FACULTAD DE INGENIERÍA – UNIVESIDAD DE LA REPÚBLICA

FRAMEWORK PARA EL DESARROLLO DE ALGORITMOS GENÉTICOS EN GPU

PROYECTO DE GRADO 2012

PABLO GARCÍA – FRANCISCO POLTI – MONTSERRAT LÓPEZ

TUTORES

MARTÍN PEDEMONTE – PABLO EZZATTI

RESUMEN

Para la resolución de problemas de optimización del tipo NP-difícil, no es posible utilizar métodos exactos debido al excesivo tiempo de cómputo que implican. Para este tipo de problemas, las metaheurísticas han demostrado ser una buena herramienta, aunque en problemas de gran dimensionalidad, pueden incrementar el tiempo de cómputo, al punto de volver también impracticable esta opción. Por esto, y por las posibilidades que brindan las nuevas arquitecturas de hardware, una buena alternativa es la aplicación de estrategias de computación de alto desempeño (HPC).

En los últimos años, las tarjetas gráficas (co-procesador gráfico, GPU) han experimentado una evolución vertiginosa. Este hecho ha motivado que muchos científicos busquen la utilización de las GPUs, para atacar problemas de propósito general (GPGPU). Sumado a esto, la arquitectura Fermi y la plataforma CUDA han evolucionado ofreciendo extensiones en C y C++, facilitando la programación sobre las GPUs.

En este proyecto se realiza el diseño, la implementación y la documentación de un *framework* para la implementación de algoritmos genéticos sobre arquitecturas híbridas que incluyen GPUs. En particular, se logró un prototipo avanzado de un *framework* para que pueda ser utilizado por el resto de la comunidad académica/científica. Este abarca diversos tipos de representación y sus correspondientes operadores evolutivos.

Los resultados obtenidos sobre casos de prueba específicos para cada tipo de representación denotan *speedups* de hasta 6.70x para representación binaria, 11.68x para representación entera y 35.10x para representación real. Los resultados obtenidos en representación de permutación no fueron buenos en cuanto al tiempo de ejecución, a pesar de haber realizado diversas implementaciones para tratar de mejorar la eficiencia de los operadores evolutivos y de la función *fitness.*

Palabras clave: Algoritmos Genéticos, GPU, GPGPU, CUDA, Framework.

CONTENIDO

1 INTRODUCCIÓN

Un problema de optimización, consiste en la toma de una decisión óptima dado un conjunto de restricciones, y corresponde a la maximización o minimización de una cierta función objetivo. En la práctica, los problemas del tipo NP-difícil no pueden ser resueltos mediante métodos exactos de búsqueda, debido al excesivo tiempo de cómputo que implican. Para ese tipo de problemas, las metaheurísticas han demostrado ser una gran herramienta, permitiendo afrontar instancias de grandes dimensiones y obteniendo soluciones cercanas al óptimo en tiempos razonables. Aunque el abordaje de instancias de alta dimensionalidad o el trabajo sobre problemas fuertemente restringidos, pueden significar un aumento considerable en el tiempo de cómputo, al punto de volver también impracticable esta opción.

En ese contexto, y a partir de las posibilidades crecientes que brindan las nuevas arquitecturas de *hardware*, la aplicación de estrategias de computación de alto desempeño (HPC, del inglés High Performance Computing) sobre metaheurísticas, se ha transformado en una alternativa natural para la disminución de los tiempos de ejecución.

En los últimos años, las tarjetas gráficas (co-procesador gráfico, GPU) han experimentado una evolución vertiginosa. Este hecho ha motivado que muchos científicos busquen la utilización de las GPUs para atacar problemas de propósito general (GPGPU, del inglés General Purpose Graphics Processing Unit, GPGPU), y no solamente para tareas vinculadas al procesamiento de gráficos 3D, objetivo original de estos dispositivos. En gran medida, el aumento en la capacidad de cómputo de las GPUs se sustenta en su arquitectura. Como gran diferencia entre las GPUs y CPUs, resalta que las primeras, utilizan estrategias de cálculo paralelo intrínseco. En particular, las GPUs utilizan una estrategia de paralelismo SPMD (del inglés S*ingle Program Multiple Data)*, es decir, un mismo programa que ejecuta en distintos elementos de procesamiento sobre diferentes datos.

La arquitectura Fermi [73], con hasta 512 núcleos CUDA ha mostrado en diversos contextos desempeños comparables a grandes *cluster* de computadores (ofreciendo mejores ratios de capacidad de cómputo vs. precio o capacidad de cómputo vs. energía). Adicionalmente, la arquitectura CUDA, que está diseñada para C++, facilita la programación paralela y acelera el rendimiento en una amplia gama de aplicaciones. Específicamente, la plataforma de computación CUDA [51], ofrece extensiones de C y C++ que permiten representar datos y asignar paralelismo de manera simple, facilitando el camino a la programación de propósito general en GPU.

Basado en lo expresado en los párrafos anteriores, en este proyecto se plantea el diseño de un *framework* para la implementación de algoritmos genéticos en GPU, su implementación y documentación. El objetivo central del proyecto es lograr un prototipo avanzado de un *framework* para que pueda ser usado por el resto de la comunidad académica/científica.

El resto del documento está estructurado de la siguiente manera.

En el Capítulo 2, se presentan las técnicas de computación evolutiva, los algoritmos evolutivos y los algoritmos genéticos, profundizando en las técnicas metaheurísticas que se utilizan en este proyecto. Se presentan también, los modelos de paralelismo de algoritmos evolutivos utilizados por la comunidad. Finalmente, se relevan los principales *frameworks* para la implementación de algoritmos evolutivos.

En el Capítulo 3, se presenta el tema computación de propósito general en unidades de procesamiento gráfico (GPGPU). También se presenta, una breve revisión histórica de la

computación en GPU, la arquitectura de las GPUs en general y la arquitectura Fermi en particular, ya que es la utilizada en la etapa de evaluación experimental, una introducción a CUDA (*Compute Unified Device Architecture*) y un resumen del relevamiento de trabajos realizados para acelerar algoritmos evolutivos sobre GPU.

En el Capítulo 4, se presentan las decisiones de diseño tomadas, una descripción conceptual acerca de la estructura del código y las formas de ejecución, así como la implementación realizada en este proyecto. Además, en este capítulo se describe el archivo de configuración del *framework,* utilizado para la parametrización de las ejecuciones.

En el Capítulo 5, se presenta la evaluación experimental del *framework,* descripción de pruebas funcionales y de performance. Se muestra un caso de estudio para cada tipo de representación implementada y los resultados obtenidos para cada uno de ellos.

En el Capítulo 6, se describen las conclusiones obtenidas que se desprenden de los objetivos planteados al inicio del proyecto. En este capítulo, se detallan también, posibles trabajos futuros que no fueron incluidos en el alcance del proyecto.

Finalmente, se describen los Anexos I, II y III. El Anexo I contiene una descripción detallada sobre acceso *coalesced* a memoria global, *compute capabilities* y divergencia de *threads*. El Anexo II contiene una descripción detallada del archivo de configuración del *framework*. Finalmente, el Anexo III contiene un resumen de instrucciones básicas para la utilización del *framework*.

2 Algoritmos Genéticos y Algoritmos Evolutivos

Resumen: En este capítulo se presentan los Algoritmos Genéticos (AG), técnica metaheurística que se utiliza en este proyecto. La estructura del capítulo es la siguiente. En la primera sección se presentan las técnicas de computación evolutiva. Luego, en la segunda sección se presentan los algoritmos evolutivos, mientras que en la tercera sección se presentan los algoritmos genéticos. En la cuarta sección se comentan los modelos de paralelismo de algoritmos evolutivos. Finalmente, en la quinta sección, se relevan los principales frameworks para la implementación de algoritmos evolutivos existentes.

2.1 Técnicas de Computación Evolutiva

La resolución de problemas de optimización puede ser abordada con métodos exactos, heurísticas o metaheurísticas [20], entre otras técnicas. La Figura 2.1, presenta un esquema de las distintas técnicas de resolución de estos problemas, que son detalladas a continuación con mayor profundidad.

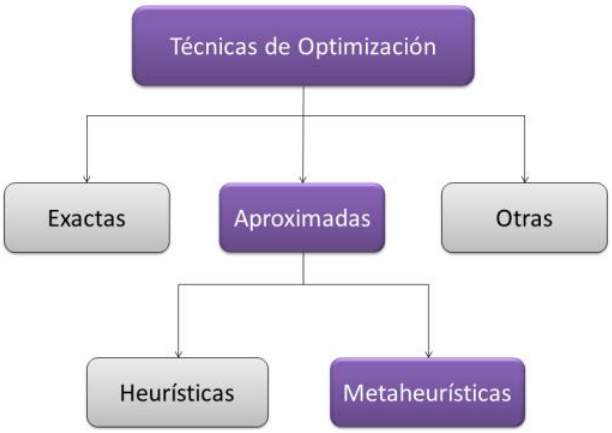


Figura 2.1- Clasificación de problemas de optimización.

Los métodos exactos permiten encontrar una solución óptima y demostrar que la solución obtenida es óptima global. La principal ventaja de estos métodos es que garantizan encontrar óptimos globales, y su principal desventaja es que cuando se enfrentan con problemas de clase NP-difícil [47] –un problema pertenece a la clase NP-difícil si puede ser resuelto en tiempo polinomial, pero usando una computadora no determinística– su tiempo de ejecución crece exponencialmente, y eso usualmente impone una limitación para resolver el problema en un tiempo razonable. Ya que los algoritmos exactos muestran un bajo desempeño para muchos problemas, se han desarrollado múltiples tipos de algoritmos aproximados, que proporcionan soluciones de alta calidad (no necesariamente óptimos) en un tiempo computacional aceptable.

Entre los métodos aproximados, están las heurísticas, que son algoritmos aproximados, y suelen estar basadas en procedimientos conceptualmente simples, éstas se caracterizan por encontrar soluciones de buena calidad (no necesariamente hallan una solución óptima) en forma eficiente. Existe una variada gama de técnicas heurísticas. En general, las más útiles son las heurísticas específicas, que incorporan conocimiento (específico) del problema de optimización a resolver. Una idea común, es incorporar mecanismos de detección de óptimos locales, para que el procedimiento de búsqueda no “quede atrapado” en valores sub-óptimos de baja calidad. Su ventaja principal por el contrario de los métodos exactos, es que las técnicas basadas en heurísticas tienden a ser muy rápidas. Por otra parte, su principal desventaja es que las soluciones obtenidas pueden no ser las óptimas, y puede ser muy difícil adaptar las técnicas o ideas a otros problemas similares.

Las técnicas metaheurísticas ofrecen un buen balance entre las dos técnicas mencionadas anteriormente, son técnicas genéricas que ofrecen buenas soluciones, usualmente en un tiempo moderado de ejecución. Estas técnicas al igual que las heurísticas, tampoco aseguran hallar el óptimo global, pero por contraparte son "menos particulares", es decir, más generalizables; aunque deben instanciarse para cada clase de problemas. Sus principales características son: permitir encontrar rápidamente soluciones factibles (puede ser en sí mismo un problema NP-difícil); encontrar soluciones factibles de buena calidad (valores de la función de *fitness* cercanos al óptimo); y recorrer el espacio de soluciones sin quedar "atrapados" en una zona del espacio de búsqueda. Para que una metaheurística obtenga buenos resultados, debe balancear la exploración del espacio de búsqueda con la explotación de la información de las buanas soluciones obtenidas. La explotación del espacio de búsqueda, es el proceso de utilizar información obtenida de puntos del espacio de búsqueda previamente visitados, para determinar los puntos que conviene visitar a continuación, este es un buen mecanismo para que un algoritmo encuentre óptimos locales. La exploración, es el proceso de visitar nuevas regiones del espacio de búsqueda para tratar de encontrar buenas soluciones, este es un buen mecanismo para evitar que un algoritmo quede "atrapado" en óptimos locales.

Algunas de las técnicas metaheurísticas que han tenido amplia aplicación son: *Simulated Annealing* [12], es una técnica que permite realizar movimientos en la solución que pueden empeorar la función de *fitness*, pero que permiten escaparle a óptimos locales; *Tabu Search* [19], realiza una búsqueda local con memoria a corto plazo para escapar de mínimos locales y evitar ciclos; *Greedy Randomized Adaptive Search Procedure* [67], es un procedimiento iterativo en dos etapas: construcción y mejora de la solución; *Ant Colony Optimization* [39], es una técnica que se basa en el comportamiento social de las hormigas, cada hormiga artificial construye una solución agregando componentes a una solución parcial en consideración; y *Algoritmos Evolutivos,* que se profundizan en la Sección 2.2.

2.2 Algoritmos Evolutivos

Los algoritmos evolutivos agrupan las técnicas que trabajan sobre una población de soluciones que evoluciona mediante mecanismos de selección y construcción de soluciones candidatas, por recombinación de características de las soluciones anteriores. Este tipo de técnicas son metaheurísticas que emulan la evolución biológica y trabajan con una población (de representaciones) de soluciones. Sus principios son: selección natural (aptitud), reproducción (recombinación y mutación) y diversidad genética. Siguen la idea de la supervivencia de los individuos más aptos, evaluados de acuerdo al problema a resolver, mediante una función de *fitness*.

Las distintas etapas de la ejecución de un algoritmo evolutivo se muestran en la Figura 2.2.

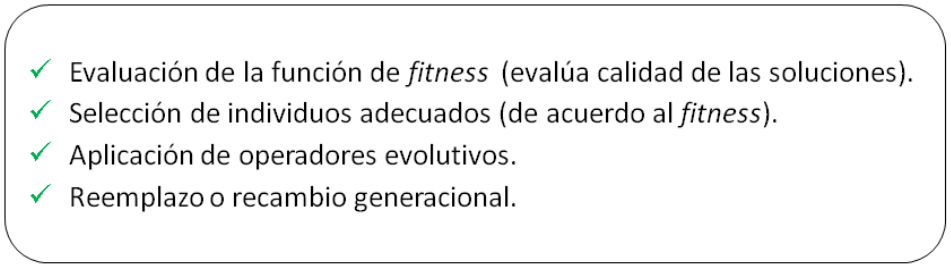


Figura 2.2- Etapas de ejecución de un algoritmo evolutivo.

En la Figura 2.3 se muestra el esquema genérico de un Algoritmo Evolutivo sobre una población *P*. El primer paso de la ejecución consiste en inicializar la población (generalmente de forma aleatoria), y el número de generación en cero. Luego, mientras no se cumpla con la condición de parada, se evalúa la población de la generación actual, se aplican los operadores de selección, y los operadores evolutivos (cruzamiento y mutación) y luego se reemplaza la generación anterior por la nueva generación (hijos) según el criterio de reemplazo seleccionado. Finalmente se incrementa el número de generación y se guarda la nueva población generada como la población actual.

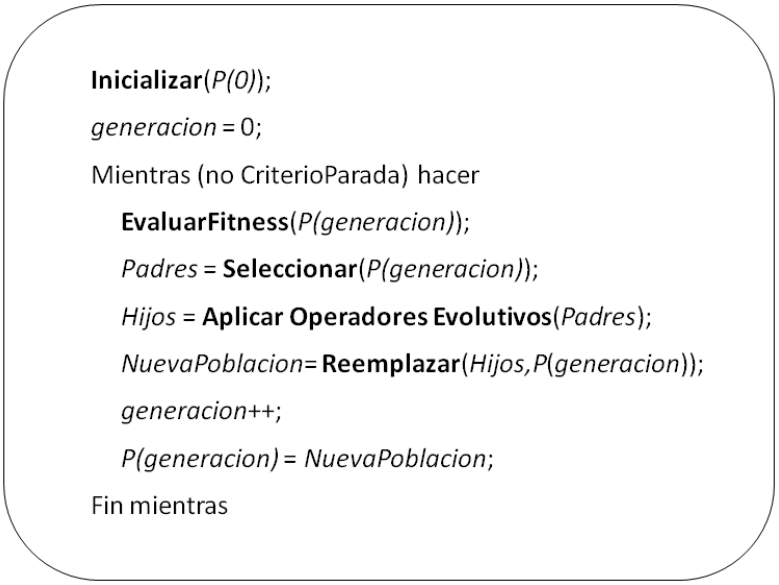


Figura 2.3 – Esquema genérico de un algoritmo evolutivo (AE) sobre una población P.

Algunas de las técnicas de algoritmos evolutivos más populares son:

- Programación Evolutiva [31]: La programación evolutiva es una propuesta realizada por John Koza, que utiliza un mecanismo evolutivo para la generación de programas. Estas técnicas se han aplicado en los últimos años principalmente en aplicaciones como entrenamiento de redes neuronales y evolución de sistemas difusos.
- Estrategias de Evolución [26]: Las estrategias de evolución fueron introducidas por Rechenberg en 1965, consiste en un método de optimización que trabaja sobre individuos compuestos por números reales para optimizar parámetros en problemas de diseño en ingeniería.
- Algoritmos Meméticos [61]: Son algoritmos híbridos que incorporan conocimientos y combinan estrategias de búsqueda. La idea de estos algoritmos consiste en incorporar conocimiento del problema a través de conceptos (memes), que pueden ser útiles para hallar eficientemente mejores soluciones a un problema de optimización.
- Algoritmos de Estimación de Distribución (AED) [56]: Los AED buscan estimar la distribución de probabilidad de cada variable. La población de soluciones candidatas se crea en cada generación, a partir de la distribución de probabilidad obtenida, y de los mejores individuos de la generación anterior. No existen operadores de cruzamiento ni de mutación, dado que la población no se genera a partir de individuos.

2.3 Algoritmos Genéticos

Fueron sugeridos por John Holland en la década del 70 [30], pero luego formalizados y sistematizados por David Goldberg en la década del 80 [8]. En los Algoritmos Genéticos se hace mayor hincapié en el operador de cruzamiento que en el de mutación, en general se utiliza selección

probabilística y usualmente es representado por una cadena binaria. El esquema algorítmico de un Algoritmo Genético sigue el estándar de un Algoritmo Evolutivo (presentado en la Figura 2.2), utilizando como operadores evolutivos, el cruzamiento y la mutación.

Componentes básicos:

1. ***Representación de las soluciones potenciales del problema.***
   A la representación de la solución se le llama genotipo, y a la solución se le denomina fenotipo. A la cadena general se le llama cromosoma. A cada sub-cadena (posición en la cadena general) se le denomina gen, y al valor dentro de esta posición se le llama alelo. La longitud de la representación depende de las características del problema (número de variables, dimensión del dominio) y de las características de la solución buscada (precisión deseada, por ejemplo), y el tipo de representación depende de las características del problema a resolver.

2. ***Procedimiento para crear una población inicial de posibles soluciones.***
   Generalmente es un proceso aleatorio aunque una inicialización no aleatoria de la población inicial puede acelerar la convergencia del Algoritmo Genético. En algunos casos, la desventaja que puede tener la inicialización no aleatoria es la convergencia prematura del algoritmo, lo que lleva a una convergencia hacia óptimos locales.

3. ***Función de evaluación que representa el ambiente, clasificando las soluciones en términos de su aptitud.***
   Esta función es denominada función de *fitness* (o aptitud). La función de *fitness* guía el mecanismo de búsqueda a través del operador de selección, ésta depende del problema y del criterio de optimización, puede cambiar dinámicamente a medida que el AG avanza, y permite contemplar objetivos múltiples o incorporar sub-objetivos. La función de *fitness* es una "caja negra" para el AG. Puede haber más de una función de *fitness* que sean aplicables a un mismo problema. Una dificultad que puede presentarse al momento de diseñar la función de *fitness* es la generación de individuos no factibles durante la evolución. Existen tres mecanismos para tratar a los individuos no factibles: evitarlos en la codificación, descartarlos o penalizarlos.

4. ***Conjunto de operadores evolutivos que alteran la composición de los individuos de la población a través de las generaciones.***
   Los operadores utilizados en los algoritmos genéticos son los de selección, cruzamiento y mutación. Estos se detallan a continuación.

Selección

El mecanismo de selección dirige al AG hacia la exploración de secciones "prometedoras" del espacio de búsqueda. La selección de cromosomas se determina por el mecanismo de muestreo. La probabilidad de selección para cada individuo es proporcional a su *fitness*. Esto puede traer como inconveniente que en generaciones tempranas unos pocos individuos muy adaptados puedan dominar la selección, mientras que en las últimas generaciones, la competencia se puede volver débil y el comportamiento del AG se parezca al de a una búsqueda aleatoria. Para solucionar éste problema se suelen utilizar mecanismos de escalado y de ordenamiento.

Cruzamiento

Los operadores de cruzamiento combinan características de dos o más individuos, con el fin de obtener descendientes con mejor *fitness.* El objetivo del cruzamiento es garantizar la explotación de buenas secciones del espacio de búsqueda.

Mutación

El operador de mutación produce modificaciones aleatorias, con lo que introduce diversidad y permite explorar nuevas secciones del espacio de búsqueda.

5. ***Una configuración paramétrica (tamaño de la población, probabilidad de mutación, probabilidad de cruzamiento, criterio de parada, etc.).***

   En general los AG trabajan sobre una población fija de *n* individuos, el tamaño de la población debe ser un valor que permita mantener la diversidad de los individuos solución, sin sacrificar la eficiencia computacional del mecanismo de búsqueda. Las probabilidades de aplicación de los operadores evolutivos, definen el balance entre la exploración del espacio de búsqueda, y la explotación de buenas soluciones. El criterio de parada puede especificar un esfuerzo prefijado o involucrar aspectos dinámicos de la evolución.

2.3.1 Tipos Básicos de Representación

Los principales y más populares tipos de representación son: binaria, entera, permutación y real. Pueden existir problemas que requieran representaciones más sofisticadas como por ejemplo, árboles.

2.3.1.1 Representación binaria

En la representación binaria se utiliza una cadena cuya longitud es la del número de genes de cada individuo y el alelo (valor que puede tomar cada elemento) es un número binario (0 o 1). Esta representación es la más usada por ser simple y fácil de manipular a través de los operadores genéticos. Los mecanismos de codificación binaria más utilizados son código binario y código de Gray [60]. La Figura 2.4 muestra una cadena con representación binaria.

Figura 2.4 – Cadena con representación binaria.

2.3.1.2 Representación entera

Algunos problemas, como por ejemplo la determinación de parámetros de procesamiento de imágenes deben ser representados con variables enteras. En este tipo de representación cada gen es un valor entero. La Figura 2.5 muestra una cadena con representación entera.

Figura 2.5 – Cadena con representación entera.

2.3.1.3 Representación de permutación

Este tipo de representación es una variante de la representación entera, con la diferencia que no permite elementos repetidos en la solución. Se utilizan en problemas de ordenamiento o de secuenciación donde se requiere decidir el orden en que deberían ocurrir ciertos eventos, como por ejemplo en el Problema del Viajante de Comercio [58]. Generalmente, las soluciones a estos problemas son expresadas como una permutación de *n* elementos predefinidos. Si el problema

posee *n* variables entonces la representación es una lista de *n* enteros (cada entero se encuentra una sola vez en la lista). La Figura 2.6 muestra una cadena con representación de permutación.

Figura 2.6 – Cadena con representación de permutación.

2.3.1.4 Representación real

En la representación real se utiliza una cadena donde cada alelo es representado por un número real. Con ésta representación se opera a nivel de fenotipo y la misma requiere de operadores genéticos especiales. La Figura 2.7 muestra una cadena con representación real.

Figura 2.7 – Cadena con representación real.

2.3.2 Mecanismos de Selección

A continuación se describen los principales mecanismos de selección (muestreo) utilizados. Estos se dividen en los siguientes grupos: estocásticos, determinísticos y mixtos (que incluyen características determinísticas y aleatorias), según el grado de intervención del azar en el proceso.

2.3.2.1 Rueda de ruleta

Se crea una ruleta [29] con los cromosomas presentes en la generación actual, cada cromosoma tendrá una parte de esa ruleta mayor o menor en forma proporcional a su *fitness*. Se hace girar la ruleta y se selecciona el cromosoma en el que se para la ruleta. El cromosoma con mayor valor de *fitness* saldrá con mayor probabilidad. En caso de que las probabilidades sean dispares, este método de selección puede dar problemas, ya que si un cromosoma tiene una alta probabilidad de ser seleccionado, el resto tendrá una probabilidad mucho menor, con lo cual se puede reducir la diversidad genética [30]. Esta es una técnica de muestreo estocástico.

En la Figura 2.8 se muestra un ejemplo de selección por rueda de ruleta.

Individuo	Fitness	Probabilidad selección
1	169	14,4
2	576	49,2
3	64	5,5
4	361	30,9
Total	1170	100

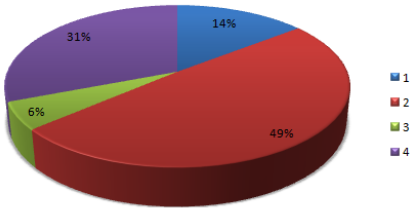


Figura 2.8 – Rueda de Ruleta.

2.3.2.2 Muestreo estocástico universal

Es un algoritmo de muestreo que se implementa en un sola fase, es un algoritmo simple y eficiente [43] desarrollado por Baker [34] en 1987. Dado un conjunto de *n* individuos y sus valores de *fitness* asociados, este algoritmo los coloca en una rueda de ruleta donde el tamaño de los cortes asignados a cada individuo, es proporcional al valor de *fitness*. Después, una segunda ruleta es marcada con *x* marcadores equiespaciados entre sí, donde *x* es la cantidad de selecciones que queremos realizar. Para finalizar, se gira la segunda ruleta y se selecciona un individuo por cada marcador. Las posiciones de los marcadores determinan los individuos seleccionados; si *k* marcadores caen sobre el

mismo individuo, esto significa que éste fue seleccionado *k* veces. Esto garantiza que los individuos sean seleccionados evitando sesgos que puedan producirse por una disparidad en las probabilidades de selección. Esta es una técnica de muestreo estocástico. En la Figura 2.9 se muestra un ejemplo de muestreo estocástico universal, en este ejemplo, el individuo I^t_1 es seleccionado dos veces, mientras que los individuos I^t_3 e I^t_4 son seleccionados una vez, en este caso el individuo I^t_2 no fue seleccionado.

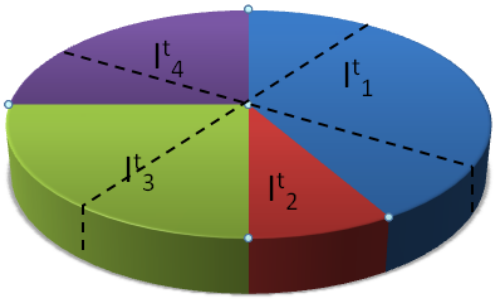


Figura 2.9 – Muestreo estocástico universal.

2.3.2.3 Selección por rango

Se ordenan los individuos según su *fitness* y la selección se realiza a partir de este rango. En este mecanismo se produce una variedad genética mucho más amplia que en el mecanismo de Rueda de Ruleta, pero por contraparte tiene el problema de que la convergencia puede ser más lenta, ya que no existe gran diferencia entre el mejor cromosoma y el resto, como ocurría en la Rueda de Ruleta [29]. La técnica de selección por rango es una técnica de muestreo determinístico.

2.3.2.4 Selección por truncamiento

En este método los individuos se ordenan según su *fitness*, luego se selecciona un porcentaje *T* de los individuos con mejor *fitness,* y se reproducen *1/T* veces. Este método es menos eficiente que los demás métodos de selección y su uso no es muy común en la práctica [55]. Esta es una técnica de muestreo determinístico.

2.3.2.5 Selección elitista

En este método se copian el o los mejores individuos de la generación anterior a la nueva generación. El elitismo puede mejorar el funcionamiento de los algoritmos genéticos, dado que asegura la permanencia de él o los mejores individuos de la generación en la nueva población [8]. Esta es una técnica de muestreo determinístico.

2.3.2.6 Selección por torneo

La selección por torneo [3], es una técnica de muestreo mixto donde se eligen *k* individuos aleatoriamente de la población y se seleccionan los *r* de mejor *fitness*.

2.3.2.7 Selección por torneo estocástico

Es una técnica de muestreo mixto que utiliza rueda de ruleta [29] para seleccionar individuos entre los cuales se efectúa un torneo [3] para determinar cuáles sobreviven.

2.3.3 Tipos de operadores de cruzamiento

A continuación se presentan los tipos de operadores de cruzamiento en representación binaria, entera, de permutaciones y real.

2.3.3.1 Representación binaria

En esta sección se muestran los operadores de cruzamiento para representación binaria.

(SPX) Cruzamiento en un punto

El cruzamiento en un punto consiste en obtener dos descendientes, H1 y H2 a partir de dos individuos padres, P1 y P2, seleccionando un punto al azar y "cortando" los padres e intercambiando los trozos de cromosomas, como se muestra en la Figura 2.10.

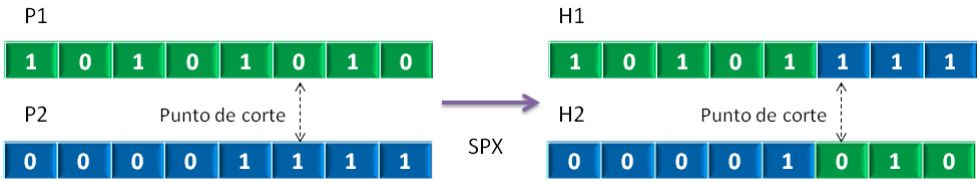


Figura 2.10 – Cruzamiento en un punto (SPX).

(2PX) Cruzamiento en dos puntos

El cruzamiento en dos puntos consiste en seleccionar aleatoriamente dos puntos de corte y luego intercambiar los cromosomas padres entre los puntos de corte, generando así los descendientes como se muestra en la Figura 2.11.

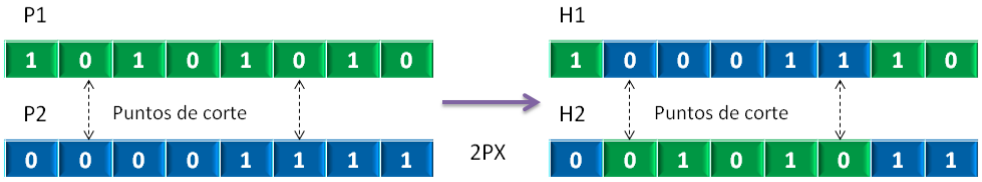


Figura 2.11 – Cruzamiento en dos puntos (2PX).

(UX) Cruzamiento uniforme

En el cruzamiento uniforme en cada posición de los progenitores se decide intercambiar el valor genético de cada punto de acuerdo a una probabilidad prefijada (generalmente 0.5), como se muestra en la Figura 2.12.

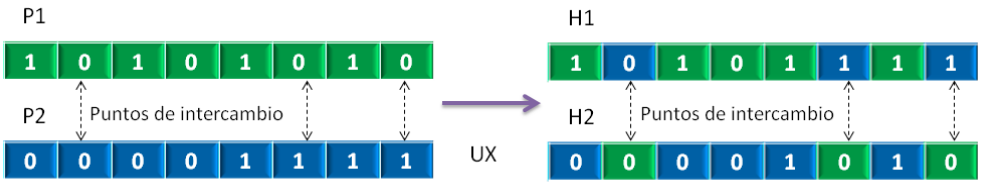


Figura 2.12 – Cruzamiento uniforme (UX).

