A survey. In *Combinatorial Optimization — Eureka, You Shrink!*, volume 2570 of *LNCS*, pages 185–207. Springer, 2003.

[138] M. Wong and T. Wong. Implementation of parallel genetic algorithms on graphics processing units. In M. Gen, D. Green, O. Katai, B. McKay, A. Namatame, R. Sarker, and B.-T. Zhang, editors, *Intelligent and Evolutionary Systems*, volume 187 of *Studies in Computational Intelligence*, pages 197–216. Springer Berlin Heidelberg, 2009.

[139] M.-L. Wong and T.-T. Wong. Parallel hybrid genetic algorithms on consumer-level graphics hardware. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 2973–2980, 2006.

[140] M.-L. Wong, T.-T. Wong, and K.-L. Fok. Parallel evolutionary algorithms on graphics processing unit. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 3, pages 2286–2293 Vol. 3, Sept 2005.

[141] A. R. Yildiz. A comparative study of population-based optimization algorithms for turning operations. *Information Sciences*, 210:81–88, 2012.

[142] S. Zhang and Z. He. Implementation of Parallel Genetic Algorithm Based on CUDA. In Z. Cai, Z. Li, Z. Kang, and Y. Liu, editors, *Advances in Computation and Intelligence*, volume 5821 of *Lecture Notes in Computer Science*, pages 24–30. Springer Berlin Heidelberg, 2009.

[143] W. Zhu. Massively parallel differential evolution—pattern search optimization with graphics

hardware acceleration: an investigation on bound constrained optimization problems. *Journal of Global Optimization*, 50(3):417–437, 2011.

[144] W. Zhu and Y. Li. Gpu-accelerated differential evolutionary markov chain monte carlo method for multi-objective optimization over continuous space. In *Proceedings of the 2nd workshop on Bio-inspired algorithms for distributed systems*, pages 1–8. ACM, 2010.

Lista de tablas

Lista de Algoritmos

Lista de figuras

Lista de términos