Implementación de Algoritmos Genéticos sobre la plataforma de desarrollo paralelo CUDA

Víctor Manuel Abascal Pelayo
Pablo Feijoo Ugalde

Dirigido por:
Prof. José Ignacio Hidalgo Pérez
Dpto. Arquitectura de Computadores y Automática

Facultad de Informática

Universidad Complutense de Madrid

Asignatura de Sistemas Informáticos. Curso 2008/2009

Proyecto: Implementación de Algoritmos Genéticos sobre la plataforma de desarrollo paralelo CUDA

Dirigido por el profesor D. José Ignacio Hidalgo Pérez

Víctor Manuel Abascal Pelayo Pablo Feijoo Ugalde

Madrid, 1 de Septiembre de 2009.

Palabras Clave

Algoritmo
CUDA
EMMRS
Genético
Paralelo
Schwefel
Viajante de comercio

Key Words

Algorithm
CUDA
EMMRS
Genetic
Parallel
Schwefel
TSP

Resumen

Español

Los **algoritmos genéticos** (**AGs**) son técnicas de búsqueda y optimización inspiradas en la naturaleza que utilizan propiedades como la herencia, mutación, selección y cruce. Una de las cualidades principales de los algoritmos genéticos es su grado de paralelismo implícito, ya que se trabaja con un conjunto de soluciones de forma simultánea. Al igual que en la naturaleza, la evolución de los individuos no depende únicamente de ellos, si no también de la población a la que pertenece.

Por otra parte, actualmente casi todos los computadores personales cuentan con una tarjeta gráfica destinada a la ejecución de aplicaciones gráficas (videojuegos). En la mayoría de las ocasiones estas tarjetas aparecen inactivas y se está desperdiciando su capacidad para realizar cálculos paralelos.

Nuestro trabajo está destinado a utilizar ese hardware desaprovechado para implementar un conjunto de algoritmos que solucionen el problema del Viajante de Comercio y la función de Schwefel. Los principales objetivos de nuestro trabajo son estudiar la sobrecarga de comunicaciones en la comunicación CPU-GPU y evaluar distintos operadores genéticos utilizando las ventajas de programación que proporciona CUDA, el nuevo lenguaje de programación paralelo de Nvidia.

English

The **genetic algorithms (GAs)** are search and improvement technique inspired by evolutionary biology such as inheritance, mutation, selection and crossover. One of the most important features of the genetic algorithms is their high degree to be parallelised, because they use a group of solutions at the same time. As in the natural world, individual's evolution depends not only the character, but also the environment.

Nowadays, almost all the medium PC's have a graphic card used to execute graphic applications (videogames). The most of the time, we are wasting all this capacity of parallel calculation.

Our work is destined to use this wasted hardware to implement GA that solvent the *Traveling Salesman Problem* (**TSP**) and the *Schwefel function* as well. The project's goals are to study the communication overload in the transfers between CPU and GPU and to evaluate different genetic operators using the CUDA's advantages, the new language for parallel programming of NVIDIA

Agradecimientos

Gracias a José Ignacio Hidalgo Pérez por su apoyo, guía y paciencia en la realización de este trabajo.

Índice

1 Introducción

1.1 Objetivos

La optimización de recursos a través de su máxima utilización, es un tema de permanente actualidad, no sólo en el campo de la informática, si no también en cualquier proyecto de ingeniería. Las numerosas líneas de investigación que hay referentes a la máxima utilización de los recursos hardware muestran también su capital importancia. Desde los continuos avances en el campo del diseño de computadores que minimicen los tiempos de espera hasta su utilización masiva en proyectos como SETI@home[1], la obsesión por la máxima **eficiencia** es una constante en el mundo de la computación.

Con la nueva generación de microprocesadores de sobremesa (multinúcleo) estamos en un redescubrimiento de los paradigmas paralelos de programación (directivas y paso de mensajes) y de las tecnologías que introdujeron a principios de los 90, compañías como *MassPar*[2] y *Thinking Machines*[3], y que derivan en el nuevo objetivo de la industria tecnológica: **paralelizar**.

La misma tendencia que ahora están siguiendo los procesadores, ya había sido iniciada anteriormente por las tarjetas gráficas, cuya capacidad de cálculo venía siendo superior en varios órdenes de magnitud a la de los procesadores desde mediados de 2003. Obviamente este tipo de dispositivos tiene limitado su rango de acción. Sin embargo con la reciente introducción de la plataforma de desarrollo **CUDA** de la empresa norteamericana Nvidia[4], se pone a disposición de los usuarios un software para

desarrollar aplicaciones que hagan uso de esa capacidad de cálculo paralelo en todas sus tarjetas de video a partir de la serie GeForce 8[5].

Todas las ideas anteriormente expuestas, son el medio y la motivación sobre los que desarrollar nuestro proyecto, es el esqueleto en el que sustentar los **algoritmos genéticos**. Los algoritmos genéticos son un conjunto de técnicas para afrontar diversos problemas, principalmente de búsquedas, basándose en la teoría de la evolución de las especies en el mundo natural (Charles Darwin) que expone que un individuo evoluciona no sólo por sus propias condiciones, si no también por las interrelaciones con sus semejantes. Los algoritmos genéticos surgieron de manera independiente y casi simultánea (década los 70) a otros dos paradigmas con lo que conforma lo que hoy día conocemos como computación evolutiva. Estos tres pilares son: los algoritmos genéticos (J. H. Holland), la programación evolutiva (L. J. Fogel), y las estrategias de evolución (Rechenberg - Schwefel).

Nuestro proyecto está orientado a la utilización de un recurso hardware como son las tarjetas de video, desaprovechadas la mayor parte del tiempo para realizar los cálculos necesarios de un algoritmo genético y mediante éste encontrar una solución óptima a los problemas planteados.

1.2 Motivación

Nuestra motivación e interés tiene como origen tanto la demanda por la eficiencia de recursos como la utilización del que creemos será el nuevo paradigma de la informática en los años venideros, la paralelización. El paralelismo en términos de multihilo no es una idea nueva. Gracias al concepto de hilo una aplicación puede realizar varias tareas a la vez (en una manera paralela/concurrente). Debido a que los hilos de un mismo proceso comparten los mismos recursos (espacio de memoria, privilegios, archivos

abiertos…) hace que cualquiera de ellos pueda modificarlo (ahí entran los mecanismos de exclusión y control).

Teóricamente, un programa que se ejecuta a través de hilos debe tener un incremento en el rendimiento total (hablando en términos de productividad y tiempo de respuesta), gracias a la introducción del código paralelo. Sin embargo, no se debe menospreciar los costes de comunicaciones que pueden hacer que el rendimiento del código paralelo no sea el esperado.
Con esta manera de desarrollo, el fragmento correspondiente al código paralelo, se subdivide en dos o más partes que son enviadas a ejecutar en sendos procesadores en un momento determinado.

Decidimos tomar esta manera de pensar y la hemos aplicado a los algoritmos genéticos. Los algoritmos genéticos (AGs) trabajan sobre un conjunto de sujetos codificados, que representan posibles soluciones a un problema dado. A cada individuo se le asigna una función de *fitness* que indica cómo de buena es dicha solución, al igual que en el mundo natural se puede hablar de individuos más o menos adaptados a su hábitat.

En cada iteración del AG se crea una nueva generación de individuos en función de sus antecesores, teniendo más probabilidad de *reproducirse* los que se hayan ajustado mejor a la función de ajuste. Estas iteraciones no cesarán hasta que se haya encontrado una solución al problema, o bien se haya alcanzado un máximo de iteraciones prefijado de antemano.

Como podemos observar, los fundamentos referentes a la manera de actuar de un AG no son complicados de entender, pero dejan entrever la enorme cantidad de cálculos necesarios para hacer funcionar uno de ellos, destinado a solventar problemas complejos con una población y unas codificaciones de individuos de tamaños importantes.

Basándonos en estas premisas hemos utilizado un rasgo capital de las tarjetas gráficas que las hace idóneas para ejecutar sobre ellas AGs, su capacidad de cálculo paralelo. Habitualmente, cuando ejecutamos un algoritmo o un programa, éste corre sobre el procesador de la CPU, dejando la tarjeta gráfica o GPU (y con ella su capacidad de calculo) inactiva. Por lo tanto las expectativas a la hora de migrar un tipo de algoritmo que a todas luces es idóneo para este elemento hardware, son muy altas y nos hace pensar en una mejora considerable en comparación del mismo algoritmo ejecutado sobre una CPU, ya sea mononúcleo (en cuyo caso la mejora sería obvia) o multinúcleo, ya que actualmente la informática de consumo del usuario medio proporciona procesadores de 4 núcleos, nada que ver con los 256 *cores* que nos ofrecen las últimas revisiones de las tarjetas graficas.

Por otra parte, en la literatura se pueden encontrar diversas técnicas de AGs para solucionar problemas clásicos. La mayoría de ellas incluye operadores especializados y técnicas de búsqueda local.
Dentro de estas, el AG con EMMRS[6] (*Evolutionary Mapping Method with Replacement and Shift*) ha demostrado ser una buena opción para solucionar problemas continuos. No obstante, aún no ha sido probado con problemas discretos. En este trabajo demostramos que el AG con EMMRS puede funcionar también concretamente para problemas como el del viajante de comercio (TSP). Los resultados experimentales invitan a profundizar en la aplicación de este operador a otros problemas del mismo tipo.

El resto del trabajo esta organizado de la siguiente manera. Empezaremos haciendo una breve descripción de los algoritmos genéticos, en la que relataremos su origen, su ubicación dentro de las técnicas de búsqueda y una descripción de los conceptos más fundamentales que se deben de tener en cuenta a la hora de desarrollarlos.

De ahí pasaremos a describir ampliamente la estructura del algoritmo genético simple mediante la explicación de sus operadores genéticos. En este apartado incluiremos también un resumen de los pros y contras más destacables de los AGs.

En el punto 4 y para dar por finalizadas nuestras explicaciones teóricas referentes a los AGs, trataremos de exponer las peculiaridades del modelo de algoritmo genético que vamos a implementar. En este capítulo trataremos sobre los algoritmos genéticos híbridos (también llamados meméticos) y sus operadores EMMRS y *Hill Climbing*.

Una vez que hemos planteado una de las herramientas que vamos a utilizar para desarrollar nuestro proyecto, llega el turno de mostrar la base sobre la que cimentar el desarrollo de las aplicaciones, CUDA. El apartado 5 se encarga de englobar todo lo relativo a esta tecnología. Desde sus orígenes a la estructura interna de las tarjetas gráficas sobre las que se aplica, pasando por sus modelos de compilación y ejecución. Al final del capítulo hemos creído conveniente incluir un glosario de términos referentes a los hilos y a su gestión por parte de CUDA, ya que estos son los encargados de la paralelización del código.

El apartado 6 nos sirve para realizar una pequeña explicación de los problemas sobre los que hemos decidido tomar datos de eficiencia de nuestros algoritmos y de su implementación. Implementación que queda puesta de manifiesto en el apartado siguiente.

Hemos reservado para el capítulo 8 la tarea de mostrar los resultados obtenidos con nuestro proyecto, de manera que en el incluimos tanto las gráficas de rendimiento logradas como una valoración del trabajo y las futuras líneas de investigación que se podrían continuar a partir de este punto.

Para acabar, hemos incluido los apéndices del código en CUDA de los problemas que hemos desarrollado.

2 Algoritmos Genéticos

2.1 Introducción

Un *algoritmo genético* (**AG**) es una técnica de optimización que se inspira en la evolución biológica, desde el punto de vista de las teorías darwinianas, como estrategia de resolución de problemas. Dado un problema específico a resolver, la entrada del AG es un conjunto de soluciones potenciales a ese problema, codificadas de alguna manera (genes), y una medida llamada función de *fitness* que permite evaluar cómo de buena es cada solución candidata. Esta generación inicial puede estar compuesta por soluciones que ya se sabe que funcionan, usando el AG para que las mejore, o puede ser inicializada aleatoriamente con la misma finalidad, la de la mejora por parte del algoritmo.

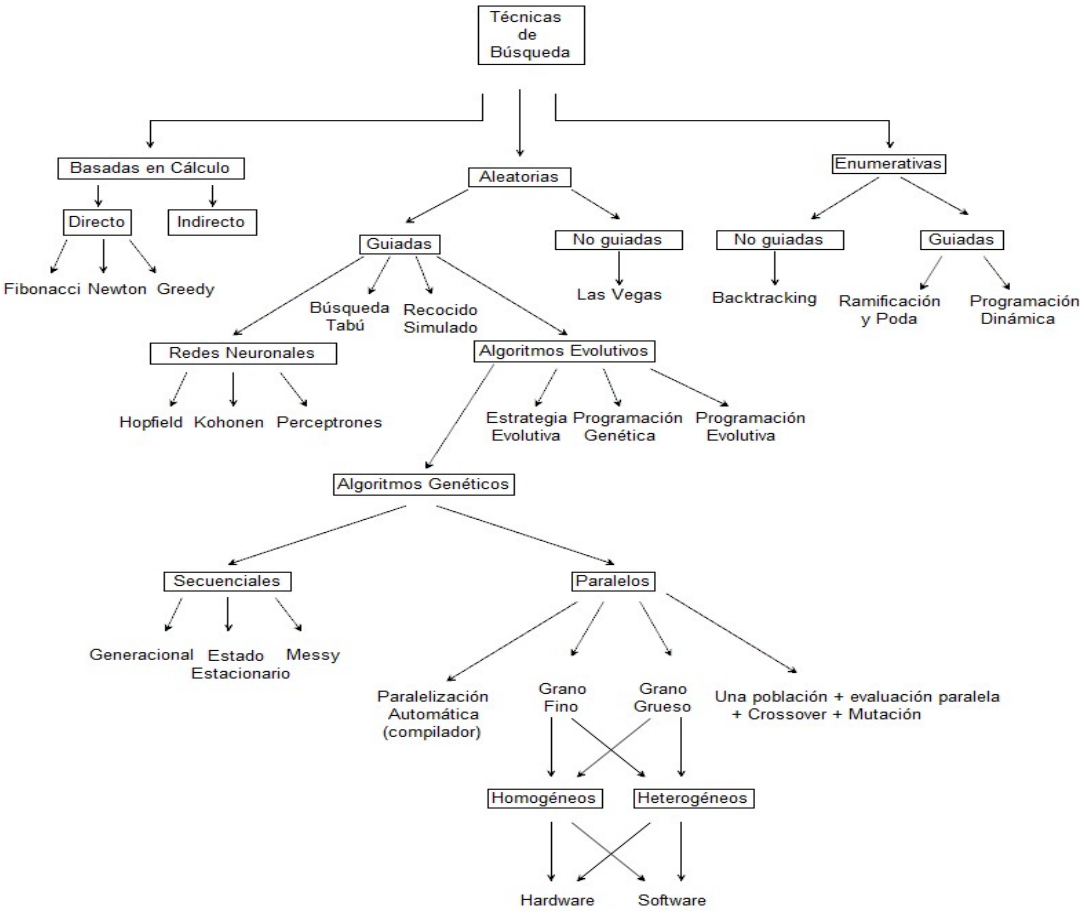


Figura 1.- Diagrama de las principales técnicas de búsqueda

Posteriormente se valora cada posible solución de acuerdo con la función de aptitud. Es fácil comprender que en un conjunto de candidatas generadas aleatoriamente, la mayoría no tendrán unos valores aceptables y por lo tanto serán eliminadas. Sin embargo, algunas de esas posibles soluciones generadas aleatoriamente pueden ser prometedoras, acercarse hacia una solución del problema.

Estas últimas no son suprimidas, y al no serlo, se les otorga más o menos posibilidades de reproducción (en función del fitness mencionado anteriormente). La reproducción en este caso, consiste en realizar cruces entre las diferentes candidatas. Luego, esta descendencia, que puede verse afectada por cambios aleatorios (mutaciones) ya sean beneficiosos o no, pasa a ser la siguiente generación, formando un nuevo *acervo* (herencia genética), y son sometidas de nuevo a la evaluación de la función de aptitud. Las soluciones que hayan empeorado como consecuencia del cruce o de las mutaciones tendrán un fitness bajo y, por tanto, pocas posibilidades de reproducir y transmitir su acervo. Al contrario, aquellas que se hayan visto mejoradas, serán mejores soluciones que las de sus progenitores y se verán premiadas con una mayor probabilidad de reproducción en la nueva generación. En cada iteración el fitness medio de cada generación mejora, por lo tanto iterando el proceso, pueden descubrirse soluciones óptimas para el problema.

Básicamente, esta es la idea original que a finales de los años 60 John Holland (investigador de la Universidad de Michigan) desarrolló bajo el nombre de “planes reproductivos” y teniendo como meta el autoaprendizaje por parte de los ordenadores. Desde entonces hasta ahora, se han utilizado algoritmos genéticos en una amplia variedad de campos para encontrar soluciones a problemas para los cuales no existe una método estándar para buscar esas soluciones. De hecho, aunque haya técnicas establecidas que funcionen bien para el problema en cuestión, puede haber una mejora importante si llegan a combinarse los Algoritmos Genéticos

2.2 Marco de Aplicación

El primer paso, antes siquiera de pensar en el AG, es decidir la codificación de las soluciones. Hay múltiples tipos de representaciones:

1.- **Números reales**, gracias a los cuales la complejidad y la precisión del algoritmo se ven aumentados. Esta codificación de los algoritmos genéticos es la más extendida en el campo de la Inteligencia Artificial para entrenar redes neuronales. Aunque no sólo es la más indicada en ese campo, como demuestra Steffen Schulze-Kremer que utilizó esta codificación para predecir la estructura de una proteína en función de las relaciones de atracción/repulsión de los aminoácidos que la componen[7].

2.- **Caracteres alfanuméricos**, dónde cada una de ellas representa un aspecto de la solución. Podemos tomar como ejemplo el trabajo de Hiroaki Kitano, usada para el perfeccionamiento de una gramática libre de contexto, empleada para crear redes neuronales[7].

3.- **Cadenas Binarias**. Ésta generalmente es la más empleada en la codificación de los individuos de un AG. De hecho fue la propuesta inicial de Holland, es la más empleada debido a su conveniencia para con los ordenadores y a la sencillez de su implementación, aunque bien es cierto que se puede complicar dependiendo de la propia codificación interna de los bits o de si empleamos funciones de mapeado para enmascarar los genes.

Estas son las 3 representaciones más usuales y fáciles, en cuanto a la introducción de operadores de cambios (cambiar letras, operar con una cantidad elegida al azar previamente…), aunque bien es cierto que hay otras representaciones más "creativas". Como ejemplo, mostramos la ideada por John Koza (Universidad de Stanford)[8]. Su

autor la ha bautizado como *programación genética* y representa las posibles soluciones como estructuras arbóreas. Es significativo que la longitud de la codificación no tiene porque ser fija, ya que, en este método los cambios aleatorios en la secuencia de bits son sustituidos por cambios aleatorios en los nodos, pero no solamente mutando un valor o un operador si no también con la posibilidad de sustituir nodos completos por otros subárboles.

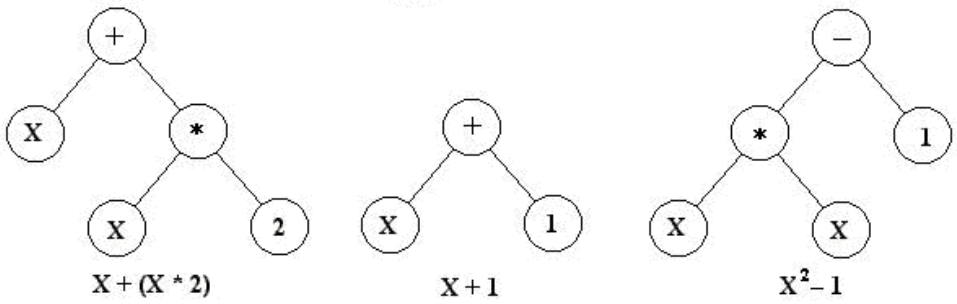


Figura 2.- Modelo desarrollado John Koza.

Otra de las principales premisas en el marco de aplicación de un AG, si queremos que los tiempos de búsqueda sean aceptables, es que el **espacio de búsqueda** debe de ser finito, aunque sea muy grande. No obstante, se puede aplicar a espacios continuos, pero preferentemente cuando exista un rango de soluciones relativamente pequeño.

También es deseable que la función de fitness aplicable no genere errores (como por ejemplo dividir por cero) y sea capaz de penalizar a las malas soluciones, y de premiar a las buenas, de forma que sean estas últimas las que se propaguen con mayor rapidez.

3 El Algoritmo Genético Simple

Una vez que se ha decidido la manera de representar el problema y de codificar las soluciones, es el momento de aplicar el algoritmo genético en sí. La estructura de éste en pseudocódigo se podría escribir como:

BEGIN /* *Algoritmo Genético Simple* */

Generar la población inicial al azar.
Calcular el *fitness* cada uno de los individuos.

WHILE NOT Fin **DO**
BEGIN /* *Crear una nueva generación* */
FOR Tamaño población/2 **DO**
BEGIN /* *Ciclo Reproductivo* */

Seleccionar individuos de la generación anterior para el cruce.
Cruzar con cierta probabilidad los individuos obteniendo sus hijos.
Mutar los descendientes con una probabilidad dada.
Calcular el *fitness* de los nuevos descendientes.
Insertar los nuevos individuos en la población.

END
IF población converge **THEN** Fin := **TRUE**
END
END

3.1 Población inicial

Dos son las características principales que debemos escoger a la hora de tomar una población. La primera de ellas es el tamaño ideal de la misma. Es lógico pensar que con poblaciones pequeñas no llegamos a abarcar todo el espacio de búsqueda del problema y con poblaciones demasiado extensas nos veremos lastrados por un excesivo coste computacional.

El otro factor a tener en cuenta es el modo de generar esa población inicial. La manera más simple es la aleatoria. En ella cada gen puede albergar con la misma probabilidad uno de los posibles valores de la representación escogida (0 ó 1, en el caso binario). Si lo individuos de la población inicial fueran el resultado de aplicar alguna técnica de optimización (heurística), se acelera la convergencia del AG. Sin embargo, esta rapidez de convergencia puede ser contraproducente y llevar al AG hacia los llamados óptimos locales.

3.2 Función de *fitness*

La función de fitness, de ajuste o función objetivo son algunas de las denominaciones que recibe la función que evalúa la bondad de los individuos de la población en relación al problema planteado. Dicha función debe de mantener cierta regularidad en sus valores, esto es, individuos que codifiquen valores próximos en el espacio de búsqueda deben de poseer valores de fitness similares.

Hay casos especiales de individuos cuya función de fitness desprende valores no contemplados por las restricciones de la función de ajuste. Hay varias maneras de tratar estos casos. La primera sería la que podríamos denominar *absolutista*, en la que los

individuos que no cumplen las restricciones, no son considerados como tales. En este caso su función de fitness automáticamente devuelve un valor predeterminado y se siguen operando con ellos de idéntica manera que con los individuos válidos.

Otra solución posible es *rehacer* los individuos que no verifican las restricciones. Este proceso suele llevarse a cabo por medio de un nuevo operador, que se aplica al individuo no válido para obtener uno que si lo es.

Por último cabe destacar otra técnica que se ha venido utilizando si nos encontramos con una función de fitness muy complicada de calcular, es la *evaluación aproximada*. Como una función que es, el cálculo de la función objetivo en ocasiones puede ser demasiado costoso, ya sea en términos de tiempo o de recursos destinados a su cálculo, en estos casos puede ser rentable el cálculo de múltiples funciones más sencillas, de manera que aproximen el resultado de la anterior.

Un problema inherente a la ejecución de un AG, y que más adelante mencionaremos más extensamente, surge debido a la velocidad con la que el algoritmo converge. En algunos casos la búsqueda del óptimo converge de una manera muy rápida hacia un óptimo local, este fenómeno se conoce con el nombre de *convergencia prematura*. La principal causa de la aparición de este problema es debido a que se da una probabilidad mucho mayor de selección a los individuos con un fitness mejor. Por consiguiente si hay un individuo mucho mejor adaptado al resto (pero sin llegar a ser el óptimo global), a medida que ejecutamos el algoritmo se irán generando copias de él que tomen por completo la población. En este caso debemos alterar la función de fitness reduciendo su rango de variación, o de tal manera que individuos que estén muy cercanos entre sí devalúen su función objetivo, con objeto de que la población gane en diversidad.

Un ejemplo de esto último es el que nos proponen Goldberg y Richardson:

Por ejemplo, si denotamos por $d(I_t^j, I_t^i)$ a la distancia de Hamming entre los individuos I_t^j e I_t^i, y por K un parámetro real positivo, podemos definir la siguiente función:

$$h(d(I_t^j, I_t^i)) = \begin{cases} K - d(I_t^j, I_t^i) & \text{si } d(I_t^j, I_t^i) < K, \\ 0 & \text{si } d(I_t^j, I_t^i) \geq K. \end{cases}$$

A continuación para cada individuo I_t^j, definimos $\sigma_j^t = \sum_{i \neq j} h(d(I_t^j, I_t^i))$, valor que utilizaremos para devaluar la función objetivo del individuo en cuestión. Es decir, $g^*(I_t^j) = g(I_t^j)/\sigma_j^t$. De esta manera aquellos individuos que están cercanos entre sí verán devaluada la probabilidad de ser seleccionados como padres, aumentándose la probabilidad de los individuos que se encuentran más aislados.

Figura 3.- Función de fit que penaliza valores próximos.

Por el contrario en otros casos el problema es una velocidad inusitadamente baja, produciendo una *convergencia lenta*. Una posible solución a estos problemas pasa por efectuar transformaciones en la función objetivo, de manera análoga, pero en este caso efectuando una expansión del rango de la función objetivo.

3.3 Métodos de Selección

Las técnicas de selección (que individuos deben utilizarse para producir la siguiente generación) en un algoritmo genético son muchas y algunas de ellas incluso nos dan la opción de utilizarlas simultáneamente en aras de una mayor eficiencia.

Atendiendo al criterio de si se asegura que todos los individuos tienen asignada una probabilidad de selección distinta de cero el método de selección se denomina preservativo o extintivo.

- *Selección elitista*, los individuos más aptos de cada generación son los seleccionados. El más extendido es el elitismo impuro, en el cual se defiende la postura de que solamente el individuo más apto, o algunos de los más aptos, son copiados hacia la siguiente generación.

- *Selección proporcional a la aptitud*, los mejores individuos son los que tienen unas probabilidades mayores de formar parte de la siguiente generación, a diferencia del anterior en el que existía seguridad de su supervivencia.

- *Selección por ruleta*, es una manera de seleccionar candidatos en función de la cómo de idóneo es un individuo en relación a sus competidores. Visualmente podemos emplear el típico gráfico de porciones (similar en aspecto a una ruleta), con la singularidad que los individuos más aptos reciben porciones de mayor tamaño.

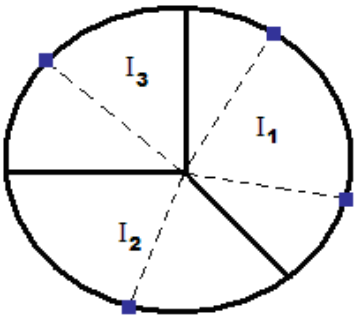


Figura 4.- Selección por ruleta. El individuo I_1 se escoge 2 veces, mientras que I_2 e I_3 una vez cada uno

- *Selección por torneo*, probablemente la más empleada. Consiste en la selección de subconjuntos de individuos de la población, que a su vez se enfrentan entre si, dando lugar a un solo vencedor.

- *Selección escalada*, consiste en definir una función de selección que sea capaz de escoger entre un conjunto de individuos con una adaptación al problema (fitness) muy similar. Esto se produce cuando el AG es ejecutado un número elevado de veces, lo que implica que el fitness medio de la población se ve incrementado y con él la presión selectiva para ser seleccionado. Como ejemplo podemos mencionar la selección de Boltzmann, en la que la selección aumenta durante la ejecución de manera similar a la variable *temperatura* en el recocido simulado[7]

- *Selección por rango*, consiste en ordenar la población en base a la función de ajusta del problema, de tal menara, que el primer elemento será el más apto y el último el menos. La selección pasa a ser entonces en función de este orden y no del fitness en sí, con lo que evitamos en gran parte un estancamiento en un óptimo local debido a la supremacía de los individuos más aptos, preservando la diversidad genética.

- *Selección generacional*, en este caso ninguno de los padres (por muy apto que sea) pasa a la siguiente generación de la población. Ésta está formada íntegramente por los descendientes de la anterior.

- *Selección jerárquica*, este tipo de selección engloba varios mecanismos de criba. Los que serán aplicados en primera instancia, implican muy poco tiempo de evaluación y por tanto no discriminan tanto como los siguientes, que llevan a cabo una selección más rigurosa, a cambio de necesitar más tiempo de cálculo. Aunque pueda resultar difícil de comprender, este método consigue reducir los tiempos de cómputo, ya que los individuos que llegan a los métodos de

selección más elaborados son relativamente pocos, habiéndose filtrado la mayoría en las etapas anteriores (mucho más rápidas).

3.4 Cruce y Mutación

En esta etapa del algoritmo ya tenemos seleccionados los sujetos que forman parte de la nueva población, ahora debemos introducir cambios en estos sujetos mediante dos operadores que, persiguiendo una mejora en la solución, aleatorizan el acervo genético de los individuos. El primero de estos operadores se denomina **cruce** (*crossover*, en inglés), y su finalidad es muy sencilla: recrear la procreación del mundo natural. Esto se consigue mediante la elección de dos individuos para que intercambien fragmentos de su código genético, generando un nuevo material genético que es el que conforma a los sucesores de los anteriores. La forma más habitual de *crossover* implica el cruce en *n* puntos, en el que se seleccionan *n* lugares aleatorios del genoma de los sujetos, y uno de los individuos contribuye con todo su código anterior a ese punto y el otro individuo contribuye a partir de ese punto para producir el nuevo individuo.