2.3.3.2 Representación entera

Los operadores de cruce de *N* puntos y uniformes, vistos en las secciones anteriores son aplicables a este tipo de representaciones.

2.3.3.3 Representación de permutación

Los operadores de cruzamiento presentados hasta el momento producen soluciones inadmisibles para esta representación. Los operadores diseñados para ésta representación se centran en combinar la información proveniente de los dos padres sin repetidos, de acuerdo al orden o la adyacencia de los elementos. A continuación se presentan los operadores de cruzamiento aplicables a la representación de permutación.

(PMX) Cruzamiento basado en correspondencia parcial

Operador de cruzamiento introducido por Goldberg y Lingle en 1985 [7]. Partiendo de los padres P1 y P2 el procedimiento para generar los hijos H1 y H2 es el siguiente:

1. Se seleccionan dos puntos de cruce al azar, y se copian los valores de P1 que se encuentran entre los puntos de cruce en H1.
2. Posicionándose en el primer punto de cruce sobre P2, se revisan qué elementos de P2 (existentes entre los dos puntos de cruce) no han sido incluidos en H1.
3. Para cada elemento *i* de los mencionados, se revisan qué elemento *j* ha sido copiado en su posición (sobre P2) en H1.
4. Se ubica *i* en H1, en la posición ocupada por *j* sobre P2 (esto es posible de hacer porque *j* ya ha sido ubicado en H1).
5. Si la posición ocupada por *j* en el P2 ya ha sido llenada en H1 por un elemento *k*, se ubica *i* en la posición ocupada por *k* en P2.
6. Una vez revisados los elementos entre los puntos de cruce, las posiciones restantes de H1 deben ser llenadas a partir de P2.

H2 es creado de manera análoga invirtiendo los roles de los padres.

Ejemplo:

Paso 1 – Se copia el segmento entre los puntos de cruce seleccionados al azar de P1 en H1

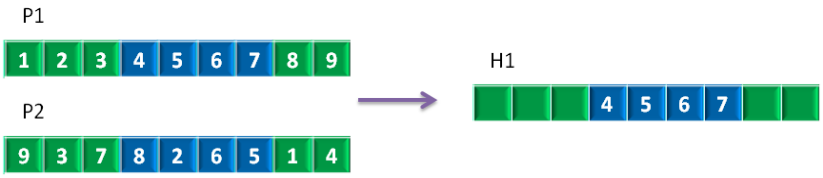


Figura 2.13 – Paso 1.

Paso 2, 3, 4 y 5 – Se revisa qué elementos de P2 (en el segmento entre los puntos de cruce) no han sido copiados a H1 (2 y 8) y se realiza la correspondencia según los pasos 3, 4 y 5 para completar H1. 2 –> 5 –> 7 y 8 –> 4

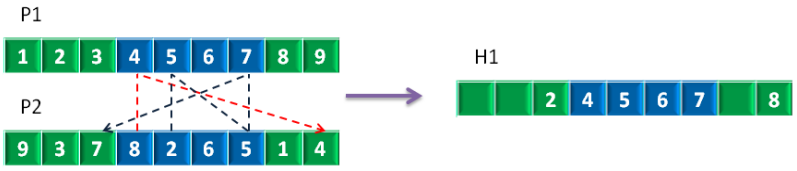


Figura 2.14 – Pasos 2 al 5.

Paso 6 – Se completan las posiciones restantes de H1 con los elementos de P2.

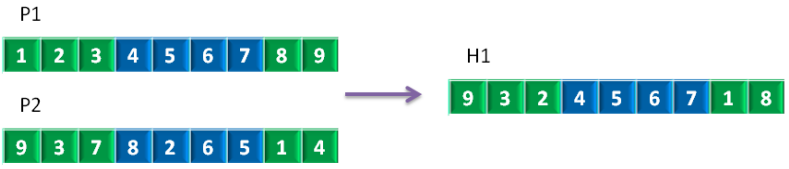


Figura 2.15 – Paso 6.

(OX) Cruzamiento basado en el orden

OX es un operador propuesto por Davis [38]. La idea es mantener el orden relativo en el cual ocurren los elementos, el procedimiento es el siguiente (partiendo de los padres P1 y P2, generando los hijos H1 y H2):

1. Elegir dos puntos de cruce al azar.
2. Copiar los valores de P1 que se encuentran entre los dos puntos de cruce en H1.
3. Copiar en H1 los valores que aún no se han incluido en dicho hijo.

1. Comenzar a partir del segundo punto de cruce de P2.
2. Copiar los valores no incluidos en H1 respetando el orden en el cual dichos valores aparecen en P2.
3. Al terminar la lista de P2, continuar con los primeros valores de la misma.

H2 es creado de manera análoga (pasos 2 y 3) invirtiendo el rol de los padres.

Ejemplo:
Se copia el segmento seleccionado al azar de P1 en H1

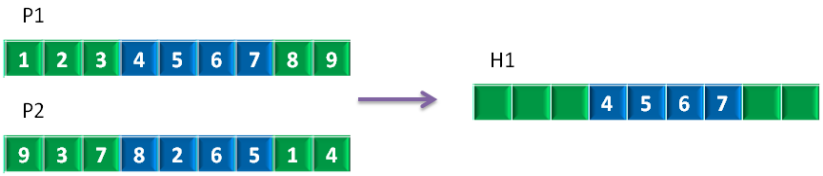


Figura 2.16 – Cruzamiento basado en orden.

Se copian los valores no incluidos en H1 respetando el orden que ellos tienen en P2 (1, 9, 3, 8, 2)

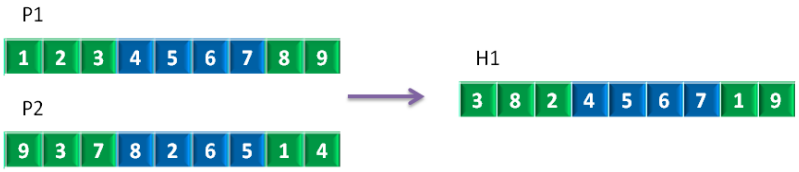


Figura 2.17 – Resultado del cruzamiento basado en orden.

(CX) Cruzamiento basado ciclos

CX es un operador introducido por Oliver en 1987 [27]. La idea es preservar la posición absoluta en la cual los elementos ocurren, el operador divide a los elementos en ciclos. El hijo es creado seleccionando de manera alternada ciclos de cada padre. El procedimiento es el siguiente:

1. Se define un ciclo de valores a partir de P1 en la siguiente forma:
    1. Comenzar con el primer valor no usado de P1.
    2. Revisar el valor ubicado en la misma posición en P2.
    3. Ir a la posición que contiene el mismo valor en P1.
    4. Sumar este valor al ciclo.
    5. Repetir los pasos 2 a 4 hasta que se llegue al primer valor de P1.
2. Ubicar los valores del ciclo en H1 (H2) respetando las posiciones que ellos tienen en P1 (P2).
3. Definir el siguiente ciclo. Ubicar a los valores de este ciclo en H1 (H2) respetando las posiciones que ellos tienen en P2 (P1).

Ejemplo:
El primer paso es identificar ciclos, a partir de la Figura 2.18 se obtienen los siguientes ciclos: (1 8 4 9), (2 5 7 3) y (6)

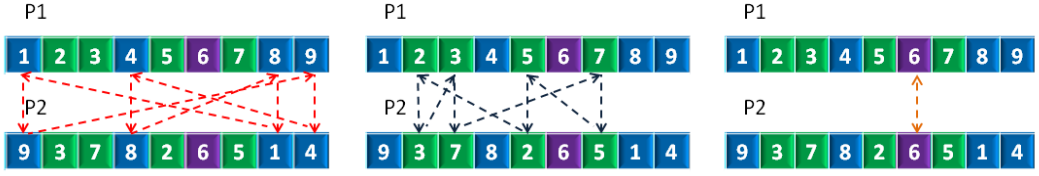


Figura 2.18 – Cruzamiento basado en ciclos.

Luego de haber identificado los ciclos se intercambian los ciclos para formar H1 y H2, como se muestra en la Figura 2.19

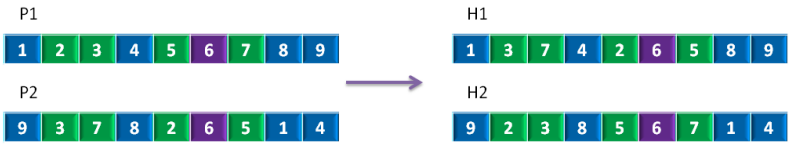


Figura 2.19 –Resultado del cruzamiento basado en ciclos.

(ER) Cruzamiento basado en arcos

ER es un operador propuesto por Whitley y Starkweather [10][11]. Este operador se basa en la idea de que un hijo debería ser creado utilizando solamente los arcos que están presentes en uno o en los dos padres. Para lograr este objetivo, se construye una tabla de arcos (listas de adyacencia). En dicha tabla, para cada elemento, se indica cuáles son los elementos adyacentes a él en cada uno de los padres (el símbolo + indica que un arco está presente en ambos padres).

Una vez construida la tabla de arcos, el procedimiento es el siguiente:

1. Elegir al azar un elemento y ubicarlo en el hijo.
2. Remover todas las referencias, que existan en la tabla, al elemento elegido.
3. Examinar la lista de arcos del elemento elegido:
   – Si existe un arco común (+) a un cierto elemento, elegir a este último para que sea el siguiente elemento en el hijo.
   – En otro caso, se elige el elemento de la lista que tenga la lista de arcos más corta.
4. En el caso de alcanzar una lista vacía:
   – El otro extremo del hijo es examinado por extensión.
   – Si por el otro extremo no se cumple ninguna condición, un nuevo elemento es elegido al azar.

Ejemplo:
Partiendo de los padres P1 y P2 que se muestran a continuación

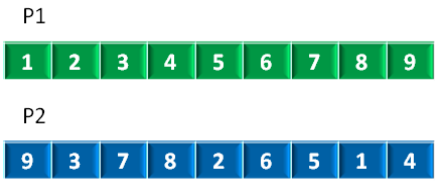


Figura 2.20 – Padres P1 y P2.

Se genera la tabla de adyacencia, que se muestra en la Tabla 2.1

Elemento	Adyacentes
1	2, 5, 4, 9
2	1, 3, 6, 8
3	2, 4, 7, 9
4	1, 3, 5, 9
5	1, 4, 6+
6	2, 5+, 7
7	3, 6, 8+
8	2, 7+, 9
9	1, 3, 4, 8

Tabla 2.1 – Tabla de adyacencia.

En la Tabla 2.2 se muestra el proceso de construcción de un hijo.

Opciones	Elemento seleccionado	Motivo	Resultado parcial
Todas	1	Aleatorio	[1]
2, 5, 4, 9	5	Lista más corta	[1 5]
4, 6	6	Arco común	[1 5 6]
2, 7	2	Aleatoria (ambos tienen 2 elem. en la lista)	[1 5 6 2]
3, 8	8	Lista más corta	[1 5 6 2 8]
7, 9	7	Arco común	[1 5 6 2 8 7]
3	3	Único elem. en la lista	[1 5 6 2 8 7 3]
4, 9	9	Aleatoria	[1 5 6 2 8 7 3 9]
4	4	Último elemento	[1 5 6 2 8 7 3 9 4]

Tabla 2.2 – Tabla del proceso de construcción de un hijo con ER.

2.3.3.4 Representación real

A continuación se presentan los operadores de cruzamiento para la representación real.

Cruce aritmético individual

Dados los padres $<x_1,..,x_n>$ y $<y_1,...,y_n>$, se elige al azar una posición *k* tal que $1 \leq k \leq n$, entonces el primer hijo se conforma de la siguiente manera: $<x_1,...x_{k-1},\propto.y_k + (1 - \propto).x_k, x_{k+1},...,x_n>$ y el segundo hijo se compone invirtiendo los roles de los padres.

Ejemplo: ($\propto = 0.5$ y $k = 7$)

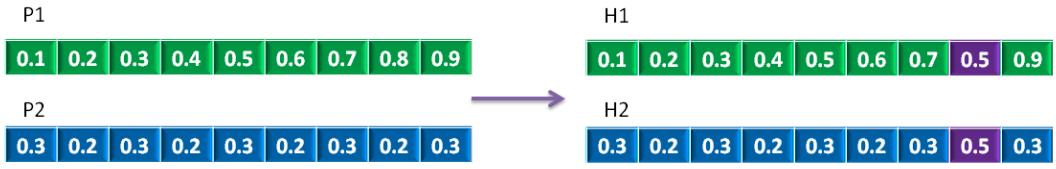


Figura 2.21 – Cruce aritmético individual.

Cruce aritmético simple

Dados los padres $<x_1,..,x_n>$ y $<y_1,...,y_n>$, se elige al azar una posición *k* tal que $1 \leq k \leq n$, entonces el primer hijo se conforma de la siguiente manera: $<x_1,...x_k,\propto.y_{k+1} + (1 - \propto).x_{k+1},..., \propto.y_n + (1- \propto).x_n>$ y el segundo hijo se compone invirtiendo los roles de los padres.

Ejemplo ($\propto = 0.5$ y $k = 6$)

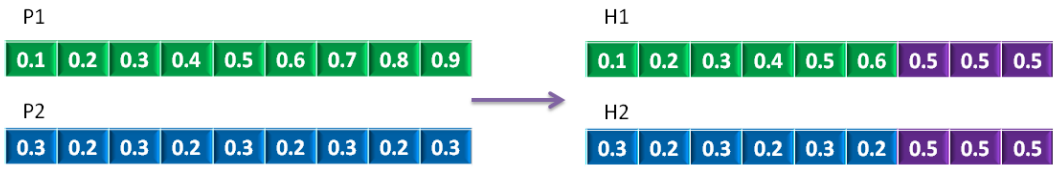


Figura 2.22– Cruce aritmético simple.

Cruce aritmético completo

Este operador es una variante del cruce aritmético simple, en el que se sigue el mismo mecanismo que en el cruce aritmético simple pero aplicado a todas las posiciones. Partiendo de los padres

<x_1,..,x_n> y <y_1,…,y_n>, el primer hijo se conforma de la siguiente manera: $\propto.\bar{x} + (1 - \propto).\bar{y}$, y el segundo hijo de la siguiente forma: $\propto.\bar{y} + (1 - \propto).\bar{x}$.

Ejemplo: ($\propto = 0.5$)

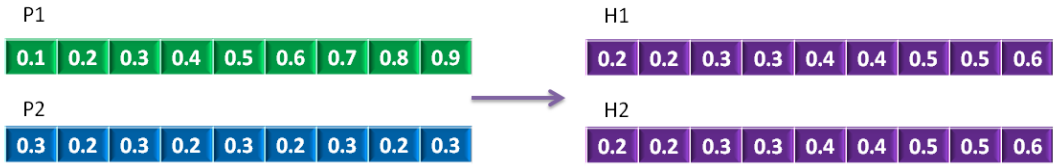


Figura 2.23 – Cruce aritmético completo.

2.3.4 Tipos de operadores de mutación

A continuación se presentan los tipos de operadores de mutación de la representación binaria, entera, de permutación y real.

2.3.4.1 Representación binaria

En esta sub-sección se describen los operadores de mutación aplicados al tipo de representación binaria.

Mutación de inversión de valor de alelo

Se modifica aleatoriamente uno de los valores binarios del cromosoma, existen dos variantes de este tipo de mutación: mutar de a un bit o mutar un único bit. La modificación consiste en invertir el valor binario de un alelo, como se muestra en la Figura 2.24.

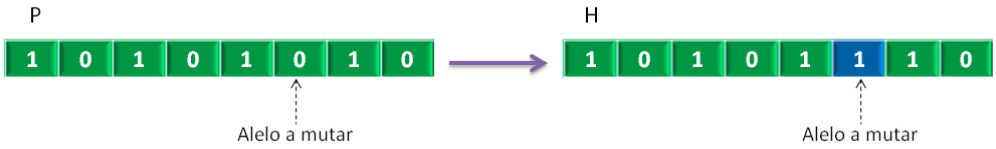


Figura 2.24 – Mutación de inversión de valor de alelo.

2.3.4.2 Representación entera

Se extiende el operador de mutación de inversión del alelo para poder ser aplicado sobre este tipo de representaciones.

Reajuste Aleatorio

Cambia el valor de cada gen por otro valor posible, el nuevo valor se obtiene sorteando aleatoriamente entre todos los valores posibles.

Ejemplo:

Partiendo del individuo P de la Figura 2.25, siendo D=[0,10] el dominio, y suponiendo que el gen a mutar es el marcado en la Figura 2.25 y el valor sorteado aleatoriamente es 10, el resultado de aplicar la mutación por reajuste aleatorio se muestra en H.

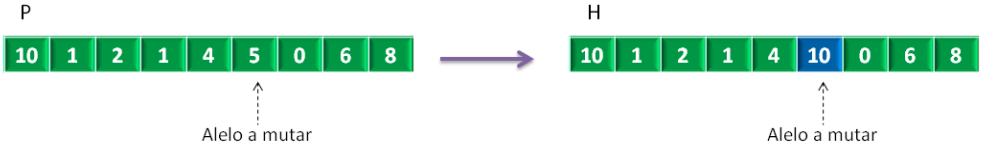


Figura 2.25 – Mutación por reajuste aleatorio.

Mutación por Deslizamiento

Suma un valor que puede ser tanto positivo como negativo al valor de cada gen. Los valores a sumar son elegidos aleatoriamente para cada posición.

Ejemplo:

Partiendo del individuo P y de los valores a sumar que se muestran en la Figura 2.26 se obtiene como resultado de la mutación por deslizamiento el individuo H.

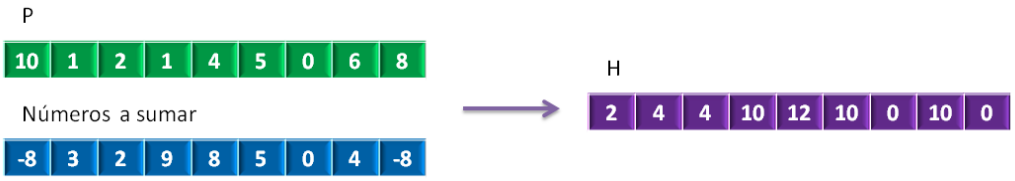


Figura 2.26 – Resultado de aplicar la mutación por deslizamiento.

2.3.4.3 Representación de permutación

Para permutaciones, no es posible considerar a cada gen independientemente. En este caso, la mutación consiste en mover los valores de los genes sobre la solución, como consecuencia, la probabilidad de mutación es interpretada como la probabilidad de que el operador sea aplicado sobre la solución completa.

(EM) Mutación por intercambio

Este operador fue introducido por Banzhaf [71]. Se eligen dos posiciones al azar y se intercambian sus valores. Esta técnica preserva la mayor parte de la información sobre adyacencias, no así la información sobre el orden.

Ejemplo:

Partiendo del individuo P1 de la Figura 2.27 y siendo 2 y 5 las posiciones seleccionadas, obtenemos como resultado el individuo H1. En este ejemplo la cantidad de links rotos es 4, inicialmente el valor 2 estaba conectado a los valores 1 y 3, mientras que el valor 5 estaba conectado a los valores 4 y 6. Luego de aplicar el operador de mutación por intercambio, estos 4 links se rompieron.

Figura 2.27 – Mutación por intercambio.

(ISM) Mutación por inserción simple

Este operador fue introducido por Fogel [6] y Michalewizc [76]. Se eligen dos valores al azar de la solución, se mueve el segundo valor a continuación del primero (se corren los valores ubicados entre medio de los dos elegidos). Este operador preserva la mayor parte del orden existente entre los valores y de la información sobre adyacencias.

Ejemplo:

Partiendo del individuo P1 de la Figura 2.28 y siendo 2 y 5 las posiciones seleccionadas, obtenemos como resultado el individuo H1. En este ejemplo la cantidad de links rotos es 3, inicialmente el valor 2 estaba conectado al valor 3, mientras que el valor 5 estaba conectado a los valores 4 y 6. Luego de aplicar el operador de mutación por inserción simple, estos 3 links se rompieron.

Figura 2.28 – Mutación por inserción simple.

(IVM) Mutación por inversión

Este operador fue introducido por Fogel [5][6]. Se eligen al azar dos posiciones de la solución y se invierte el orden de los valores existentes entre dichas posiciones. Esta técnica al igual que la mutación por intercambio (EM) preserva la mayor parte de la información sobre adyacencias pero perturba significativamente la información sobre el orden.

Ejemplo:

Partiendo del individuo P1 de la Figura 2.29 y siendo 2 y 5 las posiciones seleccionadas, se obtiene como resultado el individuo H1. En este ejemplo la cantidad de links rotos es 2, ya que inicialmente el valor 2 estaba conectado al valor 1 y el valor 5 estaba conectado al valor 6. Luego de aplicar la mutación por inversión estos dos links se rompieron.

Figura 2.29 – Mutación por inversión.

Mutación por mezcla

Se elige aleatoriamente un conjunto de posiciones de la solución, luego se reordenan (se mezclan) aleatoriamente los valores existentes en las posiciones elegidas (sobre dichas posiciones). Las posiciones elegidas pueden no ser contiguas.

Ejemplo:

Partiendo del individuo P1 de la Figura 2.30 y siendo 2, 3, 4 y 5 las posiciones seleccionadas, obtenemos como resultado el individuo H1. En este ejemplo la cantidad de links rotos es 6, pues inicialmente el valor 2 estaba conectado al valor 1 y 3, el valor 3 estaba conectado con los valores 2 y 4, el valor 4 estaba conectado con el 3, y el valor 5 estaba conectado al valor 6. Luego de aplicar la mutación por mezcla estos seis links se rompieron.

Figura 2.30 – Mutación por mezcla.

2.3.4.4 Representación real

El esquema general de la mutación es X = <x_1,…,x_n> → Y = <y_1,…,y_n> $x_i, y_i \in [LB_i, UB_i]$, surgen dos tipos de mutación al considerar la distribución de probabilidades a partir de la que se eligen los nuevos valores, mutación uniforme, mutación no uniforme.

Mutación uniforme

En el operador de mutación uniforme [76], partiendo de un único individuo padre X, se genera un único individuo hijo. Siendo X = <x_1,…,x_n>, al aplicar este operador se selecciona aleatoriamente un valor *k* con 1 <= k <= n y se genera como resultado al individuo Y = < x_1,…,y_k,…,x_n > donde y_k es un valor aleatorio con probabilidad de distribución uniforme en el rango [x_{k-1}, x_{k+1}].

Mutación no uniforme

El operador de mutación no uniforme [76], es uno de los operadores responsables de realizar los ajustes "finos" de la solución. Siendo X^t = <x_1,…,x_m> un individuo inicial, y x_k el elemento seleccionado para la mutación, donde x_k pertenece al dominio [x_{k-1}, x_{k+1}], 1 <= k <= n. El resultado de aplicar la mutación no uniforme es el vector X^{t+1} = < x_1,…,y_k,…,x_m > con 1 <=k <= n, y siendo:

$$y_k = \begin{cases} x_k + \Delta(t, right(k) - x_k) \; si \; el \; digito \; aleatorio \; vale \; 0 \\ x_k - \Delta(t, x_k - left(k)) \; si \; el \; digito \; aleatortio \; vale \; 1 \end{cases}$$

Donde $\Delta(t, y)$ retorna un valor en el rango [0,y] y la probabilidad de que $\Delta(t, y)$ se acerque a 0 se incrementa linealmente con el incremento de t. Dicha situación provoca que este operador realice una búsqueda inicialmente uniforme cuando t es pequeño y una búsqueda local a medida que t se incrementa. La fórmula utilizada para $\Delta(t, y) = y * r * (1 - \frac{t}{T})^b$ siendo r un numero aleatorio entre 0 y 1, *T* la máxima generación y *b* un parámetro del sistema que determina el grado de no uniformidad.

2.3.5 Mecanismos de reemplazo

Los mecanismos de reemplazo más conocidos son el mecanismo de reemplazo generacional, el mecanismo de reemplazo de estado estacionario (*steady-state*), el salto generacional y estrategias de elitismo [72]. A continuación se detallan cada uno de ellos.

2.3.5.1 Mecanismo de reemplazo generacional (Holland)

Se reemplaza completamente la población actual por sus descendientes. En este método todos los individuos son remplazados por los individuos de la nueva generación, no se mantienen individuos entre las generaciones. El algoritmo genético simple propuesto por Goldberg [8] en 1989 aplica este mecanismo de reemplazo.

2.3.5.2 Modelo de estado estacionario (steady-state)

En este modelo se reemplazan los *n* peores individuos de la población actual por los *n* mejores individuos del conjunto de hijos generados a partir de ésta. El paso atómico es el cálculo de un individuo.

2.3.5.3 Salto generacional (De Jong)

Se define un valor de "salto generacional" G ∈ [1/n, 1] para la fracción de la población reemplazada en cada generación.

2.3.5.4 Estrategias de elitismo

Con este modelo los individuos más adaptados de cada generación son privilegiados, y aseguran su supervivencia, es decir, pasan directamente a la siguiente generación.

2.3.6 Algoritmo Genético Simple (AGS) - Goldberg (1989)

El Algoritmo Genético Simple se caracteriza por utilizar representación binaria, selección proporcional (rueda de ruleta), cruzamiento de un punto, y mutación de inversión de valor de bit.

2.4 Modelos de paralelismo

Un AG aplica los diferentes operadores evolutivos (selección, cruzamiento y mutación) a una población, con el fin de generar una nueva población, hasta llegar a una condición de parada.

Cuando el problema que se tiene que resolver no es sencillo, la ejecución del AG puede demandar un gran tiempo de ejecución. Para solucionar este problema se ha propuesto la implementación de algoritmos genéticos paralelos, lo cual ha tenido mucho éxito debido a que la mayoría de los operadores evolutivos pueden ser fácilmente paralelizables. El uso de un AGP (algoritmo genético paralelo) lleva a una reducción en el tiempo de ejecución, pero además, suele ofrecer un rendimiento superior en cuanto a la calidad de las soluciones.

Existen diferentes criterios para clasificar los diseños propuestos de algoritmos genéticos paralelos. Se pueden identificar los siguientes criterios [17]: el modo de evaluación de la función de *fitness*

(concentrada o distribuida en diferentes procesos); el número de poblaciones utilizadas (modelo panmíctico que consiste de una única población, o utilizar poblaciones múltiples); el tamaño y organización de las subpoblaciones para modelos de poblaciones múltiples; el mecanismo de intercambio de individuos entre poblaciones múltiples; el modo de aplicación del mecanismo de cruzamiento (centralizado o distribuido); el conjunto de individuos considerados para la aplicación del mecanismo de selección (conjunto local o global); el mecanismo de sincronización entre los elementos de procesamiento. Una de las taxonomías más difundidas clasifica los modelos en: modelo maestro–esclavo, modelo distribuido, modelo celular, modelo de ejecución independiente, entre otros.

2.4.1 Modelo maestro–esclavo

Este modelo consiste en la distribución de las evaluaciones de la función de *fitness* entre varios procesadores esclavos, mientras que el ciclo principal del AG se ejecuta en un procesador maestro [20]. Cada nodo esclavo procesa una parte de la población total, y finalmente retorna el resultado al nodo maestro, que luego realiza el proceso de selección de individuos. La población en este modelo tiene comportamiento panmíctico, no hay estructura, o dicho de otro modo, todo individuo se puede combinar con otro. Estos algoritmos se comportan como un algoritmo genético secuencial porque solo paralelizan la evaluación del *fitness*. En la Figura 2.31 se muestra el modelo maestro – esclavo.

Figura 2.31 – Modelo Maestro – Esclavo.

La eficiencia de este modelo se incrementa a medida que la evaluación de la función de *fitness* es más costosa de computar, ya que el *overhead* de la comunicación se vuelve insignificante con respecto al tiempo de evaluación del *fitness*. Este modelo tiene una gran desventaja, el maestro debe esperar a que todos sus esclavos le retornen la evaluación de la función de *fitness* para continuar, por lo que está inactivo en todo ese intervalo de tiempo, que se incrementa cuando el tiempo de cálculo de la función de *fitness* es diferente para distintos individuos.

En las Figuras 2.32 y 2.33 [59] se muestra el pseudocódigo de un modelo maestro–esclavo. En este modelo se le delega al esclavo la evaluación de la función de *fitness*. El proceso maestro distribuye los individuos entre los procesos esclavos para que realicen la evaluación de la función de *fitness* y luego espera por el resultado de todos los esclavos para realizar el proceso de selección. El proceso esclavo recibe el individuo, lo evalúa y retorna el resultado al proceso maestro.

```
Mientras (NOT fin)
    Recibir(Individuo,i)
    Evaluar(Individuo)
    EnviarIndividuo(i,fitness(i))
Fin Mientras
```

Figura 2.32 – Algoritmo Esclavo (Modelo Maestro-Esclavo).

```
P,MP:Poblacion
t:generación
t=0
Inicializar (P(t));
Mientras NOT (Condición de Parada (P(t)))
   Para cada individuo en P(t) hacer:
      Enviar individuo a esclavo para evaluación de función de fitness
      Hasta no recibir el fitness de todos los individuos de la población:
         Recibir(i,fitness(i))
         ActualizarFitness(i)
      Fin Hasta
      MP(t)=Selección(P(t))
      MP(t)=Cruzamiento(MP(t));
      P(t+1)=Mutación(MP(t));
   Fin Para
   t=t+1
Fin Mientras
```

Figura 2.33 – Algoritmo Maestro (Modelo Maestro-Esclavo).

2.4.2 Modelo distribuido o de islas

En este modelo la población está estructurada en pequeñas sub-poblaciones relativamente aisladas unas de las otras. Conceptualmente la población general de un AG se divide en sub-poblaciones independientes, separadas, donde los operadores genéticos se llevan a cabo dentro de cada isla, lo que significa que cada isla puede buscar en regiones muy diferentes del espacio de búsqueda con respecto a otros. El modelo permite el intercambio de información entre las islas, a través de la migración de soluciones entre sub-poblaciones [20]. Estos modelos de migración pueden ser sincrónicos, donde cada cierto número de generaciones, las islas envían sus individuos seleccionados y esperan recibir individuos de otra isla; o asincrónico, donde cada ciertas generaciones las islas envían sus individuos pero no esperan recibir los individuos de otra isla, pudiéndolos recibir en cualquier momento [14]. En general, el mecanismo de migración asincrónica puede ser mejor que el mecanismo de migración sincrónica, ya que en este último se generan pérdidas de tiempo innecesarias cuando las islas esperan por la recepción de individuos de las otras islas.

En la Figura 2.34 se muestra el modelo distribuido o de islas.

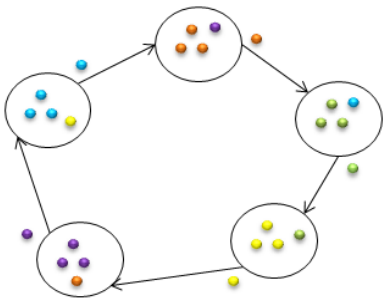


Figura 2.34 – Modelo de Islas.

En la Figura 2.35 se muestra el pseudocódigo de un algoritmo evolutivo que sigue el modelo de islas [62].