Por las investigaciones llevadas a cabo por De Jong, está comprobado que introducir un número de puntos de cruce más allá de dos, es contraproducente al desarrollo del algoritmo. Basa su teoría en el hecho de que introducir puntos de cruce no hace otra cosa que aumentar la frecuencia en el escaneo del espacio de posibles soluciones, y al elevar el número de puntos de cruce lo único que hacemos es desechar individuos que de seguir explorándolos nos llevarían a la solución óptima.

También es usado a menudo el cruzamiento uniforme, en cuyo caso el gen del descendiente en la posición *x* depende del gen en la misma posición de uno de los dos progenitores con una probabilidad *y* (generalmente, 0.5).

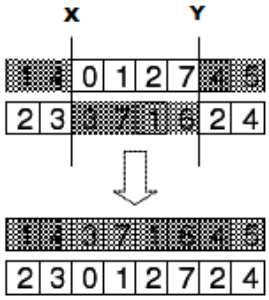


Figura 5.- Crossover en una codificación con números enteros y dos puntos de cruce (X e Y)

Una aplicación de este último caso sería el llamado operador de cruce uniforme con máscara de cruce. En él a parte de un vector (por cada uno de los dos progenitores) que representa la herencia genética de cada uno de ellos, tenemos otro vector de la misma longitud e inicializado aleatoriamente con ceros y unos. Este último vector recibe el nombre de máscara de cruce, y su utilidad es indicar al descendiente de cual de los progenitores debe extraer sus genes para cada posición. Si tenemos un 0 en la máscara de cruce, el gen elegido pertenece al primer padre, mientras que cuando exista un 1 será del segundo, tal y como se muestra en la Figura 5.

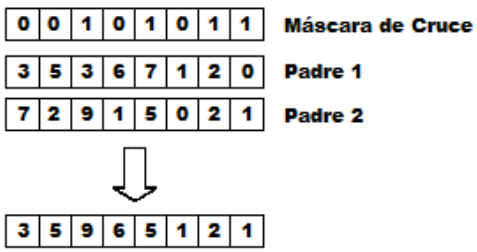


Figura 5.- Crossover uniforme en una codificación con números enteros.

Del mismo modo otros autores han propuesto múltiples técnicas de cruce, como puede ser la aplicación de un operador de cruce a los padres para obtener su descendencia. En el ejemplo de la figura 6 aplicamos la función AND a los dos padres obteniendo como resultado en los genes del hijo un 1 si y sólo si en ambos padres en la misma posición lo hay, y un cero en caso contrario.

Figura 6.- Crossover con operador de cruce AND

El otro operador, usado para modificar la información transportada en los genes de los individuos, es la mutación.

La **mutación**, al igual que en los seres vivos, cambia un gen por otro. En un algoritmo genético, con una probabilidad baja de que suceda se producen cambios aleatorios en los genes, que pueden resultar beneficiosos a la hora de adaptarse al problema o no. La forma de introducir estos cambios es variar genes concretos (y nuevamente escogidos al azar) de un individuo proporcionando un pequeño elemento de aleatoriedad en la población. Dada la frecuencia con que se utilizan, tanto el cruce como la mutación, podemos afirmar que la principal responsable de los cambios que sufre la población en un AG es la primera de ellas, no es menos cierto que la mutación evita en muchos casos el estancamiento en óptimos locales, ampliando el espacio de búsqueda, gracias a sus cambios completamente aleatorios.

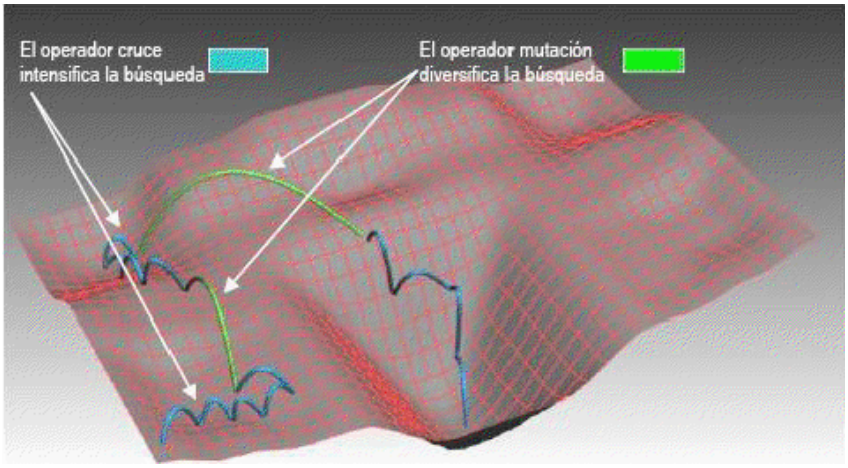


Figura 7.- Variaciones en el espacio de búsqueda, en función de los operadores de cruce y de mutación

En contraposición de la tendencia general de dar más valor al cruce que a la mutación, tal y como sucede en el mundo real, hay ciertos investigadores que consideran mucho más decisiva la mutación. Schaffer por ejemplo, ha desarrollado un AG que implementa la denominada *evolución primitiva*, en la cual el proceso evolutivo omite el cruce y solamente dispone de la selección y la mutación como operadores de cambio. A la vista de estos datos, exponen que es más importante la mutación y todo lo que la rodea (probabilidad de que se efectúe, su utilización en exclusiva junto con selección…) que las mismas características pero relativas al cruce.

En lo referente a la probabilidad con que suceden ambos eventos (cruce y mutación), la idea más natural es proporcionar un alto índice en el cruce y una muy bajo para la mutación. Encontrar unos valores óptimos es una tarea complicada pero que puede reportar a la larga un ahorro considerable en recursos y en tiempo de cómputo. Para

estos valores también hay discrepancia en la comunidad de investigadores. Aunque lo más frecuente es encontrarnos con implementaciones de AG que establecen unos valores constantes para ellos, algunos autores (Bramlette, Janikow…) han obtenido mejores resultados con valores variables. Aumentando los valores de la mutación por ejemplo, en función del número de iteraciones.

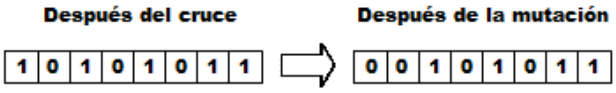


Figura 8.- Mutación de un bit
10101011 => **0**0101011

3.5 Reducción

En este instante el algoritmo genético posee dos poblaciones distintas, por un lado están los progenitores y por el otro sus descendientes, fruto de los dos operadores vistos en el apartado anterior. En este instante es cuando el AG invoca al operador de reducción, que debe escoger entre las dos poblaciones, los individuos encargados de transportar el acervo genético a la siguiente generación. Básicamente tenemos dos líneas principales a la hora de implementar dicho.

Por un lado tenemos la *reducción simple,* que consiste en que todos los individuos descendientes son los que forman parte de la población en la siguiente generación, o bien se escogen de entre los padres y los hijos, los *n* individuos más adaptados al

problema (*reducción elitista de grado n*). Con estas técnicas en mente podemos hacer reducciones mixtas. Por ejemplo, podemos seleccionar los *k* individuos de manera elitista y *n-k* con una reducción simple.

3.6 Ventajas y desventajas de los Algoritmos Genéticos

3.6.1 Puntos Fuertes

El principal atractivo de los algoritmos genéticos es su escalabilidad implícita. Con lo explicado sobre ellos hasta ahora, no es difícil comprender que muchas de las operaciones que se producen no tienen dependencias unas de otras y por lo tanto se pueden ejecutar en paralelo. La mayoría de las técnicas de búsqueda actuales, realizan la exploración del espacio de soluciones en un único camino, con la pérdida de tiempo y recursos que eso supone si resulta que ese camino no conduce al óptimo, y todos los cálculos deben ser deshechos para comenzar desde el principio. En cambio, los algoritmos genéticos trabajan con múltiples soluciones a la vez, representadas en cada uno de los individuos de la población, si resulta que una de esas soluciones no lleva al óptimo, se elimina fácilmente y prosigue la búsqueda con el resto de la población.

Aunque la ventaja del paralelismo en los AG va un paso más allá. Planteemos el siguiente ejemplo: todas las cadenas binarias de 8 dígitos conforman el espacio de búsqueda de este ejemplo (2^8 son 256 posibilidades), que puede representarse como xxxxxxxx (siendo x=0 o x=1).

La cadena 01011010 es un miembro de este espacio. Sin embargo, también es un miembro del espacio 0xxxxxxx, del espacio 0101xxxx, del espacio 0xxx1xx0…

Esto quiere decir que los resultados de las evaluaciones que llevan a cabo un algoritmos genéticos, sobre una muestra del espacio de búsqueda se puede extrapolar a un número mucho mayor de individuos. Es por ello que el AG puede ir encaminandose por el espacio de búsqueda, gracias a los individuos más óptimos, y hallar de este modo el óptimo global. Esta característica recibe el nombre de *teorema del esquema*, y es la principal ventaja de los algoritmos genéticos sobre otros esquemas de búsqueda[9] [10].

Debido a este paralelismo, los algoritmos genéticos obtienen unas tasas de mejora excelentes – en comparación con una búsqueda exhaustiva - tras muestrear directamente sólo regiones pequeñas del total[11]. La mayoría de los problemas que caen en esta categoría se conocen como *no lineales*. Esta diferencia en la mejora se debe a la naturaleza de los problemas afrontados por los AG's, problemas no lineales. Los problemas no líneales, en contraposición a los lineales, son aquellos, en los cuales las aptitudes de los individuos son dependientes entre sí, por lo que no siempre una disminución puntual en el fitness de un individuo es negativa, ya que al estar relacionados entre si, esto hace que se exploren nuevos caminos para encontrar la solución. Esta manera de pensar en la que un cambio (acontecimiento) afecta al devenir del resto es lo más frecuente si pensamos en el mundo real.

Una bondad de los algoritmos genéticos es sin duda la capacidad de adaptación que muestran en problemas cuya función de fitness es bastante compleja (con muchos óptimos locales, cambiante con el tiempo discontinua…). Casi todos los problemas reales tienen un espacio de búsqueda inabarcable para cualquier algoritmo de búsqueda exhaustiva, pero no para los AG's; la dificultad en estos últimos viene en encontrar la manera de no caer en un óptimo local -soluciones que son mejores que todas las próximas a ella, pero que no lo serían si lo comparásemos con otras ubicadas en otra zona del espacio de soluciones. Muchos algoritmos de búsqueda quedan atrapados en los óptimos locales, sin embargo los algoritmos evolutivos, han demostrado su

efectividad al escapar de los óptimos locales y hallar el óptimo global incluso en espacios de soluciones muy complejos.

Koza[8] usa como alegoría para explicar el mecanismo de búsqueda de la solución por parte de un algoritmos genético, una unidad de paracaidistas (individuos) lanzada sobre una cordillera (espacio de búsqueda/soluciones) con el objetivo de encontrar la cima más alta (solución global). Los cambios que se provocan en los individuos con ciertos operadores (por ejemplo, el cruce) permiten a los sujetos buscar cerca de su localización dicho punto, mientras que el algoritmo va guiando a esos "paracaidistas" hacia las zonas más prometedoras.

Mención especial merece el operador de cruce dado que es el encargado de entrelazar los candidatos a solución, consiguiendo así que éstos exploren el espacio de búsqueda conjuntamente, encargándose de que compartan información. El cruce es lo que distingue a los algoritmos genéticos de otros esquemas de búsqueda similares como el Hill Climbing y el recocido simulado, en el que no hay transvase de información entre candidatos. Sin él, cada solución individual va por su cuenta, explorando el espacio de búsqueda sin tener en cuenta al resto de individuos. Un ejemplo de cómo el operador de cruce moldea la herencia genética, eliminando de ella los rasgos que mermaban la aptitud en los padres se recoge en [12], donde se usa un AG para obtener un filtro paso baja partiendo de circuitos padres, en los que había o buena topología o buenos valores (inductancia, capacidad…), pero no los dos a la vez.

Otro rasgo destacable de los algoritmos genéticos es su habilidad para optimizar varios parámetros simultáneamente[11]. Por lo general, los problemas más complicados no son definibles en función de un parámetro si no que son necesarios varios para poder dar una solución válida. De igual forma las relaciones entre estos parámetros suelen ser inversas, es decir, para mejorar uno debemos empeorar otros. Haciendo uso del paralelismo anteriormente mencionado, un algoritmo genético es capaz de encontrar múltiples soluciones para cada parámetro, siendo todas ellas igual de válida[13] . Ante esta

situación se introduce el término de *solución no dominada u óptimo paretiano*[14], que hace referencia a la solución alcanzada por el AG en el momento en que no puede mejorar más un parámetro si no es a costa de empeorar otro.

Para finalizar esta apartado de ventajas haremos uso de la descripción que hace Dawkins[15] de los algoritmos genéticos, "*relojeros ciegos*". Esta expresión hace hincapié en el concepto de que un AG es totalmente independiente del problema que esté tratando en cada momento.

Al no necesitar un conocimiento previo como otros algoritmos (ya que basan toda su potencia de trabajo en la aleatoriedad), los algoritmos genéticos no descartan ningún camino a priori, manteniendo de esta forma la posibilidad de encontrar soluciones novedosas que no se les haya ocurrido a los diseñadores[12].

3.6.2 Limitaciones

Indudablemente los algoritmos genéticos no son la solución a todos nuestros problemas, ya que aparte de las importantes ventajas mencionadas anteriormente, cuenta con inconvenientes significativos que debemos reseñar.

Antes tan siquiera de pensar en el algoritmo genético en sí, debemos plantearnos la representación por la que vamos a optar para dar forma al espacio de búsqueda. Esta representación debe de ser lo suficientemente sólida como para aceptar cambios aleatorios sin llevar a producir errores ni individuos sin sentido.

La función de ajuste es otro gran reto a la hora de implantar un AG. Ésta debe cumplir los requisitos que hagan que realmente encuentre la mejor solución sin llegar a recortar la aptitud de ninguna solución. Además de esto, también los parámetros del algoritmo - tamaño de la población, tipo de selección, probabilidad de mutación y cruce…- deben ser elegidos con cautela. El tamaño de la población incide directamente sobre la cantidad de soluciones que el algoritmo explore, si este es bajo el algoritmo no trabajará lo suficiente y probablemente no encuentre soluciones muy buenas, en cambio si es muy gran deberemos sacrificar tiempo de cálculo. Si los operadores de cambio genético operan demasiado, puede dejar de buscar cerca de soluciones prometedoras ya que no convergería lo suficientemente rápido para el ritmo de cambio impuesto.

Un problema con el que los algoritmos genéticos tienen dificultades son los **problemas deceptivos**[7], en los que la información que proporcionan los elementos del espacio de búsqueda adyacentes es engañosa, a la hora de encontrar el óptimo global. Por ejemplo: imaginemos un problema en el que el espacio de búsqueda esté compuesto por todas cadenas binarias de cuatro cifras, y en el que la aptitud de cada individuo semejante al número de ceros en él - es decir, 0111 es menos apto que 0001 -, excepto en el caso de la cadena 1111 que tiene un fitness muy alto.

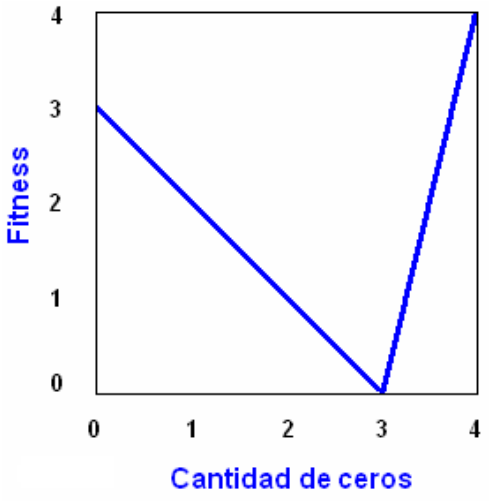


Figura 9.- Ejemplo de función deceptiva.

Otro de los problemas, del que ya hemos hecho mención anteriormente, es el de la convergencia prematura. Si en las primeras etapas del algoritmo la población dispone de un individuo netamente superior al resto (con un fitness mucho mejor), éste se reproducirá rápidamente haciendo que el AG converja, limitando la diversidad y en consecuencia las posibles soluciones[7] [11]. Las poblaciones con un bajo número de individuos son las más proclives a este tipo de problemas, ya que la más mínima variación provoca que un individuo llegue a dominar al resto.

Esta desventaja no es propia del algoritmo si no del modelo en el que se inspira, puesto que en la naturaleza también ocurre (*deriva genética*). Es cierto que es muy poco frecuente debido a que los cambios en la naturaleza, proporcionan pequeños beneficios en su función de ajuste, en ningún caso suficiente como para otorgar una ventaja tan grande que lleve a la supremacía total.

El abanico de soluciones que los investigadores han encontrado al problema de la convergencia prematura pasan por restringir la selección, para no dar tanta ventaja a los individuos excesivamente aptos. La selección *escalada*, *por rango* y *por torneo*, son tres de los métodos fundamentales para lograrlo.

Para finalizar mencionaremos las conclusiones de varios autores[10] [11] [16]ue recomiendan no usar AG's para aquellos problemas cuya solución pueda ser hallada de una manera analítica, ya que será más rápida, barata (menor consumo de recursos) y será matemáticamente demostrable que es la mejor solución.

4 Algoritmos Genéticos Híbridos o Meméticos

Una vez vista la estructura base de un algoritmo genético simple vamos a pasar a describir el tipo de AG que hemos decidido emplear en el proyecto. Buscando unos mejores resultados nos hemos decantado por implementar un AG Memético, que une a las bondades del AG simple los beneficios de una búsqueda local. La información de la búsqueda local, hace referencia a una técnica que más adelante explicaremos, el *Hill Climbing.*

4.1 EMMRS (*Evolutionary Mapping Method with Replacement and Shift)*

EMMRS[6] es un operador genético que aúna en un solo paso las técnicas del crossover y de la mutación. Está basado en el operador genético introducido por Chow, éste establece que la reproducción entre cada par de individuos se realiza (si procede), generando dos nuevos hijos con la información genética de los padres, enmascarada a través de un cromosoma, por cada individuo, destinado a tal efecto. Dicha información se recombina a través de dos puntos de cruce (elegidos al azar en cada iteración) e introduce una mutación aleatoria en uno de los genes. Al EMM se le han añadido dos pasos nuevos, *Replacement* y *Shift*. *Replacement* reemplaza el valor del segundo punto de cruce por el del primero y *shift* desplaza todos los valores entre ambos puntos de cruce una posición a la derecha.

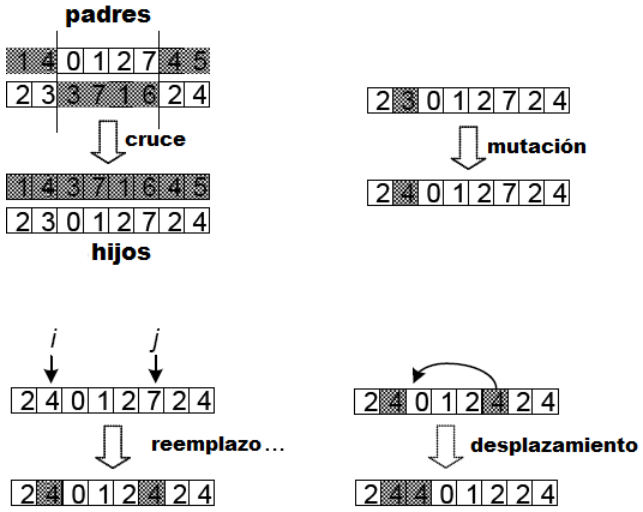


Figura 10.- Operadores genéticos del EMMRS

4.2 Hill Climbing

Es una técnica de búsqueda que atendiendo a la forma en que trabaja se podría decir que es similar a los algoritmos genéticos, aunque más metódico y menos aleatorio. Un algoritmo de HC es iniciado generalmente con una semilla aleatoria de individuos. Una vez hecho esto, el conjunto de cálculos va destinado a mutar esos individuos, y evaluando posteriormente dicha mutación, actuará en consecuencia. Si ese cambio ha implicado una mejora en su aptitud el nuevo individuo sobrevive, por el contrario si ha empeorado, se mantiene el sujeto inicial. Este algoritmo se reitera un número de veces fijo o bien hasta que no sea capaz de encontrar una mutación que mejore la población[8].

A diferencia de los algoritmos genéticos, en el Hill Climbing no esta contemplada la posibilidad de que los individuos puedan empeorar para posteriormente mejorar, una vez dado un punto en el espacio de soluciones solamente puede ir "hacia arriba"

El algoritmo de HC entra dentro de la categoría de algoritmos voraces, lo que muestra en cada paso, escogiendo siempre la mejor elección posible en cada uno, con la intención de obtener el óptimo global.

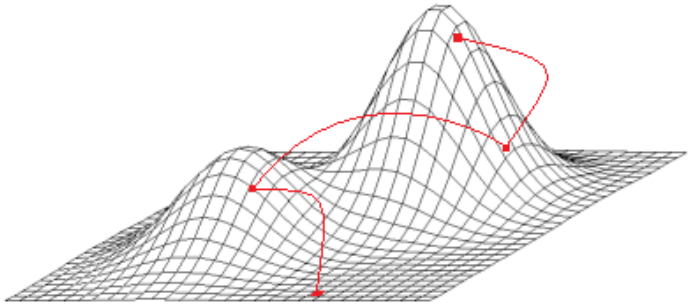


Figura 11.- Vista 3D del espacio de soluciones de un problema implementado con HC.

5 Compute Unified Device Architecture

Compute Unified Device Architecture, comúnmente conocida por su acrónimo **CUDA**, es una plataforma software para facilitar el cálculo paralelo desarrollada por *Nvidia* para sus tarjetas gráficas (*Graphics Process Units*, **GPU**), a partir de la serie 8 (en la gama GeForce). Presentado formalmente en 2006, tras una larga gestación de más de un año en modo beta, CUDA esta ganando clientes sin parar en el ámbito científico. Al mismo tiempo, Nvidia esta rediseñando el concepto de GPU y llevándolo más allá del de un simple dispositivo para juegos o gráficos 3D.

Figura 12.- *Fanart* de CUDA con el logotipo de Nvidia

5.1 Historia y motivación

A pesar de que las tarjetas gráficas de la marca Nvidia han sido siempre muy bien acogidas dentro de la comunidad de jugadores, el mercado de la aceleración gráfica es demasiado voluble. Cuando *AMD* adquirió *ATI* en el año 2006, Nvidia se quedó como la única compañía independiente dedicada a la venta de GPU's ya que el resto de competidores habían caído por el camino (*Matrox*, *S3 Graphics*, *3dfx*,…). Esta condición de superviviente debería ser envidiable, pero no lo fue tanto cuando se vislumbró que tanto AMD como *Intel* tenían planes de incluir núcleos gráficos en los futuros procesadores. Dichos procesadores, unidos a los ordenadores personales (especialmente los destinados a empresa) que ya incluían una tarjeta gráfica integrada en la placa base, se harían con un porcentaje de mercado que podría llegar a herir sensiblemente a Nvidia (sobre todo si se repetía el caso de los coprocesadores matemáticos, cuando fueron incluidos nativamente en los actuales microprocesadores)

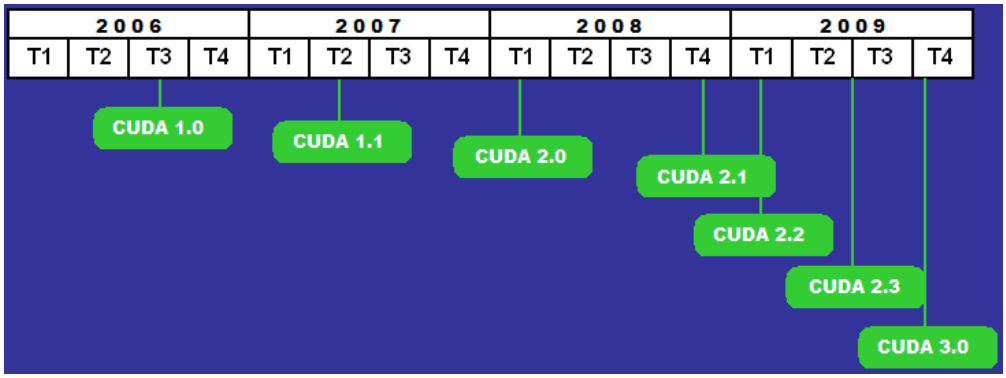


Figura 13.- Fechas de actualización (en trimestres) del SDK de CUDA

El cálculo paralelo en tarjetas gráficas no era algo nuevo para la compañía ya que desde principios de esta década, Nvidia venía desarrollando *Cg*. En lugar de escribir el código en ensamblador se usa un lenguaje de más alto nivel similar al C, pero a diferencia de CUDA, Cg se dirige específicamente a la generación de gráficos. A medida que el número de núcleos de proceso en una GPU se fue incrementando, y ganando en rendimiento al operar con datos en coma flotante, se hizo cada vez más patente la necesidad de aprovechar esa potencia bruta de cálculo para propósitos alternativos, aparte de la evidente aplicación en videojuegos de última generación. Solamente se precisaban herramientas de trabajo de corte más general, con un espectro de aplicación más amplio.

De esa necesidad de diversificación y de la necesidad de aprovechar esa capacidad de cálculo descubierta con Cg, surgió CUDA.

5.2 GPU vs. CPU

Como se puede observar en los sucesivos gráficos que muestran la evolución en los últimos años de las CPU's frente a las GPU's. Unos datos un poco más actuales, nos muestran que mientras un procesador de un ordenador de gama media (*Core 2 Duo*) puede rondar los 10-15 GigaFlops, la capacidad de una tarjeta del mismo rango (*GeForce 9800*) puede alcanzar unos 420 GigaFlops.

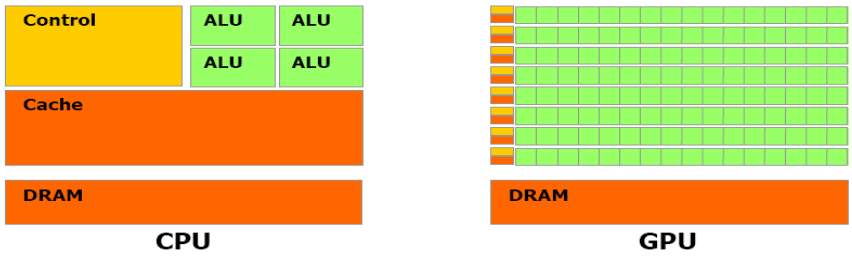


Figura 14.- Las diferencias estructurales entre CPU y GPU unido a una organización más eficiente de la memoria en estas últimas, explican las diferencias de capacidad bruta de cálculo

Con las ultimas actualizaciones, tanto de gráficas de cómo de procesadores, el valor pico de unas ronda los 500 GigaFlops y el de los otros no ha aumentado más haya de los 20 GigaFlops.

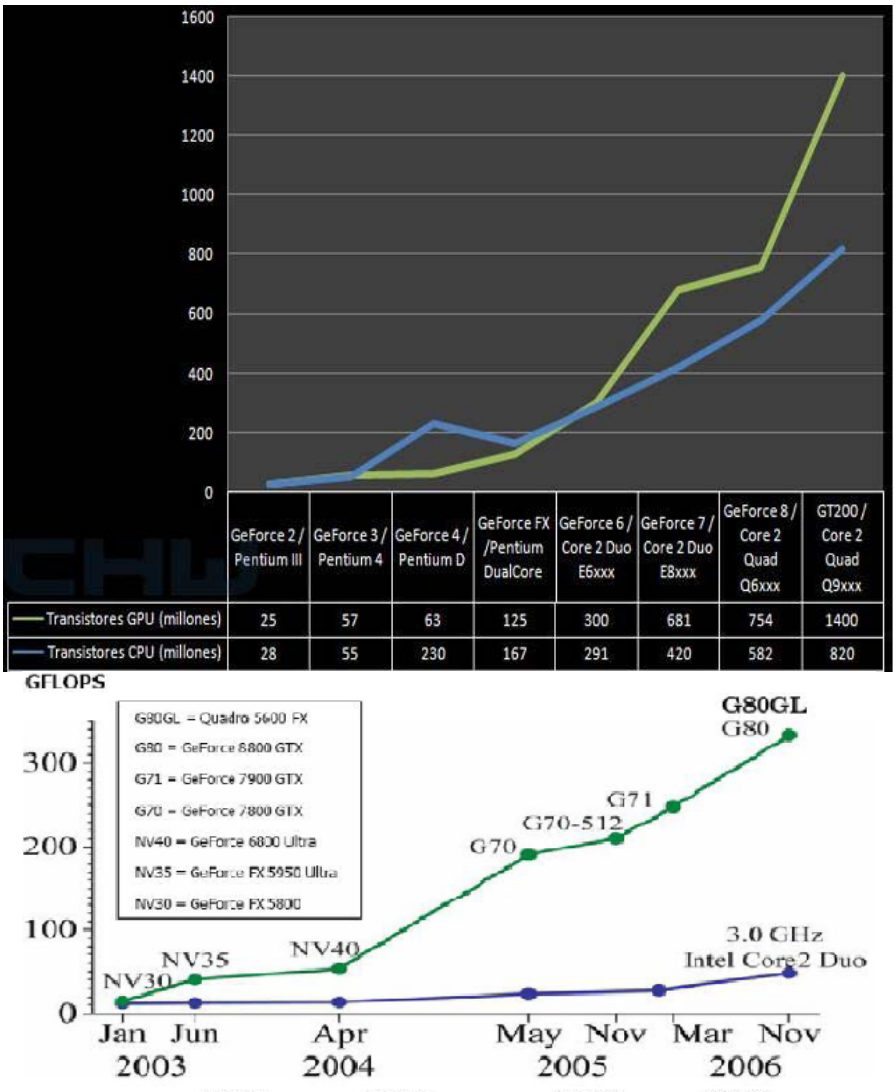


Figura 15.- Comparativa de la evolución CPU-GPU en función del número de transistores (página anterior) y de operaciones en punto flotante.