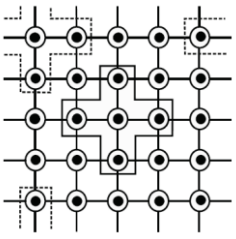
```
Inicializar(P(0))
generacion = 0
Mientras (NOT CriterioParada) hacer
   Asignar fitness(P(generacion))
   Padres = Seleccion(P(generacion))
   Hijos = Operadores de Reproduccion(Padres)
   NuevaPop = Reemplazar(Hijos, P(generacion))
   generacion ++
   P(generacion) = NuevaPop
   SI (CondicionMigracion)
      Emigrantes = SeleccionMigracion(P(generacion))
      Inmigrantes = Migracion(Emigrantes)
      Insertar(Inmigrantes, P(generacion))
 retornar Mejor Solucion Hallada
```

Figura 2.35 – Esquema de Algoritmo Evolutivo de población distribuida.

El esquema de la Figura 2.35, sigue en líneas generales el esquema presentado en la Sección 2.2, con la diferencia de que en este caso además de la evaluación del *fitness,* también se evalúa la condición de migración para determinar qué individuos migrar.

2.4.3 Modelo celular

El modelo celular o masivamente paralelo utiliza una distribución espacial de individuos en una única población. Esta población se divide en nodos que contienen muy pocos o un único individuo, la principal característica de este modelo es la estructuración en vecindarios, donde cada individuo solo puede interactuar con sus vecinos y los operadores genéticos son ejecutados paralelamente por cada nodo, por tanto, la selección, el cruzamiento y la mutación se llevan a cabo considerando sólo los individuos adyacentes. Como las buenas soluciones posiblemente surjan en diferentes áreas de la topología general, se extienden lentamente a lo largo de toda la población. Se conoce a este fenómeno como difusión [20]. En la Figura 2.36 (A) se muestra el modelo celular, y en la Figura 2.36 (B) se muestra un pseudocódigo del modelo celular [59]. En este caso, cada nodo contiene un único individuo, el proceso de selección se efectúa sólo sobre sus vecinos y se elige un sólo individuo para cruzarse con el actual. Luego de obtener el nuevo individuo, si éste es mejor que el anterior, sustituye al individuo anterior.



(a)

```
Mientras NOT (Condicion de Parada)
   Evaluar (individuo i)
   Seleccionar de la vecindad un individuo k
   L=Cruzamiento(i,k)
   Lm=Mutación(L)
   SI Lm mejor que i
         i=L
Fin Mientras
```

(b)

Figura 2.36: (a) Modelo Celular, (b) Pseudocódigo modelo celular.

2.4.4 Modelos de ejecución independiente

En este modelo, se realiza la ejecución del algoritmo genético en forma totalmente independiente, pero en paralelo y no hay interacción alguna entre las ejecuciones. Este método es muy útil para la ejecución de trabajos simultáneos, como por ejemplo, para ejecutar varias versiones del mismo problema con diferentes condiciones iniciales [20].

2.4.5 Otros modelos

Es común encontrar muchas implementaciones que son difíciles de clasificar, en general, se denominan algoritmos paralelos híbridos, ya que implementan características de diferentes modelos de paralelismo. Por ejemplo, un algoritmo implementado con el modelo de islas, pero que cada isla realice sus pasos evolutivos mediante otro modelo que puede ser el maestro esclavo o el celular. En la Figura 2.37 se muestran ejemplos de modelos híbridos [20].

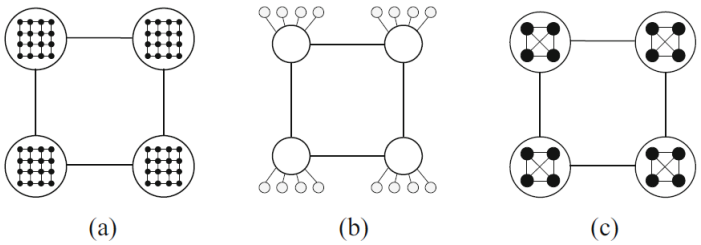


Figura 2.37 – Modelos Híbridos: (a) AG celular distribuido, (b) AG maestro-esclavo distribuido y (c) AG de dos niveles distribuido.

2.5 Frameworks para Algoritmos Evolutivos

En esta sección se presenta un relevamiento de algunos de los *Frameworks* más populares para la implementación de Algoritmos Evolutivos.

2.5.1 Galib

Galib [21] es una biblioteca de C++ que incluye herramientas para utilizar Algoritmos Genéticos en cualquier programa desarrollado en C++. Al utilizar Galib se trabaja con dos clases básicas: un genoma y un algoritmo genético, cada genoma representa una solución al problema, y el algoritmo genético define como será llevada a cabo la evolución.

El usuario debe definir tres puntos importantes, la representación de los individuos, los operadores genéticos y la función de *fitness*. Galib provee ejemplos y brinda ayuda para crear representaciones y operadores genéticos, tales como estructuras de datos predefinidas (listas, árboles, vectores y *strings* de bits), pero el usuario puede crear su propia estructura si lo desea. La función de *fitness* debe ser implementada completamente por el usuario. La biblioteca incluye cuatro tipos básicos de algoritmos genéticos: algoritmo genético simple, incremental, de estado estacionario (*steady-state*) y el modelo de islas. El algoritmo genético simple se detalló en la Sección 2.3.6. El algoritmo genético incremental [13] está basado en el modelo de islas pero sin migraciones. El algoritmo *steady-state* [9] utiliza poblaciones solapadas, es decir, uno puede especificar qué porcentaje de la población vieja va a ser reemplazado por la nueva. El algoritmo genético con el modelo de islas se especificó en la sección 2.4.2. Adicionalmente, Galib, define los componentes que el usuario necesita para derivar sus propias clases de algoritmos genéticos.

Una ventaja de esta biblioteca es que está implementada en un lenguaje orientado a objetos, que permite modularidad y eficiencia, además provee funcionalidades y clases fácilmente extensibles por el usuario. Pero tiene como desventajas que implementa únicamente el modelo de islas, y a pesar de brindar ayuda para la implementación paralela en múltiples procesadores, el usuario terminará adaptando el modelo de islas. Además, Galib no fue diseñada para dar soporte a la programación *multithread*, por ejemplo, las funciones aleatorias que posee la biblioteca no pueden utilizarse para estos contextos y para poder soportar el modelo maestro-esclavo se deben realizar varias modificaciones, incluyendo, la implementación de la creación, el manejo y la sincronización de *hilos*, y aplicación de mutua exclusión en la selección, en la población y en variables estadísticas.

2.5.2 Mallba

El objetivo del proyecto Mallba [66] es el desarrollo de una biblioteca de esqueletos algorítmicos para la resolución de Problemas de Optimización Combinatoria. El nombre de Mallba, se debe a que el proyecto fue llevado a cabo por las universidades de Málaga (MA), La Laguna (LL) y Barcelona (BA). Mallba proporciona un esquema algorítmico de algunas de las metaheurísticas más populares que se pueden aplicar para resolver un problema dado. La arquitectura es flexible y extensible, por lo cual pueden agregarse nuevos esqueletos de software de un modo relativamente simple. Toda la biblioteca está desarrollada en C++, lenguaje de alto nivel orientado a objetos que provee modularidad y eficiencia. Las tres principales características que se pueden destacar de la biblioteca Mallba son la integración de todos los esqueletos bajo los mismos principios de diseño, la facilidad de cambiar entre entornos secuenciales y paralelos, y la cooperación entre los esqueletos para proporcionar nuevos entornos híbridos. Un esqueleto Mallba es un esquema algorítmico que implementa un método de resolución de problemas de optimización. Para que dicho esquema resuelva un problema concreto es necesario que se especifiquen sus características siguiendo un formato determinado, este proceso se llama Instanciación del Esqueleto. Los métodos de resolución genéricos existentes en Mallba son: Algoritmo Genético (AG), Simulated Annealing (SA), Ant Colony Optimization (ACO), entre otros.

Los esqueletos de Mallba se separan en dos componentes, el método de resolución a utilizar y el problema a resolver. Las características particulares relacionadas al problema deben ser provistas por el usuario, mientras que las características relacionadas con el método de resolución genérico son implementadas en el esqueleto, así como también la resolución de los problemas de paralelismo. Los esqueletos son implementados por un conjunto de clases requeridas (definidas por el usuario) y provistas (definidas por Mallba), que representan la abstracción de las entidades participantes en la resolución de los métodos, las clases provistas implementan los aspectos internos de los esqueletos en un problema independiente, y las clases requeridas especifican información y comportamiento relacionado al problema. Cada esqueleto incluye las clases requeridas *Problem* y *Solution,* que encapsulan las entidades dependientes del problema necesarias para el método de resolución. Incluye además, dos clases, *Solver* y *SetUpParams*, que proveen métodos para la ejecución del esquema de resolución y para configurar parámetros necesarios para la ejecución respectivamente. La información requerida por el *Solver,* es una instancia del problema a resolver y los parámetros de configuración. En la Figura 2.38 se muestra el diagrama de diseño de los esqueletos Mallba.

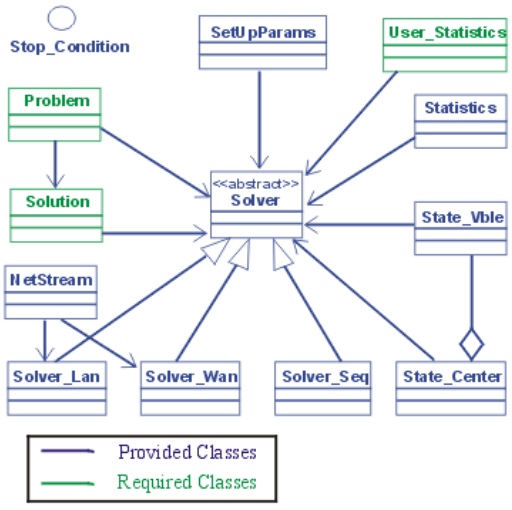


Figura 2.38 – Diagrama de diseño de los esquletos Mallba.Extraído de [66].

Los patrones que proporciona la biblioteca Mallba se agrupan en tres familias según la naturaleza de la

solución que se obtiene, estos son: Métodos Exactos, Métodos Heurísticos y Métodos Híbridos. Para algoritmos genéticos, se deben configurar los siguientes parámetros: número de ejecuciones independientes, número de generaciones, tamaño de la población, número de hijos creados en cada generación, estrategia de reemplazo, parámetros de la selección, parámetros de los operadores de cruzamiento y mutación. Mallba proporciona los siguientes métodos de selección: aleatorio, por torneo, rueda de ruleta, por rango y selección elitista. Entre los problemas de ejemplo resueltos en Mallba se encuentran: OneMax [32], MaxSat [45], entre otros. En estos se implementaron operadores de cruzamiento tradicionales como SPX, 2PX, UPX, cruzamientos para permutaciones (PMX, UX) y operadores de mutación de inversión de bits, mutaciones para permutaciones, entre otros.

2.5.3 Paradiseo

Paradiseo es un *framework* para la implementación de metaheurísticas [57]. Este *framework* al igual que Galib y Mallba está desarrollado en C++. Es de código abierto y está en continuo desarrollo, ya que es un *framework* en el cual toda la comunidad puede contribuir. Entre los problemas tratados por Paradiseo se encuentran: TSP [15], ZDT [77], y DTLZ [37]. Algunas representaciones de algoritmos en Paradiseo son: representación de permutación, representación de vector real de largo variable, representaciones específicas, entre otras. Los operadores de cruzamiento de los algoritmos que se han implementado son: mapeo parcial, cruzamiento SBX, cruzamiento específico y 2PX. Los operadores de mutación son: basada en el intercambio, mutación polinomial, específica, *shift swap*, entre otros.

La arquitectura de Paradiseo es una arquitectura basada en módulos. En la Figura 3.39 se muestra el esquema de la arquitectura de Paradiseo.

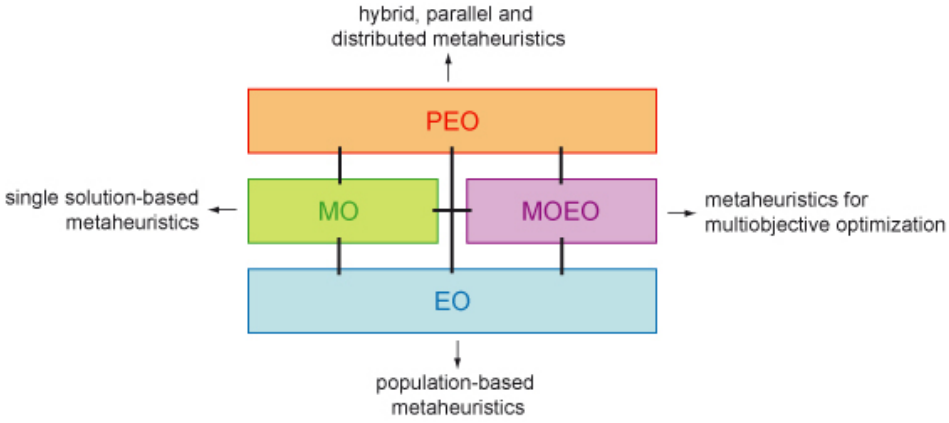


Figura 2.39 – Esquema de arquitectura de Paradiseo – Extraído de [13].

Paradiseo provee extensiones EO [18] (Evolutionary Objects) para contar con un diseño flexible para la resolución de metaheurísticas.

3 Computación de Propósito General en Unidades de Procesamiento Gráfico

Resumen: En este capítulo se presenta el tema computación de propósito general en unidades de procesamiento gráfico (General purpose graphics processing unit, GPGPU). La estructura del capítulo es la siguiente. En la primera sección se presenta una introducción a la computación de propósito general en unidades de procesamiento gráfico. En la segunda sección se presenta una breve revisión histórica de la arquitectura de la GPU. En la tercera sección, se presenta la arquitectura de las GPUs en la actualidad, y en particular la arquitectura Fermi, que es el modelo de arquitectura en el que se basa este proyecto. En la cuarta sección se introduce CUDA (Compute Unified Device Architecture), plataforma de computación paralela y modelo de programación creado por NVIDIA, se describe también la memoria en CUDA. En la quinta sección se discuten consideraciones sobre la performance al utilizar GPUs para programas de propósito general. La sexta describe los algoritmos aplicables a GPU, y finalmente, la última sección cuenta con un relevamiento de trabajos realizados para acelerar algoritmos evolutivos sobre GPU.

3.1 Introducción

Una GPU es un procesador originalmente diseñado para realizar los cómputos asociados a la generación de gráficos, pero debido a sus prestaciones, en los últimos años comenzó a utilizarse para el cálculo científico. Algunas de sus características son, su bajo costo en relación a su potencia de cálculo, su capacidad de procesamiento paralelo (generación de "millones" de hilos independientes ejecutando en paralelo), y su uso para aplicaciones gráficas, consideradas atractivas en el ámbito científico y de simulación.

Por otra parte, GPGPU *(General Purpose Graphics Processing Unit)* es un concepto bastante reciente dentro de la informática, que trata de aprovechar las capacidades de cómputo de una GPU para realizar operaciones de cálculo científico o técnico, es decir, de propósito general. En los últimos años existe un auge en esta técnica. Este crecimiento se basa fundamentalmente en que la arquitectura es intrínsecamente paralela, en contraste con la arquitectura secuencial de las CPUs. El modelo empleado para esta tecnología se basa en el uso combinado de una CPU y una GPU, en un sistema de co-procesamiento heterogéneo. En la CPU se ejecuta la parte secuencial de la aplicación, mientras que las partes de mayor carga computacional se aceleran en la GPU. Para el usuario, la aplicación se ejecuta más rápido ya que además de utilizar los recursos brindados por la CPU, aprovecha la capacidad de la GPU para multiplicar el rendimiento. En la Figura 3.1 se muestra una comparación típica del número de *cores* en CPU y en GPU.

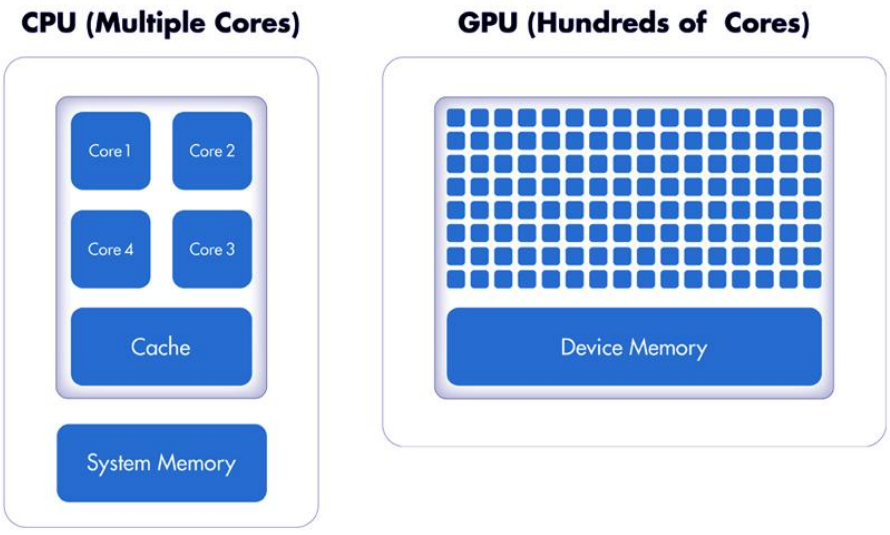


Figura 3.1 – Cores en CPU vs. Cores en GPU – Extraída de [51].

La GPU ha evolucionado hasta alcanzar *teraflops* de rendimiento en las operaciones de cálculo de punto flotante. NVIDIA [53] revolucionó la GPGPU con la introducción de su plataforma de cálculo paralelo masivo "CUDA" (2006-2007), compuesta por cientos de *cores* de procesamiento que trabajan juntos para manejar los datos de la aplicación. Esta revolución está vinculada a la unificación de los "*shaders*" en un único tipo de unidades de procesamiento, y a la disponibilidad de un lenguaje de alto nivel para el desarrollo de aplicaciones que utilicen la GPU (CUDA – C).

En la Figura 3.2 se muestra el algoritmo básico de programación paralela sobre GPU. Cabe mencionar que a la CPU, como es el procesador principal del sistema heterogéneo, se la denomina *Host*.

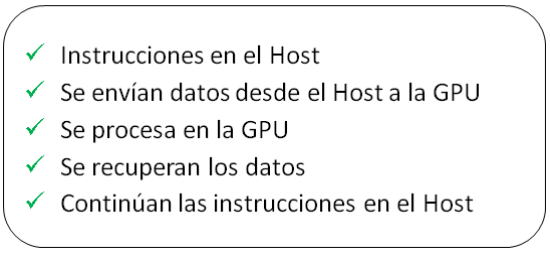


Figura 3.2 – Algoritmo básico de programación paralela sobre GPU.

3.2 Revisión histórica de la arquitectura de la GPU

A continuación se presenta la evolución histórica del pipeline gráfico y el pasaje a la arquitectura unificada, lo que ayuda a comprender las decisiones de diseño de la arquitectura de las GPUs programables de la actualidad, como el *multithreading* masivo, las memorias caché relativamente pequeñas, entre otras [46].

3.2.1 Evolución de los pipelines gráficos

Los *pipelines* gráficos [46] pasaron de ser sistemas muy grandes y costos a principios de la década de los 80' a ser muy pequeños y reducir significativamente sus costos a fines de la década de los 90'. El costo pasó del entorno de 50.000 dólares a principios de los años 80' a 200 dólares a fines de la década de los 90'. Durante este período también aumentó la *performance*, tanto a nivel de número de píxeles como de vértices por segundo, en los años 80' la *performance* pasó de ser de 100.000 vértices por segundo y de 50.000.000 píxeles por segundo, a ser de 10.000.000 de vértices por segundo y de 1.000.000.000 de píxeles por segundo respectivamente. Estos avances se debieron fundamentalmente a las innovaciones en algoritmos gráficos y diseños de *hardware*, así como también a las exigencias del mercado de obtener gráficos en tiempo real y de alta calidad.

Entre la época de los 80' y los 90' el mejor rendimiento de *hardware* gráfico eran los *pipelines* de función fija (*fixed-function pipelines*) que eran configurables pero no programables. También surgieron bibliotecas APIs (*Application Programming Interface*) que permitieron hacer uso de *hardware* o *software* por aplicaciones tales como videojuegos. En la Figura 3.3 se muestra el *pipeline* gráfico en los comienzos de las GPUs NVIDIA *GeForce*.

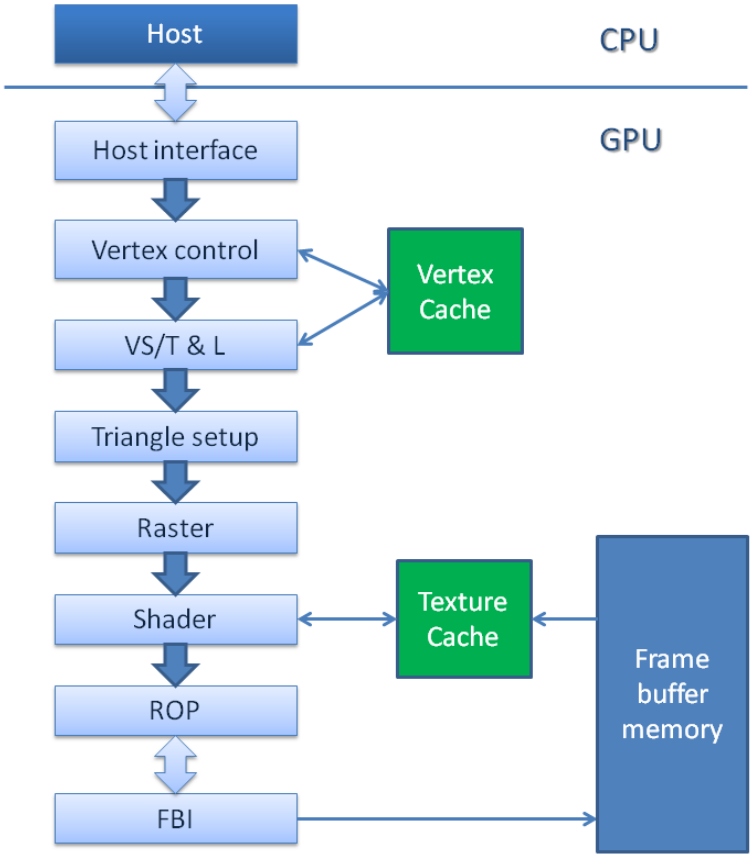


Figura 3.3 – Pipeline gráfico de la GPU NVIDIA GeForce. Extraída de [46].

La interfaz del *host (Host interface)* recibe comandos gráficos y datos de la CPU, estos comandos son enviados por aplicaciones mediante llamadas a una función de la API. La interfaz del *host* posee *hardware* específico para el acceso directo a memoria (DMA) lo que posibilita la transmisión eficiente de datos entre el *host* y la memoria del sistema. El *pipeline* de la GPU *GeForce* está diseñado para reproducir triángulos, el control de vértices *(Vertex control)* recibe los datos de los triángulos y luego los convierte a un formato entendible por el *hardware* y los envía a la caché de vértices *(Vertex cache).* En VS/T&L *(Vertex shading, transform, and lighting)* se aplica el sombreado, la transformación e iluminación de vértices. En la configuración del triángulo *(Triangle setup)* se crean ecuaciones que luego son utilizadas para interpolar colores u otros datos de los vértices a través de los píxeles contenidos en el triángulo. El rasterizador *(Raster)* determina qué píxeles están contenidos en qué triángulos, mientras que en el sombreado *(Shader),* se determina el color de cada píxel. En la operación de rasterizado *(Raster Operation - ROP) se* realizan las operaciones finales de rasterizado de cada píxel, y en la interfaz de marco del buffer *(Frame Buffer Interface - FBI)* se manejan las lecturas y escrituras a la memoria del marco del buffer de visualización.

Durante dos décadas, cada generación de *hardware* y su respectiva API mejoraban varios aspectos del *pipeline* gráfico, introduciendo recursos de *hardware* y configuración de las etapas del *pipeline*. Sin embargo, todas las mejoras propuestas por los desarrolladores, sólo podían ser ofrecidas como funciones fijas, ya que para no serlo, requerían de características con las que los *pipelines* no contaban. Es por esto que el siguiente paso fue introducir los procesadores programables.

Los procesadores gráficos programables con arquitectura unificada fueron introducidos en la GPU *GeForce* 8800 en 2006 [46], estos procesadores mapean las etapas separadas de gráficos programables a un arreglo de procesadores unificados. En la Figura 3.4 se muestra el *pipeline* gráfico lógico. Este

pipeline es físicamente un camino que recircula visitando tres veces los procesadores, con muchas más funciones fijas entre cada visita. El arreglo de procesadores unificados permite particionar dinámicamente el arreglo de sombreado de vértices, el procesamiento de la geometría y de píxeles. Esta unificación permite que el mismo conjunto de recursos permita la ejecución de diferentes etapas del *pipeline* y así alcanzar un mejor balance de carga.

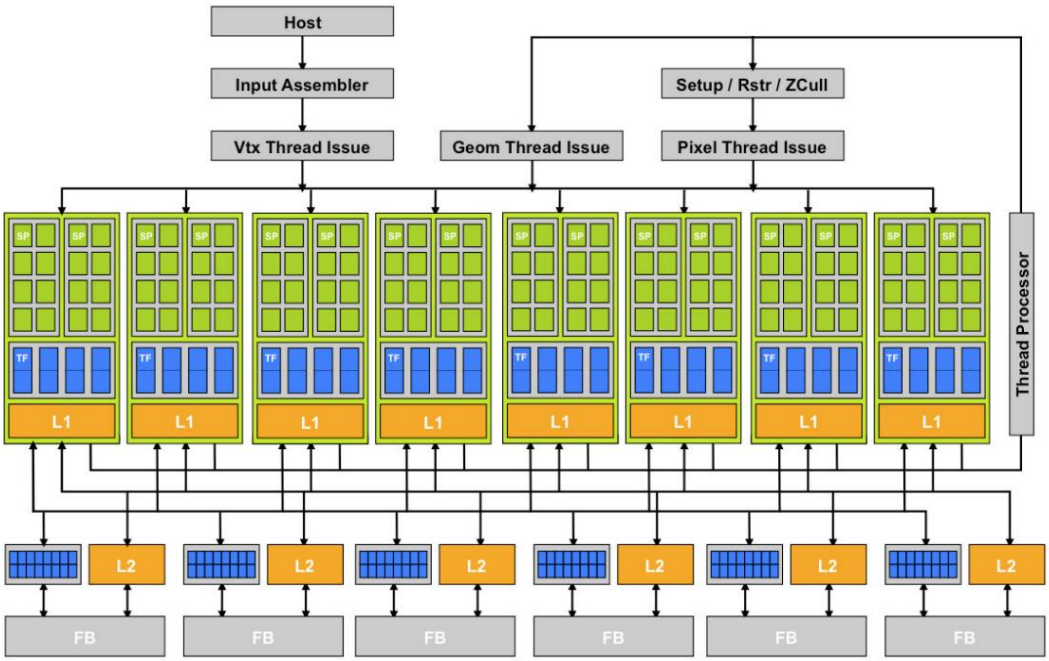


Figura 3.4 – Arreglo de procesadores con arquitectura unificada del pipeline gráfico de la GPU GeForce 8800. Extraído de [46].

Al inicio, la programación de la GPU se realizaba con llamadas a servicios de interrupción de la BIOS. Se programaba en lenguaje ensamblador específico a cada modelo. Un tiempo después, se agregó un nivel más entre el *hardware* y el *software*, con la creación de interfaces de programación de aplicaciones (API) específicas para gráficos, que proporcionaron un lenguaje más homogéneo para los modelos existentes en el mercado. La primera API usada ampliamente fue el estándar abierto *OpenGL*, tras el cual Microsoft desarrolló *DirectX*. Luego del desarrollo de la API, se decidió crear un lenguaje más natural y cercano al programador, es decir, desarrollar lenguajes de alto nivel para gráficos. El lenguaje estándar de alto nivel, asociado a la biblioteca *OpenGL* es el *OpenGL Shading Language* (GLSL), implementado en principio por todos los fabricantes. En particular, NVIDIA creó un lenguaje propietario llamado Cg (*C for graphics*), con mejores resultados que GLSL en las pruebas de eficiencia. En colaboración con NVIDIA, Microsoft desarrolló su *High Level Shading Language* (HLSL), prácticamente idéntico a Cg, pero con ciertas incompatibilidades menores. Sin embargo, estas herramientas continuaban siendo muy dependientes de la arquitectura de la GPU, del modelo, entre otros [46].

3.2.2 Computación en GPU

NVIDIA consideró que el uso potencial de las GPUs sería mayor, si los desarrolladores pudieran pensar en éstas cómo un procesador de propósito general, por lo que se basó en un enfoque de programación en el cual los desarrolladores pudieran declarar explícitamente los aspectos de paralelismo de datos. Para la décima generación de gráficos *DirectX*, NVIDIA trabajó en mejorar la eficiencia de operaciones enteras y de punto flotante, que podían ejecutar una variedad de cargas de trabajo simultáneamente. Luego agregó carga de memoria e instrucciones de almacenamiento con capacidad de direccionamiento de bytes aleatorio, lo que permite soportar los requerimientos de programas compilados en C, desarrolló también, el compilador CUDA C/C++, bibliotecas, y *software,* para permitir a los

programadores fácil acceso a nuevos modelos de computación de paralelismo de datos, y de aplicaciones ya implementadas. Los programadores ya no necesitaban utilizar la API para acceder a la capacidad de procesamiento paralelo de las GPUs. El *chip* G80 fue utilizado en la *GeForce* 8800 GTX, que luego fue sucedido por G92 y GT200 [46].

3.3 Arquitectura

A continuación se describe la arquitectura de la GPU y del modelo de Fermi en particular.

3.3.1 Arquitectura de la GPU

La arquitectura de una GPU es muy distinta a la de una CPU. En una CPU tradicional gran parte de los transistores están dedicados a realizar otro tipo de tareas, como por ejemplo, predicción de *branches*, *prefetch* de memoria, ejecución fuera de orden y caché de datos. En las GPUs hay más transistores dedicados al cálculo, y son altamente segmentadas, por lo cual poseen gran cantidad de unidades funcionales. La Figura 3.5 muestra el esquema de arquitectura de la GPU en contraste con el esquema de la CPU.

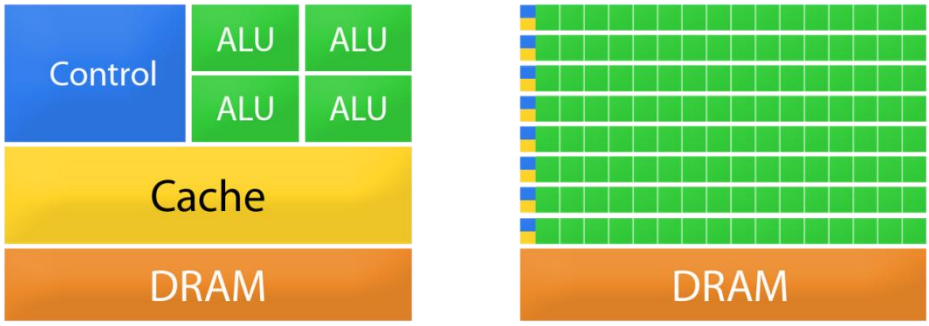


Figura 3.5 – Arquitectura de CPU vs. Arquitectura de GPU. Extraída de [51].

La CPU es mejor para la ejecución de cargas de trabajo dinámicas y control de flujo impredecible, y la GPU por el contrario es mejor para la ejecución de cargas de trabajo con un control de flujo simple y bien definido, ya que optimizan la ejecución de cálculo de *streaming*.