5.3 Características principales.

5.3.1 Abstracción

Una de las principales característica de CUDA es la **abstracción**. Nvidia se ha caracterizado por dar un trato de caja negra a todas sus arquitecturas, y es que siempre la ha ocultado bajo una *interface* usada para programar las aplicaciones (*API*). Como resultado de esto, los programadores nunca escribían código directamente sobre la tarjeta, en lugar de eso, usaban llamadas a funciones gráficas incluidas en esa API.

La abstracción del hardware tiene dos beneficios. Por un lado, simplifica el modelo de programación de alto nivel, aislando a los programadores ajenos a Nvidia de los complejos detalles de hardware. Y por otro esta abstracción permite cambiar la arquitectura interna de las GPU's cuando se quiera. NVIDIA puede rediseñar completamente la arquitectura para la nueva generación de tarjetas gráficas (cosa que sucede aproximadamente cada 18 meses) sin tener que tocar para nada la API ni provocar fallos en el software ya creado.

Gracias a la abstracción de la capa hardware, Nvidia es libre de diseñar GPU's con cualquier número de *cores*, *threads*, registros, incluso con diferente repertorio de instrucciones. El código escrito será reutilizable en la nueva tarjeta. Aunque probablemente no exploten todo el potencial de la nueva tarjeta, si que estará disponible esa "*retrocompatibilidad*".

En contraposición con esto, para el resto de compañías las arquitecturas son inamovibles. Dichas arquitecturas no pasan de meras extensiones de una generación a otra, como Intel hizo con *MMX* primero y con *SSE* después, porque borrando una

simple instrucción o alterando la visibilidad de un registro haría que mucho software escrito no funcionará.

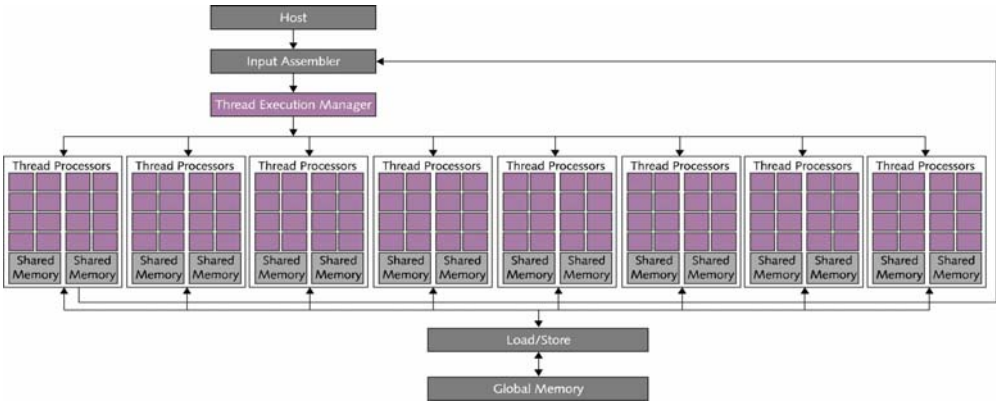


Figura 16.- Arquitectura de una GeForce de la serie 8. Esta en particular tiene 128 thread processors. Cada uno de ellos con una unidad de punto flotante y 1.024 registros con una longitud de palabra de 32 bits. Cada cluster de 8 thread processors tiene 16KB de memoria local compartida con soporte para accesos paralelos. El máximo de hilos concurrentes es 12,288.

5.3.2 Gestión de los hilos.

El lenguaje de programación que impera es básicamente C (aunque existen *wrappers* para Python, Fortran y Java) con algunas variaciones propias para poder codificar los algoritmos en las GPU's. A pesar de esta similitud, el modelo de programación en

CUDA difiere significativamente del código para una CPU monoprocesador. En una CPU de estas características se trae una instrucción a ejecución que opera con los datos. Un superescalar es capaz de dirigir el flujo de instrucciones a través de múltiples *pipelines* pero aún así, hay un solo flujo de instrucciones y el nivel de paralelismo se ve seriamente afectado por las dependencias de datos y de recursos. Incluso el mejor superescalar de 4, 5 ó 6 vías no logra resultados más allá de las 1,5 instrucciones por ciclo. La extensión *Single-instruction Multiple-data* (**SIMD**) permite a varias CPU's extraer paralelismo a nivel de datos del código, pero en la práctica está limitada a 3 ó 4 operaciones por ciclo.

Otro modelo de programación es **GPGPU** (*general-purpose GPU*). Este modelo es relativamente nuevo y ha ganado muchos adeptos en los últimos años. Se basa en el hecho de que los desarrolladores usan las GPU's como procesadores de propósito general, entendiendo "propósito general" como aplicaciones intensivas en datos, usadas con fines científico-técnicos. Los programadores usan los *pixel shaders* de las GPU's como si se tratasen de unidades de punto flotante. El modelo de programación GPGPU es altamente escalable, pero depende en gran parte de la memoria de video (ajena a la tarjeta) para grupos de datos grandes. Esta memoria, usada para mapas de texturas en aplicaciones gráficas, puede almacenar cualquier tipo de dato en este modelo de programación. Diferentes hilos deben interactuar entre sí fuera de la memoria del chip, lo que limita el rendimiento considerablemente.

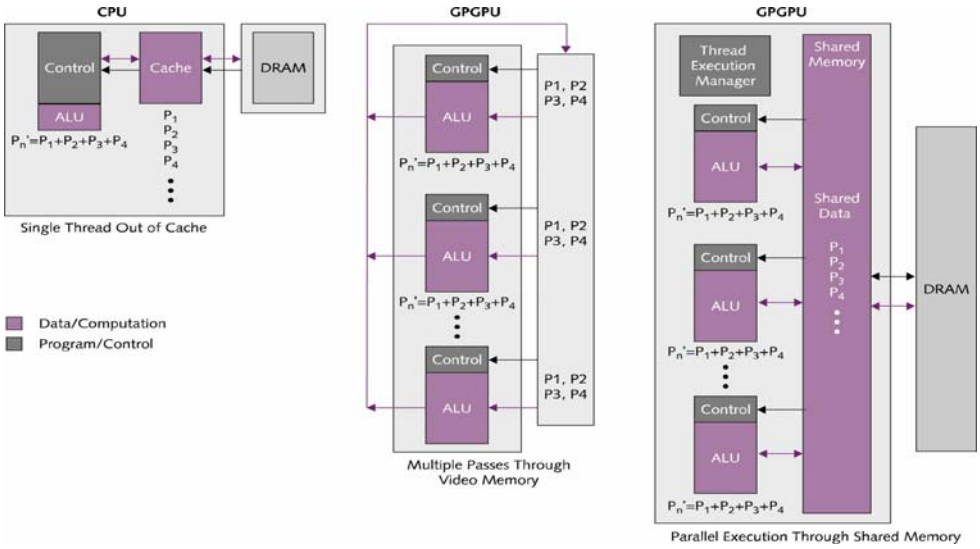


Figura 17.- Comparativa de arquitecturas (de izq. a dcha.), CPU monoprocesador, GPGPU con memoria de video y GPGPU con memoria compartida.

Al igual que en el modelo GPGPU, CUDA también es altamente escalable. Divide el conjunto de datos en pequeños trozos almacenados en la memoria de la tarjeta, lo que permite que los hilos compartan esa información. Almacenando los datos localmente se reduce la necesidad de acceder a memoria externa al chip mejorando el rendimiento.

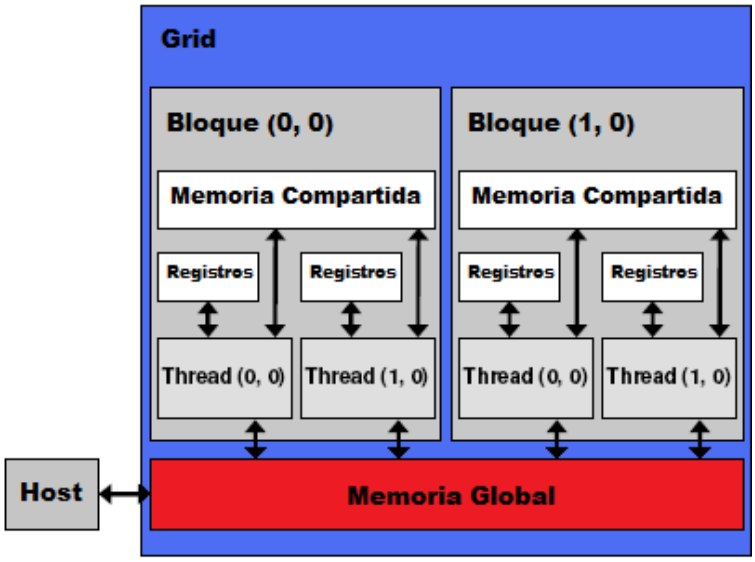


Figura 18.- Esquema de la colaboración a nivel de hilos en la arquitectura CUDA

Por supuesto que de vez en cuando, un hilo necesita acceder a la memoria externa al chip, como por ejemplo cuando carga datos de memoria principal a la memoria local. En el modelo CUDA, los accesos a la memoria externa no paran un *thread processor*, en lugar de esto, el hilo entra en una cola de procesos inactivos y es sustituido por uno que este listo para ejecutarse. Cuando los datos del hilo detenido están disponibles, los hilos pasan a una cola de procesos *listos*. Los hilos se ejecutan con una política round-robin.

Una característica importante de CUDA es que los programadores de aplicaciones no escriben código explícito para el tratamiento de hilos. Un gestor hardware se encarga de administrar automáticamente los hilos. Esta gestión automática de hilos es esencial en un sistema de programación multihilo que maneja miles de hilos.

Sin tener que cargar con el trabajo de la gestión de los hilos, Nvidia simplifica el modelo de programación y elimina un campo enorme de errores a la hora de programar. En teoría, CUDA hace imposible la inanición entre hilos. Lo cual ocurre cuando dos o más hilos se bloquean mutuamente por el intento de acceder al mismo dato, creando un callejón sin salida en el que ninguno de ellos puede continuar. El peligro de esta exclusión mutua es que puede pasar desapercibida.

5.4 Modelos de compilación y ejecución

5.4.1 Compilación

Compilar un programa en CUDA no es tan sencillo como ejecutar un compilador de C para convertir el código en un ejecutable. Hay pasos adicionales que deben hacerse, debido a que el programa esta dirigido a dos arquitecturas diferentes (CPU / GPU) y a la abstracción implícita de CUDA.

En cualquier ejemplo de código escrito en CUDA, en el mismo archivo se mezcla código de C destinado a la CPU y a la GPU (cuyo código es identificado por las extensiones especiales). Lo primero que se debe de hacer es separar el código en función de a que arquitectura vaya destinado.

Para este propósito, Nvidia proporciona un preprocesador de C/C++ realizado por el *Edison Design Group* (**EDG**), el cual analiza sintácticamente el código fuente y crea diferentes archivos para cada arquitectura.

Para la CPU, EDG crea archivos fuente estándar .cpp, listos para poder ser usados por un compilador de C++ ya sea de Microsoft o GNU (Nvidia no lo proporciona). Para la GPU, EDG crea un conjunto de archivos, también con extensión .cpp, compilados usando **Open64** (un compilador de código abierto diseñado originalmente para la arquitectura *Itanium* de Intel).

Las razones esgrimidas desde la compañía para justificar la elección de Open64 frente a otros compiladores son que, a parte de ser un buen compilador para la programación paralela, su personalización es muy intuitiva (obviando el hecho de que sea código abierto, y por tanto gratuito). La implementación que Nvidia hace de Open64 genera archivos .ptx (*Parallel Thread eXecution*) que no son más que código en lenguaje ensamblador.

En las figuras posteriores mostramos el proceso de compilación de un programa escrito para CUDA. A pesar de que es un poco más complicado que el tradicional paso de fichero fuente a fichero binario, tiene varias ventajas.

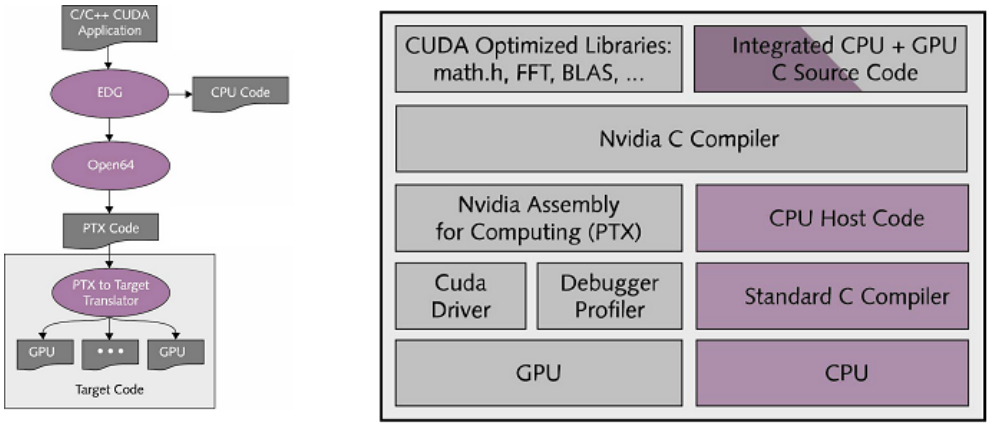


Figura 19.- Proceso de compilación de un archivo .cuda (izquierda) y sus diferentes capas de abstracción (derecha)

1. Los programadores pueden mezclar en el mismo archivo código destinado tanto a la CPU como a la GPU, en lugar de escribirlo en archivos separados.

2. El proceso de compilación de CUDA usa diferentes herramientas para compilar el código de la CPU o de la GPU, cada una optimizada para la arquitectura a la que esté destinada. En lugar de forzar a un único compilador a hacer el trabajo de los anteriores, con lo que al ser menos específico para cada paso, será menos eficiente.

3. No es necesario reescribir el código para una futuras arquitectura (recompilar los archivos .ptx es lo único que sería necesario).

Aunque recompilar los archivos .ptx no es estrictamente necesario para la retrocompatibilidad, si que es aconsejable para aumentar el rendimiento. Como hemos mencionado con anterioridad, el código fuente de CUDA es independiente de la arquitectura de la GPU. Gracias a la abstracción del hardware y al traductor PTX, el mismo código puede funcionar en futuras tarjetas de Nvidia. Sin embargo, cuando los desarrolladores analicen sus datos también tendrán que tener en cuenta que una futura tarjeta gráfica presumiblemente tendrá más *thread processors*, más memoria y otras mejoras, que quedarán desaprovechadas si no se reescribe o al menos se recompila el código.

Por lo tanto, el mismo código fuente funcionará en una futura GPU y su rendimiento se escalará en función del número de *thread processors*. Pero con un conjunto menor de modificaciones en el código probablemente el rendimiento mejore.

5.4.2 Ejecución

La parte que ejecuta un programa en CUDA sobre la GPU se inicia, realizando una copia de los datos de memoria principal a la memoria del dispositivo, para posteriormente ceder la ejecución de la CPU a los *thread processor* de la tarjeta. Posteriormente, el *kernel* se ejecuta sobre una configuración *grid* que gracias a las características vistas anteriormente (multihilo, memoria compartida…) logra acelerar la ejecución. Por último, la GPU copia desde su memoria local a la principal los resultados obtenidos en el proceso y devuelve la ejecución a la CPU, dando así por finalizada la utilización de la tarjeta.

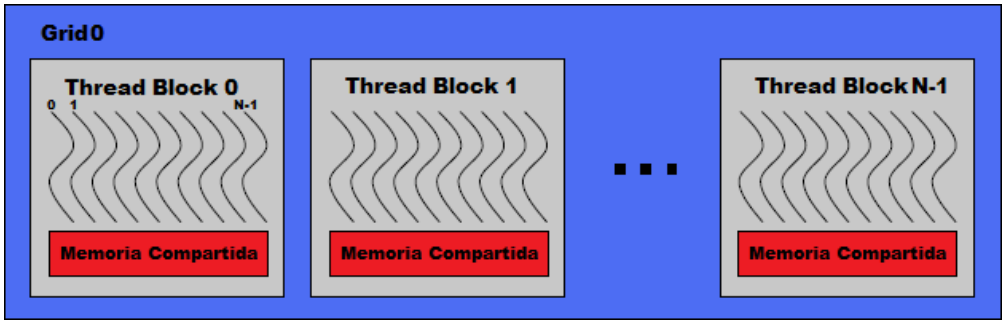


Figura 20.- Cooperación a nivel de hilos.

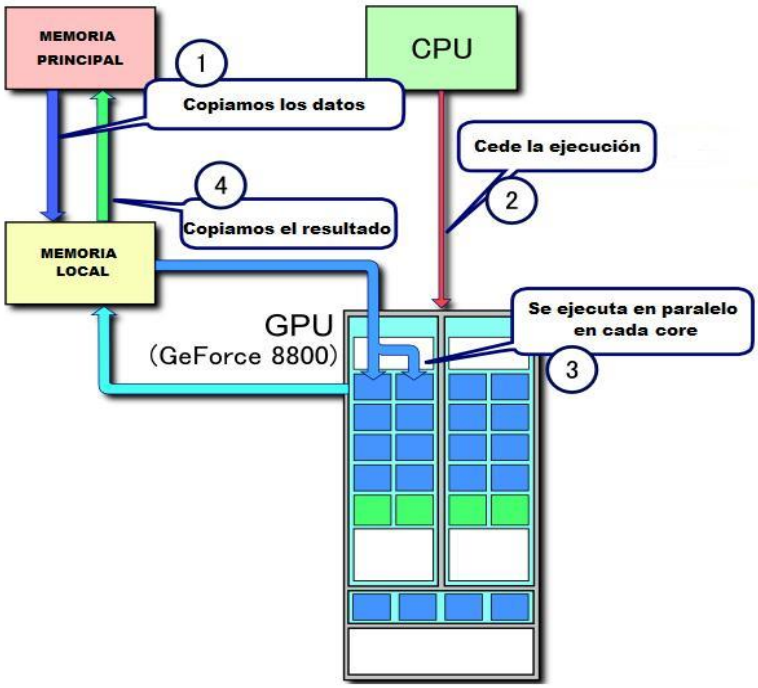


Figura 21.- Pasos en la ejecución de un programa escrito en CUDA

5.5 GLOSARIO

Bloques debido a que el número de hilos del *warp* quizás no sea el más óptimo, los bloques pueden albergar un número que va desde 64 a 512.

Grids, albergan los bloques. La ventaja de operar de esta manera está en que el número de bloques procesados simultáneamente está ligado muy de cerca a los recursos hardware (multiprocesadores o *stream processors*) de la tarjeta.

Host y Device, hacen referencia a la CPU y a la GPU, respectivamente

Kernel, función/algoritmo que se ejecuta sobre la tarjeta. Solamente se ejecuta una cada vez a través de varios hilos.

Thread/hilo, un hilo de la GPU es el elemento básico de proceso de datos. A diferencia de los hilos de la CPU, los hilos en CUDA son muy "ligeros", lo que significa que un cambio de contexto entre dos hilos no es una operación costosa.

Warp, Un *warp* en CUDA, es un grupo de 32 hilos, que es el tamaño mínimo de la los datos tratados por cada multiprocesador en CUDA.

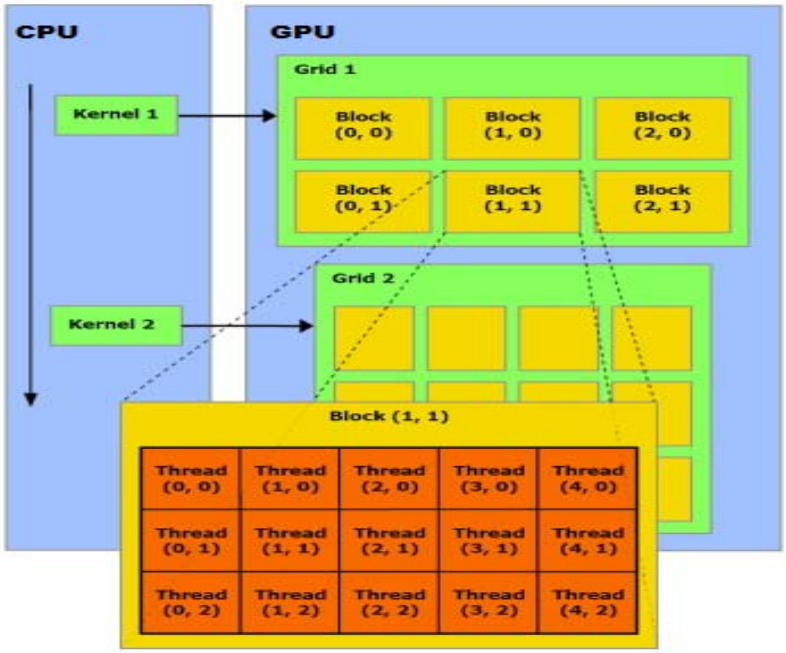


Figura 22.- Muestra los diferentes elementos ubicados en conjunto

6 Problemas implementados

Para probar la capacidad de cómputo de los AGs sobre las tarjetas gráficas hemos decidido usar problemas NP-completos. Los problemas NP-completos son aquellos para los que no se conoce ningún algoritmo eficiente, y para los que tampoco se ha descubierto su complejidad.

- Podrá existir algún algoritmo eficiente para resolverlos que todavía no se ha encontrado.
- O bien estos problemas tienen una complejidad intrínseca alta, pero no disponemos todavía de las técnicas necesarias para demostrarlo.

Decimos que un algoritmo es eficiente si existe un polinomio *p(n)* tal que la complejidad temporal del algoritmo está en O*(p(n))*.

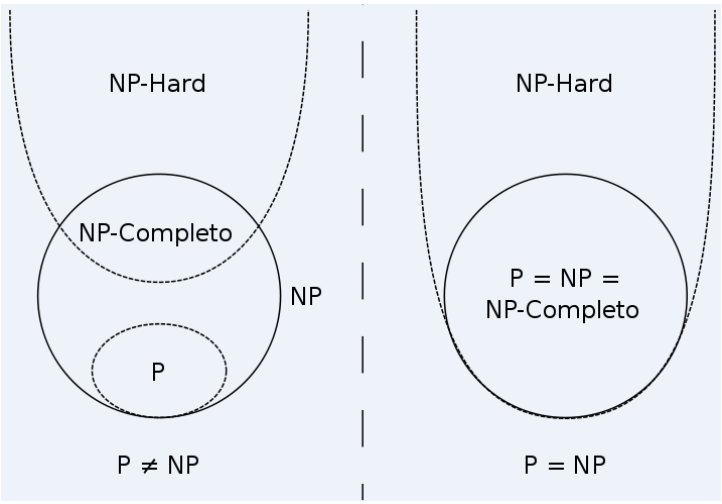


Figura 23.- Diagramas de Venn de las familias de problemas P, NP, NP-completo, y NP-hard

6.1 Problemas continuos. Funciones multimodales.

Son aquellas que no son representables por ninguna expresión analítica (fórmula cerrada). Es por ello que buscan un óptimo (máximo/mínimo) global en una función con muchos óptimos locales. Como ejemplo de función multimodal hemos decidido tomar la función de Schwefel:

$$f(\vec{x}) = \sum_{j=1}^{d} x_j \sin(\sqrt{|\ x_j\ |})$$

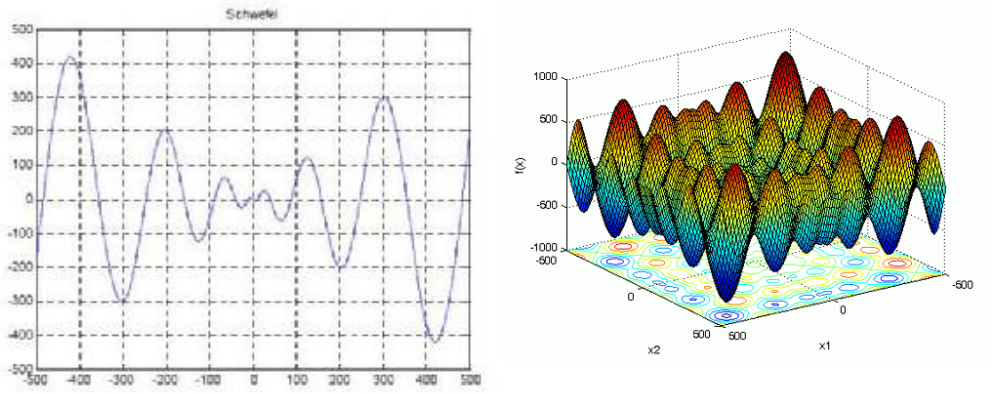


Figura 24.- Soluciones de la función de Schwefel 2D y 3D

6.2 Problemas discretos NP-completos

Para este caso hemos elegido el Problema del Viajante de Comercio (TSP, por sus siglas en inglés). El TSP trata de encontrar un recorrido de valor mínimo por un grafo valorado. Este recorrido debe ser un ciclo hamiltoniano, esto es, debe pasar una sola vez por cada nodo y tener su principio y fin en el mismo nodo. Como caso puntual, incluimos en el Apéndice A (página 110) los resultados obtenido para un caso puntual del TSP, llamado gr48.

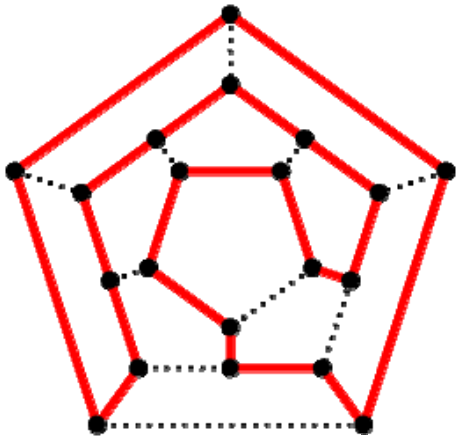


Figura 25.- Ejemplo de ciclo hamiltoniano

7 Implementación y Desarrollo

7.1 Detalles técnicos

El desarrollo de nuestro proyecto se ha fundamentado en los siguientes recursos hardware y software:

Procesador (CPU): *Intel Core2 Duo E6600* (2.4 GHz)
Memoria RAM: 2 GB
Tarjeta gráfica (GPU): *Nvidia 8800 GTX*

Sistema Operativo: *Windows XP Service Pack 3* / *Ubuntu 8.04 LTS*
Entorno de programación: *Notepad++*, *Eclipse 3.5* y *C++ Builder 5.0*
Entorno de red: *WinSCP* y *PuTTY*
Versión: CUDA 2.0

Los recursos hardware hacen referencia al clúster llamado K2, facilitado por el grupo de investigación ArTeCS y ubicado en la Facultad de Ciencias Físicas de nuestra universidad. K2 es un clúster de tipo *Beowulf* formado por 4 nodos de cómputo y un *frontend* de las mismas características. De ahí que todas las pruebas para medir el potencial de CUDA fueran en remoto y, por lo tanto, necesaria la utilización de clientes *SSH* y *Telnet* (nativos, en el caso de Ubuntu).

Figura 26.- Logotipo del grupo ArTeCS

7.1.1 NVIDIA GeForce 8800 GTX

Graphic Process Unit	GeForce 8800 GTX
Número de transistors (millones)	681
Tecnología (nm)	90
Stream Processors	128
Reloj central (MHz)	575
Reloj de las unidades de sombreado (MHz)	1350
Reloj de la memoria (MHz)	900
Cantidad de memoria	768MB
Tipo de memoria	GDDR-3
Interfaz de memoria	384-bit
Ancho de banda de memoria (GB/s)	86.4
Tasa de relleno de texturas (miles de millones/s)	36.8

Figura 27.- Especificaciones del modelo GeForce 8800 GTX

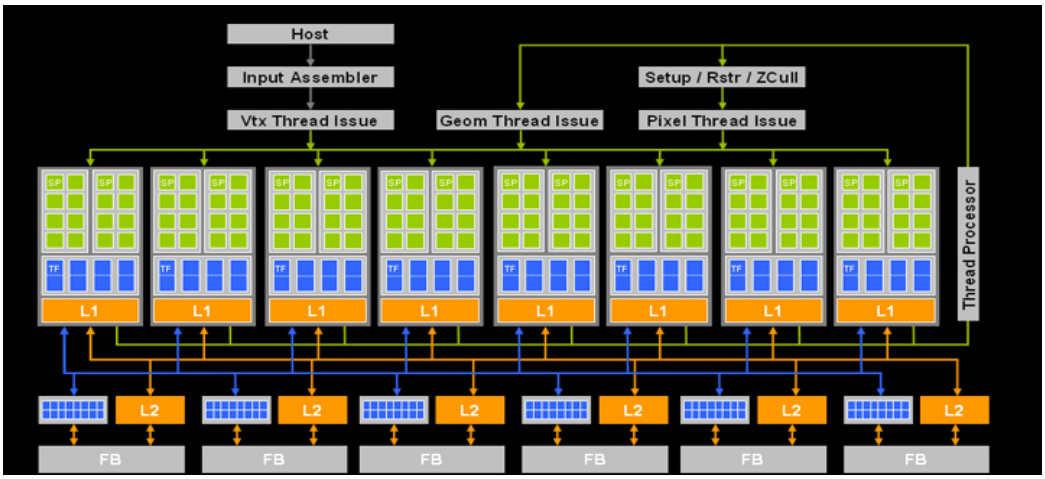


Figura 28.- Diagrama de bloques del modelo GeForce 8800 GTX

7.2 Implementación

En el periodo de desarrollo de nuestro proyecto, se pueden distinguir claramente dos periodos:

1.- Implementación de AG's sobre la CPU.

2.- Implementación de AG's sobre la tarjeta gráfica.

3- Testeo y obtención de resultados.

4- Estudio de los resultados obtenidos.

7.2.1 Representación de los datos, codificación y función de ajuste

La representación de los datos en un problema es fundamental, ya que con una mala representación puede que el coste del problema aumente de forma considerable.
En nuestro caso, para codificar los individuos hemos usado un método en el que el fenotipo de cada individuo queda caracterizado por la aplicación de una función de mapeo a su genotipo, en lugar de haber una correspondencia directa fenotipo-genotipo.
Para cada individuo tenemos dos ristras de números (función de *mapping* y genotipo, respectivamente). Entre las dos son las encargadas de codificar la verdadera información genética que porta el sujeto (fenotipo). Aplicando la función de mapping al genotipo obtendremos dicha información genética.

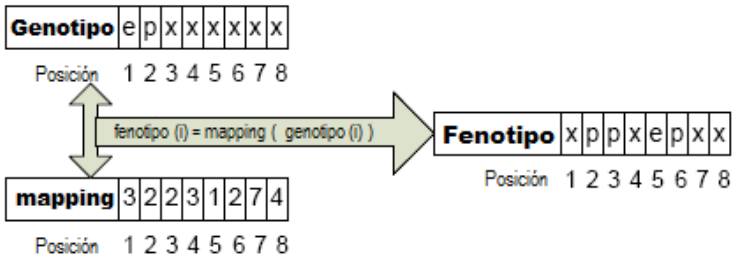


Figura 29.- Obtención del fenotipo de un individuo, a través de una función de mapeo aplicada al genotipo.

Al utilizar esta técnica para la codificación de la información genética de los sujetos, hacemos que los cromosomas no sean tan sensibles a los cambios.

En el caso de Schwefel el genotipo esta codificado de manera que contenga 10 cromosomas diferentes, por lo que en todo el genotipo habrá 10 datos. Estos datos, en nuestro caso, se corresponden con números decimales, los cuales van codificados del siguiente modo:

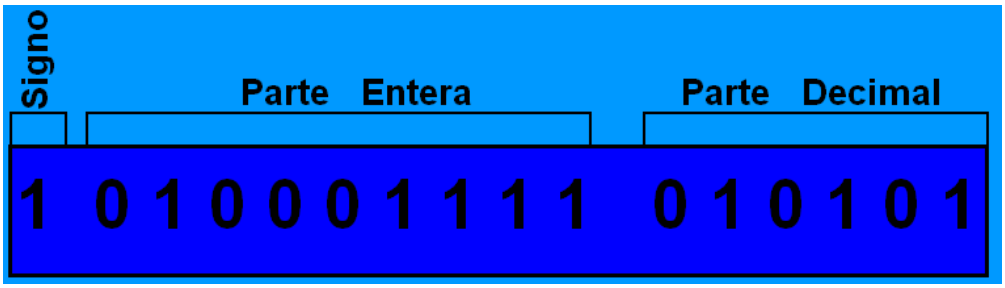


Figura 30.- Codificación en el problema de Schwefel: 1 bit de signo, 9 bits para la parte entera y 6 bits para la parte decimal

En el caso del problema del viajante de comercio el genotipo representa la forma en la que se van visitando las ciudades, esto vendrá condicionado por el vector de mapping.

En cuanto al calculo del fitness que cuantificará la bondad de las soluciones halladas, hemos empleado para el modelo multimodal, obviamente la función que representa (función de Schwefel).

$$f(\vec{x}) = \sum_{j=1}^{d} x_j \sin(\sqrt{|\ x_j\ |})$$

En el caso del problema del viajante de comercio, el cálculo de su función de ajuste se ha basado en realizar la suma de las distancias entre las ciudades recorridas por cada solución, y cuyos valores se encuentran almacenados en una matriz de distancias.

7.2.2 Estructura

Como ya hemos comentado con anterioridad el Algoritmo Genético tiene una estructura general que mantiene el siguiente esquema.

BEGIN

Generar la población inicial al azar.
Calcular el *fitness* cada uno de los individuos.

WHILE NOT Fin **DO**
BEGIN
FOR Tamaño población/2 **DO**
BEGIN /* *Ciclo Reproductivo* */

Seleccionar individuos de la generación anterior para el cruce.
Cruzar con cierta probabilidad los individuos obteniendo sus hijos.
Mutar los descendientes con una probabilidad dada.
Calcular el *fitness* de los nuevos descendientes.
Insertar los nuevos individuos en la población.

END
IF población converge **THEN** Fin := **TRUE**
END
END

Sin embargo, cada AG es diferente y tiene sus peculiaridades, como ocurre con los que hemos implementado nosotros para la función de Schwefel y para el TSP.

Con formato: Título 3 Car

Para Schwefel el algoritmo lo hemos estructurado de la siguiente manera:

- Selección
- Cruce
- Mutación
- Cálculo del fit
- Inserción

En el caso de TSP, nosotros lo hemos estructurado del siguiente modo:

Con formato: Interlineado: 1,5 líneas

- Selección
- Cruce
- Mutación
- Cálculo del fit
- Eliminación de duplicados
- Inserción

7.2.3 Operadores Evolutivos

OPERADOR DE SELECCIÓN

La selección que llevamos a cabo para elegir los individuos de la siguiente generación es una selección mixta: garantizamos la supervivencia de los dos individuos más aptos (preservativa) y el resto de "padres" son sustituidos por sus hijos (extintiva). Posteriormente de esta generación, seleccionamos una muestra de la misma cantidad de sujetos (*sampling*) entre los ya existentes mediante torneo. El torneo va a favorecer el

elitismo, ya que mediante la comparación de la función de aptitud (*fitness*) entre los individuos, vamos a pasar a la siguiente generación siempre el mejor.

Al pasar los dos mejores individuos directamente a la siguiente generación nos aseguramos tener el mejor, al menos de la generación anterior.

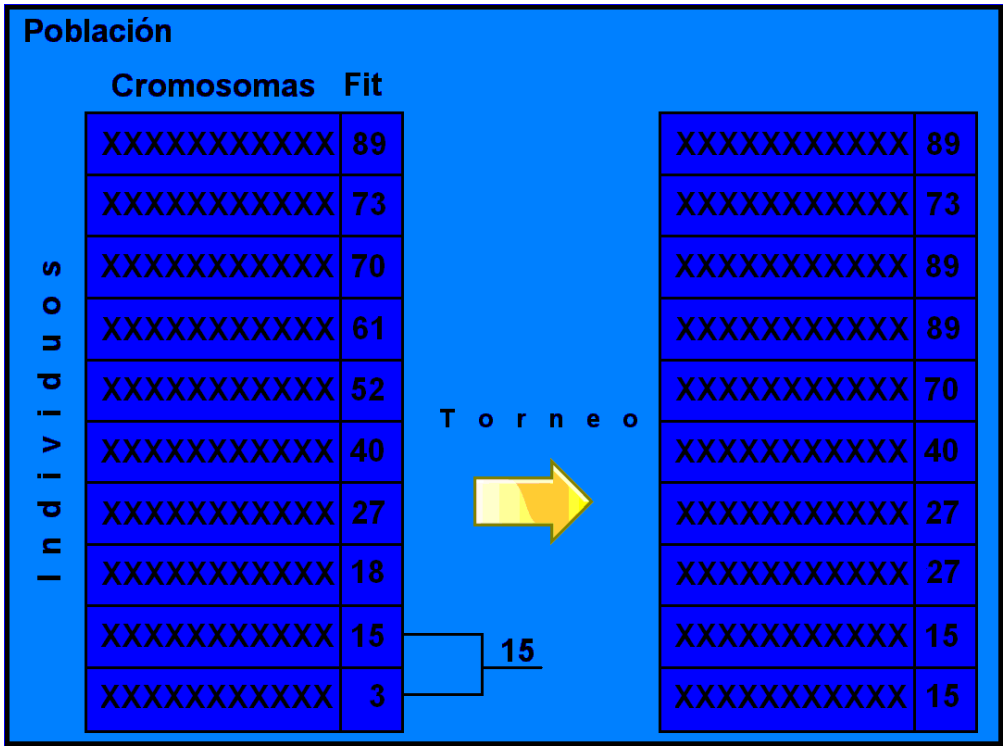


Figura 31.- Metodología de trabajo del operador sampling

OPERADOR DE EVALUACIÓN

En la evaluación, recorremos todos los individuos calculando el valor de su aptitud con respecto a la función de *fitness* de cada problema. El operador de evaluación en el caso de la función de Schwefel, únicamente la evaluación del valor del fenotipo en dicha función.

En el caso del TSP, la función de fit debe acceder a una matriz auxiliar (que alberga las distancias entre los nodos) y a partir de ahí, ir sumando las distancias entre ciudades.

HILL CLIMBING: PROBLEMAS

Después de una primera aproximación a la solución, vimos que nuestro código era ineficiente y tras muchas pasadas el AG no era capaz de encontrar soluciones óptimas, quedando atrapado en la llamada **convergencia prematura**. Esto se debía a que un cierto individuo tenía tal aptitud respecto al resto de la población, que rápidamente se reproducía haciendo ineficientes nuestros esfuerzos por respetar la variedad genética en la población. Por ello decidimos introducir en nuestro algoritmo la ejecución cada cierto tiempo de un fragmento de código que variaba aleatoriamente la población, de tal manera que introducía en ella la suficiente variedad para no quedar atrapada en estos óptimos locales y seguir cambiando.
Este código sigue el esquema de *Hill Climbing*, por el cual cambiamos aleatoriamente los genes del individuo, si hay mejora nos quedamos con la nueva configuración genética, y si no la descartamos.

Con formato: Interlineado: 1,5 líneas

7.2.4 Problemas durante el desarrollo de la implementación

En el transcurso del proyecto nos hemos encontrado algunos problemas, los cuales detallamos a continuación:

SCHWEFEL

Con formato: Interlineado: 1,5 líneas

Con formato: Interlineado: 1,5 líneas, Con viñetas + Nivel: 1 + Alineación: 0,63 cm + Tabulación después de: 1,27 cm + Sangría: 1,27 cm

- El primer problema que nos encontramos fue la representación de los números, debíamos representar números decimales positivos y negativos. Teníamos varias representaciones posibles, pero optamos por una representación fácil y eficaz al mismo tiempo, así que utilizamos un bit para el signo, nueve bits para la parte entera y seis para la parte decimal.
- Una vez creado el código en C, comenzamos con pruebas a ver si obteníamos los resultados esperados, pero observamos que el resultado no convergía al ritmo que esperábamos y el resultado final no se acercaba al óptimo. La solución para este problema fue incorporar a nuestro código un Hill Climbing con el cual obtuvimos mejores resultados y de forma más rápida.

- Otro de los problemas que nos encontramos fue que al ejecutar el código en el ordenador e insertarle un número de individuos superior a 512 nos daba un stack overflow (esto quiere decir que la pila se quedaba demasiado pequeña para la cantidad de individuos que metíamos). Este problema no lo hemos podido solucionar ya que no está dentro de nuestro alcance.

- Por último, uno de los problemas que más quebraderos de cabeza nos ha dado ha sido la obtención de números aleatorios en funciones que se estén ejecutando de forma paralela ya que dentro del código paralelo no podemos hacer llamadas random(). Una de las primeras soluciones que se nos ocurrió fue la incorporación de un código generador de números aleatorios dentro de la parte paralela, pero ésta no fue una buena idea ya que le proporcionaba demasiada

carga al código paralelo haciéndolo más lento, con lo que perdíamos eficiencia. La segunda solución que se nos ocurrió fue la generación de un vector de números aleatorios en la parte secuencial y pasárselo como argumento a la función paralelizada. Ésta solución se ajusto mucho mejor a nuestros requisitos ya que no le imprimía sobrecarga alguna a la parte paralela y obteníamos los resultados esperados.

PROBLEMA DEL VIAJANTE DE COMERCIO

- Un problema muy sencillo que nos encontramos a la hora de implementar el algoritmo del viajante de comercio fue, como representar la inexistencia de camino entre dos ciudades. En un comienzo pensamos en representarlo mediante 0 pero rápidamente nos dimos cuenta que non podía ser ya que no tenía sentido, así que decidimos darle un valor muy alto de tal forma que ese camino nunca se escogiera para formar parte del resultado final. El número en concreto para la representación fue "maxint".

- Otro problema que nos encontramos fue que después de realizar el EMMRS (o aplicar el operador oportuno) en el vector que nos daba la solución, podía contener números duplicados, cosa que era errónea, ya que cada ciudad solo se podía visitar una sola vez. Para solventar este problema tuvimos que implementar una función que eliminaba los duplicados del vector solución. Lo malo de esta solución es que hace más pesado el programa y hace que vaya más lento.

- En el caso del problema del viajante de comercio también nos encontramos con problema de los números aleatorios que lo solucionamos del mismo modo que en el algoritmo de Schwefel.

- Aquí también incorporamos el Hill Climbing para mejorar la convergencia al óptimo.

8 Gráficas de Rendimiento y Comparativas

Para evaluar los resultados de nuestro proyecto vamos a analizar los datos que se han desprendido de éste, desde dos puntos de vista: la calidad y la velocidad.

Por un parte tenemos los resultados de calidad, que son aquellos que se basan en la relación de los datos obtenidos por la función de fitness y el óptimo global del problema en cuestión.
En este apartado, tal y como hemos mencionado con anterioridad, la función de Schwefel tiene su óptimo global en 0, por lo que los valores más cercanos a esta cifra suponen una mejor adaptación al problema. Análogamente para el caso del TSP, su mínimo global estará en el número de ciudades que estemos tratando en ese momento, ya que aunque la inicialización es aleatoria, hemos habilitado un camino mínimo de peso igual a 7, 11 ó 16.

Creemos conveniente aclarar la forma, por la que hemos optado para representar este rasgo. A partir de un conjunto de mediciones para cada caso particular hemos extraído sus valores medios, superiores e inferiores. Por lo tanto, lo que nos vamos a encontrar en nuestras gráficas son un conjunto de 3 puntos para cada contexto. Contexto que viene determinado para cada problema por una serie de parámetros: hardware sobre el que se han tomado los datos (CPU o CPU+GPU), número de iteraciones, número de individuos en la población, método empleado para su resolución (EMMRS, cruce en un punto y cruce en dos puntos) y, en el caso del TSP, el número de ciudades.

El otro rasgo que será examinado en esta sección son los tiempos de ejecución de los algoritmos. Para ello representaremos el cociente del tiempo en CPU entre el tiempo que implique esa misma ejecución sobre la GPU, expresión llamada comúnmente ganancia o *speed-up*. No es necesario aclarar, que los mejores resultados serán aquellos

que consuman un menor tiempo de cómputo en la ejecución mixta (CPU+GPU), dando para ello un valor de *speed-up* mayor.

Para ambos casos hemos creído conveniente incluir, a modo aclaratorio, una pequeña tabla con el resumen de los datos que queremos representar en la gráfica correspondiente.
Por último puntualizar mediante la listad adjunta a continuación, el significado de las siglas empleadas.

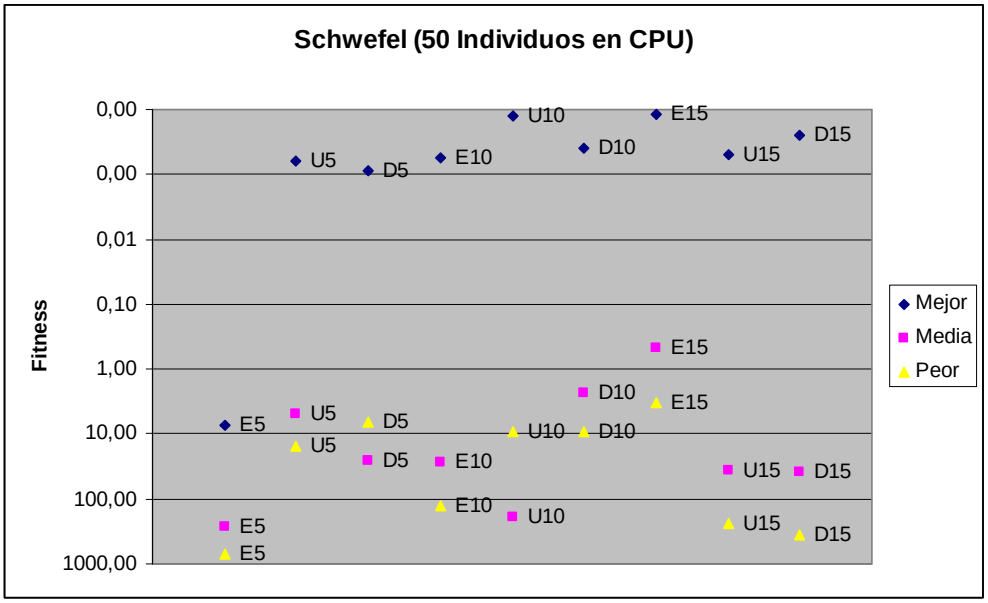
En el caso de la función de Schwefel,

E5	5.000 iteraciones con el método EMMRS
U5	5.000 iteraciones con el método de cruce en un punto
D5	5.000 iteraciones con el método de cruce en dos puntos
E10	10.000 iteraciones con el método EMMRS
U10	10.000 iteraciones con el método de cruce en un punto
D10	10.000 iteraciones con el método de cruce en dos puntos
E15	15.000 iteraciones con el método EMMRS
U15	15.000 iteraciones con el método de cruce en un punto
D15	15.000 iteraciones con el método de cruce en dos puntos

En el caso del TSP,

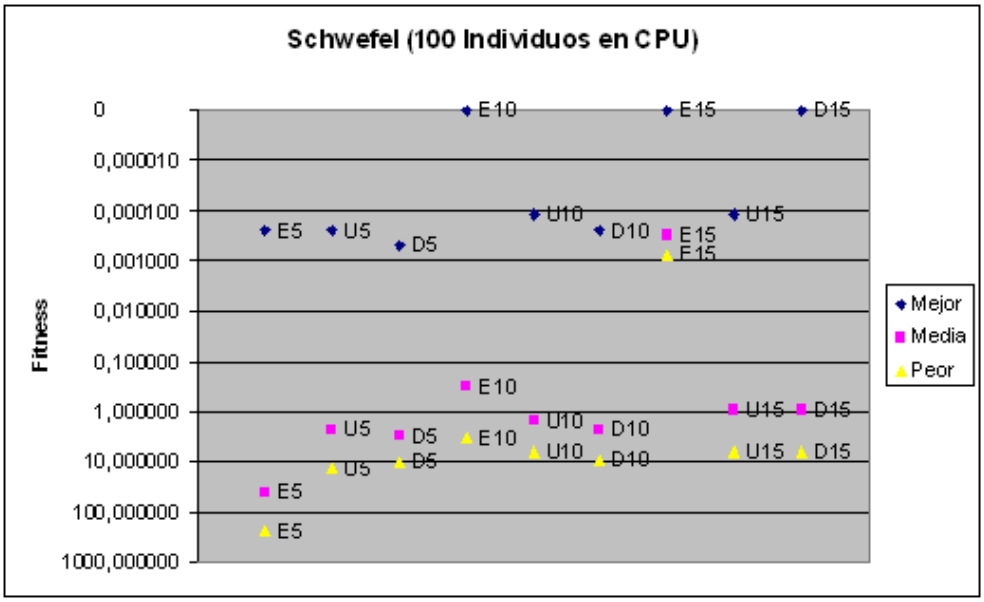
E1c	100 iteraciones con el método EMMRS
U1c	100 iteraciones con el método de cruce en un punto
D1c	100 iteraciones con el método de cruce en dos puntos
E1K	1.000 iteraciones con el método EMMRS
U1K	1.000 iteraciones con el método de cruce en un punto
D1K	1.000 iteraciones con el método de cruce en dos puntos
E10K	10.000 iteraciones con el método EMMRS
U10K	10.000 iteraciones con el método de cruce en un punto
D10K	10.000 iteraciones con el método de cruce en dos puntos

8.1 Resultados de calidad: Función de Schwefel



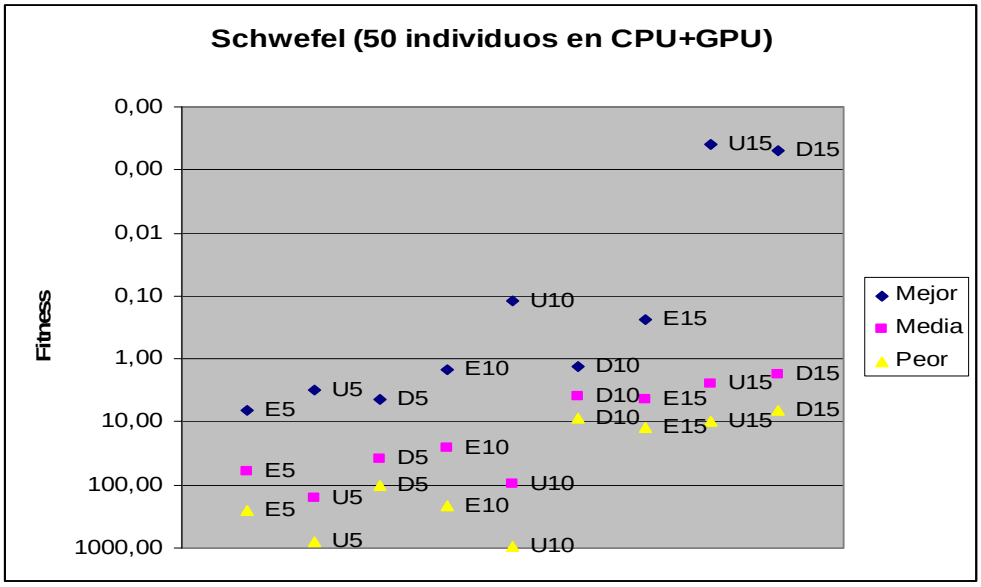
	Mejor	Media	Peor
E5	7,49	267,96	698,87
U5	0,000616	4,8667	15,6251
D5	0,000894	26,185	6,490
E10	0,000550	27,855	129,870
U10	0,000123	189,984	9,375
D10	0,000401	2,301	9,375
E15	0,000120	0,4756	3,3695
U15	0,000493	36,3712	236,1819
D15	0,000247	37,9800	354,2118

El primer caso de estudio es el de la función de Schwefel, para 50 individuos. Hemos optado por elegir una escala logarítmica, para poder apreciar mejor los detalles obtenidos. Podemos observar como los valores están relativamente próximos, obteniendo los mejores resultados con el método del EMMRS para 15.000 iteraciones.



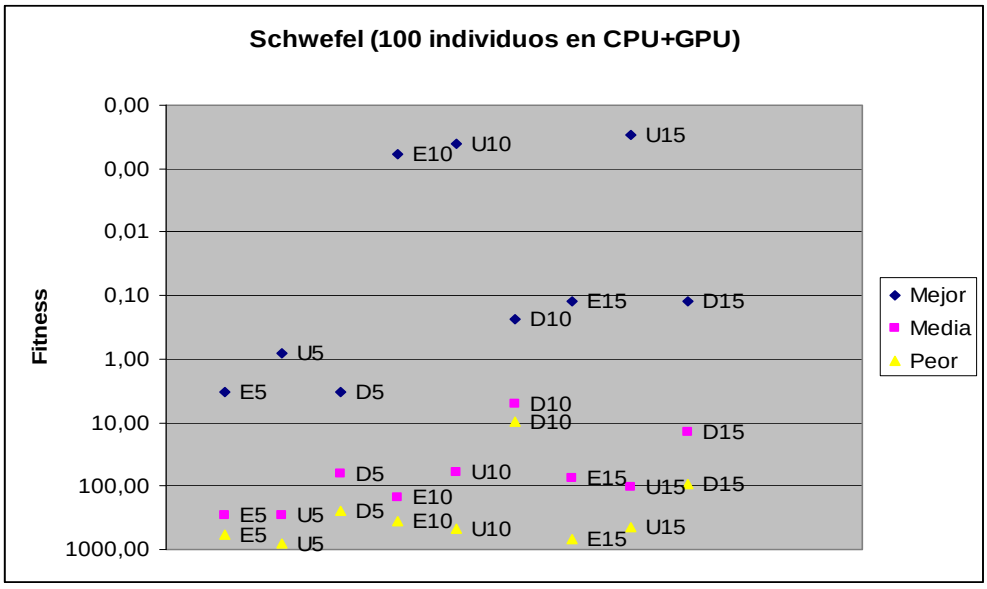
	Mejor	Media	Peor
E5	0,000247	38,62	234,10
U5	0,000246	2,25	12,74
D5	0,000493	2,976	9,619
E10	0	0,3127	3,125
U10	0,000123	1,575	6,372
D10	0,000247	2,212	9,375
E15	0	0,000312	0,00074
U15	0,000123	0,9378	6,25
D15	0	0,9501	6,3723

En este otro caso de estudio vemos, como al aumentar el número de individuos de una población los resultados también mejoran en precisión, y como nuevamente EMMRS se muestra superior a sus competidores, tanto en los resultados como en la mejora que experimenta en sus valores, a medida que ejecutamos más veces el algoritmo.



	Mejor	Media	Peor
E5	6,40	61,76	252,34
U5	3,13	164,99	807,73
D5	4,42	39,20	101,73
E10	1,52	25,09	207,89
U10	0,1228	95,85	919,23
D10	1,33	3,94	8,86
E15	0,24	4,42	12,62
U15	0,0004	2,50	9,50
D15	0,0005	1,76	6,37

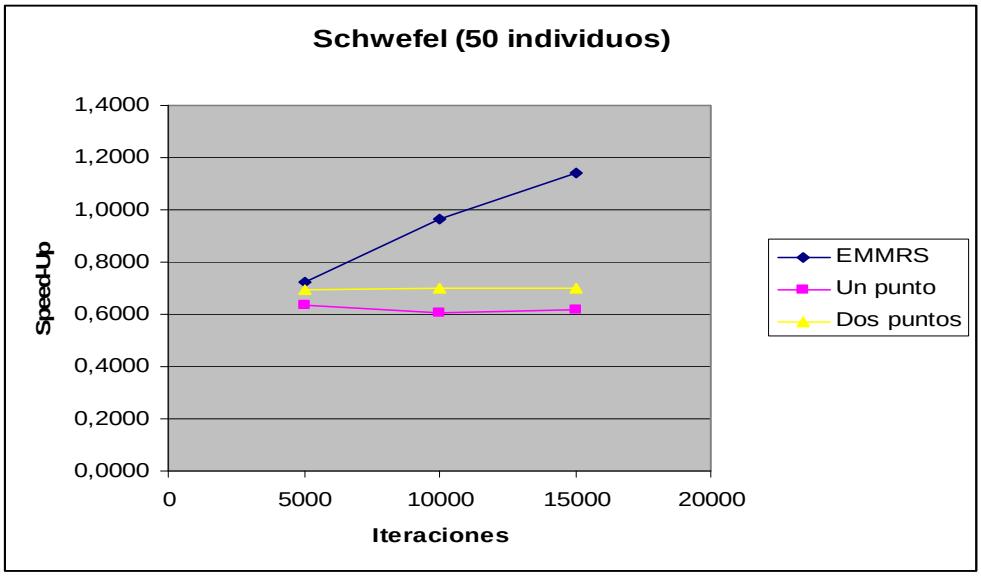
Con respecto a esta ejecución y su análoga en CPU, cabe destacar como los resultados medios mejoran ligeramente de una a otra, y salvo los casos óptimos de cruce en un punto y cruce en dos puntos todos los resultados son parejos.



	Mejor	Media	Peor
E5	3,37	290,30	583,90
U5	0,80	292,20	825,64
D5	3,28	64,80	242,55
E10	0,0006	149,70	354,21
U10	0,0004	61,80	466,00
D10	0,24	5,18	9,62
E15	0,12	73,45	692,55
U15	0,0003	105,10	440,97
D15	0,12	14,40	92,81

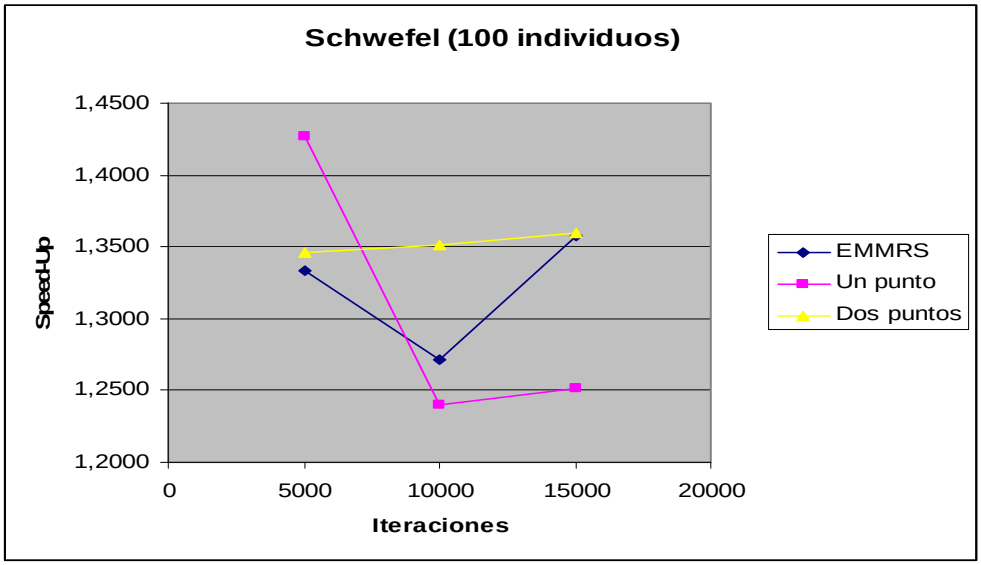
La toma de datos en el caso de 100 individuos en CUDA para la función de Schwefel arrojó unos datos relativamente pobres, sobre todo si lo comparamos con los ejemplos anteriores. Una posible explicación para ello, sea que una mala incialización juntos con algún estancamiento en óptimos locales que no supo solventar nuestro Hill Climbing, condenaron a los algoritmos a unos valores nada alentadores.