3.3.2 Arquitectura CUDA Fermi

La arquitectura Fermi [73] es la penúltima generación lanzada al mercado por NVIDIA (la última generación recientemente desarrollada es la arquitectura Kepler [74]). A continuación se detallan las principales características de la arquitectura Fermi, que es la arquitectura en la que se basa este proyecto.

La *performance* lograda por la arquitectura Fermi supera ampliamente la *performance* de las CPUs convencionales que ejecutan *software* paralelo y es la primera que brinda todas las funcionalidades requeridas para aplicaciones de computación de alta *perfomance,* tales como: el alto nivel de *performance* en operaciones de doble precisión y de punto flotante, un único *chip* que cuenta con una jerarquía de memoria flexible y con protección de errores, soporte a operaciones *fused multiply-add* (FMA) que siguen el estándar IEEE 754-2008 [28] y soporte a una gran variedad de lenguajes de programación que incluyen C++, FORTRAN, Java, Matlab, y Python, entre otros.

La complejidad de la arquitectura Fermi, es manejada por un modelo de programación multinivel que incrementa la productividad, ya que permite a los desarrolladores focalizarse en el diseño del algoritmo

y no en el mapeo del algoritmo al *hardware*. En la Figura 3.6 se muestra un diagrama de bloques de alto nivel del primer *chip* Fermi.

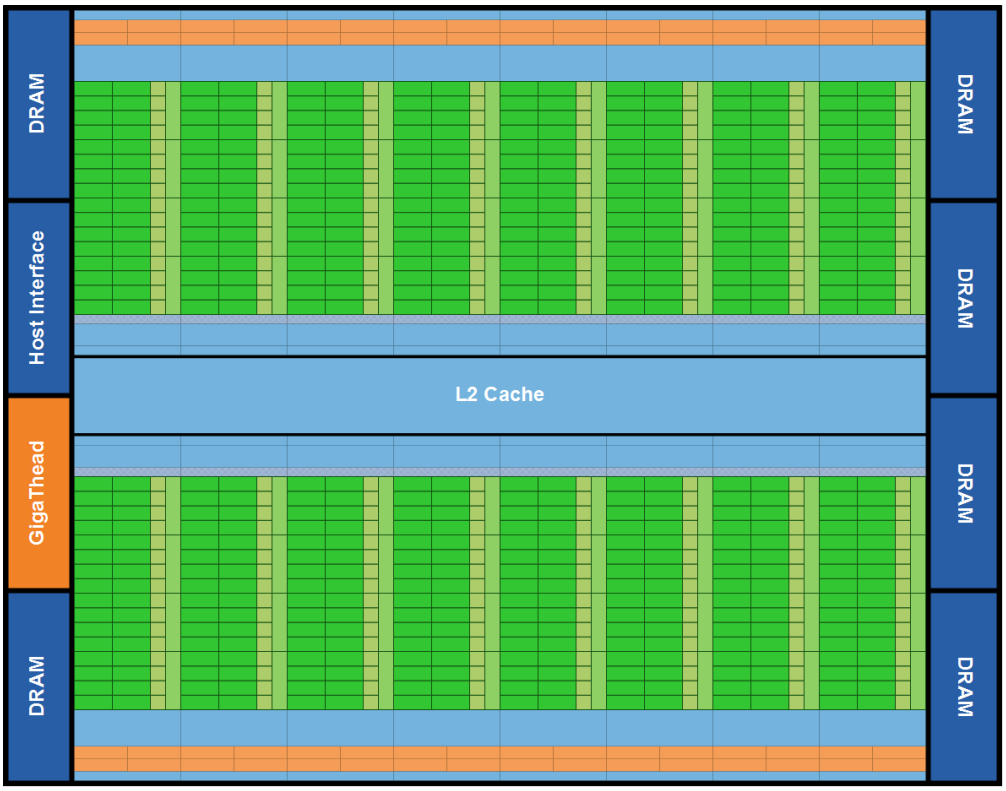


Figura 3.6 – Primer chip Fermi – Extraída de [73].

La arquitectura de GPU Fermi consiste en multiprocesadores (SMs) de *streaming,* cada uno consta de 32 *cores* que pueden ejecutar una instrucción entera o de punto flotante por ciclo de reloj, 16 unidades para operaciones de acceso a memoria (*load – store*), cuatro unidades para operaciones especiales (SFU – *Special Functions Units*), y 64K de SRAM (*Static Random Access Memory*) dividida entre la memoria caché y la memoria local. En la Figura 3.7, se muestra un esquema de los SMs de Fermi.

Dentro de cada SM, los *cores* están divididos en dos bloques de ejecución de 16 *cores* cada uno, que junto con las 16 unidades para operaciones *load-store* y las cuatro unidades para operaciones especiales (SFU) forman cuatro bloques de ejecución. En cada ciclo, se pueden despachar 32 instrucciones hacia estos bloques de ejecución desde uno o dos *warps*. Para las 32 instrucciones de cada *warp*, la ejecución en los *cores* o en las unidades *load-store* lleva dos ciclos. Un *warp* de 32 instrucciones de funciones especiales lleva un único ciclo, pero lleva ocho ciclos en completarse en las cuatro unidades de SFU. La Figura 3.8 muestra la distribución de una secuencia de instrucciones en los bloques disponibles.

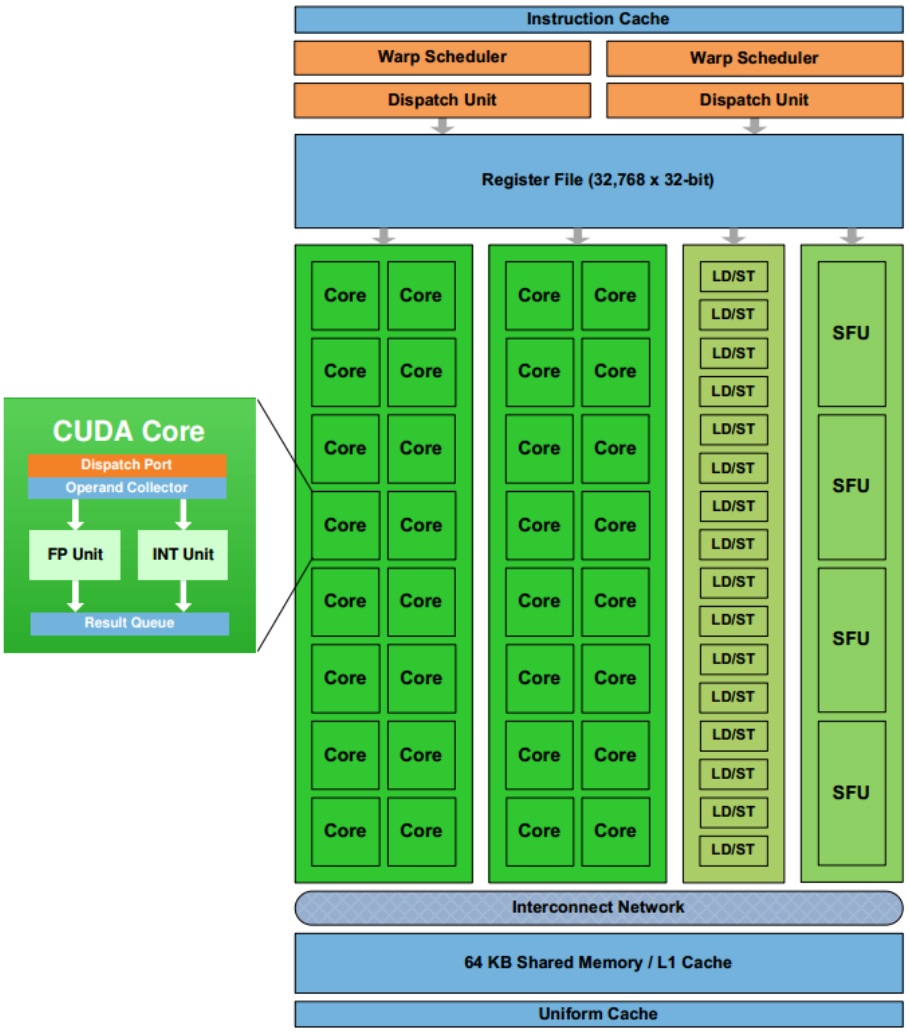


Figura 3.7 – Esquema de SMs en Fermi – Extraído de [73].

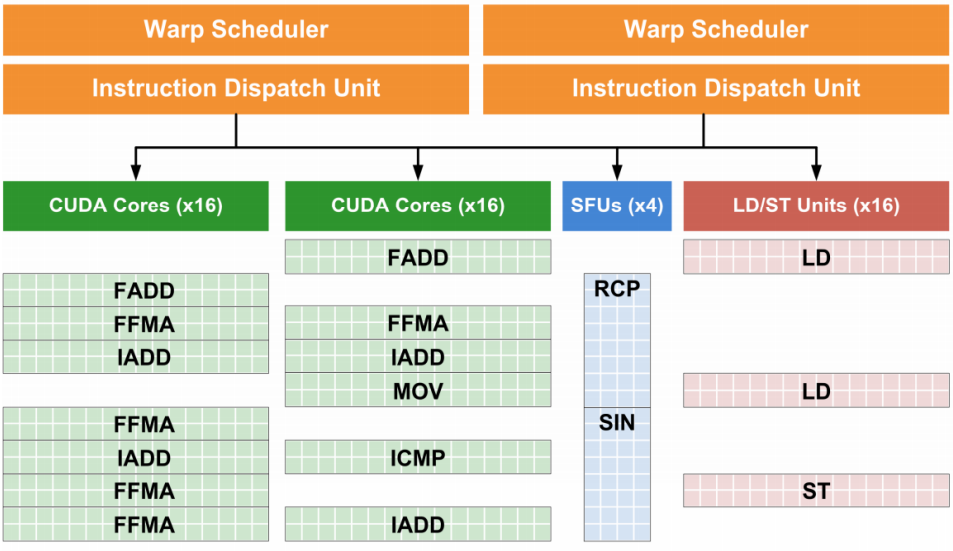


Figura 3.8 – Distribución de secuencia de instrucciones dentro de un SM – Extraído de [73].

3.4 CUDA

CUDA *(Compute Unified Device Architecture)* es una arquitectura de computación paralela unificada sin distinción entre procesadores de píxeles y vértices, utilizada para el cómputo de problemas de propósito general, presentada por NVIDIA en 2007. CUDA, produjo un cambio radical en la arquitectura y en el *software* para desarrollo de aplicaciones de las GPUs NVIDIA. La arquitectura está enfocada al cálculo masivamente paralelo y en las capacidades de procesamiento que brindan las tarjetas gráficas de NVIDIA, además, permite programar los dispositivos a través de extensiones de lenguajes de programación estándar (C y Fortran) y está disponible para las tarjetas gráficas *GeForce* de la serie 8 en adelante. Es compatible con Linux de 32/64 bits y Windows XP (y sucesores) de 32/64 bits.

La programación en CUDA sigue el modelo de programación SPMT *(Single Program Multiple Thread)*, este modelo es similar a SPMD *(Single Program Multiple Data)* ya que cada multiprocesador ejecuta el mismo programa sobre distintos datos, pero con la diferencia de que cada multiprocesador no necesariamente tiene que estar ejecutando la misma instrucción al mismo tiempo.

3.4.1 Funciones CUDA

En CUDA, a las funciones que se ejecutan sobre la CPU se las conoce como funciones del *host*, las que se ejecutan sobre la GPU se denominan funciones del *device*. Las funciones que especifican el código a ser ejecutado por todos los hilos en forma paralela, se denominan *kernels*.

En general, es posible realizar declaraciones en C (CUDA) para *kernels* que se ejecutan en la GPU y sólo se pueden llamar desde la CPU, funciones del *device* que se ejecutan en la GPU y sólo se pueden llamar desde otro *kernel*, y para funciones del *host*, que son funciones tradicionales de C que se ejecutan en el *host* y sólo se pueden llamar desde funciones del *host*.

3.4.2 Estructura de un programa CUDA

Un programa CUDA cuenta con una o más fases que pueden ser ejecutadas en una CPU o en un dispositivo como una GPU. Las fases que no exhiben o exhiben poco paralelismo de datos suelen ser implementadas sobre la CPU, mientras que las fases que presentan gran paralelismo de datos, se implementan sobre la GPU, tratando siempre de minimizar las transferencias de datos entre la CPU y la GPU, ya que reducen notoriamente la *performance* como se explica en la Sección 3.5.

3.4.3 Arreglo de hilos en CUDA

Un *kernel* es ejecutado por un arreglo de hilos, a cada hilo se le asigna un identificador denominado thread*Idx,* que es utilizado para computar direcciones de memoria y tomar decisiones de control.

El lanzamiento de un *kernel* CUDA crea un *grid* de hilos*,* donde todos los hilos ejecutan el mismo *kernel*. Los hilos están organizados en una jerarquía de dos niveles y se utiliza la coordenada *blockIdx* para identificar el índice de bloque y *threadIdx* para identificar el índice de hilo, estos identificadores son asignados en tiempo de ejecución. La utilización de bloques de hilos*,* hace posible un alto nivel de escalabilidad, ya que permite dividir un arreglo de hilos en múltiples bloques. Los hilos de un mismo bloque cooperan mediante memoria compartida, mientras que los que se encuentran en bloques diferentes pueden cooperar utilizando memoria global. Los bloques dentro de un *grid* son independientes, ya que son ejecutados en cualquier orden y además, pueden ser ejecutados en multiprocesadores diferentes. Las llamadas a un *kernel* son asincrónicas, por lo que es necesario utilizar sincronizaciones explícitas. En la Figura 3.9 se muestra un diagrama de jerarquía de hilos en el modelo de programación CUDA.

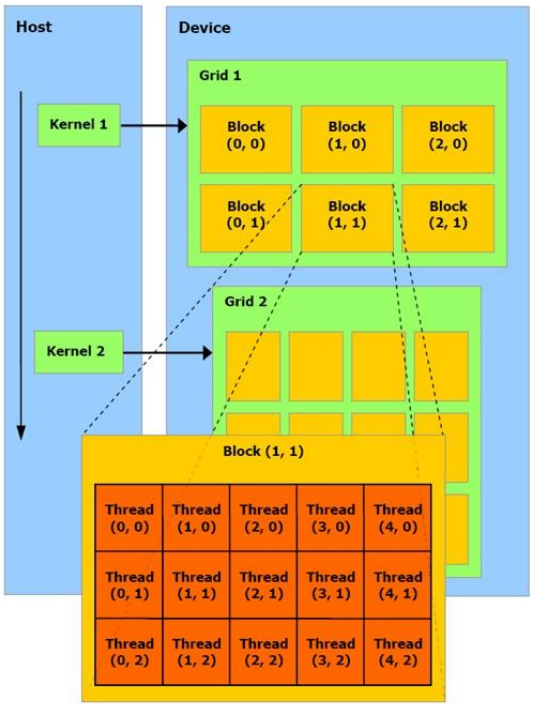


Figura 3.9 – Esquema de jerarquía de hilos en CUDA.Extraída de [73].

3.4.5 Memoria en CUDA

CUDA especifica una jerarquía de memoria compuesta por: memoria global (todos los hilos que ejecutan en la GPU tienen el mismo espacio global de memoria); memoria local (cada hilo tiene su propia memoria local, es de las memorias más lentas de la GPU); memoria compartida (cada bloque tiene un espacio de memoria compartida que es prácticamente tan rápida como los registros); registros (es la más rápida de la GPU y sólo son accesibles por cada hilo); memoria constante (es una memoria rápida solamente de lectura) y texturas (tiene características similares a la memoria constante). En la Tabla 3.1 se muestra un resumen de las características de cada tipo de memoria y en la Figura 3.10 se muestra un esquema de la jerarquía de memoria en CUDA.

	Ubicación	Caché	Acceso	Alcance	Existencia
Constante	*off-chip*	Sí	R	*Device*	Aplicación
Local	*off-chip*	No	R-W	*Thread*	Aplicación
Global	*off-chip*	No	R-W	*Device*	Aplicación
Compartida	*chip*	No	R-W	Bloque	Bloque
Registros	*chip*	No	R-W	*Thread*	*Thread*
Texturas	*off-chip*	Sí	R-W	*Device*	Aplicación

Tabla 3.1 – Resumen de Jerarquía de Memoria en CUDA.

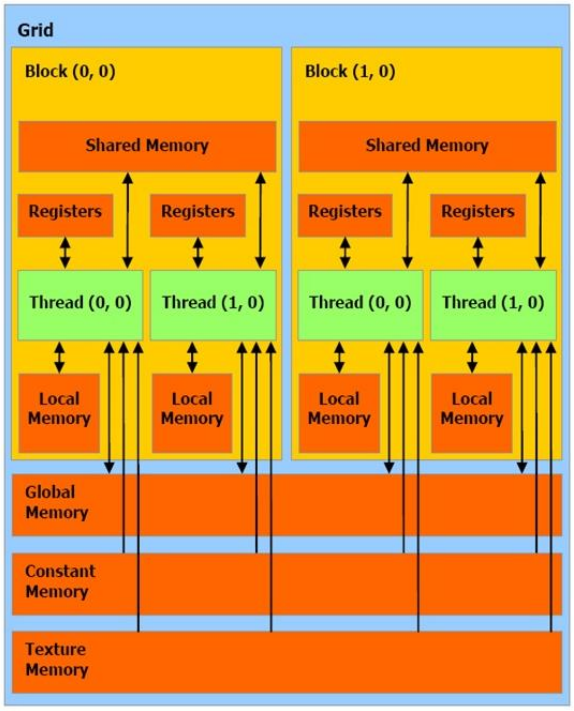


Figura 3.10 – Esquema de jerarquía de memoria en CUDA. Extraído de [53].

3.4.5.1 Memoria compartida

La memoria compartida es cientos de veces más rápida que la memoria global pero es más reducida, por tanto puede ser usada como una caché, con el fin de reducir los accesos a memoria global. Además, permite que los hilos de un mismo bloque puedan cooperar, y se puede utilizar para evitar accesos no *coalesced* (ver Anexo I) a la memoria global, esto se realiza de la siguiente forma:

– Los datos se almacenan en forma intermedia en la memoria compartida.

– Se reordena el acceso a los datos para que cuando se copien de memoria compartida a memoria global el acceso sea *coalesced*.

Debido a que se encuentra en el *chip* de la GPU, la memoria compartida tiene un ancho de banda mucho mayor, y una latencia mucho menor que la memoria local o global. Para lograr un mayor ancho de banda, la memoria compartida se divide en módulos del mismo tamaño llamados bancos, que pueden ser accedidos simultáneamente. Si las lecturas o escrituras sobre distintas direcciones de memoria son todas en bancos distintos, éstas pueden ser atendidas simultáneamente, obteniéndose un ancho de banda *n* veces mayor al ancho de banda del módulo. Si dos solicitudes de acceso a memoria caen en el mismo banco, se produce un conflicto de bancos y el acceso debe ser serializado. El *hardware* divide cada solicitud con conflicto de banco, en tantas solicitudes como sean necesarias para eliminar el conflicto, decrementando el rendimiento en un factor igual al número de peticiones de memoria separadas. Cuando todos los hilos del *half-warp* acceden a diferentes bancos, o todos leen la misma dirección de memoria (*broadcast*), no hay conflicto y el acceso es rápido, pero cuando varios hilos del *half-warp* acceden al mismo banco, el acceso es lento y se produce conflicto.

En la Figura 3.11 (A) se muestran dos tipos de accesos donde no se generan conflictos de bancos, el primero es un acceso lineal, donde cada hilo accede a un único banco y cada banco es accedido por un único hilo de forma ordenada, es decir, el hilo 0 accede al banco 0, y así sucesivamente. En el segundo caso de la Figura 3.11 (A), cada banco también es accedido por un único hilo pero de forma desordenada. La Figura 3.11 (B) muestra dos casos donde se presentan conflictos de bancos, ya que un mismo banco es accedido por más de un hilo al mismo tiempo.

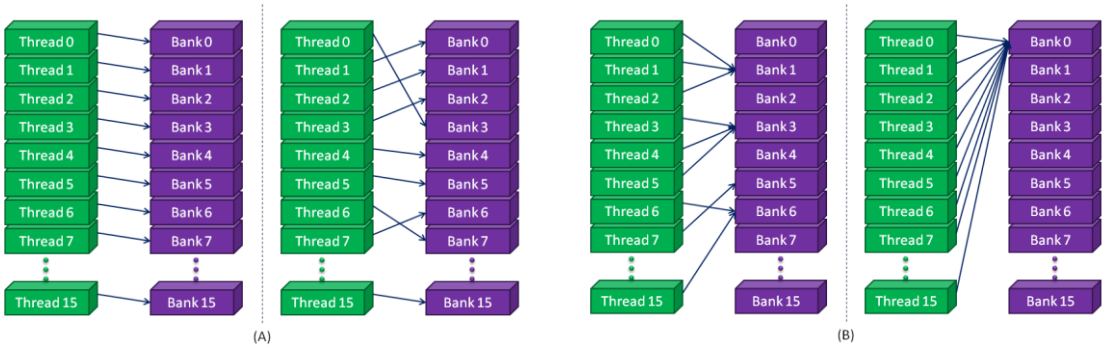


Figura 3.11 – (A) Ejemplo de acceso a memoria sin conflictos de bancos, (B) ejemplo de acceso a memoria con conflictos de bancos. Extraído de [51].

3.5 Consideraciones de performance

Aunque un *kernel* puede ejecutar en cualquier *device* de CUDA, su tiempo de ejecución depende en gran medida, de las limitaciones de cada *device.* A continuación, se muestran las limitaciones más comunes en estos dispositivos y los mecanismos para mejorar la *performance* de los algoritmos en CUDA [50].

Para optimizar la *performance* de los algoritmos en CUDA las recomendaciones de mayor importancia son: encontrar la forma de paralelizar código secuencial, minimizar las transferencias entre el *host* y el *device* (se explica en detalle a continuación), ajustar la configuración del *kernel* para maximizar la utilización del *device*, asegurar que el acceso a memoria sea *coalesced* (ver Anexo I), reemplazar los accesos a memoria global por accesos a memoria compartida siempre que sea posible, evitar la divergencia de hilos, es decir diferentes caminos de ejecución dentro de un mismo *warp* (ver Anexo I), entre otros.

Las transferencias entre el *host* y el *device* usando la operación *CudaMemcpy()* bloquean la ejecución de otras transferencias, este bloqueo termina cuando finalizan las transferencias entre el *host* y el *device*. Esto disminuye significativamente la *performance*, ya que no es posible realizar otras transferencias mientras se estén realizando transferencias de este tipo. Para evitar esto, se debe utilizar la función *cudaMemcpyAsync()* que a diferencia de la anterior, devuelve el control inmediatamente al hilo en CPU, y no bloquea la ejecución. Para poder utilizar esta operación se requiere el uso de *pinned memory* [50], esta memoria realiza las transferencias alcanzando el máximo ancho de banda entre el *host* y el *device*.

Por otra parte, los accesos a memoria global, o local sin caché tienen una latencia de 400 a 600 ciclos de reloj. La latencia en memoria global se puede ocultar, si hay suficientes instrucciones aritméticas independientes, que puedan ser computadas mientras se espera a que se complete el acceso a memoria global. Sin embargo, siempre que sea posible es mejor evitar el acceso a memoria global.

3.6 Algoritmos aplicables a GPU

Los algoritmos que más se adecuan a la ejecución en GPU son aquellos cuya ejecución pueda ser paralelizable, para poder explotar al máximo el poder de cómputo de la GPU, como por ejemplo: multiplicación de matrices, trasposición de matrices, suma de prefijos de *arrays* grandes en paralelo, convolución de imágenes, algoritmos de generación de números aleatorios. Algunos de estos ejemplos pueden encontrarse en el kit de desarrollo (SDK) para familiarizar a los programadores con el lenguaje CUDA.

Uso de GPU para algoritmos evolutivos

La programación genética es la metaheurística sobre la que se ha trabajado más ampliamente en GPU, esto se debe fundamentalmente a que la evaluación de *fitness* agrega un nivel de paralelismo adicional,

ya que suele consistir en evaluar una función para varios valores distintos, que no está presente en otras técnicas.

3.7 Relevamiento de proyectos de Algoritmos Evolutivos hechos en GPU

A continuación se presenta el relevamiento de algunos proyectos en el área del uso de GPUs para la aceleración de algoritmos evolutivos.

En 2009, Abascal y Feijoo [70] realizaron una implementación de algoritmos genéticos sobre la plataforma de desarrollo paralelo CUDA. Este proyecto está orientado a la utilización de las GPUs para implementar un conjunto de algoritmos que solucionen el problema del viajante de comercio TSP [15] y la función de Schwefel[64] . Los principales objetivos de este proyecto son estudiar la sobrecarga de comunicación entre CPU y GPU y evaluar distintos operadores genéticos utilizando las ventajas de programación que proporciona CUDA. Para este proyecto se utilizó un procesador Intel Core2 Duo E6600 (2.4 GHz), una memoria RAM de 2 GB y una GPU NVIDIA 8800 GTX. La versión de CUDA utilizada fue la 2.0. Para la resolución de estos problemas se utilizó un algoritmo memético [61], y los operadores EMMRS [33] y *Hill Climbing* [46]. Los autores realizaron la implementación de los algoritmos en CPU y luego la realizaron en GPU, para poder comparar tiempos. Luego de las pruebas realizadas los autores pudieron concluir que para la función de Schwefel los resultados tanto de calidad como de tiempo hacen del EMMRS una herramienta a tener en cuenta en futuras implementaciones de problemas discretos, no tanto así para el problema del viajante de comercio, ya que el tiempo de ejecución en CPU es muy similar al tiempo de ejecución en GPU, esto se debe al cuello de botella en la transferencia de datos entre CPU y GPU, tema que proponen investigar como trabajo a futuro. En lo referente a calidad, destacan que los operadores EMMRS son capaces de encontrar soluciones cercanas al óptimo para la función de Schwefel.

En 2009 A. Munawar y otros [1], implementaron un algoritmo genético híbrido con búsqueda local para resolver el problema de MAX-SAT (*MAXimum SATisfiability*) [45] utilizando CUDA. La motivación principal de este proyecto es mostrar el poder computacional de las GPUs para reducir el tiempo de ejecución en la resolución del problema de MAX-SAT, usando un algoritmo genético híbrido y búsqueda local. En este proyecto se realizaron modificaciones al algoritmo convencional de búsqueda local y a sus operadores genéticos para adaptarlos a la arquitectura SIMT (*Single Instruction Multiple Thread*), además se sugieren varias técnicas de optimización para reducir el tiempo de ejecución. En el algoritmo propuesto se utilizó un algoritmo jerárquico, con subpoblaciones estructuradas en 2 dimensiones, organizadas en islas en un *grid* de 2 dimensiones, donde cada individuo tiene cuatro individuos vecinos (norte, sur, este, oeste) y cada sub-población tiene también cuatro sub-poblaciones vecinas (norte, sur, este, oeste). En CUDA se organizó la población en un bloque de 2 dimensiones y un *grid* de 2 dimensiones. Respecto a la implementación del algoritmo, luego de la inicialización, los datos son transferidos al *device* y el código entra en un *loop.* Dentro del *loop,* la primera tarea es invocar al primer *kernel*, que se encarga de generar los números aleatorios, el siguiente paso es invocar al segundo *kernel*, que se encarga de ejecutar el AG, el *loop* finaliza cuando se cumple con el criterio de parada. Luego de finalizado el *loop,* la población resultado se copia a la memoria del *host*, y se selecciona el individuo con mayor *fitness* como mejor solución. Para realizar las pruebas de *performance* se utilizó una GPU NVIDIA Tesla C1060, y un procesador Intel Core i7 920 con 2.67 GHz. En las pruebas realizadas, se obtuvo una aceleración de 25x en el mejor caso.

En el año 2010 J. Blengio y otros, realizaron un estudio de estrategias híbridas de cómputo CPU-GPU para acelerar algoritmos evolutivos [16]. En este proyecto, se implementaron varias formas de procesar los algoritmos evolutivos y luego se realizaron comparaciones de las distintas versiones, para demostrar que utilizar la GPU de distintas formas, afecta el rendimiento de la velocidad de procesamiento del algoritmo evolutivo. La propuesta presentada se basó en el esquema de paralelismo de GAs Maestro-

Esclavo y los resultados obtenidos se realizaron utilizando una GPU NVIDIA 9800 GTX+. La primera implementación fue hecha únicamente en la CPU, la segunda versión, utiliza la GPU para evaluar la función de *fitness*, la tercer versión es similar a la segunda pero utilizando la memoria compartida de la GPU y la última es una estrategia híbrida en donde la evaluación del *fitness* de una parte de la población se realiza en la GPU y el resto en la CPU, de forma de aprovechar el tiempo que la CPU está ociosa mientras se evalúa la función de *fitness* en la GPU. Con los resultados obtenidos en este estudio se pudo concluir que es más eficiente utilizar la GPU ya que los tiempos de ejecución cuando se usa solamente la CPU son mayores. Si se utiliza la memoria compartida en la GPU, el tiempo de ejecución es menor que si esta no se utiliza. La estrategia híbrida fue la más eficiente en esta prueba. Además se concluye que las versiones de memoria compartida y la versión híbrida son las que obtienen los mejores resultados en cuanto al tiempo de ejecución, pero también son los algoritmos con mayor degradación de la performance al aumentar el tamaño de la población.

En 2010 N. Soca y otros desarrollaron PUGACE [49], un *framework* genérico para la implementación de Algoritmos Evolutivos Celulares. Este *framework* sigue la línea de Mallba [66], está implementado en C y utiliza la versión 2.1 de CUDA para manipular la GPU. PUGACE soporta varios tipos de representación, varios mecanismos de selección, cruzamiento y mutación, permite también, la configuración de varios parámetros, como el tamaño de la población, el número de generaciones, entre otros. Incluye el desarrollo de varios operadores evolutivos y su diseño es extensible, soporta además el mecanismo de búsqueda local para mejorar las soluciones. En PUGACE, la población se almacena en la memoria de la GPU, y es transferida desde la CPU hacia la GPU al inicio del algoritmo y desde la GPU a la CPU al final del algoritmo. Cada individuo utiliza un hilo de ejecución diferente y los hilos están organizados en bloques de diversos tamaños. El *framework* explota los diferentes niveles de memoria de la GPU, en particular la información del problema es almacenada en memoria constante. Debido a que los operadores de cruzamiento y de mutación son probabilísticos, algunos hilos los aplicarían y otros no, lo que causaba divergencia de hilos, para evitar esto, los autores decidieron aplicar los operadores a nivel de bloque. Se realiza una única decisión por operador en cada bloque, lo que significa que dentro de un mismo bloque se utilizan los mismos puntos de cruzamiento. Por otro lado, los puntos de mutación y los nuevos valores de mutación se seleccionan de manera independiente para cada hilo. En el diseño de PUGACE ciertos aspectos de GPU, como maximizar el uso de memoria compartida y el acceso *coalesced* a memoria global, no fueron tenidos en cuenta. Las pruebas realizadas se ejecutaron en un procesador Intel Pentium Dual E2220 con 2GB RAM y una tarjeta gráfica GeForce 9600 GT con 512 MB de memoria RAM. La aceleración alcanzada fue del orden de 5.65x y 10.65x con tamaños de población entre 4096 y 16384, para el problema de asignación cuadrática.

En 2011, M. Pedemonte y otros presentaron el artículo *Bitwise Operations for GPU Implementation of Genetic Algorithms* [41]. Los autores analizaron las variaciones de performance de un algoritmo genético implementado en GPU, utilizando el tipo de datos *bool* y empaquetando múltiples bits en una estructura de datos no booleana que almacena tiras de bits. Los operadores de cruzamiento utilizados fueron, cruzamiento en un punto y cruzamiento en dos puntos para la resolución del problema One-Max. One-Max, es un problema muy utilizado en resultados experimentales, ya que es muy simple y está enfocado en las características del algoritmo y no en las características del problema. Los autores realizaron una implementación en CPU y otra en GPU del algoritmo para poder comparar los resultados obtenidos y verificar la correctitud de los mismos. Los números aleatorios utilizados para el desarrollo, fueron generados en CPU y luego transferidos a GPU, para tener así, exactamente los mismos números aleatorios tanto en CPU como en GPU y facilitar la comparación. Para la generación de números aleatorios se utilizó la biblioteca *Mersenne twister* [40]. En el algoritmo propuesto, la ejecución comienza con la inicialización de la población que es ejecutada en la CPU, luego se transfiere a memoria global de GPU. Cuando el algoritmo alcanza la condición de parada, la población final es transferida de GPU a CPU. La generación de números aleatorios es ejecutada también en CPU y transferida a memoria

global de GPU en cada iteración. La evaluación del *fitness*, el torneo, el cruzamiento y la mutación se ejecutan completamente en GPU. Los resultados obtenidos muestran que el empaquetado de bits puede reducir hasta un 50% el tiempo de ejecución. Las pruebas realizadas se ejecutaron en un procesador Intel Xeon Quad Core con 2.40 GHz y 48 GB de memoria RAM, y una GPU Tesla C1060 con 240 CUDA *cores*, utilizando el sistema operativo CentOS Linux 5.4.

En 2011, I. Castro realizó su tesis de maestría acerca de paralelización de algoritmos de optimización basados en poblaciones por medio de GPGPU [24]. El objetivo principal es obtener como producto el desarrollo de bibliotecas modificables, en una plataforma basada en GPGPU, para algoritmos de optimización basados en poblaciones, específicamente algoritmos genéticos y optimización por enjambre de partículas. El algoritmo se implementó en C++ con las librerías de CUDA C 3.2, en un procesador Intel Core *i5* (3.20GHz), 4 GB de RAM, con una GPU NVIDIA *GeForce GTX 460* con 1GB de memoria global. La estrategia de paralelización utilizada en este problema fue la siguiente: se definieron *B* enjambres de *t* partículas para ser procesados por un bloque, a cada hilo dentro del bloque se le asignó la tarea de actualizar los parámetros de una partícula. Se determinó un período migratorio del 1% del total de generaciones y la migración se implementó recibiendo la mejor partícula del siguiente bloque, en forma circular. Se realizaron pruebas en CUDA y en un *cluster* de 16 computadoras para comparar el tiempo de ejecución, y se obtuvieron *speedups* de hasta 1.96x, siendo más eficiente en cuanto al tiempo de ejecución la implementación en CUDA.

En 2011, I. Contreras y otros presentaron su trabajo para acelerar un AG basado en el comercio de la bolsa de valores [25]. Para ello se deben tomar en cuenta distintos indicadores que hacen al negocio de comercio en la bolsa de valores y utilizaron un AG para seleccionar el umbral entre los valores de los distintos indicadores. Los autores basaron su implementación en GPU sobre Jacket implementado por Accelereyes [65], que permite el uso de GPUs a usuarios que no tienen que ser expertos en arquitectura de computadoras, pero sí deben tener conocimientos de Matlab. Jacket es un software que permite acelerar códigos basados en Matlab utilizando la GPU, permite la manipulación de matrices de manera sencilla, y provee además, un conjunto de versiones en GPU de la mayoría de las funciones de Matlab. Las pruebas realizadas fueron ejecutadas tanto sobre CPU como GPU. Para CPU, se utilizaron tres equipos con diferentes configuraciones, Pentium 4, Pentium SU 4100 y *core* i7-860. La GPU utilizada fue una NVIDIA GTX 460 OC, con 336 CUDA *cores*, 768 MB de memoria, conectado a la i7. Inicialmente con poblaciones de 100 individuos se obtuvieron mejores resultados sobre la CPU en comparación con la GPU, al aumentar la cantidad de individuos de cada población, se redujo la diferencia de performance de la CPU respecto a la GPU, y finalmente, al utilizar poblaciones con más de 200 individuos se obtuvieron mejores tiempos en GPU respecto a la CPU.

En 2011 N. Fujimoto y S. Tsutsui presentaron su trabajo para resolver el conocido problema del Viajante de Comercio TSP [15] utilizando una GPU para acelerar los cálculos [48]. Las pruebas se realizaron sobre una tarjeta NVIDIA GeForce GTX 285 y una CPU Intel Core 2 Duo E6850 3.0 GHz. Se utilizó un modelo de islas de aproximadamente 120 islas con 512 individuos cada una, donde el algoritmo propuesto alcanzó una aceleración de 24.2x ejecutándose en la GPU respecto a su ejecución en CPU, maximizándose cuando el número de islas es de 200 a 226 ya que el número de hilos se maximiza para esa cantidad de islas.

En 2011, S. Potti y S. Pothiraj presentaron su trabajo [63] para optimizar un algoritmo memético paralelo que resuelva el problema VLSI *Floorplanning* [68]. Este trabajo propone ejecutar un algoritmo memético paralelo en GPU donde tanto la función de *fitness* como los operadores aplicados se ejecutan en la GPU. Las pruebas se realizaron sobre un equipo Pentium Core 2 Quad 2.6 GHz con una tarjeta NVIDIA Tesla C1060 con 512MB de memoria y 8GB de memoria RAM. Se hicieron pruebas con

poblaciones de 40, 80 y 100 individuos tanto sobre CPU como GPU y se obtuvieron aceleraciones de 2.5, 4 y 6.25 respectivamente.

En 2011, O. Maitre y otros presentaron su trabajo para especificar y ejecutar diversos tipos de AG sobre la plataforma EASEA (*Easy Specification of Evolutionary Algorithm*) [54]. EASEA es una plataforma de *software*, que permite especificar y ejecutar distintos tipos de AG sobre GPU, como algoritmos meméticos, entre otros, utilizando la GPU para realizar la evaluación de la función de *fitness*. La población se distribuye en bloques, para ser asignada a múltiples procesadores en la tarjeta, de manera de mantener un buen equilibrio de carga. Idealmente el número de bloques debe ser mayor o igual al número de multiprocesadores y el mínimo número de hilos por bloque es igual al tamaño de warp. Para demostrar la capacidad de la plataforma se realizaron experimentos con la función de Weierstrass–Mandelbrot [36], y la función de Rosenbrock [2]. Las pruebas se realizaron en una máquina Pentium 4 de 3.6 GHz con una tarjeta GTX8800. Se realizaron distintos experimentos para evaluar EASEA con diferentes tamaños de población y se determinó que con poblaciones mayores a 3000 individuos era donde la GPU se encontraba realmente cargada. El uso de GPGPU en EASEA alcanzó aceleraciones muy interesantes con un mínimo esfuerzo. Con esto se obtiene una gran ganancia cuando el problema es el tiempo de búsqueda. Además como la aceleración alcanzada es del orden de cientos de veces, permite la exploración en el espacio de soluciones que son inaccesibles a los Algoritmos Evolutivos. EASEA puede paralelizar sobre varias tarjetas gráficas dentro de una misma máquina e implementa el modelo de islas que puede lidiar con máquinas heterogéneas que pueden o no contar con GPUs.

En 2012 J. Jaros presentó su trabajo para resolver el problema de la Mochila [22], utilizando AG basados en islas sobre múltiples GPUs [35]. Los algoritmos propuestos en el trabajo fueron probados utilizando dos servidores TYAN, ambos servidores cuentan con dos procesadores de 6 núcleos Intel Xeon X5650 2.6 GHz y 24 GB de memoria RAM y 7 tarjetas NVIDIA GTX 580. Los bloques de hilos se organizan en dos dimensiones, la dimensión *x* corresponde a los genes dentro de los cromosomas mientras que la dimensión *y* corresponde a distintos cromosomas. La dimensión *x* de cada bloque es de 32 y la dimensión *y* es de 8, por tanto, se tienen 256 hilos por bloque. El *grid* tiene una dimensión y corresponde al tamaño de la población descendiente dividida por el doble del tamaño del bloque (se producen dos descendientes a la vez). Cada *warp* es responsable de generar 2 individuos y la selección es realizada por un solo hilo dentro del *warp.* La evaluación de la función de *fitness* sigue la misma descomposición de bloque y *grid* mencionada anteriormente. Las pruebas realizadas se basaron en el problema de la mochila con un peso máximo de 10.000, la cantidad de individuos varió entre 128 y 2048 por isla. Se probaron distintas configuraciones de GPUs en islas de 6 y 12 GPUs para realizar una analogía con una CPU de 6 núcleos y dos CPUs de 6 núcleos en un servidor respectivamente. La configuración máxima se da con 7 GPUs o 14 GPUs en uno o dos servidores respectivamente. La aceleración que se da con 14 GPUs llega a 35, 194, y 781 en comparación con 4 CPU, 1 CPU y un sólo hilo, respectivamente.

4 DISEÑO E IMPLEMENTACIÓN

Resumen: En este capítulo se presentan las decisiones de diseño tomadas, así como también la implementación realizada en este proyecto. El capítulo está estructurado de la siguiente manera. En la primera sección se presentan las decisiones de diseño. En la segunda sección, se presenta la estructura del framework. En la tercera sección se describen las formas de ejecución del framework. Luego, en la cuarta sección se presenta el mapeo del algoritmo en la GPU. En la quitan sección, se describe en detalle la implementación del framework. Y finalmente, en la última sección, se describe el archivo de configuración del framework.

4.1 Decisiones de diseño

Del relevamiento de los trabajos realizados descritos en la Sección 3.6 se concluye que el modelo de paralelismo de islas es uno de los más populares sobre arquitecturas paralelas distribuidas tradicionales, aunque no existen muchas implementaciones de este modelo en GPU. En base a esto, se tomó la decisión que el *framework*, al que llamamos *ag-cuda-framework*, utilice el modelo de paralelismo de islas, con una implementación híbrida, en la que es posible que haya islas en la GPU y en la CPU al mismo tiempo. Este tipo de implementación permite explotar tanto la capacidad de cómputo de la tarjeta gráfica, como la de la CPU en forma concurrente.

En el *ag-cuda-framework* se implementaron dos formas de ejecución para la GPU. La primera opción consiste en que cada bloque procesa una población y cada hilo dos individuos, y la llamaremos un bloque por población; mientras que en la segunda forma de ejecución cada bloque procesa dos individuos y el usuario ingresa la cantidad de hilos a ejecutarse en cada bloque, y la llamaremos un bloque cada dos individuos. Esto condiciona la forma de configurar la cantidad de hilos y bloques, con la primera opción se configuran *#Poblaciones* bloques *con #individuosXpoblacion/2* hilos por bloque, y con la segunda opción se configuran *#individuos/2* bloques con *#hilosPorBloque* hilos cada uno. En la Sección 4.3 se detallan los tipos de configuración mencionados anteriormente.

Dos grandes aspectos que condicionan el diseño y la implementación del *ag-cuda-framework* son: el tipo de representación usada por el algoritmo genético y la configuración de bloques e hilos para ejecutar en la GPU. Para poder trabajar sobre los distintos escenarios se optó por dividirlos en cada posible pareja (*tipo de representación – configuración de bloques e hilos*), ya que cada representación condiciona a la aplicación a trabajar sobre distintos tipos de datos (por ejemplo, en representación binaria se utiliza el tipo de datos *bool* para representar a los genes, mientras que en representación entera se utiliza el tipo de datos *int*) y por tanto pueden utilizar distintos operadores, la configuración de bloques e hilos también se debe independizar, ya que su comportamiento difiere según la configuración seleccionada.

En el *ag-cuda-framework* los tipos de representaciones implementados son: Representación Binaria, Representación Entera, Representación de Permutación y Representación Real. Los operadores evolutivos implementados para cada una de las representaciones se detallan en la Sección 4.3.

4.2 Estructura del framework

El propósito del *ag-cuda-framework,* es proporcionar un diseño genérico y extensible para permitir agregar de manera sencilla nuevos tipos de representación u operadores para cada tipo de representación, y cubrir los tipos de representaciones mencionados en la Sección 4.1, y sus distintos operadores evolutivos. Para ello se realizó un diseño de la herramienta como el que se muestra en la Figura 4.1.

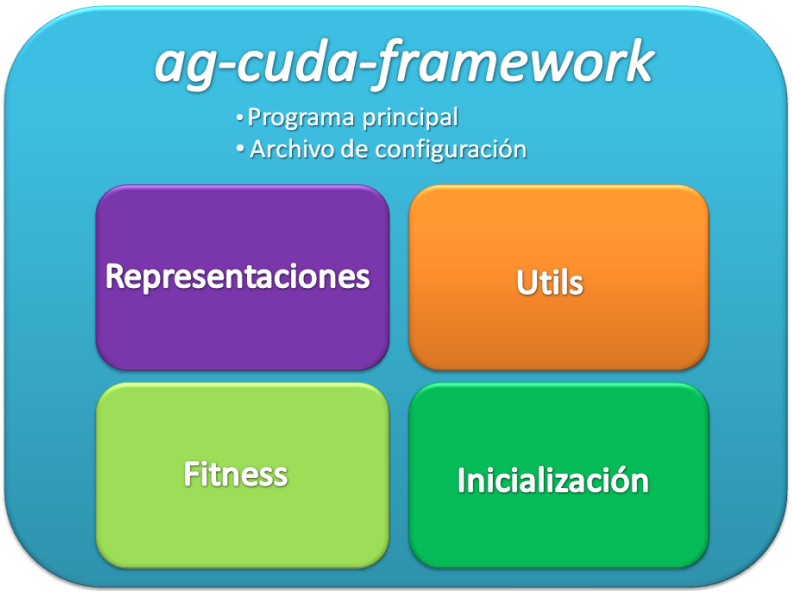


Figura 4.1 – Esquema de Diseño del ag-cuda-framework.

Como se puede ver en la Figura 4.1, el *ag-cuda-framework* está estructurado en cuatro grandes módulos: Representaciones, Fitness, Utils e Inicialización. En el mismo nivel que los módulos principales, se encuentra el programa principal, que es el encargado de invocar a las funciones adecuadas dependiendo del tipo de representación y configuración seleccionadas, y se encuentra también el archivo de configuración, donde se especifican los parámetros de ejecución del *ag-cuda-framework* (ver detalle en el Anexo III).

Figura 4.2 –Módulo Representaciones.

En el módulo Representaciones, se encuentran los archivos que implementan los diferentes tipos de representación en sus posibles configuraciones. En la Figura 4.2, se muestran los archivos de este módulo. En cada archivo se implementa el algoritmo evolutivo con los diferentes operadores evolutivos correspondientes al tipo de representación, la forma de implementación de cada una se detalla en la Sección 4.4. Para la representación binaria, se implementaron tres módulos, uno correspondiente a la configuración de un bloque cada dos individuos, *binariaGen.cu*, otro correspondiente a la configuración de un bloque por población, *binaria.cu* y otro correspondiente a la implementación en CPU, *binariaCPU.cu*. De manera análoga se implementaron las representaciones entera y real. Sin embargo, para la representación de permutación, no se realizó la implementación de la configuración de un

bloque cada dos individuos, debido a que en los operadores evolutivos utilizados no es posible considerar cada gen independientemente, ya que éstos requieren de la información de los otros genes.

El módulo Fitness, como se muestra en la Figura 4.3, contiene las funciones de *fitness* correspondientes a cada representación, que serán definidas por el usuario final. Para lograr un diseño lo más desacoplado posible, se implementó un módulo independiente para cada representación. De este modo las funciones de *fitness* correspondientes a la implementación en GPU y en CPU en representación binaria deben ser implementadas en *fitnessBinaria.cu*, análogamente los archivos *fitnessEntera.cu*, *fitnessPermutacion.cu y fitnessReal.cu* permiten definir funciones de *fitness* para representación entera, permutación y real respectivamente.

Figura 4.3 –Módulo Fitness.

En el módulo Utils, que se muestra en la Figura 4.4, se encuentran los archivos auxiliares que se utilizan en el *ag-cuda-framework*. Los módulos son: *parser.cpp* (se encarga de analizar el archivo de configuración y obtener los parámetros de entrada del mismo, como por ejemplo, el tipo de representación a utilizar, el operador de cruzamiento, el operador de mutación, etc.); *global.cu* (se declaran todas las variables y funciones globales que serán utilizadas desde los demás archivos); *random.cu* (generador de números aleatorios en GPU utilizando la biblioteca CURAND [52]); *mt.cpp* (*Mersenne Twister* [40] utilizada para generar números aleatorios en CPU) y *cuPrintf.cu* [52] (utilizada para imprimir datos en pantalla desde la GPU).

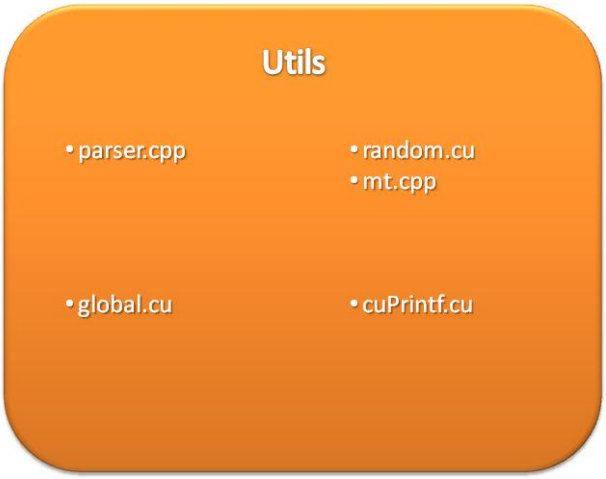


Figura 4.4 – Módulo Utils.

En el módulo Inicialización que se muestra en la Figura 4.5, se encuentran los archivos que se utilizan en el *ag-cuda-framework* para inicializar las poblaciones de forma manual. Se incluye un archivo para cada una de las representaciones, para las distintas configuraciones posibles. En caso de que el usuario final desee implementar su propio mecanismo de inicialización de las poblaciones, debe implementar el método de inicialización en el archivo correspondiente a la configuración que desee ejecutar. Por defecto, el *ag-cuda-framework* proporciona un método de inicialización aleatoria para los individuos de una población.

Figura 4.5 – Módulo Inicialización.

4.3 Formas de ejecución

Hay tres formas de ejecución de los algoritmos sobre el *ag-cuda-framework*, estas son: ejecución en GPU, ejecución en CPU y ejecución híbrida. A continuación, se describe en detalle cada una de ellas. Para más información sobre configuraciones previas a la ejecución del *ag-cuda-framework* consultar Anexo II.

4.3.1 Ejecución en GPU

Es posible configurar el *ag-cuda-framework* para que la ejecución del algoritmo evolutivo se realice únicamente en GPU, es decir, las islas están alojadas en GPU y realizan el procesamiento de los operadores evolutivos exclusivamente en GPU. Para esto, se puede seleccionar una de las dos formas de configurar la GPU que se mencionaron en la Sección 4.1 (un bloque por población, o un bloque cada dos individuos). La segunda opción de ejecución resulta en general más razonable, ya que permite alcanzar un mejor nivel de uso de la GPU y por ende un alto desempeño computacional.

4.3.2 Ejecución en CPU

El propósito de la implementación del *ag-cuda-framework* en CPU es permitir la ejecución híbrida de los algoritmos para incrementar al máximo la performance obtenida. Además, se puede configurar el *ag-cuda-framework* para que el algoritmo evolutivo ejecute únicamente en CPU. Esta opción de ejecución en general, es la más ineficiente, ya que no utiliza la capacidad de cómputo de la GPU, además, la implementación realizada no utiliza *multithreading* [44].

4.3.3 Ejecución híbrida

Para aprovechar al máximo los recursos utilizados por el *ag-cuda-framework* es posible realizar una ejecución híbrida de los algoritmos. De esta forma, mientras la GPU procesa las islas que están alojadas en la GPU, la CPU no está ociosa y procesa una parte de las islas. Cabe destacar que la CPU debe

procesar una parte de menor tamaño de las islas, ya que la ejecución en CPU es considerablemente más lenta. Los resultados obtenidos de las ejecuciones híbridas se muestran en la Sección 5.3.

4.4 Configuración de la GPU

Como se mencionó anteriormente, se realizaron dos implementaciones diferentes respecto al uso de la tarjeta gráfica. La primera no es la más óptima, ya que se lanza un bloque por población, y cada hilo que ejecuta en la GPU se encarga de procesar dos individuos de la población, lo que implica que el proceso del hilo sea "pesado" y por tanto, va en contra de las recomendaciones de implementar *kernels* lo más liviano posibles para ejecutar en la tarjeta, además complica los accesos *coalesced* a memoria. Aun así, se decidió que esta implementación sea parte del proyecto final ya que en ciertos contextos, por ejemplo: con poblaciones con miles de individuos como las estudiadas por O. Maitre en EASEA [54], ésta puede ser una mejor estrategia. Además, fue un primer paso hacia a una implementación que está más en concordancia con la idea de que cada hilo sea lo más liviano posible. En la segunda implementación, se lanza un bloque cada dos individuos, y cada hilo se ocupa de procesar *#genes/#hilosPorBloque*2* genes y cada bloque procesa dos individuos.

4.4.1 Un bloque por población

En la primera opción, para procesar cada población, es necesario lanzar *#Poblaciones* bloques, con *#Individuos/2 hilos* por bloque. La Figura 4.6 muestra el mapeo de esta configuración en la GPU, siendo *M = #Poblaciones* y *N = #Individuos.* Los bloques utilizados son unidimensionales.

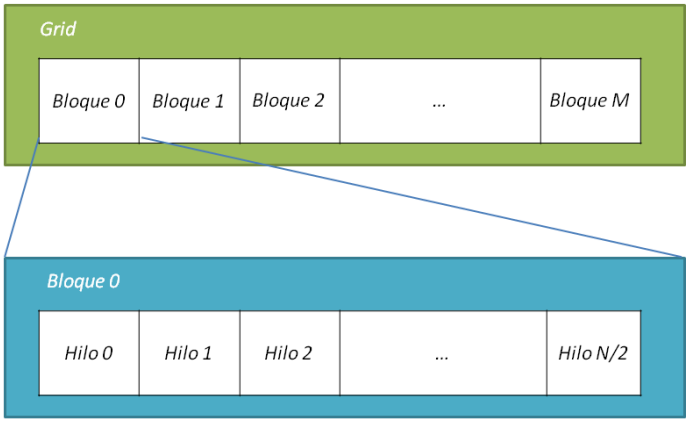


Figura 4.6 –Mapeo en GPU de un bloque por población.

A modo de ejemplo, la Tabla 4.1 muestra las funciones globales del *ag-cuda-framework* para la representación binaria y un bloque por población. Las funciones son análogas para representación entera, de permutación y real.

FUNCION	#BLOQUES	#HILOS
inicializacion	#PoblacionesGPU	#CromosomasPoblacion/2
mainKernel	#PoblacionesGPU	#CromosomasPoblacion/2
obtenerMejoresIndividuoBinaria	#PoblacionesGPU	1
migracionRepresentacionBinaria	#PoblacionesGPU	1

Tabla 4.1 – Funciones globales para un bloque por población.

A continuación se detallan las principales funciones para esta configuración.

inicializacion

En el *kernel* de inicialización, cada hilo se encarga de inicializar dos individuos por lo que cada hilo inicializa *#genesCromosoma*2* genes. Para realizar la inicialización de las poblaciones, se lanza el *kernel* de inicialización con *#Poblaciones* bloques y *#CromosomasPoblacion/2* hilos por bloque.

mainKernel

Dentro de esta función, cada hilo obtiene los dos individuos a cruzar y mutar invocando a la función de selección correspondiente al mecanismo configurado. Luego, se invoca a la función correspondiente al tipo de cruzamiento seleccionado, y a la función correspondiente al tipo de mutación seleccionado. A continuación, se calcula el *fitness* para los dos individuos procesados y realiza el mecanismo de reemplazo seleccionado (generacional o elitista). Finalmente, se sincronizan los hilos y el hilo 0 se encarga, si corresponde, de calcular el individuo a migrar de la población que se procesa en ese bloque, y de obtener el mejor individuo de la población. Esta función se llama con *#Poblaciones* bloques y *#CromosomasPoblacion/2* hilos, ya que cada bloque procesa una población y cada hilo dos individuos.

obtenerMejoresIndividuosBinaria

Esta función, es la encargada de obtener el mejor individuo de cada población y almacenarlo en una estructura auxiliar. Cabe destacar que no tiene buena performance invocar a la GPU con un único hilo por bloque, pero siguiendo los lineamientos de esta opción de configurar a la GPU, se optó por esta forma de resolver el problema de ordenamiento. Una mejor alternativa, podría ser utilizar bibliotecas de ordenamiento, pero fue descartada por temas de alcance del proyecto. A esta función se la llama con un bloque por población y un hilo por bloque.

migracionRepresentacionBinaria

A esta función se la llama con un bloque por población y un hilo por bloque, es la encargada de realizar la migración de los individuos. En la Sección 4.4.4, se describe en detalle la implementación de la migración. Para esta función se utilizó también un único hilo por bloque, y el fundamento es el mismo que el explicado en la función obtenerMejoresIndividuosBinaria**.**

4.4.2 Un bloque cada dos individuos

En la segunda opción de configuración, para procesar cada población es necesario lanzar *(#IndividuosPoblacion/2)* bloques, lo que da un total de *(#Individuos/2)*#Poblaciones* bloques, con *#hilosPorBloque* hilos cada uno. La Figura 4.7, muestra el mapeo de esta configuración en la GPU, siendo *M = #Poblaciones, N = #Individuos* y *#hilosPorBloque* el número de hilos por bloque configurado por el usuario. Los bloques utilizados son unidimensionales.

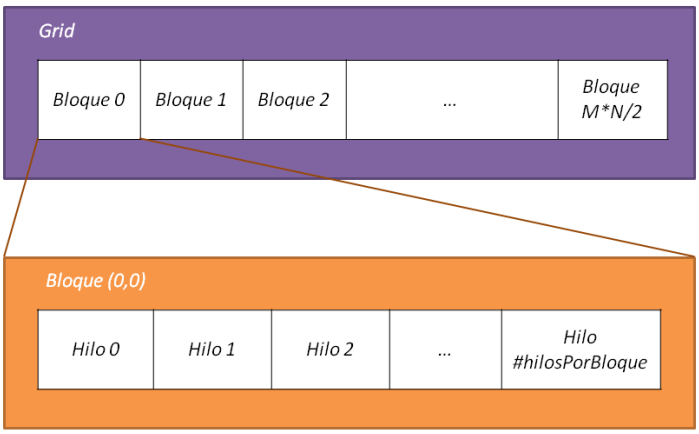


Figura 4.7 –Mapeo en GPU de un bloque cada dos individuos.

A continuación, se detallan las funciones principales del *ag-cuda-framework* para la implementación de un bloque cada dos individuos. En la Tabla 4.2, se muestran las funciones globales del *ag-cuda-framework* para la representación binaria y un bloque cada dos individuos. Las funciones son análogas para representación entera y real.

FUNCION	#BLOQUES	#HILOS
inicializacion	#PoblacionesGPU*#CromosomasPoblacion/2	#HilosBloque
mainKernel	#PoblacionesGPU*#CromosomasPoblacion/2	#HilosBloque
asignarProbabilidadRuleta	#PoblacionesGPU*#CromosomasPoblacion/2	#HilosBloque
obtenerMejoresIndividuos	#PoblacionesGPU	64
individuosAMigrar	#PoblacionesGPU	#CromosomasPoblacion/2
migracion	#PoblacionesGPU	#HilosBloque
calcularFitness	#PoblacionesGPU*#CromosomasPoblacion	64
reemplazoElitistaSeleccionIndivid	#PoblacionesGPU	1
reemplazo	#PoblacionesGPU*#Aremplazar	#HilosBloque

Tabla 4.2 – Funciones globales para un bloque cada dos individuos.

inicializacion

Para realizar la inicialización de las poblaciones, se lanza el *kernel* de inicialización con *#PoblacionesGPU*#CromosomasPoblacion/2* bloques y *#hilosPorBloque* hilos por bloque. En el *kernel* de inicialización, cada hilo se encarga de inicializar *#genesCromosoma/#hilosPorBloque* genes por cada individuo es decir que cada hilo se encarga de inicializar *#genesCromosoma/#hilosPorBloque*2* genes.

mainKernel

En esta función se obtienen los dos individuos a cruzar y mutar invocando a la función de selección correspondiente al mecanismo de selección configurado, luego se invoca a la función correspondiente al tipo de cruzamiento seleccionado y finalmente a la función correspondiente al tipo de mutación seleccionado. Esta función se llama con *#PoblacionesGPU*#CromosomasPoblacion/2* bloques y *#hilosPorBloque* hilos, ya que cada bloque procesa dos individuos y debe respetar la cantidad de hilos que se definió en el archivo de configuración.

asignarProbabilidadRuleta

En esta función, el hilo *0* del primer bloque de cada población se encarga de calcular la probabilidad de cada individuo y almacenarla en una estructura auxiliar. Esta función es invocada en el caso de que se utilice el mecanismo de selección Rueda de Ruleta, y se invoca con *#PoblacionesGPU*#CromosomasPoblacion/2* bloques y *#hilosPorBloque* hilos.

obtenerMejoresIndividuos

Esta función es invocada con *#Poblaciones* bloques y 64 hilos por bloque. Es la encargada de determinar el mejor individuo de cada población, y almacenarlo en una estructura que guarda el mejor individuo de todas las poblaciones, además se almacena en otra estructura los valores de *fitness* correspondientes a los mejores individuos.

individuosAMigrar

Esta función es la encargada de determinar el individuo a migrar de cada población, esto es, el individuo de mejor *fitness* de cada población. Si se configuraron poblaciones en CPU y el bloque es el

correspondiente a la última población de GPU, entonces el individuo se almacena en una estructura que guarda el individuo a migrar de GPU a CPU, sino se almacena en una estructura que guarda todos los individuos a migrar en GPU. Es invocada con *#Poblaciones* bloques y *#CromosomasPoblacion/2* hilos por bloque.

migracion

En esta función se realiza la migración, esto es, se obtiene el peor individuo de la población, y se lo reemplaza por el individuo migrado desde otra población. En la sección 4.4 se explica en detalle el mecanismo de migración. Esta función es invocada con *#Poblaciones* bloques y *#hilosPorBloque* hilos por bloque.

calcularFitness

En representación binaria, cada hilo se encarga de calcular el *fitness* de *#genesCromosoma/#hilosPorBloque* genes, para esto, cada hilo realiza la suma parcial de *#genesCromosoma/#hilosPorBloque* genes, y a este resultado parcial lo suma al *fitness* total en memoria global utilizando la función atomicAdd(). Cabe destacar, que la función es análoga para representación real, con diferencias en el cálculo del *fitness* parcial, ya que el *fitness* se calcula en base a otras operaciones (ver función de *fitness* implementada para representación real en Sección 5.3.3), pero para representación entera, la implementación de la función de *fitness* es más compleja, ya que depende de otros factores (ver función de *fitness* implementada para representación entera en la Sección 5.3.2), por lo cual, para realizar esta implementación se utilizó memoria *shared*, de tamaño *#HilosFitness*#Maquinas* (siendo *#HilosFitness* un valor definido en 64, y *#Maquinas* la cantidad de máquinas configuradas para el problema resuelto, HCSP [23]). A esta función se la invoca con *#PoblacionesGPU*#CromosomasPoblacion* bloques y *#HilosFitness* hilos. Cada hilo se encarga de almacenar el valor correspondiente de la máquina que está procesando en su espacio de memoria *shared*, luego se realiza una reducción, dejando en las primeras *#Maquinas* posiciones del arreglo de memoria *shared* la suma del costo de cada máquina. Finalmente, se realiza un syncthreads() y el hilo 0 se encarga de obtener el valor máximo de dicho arreglo, que será el *fitness* resultante.

reemplazoElitistaSeleccionIndividuos

Esta función se llama con *#Poblaciones* bloques y *un* hilo por bloque. Se invoca cuando el mecanismo de reemplazo seleccionado es elitista, y se encarga de determinar los individuos que serán reemplazados de la nueva generación, y los que reemplazarán a estos últimos. Para esto se utilizan dos estructuras auxiliares, una para almacenar las posiciones de los individuos a reemplazar de la nueva generación, y otra para almacenar las posiciones de los individuos que deben pasar de la generación anterior a la nueva generación. Se invoca al *kernel* con un único hilo debido a que es necesario obtener los *N* mejores individuos de cada población, y para esto es necesario comparar cada individuo con el resto. Debido a esto no es posible realizar una reducción en la que se obtenga como resultado el mejor individuo. Se podría utilizar alguna técnica de ordenamiento que optimice el uso de la tarjeta gráfica, pero esto no fue considerado por motivos de alcance del proyecto.

reemplazo

Esta función se invoca con *cantidadPoblacionesGPU * cantAremplazar* bloques y *#hilosPorBloque* hilos por bloque, esta función se encarga de realizar el mecanismo de reemplazo seleccionado, en este caso, reemplazo generacional o elitista. Si el reemplazo es generacional, no se ejecuta ninguna función, ya que los operadores ya están aplicados sobre la nueva generación. Pero si el reemplazo es el elitista, se invoca a una función, que utiliza las estructuras determinadas en la función reemplazoElitistaSeleccionIndividuos para realizar el reemplazo elitista.