8.2 Speed-Up: Función de Schwefel



	5000 iteraciones	10000 iteraciones	15000 iteraciones
EMMRS	0,7251	0,9669	1,1436
Un punto	0,6347	0,6070	0,6185
Dos puntos	0,6945	0,7027	0,7029

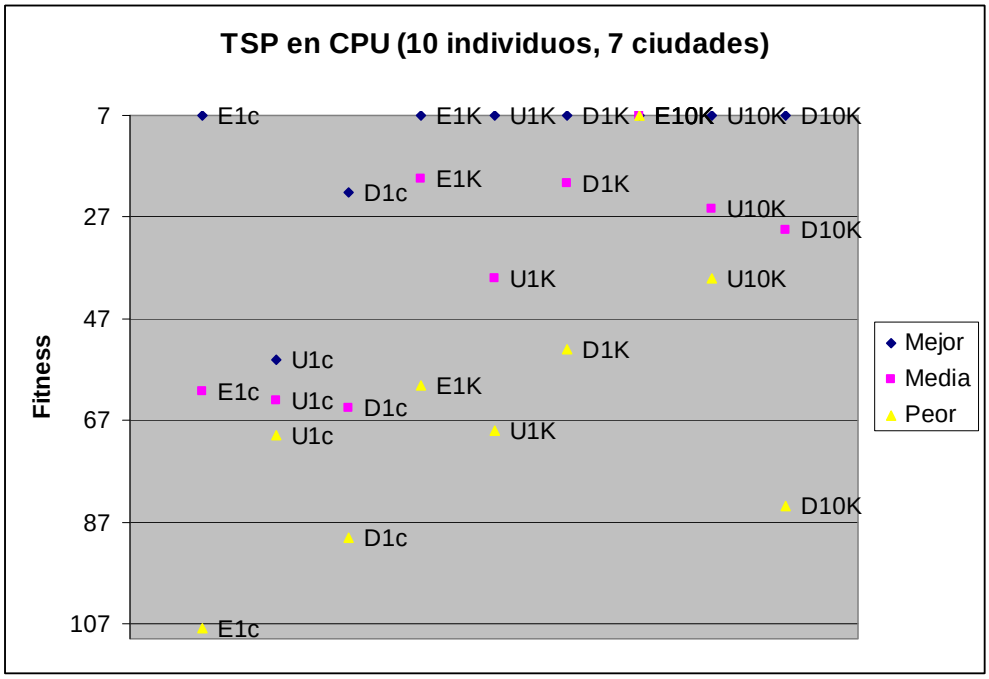
La primera gráfica de comparativas de tiempos nos arroja unos resultados gratificantes para el EMMRS si lo comparamos con sus rivales. Mientras que el EMMRS realiza una mejora de 4 décimas de punto y acaba el caso de 15.000 iteraciones con una ganancia respecto a la versión de CPU cercana al 15%, tanto cruce en un punto como en dos no sólo no reducen sino que aumentan sus tiempos de ejecución en un 40% y en un 30% respectivamente.



	5000 iteraciones	10000 iteraciones	15000 iteraciones
EMMRS	1,3334	1,2717	1,3572
Un punto	1,4267	1,2403	1,2514
Dos puntos	1,3465	1,3512	1,3597

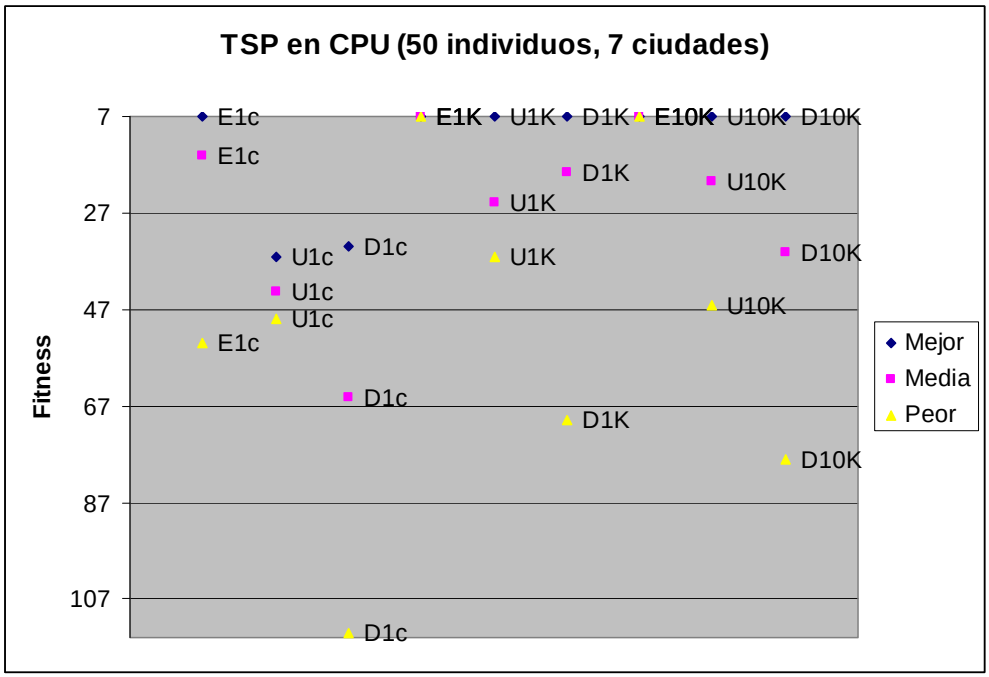
La otra gráfica de comparativas de tiempo de Schwefel, nos lleva al caso en que la población pasa de 50 a 100 individuos. En ella podemos apreciar un significativo descenso de la mejora de tiempos para los algoritmos con EMMRS y cruce en un punto al ejecutar 10.000 iteraciones, si lo comparamos con las ejecuciones de 5.000 y 15.000. Esto puede ser debido a la mala inicialización que comentábamos antes, y al trabajo extra que deberá de realizar la función de Hill Climbing para salir de los óptimos locales. No obstante, la mejora temporal en términos absolutos es mayor para los 3 métodos que en el caso de 50 individuos. Basta mirar el caso del mayor número de iteraciones para observar que hay una mejora del 25% para cruce en un punto y valores cercanos al 36% para cruce en dos puntos y EMMRS.

8.3 Resultados de calidad: TSP



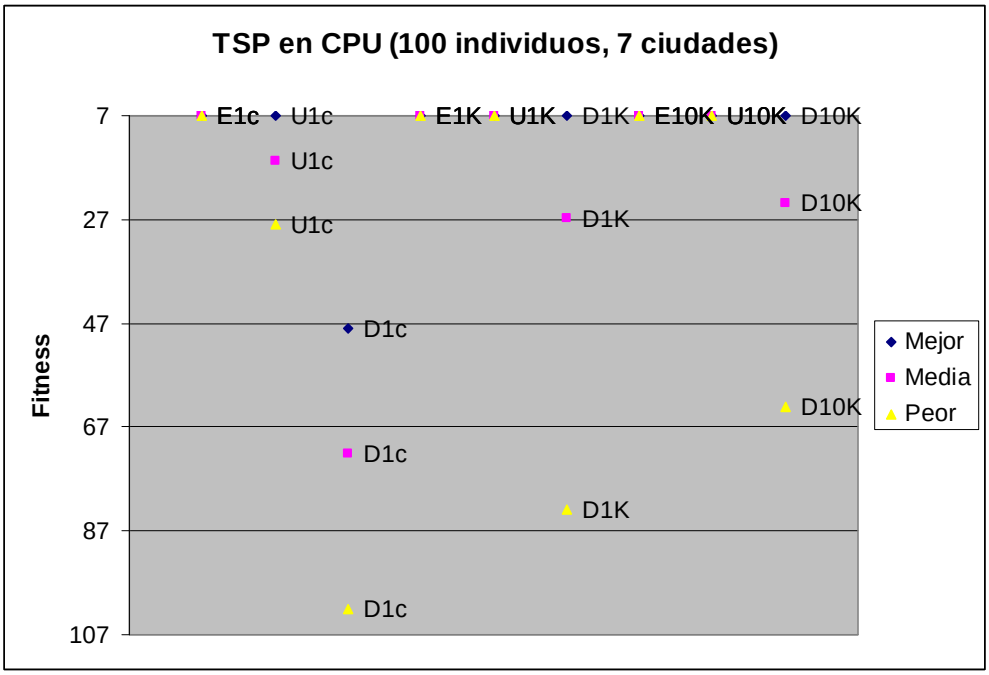
	Mejor	Media	Peor
E1c	7	61,2	108
U1c	55	63,1	70
D1c	22	64,6	90
E1K	7	19,4	60
U1K	7	39,2	69
D1K	7	20,4	53
E10K	7	7,0	7
U10K	7	25,3	39
D10K	7	29,6	84

Empezamos el estudio de la calidad del TSP con un ejemplo para resolver el camino mínimo entre 7 ciudades. Dado el número de individuos los resultados tienen una gran dispersión y no son del todo óptimos, pero eso no es óbice para que EMMRS (con suficientes iteraciones) encuentre siempre el camino más corto.



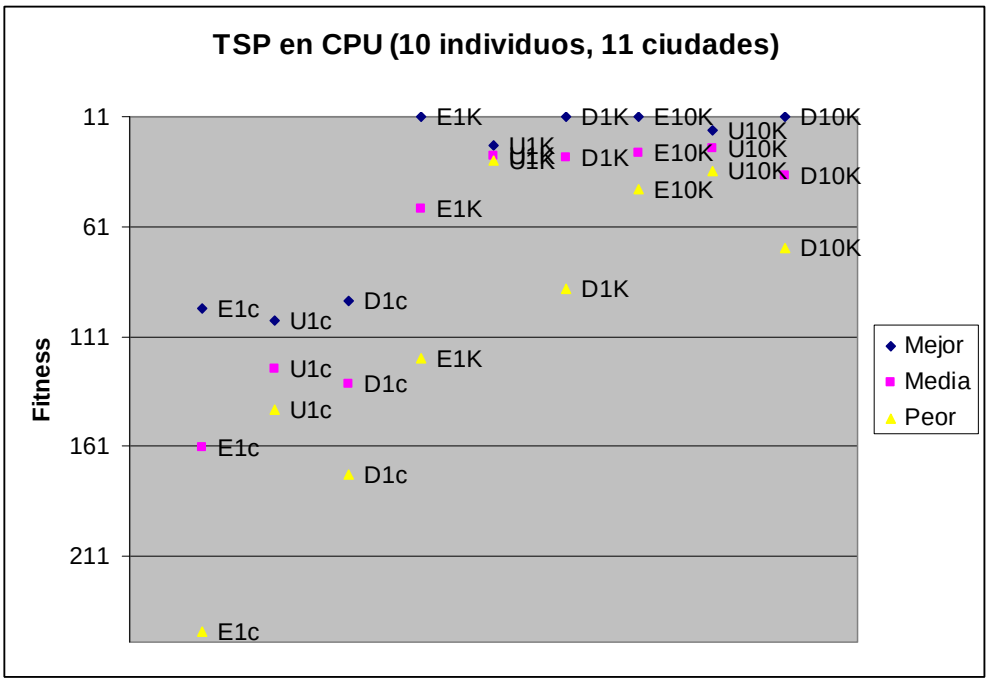
	Mejor	Media	Peor
E1c	7	15,0	54
U1c	36	43,4	49
D1c	34	65,2	114
E1K	7	7,0	7
U1K	7	24,7	36
D1K	7	18,6	70
E10K	7	7,0	7
U10K	7	20,5	46
D10K	7	35,3	78

Manteniendo el número de ciudades, aumentamos ahora si, el número de individuos a computar, con la consiguiente mejora para todos los algoritmos de sus resultados. No obstante, EMMRS sigue siendo quien mejores resultados consigue.



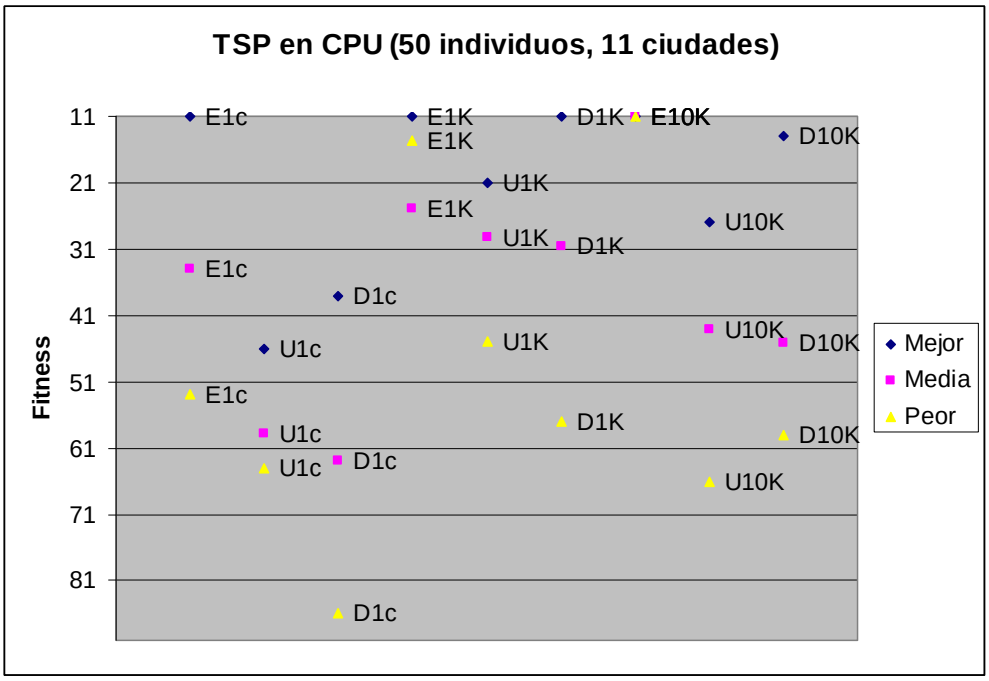
	Mejor	Media	Peor
E1c	7	7,0	7
U1c	7	15,6	28
D1c	48	72,1	102
E1K	7	7,0	7
U1K	7	7,0	7
D1K	7	26,7	83
E10K	7	7,0	7
U10K	7	7,0	7
D10K	7	23,9	63

En este caso, los resultados en todos los algoritmos pasan a ser los óptimos, gracias al aumento a 100 individuos de todas las poblaciones.



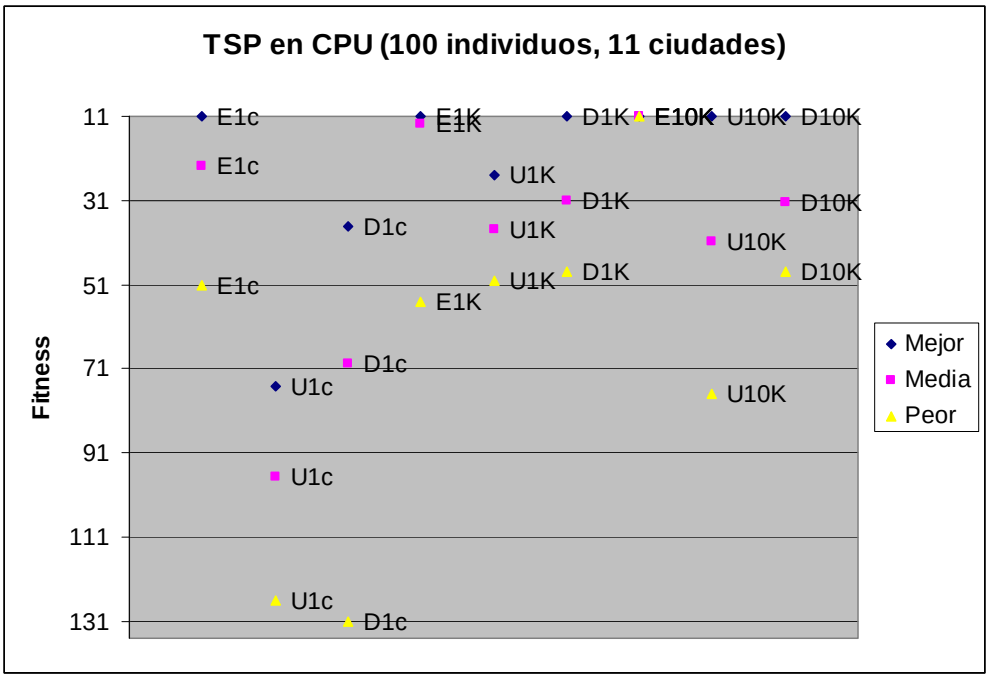
	Mejor	Media	Peor
E1c	98	161,3	245
U1c	104	125,7	144
D1c	95	132,5	174
E1K	11	53,1	121
U1K	24	28,7	31
D1K	11	29,7	89
E10K	11	27,6	44
U10K	17	25,7	36
D10K	11	37,5	71

Pasamos a otro caso de estudio, un ejemplo con 11 ciudades y 10 individuos para las poblaciones. Aquí descubrimos que los resultados son muy similares fruto, probablemente, del bajo número de individuos. Destacan los valores mínimos de EMMRS y cruce en dos puntos, que son los únicos que logran alcanzar la solución, 11.



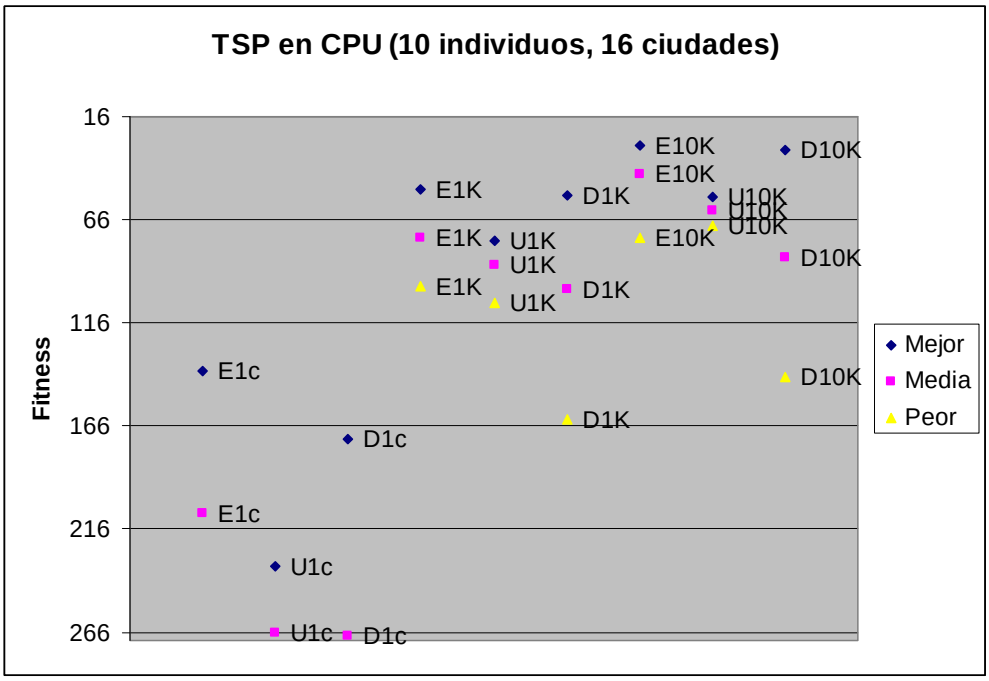
	Mejor	Media	Peor
E1c	11	34,1	53
U1c	46	58,8	64
D1c	38	62,9	86
E1K	11	25,0	15
U1K	21	29,3	45
D1K	11	30,6	57
E10K	11	11,0	11
U10K	27	43,1	66
D10K	14	45,2	59

Incrementamos nuevamente el número de individuos a 50, y apreciamos la superioridad de EMMRS frente a los otros dos algoritmos de cruce, no sólo en que es el único que logra alcanzar el óptimo al menos una vez en cada caso de estudio (100, 1.000 y 10.000 iteraciones), para 10.000 iteraciones lo alcanza siempre.



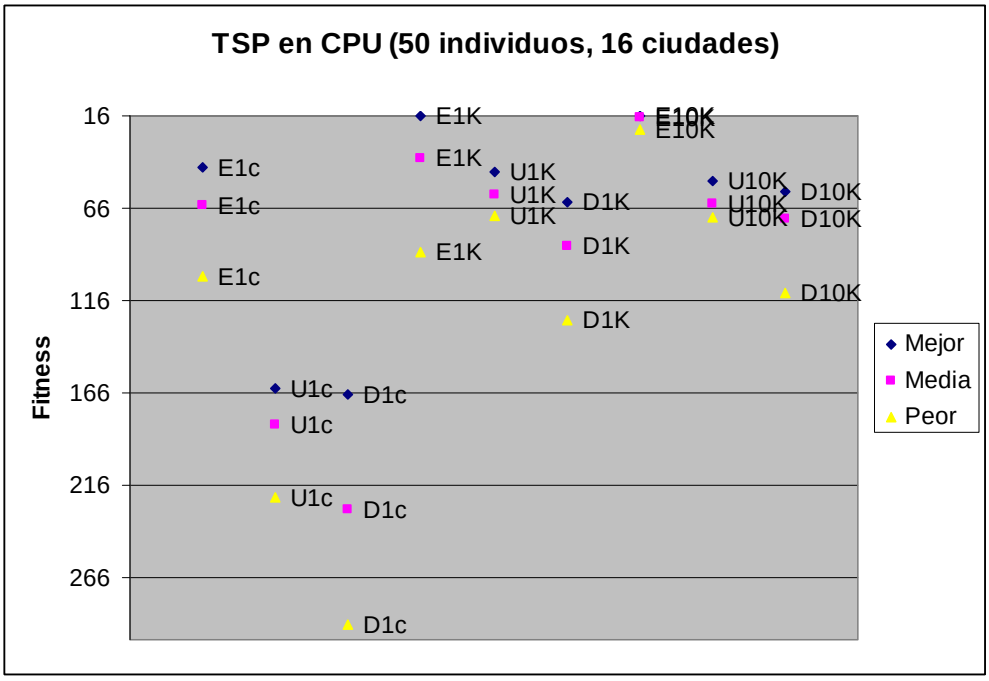
	Mejor	Media	Peor
E1c	11	22,8	51
U1c	75	96,7	126
D1c	37	69,8	131
E1K	11	12,8	55
U1K	25	37,7	50
D1K	11	31,1	48
E10K	11	11,0	11
U10K	11	40,7	77
D10K	11	31,4	48

Con 100 individuos, prácticamente son todos los algoritmos los que encuentran el óptimo. A destacar los bajos resultados obtenidos por cruce en un punto debido, a nuestro entender, al bajo número de iteraciones, ya que después se recupera y logra alcanzar el óptimo, aunque con cierta dispersión en sus valores.



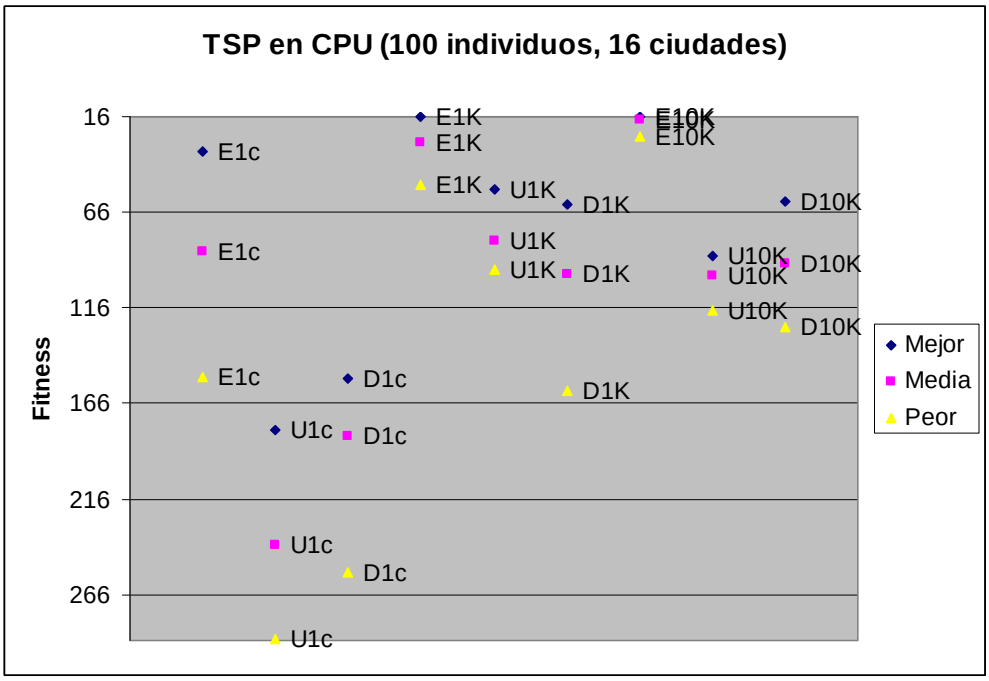
	Mejor	Media	Peor
E1c	139	208,4	317
U1c	234	266,3	310
D1c	172	267,5	327
E1K	51	74,4	98
U1K	76	88,2	106
D1K	54	99,4	163
E10K	30	43,9	75
U10K	55	61,5	69
D10K	32	84,3	142

Llegamos al caso en el que hay una mayor dependencia de una buena inicialización para encontrar la solución óptima, debido al ratio tan pequeño entre número de individuos y ciudades. Es por ello que los valores conseguidos para todos los algoritmos son muy similares.



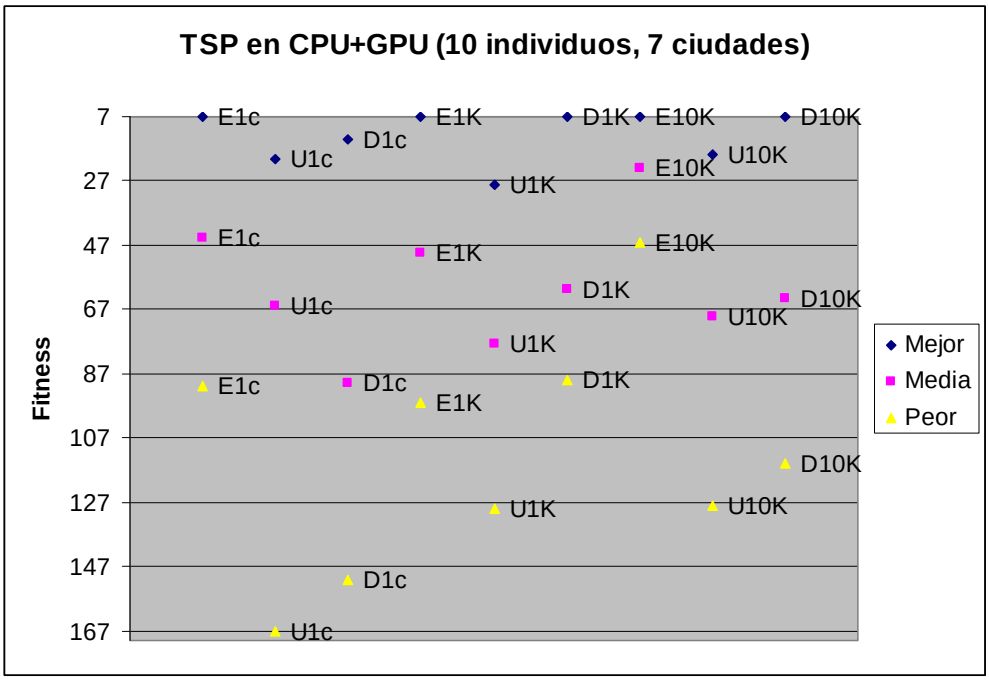
	Mejor	Media	Peor
E1c	44	64,6	103
U1c	164	183,7	223
D1c	167	229,6	292
E1K	16	38,9	90
U1K	46	58,3	70
D1K	63	86,9	127
E10K	16	17,1	23
U10K	51	63,7	71
D10K	57	71,9	112

Como hemos señalado anteriormente, al aumentar el número de individuos, los valores alcanzados por las soluciones mejoran, siendo destacable una vez más, los logrados por EMMRS



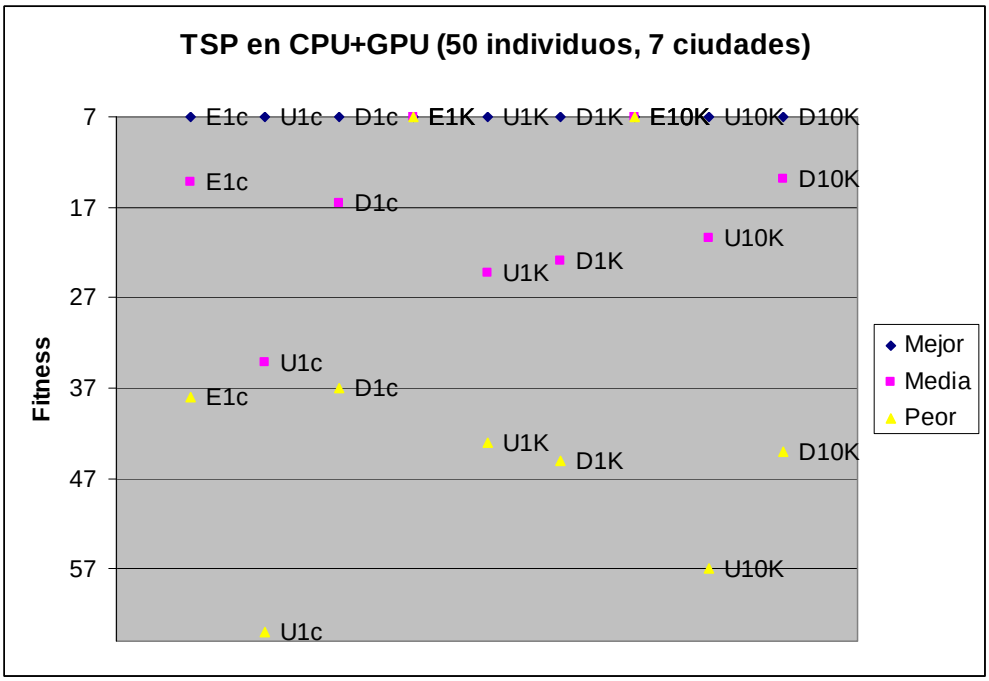
	Mejor	Media	Peor
E1c	34	86,5	152
U1c	180	240,1	289
D1c	153	183,1	254
E1K	16	29,3	52
U1K	54	81,3	96
D1K	62	98,5	159
E10K	16	17,2	26
U10K	89	99,3	117
D10K	60	92,8	126

Para acabar con las ejecuciones en CPU del TSP, analizamos el caso de 16 ciudades con 100 individuos. Para un número no muy elevado de iteraciones, los valores de los algoritmos muestran una amplia dispersión, pero aún así los peores datos del EMMRS son mejores que los de sus rivales. Al aumentar el número de iteraciones todos ellos, mejoran sus resultados, poniendo de manifiesto la superioridad del EMMRS, al hallar más frecuentemente mejores resultados.