4.5 Implementación

En esta sección se describe la implementación del *ag-cuda-framework*, detallando los tipos de representación y operadores implementados, las estructuras de datos y la migración de individuos.

4.5.1 Tipos de representación y operadores implementados

A continuación se describen los tipos de representación y los operadores evolutivos implementados en base al relevamiento realizado en el Capítulo 2.

Tipos de representación

Los tipos de representaciones implementados en el *ag-cuda-framework* fueron: Representación Binaria, Representación Entera, Representación de Permutación y Representación Real. Para cada una de ellas se implementaron los siguientes mecanismos de selección: Selección por Rango, Selección por Torneo y Selección por Rueda de Ruleta. Para todas ellas, se implementaron también, los mecanismos de Reemplazo Generacional y Reemplazo Elitista. En la Tabla 4.3 se muestran los operadores evolutivos implementados para cada una de las representaciones mencionadas.

	OPERADORES DE CRUZAMIENTO	OPERADORES DE MUTACION
REPRESENTACION BINARIA	✓ (SPX) Cruzamiento en un punto ✓ (2PX) Cruzamiento en dos puntos ✓ (UX) Cruzamiento Uniforme	✓ Mutación por inversión
REPRESENTACION ENTERA	✓ (SPX) Cruzamiento en un punto ✓ (2PX) Cruzamiento en dos puntos ✓ (UX) Cruzamiento Uniforme	✓ Mutación por reajuste aleatorio ✓ Mutación por deslizamiento
REPRESENTACION DE PERMUTACION	✓ (PMX) Cruzamiento basado en correspondencia parcial ✓ (OX) Cruzamiento basado en el orden ✓ (CX) Cruzamiento basado ciclos	✓ (EM) Mutación por intercambio ✓ (ISM) Mutación por inserción simple ✓ (IVM) Mutación por inversión
REPRESENTACION REAL	✓ Cruzamiento aritmético individual ✓ Cruzamiento aritmético simple ✓ Cruzamiento aritmético completo	✓ Mutación uniforme ✓ Mutación no uniforme

Tabla 4.3 – Operadores evolutivos implementados para cada representación.

4.5.2 Estructuras de datos

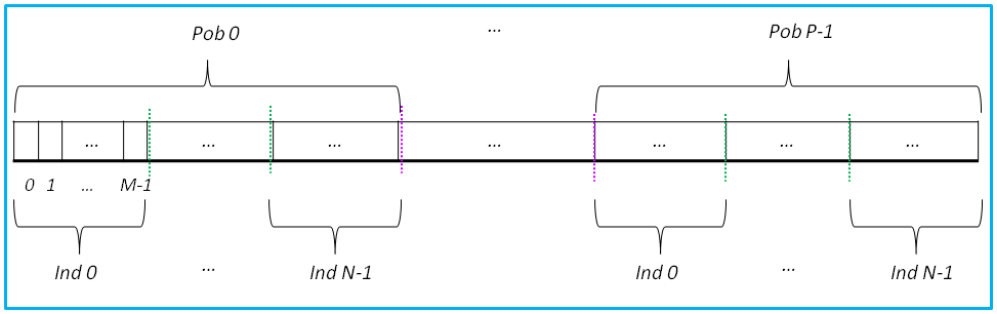


Figura 4.7 – Esquema de representación de la población.

Las estructuras de datos utilizadas son análogas para la configuración en GPU y en CPU. Si se consideran *P* poblaciones de *N* individuos con *M* genes cada uno, la estructura de datos utilizada para almacenar los individuos es un arreglo de tamaño *P*N*M,* como se puede ver en la Figura 4.7. Al utilizar

representación binaria, el tipo de datos de este arreglo es *bool*, en la representación entera y de permutación es *int* y en la representación real es *float.*

Para representación Binaria, Entera y Real, cada población se almacena en memoria de la siguiente manera: se divide el arreglo en *#Cromosomas* intervalos y cada intervalo almacena un cromosoma de forma ordenada. Para representación de Permutación, además de utilizar esta forma de almacenar la población, se desarrolló una nueva metodología de almacenamiento debido a su alto tiempo de procesamiento en GPU. Esta metodología permite el acceso *coalesced* a memoria global. Se divide el arreglo en *#Genes* intervalos, cada intervalo contiene *#Cromosomas* posiciones, por lo cual en el primer intervalo se almacena el primer gen de cada cromosoma, en el segundo intervalo el segundo gen de cada cromosoma y así hasta llegar al último gen de cada cromosoma. Los resultados obtenidos se pueden ver en la Sección 5.3. En la Figura 4.8 se muestra este esquema de representación.

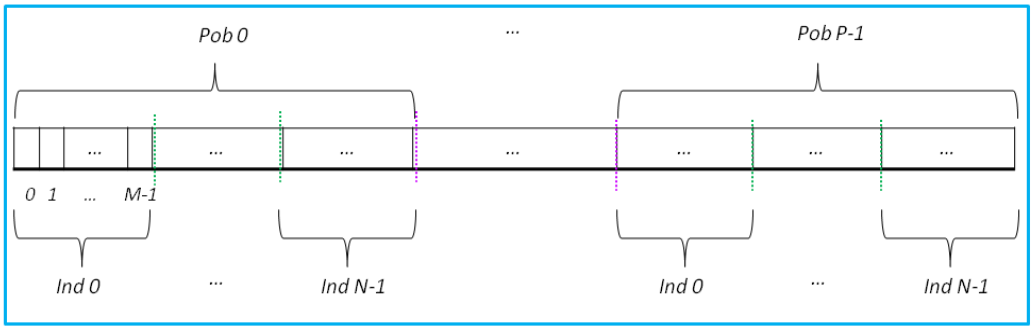


Figura 4.8 – Esquema de representación de la población para Permutación.

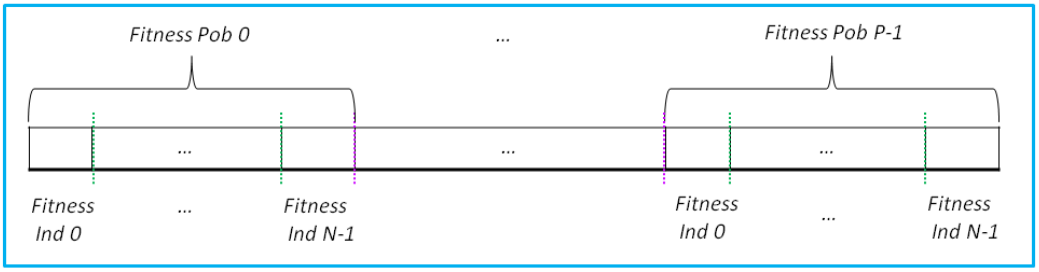


Figura 4.9 – Esquema de representación del fitness.

Para almacenar los *fitness* de todos los individuos de todas las poblaciones se utilizó una estructura como se muestra en la Figura 4.9. Esta estructura fue utilizada en todas las representaciones y tiene un tamaño de *P*N*, el tipo de datos utilizado es *float* para todas las representaciones.

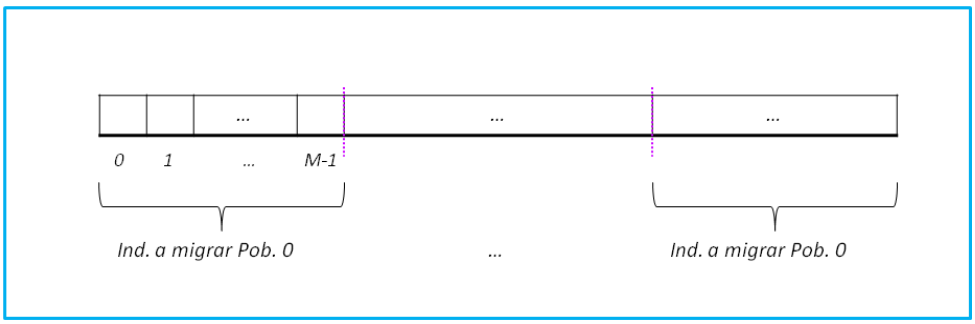


Figura 4.10 – Esquema de representación de individuos a migrar.

Para almacenar los individuos a migrar de una población a otra, la estructura de datos utilizada para todas las representaciones fue un arreglo de tamaño *P*M* como se puede ver en la Figura 4.10.

4.5.3 Migración de Individuos

La Figura 4.11 muestra el esquema de migración en la ejecución en GPU, que es análoga a la ejecución en CPU, siendo *P* el número de poblaciones. La migración se realiza reemplazando el peor individuo de la población *0* por el mejor individuo de la población *k-1*, y así sucesivamente.

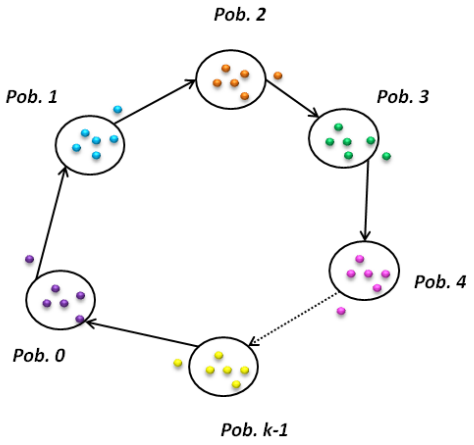


Figura 4.11 – Esquema de migración con poblaciones sólo en GPU o sólo en CPU.

En la Figura 4.12 se muestra el esquema de migración para la ejecución híbrida, suponiendo *P* poblaciones en GPU y *Q* poblaciones en CPU. Dentro de las poblaciones en GPU y en CPU el comportamiento es análogo al descrito anteriormente, pero difiere en las poblaciones extremo de cada una, ya que el peor individuo de la población *0* de GPU se reemplaza por el mejor individuo de la población *j-1* de CPU y el peor individuo de la población *0* de CPU se reemplaza por el mejor individuo de la población *k-1* de GPU.

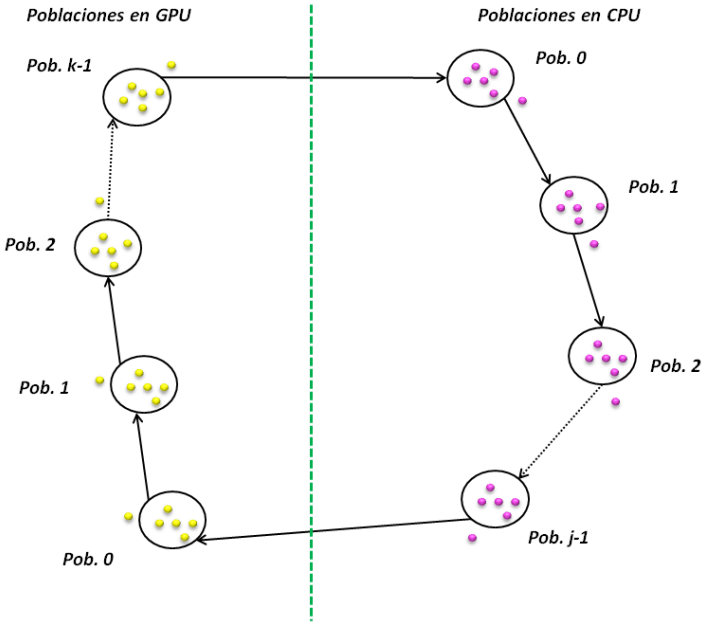


Figura 4.12 – Esquema de migración con poblaciones en GPU y en CPU a la vez.

4.5.4 Descripción de funcionalidades y pseudocódigo

A continuación se describen algunas de las funciones desarrolladas y el pseudocódigo del programa principal del *ag-cuda-framework* para sus dos formas de configuración de la GPU.

Un bloque por población

En la Figura 4.14 se muestra el pseudocódigo del programa principal para la variante que utiliza un bloque por población.

```
Declaración, reserva de memoria e inicialización de variables (GPU y CPU)
Si hay poblaciones en GPU
        Inicialización
        obtenerMejoresIndividuoBinaria
         Si el tipo de selección es por rango
                ordenarDescendente
Si hay poblaciones en CPU
        inicialización CPU
         Si el tipo de selección es por rango
                ordenarDescendente
Mientras no se llegue al número de iteraciones deseadas
        Si hay poblaciones en GPU
                Si la iteración actual es múltiplo de 2
                        mainKernelBinaria
                                Si el mecanismo de selección es por rango.
                                        ordenarDescendente
                Sino, si la iteración actual NO es múltiplo de 2
                                implementación análoga(switch población actual por nueva)
        Si hay poblaciones en CPU
                representacionBinariaCPU
                Si el mecanismo de selección es por rango
                        ordenarDescendenteCPU
        Si hay que migrar
                Si hay poblaciones en CPU y en CPU
                        migracionRepresentacionBinaria
                        migracionRepresentacionBinariaCPU
                Si no, si hay poblaciones en CPU y no hay en GPU
                        migracionRepresentacionBinariaCPU
                Si no, si hay poblaciones en GPU y no en CPU
                        migracionRepresentacionBinaria
                Si hay poblaciones en CPU
Fin Mientras
Se libera memoria de todas las estructuras
```

Figura 4.13 – Pseudocódigo del programa principal (un bloque por población).

A continuación se explica el funcionamiento del *ag-cuda-framework* con un ejemplo de algoritmo evolutivo concreto. Suponiendo representación binaria, inicialización aleatoria, selección por torneo, cruzamiento uniforme, mutación por inversión y reemplazo elitista, con *P* poblaciones de *N* individuos cada una y *M* genes por individuo. En la Sección 4.5.1 se pueden ver los distintos operadores evolutivos para cada tipo de representación.

Kernel principal

Inicialmente, si el tipo de selección es rueda de ruleta, se asigna las probabilidades para realizar la selección, se invoca dos veces a la función de selección para obtener las posiciones de los individuos a procesar. Se llama a la función de cruzamiento, la cual dependiendo del operador de cruzamiento cruza los dos individuos seleccionados y se invoca a la función de mutación que muta los dos individuos resultantes del cruzamiento, luego de la mutación, se realiza el reemplazo elitista, si corresponde. Si hay que migrar entonces se realiza un syncthreads()[1] y se obtienen los individuos a migrar.

Inicialización automática

Existen dos mecanismos de inicialización: manual y automático. En el mecanismo de inicialización automática cada gen de cada individuo de las diferentes poblaciones se inicializa con valores generados aleatoriamente y en el caso de la inicialización manual los valores de los genes, serán definidos por el usuario. Para el caso del tipo de representación binaria y ejecución en GPU, cada hilo que se lanza, se

1 syncthreads() realiza la sincronización de hilos dentro de un mismo bloque.

encarga de inicializar *2*#GenesEnCromosoma* posiciones en el arreglo, y estas posiciones están dadas por *threadIdx.y*#GenesEnCromosoma*2+blockIdx.y*totalGenesEnPoblacion* y *threadIdx.y*#GenesEnCromosoma*2+blockIdx.y*totalGenesEnPoblacion+#GenesEnCromosoma.*

Selección por torneo

Para este mecanismo de selección, cada hilo se encarga de seleccionar aleatoriamente dos valores, con los cuales obtiene la posición de memoria del *fitness* correspondiente, esta posición de memoria está dada por *blockIdx.y*#CromosomasPoblacion+R,* siendo *R* uno de los valores aleatorios, luego compara los *fitness* y retorna la posición del individuo con mejor *fitness*.

Cruzamiento uniforme

En este cruzamiento, cada hilo se encarga de cruzar dos individuos, inicialmente se obtienen las dos posiciones de memoria a escribir en GPU correspondientes a los individuos a cruzar, estas posiciones son análogas a las posiciones obtenidas en la inicialización. Luego, se itera desde cero a *#GenesEnCromosoma* y se obtiene en cada iteración un número aleatorio que indica si se debe o no realizar el cruzamiento. En caso de cruzar, se intercambian los valores de los genes de los individuos a cruzar, y en caso contrario se mantienen.

Mutación por inversión

En la mutación por inversión para la representación binaria, cada hilo se encarga de aplicar el operador de mutación a dos individuos, inicialmente se obtienen las dos posiciones de memoria de los individuos a mutar, estas posiciones son análogas a las posiciones obtenidas en la inicialización. Luego se obtiene un número aleatorio que indica si el primer individuo debe ser mutado, de ser así, se selecciona otro número aleatorio que indica el gen a mutar, y se invierte el valor de dicho gen, en caso contrario se mantiene como estaba. De manera análoga se procede con el segundo individuo a mutar.

Reemplazo elitista

El *ag-cuda-framework* tiene dos mecanismos de reemplazo, generacional y elitista. El generacional se aplica siempre, es decir, siempre se reemplaza la población actual por la nueva población. Si el reemplazo seleccionado es el elitista, también se realiza el generacional con la diferencia de que los mejores *#aReemplazar* individuos de la población actual pasan directamente a la nueva población, siendo *#aReemplazar* la cantidad de individuos que deben sobrevivir.

El reemplazo elitista se realiza de la siguiente manera: se utilizan cuatro estructuras auxiliares, dos de ellas (poblacionActualAcambiar y nuevaPoblacionAcambiar) contienen los individuos a reemplazar de cada población, indicando con uno los individuos a reemplazar y con cero el resto, las otras dos (posIndAreemplazarViejosElitistaGPU y posIndAreemplazarNuevosElitistaGPU) almacenan las posiciones de estos individuos.

Luego de aplicar los operadores evolutivos a la población actual y tener el resultado en la nueva población, se determinan los *#aReemplazar* mejores individuos de la población actual y se almacenan sus posiciones en el arreglo posIndAreemplazarViejosElitistaGPU, también se determinan los *#aReemplazar* peores individuos de la nueva población y se almacenan sus posiciones en el arreglo posIndAreemplazarNuevosElitistaGPU. Para obtener estos individuos se realiza una iteración que se encarga de inicializar los primeros *#aReemplazar* valores de la estructura auxiliar poblacionActualAcambiar en uno y almacena el peor *fitness* de los primeros *#aReemplazar* individuos, se realiza lo mismo para la nueva población pero almacenando el mejor *fitness*.
Una vez inicializados los primeros valores de las estructuras auxiliares con unos, se realiza otra iteración que se encarga de obtener los *#aReemplazar* peores valores de la nueva población marcando con unos

los mismos y dejando en cero el resto, y se obtienen también los mejores *#aReemplazar* valores de la población actual marcando estos con unos y el resto con ceros. Se realiza una tercera iteración en donde se obtiene la primer posición con valor uno del arreglo auxiliar de los mejores individuos, y se calcula la posición correspondiente al individuo, esta posición se almacena en posIndAreemplazarNuevosElitistaGPU, análogamente se realiza para los peores individuos y se guardan sus posiciones en la estructura posIndAreemplazarViejosElitistaGPU.

Luego, se procede a realizar el reemplazo de los *#aReemplazar* peores individuos de la nueva población por los *#aReemplazar* mejores individuos de la población actual. Gracias a las estructuras auxiliares posIndAreemplazarViejosElitistaGPU y posIndAreemplazarNuevosElitistaGPU se conocen los individuos a remplazar.

Un bloque cada dos individuos

En la Figura 4.14 se muestra el pseudocódigo del programa principal para la variante que utiliza un bloque cada dos individuos.

```
Declaración, reserva de memoria e inicialización de variables (GPU y CPU)
Si hay poblaciones en GPU
       Inicialización
       calcularFitnessBinariaGen
       obtenerMejoresIndividuoBinariaGen
       Si el tipo de selección es por rango
              ordenarDescendente
Si hay poblaciones en CPU
      inicialización CPU
      Si el tipo de selección es por rango
            ordenarDescendente
Mientras no se llegue al número de iteraciones deseadas
       Si hay poblaciones en GPU
              Si la iteración actual es múltiplo de 2
                     Si el tipo de selección es por rueda de ruleta
                            asignarProbabilidadRuletaBinariaGen
                     mainKernelBinariaGen
                     calcularFitnessBinariaGen
                     Si el tipo de reemplazo es elitista.
                            reemplazoElitistaSeleccionIndividuosBinariaGen
                     reemplazoBinariaGen
                     Si el mecanismo de selección es por rango.
                            ordenarDescendente
              Sino, si la iteración actual NO es múltiplo de 2
                            implementación análoga(switch población actual por nueva)
       Si hay poblaciones en CPU
              representacionBinariaCPU
              Si el mecanismo de selección es por rango
                     ordenarDescendenteCPU
       Si hay que migrar
              Si hay poblaciones en GPU
                     individosAMigrarRepresentacionBinariaGen
              Si hay poblaciones en CPU y en CPU
                     migracionRepresentacionBinariaGen
                     migracionRepresentacionBinariaCPU
              Si no, si hay poblaciones en CPU y no hay en GPU
                     migracionRepresentacionBinariaCPU
              Si no, si hay poblaciones en GPU y no en CPU
                     migracionRepresentacionBinariaGen
              Si hay poblaciones en CPU
Fin Mientras
Se libera memoria de todas las estructuras
```

Figura 4.14 – Pseudocódigo del programa principal (un bloque cada dos individuos).

Mapeo a memoria

En un bloque cada dos individuos, la idea es procesar los individuos en pedazos de #*hilosPorBloque.* La Figura 4.15 muestra un esquema del mapeo a memoria de los hilos en cada bloque.

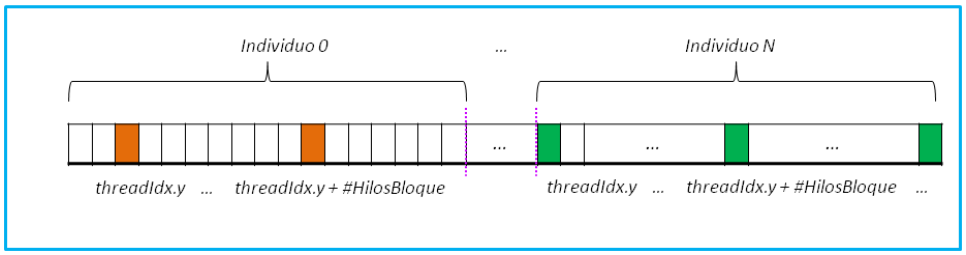


Figura 4.15 – Mapeo a memoria de hilos en un bloque cada dos individuos.

A continuación, se presentan detalles del *kernel* principal y de algunos de los operadores evolutivos implementados para representación binaria. Los operadores que se detallan son: inicialización automática, selección por rango, cruzamiento en un punto, mutación por inversión y reemplazo elitista. En la sección 4.5.1 se describen los distintos operadores evolutivos para cada tipo de representación.

Kernel principal

El *kernel* con el manejo de un bloque cada dos individuos es uno de los principales cambios con respecto a un bloque por población, ya que en caso de que deban realizarse reemplazo elitista y migración, éstos no se realizan en el *kernel* principal, y son funciones globales invocadas desde el *host*, así como también el cálculo del *fitness* de los individuos procesados. Se invoca dos veces a la función de selección para obtener los individuos a cruzar y mutar. Se llama a la función de cruzamiento, la cual ejecuta el cruzamiento según el operador seleccionado, y por último, se invoca a la función de mutación que muta a los dos individuos resultantes del cruzamiento.

Inicialización automática

Para el método de inicialización automática en un bloque cada dos individuos, se sigue la misma idea que en el descrito en un bloque por población, con la diferencia de que cada hilo que se lanza, se encarga de inicializar *2*#GenesEnCromosomaDevice/hilosPorBloque* posiciones en el arreglo, siguiendo el esquema de hilos que se muestra en la Figura 4.15. Todas esas posiciones son inicializadas en *#GenesEnCromosomaDevice/hilosPorBloque* iteraciones, y en cada iteración se inicializan dos posiciones del arreglo. Estas posiciones están dadas por *blockIdx.y*#GenesEnCromosomaDevice*2 + #hilosPorBloque,* y *blockIdx.y*#GenesEnCromosomaDevice*2 + #GenesEnCromosomaDevice + #hilosPorBloque.*

Selección por rango

Para este mecanismo de selección, es necesario ordenar de manera descendente el *fitness* de los individuos de cada población, para lo que se utiliza un vector auxiliar. Dentro del método de selección por rango, cada hilo del bloque selecciona aleatoriamente un valor *x* entre 0 y la cantidad de individuos a seleccionar (que es un parámetro configurable del *ag-cuda-framework*). Luego se accede al índice *x* del vector de posiciones de *fitness* ordenadas, y se obtiene la posición del *fitness* del individuo. Finalmente, a partir de la posición del *fitness* se calcula la posición correspondiente al individuo dentro de la población y se retorna dicha posición.

Cruzamiento en un punto

En este operador se obtienen las dos posiciones de memoria a escribir en GPU, ya que cada hilo realiza el cruzamiento parcial de dos individuos. Luego, se obtiene un número aleatorio para saber si se debe cruzar o no, y otro número aleatorio para obtener el punto de cruzamiento. Para copiar los genes, cada hilo realiza *#genesCromosoma/#hilosPorBloque* iteraciones, cada hilo copia *#genesCromosoma/#hilosPorBloque*2* genes. Esta iteración va desde el número de hilo a *#genesCromosoma* y se incrementa cada *#hilosPorBloque*, siguiendo el esquema de hilos que se muestra en la Figura 4.15. En cada iteración, si la posición del gen a copiar es menor al punto de cruce, entonces

se copia el gen del primer individuo seleccionado en la posición correspondiente a la primera posición de memoria calculada (según el hilo y el bloque) y el gen del segundo individuo seleccionado a la segunda posición de memoria calculada. Si la posición del gen a copiar es mayor o igual al punto de cruce entonces se copia el gen del segundo individuo seleccionado a la primera posición de memoria calculada y el gen del primer individuo seleccionado a la segunda posición de memoria calculada. Si no se debe cruzar, entonces se copia el primer individuo seleccionado en la primera posición calculada y el segundo individuo seleccionado en la segunda posición calculada. Este procedimiento es igual al anterior en cuanto a la cantidad de iteraciones por hilo.

Mutación por inversión

En la mutación por inversión, el hilo *0* calcula las posiciones de memoria a escribir según el bloque y obtiene un número aleatorio que indica si se debe mutar el primer individuo. Si este se debe mutar, entonces se obtiene un valor aleatorio para el punto de mutación, y finalmente se realiza la mutación del gen correspondiente al valor aleatorio obtenido. Se vuelve a obtener un valor aleatorio que indica si se debe mutar el segundo individuo y de ser así, se invierte el valor del gen de la misma manera que el individuo anterior.

Reemplazo elitista

El reemplazo para un bloque cada dos individuos, se realiza de manera análoga a un bloque por población, con la diferencia que para el reemplazo elitista, se utilizan dos funciones globales, una que se encarga de calcular los individuos a reemplazar y los que pasan a la siguiente generación, y otra que se encarga de realizar el reemplazo. A ésta última, se la llama con *#hilosPorBloque* hilos y *#aReemplazar*#Poblaciones* bloques, donde cada bloque se encarga de reemplazar un individuo y cada hilo se encarga de copiar *genesCromosoma/#hilosPorBloque* genes del individuo.

4.6 Archivo de configuración.

El *ag-cuda-framework* incluye un archivo de configuración para parametrizar las ejecuciones. Mediante el archivo de configuración, es posible configurar los siguientes parámetros: el tipo de representación, la cantidad de poblaciones a ser ejecutadas en GPU, la cantidad de poblaciones ejecutarse en CPU, la cantidad de individuos por población, la cantidad de genes por individuo, el mecanismo de selección, el operador de cruzamiento, el operador de mutación, el mecanismo de reemplazo, la probabilidad de cruzamiento, la probabilidad de mutación, el número de iteraciones, el grado de no uniformidad (utilizado en representación real), la configuración de bloques de GPU (un bloque por población o un bloque cada dos individuos), el porcentaje de elitismo (utilizado en selección elitista), el número que indica cada cuantas generaciones migrar. Es posible configurar el modo *debug* para imprimir resultados intermedios, la cantidad de individuos que serán seleccionados en selección por rango, el tipo de inicialización (manual, o aleatoria), si se desean imprimir todos los individuos de cada población o sólo el mejor de cada una. El rango de valores que pueden tomar los genes, y por último, la cantidad de hilos por bloque que se desea configurar (para cuando se selecciona un bloque cada dos individuos). Por más detalle sobre el archivo de configuración consultar el Anexo II.

5 Evaluación Experimental

Resumen: En este capítulo se presenta la evaluación experimental del ag-cuda-framework. Se evalúa la resolución de un caso de estudio para cada tipo de representación disponible en el ag-cuda-framework, y se muestran los resultados obtenidos para cada uno.

5.1 Introducción

El criterio para seleccionar los casos de prueba a ejecutar para cada una de las representaciones no estuvo centrado en el objetivo de encontrar soluciones competitivas en cuanto a calidad con respecto al estado del arte, sino que fue con el objetivo de evaluar los resultados en cuanto a la calidad y performance de GPU vs. CPU, y abarcar una gama importante de problemas donde se puede evaluar la simpleza de implementarlos en el *ag-cuda-framework*. La evaluación se separa en dos partes, la primera parte abarca las pruebas funcionales para corroborar que la implementación en GPU no presenta errores y la segunda parte abarca las pruebas de *performance*.

Las plataformas sobre las cuales se realizaron las pruebas, son para el caso de GPU, una tarjeta NVIDIA GeForce GTX 480 con CUDA 4.1 y para el caso de CPU, un procesador Intel Core i3-2100 3.10 GHz con 8 GB de memoria RAM y sistema operativo Linux Fedora 15.

5.2 Pruebas funcionales

Para realizar las pruebas funcionales, se adaptó el *ag-cuda-framework* para utilizar arreglos de números aleatorios generados en CPU. De esta manera se pueden comparar los resultados de los algoritmos implementados en GPU, con los algoritmos implementados en CPU, utilizando exactamente los mismos números aleatorios, y poder así, verificar su correcta implementación. Para esto, en la implementación correspondiente a cada representación, se generó un arreglo de números aleatorios para cada operador. El tamaño de cada uno de estos arreglos depende del operador y del tipo de representación seleccionado.

En la Figura 5.1, se muestra un ejemplo de arreglos de números aleatorios generados en caso de representación binaria, con *P* poblaciones de *N* individuos de *M* genes cada uno y suponiendo inicialización aleatoria, selección por torneo, cruzamiento en un punto, y mutación por inversión. Como se ve en la Figura 5.1, se generó un arreglo de números aleatorios para la inicialización de tamaño *P*N*M* (es necesario un número aleatorio por cada gen), un arreglo de números aleatorios para la selección de tamaño *P*N*2* (se requieren dos números aleatorios por cada individuo para hacer el torneo), un arreglo de números aleatorios para la probabilidad de cruzamiento de tamaño *(P*N)/2* (es necesario un valor de probabilidad por cada dos individuos a cruzar), un arreglo de números aleatorios para los puntos de cruce de tamaño *(P*N)/2* (se requiere un punto de cruce por cada dos individuos a cruzar), un arreglo de números aleatorios para la probabilidad de mutación de tamaño *P*N* (es necesario un valor de probabilidad por cada individuo a mutar) y un arreglo de números aleatorios para los puntos de mutación de tamaño *P*N* (se necesita un punto de mutación por cada individuo a mutar).

Siguiendo la estrategia descrita previamente se pudo corroborar para un conjunto de pruebas que se considera significativo, que las implementaciones en CPU y GPU para todas las representaciones, operadores de selección, cruzamiento y mutación implementados (ver Sección 4.5.1) coinciden en los resultados.

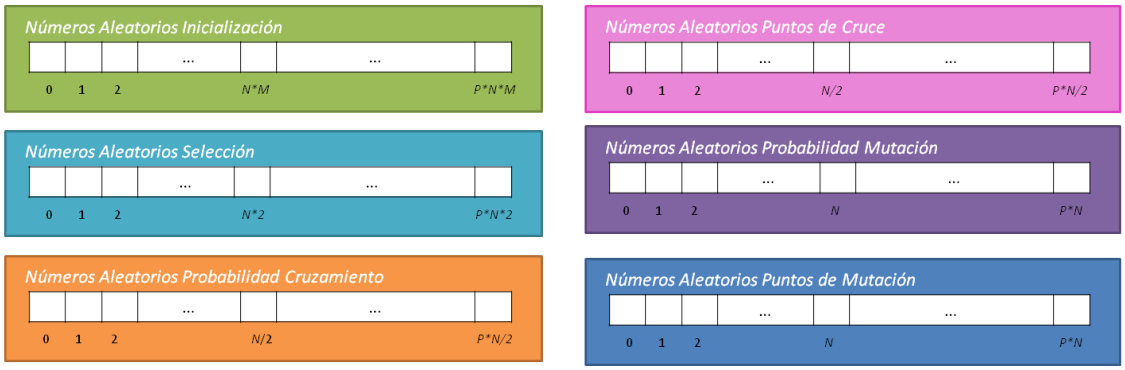


Figura 5.1 – Arreglos de números aleatorios.

5.3 Pruebas de performance

Para realizar las pruebas de performance se eligió un problema representativo para cada tipo de representación. A continuación, se describe cada problema ejecutado para cada una de las representaciones, donde se muestran tanto las configuraciones definidas, como los resultados obtenidos. Se registra en todos los casos el mejor valor obtenido, y el tiempo de ejecución efectivo de cada caso, es decir, se mide el tiempo en que la GPU está procesando, sin incluir funciones de entrada/salida ni tiempo de inicialización de la población.

Para la ejecución de los distintos casos de pruebas correspondientes a cada una de las representaciones en GPU se utilizan tanto métodos de selección, como operadores de cruzamiento diferentes, para mostrar cómo se comportan las diferentes funcionalidades implementadas en el *ag-cuda-framework*. Además, para cada uno de los diferentes tipos de representación, se realizaron ejecuciones con distintas combinaciones de cantidad de poblaciones (1, 2, 4, 8 y 16 poblaciones), tamaños de población (480, 240, 120, 60 y 30 individuos) y cantidad de genes por individuo (5000, 10000 y 20000 genes por individuo) lo cual redunda en un total de 15 casos de prueba a ejecutar para cada una de las distintas representaciones. Cada uno de los casos de prueba fue ejecutado 20 veces, y se presentó el valor promedio de los resultados obtenidos. Antes de ejecutar los casos de prueba finales, se realizó un sondeo de tiempos de ejecución configurando 32, 64, 128, 256 y 512 hilos por bloque. Esto permitió determinar que la configuración óptima es de 64 hilos por bloque, por lo que todas las pruebas de GPU se configuran con 64 hilos por bloque. Para el caso de las ejecuciones en CPU, solamente se consideran ejecuciones con una única población de 480 individuos. El resto de los parámetros se configura igual que para las ejecuciones en GPU, lo cual da un total de 3 casos de pruebas y al igual que en GPU, fueron ejecutados 20 veces. También se realizó la ejecución del algoritmo genético en forma híbrida para el caso de representación entera, con el objetivo de mostrar que puede utilizarse el *ag-cuda-framework* para obtener un mejor *speedup* respecto a la ejecución en GPU. De esta manera, se aprovecha el poder de cálculo de la CPU para que no esté ociosa mientras que está ejecutando la GPU. Con esta configuración se ejecutaron 3 casos de prueba y cada uno de ellos se ejecutó 20 veces.

5.3.1 Caso de estudio representación binaria

El problema estudiado en representación binaria es One Max [41]. Este es un problema clásico en optimización, y es utilizado frecuentemente en estudios experimentales ya que es simple y permite enfocarse en las características del algoritmo y no en las del problema. Este problema consiste en maximizar la cantidad de unos en una cadena binaria. Es por eso que también es conocido como el problema del conteo de unos. La formulación del problema es: $\max \sum_{i=1}^{N} x_i,\ x_i = \{0,1\}\ \forall\ i = 1..N$.

En la Tabla 5.1, se muestran los distintos valores para los parámetros de configuración del *ag-cuda-framework*. Para este caso, el método de selección utilizado es selección por torneo, mientras que el operador de cruzamiento es cruzamiento de un punto, y el operador de mutación es por inversión.

Parámetro	Valor	Parámetro	Valor
tipoRepresentacion	0	gradoNoUniformidad	1
cantidadDePoblacionesEnGpu	1; 2; 4; 8; 16	bloquePorPoblacion	0
cantidadDePoblacionesEnCpu	0	porcentajeElitismo	50
tamanioPoblacion	480; 240; 120; 60; 30	generacionesMigrar	10
cantidadGenesPorIndividuo	5000; 10000; 20000	modoDebug	0
metodoDeSeleccion	1	cantidadIndividuosSeleccionPorRango	100
operadorCruzamiento	0	inicializacionAutomatica	1
operadorMutacion	0	imprimirPoblacionEntera	0
mecanismoDeReemplazo	0	inicioRango	0
probabilidadCruzamiento	90	finRango	0
probabilidadMutacion	50	cantHilosBloque	64
numeroGeneraciones	5000		

Tabla 5.1 – Tabla de configuración de parámetros.

En las Tablas 5.2 y 5.3, se muestran los resultados obtenidos de ejecutar los casos de prueba en GPU y CPU respectivamente.

Caso de Prueba	#Pob.	#Ind.	#Genes	Tiempo Promedio (s)	Mejor Fitness
gpu1.1	1	480	5000	4,36	5000
gpu1.2	1	480	10000	6,79	10000
gpu1.3	1	480	20000	11,65	19387
gpu1.4	2	240	5000	4,21	5000
gpu1.5	2	240	10000	6,61	10000
gpu1.6	2	240	20000	11,41	19254
gpu1.7	4	120	5000	4,16	5000
gpu1.8	4	120	10000	6,53	10000
gpu1.9	4	120	20000	11,35	18987
gpu1.10	8	60	5000	4,13	5000
gpu1.11	8	60	10000	6,50	10000
gpu1.12	8	60	20000	11,30	18596
gpu1.13	16	30	5000	4,12	5000
gpu1.14	16	30	10000	6,49	10000
gpu1.15	16	30	20000	11,28	17906

Tabla 5.2 – Resultados ejecución en GPU.

Como se puede observar en la Tabla 5.2, el aumento en la cantidad de poblaciones de GPU no empeora el tiempo de ejecución aunque dependiendo del problema y la cantidad de individuos, puede empeorar la calidad de la solución obtenida, esto puede deberse a que las poblaciones son de tamaño pequeño y eso hace que se pierda diversidad.

Caso de Prueba	#Pob.	#Ind.	#Genes	Tiempo Promedio (s)	Mejor Fitness
cpu1.1	1	480	5000	19,64	5000
cpu1.2	1	480	10000	39,31	10000
cpu1.3	1	480	20000	78,41	18190

Tabla 5.3 – Resultados ejecución en CPU.

Para este caso, realizando una comparación para las ejecuciones realizadas en GPU vs. CPU se puede observar cómo para los tres casos de GPU y CPU (1.1 a 1.3) el tiempo de ejecución en GPU es notoriamente menor que el tiempo de ejecución en CPU y en general la solución obtenida en GPU es

mejor que la obtenida en CPU, exceptuando el caso de 16 poblaciones. A continuación, en la Tabla 5.4 se muestra el análisis del *speedup* obtenido.

Speedup GPU vs CPU					
#Pob.	#Ind.	#Genes	Tiempo GPU (s)	Tiempo CPU (s)	Speedup
1	480	5000	4,36	19,64	4,50
1	480	10000	6,79	39,31	5,80
1	480	20000	11,65	78,41	6,70

Tabla 5.4 – Tabla Speedup GPU vs. CPU.

Se incluyeron resultados con dos implementaciones distintas de la función de *fitness,* con el objetivo de mostrar cómo pueden variar los tiempos de ejecución de una implementación eficiente con respecto a una implementación ineficiente. En la implementación eficiente del cálculo del *fitness,* se lanza un bloque de hilos por individuo con 64 hilos por bloque y cada hilo se encarga de sumar un gen, mientras que en la implementación ineficiente se lanza un bloque de hilos por población, con *#CromosomasPoblacion/2* hilos, donde cada hilo se encarga de calcular el *fitness* de dos individuos. En la Tabla 5.5 se muestra la comparativa en los tiempos de ejecución (en segundos) obtenidos para ambas pruebas.

Caso de Prueba	#Pob.	#Ind.	#Genes	Tiempo Imp. Eficiente (s)	Tiempo Imp. NO Eficiente (s)	Speedup
gpu1.1	1	480	5000	4,36	73,36	16.83
gpu1.2	1	480	10000	6,79	151,78	22,35
gpu1.3	1	480	20000	11,65	301,60	25,89
gpu1.4	2	240	5000	4,21	59,29	14,08
gpu1.5	2	240	10000	6,61	116,94	17,69
gpu1.6	2	240	20000	11,41	232,30	20,36
gpu1.7	4	120	5000	4,16	54,90	13,20
gpu1.8	4	120	10000	6,53	105,85	16,21
gpu1.9	4	120	20000	11,35	209,85	18,49
gpu1.10	8	60	5000	4,13	48,99	11,86
gpu1.11	8	60	10000	6,50	96,12	14,79
gpu1.12	8	60	20000	11,30	191,24	16,92
gpu1.13	16	30	5000	4,12	32,41	7,87
gpu1.14	16	30	10000	6,49	65,12	10,03
gpu1.15	16	30	20000	11,28	125,14	11,09

Tabla 5.5 – Comparativa implementación eficiente vs. no eficiente de la función de fitness.

Como se puede ver en los resultados obtenidos, la diferencia de tiempos entre una implementación eficiente con una implementación ineficiente es muy notoria. Cabe destacar, que la implementación de la función de *fitness* queda a cargo del usuario ya que es dependiente del problema a resolver. Con los resultados obtenidos, se puede concluir, que la implementación de forma eficiente de esta función es crítica para obtener un buen desempeño computacional de la herramienta.

5.3.2 Caso de estudio representación entera

El problema estudiado en la representación entera es HCSP *(Heterogeneous Computing Scheduling Problem)* [23], y consiste de encontrar una estrategia de planificación de un conjunto de tareas, a ser ejecutadas, con el objetivo de asignar recursos computacionales de forma inteligente, satisfaciendo

algún criterio de eficiencia. A continuación, se presenta el modelo matemático que resuelve el problema. Dado un sistema de computación heterogéneo, compuesto por un conjunto de recursos computacionales $P = \{m_1, \dots, m_M\}$ (dimensión M), y una colección de taras $T = \{t_1, \dots, t_N\}$ (dimensión N) para ser ejecutadas en el sistema heterogéneo, dada una función de tiempo de ejecución $ET: \{1, \dots, N\} \times \{1, \dots, M\} \rightarrow R^+$, donde $ET(i,j)$ es el tiempo requerido para ejecutar una tarea T_i en la máquina m_j, el propósito del HCSP enfocado en minimizar el lapso de tiempo de ejecución de las tareas, es encontrar una función de asignación de tareas que minimice $max_{j \in \{m_1,\dots,m_M\}} \sum_{i \in T, f(i)=j} ET(i,j)$. Esta función se denomina *makespan.*

En la Tabla 5.6 se muestran los valores para los distintos parámetros de configuración del *ag-cuda-framework.* Para este caso el método de selección utilizado es selección por rueda de ruleta, el operador de cruzamiento, es cruzamiento en dos puntos, y el operador de mutación es reajuste aleatorio.

Parámetro	Valor	Parámetro	Valor
tipoRepresentacion	1	gradoNoUniformidad	1
cantidadDePoblacionesEnGpu	1; 2; 4; 8; 16	bloquePorPoblacion	0
cantidadDePoblacionesEnCpu	0	porcentajeElitismo	50
tamanioPoblacion	480; 240; 120; 60; 30	generacionesMigrar	10
cantidadGenesPorIndividuo	512; 1024; 2048	modoDebug	0
metodoDeSeleccion	2	cantidadIndividuosSeleccionPorRango	100
operadorCruzamiento	1	inicializacionAutomatica	1
operadorMutacion	0	imprimirPoblacionEntera	0
mecanismoDeReemplazo	0	inicioRango	0
probabilidadCruzamiento	90	finRango	15, 31, 63
probabilidadMutacion	50	cantHilosBloque	64
numeroGeneraciones	5000		

Tabla 5.6 – Configuración de parámetros.

En las Tablas 5.7 y 5.8 se muestran los resultados obtenidos de ejecutar los casos de prueba en GPU y CPU respectivamente, estos resultados son el tiempo promedio de ejecución y el mejor *makespan* obtenido.

Caso de Prueba	#Genes (#tareas)	#Máquinas	#Pob.	#Ind.	Tiempo Promedio (s)	Mejor Makespan
gpu1.1	512	16	1	480	21,28	26.634,70
gpu1.2	512	16	2	240	11,94	24.149,98
gpu1.3	512	16	4	120	7,28	21.793,98
gpu1.4	512	16	8	60	4,95	23.107,85
gpu1.5	512	16	16	30	3,85	23.439,67
gpu1.6	1024	32	1	480	23,18	24.428,73
gpu1.7	1024	32	2	240	13,85	24.167,06
gpu1.8	1024	32	4	120	9,21	24.328,99
gpu1.9	1024	32	8	60	6,87	22.441,30
gpu1.10	1024	32	16	30	5,68	22.644,17
gpu1.11	2048	64	1	480	29,81	19.134,98
gpu1.12	2048	64	2	240	20,44	25.684,76
gpu1.13	2048	64	4	120	15,82	23.860,05
gpu1.14	2048	64	8	60	13,58	26.421,78
gpu1.15	2048	64	16	30	12,41	32.320,55

Tabla 5.7 – Resultados ejecución en GPU.

Como se puede observar en la Tabla 5.7 el tiempo de ejecución se reduce en forma significativa al aumentar la cantidad de poblaciones.

Caso de Prueba	#Genes (#tareas)	#Máquinas	#Pob.	#Ind.	Tiempo Promedio (s)	Mejor Makespan
cpu1.1	512	16	1	480	33,67	11.200,43
cpu1.2	1024	32	1	480	66,38	12.493,59
cpu1.3	2048	64	1	480	133,61	13.927,31

Tabla 5.8 – Resultados ejecución en CPU.

Para este caso, realizando una comparación para las ejecuciones realizadas en GPU vs. CPU se puede observar cómo para los 3 casos de GPU y CPU (1.1 a 1.3) el tiempo de ejecución en GPU es notoriamente menor que el tiempo de ejecución en CPU. Cabe destacar que la cantidad de generaciones ejecutadas no aseguran la convergencia del AG, sin embargo se puede observar que el *makespan* es mejor en CPU que en GPU, lo que parece estar asociado a la utilización de diferentes generadores de números aleatorios, Mersenne Twister [40] para CPU y CURAND [52] para GPU. En la Tabla 5.9 se muestra una tabla con el análisis del *speedup* obtenido, de esta tabla se desprende que al aumentar la carga en la GPU, aumenta la mejora en performance. Por lo tanto, se puede concluir que para este problema en particular conviene ejecutar una cantidad mayor de poblaciones para lograr un mejor aprovechamiento del recurso GPU.

Speedup GPU vs CPU								
#Genes	#Máquinas	GPU		CPU		Tiempo GPU Promedio (s)	Tiempo CPU Promedio (s)	Speedup
		#Pob.	#Ind.	#Pob.	#Ind.			
512	16	1	480	1	480	21,28	33,67	1,60
1024	32	1	480	1	480	23,18	66,38	2,90
2048	64	1	480	1	480	29,81	133,61	4,50
512	16	16	30	1	480	3,85	33,67	8,74
1024	32	16	30	1	480	5,68	66,38	11,68
2048	64	16	30	1	480	12,41	133,61	10,76

Tabla 5.9 –Speedup GPU vs. CPU.

Como se mencionó en la Sección 5.3 para el caso de representación entera se realizó una ejecución híbrida del *ag-cuda-framework*. Para esta ejecución se seleccionaron tres casos de pruebas, donde se configuraron, 512, 1024 y 2048 genes por individuo, y 16, 32 y 64 máquinas respectivamente para los tres casos, además, cada caso se configuró con 15 poblaciones en GPU y una población en CPU. El resto de las configuraciones se mantienen incambiadas respecto a los demás casos ejecutados para GPU y CPU. Se decidió realizar las pruebas que se presentan con 15 poblaciones en GPU y una población en CPU (con algunas otras configuraciones evaluadas, por ejemplo 14 poblaciones en GPU y 2 poblaciones en CPU los tiempos de procesamiento empeoraban). En la Tabla 5.10 se muestran los resultados obtenidos al realizar la ejecución híbrida.

Caso de Prueba	#Genes (#tareas)	#Máquinas	#Pob. GPU	#Pob. CPU	#Ind.	Tiempo Promedio (s)	Mejor Makespan
hibrido1.1	512	16	15	1	30	3,82	12.154,92
hibrido1.2	1024	32	15	1	30	5,52	14.033,72
hibrido1.3	2048	64	15	1	30	11,97	15.867,28

Tabla 5.10 – Resultados ejecución híbrida.

En los resultados obtenidos hay mejoras marginales a medida que se incrementan los tamaños en cuanto a cantidad de genes y poblaciones, aunque no son muy significativas, hay una reducción máxima

de 0,44 segundos. Es de esperar, que se pueda obtener mayor provecho de este tipo de ejecución con dimensionamientos de problemas mayores a los resueltos en este proyecto.

5.3.3 Caso de estudio representación real

El problema estudiado en la representación real es el problema generalizado de Schwefel [64], que consiste en una función para comprobar el rendimiento de distintos modelos de optimización numérica. Es una función multimodal para la que existen varios mínimos locales cercanos a un mínimo global. A continuación se presenta la función de Schwefel. La función de *fitness* es $f_{11}(x) = 418{,}9829 * n - \sum_{i=1}^{n} x_i * \sin\sqrt{|x_i|}$, dentro de un dominio de búsqueda de $-500 \leq x_i \leq 500$, mínimo global $x^* = (420{,}9687, \ldots, 420{,}9687)$, *fitness* óptimo $f_{11}(x^*) = 0$, tamaño del individuo $n = 5000, 10000\ y\ 20000$. Con este problema lo que se busca es minimizar la función descrita anteriormente.

En la Tabla 5.11 se muestran los valores de los distintos parámetros de configuración del *ag-cuda-framework*. Para este caso el método de selección utilizado es selección por torneo, el operador de cruzamiento es cruzamiento aritmético individual y el operador de mutación es mutación uniforme.

Parámetro	Valor	Parámetro	Valor
tipoRepresentacion	3	gradoNoUniformidad	1
cantidadDePoblacionesEnGpu	1; 2; 4; 8; 16	bloquePorPoblacion	0
cantidadDePoblacionesEnCpu	0	porcentajeElitismo	50
tamanioPoblacion	480; 240; 120; 60; 30	generacionesMigrar	10
cantidadGenesPorIndividuo	5000; 10000; 20000	modoDebug	0
metodoDeSeleccion	1	cantidadIndividuosSeleccionPorRango	100
operadorCruzamiento	0	inicializacionAutomatica	1
operadorMutacion	0	imprimirPoblacionEntera	0
mecanismoDeReemplazo	0	inicioRango	-500
probabilidadCruzamiento	90	finRango	500
probabilidadMutacion	50	cantHilosBloque	64
numeroGeneraciones	5000		

Tabla 5.11 – Configuración de parámetros.

En las Tablas 5.12 y 5.13 se muestran los resultados obtenidos de ejecutar los casos de prueba en GPU y CPU respectivamente. En la Tabla 5.13 podemos ver como al aumentar las poblaciones no empeora el tiempo de ejecución sino que mejora levemente.

Para este caso, realizando una comparación para las ejecuciones realizadas en GPU vs. CPU se puede observar que para todos los casos, la ejecución en GPU es más rápida que la ejecución en CPU. Además, es importante destacar, que la calidad de las soluciones obtenidas en GPU es mejor que en CPU. En la Tabla 5.14 se muestra el *speedup* obtenido.

En la Tabla 5.14 se puede observar un *speedup* elevado en la ejecución de GPU vs. la de CPU. Esto se debe a que la GPU es un dispositivo que actualmente está muy orientado al cálculo de operaciones de punto flotante. Las operaciones utilizadas en este problema son operaciones trascendentales (como seno y raíz cuadrada) y la GPU tiene unidades especiales llamadas SFU (*Special Functions Units*) para resolverlas (ver Sección 3.3.2), por esto el desempeño es bastante mejor que en CPU. Este tipo de problemas es uno de los mejores escenarios para la ejecución en GPU, ya que se tiene un gran paralelismo sin dependencias de datos, donde se deben resolver operaciones complejas, y la GPU es especialmente eficiente para tratarlas.

Caso de Prueba	#Pob.	#Ind.	#Genes	Tiempo Promedio (s)	Mejor F(x)
gpu1.1	1	480	5000	15,13	1.191.190,12
gpu1.2	1	480	10000	28,50	3.033.728,75
gpu1.3	1	480	20000	55,16	7.027.185,00
gpu1.4	2	240	5000	14,95	1.184.166,00
gpu1.5	2	240	10000	28,26	3.047.566,25
gpu1.6	2	240	20000	54,79	7.048.665,00
gpu1.7	4	120	5000	14,90	1.215.280,37
gpu1.8	4	120	10000	28,20	3.075.218,25
gpu1.9	4	120	20000	54,75	7.081.044,00
gpu1.10	8	60	5000	14,88	1.247.485,37
gpu1.11	8	60	10000	28,17	3.112.783,00
gpu1.12	8	60	20000	54,67	7.139.546,50
gpu1.13	16	30	5000	14,84	1.284.621,75
gpu1.14	16	30	10000	28,14	3.170.169,50
gpu1.15	16	30	20000	54,66	7.213.832,50

Tabla 5.12 – Resultados ejecución en GPU.

Caso de Prueba	#Pob.	#Ind.	#Genes	Tiempo Promedio(s)	Mejor F(x)
cpu1.1	1	480	5000	484,46	1.345.113,25
cpu1.2	1	480	10000	967,54	3.271.772,50
cpu1.3	1	480	20000	1.936,45	7.360.746,00

Tabla 5.13 – Resultados ejecución en CPU.

Speedup GPU vs CPU					
#Pob.	#Ind.	#Genes	Tiempo GPU Promedio(s)	Tiempo CPU Promedio(s)	Speedup
1	480	5000	15,13	484,46	32,01
1	480	10000	28,50	967,54	33,94
1	480	20000	55,16	1.936,45	35,10

Tabla 5.14 – Speedup GPU vs. CPU.

5.3.4 Caso de estudio representación de permutación

El problema estudiado en representación de permutación es TSP *(Travelling Salesman Problem)* [15], que consiste de encontrar la ruta de costo mínimo para un viajante que debe visitar cada ciudad en una lista, exactamente una vez, y regresar al punto de partida. Este problema es fácil de describir pero difícil de resolver, y se ha demostrado que es NP-Completo, es decir, que no existe ningún algoritmo capaz de encontrar una solución en tiempo polinomial. Además, es aplicable a una gran cantidad de problemas reales de ruteo y *schedulling.* Por los aspectos mencionados anteriormente en los últimos tiempos se ha abordado el TSP mediante métodos alternativos, tales como, algoritmos evolutivos. Sea $G = (V, E)$ donde $V = \{1, \dots, n\}$ y $E = \{(i, j): i, j \in V\}$ un grafo y sea c_{ij} el costo asociado al arco (i, j) el problema TSP se puede formular de la siguiente manera:

$$\min \sum_{(i,j)\in E} c_{ij} x_{ij}$$

$$\sum_{\{c_j:(i,j)\in E\}} x_{ij} = 1 \; \forall \; i$$

$$\sum_{\{i:(i,j)\in E\}} x_{ij} = 1 \; \forall \; j$$

$$\sum_{\{(i,j)\in E, i\in S, j\in S\}} x_{ij} \leq |S| - 1 \; para \; S \subset V, 2 \leq |S| \leq |V| - 2$$

En la Tabla 5.15 se muestran los valores de los distintos parámetros de configuración del *ag-cuda-framework*. Para este caso el método de selección utilizado es selección por rueda de ruleta, el operador de cruzamiento es OX y el operador de mutación es por intercambio. Se ejecutan las pruebas con 30 poblaciones de 32 individuos cada una. Cabe destacar que para esta representación se ejecutaron menos pruebas que para el resto de las representaciones, esto se debe a que los tiempos de ejecución de GPU son levemente mayores a los de CPU y no fue posible realizar una comparativa favorable para GPU.

Parámetro	Valor	Parámetro	Valor
tipoRepresentacion	2	gradoNoUniformidad	1
cantidadDePoblacionesEnGpu	30	bloquePorPoblacion	1
cantidadDePoblacionesEnCpu	0	porcentajeElitismo	25
tamanioPoblacion	32	generacionesMigrar	50
cantidadGenesPorIndividuo	101; 493; 1000	modoDebug	0
metodoDeSeleccion	2	cantidadIndividuosSeleccionPorRango	100
operadorCruzamiento	1	inicializacionAutomatica	1
operadorMutacion	0	imprimirPoblacionEntera	0
mecanismoDeReemplazo	1	inicioRango	0
probabilidadCruzamiento	90	finRango	100, 492, 999
probabilidadMutacion	50	cantHilosBloque	512
numeroGeneraciones	5000		

Tabla 5.15 – Configuración de parámetros.

Debido a que los tiempos de ejecución en GPU obtenidos mediante esta representación no fueron ventajosos con respecto a CPU, se decidió almacenar en memoria de otra forma a las poblaciones de Permutación (ver Sección 4.5.2) para poder acceder de forma *coalesced* a memoria global. Cabe destacar, que los tiempos no mejoraron con respecto a la implementación anterior. Esto se debe a que, a pesar de acceder de forma *coalesced* a memoria global, la forma de configuración que se implementó para esta representación es de un bloque por población, y no de un bloque cada dos individuos, ya que en permutación no es posible considerar a cada gen independientemente (ver Sección 4.2.1). En las Tablas 5.16, 5.17, y 5.18 se muestran los resultados obtenidos en GPU con la población ordenada, en GPU con acceso *coalesced* y en CPU, respectivamente.

Caso de Prueba	#Pob.	#Ind.	#Genes	Tiempo Promedio (s)	Mejor Costo Solución
gpu1.1	30	32	101	61,80	708,83
gpu1.2	30	32	493	1315,61	114.658,25
gpu1.3	30	32	1000	5124,93	163.441.472,00

Tabla 5.16– Resultados obtenidos en GPU con población ordenada.

Caso de Prueba	#Pob.	#Ind.	#Genes	Tiempo Promedio (s)	Mejor Costo Solución
gpu1.1	30	32	101	58,00	689,97
gpu1.2	30	32	493	1481,05	112.078,96
gpu1.3	30	32	1000	5999,46	162.151.040,00

Tabla 5.17– Resultados obtenidos en GPU con acceso coalesced a memoria global.

Caso de Prueba	#Pob.	#Ind.	#Genes	Tiempo Promedio (s)	Mejor Costo Solución
cpu1.1	30	32	101	31,78	732,68
cpu1.2	30	32	493	566,15	117.229,77
cpu1.3	30	32	1000	2243,78	166.231.76
cpu1.4	1	960	101	38,27	690,56
cpu1.5	1	960	493	586,02	91.037,32
cpu1.6	1	960	1000	2314,65	137.381.056,00

Tabla 5.18– Resultados obtenidos en CPU.

5.4 Resumen

En este capítulo se abordaron distintos problemas conocidos de optimización, utilizando el *ag-cuda-framework* desarrollado, mostrando la utilidad y la flexibilidad de la herramienta.

Los resultados obtenidos, en representación de permutación, no fueron buenos en cuanto al tiempo de ejecución. A pesar de haber modelado de dos formas distintas el almacenamiento de la población en memoria, para tratar de mejorar la eficiencia de los operadores evolutivos y de la función de *fitness* (ver Sección 4.5.2), los resultados no mejoraron. Sin embargo, para el resto de las representaciones, los resultados obtenidos muestran que la implementación en GPU es muy superior en eficiencia que la versión en CPU, alcanzando *speedups* de hasta 6.70x para representación binaria, 11.68x para representación entera y 35.10x para representación real.

6 CONCLUSIONES Y TRABAJO FUTURO

Analizando los objetivos planteados al iniciar el proyecto, se puede afirmar que en general éstos han sido alcanzados de manera exitosa.