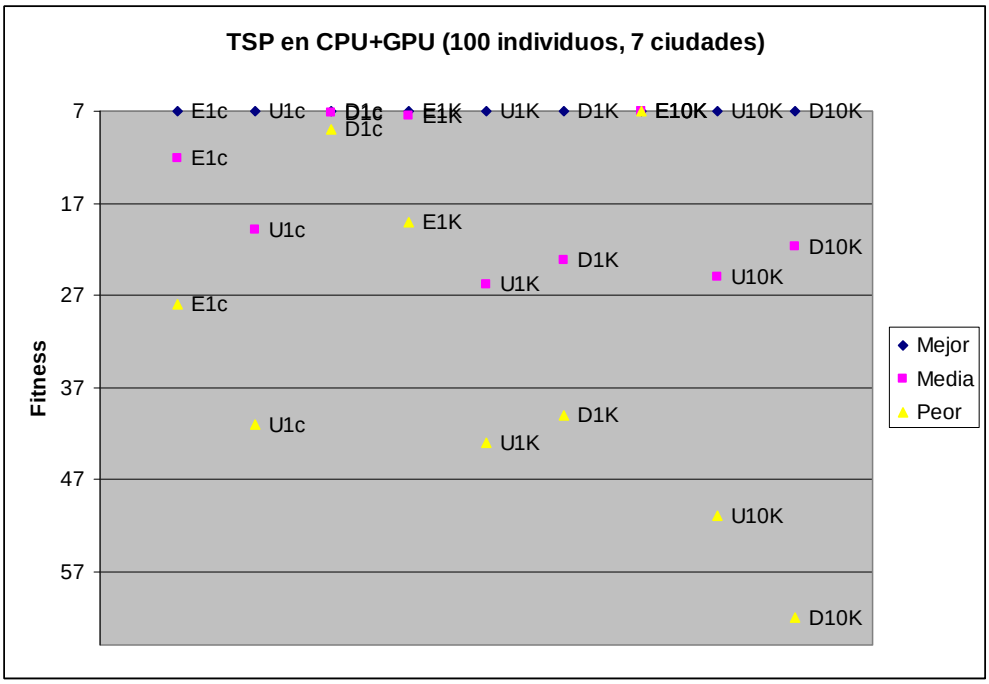
	Mejor	Media	Peor
E1c	7	44,5	91
U1c	20	65,9	167
D1c	14	89,9	151
E1K	7	49,6	96
U1K	28	77,5	129
D1K	7	60,8	89
E10K	7	23,0	46
U10K	19	69,4	128
D10K	7	63,5	115

La gráfica de más arriba nos muestra la enorme dispersión que se produce en la ejecución de los algoritmos en CPU+GPU, aún así destacan cruce en dos puntos y EMMRS, que son capaces de obtener al menos en dos y tres de los casos respectivamente el mejor valor, 7.



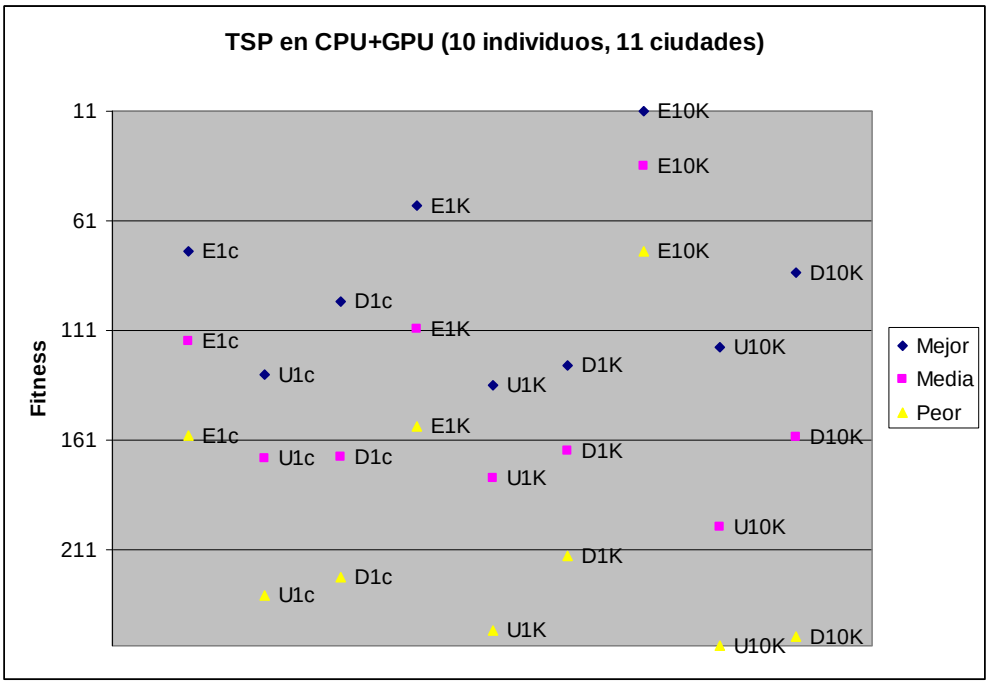
	Mejor	Media	Peor
E100	7	14,2	38
U100	7	34,2	64
D100	7	16,5	37
E1000	7	7,0	7,0
U1000	7	24,3	43
D1000	7	22,9	45
E10000	7	7,0	7
U10000	7	20,4	57
D10000	7	13,8	44

Al aumentar el número de individuos vemos como todos los algoritmos son capaces de encontrar unas buenas soluciones, sin embargo, EMMRS es el que menor dispersión en sus datos demuestra



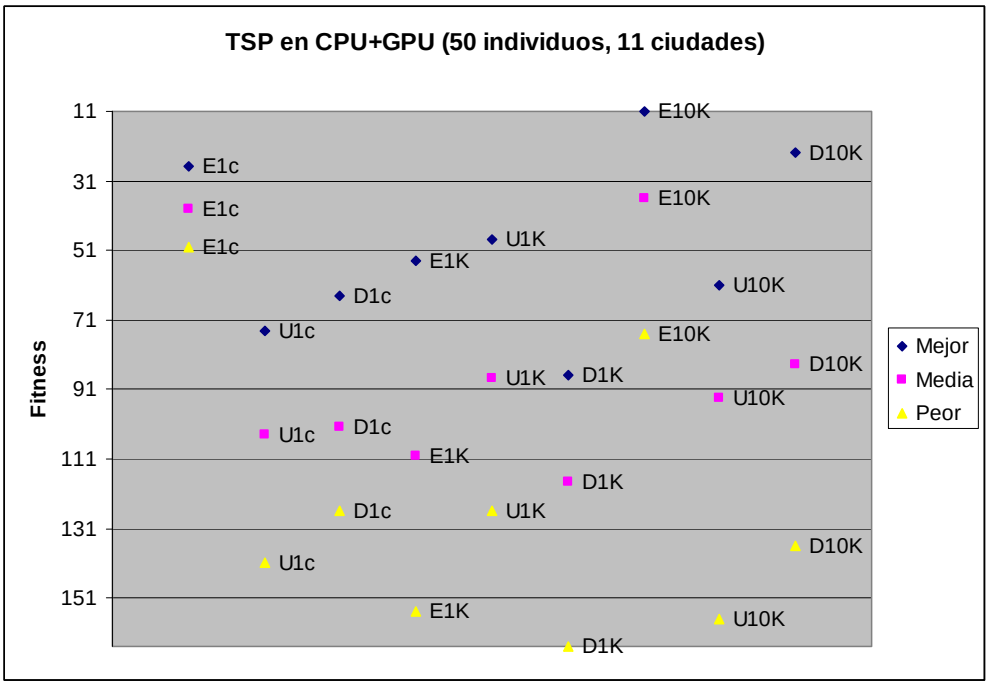
	Mejor	Media	Peor
E1c	7	12,1	28
U1c	7	19,9	41
D1c	7	7,2	9
E1K	7	7,5	19
U1K	7	25,8	43
D1K	7	23,2	40
E10K	7	7,0	7
U10K	7	25,0	51
D10K	7	21,7	62

El mismo comentario realizado para 50 individuos es extrapolable al que nos ocupa ahora mismo. Con una población tan grande, en comparación con el número de ciudades que manejamos, todos los algoritmos logran alcanzar el camino óptimo al menos una vez para cada caso de estudio, sin embargo sus dispersiones son diferentes. Mientras que EMMRS, logra unos valores bajos en todas las ejecuciones, sus rivales muestran un amplio rango de valores.



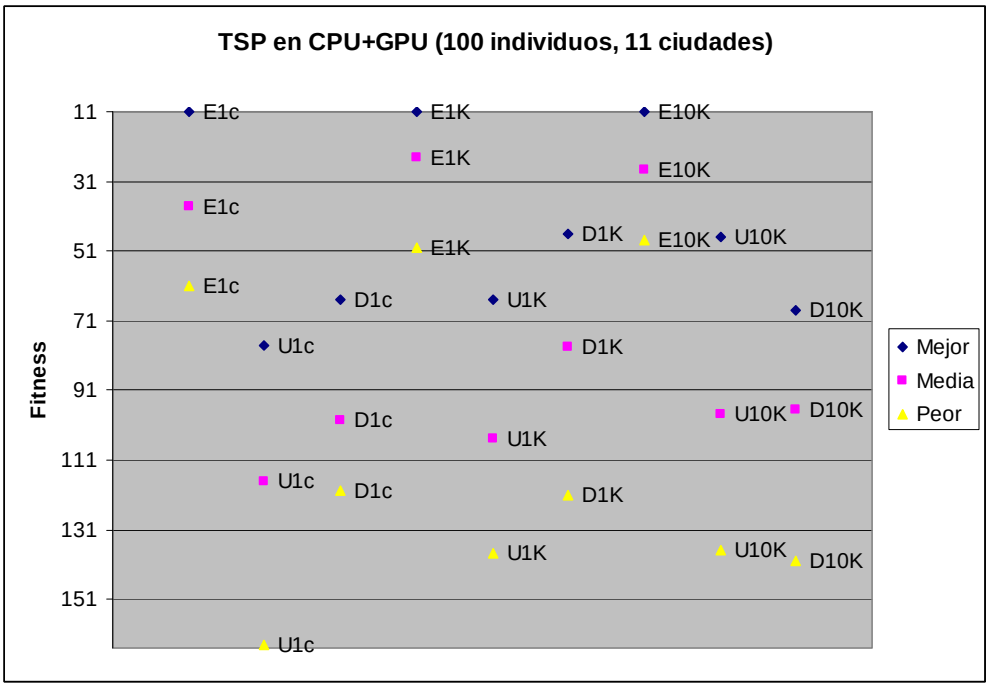
	Mejor	Media	Peor
E1c	75	115,7	159
U1c	131	169,2	232
D1c	98	168,8	224
E1K	54	110,1	155
U1K	136	178,5	248
D1K	127	165,7	214
E10K	11	35,8	75
U10K	119	201,0	255
D10K	85	159,7	251

Aumentamos nuevamente el número de ciudades y comprobamos como el peor resultado de EMMRS es mejor que todas las medias de cruce en un punto y cruce en dos puntos. Con este dato no es de extrañar que los mejores resultados, incluyendo el óptimo, se obtengan nuevamente a través de EMMRS.



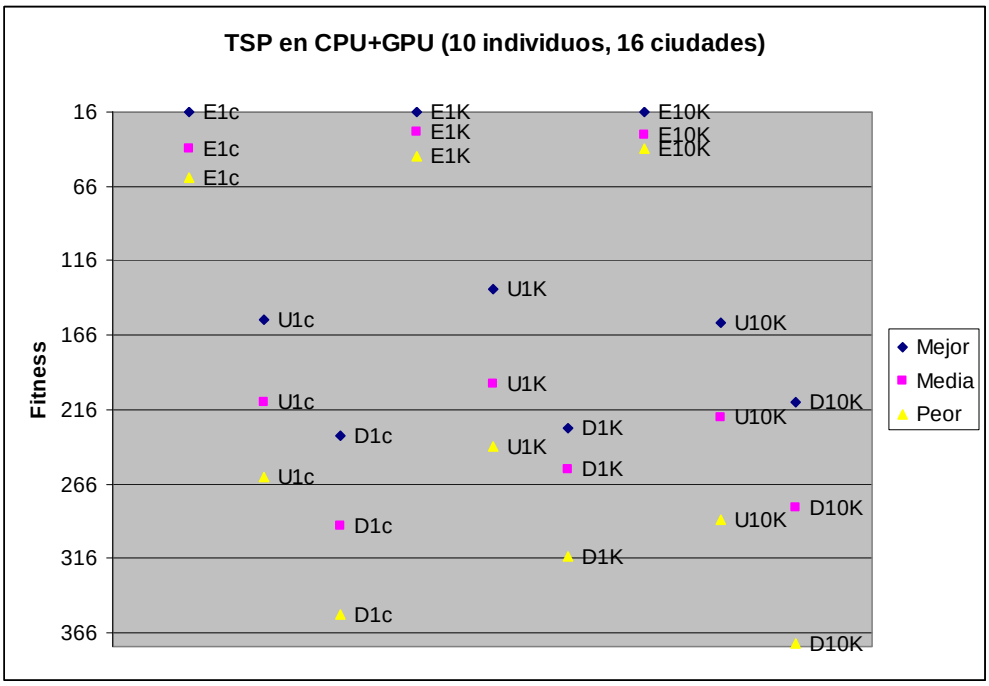
	Mejor	Media	Peor
E1c	27	39,0	50
U1c	74	104,2	141
D1c	64	101,8	126
E1K	54	110,1	155
U1K	48	87,7	126
D1K	87	117,5	165
E10K	11	35,8	75
U10K	61	93,3	157
D10K	23	84,0	136

En este caso, también EMMRS muestra los mejores resultados, tanto para 100 como para 10.000 iteraciones, sin embargo, muestra un empeoramiento hacia las 1.000 iteraciones. Probablemente, se debe a unos malos resultados en la inicialización y en la generación de números aleatorios empleados en el algoritmo.



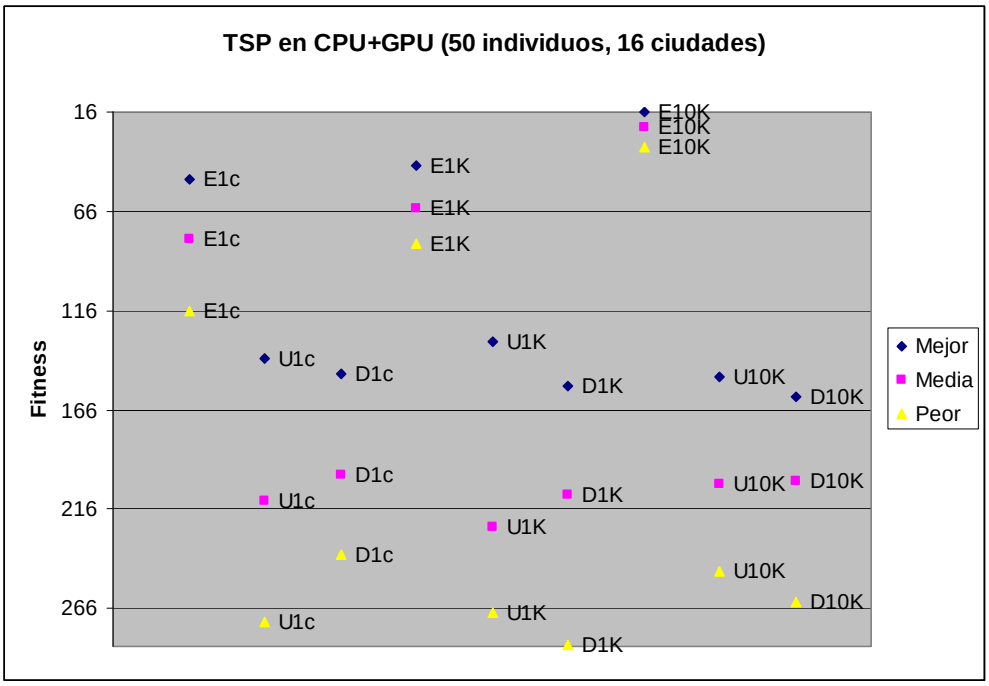
	Mejor	Media	Peor
E1c	11	38,0	61
U1c	78	117,3	164
D1c	65	99,5	120
E1K	11	24,1	50
U1K	65	104,9	138
D1K	46	78,6	121
E10K	11	27,6	48
U10K	47	97,8	137
D10K	68	96,6	140

Llevando el caso de las 11 ciudades a una población de 100 individuos, el EMMRS vuelve a dar síntomas de unos mejores resultados, tanto en valores absolutos (Caso mejor) como en frecuencia (Caso medio), ya que es el único que obtiene el óptimo y sus valores medios son mucho mejores que los de sus competidores.



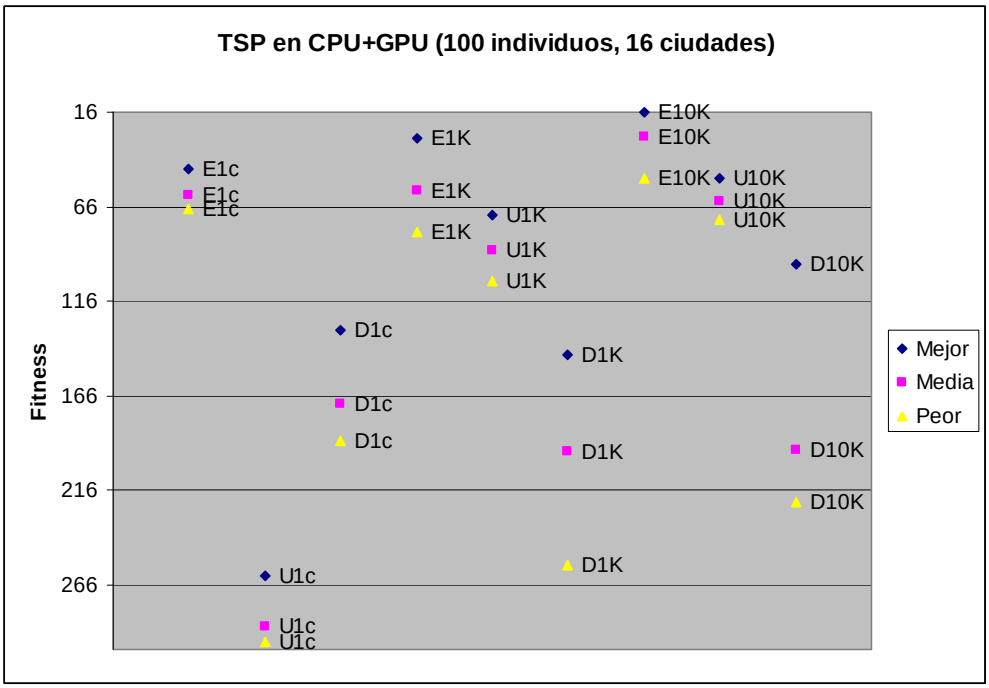
	Mejor	Media	Peor
E1c	16	40,2	60
U1c	156	210,7	261
D1c	233	293,7	353
E1K	16	29,7	46
U1K	135	198,7	241
D1K	228	256,0	314
E10K	16	31,0	41
U10K	158	220,7	290
D10K	211	282,0	373

Especialmente significativo es el caso de la gráfica de arriba. En el podemos observar como para una población muy baja (10 individuos) y para el número más elevado de ciudades de nuestros casos de estudio, las diferencias entre el EMMRS y cruce en uno y dos puntos, son enormes.



	Mejor	Media	Peor
E1c	50	80,2	116
U1c	140	211,6	273
D1c	148	199,0	239
E1K	43	64,4	82
U1K	132	224,8	268
D1K	154	209,0	284
E10K	16	23,8	34
U10K	149	203,5	247
D10K	159	202,0	263

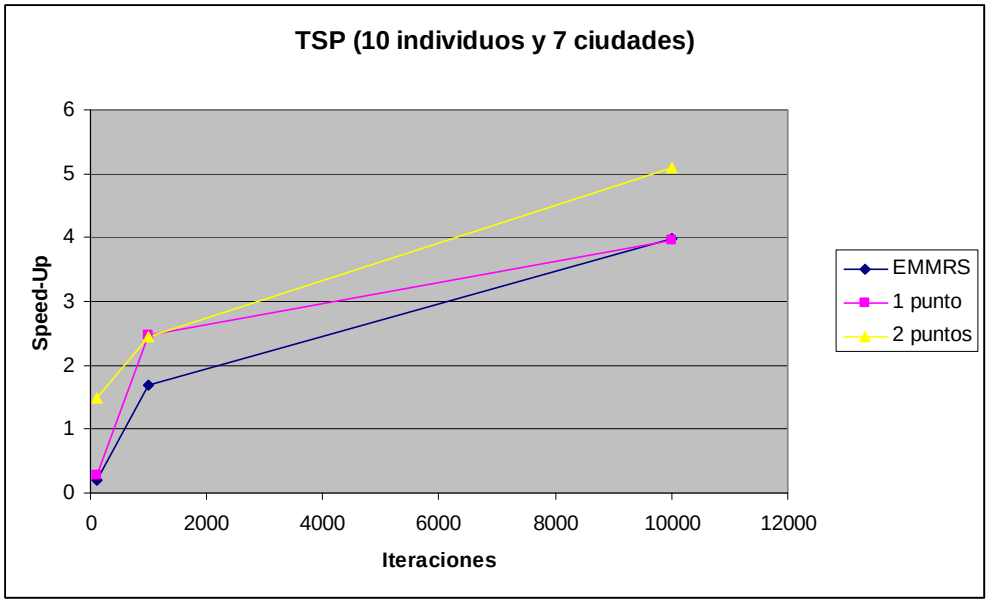
Igual que en el caso anterior la diferencia de resultados entre los tres tipos de algoritmos estudiados se refleja claramente en la gráfica correspondiente al cuadro de arriba. Mientras que EMMRS no sólo obtiene unos valores muy buenos, si no que además tiene una clara progresión y mejora (tanto de valores como de dispersión), los otros dos algoritmos mantienen intactas sus tendencias y sus pobres resultados.



	Mejor	Media	Peor
E1c	46	60,2	67
U1c	261	287,7	296
D1c	131	170,0	190
E1K	30	57,4	79
U1K	70	89,2	105
D1K	144	195,0	255
E10K	16	28,6	51
U10K	51	62,7	73
D10K	96	194,4	222

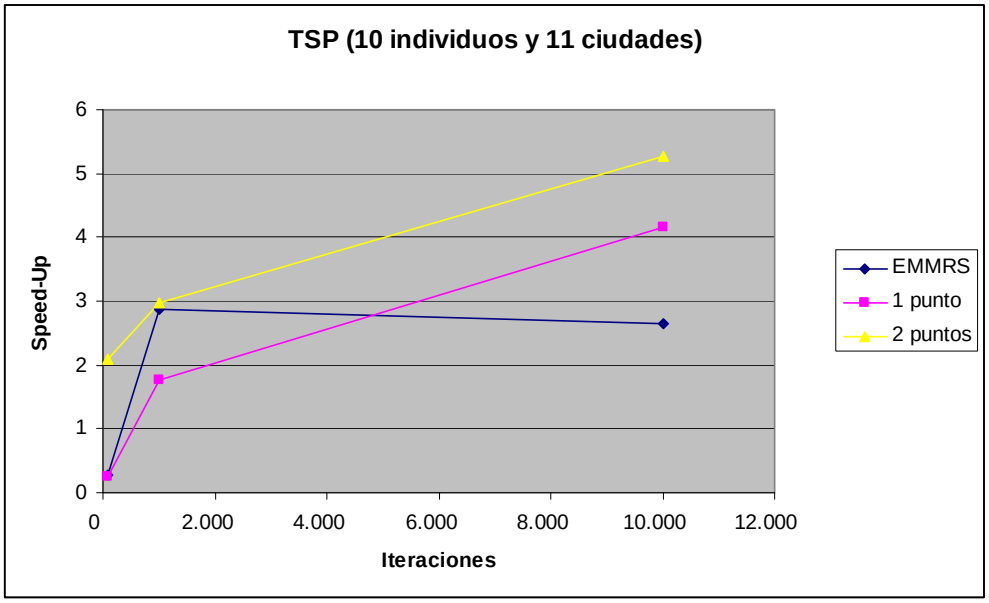
Para finalizar, los casos de estudio del TSP mostramos el ejemplo de buscar la solución de camino mínimo para 16 ciudades con 100 individuos. En él, podemos constatar nuevamente como EMMRS es más estable (no tiene demasiada dispersión) y certero (encuentra mejores soluciones) que cruce en un punto y dos puntos.

8.4 Speed-Up: TSP



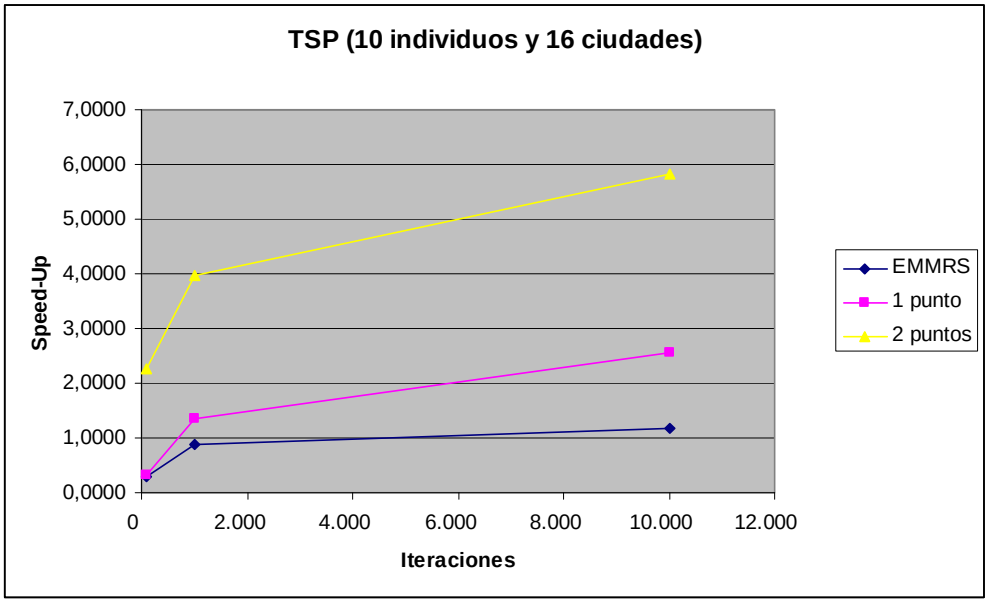
	100 iteraciones	1000 iteraciones	10000 iteraciones
EMMRS	0,1970	1,6843	3,9750
1 punto	0,2729	2,4765	3,9648
2 puntos	1,4915	2,4519	5,0930

Comenzamos las comparaciones de tiempos con el caso más básico de estudio (10 individuos y 7 ciudades). Como se puede apreciar en el gráfico, la mayor ganancia la obtiene el algoritmo de cruce en dos puntos, llegando a ejecutarse 5 veces más rápido que su versión en CPU. Tampoco son nada despreciables las ganancias obtenidas por EMMRS y cruce en un punto, con casi una reducción al 25% del tiempo original.