Se desarrolló un *framework (ag-cuda-framework)* para instanciar algoritmos genéticos en arquitecturas híbridas que incluyan GPUs de manera sencilla para el usuario. La herramienta abarca diversos tipos de representación y sus correspondientes operadores evolutivos. Además, el *ag-cuda-framework*, soporta ejecuciones tanto en GPU, como en CPU y además ejecuciones híbridas.

Es importante destacar, que la curva de aprendizaje del lenguaje de programación CUDA todavía es muy alta, por lo que se realizó un esfuerzo significativo en cuanto a la investigación del mismo. Con la implementación de esta herramienta, se facilita el uso de la tarjeta gráfica como recurso computacional para la resolución de problemas de optimización, sin la necesidad de tener conocimientos avanzados en programación CUDA. Sin embargo, el usuario que utilice esta herramienta, debe tener conocimientos básicos en programación en GPU, ya que la implementación de la función de *fitness* queda a su cargo, debido a que es dependiente del problema a resolver.

Se realizó un diseño modular y extensible, en el que es sencillo agregar nuevas funcionalidades. El *ag-cuda-framewrok,* a nuestro criterio, es simple de usar ya que durante la evaluación experimental de los distintos casos de prueba, nos resultó sencillo diseñar los diferentes escenarios de ejecución utilizando el archivo de configuración del *ag-cuda-framework*.

Se diseñaron y ejecutaron un conjunto de problemas para cada tipo de representación, que permitieron realizar pruebas funcionales y de performance. Gracias a estas pruebas, se pudo evaluar y determinar que el *ag-cuda-framework* se comporta de la manera esperada. Es importante notar el esfuerzo adicional realizado en la implementación de un *framework* complementario, que genera los mismos números aleatorios en GPU y CPU, para poder verificar la correctitud de los resultados.

A partir de los casos de prueba seleccionados, se evaluaron los resultados obtenidos en cuanto a la performance, y éstos superan las expectativas, obteniendo un *speedup* de hasta 6.70x para representación binaria, 11.68x para representación entera y 35.10x para representación real, obteniendo para esta última representación el mejor *speedup.* Cabe mencionar, que los resultados obtenidos en representación de permutación, no fueron buenos en cuanto al tiempo de ejecución. A pesar de haber realizado un esfuerzo considerable para tratar de mejorar la eficiencia de los operadores evolutivos y de la función de *fitness*, este no tuvo éxito.

Se realizó una completa documentación de la herramienta, donde se detalla la implementación de las principales funciones del *ag-cuda-framework*, y se provee además de un manual de usuario, donde se explica detalladamente cómo utilizarlo.

Existen diversos trabajos que por motivos de alcance del proyecto pueden considerarse como trabajos futuros.

En primer lugar, destacamos como trabajo futuro, realizar una investigación en profundidad sobre la implementación de la representación de permutación, con un enfoque similar al que se utilizó para el resto de las representaciones (un bloque cada dos individuos), contexto en el que se hace un mejor aprovechamiento de la GPU como recurso de cómputo. En la representación de permutación, no es trivial manejar los genes de manera independiente, ya que cada gen necesita conocer la información de

los demás (no puede haber elementos repetidos), esto es un problema en sí mismo, y es por esto que quedó fuera del alcance del proyecto.

En segundo lugar, se podría ampliar la investigación sobre alternativas de ordenamiento en GPU, especialmente utilizando bibliotecas disponibles públicamente, para mejorar la implementación del mecanismo de remplazo elitista.

Adicionalmente, sería bueno ahondar en la investigación sobre la generación de números aleatorios en GPU, porque como se puede ver en la Sección 5.3, en el caso de estudio de representación entera, se dan mejores resultados en cuanto a la función a optimizar en las ejecuciones en CPU. La correctitud de la implementación de la función de *fitness* fue verificada por lo que es de suponer que el problema está en la utilización de distintos generadores de números aleatorios en GPU y CPU, ya que es la única diferencia entre la implementación de GPU y CPU. Esto puede ocurrir debido a la influencia de los números aleatorios en los cálculos que se realizan en la función de *fitness* para este problema particular. Cabe destacar, que esto no ocurre para el resto de las implementaciones.

En cuarto lugar, se podría extender la implementación del *ag-cuda-framework,* agregando nuevos mecanismos de selección y reemplazo, y nuevos operadores evolutivos, tales como el operador de cruzamiento basado en arcos y mutación por mezcla de representación de permutación. Estos operadores no fueron incluidos dentro del alcance del proyecto, debido a su complejidad y a las limitantes que se presentan a la hora de implementar funciones en la arquitectura de las GPUs utilizadas para desarrollar el proyecto, principalmente la de no poder declarar memoria dinámica.

En quinto lugar, sería interesante poder adaptar el *ag-cuda-framework* para que pueda realizar su ejecución en una plataforma multi-GPU y de esta manera es de esperar que se acelere aún más el tiempo de ejecución, así como poder escalar en la dimensionalidad de los problemas tratables.

Finalmente, se podrían evaluar los casos de prueba presentados en este proyecto sobre tarjetas de la nueva generación de arquitecturas de GPU (*Kepler)* para poder evaluar el desempeño del *ag-cuda-framework* en esta nueva arquitectura ya que presenta una evolución considerable en la capacidad de cómputo.

ANEXO I – ALGUNOS CONCEPTOS SOBRE CUDA

A continuación se describe el acceso *coalesced* a memoria global, un factor importante que afecta significativamente la *perfomance* en aplicaciones que utilizan la GPU, también se describen las *compute capabilities* de la arquitectura Fermi que son las *compute capabilities* 2.x.

Acceso coalesced a memoria global

El acceso a memoria global es por segmentos, aun cuando se quiera leer una palabra, si no se utilizan todos los datos del segmento, se desperdicia ancho de banda. Los segmentos están alineados a múltiplos de 32, 64 y 128 bytes y cada solicitud de acceso a memoria global de un *warp* se parte en dos solicitudes, una para cada *half-warp*, que son atendidas independientemente. Los accesos a memoria de hilos de un *half-warp* se fusionan en una o más transacciones según características que dependen de las *compute capabilities* de la tarjeta.

En la Figura 3.11 se muestra una comparación entre un programa de CUDA con acceso *coalesced* a memoria y uno con acceso no *coalesced,* para el caso de un *warp* de tamaño 32. En la Figura 3.11A el acceso a los datos se realiza en un *loop* donde cada hilo lee una fila de la matriz Md. Asumiendo que los hilos de un *warp* leen filas adyacentes, tenemos que en la iteración 0, los hilos de un *warp* leen el elemento 0 de la fila 0, hasta el elemento 31 de la fila 0. En este caso no hay ningún acceso *coalesced*. Un patrón de acceso más eficiente se muestra en la Figura 3.11B, donde cada hilo lee una columna de Nd. En la iteración 0, los hilos de un *warp* leen el elemento 0 de la columna 0 a la columna 31, todos estos accesos son *coalesced*.

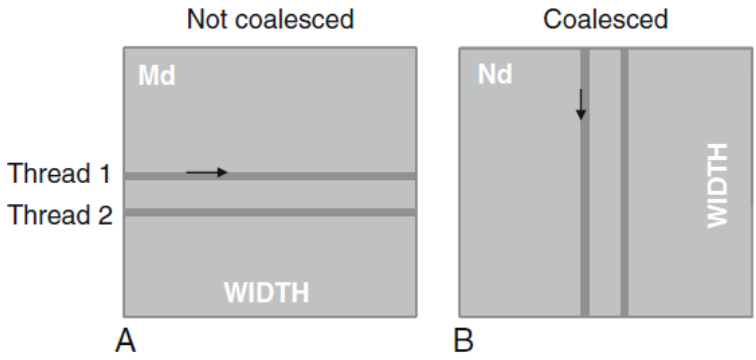


Figura 3.11 – Acceso coalesced vs. Acceso no coalesced.

Compute Capabilities 2.x

A continuación se describen las características más relevantes de las GPU con *compute capabilities* 2.x [69].

En las GPUs con compute *capabilities* 2.x se usa la jerarquía de caches L1/L2 para cache local y acceso a memoria global, hay una cache L1 para cada multiprocesador y una cache L2 compartida por todos los multiprocesadores. El mismo *chip* de memoria es utilizado tanto para L1 como para memoria compartida y el tamaño que se le dedica a cada uno se puede configurar en cada llamada al *kernel*. Los accesos a memoria global en las tarjetas con *compute capabilities* 2.x se procesan por *warp*. Los multiprocesadores de estas tarjetas pueden ser configurados para tener un tamaño de 48 KB de memoria compartida. Esta memoria compartida tiene 32 bancos, y al igual que en memoria global, los accesos se procesan por *warp*. Los conflictos de bancos se dan entre hilos pertenecientes a la primera mitad de un *warp* e hilos pertenecientes a la segunda mitad del *warp,* para evitar estos conflictos de bancos hay que ajustar la disposición de los datos en memoria compartida. Se ha mejorado el hardware de la memoria compartida en las GPUs con *compute capabilities* 2.x para soportar múltiples accesos a

palabras y para disminuir los conflictos de bancos para accesos de 8-bits, 16-bits, 64-bits, o 128-bits por hilo.

Divergencia de threads

Las instrucciones de control de flujo como *if*, *while*, *switch*, impactan significativamente en el rendimiento, causando la divergencia de hilos de un mismo *warp* [50], es decir que sigan caminos de ejecución diferentes. Si esto ocurre, los diferentes caminos de ejecución serán serializados hasta que todos los hilos del warp vuelvan al mismo el *program counter* [75]. Esto incrementa el número de instrucciones ejecutadas en el *warp*. Cuando todos los caminos de ejecución diferentes sean completados, los hilos convergerán al mismo camino de ejecución. Para obtener mejor *performance* en los casos donde el control de flujo depende del *threadID*, la condición de control debe estar implementada con el fin de minimizar el número de *warps* divergentes. Esto es posible porque la distribución de los *warps* dentro de los bloques es determinística.

ANEXO II – ARCHIVO DE CONFIGURACIÓN

El archivo de configuración contiene diferentes parámetros que se deben configurar previamente a ejecutar el *Framework,* teniendo en cuenta cual es el problema particular que se desea resolver. El archivo de configuración de nombre *config.cfg* se encuentra en el directorio /*ag-cuda-framework*/ y es un archivo con extensión .cfg que puede ser editado con cualquier editor de texto. A continuación se describirán uno a uno los parámetros existentes en el archivo.

"representación", indica el tipo de representación del algoritmo genético que se va a utilizar en el *ag-cuda-framework* para resolver el problema planteado. Existen cuatro tipos de representaciones las cuales están descritas en la sección 4.3.1. Es un parámetro de tipo *Integer* que debe cumplir la condición 0 <= representación <= 3.

"*cantidadDePoblacionesEnGPU*", indica la cantidad de poblaciones que van a ser ejecutadas en la GPU. Es un parámetro de tipo *Integer* que debe cumplir la condición 0 <= cantidadDePoblacionesEnGpu.

"*cantidadDePoblacionesEnCPU*", indica la cantidad de poblaciones que van a ser ejecutadas en la CPU. Es un parámetro de tipo *Integer* que debe cumplir la condición 0 <= cantidadDePoblacionesEnCpu.

"*tamanioPoblacion*", indica la cantidad de individuos por población, tanto las poblaciones de CPU como de GPU tendrán la misma cantidad de individuos. Es un parámetro de tipo *Integer* que debe cumplir las condiciones (0 < tamanioPoblacion) && ((tamanioPoblacion mod 2) == 0).

"cantidadGenesPorIndividuo", indica la cantidad de genes que tendrá cada individuo de las distintas poblaciones tanto en CPU como en GPU. Es un parámetro de tipo *Integer* que tiene que cumplir la condición 0 < cantidadGenesPorIndividuo.

"metodoDeSeleccion", indica que operador de selección se va a utilizar durante la ejecución del *ag-cuda-framework*. Es un parámetro de tipo *Integer* que debe cumplir la condición 0 <= metodoDeSeleccion <= 2. En la Sección 4.5.1 se muestran cuáles son las distintas opciones disponibles.

"operadorCruzamiento", indica que operador de cruzamiento va a ser utilizado en la ejecución del *ag-cuda-framework*. Es un parámetro de tipo *Integer* que debe cumplir las siguientes condiciones dependiendo del tipo de representación seleccionado, si (representacion == 0) || (representacion == 1) entonces 0 <= operadorCruzamiento <= 2, si (representacion == 2) entonces 0 <= operadorCruzamiento <= 2, si (representacion == 3) entonces 0 <= operadorCruzamiento <= 2. En la Sección 4.5.1 se muestran cuáles son las distintas opciones disponibles.

"operadorMutacion", indica que operador de mutación va a ser utilizado en la ejecución del *ag-cuda-framework*. Es un parámetro de tipo *Integer* que debe cumplir las siguientes condiciones dependiendo del tipo de representación seleccionado, si (representacion == 0) entonces operadorMutacion = 0, si (representacion == 1) entonces 0 <= operadorMutacion <= 1, si (representacion == 2) entonces 0 <= operadorMutacion <= 2, si (representacion == 3) entonces 0 <= operadorMutacion <= 1. En la Sección 4.5.1 se muestran cuáles son las distintas opciones disponibles.

"mecanismoDeReemplazo", indica cual va a ser el mecanismo de reemplazo utilizado en la ejecución del *ag-cuda-framework*. Es un parámetro del tipo *Integer* que debe cumplir la condición 0 <= mecanismoDeReemplazo <= 1.

“probabilidadCruzamiento”, indica cual es la probabilidad con la que se aplica el operador de cruzamiento elegido. Es un parámetro de tipo *Float* que debe cumplir la condición 0 <= probabilidadCruzamiento <= 100.

“probabilidadMutacion”, indica cual es la probabilidad con la que se aplica el operador de mutación elegido. Es un parámetro de tipo *Float* que debe cumplir la condición 0 <= probabilidadCruzamiento <= 100.

“numeroGeneraciones”, indica la cantidad de iteraciones que iterará el algoritmo. Es un parámetro de tipo *Integer* que debe cumplir la condición 0 < numeroGeneraciones.

“gradoNoUniformidad”, indica el grado de no uniformidad y es utilizado como un coeficiente en el cálculo de la mutación no uniforme de la representación real. Es un parámetro de tipo *Float* que debe cumplir la condición, si (representacion == 3 && operadorMutacion == 1) entonces gradoNoUniformidad <= 1000 (Michalewicz sugiere el valor 5).

“bloquePorPoblacion”, indica la configuración de la cantidad de bloques por población que serán ejecutados en el algoritmo como es explicado en la Sección 4.4. Es un parámetro de tipo *Integer* que debe cumplir la condición 0 <= bloquePorPoblacion <= 1. Ver Sección 4.4 para las distintas configuraciones.

“porcentajeElitismo”, indica qué porcentaje de la población serán seleccionados para pasar a la nueva población. Es un parámetro de tipo *Float* que debe cumplir la condición 0 <= porcentajeElitismo <= 100 si (mecanismoDeReemplazo == 1).

“generacionesMigrar”, indica cada cuantas generaciones se llevaran a cabo las migraciones. Es un parámetro de tipo *Integer* que debe cumplir la condición 0 <= generacionesMigrar, si generacionesMigrar vale cero o mayor que numeroGeneraciones no se migra.

“modoDebug”, indica si se debe imprimir resultados intermedios en cada generación. Es un parámetro de tipo *Integer* que debe cumplir la condición 0 <= modoDebug <= 1. Cero, indica que solo se imprimen los valores correspondientes a las generaciones inicial y final. Uno, indica que se imprimen los valores correspondientes a las generaciones intermedias, además de la inicial y final.

“cantidadIndividuosSeleccionPorRango”, indica el tamaño del rango de individuos que pueden ser seleccionados según su *fitness* en la selección por rango. Es un parámetro de tipo *Integer* que debe cumplir la condición 0 < cantidadIndividuosSeleccionPorRango <= tamanioPoblacion si (metodoDeSeleccion == 0).

“inicializacionAutomatica”, indica el método de inicialización de las poblaciones que se utilizará. Es un parámetro de tipo *Integer* que debe cumplir la condición 0 <= inicializacionAutomatica <= 1. Cero, indica que se utilizará la inicialización manual implementada por el usuario (ver Sección 4.2). Uno, indica que se utilizará la inicialización automática que inicializa las poblaciones aleatoriamente.

“imprimirPoblacionEntera”, indica si se debe imprimir la población entera o el mejor individuo de la población. Es un parámetro de tipo *Integer* que debe cumplir la condición 0 <= imprimirPoblacionEntera <= 1. Cero, indica que se debe imprimir el mejor individuo de la población. Uno, indica que se debe imprimir la población entera.

“inicioRango” y “finRango”, especifican un rango de valores para los individuos de las poblaciones en el caso de la representación entera y real. Son parámetros de tipo *Float*. Si inicioRango == finRango no se tienen en cuenta, si son distintos, los genes de cada individuo de las distintas poblaciones deben estar en el rango.

#hilosPorBloque indica la cantidad de hilos por bloque que se configurará para ejecutar en la GPU. Es un parámetro de tipo *Integer* que debe cumplir las condiciones (2 <= *#hilosPorBloque* <= 1024) && (*#hilosPorBloque* es potencia de 2) si (bloquePorPoblacion == 0).

ANEXO III – INSTRUCCIONES BÁSICAS DEL FRAMEWORK

Para realizar una ejecución del *ag-cuda-framework* es necesario previamente realizar algunas configuraciones que se detallan a continuación.

Primeramente hay que realizar la configuración del archivo de configuración */ag-cuda-framework/config.cfg*, en el Anexo II se detallan cada uno de los parámetros de configuración así como también sus posibles valores.

Si se configuró el *ag-cuda-framework* para que utilice inicialización manual de las poblaciones (inicializacionAutomatica = 0 en */ag-cuda-framework/config.cfg*) es necesario implementar como se inicializan las poblaciones, para ello, se debe modificar el archivo de inicialización correspondiente según el tipo de representación y la modalidad de ejecución (parámetros representación y bloquePorPoblacion en *config.cfg*) que se encuentran en el directorio */ag-cuda-framework/inicializacion*. Por ejemplo, si se configuró para utilizar el tipo de representación binaria y el modo de ejecución un hilo por gen (representacion = 0 y bloquePorPoblacion = 0) se debe modificar el archivo */ag-cuda-framework/inicializacion/initBinariaGen.cu*, en caso de haber configurado el modo de ejecución un bloque por población (bloquePorPoblacion = 1) se debe modificar el archivo */ag-cuda-framework/inicializacion/initBinaria.cu*, en caso de haber configurado para ejecutar poblaciones en CPU (cantidadDePoblacionesEnCpu > 0 en *config.cfg*) se debe modificar el archivo */ag-cuda-framework/inicializacion/initBinariaCPU.cu*.

Luego es necesario implementar la función de *fitness* según el problema que se desee resolver. En este caso y teniendo en cuenta la representación binaria como ejemplo para la ejecución en GPU se debe modificar el método *calcularFitnessBinariaGPU* correspondiente a la clase */ag-cuda-framework/fitness/fitnessBinaria.cu* y para la ejecución en CPU se debe modificar el método *calcularFitnessBinariaCPU* correspondiente a la misma clase.

Finalmente para compilar y *linkeditar* el proyecto se debe ejecutar el comando *make* situándose en la raíz del proyecto */ag-cuda-framework/* y luego para correrlo, se debe ejecutar el comando *make run*.

REFERENCIAS

[1] A. Munawar, M. Wahib, M. Munetomo, K. Akama, Hybrid of genetic algorithm and local search to solve MAX, SAT problem using nVidia CUDA framework, Genet Program Evolvable 10, pp. 391-415, 2009.

[2] A. D. Freed, ll. S. Iskovitz, Development and applications of a Rosenbrock integrator, National Aeronautics and Space Administration, Office of Management, Scientific and Technical Information Program, 1996.

[3] B. Miller y D. Goldberg, Genetic Algorithms, Tournament Selection, and the Effects of Noise, Complex Systems 9, pp. 193-212, 1995.

[4] D. Applegate, R. Bixby, V. Chvátal, W. Cook, "The Traveling Salesman Problem", ISBN 0-691-12993-2, 2006.

[5] D. B. Fogel, An evolutionary approach to the traveling salesman problem, Biological Cybernetics, 60, pp. 139-144, 1988.

[6] D. B. Fogel, Applying evolutionary programming to selected traveling salesman problems, Cybernetics and Systems, 24, pp. 27-36, 1993.

[7] D. E. Goldberg, Jr. R. Lingle, Alleles, loci and the traveling salesman problem, en Proceedings of the First International Conference on Genetic Algorithms and Their Applications, pp. 154-159, 1985.

[8] D. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison, Wesley, 1989.

[9] D. Whitley, J. Kauth, Genitor: A different genetic algorithm. Proceedings of the Rocky Mountain Conference on Artificial Intelligence, pp. 118-130, 1988.

[10] D. Whitley, T. Starkweather, Genitor II: A distributed genetic algorithm, Journal of Experimental and Theorical Artficial Intelligence, 2, pp. 189-214, 1990.

[11] D. Whitley, T. Starkweather, D. Shaner, The traveling salesman and sequence scheduling: Quality solutions using genetic edge recombination, en Davis, L. (ed.) Handbook of Genetic Algorithms, pp. 350-372, 1991.

[12] E. Aarts y J. Korst, Simulated Annealing and Boltzmann Machines, John Wiley & Sons, 1989.

[13] E. Alba, J. Troya. Algoritmos Genéticos Incrementales. Informe técnico, 1995.

[14] E. Alba, J. Troya, An Analysis of Synchronous and Asynchronous Parallel Distribuited Genetic Algorithms with Structured and Panmictic Islands, pp. 248-256, 1999.

[15] E.L. Lawler, A.H.G. Rinnooy Kan, J.K. Lenstra, D.B. Shmoys, The Traveling Salesman Problem, John Wiley & Sons, 1985.

[16] J. Blengio, N. Soca, M. Pedemonte y P. Ezzatti, Estudio preliminar de estrategias híbridas de cómputo CPU, GPU para acelerar algoritmos evolutivos.

[17] S. Nesmachnow, Evolución en el diseño y clasificación de Algoritmos Genéticos Paralelos, Artículo presentado en el Conferencia Latinoamericana de Informática, CLEI 2002.

[18] Evolving Objects (EO): an Evolutionary Computation Framework http://eodev.sourceforge.net/. Disponible el 15/07/2013.

[19] F. Glover, Tabu Search, Part I, ORSA Journal on Computing, First comprehensive description of tabu search, vol. 1, 3, pp. 190-206, 1989.

[20] G. Luque y E. Alba, Parallel Genetic Algorithms, Springer-Verlag Berlin Heidelberg, 2011.

[21] Galib Web Site - http://lancet.mit.edu/galib-2.4, Disponible el 15/07/2013.

[22] H. Rashid, B. C. Novoa, C. A. Qasem, and S. Marcos, An evaluation of parallel knapsack algorithms on multicore architectures. CSC'10, pp. 230–235, 2010.

[23] HCSP, Heterogeneous Computing Scheduling Problem, http://www.fing.edu.uy/inco/grupos/cecal/hpc/HCSP/index.htm, Disponible el 15/07/2013.

[24] I. Castro, Paralelización de algoritmos de optimización basados en poblaciones por medio de GPGPU, Tesis de maestría, instituto tecnológico de la paz, http://posgrado.itlp.edu.mx/uploads/4f335b50b576b.pdf, Disponible el 15/07/2013.

[25] I. Contreras, Y. Jiang, J. Hidalgo, L. Nuñez, Letamendia, Using a GPU,CPU architecture to speed up a GA,based real,time system for trading the stock market, pp. 203-215, 2011.

[26] I. Rechenberg, Evolutions strategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, Frommann–Holzboog, 1973.

[27] I.M. Oliver, D.J. Smith, J.R.C. Holland, Astudy of permutation crossover operators on the TSP, en Genetic Algorithms and Their Applications, Proceedings of the Second International Conference, pp. 224-230, 1987.

[28] IEEE 754-2008, http://grouper.ieee.org/groups/754, Disponible el 15-07-2013.

[29] J. Baker, Adaptive Selection Methods for Genetic Algorithms, 1st International Conference on Genetic Algorithms, pp. 101-111, 1985.

[30] J. Holland, Adaptation in natural and artificial systems, University of Michigan Press, 1975.

[31] J. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems), Cambridge, Mass. : MIT Press, 1992.

[32] J.D. Schaffer and L.J. Eshelman, On Crossover as an Evolutionary Viable Strategy , Proceedings of the 4th International Conference on Genetic Algorithms, pp. 61-68, 1991.

[33] 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, 2008.

[34] J. E. Baker, Reducing bias and ineficiency in the selection algorithm, In Proc. of the 2nd Intl Conf on GA, Lawrence Erlbaum Associates, Inc. Mahwah, pp. 14-21, 1987.

[35] J. Jaros, Multi, GPU Island, Based Genetic Algorithm for Solving the Knapsack Problem, WCCI 2012 IEEE World Congress on Computational Intelligence, pp. 217-224, 2012.

[36] J. C. Russ, Fractal Surfaces, Plenum Press, 1994.

[37] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, Scalable Test Problems for Evolutionary Multi, Objective Optimization, Zurich, Switzerland, Tech. Rep. 112, 2001.

[38] L. Davis, Applying adaptive algorithms to epistatic domains, in Proceedings of the International Joint Conference on Artificial Intelligence, pp. 162-164, 1985.

[39] M. Dorigo y T. Stützle, Ant Colony Optimization, MIT Press, Computational Intelligence Magazine, IEEE, M. Univ. Libre de Bruxelle, Brussels Birattari, M., Stutzle, T. Volume: 1 , Issue: 4 ,Page(s): 28, 39, . 2006.

[40] M. Matsumoto and T. Nishimura, Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation, 8, pp. 3-30, 1998.

[41] M. Pedemonte, E. Alba, F. Luna, Bitwise Operations for GPU Implementation of Genetic Algorithms, Accepted for the workshop on Computational Intelligence on Consumer Games and Graphics Hardware (CIGPU 2011) hold at the Genetic and Evolutionary Computation Conference (GECCO 2011), 2011.

[42] M. Pharr and R. Fernando. GPU Gems 2: Programming Techniques for High,Performance Graphics and General,Purpose Computation. Addison,Wesley Professional, 2005.

[43] M. Stillwell, Studies in evolutionary algorithms, Master's thesis, School of Computer Science and Software Engineering, Monash University, 2001.

[44] M. Nemirovsky, D. M. Tullsen, Multithreading Architecture, Morgan & Claypool, 2013.

[45] M. Krentel, The Complexity of Optimization Problems. Proc. of STOC 86, 1986.

[46] D. Kirk y W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Elsevier Inc, 2010.

[47] M. R. Garey, D. S. Johnson, Computers and intractability, A guide to the theory of NP-completeness W. H. Freeman & Co., 1979.

[48] N. Fujimoto, S. Tsutsui, A Highly, Parallel TSP Solver for a GPU Computing Platform, Proceedings of the 7th international conference on Numerical methods and applications, pp. 264-271, 2011.

[49] N. Soca, J. Blengio, M. Pedemonte y P. Ezzatti, PUGACE, a cellular Evolutionary Algorithm framework on GPUs, Evolutionary Computation (CEC), IEEE Congress on, 2010.

[50] NVIDIA CUDA C Best Practices Guide from CUDA Toolkit 4.0, http://www.nvidia.com, Disponible el 15/07/2013.

[51] NVIDIA Cuda Programing Guide, Sitio Web de NVIDIA, http://www.nvidia.com Disponible el 15/07/2013.

[52] NVIDIA Developer Zone, https://developer.nvidia.com, Disponible el 15/07/2013

[53] NVIDIA Web Site, http://www.nvidia.com, Disponible el 15/07/2013.

[54] O. Maitre, S. Querry, N. Lachiche y P. Collet, EASEA Parallelization of Tree-Based Genetic Programming, 2010 IEEE World Congress on Computational Intelligence, pp. 1997-2004, 2010.

[55] P. Hancock, An Empirical Comparison of Selection Methods in Evolutionary Algorithms, Springer Berlin Heidelberg, pp. 80-94, 1994.

[56] P. Larrañaga, J. A. Lozano, Estimation of Distribution Algorithms, A New Tool for Evolutionary Computation, Kluwer Academic Publishers, 2001.

[57] Paradiseo: A Software Framework for Metaheuristics, http://paradiseo.gforge.inria.fr/, Disponible el 12/07/2013.

[58] Problema del Viajante de Comercio (TSP – Traveling Salesman Problem) http://catarina.udlap.mx/u_dl_a/tales/documentos/lii/martinez_g_ag/capitulo2.pdf, Disponible el 15/07/2013.

[59] P. Musso, F. Dominioni, Proyecto Algoritmos Genéticos Incrementales, Proyecto de Grado, Facultad de Ingeniería UdelaR, 2003.

[60] F. GRAY, Pulse Code Communication, Número de patente: 2632058, Fecha de presentación 13/11/1947, Fecha de emisión 17/03/1953.

[61] R. Dawkins, The Selfish Gene, Oxford University Press, 1976.

[62] S. Nesmachnow, Una Versión Paralela del Algoritmo Evolutivo para Optimización Multiobjetivo NSGA, II y su Aplicación al Diseño de Redes de Comunicaciones Confiables, Actas del X Congreso Argentino de Ciencias de Computación, pp. 1933-1944, 2004.

[63] S. Potti, S. Pothiraj, GPGPU Implementation of Parallel Memetic Algorithm for VLSI Floorplanning Problem, Engineering and Information Technology, Communications in Computer and Information Science, Volume 204, 1, pp. 432-441, 2011.

[64] H. P. Schwefel, Numerical optimization of computer models, John Wiley & Sons, 1981.

[65] Sitio Web Accelereyes, http://www.accelereyes.com/, Disponible 15/07/2013.

[66] Sitio Web de Mallba, http://neo.lcc.uma.es/mallba/easy-mallba, Disponible el 15/07/2013.

[67] T. Feo y M. Resende, Greedy Randomized Adaptive Search Procedures, Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial, No.19, pp. 61-76, 2003.

[68] M. Tang, X. Yao, A Memetic Algorithm for VLSI Floorplanning, IEEE Trans, On Systems, Man, and Cybernetics 37, pp. 62–68, 2007.

[69] Tuning CUDA applications for Fermi, Application Note, 2011.

[70] V. Abascal Pelayo, P. Feijoo Ugalde, Implementación de Algoritmos Genéticos sobre la plataforma de desarrollo paralelo CUDA, Facultad de Informática Universidad Complutense de Madrid, http://eprints.ucm.es/9514/1/TC2009-33.pdf, Disponible 15/07/2013.

[71] W. Banzhaf, The “molecular" traveling salesman, Biological Cybernetics, Volume 64, 1, pp. 7-14, 1990.

[72] W. Banzhaf y R. Colin, Foundations of Genetic Algorithms 5, M. Kaufmann, 1999.

[73] Whitepaper, NVIDIA’s Next Generation CUDA Compute Architecture: Fermi.

[74] Whitepaper, NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, The Fastest, Most Efficient HPC Architecture Ever Built.

[75] W. Stallings, Computer Organization and Architecture 5th Edition, Prentice Hall, 2000.

[76] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Springer, Verlag, Berlin Heidelberg, 1992.

[77] E. Zitzler, K. Deb, L. Thieler, Comparison of multiobjective evolutionary algorithms: Empirical results. IEEE Trans. on Evol. Computation 8, pp. 173-195, 2000.