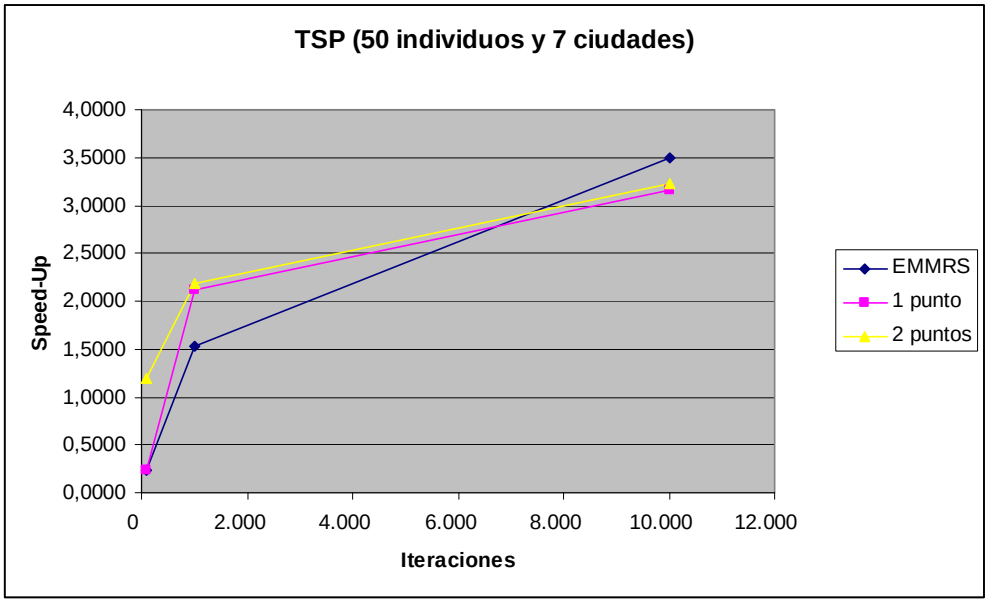
	100 iteraciones	1000 iteraciones	10000 iteraciones
EMMRS	0,2804	2,8627	2,6531
1 punto	0,2448	1,7755	4,1606
2 puntos	2,0968	2,9647	5,2614

La siguiente gráfica pone de manifiesto la aceleración respecto a las implementaciones para CPU que sufren los algoritmos de cruce en un punto y cruce en dos puntos. Sin embargo, EMMRS no logra alcanzar tales cotas, aunque mejora el tiempo de CPU considerablemente.



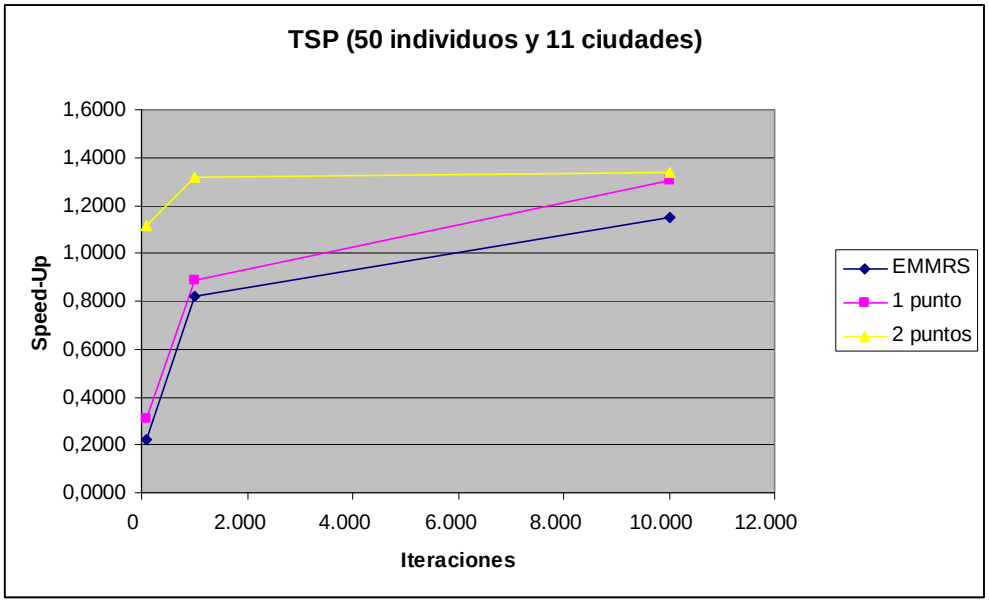
	100 iteraciones	1000 iteraciones	10000 iteraciones
EMMRS	0,3062	0,8805	1,1715
1 punto	0,3109	1,3657	2,5674
2 puntos	2,2787	3,9673	5,8366

En el último ejemplo con 10 individuos, vemos como las gráficas de rendimiento son prácticamente paralelas, pero ya se empiezan a atisbar unas diferencias más que notables, puesto que el EMMRS, solamente obtiene mejoras en torno al 17%, creemos que esto puede ser debido a la sobrecarga de comunicaciones a las que puede ser expuesto el algoritmo



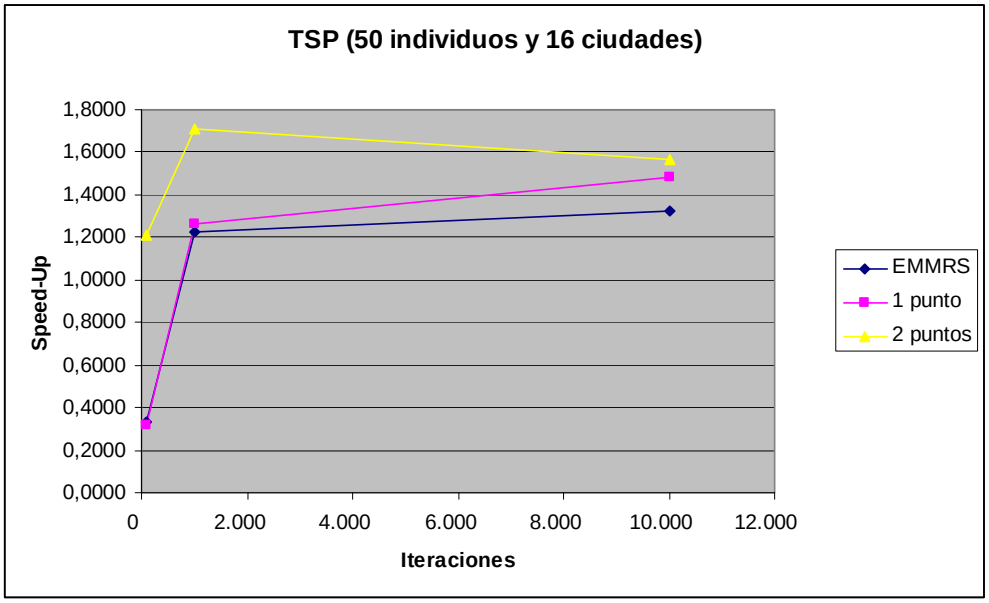
	100 iteraciones	1000 iteraciones	10000 iteraciones
EMMRS	0,2414	1,5374	3,4918
1 punto	0,2350	2,1135	3,1643
2 puntos	1,1975	2,1918	3,2292

Pasemos a los ejemplo sobre poblaciones mayores, concretamente a las de 50 individuos. En el ejemplo que acompaña esta sección podemos ver que los trazos correspondientes al EMMRS y a los cruces de uno y dos puntos, son prácticamente iguales, obteniendo con el mayor número de iteraciones su rendimiento máximo superior a 3.



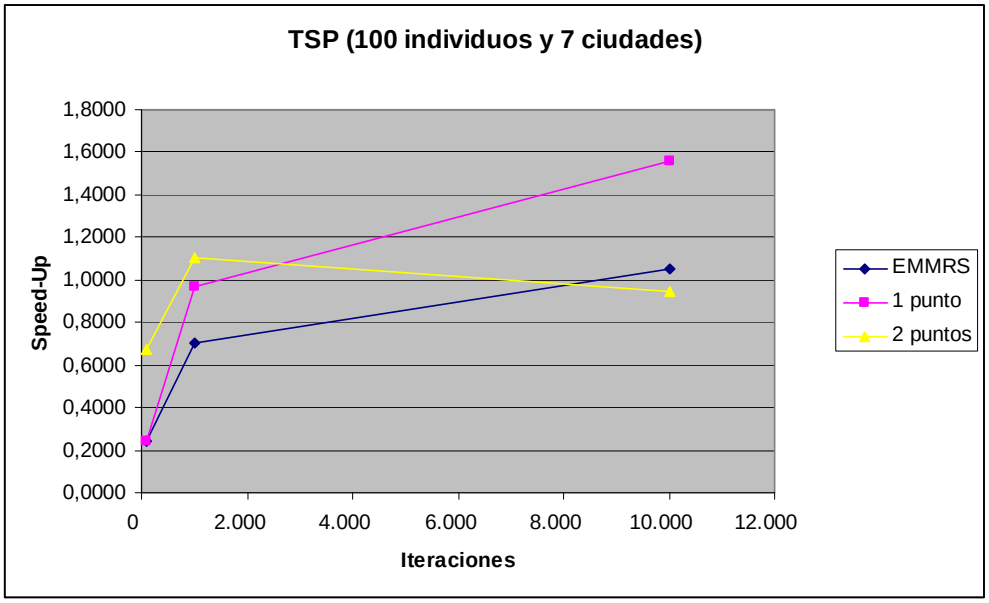
	100 iteraciones	1000 iteraciones	10000 iteraciones
EMMRS	0,2210	0,8211	1,1496
1 punto	0,3074	0,8885	1,3030
2 puntos	1,1151	1,3197	1,3348

Al igual que en el caso anterior, el ejemplo de 50 individuos con 11 ciudades, hace que las gráficas de *speed-up* sigan caminos muy similares, con unas ganancias del 15% para el EMMRS.



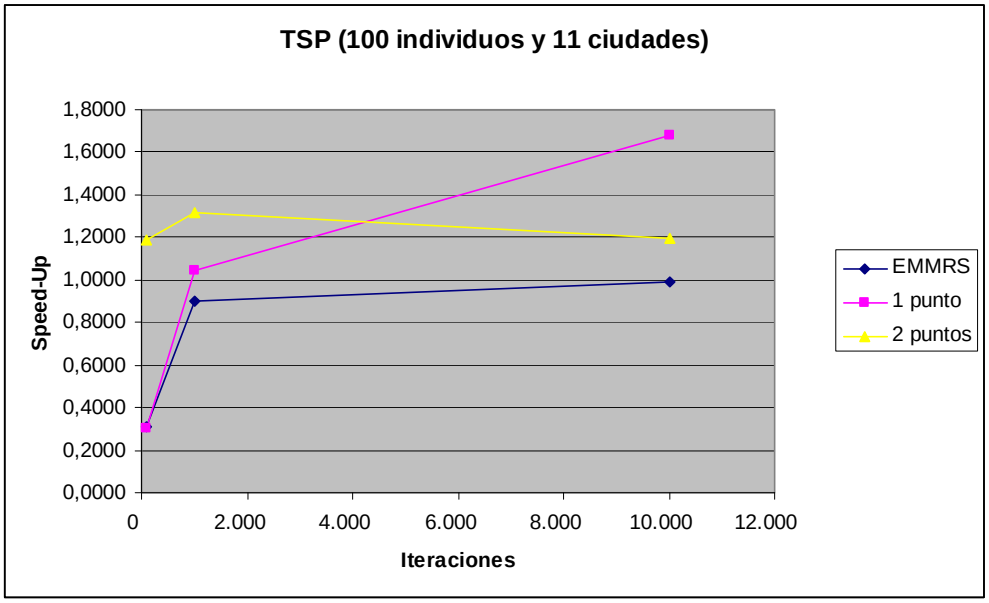
	100 iteraciones	1000 iteraciones	10000 iteraciones
EMMRS	0,3292	1,2272	1,3206
1 punto	0,3195	1,2666	1,4836
2 puntos	1,2067	1,7063	1,5684

En la gráfica que acompaña esta tabla podemos apreciar, nuevamente, como las líneas de ganancia en todos los casos son muy similares, exceptuando el caso del cruce en dos puntos que parte con una mejora del 20% respecto a su versión de CPU para pocas iteraciones (cien).



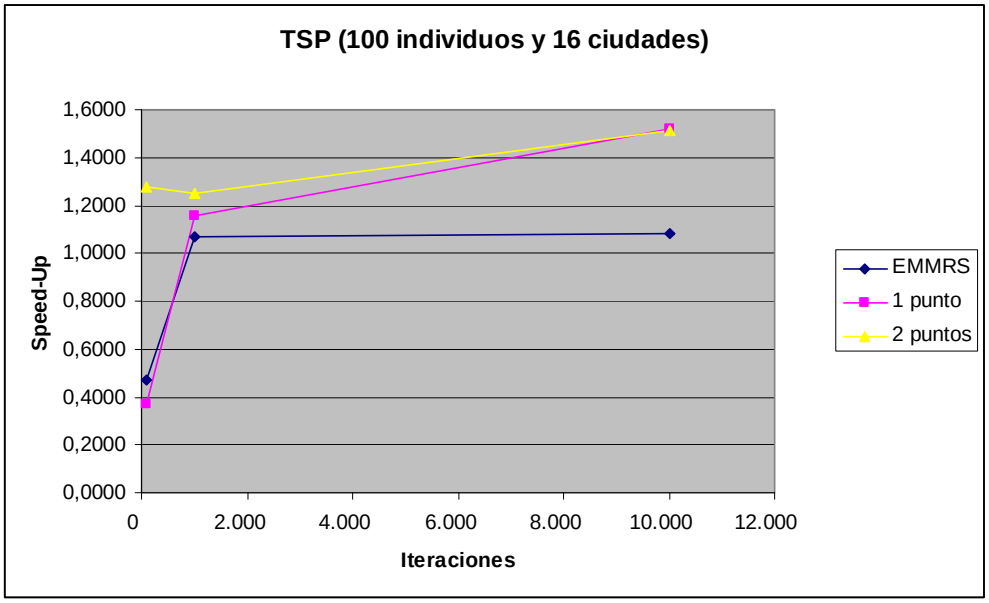
	100 iteraciones	1000 iteraciones	10000 iteraciones
EMMRS	0,2395	0,7016	1,0505
1 punto	0,2448	0,9658	1,5563
2 puntos	0,6711	1,1047	0,9491

Y llegamos a los ejemplos en los que aumentamos la población a 100 individuos. En esta ocasión, exceptuando al cruce en un punto que logra una mejora casi del 56%, los algoritmos apenas ofrecen mejoras respecto a sus versiones para CPU, esto puede ser debido como señalábamos antes, a una sobrecarga en las comunicaciones de datos entre la CPU y la GPU



	100 iteraciones	1000 iteraciones	10000 iteraciones
EMMRS	0,3092	0,8987	0,9873
1 punto	0,3016	1,0416	1,6764
2 puntos	1,1840	1,3139	1,1918

Los resultados para 16 ciudades siguen la línea continuista planteada en el anterior gráfico: una mejora a tener en cuenta por el algoritmo de cruce en un punto (que puede ser debida a una inicialización más óptima de sus valores) y un estancamiento por parte de los otros dos algoritmos en torno a 1.



	100 iteraciones	1000 iteraciones	10000 iteraciones
EMMRS	0,4702	1,0682	1,0851
1 punto	0,3682	1,1533	1,5178
2 puntos	1,2778	1,2526	1,5114

La última gráfica no hace si no incrementar nuestras sospechas de que se está produciendo una sobrecarga en la comunicación entre CPU y GPU en el caso del algoritmo EMMRS**,** lo que hace que éste se estanque en valores próximos a sus tiempos de ejecución en CPU.

8.5 Conclusiones y futuras líneas de estudio

Tras el análisis de los datos obtenidos gracias las numerosas pruebas realizadas podemos concluir que para la función de Schwefel los resultados tanto de calidad (cercanía al óptimo y escasa dispersión) como de tiempo (quizás no con una mejora espectacular, pero demostrándose más solvente que el resto de algoritmos implementados) hacen del EMMRS una herramienta a tener en cuenta en futuras implementaciones de problemas discretos

Las dudas nos vienen en lo referente al problema del viajante de comercio. Dudas en cuanto a sus tiempos de ejecución en comparación con su versión para CPU, ya que se mantiene muy próximo, lo que hace nos hace sospechar de un cuello de botella en la transferencia de datos entre CPU y GPU, lo que se traduciría en unos tiempos similares en ambas ejecuciones. En lo referente a la calidad, no nos cabe la menor duda de que EMMRS sigue proporcionando, al igual que con la función de Schwefel, unos resultados más constantes y agrupados en torno a unas medias de valores más cercanos al óptimo.

Queda pues, investigar más a fondo el tema de la sobrecarga en las comunicaciones CPU-GPU. Proponemos como posible camino a seguir el mantenimiento de dos poblaciones distintas, una en la CPU y otra en la GPU con intercambio de individuos cada cierto tiempo. Otro posible variante al trabajo sería una implementación del mismo explotando aún más el paralelismo a través de un algoritmo genético paralelo, en alguna de sus variantes (topología de anillo, maestro-esclavo, modelo de islas...) aprovechando de este modo al máximo el cluster K2

8 Bibliografía y Referencias

1 http://setiathome.ssl.berkeley.edu/

2 http://en.wikipedia.org/wiki/Thinking_Machines

3 http://en.wikipedia.org/wiki/Maspar

4 http://www.nvidia.com/page/home.html

5 http://www.nvidia.com/page/geforce8.html

6 J.I. Hidalgo, J.L. Risco, J. Lanchares and O. Garnica. Solving discrete deceptive problems with EMMRS. Proceedings of the 2008 Genetic and Evolutionary Computation Conference

7 Mitchell, Melanie. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

8 Koza, John, Martin Keane, Matthew Streeter, William Mydlowec, Jessen Yu y Guido Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

9 Goldberg, David. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

10 Holland, John. "Genetic algorithms" *Scientific American*, julio de 1992

11 Forrest, Stephanie. "Genetic algorithms: principles of natural selection applied to computation" *Science*, vol.261, (1993).

12 Koza, John, Forest Bennett, David Andre y Martin Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann Publishers, 1999.

13 Haupt, Randy y Sue Ellen Haupt. *Practical Genetic Algorithms*. John Wiley & Sons, 1998.

14 Coello, Carlos. "An updated survey of GA-based multiobjective optimization techniques". *ACM Computing Surveys*, vol.32, no.2, p.109-143 (junio de 2000).

15 Dawkins, Richard. *The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe Without Design*. W.W. Norton, 1996.

16 Haupt, Randy y Sue Ellen Haupt. *Practical Genetic Algorithms*. John Wiley & Sons, 1998.

17 Algoritmos genéticos y computación evolutiva. Nikos Drakos, Ross Moore

18 Soule, Terrence y Amy Ball. "A genetic algorithm with multiple reading frames". En *GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference*, Lee Spector y Eric Goodman (eds). Morgan Kaufmann, 2001.

19 Algoritmos Genéticos http://eddyalfaro.galeon.com/

20 Genetic Algorithms and Evolutionary Computation http://www.talkorigins.org/faqs/genalg/genalg.html

21 R. Baraglia, J.I. Hidalgo and R. Perego. "*A Hybrid Heuristic for the Traveling Salesman Problem*". IEEE Transactions Evolutionary Computation, Vol. 5. Nº6. Diciembre 2001.

Figura 1 Algoritmos Genéticos http://eddyalfaro.galeon.com/

Figura 2: Koza, John, Martin Keane y Matthew Streeter. "Evolving inventions"' *Scientific American*, febrero de 2003

Figura 3: Algoritmos Genéticos http://eddyalfaro.galeon.com/

Figura 5: J.I. Hidalgo, J.L. Risco, J. Lanchares and O. Garnica. Solving discrete deceptive problems with EMMRS. Proceedings of the 2008 Genetic and Evolutionary Computation Conference

Figura 7: Implentación de un algoritmo genético paralelo sobre HW grafico de último generación, Carmen Córdoba González y Juan Carlos Pedraz Sánchez. Dirigido por José Ignacio Hidalgo Pérez

Figura 10: J.I. Hidalgo, J.L. Risco, J. Lanchares and O. Garnica. Solving discrete deceptive problems with EMMRS. Proceedings of the 2008 Genetic and Evolutionary Computation Conference

Figura 12 http://www.geeks3d.com/public/common/CUDA-logo-jegx.jpg

Figura 14 http://www.nvidia.com

Figura 15: http://img.chw.net/sitio/stories/tecnico/1213415850_transistores.jpg

Figuras: 16-17-19-21-22: http://www.nvidia.com

Figura 23: http://en.wikipedia.org/wiki/NP-hard

Figura 24 Estudio de la tolerancia a fallos en Algoritmos Genéticos Paralelos. A. Barroso Cuarental, D. Boíllos Fernández y A. B. Jerónimo Pérez. Dirigido por José Ignacio Hidalgo Pérez

Figura 25: http://es.wikipedia.org/wiki/Archivo:Hamiltonian_path.svg

Figura 26: http://artecs.dacya.ucm.es/atc/homepage.php

Figuras 27-28: http://www.nvidia.com

10 Apéndice A: TSP gr48

Como caso especial de estudio, hemos aplicado nuestro algoritmo genético con EMMRS y Hill Climbing a un modelo conocido para evaluar tanto los tiempos de respuesta como la aproximación al óptimo.

A continuación, incluimos una tabla en la que se pueden ver los resultados obtenidos. El estudio está realizado para poblaciones de 500 individuos y para 30.000, 20.000 y 10.000 iteraciones. Cada una de las filas de la tabla representa los resultados de una muestra de 10 ejecuciones del algoritmo, de las cuales, mostramos las medias (tanto de tiempo de ejecución como de resultado) y el mejor valor obtenido.

TSP Gr48	Número de individuos	Número de iteraciones	Función de ajuste	Tiempo (en segundos)
	500	30.000	Mejor: 5050 Media: 5502	484
	500	30.000	Mejor: 5063 Media: 5482	482
	500	20.000	Mejor: 5286 Media: 5592	320
	500	20.000	Mejor: 5143 Media; 5398	347
	500	10.000	Mejor: 5257 Media: 5640	166
	500	10.000	Mejor: 5046 Media: 5663	180

Los resultados obtenidos no son tan favorables como los ejemplo que hemos usado de referencia[21], pero de todos modos podemos ver que el algoritmo se aproxima de forma considerable a la solución del problema (5046), cuando se ejecuta un número elevado (30.000).

11 Apéndice B: Código función de Schwefel

```
#include <stdio.h>     //Printf
#include <stdlib.h>    //Rand
#include <time.h>      //Inicializacion de la semilla y toma de tiempos
#include <math.h>      //Función seno
#include <cutil.h>

#define NumInd 100  //Definimos el numero de individuos que formaran la generación

typedef struct Chrom
  {
        int bit[160];
        int map[160];
        double fit;
  }chrom;

//Estructura del gen auxiliar, usado para calcular el fitness.

typedef struct Estruct_gen{
        int bit[160];
} estruct_gen;

//Cabeceras de las funciones que usaremos

void *evpop(chrom popcurrent[NumInd]);
double x(int gen[16]);
double y(double x);
estruct_gen mapping (chrom EMMRS);
void *sampling  (chrom popnext[NumInd]);
void *pickchroms(chrom popnext[NumInd]);
void *crossover (chrom popnext[NumInd]);
void *mutation  (chrom popnext[NumInd]);
double calfunc (estruct_gen popcurrent);
double calfunc2 (chrom popnext);




__global__ void cu_sampling (int *al1,int *al2,chrom *popnext){
```

```
//Empleamos un vector auxiliar puesto auxpop, de esta manera. Comparamos los dos
cromosomas que nos indica el vector al1 y al2 (vectores inicializados previamente de
manera aleatoria).

        chrom auxpop;

//Metemos el mejor de los dos a partir de la segunda posición

        if ((popnext[al1[threadIdx.x]].fit)<(popnext[al2[threadIdx.x]].fit)){
                auxpop= popnext[al1[threadIdx.x]];}

        else{auxpop= popnext[al2[threadIdx.x]];}

        syncthreads();
        popnext[(threadIdx.x)+2]=auxpop;

}

__global__ void cu_MMRS (int *al1,int *al2,Chrom *popnexto){

        int cruce,randomaux,random,auxmap,auxmap1,aux;


//Con esta instrucción hacemos que sólo coja los individuos pares a partir del segundo,
excepto el último.

        if ((blockIdx.y % 2 == 0)&&(blockIdx.y>1)&&(blockIdx.x<96)){

//Generamos dos puntos de cruce aleatorios y nos aseguramos que la posición menor
está ubicada en randomaux.

                if (al1[((blockIdx.y/2)-1)]< al2[((blockIdx.y/2)-1)]) {
                        random=al2[((blockIdx.y/2)-1)];
                        randomaux=al1[((blockIdx.y/2)-1)];}
                else{
                        randomaux=al2[((blockIdx.y/2)-1)];
                        random=al1[((blockIdx.y/2)-1)];
                }

                cruce=(al2[((blockIdx.y/2)-1)]*(NumInd/160));

//Usamos el propio aleatorio de al1 para ver si cruza o no

                if (cruce<81){
```

```
//Aquí comienza a trabajar el operador EMMRS

                if ((threadIdx.x>=randomaux)&&(threadIdx.x<=random)){

                        auxmap = popnexto[blockIdx.y].map[threadIdx.x];
                        auxmap1= popnexto[blockIdx.y+1].map[threadIdx.x];

//Utilizamos una barrera para sincronizar los hilos y evitar accesos con información
erronea
                        syncthreads();
                        popnexto[blockIdx.y].map[threadIdx.x]=auxmap1;
                        popnexto[blockIdx.y+1].map[threadIdx.x]=auxmap;
                        syncthreads();

//Con una probabilidad baja (2%), ocurre una mutación en el individuo.

                        if (cruce<2){
                                popnexto[blockIdx.y+1].map[randomaux]
                                                = (al2[((blockIdx.y/2)-1)]%16);}
                }
        }

//Reemplazo

popnexto[blockIdx.y+1].map[random]=popnexto[blockIdx.y+1].map[randomaux];

//En el segundo cromosoma, el valor que hay en la posición randomaux se duplica en la
posición random

//Shift

if((threadIdx.x>=randomaux)&&(threadIdx.x<random)){

        aux=popnexto[blockIdx.y+1].map[threadIdx.x];
        syncthreads();
        popnexto[blockIdx.y+1].map[threadIdx.x+1]=aux;
        syncthreads();
                }

//Guardamos en una variable auxiliar el valor del fenotipo en la posición más alta de
cruce (cuyo valor después de la fase replacement es igual al de la posición menor)
y vamos desplazando valores hacia la derecha entre esos dos extremos. Al final,
sobrescribimos el último valor con el de la variable auxiliar.
   }
```

```
}
//Main
int main()   {

//Variables de control
int num,i,j,k,l;

// Variable auxiliares, incluidas las dos generaciones que usaremos: la actual
(popcurrent) y la siguiente (popnext)

chrom aux_current, popcurrent[NumInd], popnext[NumInd];

//Inicializamos la semilla del random y obtenemos una variable para las medidas de
tiempo

srand((unsigned)time(0));
unsigned int timer=0;

//Introducción por teclado de las iteraciones que deseamos que realice el algoritmo

printf("\nALGORITMO GENÉTICO\nFUNCIÓN DE SCHWEFEL y= -x * sin (sqrt
(abs (x)))\n");
  enter: printf("\nNumero de iteraciones:  ");
  scanf("%d",&num);

  if (num<1) {
  printf("numero no valido");
  goto enter;}

// Hasta que no sea un número valido, repetimos.

//Inicializamos  la generación actual de manera aleatoria

  printf ("\nEVOPOP    ");
  evpop(popcurrent);

//Comenzamos la mediación de tiempo de ejecución del algoritmo

  cutCreateTimer(&timer);
  cutStartTimer(timer);



//Ejecutamos el algoritmo el número de veces deseado
```

```
  for(i=0;i<num;i++){

    printf("\n\n Iteracion número = %d\n",i);

//Copiamos la generación actual para que sirva de semilla
    for (j=0;j<NumInd;j++) popnext[j]=popcurrent[j];

// Escogemos los mejores cromosomas a la siguiente

   pickchroms(popnext);

//Aleatoriamente escogemos dos individuos que mediante torneo compiten por formar
parte de la nueva generación

   sampling (popnext);

//Cruzamos los anteriores para obtener sus hijos (EMMRS)

   crossover(popnext);

//Para cada individuo calculamos su fitness y hacemos que ese individuo de la
generación siguiente pase a formar parte de la actual.


    for (j=0;j<NumInd;j++){
      popnext[j].fit = calfunc (mapping(popnext[j]));
      popcurrent[j]=popnext[j];

// Cada 1000 iteraciones hacemos Hill Climbing para evitar caer en óptimos locales.

      if ((i+1)%1000==0){
            k=0;
            aux_current.fit =  popcurrent[j].fit;

//Guardamos el fitness actual y hacemos 250 intentos de mejora por cada individuo

            while (k<250){
               aux_current =  popcurrent[j];
               for (l=0;l<16;l++) {

//Cambiamos el valor aleatoriamente de 16 posiciones (también aleatorias) del map del
individuo...
                     aux_current.map[rand()%160]= rand()%16;
                     aux_current.fit= calfunc (mapping (aux_current));
```

```
//...calculamos su nuevo fitness, y si mejora lo sustituimos.

                    if (aux_current.fit <= popcurrent[j].fit) popcurrent[j] = aux_current;
                    }
                 k++;
               }

        }
    }
}

//Mostramos por pantalla los 3 mejores de la generación…

for (k=0;k<3;k++) printf ("\nINDIVIDUO NÚMERO %d SU FITNESS ES
%f",k,popnext[k].fit);


// … y el tiempo total de cómputo

cutStopTimer(timer);
printf("\nTiempo en realizar todas las iteraciones  %f ms)\n",cutGetTimerValue(timer));
cutDeleteTimer(timer);
fflush(stdin);
return (0);
} //MAIN


//Inicializa aleatoriamente la generación de individuos (genotipos y fenotipos)

void *evpop(chrom popcurrent[NumInd])
{

 int j,i,valor,valor2;
 srand((unsigned)time(0));

//Recorremos la generación de individuos

  for (j=0;j<NumInd;j++) {

// Recorremos los genes

      for (i=0; i<160; i++){

//Tomamos valores aleatorios para el fenotipo y el genotipo
```

```
        valor= rand();
        valor2= rand();
        popcurrent[j].bit[i]=valor2%2
        popcurrent[j].map[i]=valor%16;
      }

//Calculamos el fitness de cada individuo

  popcurrent[j].fit= calfunc (mapping (popcurrent[j]));
  }
  return(0);
}//Fin evpop


//La función de ajuste (fitness), en este caso Schwefel

double y (double EXE) {
double y;
double aux=EXE;
if (EXE<0) {aux= -EXE;}

// Función de Schwefel, normalizada para que el óptimo sea 0.

y= -EXE * sin (sqrt (aux))+418.9828872721624;
return(y);
}// Fin de la funcion de fitness


// Devuelve los cromosomas ordenados de mayor a menor fitness

void *pickchroms(chrom popnext[NumInd])  {
 int i,j;
 chrom temp;
 srand((unsigned)time(0));

//Va comparando los cromosomas e intercambiando en función del valor del fitness

 for(i=0;i<(NumInd-1);i++)
    for(j=0;j<(NumInd-1);j++){
        if (popnext[j+1].fit<popnext[j].fit) {
                temp=popnext[j+1];
                popnext[j+1]=popnext[j];
                popnext[j]=temp;
```

```
            }
        }
  fflush(stdin);
  return(0);
}//Fin pickchroms


double x (int gen[16]){

//Dado un gen de 16 bits devuelve el valor que codifica, con la representación que
hemos elegido

   int p,j;
   double ent=0;
   double numero,dec=0;

   for (j=1; j<10;j++)    ent=ent+pow((float)2,j-1)*gen[j];
   for (p=10; p<16; p++)  dec=dec+(1 / pow((float)2,(p-9)))*gen[p];

   numero= dec + ent;
   if (gen[0]==1) numero=numero*(-1);

   return numero;

}//Fin x


//Dado un cromosoma (una vez aplicado el fenotipo al genotipo), esta función calcula su
valor

double calfunc (estruct_gen popcurrent){

 int k,dim,gen[16];
 double auxfit=0;

       for (dim=0;dim<10;dim++){
           for (k=0;k<16;k++) {gen[k] = popcurrent.bit[k+dim*16];}
           auxfit = y(x(gen))+ auxfit;
       }
      return (auxfit);

}//Fin calfunc


//Devuelve un único array que es el resultado de aplicar el fenotipo al genotipo.
```

```
estruct_gen mapping (chrom EMMRS){
 int k;
 estruct_gen aux;
 for (k=0;k<160;k++){aux.bit[k] = EMMRS.bit[EMMRS.map[k]]; }
 return aux;
}//Fin mapping


//Este operador es el encargado de hacer el cruzamiento y generar los nuevos individuos
siguiendo el esquema del EMMRS, sobre los individuo resultante de aplicar el fenotipo
al genotipo de cada cromosoma de la generación

void *crossover(chrom popnext[NumInd]){

   int *al1_d, *al2_d;
   chrom *popnext_d;
   int al1[((NumInd-2)/2)];
   int al2[((NumInd-2)/2)];
   srand((unsigned)time(0));

//Reservamos este tamaño de memoria (NumInd-2/2) para al1 y 2 ya que cada vez que
usa un hilo lo hace con dos individuos

   cudaMalloc((void **) &al1_d, (((NumInd-2)/2)*sizeof(int)));
   cudaMalloc((void **) &al2_d, (((NumInd-2)/2)*sizeof(int)));
   cudaMalloc((void **) &popnext_d, (NumInd*sizeof(chrom)));

//Creamos los vectores de números aleatorios comprendidos entre 1 y 160

   for (int j=0;j<((NumInd-2)/2);j=j++){
       al1[j]=(rand()%160);
       al2[j]=(rand()%160);
    }

   dim3 dimBlock(16,1);        //Esto es como esta definido el cromosoma
   dim3 dimGrid(1,NumInd); //Esto es el numero de individuos

cudaMemcpy(al1_d, al1, ((NumInd-2)/2)*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(al2_d, al2, ((NumInd-2)/2)*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(popnext_d, popnext, NumInd*sizeof(chrom),
                                                        cudaMemcpyHostToDevice);


cu_MMRS<<< dimGrid, dimBlock >>> (al1_d,al2_d,popnext_d);
```

```
cudaMemcpy(al1, al1_d, ((NumInd-2)/2)*sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(al2, al2_d, ((NumInd-2)/2)*sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(popnext, popnext_d, NumInd*sizeof(chrom),
                                                    cudaMemcpyDeviceToHost);
//Liberamos la memoria

cudaFree(al1_d);
cudaFree(al2_d);
cudaFree(popnext_d);
return(0);

} //Fin crossover


//Este procedimiento se encarga de obtener la nueva generación mediante cruces
aleatorios entre los individuos existentes. Cabe mencionar que los dos primeros
individuos de cada generación (los dos que tiene el mejor fitness) pasan directamente a
la siguiente. El resto se selecciona aleatoriamente dos a dos y de cada pareja de escoge
el mejor (torneo)

void *sampling  (chrom popnext[NumInd]){


  int *al1_d;
  int *al2_d;
  chrom *popnext_d;
  int al1[(NumInd-2)];
  int al2[(NumInd-2)];
  srand((unsigned)time(0));

  cudaMalloc((void **) &al1_d, (NumInd-2)*sizeof(int));
  cudaMalloc((void **) &al2_d, (NumInd-2)*sizeof(int));
  cudaMalloc((void **) &popnext_d, NumInd*sizeof(chrom));

//Aquí creamos un array de números aleatorios entre 2 y 9 para utilizar posteriormente
en la GPU

  for (int j=0;j<((NumInd-2));j=j++){
      al1[j]=(rand()%NumInd);
      al2[j]=(rand()%NumInd);
   }

  dim3 dimBlock((NumInd-2),1); //Esto es como esta definido el cromosoma
  dim3 dimGrid(1,1); //Esto es el numero de individuos
  cudaMemcpy(al1_d, al1, (NumInd-2)*sizeof(int), cudaMemcpyHostToDevice);
```

```
  cudaMemcpy(al2_d, al2, (NumInd-2)*sizeof(int),cudaMemcpyHostToDevice);
  cudaMemcpy(popnext_d, popnext, numInd*sizeof(chrom),
                                                    cudaMemcpyHostToDevice);

  cu_sampling<<< dimGrid, dimBlock >>> (al1_d,al2_d,popnext_d);
  cudaMemcpy(al1, al1_d, (NumInd-2)*sizeof(int), cudaMemcpyDeviceToHost);
  cudaMemcpy(al2, al2_d, (NumInd-2)*sizeof(int), cudaMemcpyDeviceToHost);
  cudaMemcpy(popnext, popnext_d, NumInd*sizeof(chrom),
                                                    cudaMemcpyDeviceToHost);


  cudaFree(al1_d);
  cudaFree(al2_d);
  cudaFree(popnext_d);

return(0);

}//Fin Sampling
```

12 Apéndice C: Código TSP

```
#include <stdio.h>     //Printf
#include <stdlib.h>    //Rand
#include <time.h>      //Inicialización de la semilla y toma de tiempos
#include <cutil.h>

#define MAXINT 67000
#define NumCiu 10
#define NumInd 25

typedef struct Chrom
  {
    int bit[NumCiu];
    int map[NumCiu];
    int fit;
  }chrom;
// Estructura de los cromosomas

//Definimos un valor máximo para la distancia entre dos ciudades en caso de que no
haya camino

void *evpop(chrom popcurrent[NumInd], int distancias [NumCiu][NumCiu]);
void *sampling  (chrom popnext[NumInd]);
void *pickchroms(chrom popnext[NumInd]);
void *crossover (chrom popnext[NumInd]);
void *duplicados (chrom popnext[NumInd]);
int calculafit (chrom individuo, int distancias[NumCiu][NumCiu]);
void *HillClimbing1 (chrom popnext[NumInd], int distancias[NumCiu][NumCiu]);
void *intervalo (chrom popnext[NumInd]);


//Cabeceras de las funciones que usaremos



__global__ void cu_MMRS (int *al1,int *al2,chrom *popnext){

        int cruce,randomaux,random,auxmap,auxmap1,aux;
```

```
//Con este if hacemos que sólo coja individuos pares a partir del segundo, excepto el
último

        if ((blockIdx.y % 2 == 0)&&(blockIdx.y>=2)&&(blockIdx.y!=(NumInd-1))){

                if (al1[((blockIdx.y/2)-1)]< al2[((blockIdx.y/2)-1)]) {
                        cruce=al2[((blockIdx.y/2)-1)];
                        randomaux=al1[((blockIdx.y/2)-1)];
                        random=cruce;
                }
                else{
                        randomaux=al2[((blockIdx.y/2)-1)];
                        random=al1[((blockIdx.y/2)-1)];
                }

                //Generamos dos puntos de cruce aleatorios y nos aseguramos que la
posición menor está guardada en randomaux

                cruce = (al2[((blockIdx.y/2)-1)]*(100/NumCiu));

 //Usamos el propio aleatorio de al1 para ver si cruza o no

                if (cruce<81){


                        //Esto es MMRS

                        if((threadIdx.x>=randomaux)&&(threadIdx.x<=random)){
                                auxmap = popnext[blockIdx.y].map[threadIdx.x];
                                auxmap1= popnext[blockIdx.y+1].map[threadIdx.x];
                                syncthreads();
                               popnext[blockIdx.y].map[threadIdx.x]=auxmap1;
                                popnext[blockIdx.y+1].map[threadIdx.x]=auxmap;
                                syncthreads();

                    //MUTACION
                                if (cruce<2){

                               //Aqui cruza si %2 y le metemos el numero aleatorio del al2

                               popnext[blockIdx.y+1].map[randomaux] =al2[threadIdx.x];
                                 }

                        }
                }
```

```
                //... WITH REPLACEMENT

        popnext[blockIdx.y+1].map[random]=popnext[blockIdx.y+1].map[randomaux];

//En el segundo cromosoma, el valor que hay en la posición randomaux se duplica en la
posición random

                //... AND SHIFT

                if((threadIdx.x>=randomaux)&&(threadIdx.x<random)){
                        aux=popnext[blockIdx.y+1].map[threadIdx.x];
                        syncthreads();
                        popnext[blockIdx.y+1].map[threadIdx.x+1]=aux;
                        syncthreads();
                }

                //Guardamos en una variable auxiliar el valor del fenotipo en la posición
más alta de cruce (cuyo valor después de la fase REPLACEMENT es igual al de la
posición menor) y vamos desplazando valores hacia la derecha entre esos dos extremos.
Al final, sobrescribimos el último valor con el de la variable auxiliar.
   }
}

__global__ void cu_sampling (int *al1,int *al2,chrom *popnext){

        chrom auxpop;

        //Comparamos los dos cromosomas que nos indica el vector al1 y al2

        if ((popnext[al1[threadIdx.x]].fit)<(popnext[al2[threadIdx.x]].fit)){

                //Metemos el mejor de los dos a partir de la segunda posición

                auxpop= popnext[al1[threadIdx.x]];
        }
        else{
                auxpop= popnext[al2[threadIdx.x]];
        }
        syncthreads();


        popnext[(threadIdx.x)+2]=auxpop;

}
int main()
```

```
//MAIN
{
  int num,i,j,is;

  //Declaración de variables de control

  chrom popcurrent[NumInd], popnext[NumInd];

  //Declaración de la generación de individuos actual y siguiente, así como un individuo
auxiliar.

  int distancias [NumCiu][NumCiu];

  //Declaramos la matriz de distancias entre ciudades

  srand((unsigned)time(0));

  //Inicializamos el random

  unsigned int timer = 0;
  printf("\nALGORITMO GENETICO\nTraveling Salesman Problem\n");
  enter: printf("\nNumero de iteraciones:  ");
  scanf("%d",&num);

  //Recogemos por teclado las iteraciones


  if (num<1) {
    printf("numero no valido");
    goto enter;}

  //Hasta que no sea un número válido, repetimos


  printf ("\nEVOPOP    \n");
  evpop(popcurrent,distancias);

  //Inicializamos

  printf ("\nPICKCHROMS        \n");
  pickchroms(popnext);

 //Escogemos los mejores cromosomas a la siguiente
```

```
 cutCreateTimer(&timer);
 cutStartTimer(timer);

  for (is=0;is<num;is++){

  //Ejecutamos el algoritmo el numero de veces deseado

    printf("\n\n Iteracion numero = %d\n",is);

   for (int p=0;p<NumInd;p++) {popnext[p]=popcurrent[p];}

//Copiamos la generación actual para que sirva de semilla. Lo hace bien

   printf ("\nSAMPLING       \n");
   sampling (popnext);

   //Aleatoriamente escogemos dos individuos que mediante torneo compiten
   //por formar parte de la nueva generación

   intervalo(popnext);
   printf ("\nCROSSOVER       \n");
   crossover(popnext);

   //Cruzamos los anteriores para obtener sus hijos

   if ((is+1)%500==0){
                printf("\nHILLCLIMBING       \n");
                HillClimbing1 (popcurrent,distancias);}

   //En el caso de que hayamos hecho 500 iteraciones y para pervenir estancamiento en
los valores, llamamos a una función Hill Climbing

  //Calculamos los fitnesss de los individuos y actualizamos popcurrent

  for (j=0; j<NumInd;j++){
     popnext[j].fit = calculafit(popnext[j],distancias);
     popcurrent[j]=popnext[j];
   }

  //Calculamos el fitness de la generación y actualizamos con ambos la generación
semilla de la siguiente iteración

  pickchroms(popcurrent);
  printf ("\nPICKCHROMS       ");
```

```
  //Tras la llamada a HC y el cálculo nuevo de los valores del fitness deberemos
reordenar nuevamente los individuos en función de su fitness

  for (int j=0; j<2;j++){
        printf("\n    %d \n",popcurrent[j].fit);
        for (i=0;i<NumCiu;i++){printf ("%d  ",popnext[j].bit[i]);}
        printf("\n");
        for (i=0;i<NumCiu;i++){printf ("%d  ",popnext[j].map[i]);}

    }
  //Muestra por pantalla los j primeros cromosomas y/o su fitness

  }
cutStopTimer(timer);
printf("\nTiempo en realizar todas las iteraciones%f (ms) \n",cutGetTimerValue(timer));
cutDeleteTimer(timer);
fflush(stdin);
return 0;

}
//Fin del MAIN


void *evpop(chrom popcurrent[NumInd], int distancias [NumCiu][NumCiu])

//Inicializa tanto la matriz de distancias como la generación de individuos (genotipos y
fenotipos)
{
 int k,i,l;
 int rex[NumCiu],rex2[NumCiu];
 srand((unsigned)time(0));

//Declaración de variables auxiliares

for(i=0;i<NumCiu;i++){
     for (int j=0;j<=i;j++) {
//Recorremos la matriz

        if (i==j) distancias[i][j] = 0;

       //Si las coordenadas son iguales (diagonal principal) ponemos un cero porque
desplazarse de una ciudad a ella misma consideramos que tiene coste 0.

             else if ((i==j+1) || (j==i+1)) {distancias[i][j] = 1;distancias[j][i]=1;}
```

```
            //Si las ciudades que estamos considerando son continuas, esto es, su número
es inmediatamente posterior o anterior al de otra dada, tomamos como distancia entre
ambas 1. Esto lo hacemos para asegurarnos que al menos hay una ruta mínima, de la
cual conocemos su tamaño (Tamaño de la ruta mínima = n*1, siendo n=número de
ciudades) y la ruta será del estilo 0 - 1 - 2 - 3 -...- n

            else {
                l = rand()%100;
                if (l>90) {
                    distancias [i][j] = MAXINT;
                    distancias [j][i] = MAXINT;
                    }

//Si no es así tenemos dos opciones, en función de un valor aleatorio:

1.- Si no hay camino colocamos una constante definida como MAXINT, para que nos
de un coste inaceptable y así poder descartar esa ruta.

2.-Colocar en esa posición de la matriz (y en su simétrica) un valor aleatorio entre uno y
un valor que escojamos (inicialmente 50).

                else{
                    int p=rand()%50+1;
                    distancias [i][j] = p;
                    distancias [j][i] = p;
                    }
            }


        }
    }
     distancias[0][NumCiu-1] = 1;
     distancias[NumCiu-1][0] = 1;

//Añadimos estas dos inicializaciones a la matriz, porque el TSP debe retornar a la
ciudad de partida y de esta manera nos aseguramos que el camino de la última ciudad a
la primera es 1.

//Llegados a este punto, hemos inicializado la matriz

//Declaramos dos vectores auxiliares y guardamos en cada uno de ellos su posición,
excepto en las dos primeras que introducimos -1, para que no se vuelvan a repetir

for (int j=0;j<NumInd;j++){
```

```
rex[0]=-1;
for (int i=1;i<NumCiu; i++) {
        rex[i]= i;
}



for (int i=0; i<NumCiu; i++){

// Inicializamos el genotipo (individuo.bit) aleatoriamente y sin repeticiones

  if (i==0) {popcurrent[j].bit[i]=0;}

//Como vamos a hallar la ruta óptima que empieza en la primera ciudad, en caso de que
estemos inicializando la primera posición de cada individuo va a contener esa primera
ciudad

            else{
                k = rand()%NumCiu;

//Si no es la primera ciudad, hacemos el random sobre las ciudades y si se repite alguna
en el vector auxiliar rex, tendremos una marca (-1) que nos indicará que esa ciudad ya
ha sido insertada en la ruta actual. Repetimos el algoritmo hasta que encontremos una
ciudad que no haya sido incluida.

                while (rex[k]==-1)  k = rand()%NumCiu;
                    popcurrent[j].bit[i]= rex[k];
                    rex [k]=-1;

//Cuando la encontremos, actualizamos su posición en el vector rex introduciendo un -1
que nos indicará en las siguiente pasadas que ya ha sido incluida

//Reiniciamos el vector auxiliar de forma paralela

                    if (i==NumCiu-1){
                        rex[0]=-1;
                        for (int i=1;i<NumCiu; i++) {
                            rex2[i]= i;
                        }

                    }
```

```
//Si hemos alcanzado la última iteración (hemos generado una ruta que pasa por todas
las ciudades) reiniciamos las posciones del vector auxiliar (excepto la de la primera
ciudad)
                        }

                }

                rex2[0]=-1;
                for (int i=1;i<NumCiu; i++) {
                        rex2[i]= i;
                }
        for (int i=0;i<NumCiu;i++){

//Análogamente para el fenotipo (individuo.map)

                  if (i==0) popcurrent[j].map[i]=0;
                  else{
                        k = rand()%NumCiu;
                        while (rex2[k]==-1)  k = rand()%NumCiu;
                        popcurrent[j].map[i]= rex2[k];
                        rex2[k]=-1;
                        if (i==NumCiu-1) {
                        rex2[0]=-1;
                      for (int i=1;i<NumCiu; i++) {
                                      rex2[i]= i;
                              }

                        }
                      }
                }
                printf("%d\n",j);
  }

//Esto imprime la matriz de distancias

        for (int u=0;u<NumCiu;u++){
                for (int p=0;p<NumCiu;p++){
                        printf("%d ",distancias[u][p]);
                        }
                printf("\n");
        }

//Aquí lo que hacemos es calcular el fitness

  for (int j=0;j<NumInd;j++) {
```

```
      popcurrent[j].fit = calculafit(popcurrent[j], distancias);

//Para cada individuo llamamos a la función calculafit y obtenemos su fitness

      printf("\n%d  ",popcurrent[j].fit);

      for (int i=0;i<NumCiu;i++) printf ("%d  ",popcurrent[j].bit[popcurrent[j].map[i]]);

//Mostramos por pantalla el genotipo del individuo y su fitness
         }
  return(0);
}

//Fin EVPOP



void *pickchroms(chrom popnext[NumInd])

//Devuelve los cromosomas ordenados de mayor a menor fitness
{

 chrom temp;

 //cromosoma auxiliar

 srand ((unsigned)time(0));
 for (int i=0;i<NumInd;i++)

//Va comparando los cromosomas e intercambiando en función del valor del fitness

      for (int j=0;j<NumInd-1;j++)
       {
          if (popnext[j+1].fit<popnext[j].fit){

              temp=popnext[j+1];
              popnext[j+1]=popnext[j];
              popnext[j]=temp;
          }
       }
  fflush(stdin);
  return(0);
}
//Fin PICKCHROMS
```

```
void *crossover(chrom popnext[NumInd]){

 //Es el encargado de hacer los nuevos cruces y generar los nuevos individuos
siguiendo el esquema del EMMRS, aplicado al individuo resultante de aplicar el
fenotipo al genotipo de cada cromosoma de la generación popnext

 int *al1_d;
 int *al2_d;
 chrom *popnext_d;
 int al1[(NumInd-2/2)];
 int al2[(NumInd-2/2)];
 srand((unsigned)time(0));

//Reservamos esta memoria (NumInd-2/2) para al1 y 2 ya que el cada ves que usa un
hilo lo hace con dos individuos

 cudaMalloc((void **) &al1_d, ((NumInd-2)/2)*sizeof(int));
 cudaMalloc((void **) &al2_d, ((NumInd-2)/2)*sizeof(int));
 cudaMalloc((void **) &popnext_d, NumInd*sizeof(chrom));

//Reserva memoria del dispositivo

 for (int j=0;j<((NumInd-2)/2);j=j++){

//Aquí creamos un array de números aleatorios entre 1 y NumCiu-1

      al1[j]=(rand()%(NumCiu-1))+1;
      al2[j]=(rand()%(NumCiu-1))+1;
   }

//Las columnas van primero. Tenemos NumCiu como columnas y una fila que es el
cromosoma

 dim3 dimBlock(NumCiu,1);
// De esta manera queda definido el cromosoma, con el numero de threads por
individuos

 dim3 dimGrid(10,NumInd);

//Con esta instrucción ligamos al grid el número de bloques

 cudaMemcpy(al1_d, al1, ((NumInd-2)/2)*sizeof(int), cudaMemcpyHostToDevice);
 cudaMemcpy(al2_d, al2, ((NumInd-2)/2)*sizeof(int),cudaMemcpyHostToDevice);
 cudaMemcpy(popnext_d,
```

```
                    popnext,NumInd*sizeof(chrom),cudaMemcpyHostToDevice);

 cu_MMRS<<< dimGrid, dimBlock >>> (al1_d,al2_d,popnext_d);
 cudaMemcpy(al1, al1_d, ((NumInd-2)/2)*sizeof(int), cudaMemcpyDeviceToHost);
 cudaMemcpy(al2, al2_d, ((NumInd-2)/2)*sizeof(int), cudaMemcpyDeviceToHost);
 cudaMemcpy(popnext, popnext_d, NumInd*sizeof(chrom),
                                                    cudaMemcpyDeviceToHost);

//Eliminamos los posibles duplicados que se hayan podido crear

 duplicados (popnext);

//Para obtener un recorrido valido, que se ajuste al algoritmo TSP, debemos pasar una
sola vez por cada ciudad. Después de haber aplicado el EMMRS, obviamente tendremos
repeticiones en el array correspondiente al fenotipo, es por ello que debemos eliminar
los valores repetidos.

 cudaFree(al1_d);
 cudaFree(al2_d);
 cudaFree(popnext_d);

 return(0);} //FIN CROSSOVER

void *sampling  (chrom popnext[NumInd]){

//Este procedimiento se encarga de obtener la nueva generación mediante cruces
aleatorios entre los individuos existentes. Cabe mencionar que los dos primeros
individuos de cada generación (los dos que tiene el mejor fitness) pasan directamente a
la siguiente. El resto se seleccionan aleatoriamente dos a dos y de cada pareja de escoge
el mejor (TORNEO)


 int *al1_d;
 int *al2_d;
 chrom *popnext_d;
 int al1[NumInd-2];
 int al2[NumInd-2];
 srand((unsigned)time(0));

 cudaMalloc((void **) &al1_d, (NumInd-2)*sizeof(int));
 cudaMalloc((void **) &al2_d, (NumInd-2)*sizeof(int));
 cudaMalloc((void **) &popnext_d, NumInd*sizeof(chrom));

 for (int j=0;j<(NumInd-2);j=j++){
```

```
//Aquí se crea un array de números aleatorios para poder pasarlo como parámetro a la
tarjeta gráfica

 al1[j]=(rand()%NumInd);
 al2[j]=(rand()%NumInd);
}

//Las columnas van primero. Tenemos NumCiu como el máximo valor de las columnas
y como fila, el cromosoma

 dim3 dimBlock(NumInd-2,1);         //Esto es como esta definido el cromosoma
 dim3 dimGrid(10,10);          //Esto es el numero de individuos

 cudaMemcpy(al1_d, al1, (NumInd-2)*sizeof(int), cudaMemcpyHostToDevice);
 cudaMemcpy(al2_d, al2, (NumInd-2)*sizeof(int),cudaMemcpyHostToDevice);
 cudaMemcpy(popnext_d, popnext,
                                    NumInd*sizeof(chrom),cudaMemcpyHostToDevice);

 cu_sampling<<< dimGrid, dimBlock >>> (al1_d,al2_d,popnext_d);
 cudaMemcpy(al1, al1_d, (NumInd-2)*sizeof(int), cudaMemcpyDeviceToHost);
 cudaMemcpy(al2, al2_d, (NumInd-2)*sizeof(int), cudaMemcpyDeviceToHost);
 cudaMemcpy(popnext, popnext_d, NumInd*sizeof(chrom),
                                                    cudaMemcpyDeviceToHost);

 cudaFree(al1_d);
 cudaFree(al2_d);
 cudaFree(popnext_d);
 return(0);}


void *duplicados (chrom popnext[NumInd]){

//Se encarga de comprobar que los recorridos generados no tienen ciudades repetidas
y si las tiene, eliminarlas.

 int aux_arr [NumCiu];
 int i,j,k,cero,dos,numceros,l,noesta;

 for (j=0;j<NumInd;j++){

//Ejecutamos el algoritmo para cada individuo dentro de la generación

        numceros=0;

//Inicializamos el numero de ceros (ciudades que faltan)...
```

```
        aux_arr [0] = 1;

//...la primera ciudad queda incluida, ya que por defición siempre va a estar...

        for (i=1;i<NumCiu;i++) aux_arr [i] = 0;

//y el resto de ciudades aún ono sabemos si están o no, así que suponemos que no están
en la solución.

        for (i=1;i<NumCiu;i++){aux_arr[popnext[j].map[i]]++;}

//Actualizamos el array auxiliar con la verdadera carga de ciudades que hay en la
solución que se está analizando (si en la solución hay 15 veces la ciudad 3, debemos
tener aux_arr[3]=15).

        for (i=1;i<NumCiu;i++) {if (aux_arr[i]==0) numceros++;}

//Por lo tanto recorremos el array, obteniendo el numero de posiciones en las que hay
ceros (esa ciudad no está incluida en la solución).

        for (k=0;k<numceros;k++){

//Hacemos k pasadas, siendo k el numero de ciudades que no hayan sido incluidas en la
solución

        cero=0;
        while ((aux_arr[cero]!=0) && (cero<NumCiu)){cero++;}

//Con este bucle, hayamos la primera posición en la que hay un 0  (la primera ciudad
que no ha sido incluida en la solucion). Si no la hay es porque las hemos recorrido todas

        if (aux_arr[cero]==0){

//En caso de que hayamos encontrado una ciudad que no esté en la solución...

        dos=0;
        while (aux_arr[dos]<2) {dos++;}

//Buscamos la primera posición (que se corresponde con el valor) en la que haya un
número igual o superior a 2

          l=0;
          noesta = 1;
          while (l<NumCiu && noesta){
```

```
//Recorremos el array buscando la primera posición que contiene el valor hallado
anteriormente

              if (popnext[j].map[l]==dos) noesta = 0;
              l++;
            }

            popnext[j].map[l-1]=cero;

//Guardamos en esa posición menos 1 (porque siempre incrementa) el valor de la ciudad
que no tenía representación en la solución inicial.

            aux_arr[cero]++;
            aux_arr[dos]--;

//Actualizamos correctamente los valores del array auxiliar.
                        }
                    }
        }
        return (0);
}

int calculafit ( chrom individuo, int distancias[NumCiu][NumCiu]){

//No es más que la función encargada de, aplicando el fenotipo al genotipo, calcular el
fitness de cada individuo.

    int aux_fit,i;
    aux_fit=0;

    for (i=0;i<NumCiu;i++){
       aux_fit=aux_fit + distancias [individuo.bit[individuo.map[i]]]
                                    [individuo.bit[individuo.map[(i+1)%NumCiu]]];
    }
    return (aux_fit);
}

void *HillClimbing1 (chrom popcurrent[NumInd], int distancias [NumCiu][NumCiu]) {

//El HillClimbing es un procedimiento encargado de crear una nueva generación
aleatoriamente, cada cierto número de iteraciones. De este modo evitamos que el AG se
estanque en un falso mínimo.

int i,k,aux,aux1,aux2;
```

```
chrom aux_current;

      for (i=0;i<NumInd;i++){

//Para cada individuo...

            k=0;
            while (k<200){
//...ejecutamos 200 veces

             aux_current =  popcurrent[i];
             aux1 = rand()%(NumCiu-1)+1;
             aux2 = rand()%(NumCiu-1)+1;
             aux = aux_current.map[aux1];
             aux_current.map[aux1] = aux_current.map[aux2];
             aux_current.map[aux2] = aux;

//Cambiamos dos posiciones aleatorias en el fitness y comprobamos si hay mejora en el
fitness del individuo. Si ha mejorado lo cambiamos por el que teníamos. Si no es así, lo
dejamos como estaba.

             aux_current.fit=calculafit(aux_current, distancias);
             if (aux_current.fit <= popcurrent[i].fit) popcurrent[i] = aux_current;
             k++;
       }


  }
     return (0);
}
```