PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

# IMPLEMENTACIÓN Y EVALUACIÓN DE UN ALGORITMO ABC EN GPU

MARIO ANDRÉS HERRERA LETELIER
MARCELO ALEJANDRO SALAS QUEZADA

INFORME FINAL DE PROYECTO II
PARA OPTAR AL TÍTULO DE
INGENIERO DE EJECUCIÓN EN INFORMÁTICA

MARZO 2014

PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

# IMPLEMENTACIÓN Y EVALUACIÓN DE UN ALGORITMO ABC EN GPU

MARIO ANDRÉS HERRERA LETELIER
MARCELO ALEJANDRO SALAS QUEZADA

**Profesor Guía:** Wenceslao Palma Muñoz
**Profesor Co-referente:** Ricardo Soto de Giorgis

**Carrera:** Ingeniería de Ejecución en Informática

MARZO 2014

# Índice

# Abstract

Today, the graphic processor units, also known as GPUs, does count with great calculus capacity, dedicated to the textures generation on graphic design applications or sizeable industries, such as the video games. Because of the inner nature of this co-processors, born the idea to use them like a support unit to Central Processor Unit (CPU). This create the GPGPU concept (an English acronym to General-Purpose Computing on Graphics Processing Units), which means to take the great GPU's processing power to be used in different application types. In other words, power up applications designed to work with CPU's by using GPU's support.

This paper, is related to application's execution acceleration which require a great processing power, specifically complex mathematical problems, such as the optimization problems, that may be resolved using Metaherustics. Here will be implemented an ABC Algorithm to resolve the Set Covering Problem.

# Resumen

En la actualidad, las unidades de procesamiento gráfico o GPU (acrónimo del inglés graphics processing unit) cuentan con una gran capacidad de cálculo dedicado a la generación de texturas en aplicaciones de diseño gráfico o en otros casos, aplicaciones en la industria del entretenimiento como son los video juegos. Debido a la naturaleza intrínseca de estos coprocesadores, surge la idea de utilizarlos como unidad de apoyo a la unidad central de procesamiento o CPU, por esto nace un concepto llamado GPGPU (acrónimo del inglés General-Purpose Computing on Graphics Processing Units) que busca aprovechar la gran capacidad de procesamiento de estos coprocesadores para aplicaciones de distinta naturaleza para los cuales fueron concebidos, en otras palabras, potenciar las aplicaciones diseñadas para trabajar en CPU mediante la cooperación de un GPU.

El presente trabajo, trata el tema de acelerar la ejecución de aplicaciones que requieren una gran cantidad de procesamiento, específicamente problemas matemáticos de gran complejidad, como son los problemas de optimización, los cuales pueden ser resueltos utilizando Metaheuristicas. En particular, se implementará y evaluará un algoritmo ABC para resolver el problema del Set Covering.

# Lista de Figuras

# Lista de Tablas

# 1 Introducción

La Unidad de Procesamiento Grafico está diseñada para realizar computación aritméticamente intensiva, con un alto paralelismo, ya que es lo se necesita para la representación de gráficos, por este motivo en las tarjetas gráficas se pueden encontrar muchos más transistores, que están dedicados al procesamiento de datos en lugar del control de flujo y de almacenar los datos en caché como es el caso de los CPU tradicionales [1].

Si se consideran las características generales anteriormente mencionadas, utilizar la GPU de una tarjeta gráfica como una unidad de apoyo a la CPU, hace que sea una alternativa atractiva de ser considerada para mejorar el rendimiento de variados tipos de algoritmos, ya que a grandes rasgos, los algoritmos que se van a ejecutar en un GPU son versiones paralelas de los que se van a ejecutar en un CPU.

Como aplicaciones que comúnmente hacen uso del GPU para acelerar los tiempos de cómputo se pueden encontrar programas relacionados con el procesamiento de imágenes como también de datos multimedia, así como la codificación y descodificación de videos y audios, y últimamente para la compresión de archivos de gran tamaño. Sin embargo se pueden encontrar un sin número de algoritmos que son susceptibles de paralelizar, según sea el problema que se quiere solucionar.

Este trabajo desarrolla el tema de computación paralela utilizando la GPU aplicada a la resolución de problemas de optimización muy complejos. Para la resolución de estos problemas se hará uso de la metaheurística Artificial Bee Colony Algorithm [2], basada en el comportamiento que realizan las abejas en su tarea de recolectar alimentos para la colonia.

El problema a resolver usando ABC será el Set-Covering Problem [3][4]. Este se resolverá mediante distintas estrategias de paralelismo que permitan obtener el máximo rendimiento en GPU.

# 2 Objetivos y metodología

## 2.1 Objetivo General

Diseño e implementación de un algoritmo paralelo utilizando CUDA basado en Artificial Bee Colony (ABC) Algorithm para la resolución del Set-Covering Problem (SCP) con el fin de obtener ganancias en términos de "tiempo de ejecución" manteniendo la calidad de la solución.

## 2.2 Objetivos Específicos

- Conocer y comprender las ventajas de utilizar un GPU como unidad de apoyo al CPU.
- Entender y aplicar el modelo de programación en GPU de NVIDA llamado CUDA.
- Comprender y resolver el Set-Covering Problem.
- La comprensión de la metaheurística ABC Algorithm.
- El análisis y diseño de un algoritmo en serie y paralelo para la resolución del Set-Covering Problem.
- Ejecutar pruebas de rendimiento a instancias del SCP para medir el rendimiento de los distintos algoritmos del ABC Algorithm.

## 2.3 Metodología

Como metodología para el presente trabajo se ha optado por un enfoque iterativo, otorgándole inicialmente mayor énfasis a las etapas de investigación y diseño para luego volcarse gradualmente en la implementación.

El desarrollo de este proyecto se llevará a cabo de la siguiente manera:

1. Investigación general del modelo de programación CUDA.
2. Investigación y análisis al problema del Set-Covering.
3. Investigación y análisis de ABC Algorithm.
4. Elaboración de una versión serial del ABC Algorithm para la resolución del SCP.
5. Elaboración de variadas versiones del ABC Algorithm para la resolución del SCP utilizando el modelo de programación CUDA.
6. Ejecución de pruebas de rendimiento a las distintas versiones del algoritmo y obtener conclusiones.

# 3 Arquitectura GPU y el modelo de programación CUDA

En el año 2006 NVIDIA introduce al mercado CUDA [5], una plataforma y un modelo de programación para GPUs que permite que estas sean utilizadas para propósitos distintos a la generación de gráficos [6]. El principal motivo que llevó a NVIDIA a desarrollar CUDA fue que al principio utilizar un GPU para otros propósitos requería que sus potenciales programadores conocieran detalles demasiado técnicos, generalmente los relacionados con la *tubería de rendereo*[1], el cual es el método en que se basan las tarjetas gráficas para la elaboración de texturas en las imágenes, además de APIs como OpenGL o DirectX.

CUDA se basa en un compilador y un conjunto de herramientas que permiten a los desarrolladores hacer uso de una variación del lenguaje C para codificar algoritmos que originalmente se encontraban desarrollados para CPUs, de esta manera se logra reducir la curva de aprendizaje con el objetivo de mejorar el rendimiento en aplicaciones de alta carga de procesamiento.

Dentro de CUDA, hay dos conceptos que hay que mencionar los cuales son:

- *Host*: Es el computador que contiene a la tarjeta gráfica con que se va a ejecutar los distintos algoritmos, además de controlar el funcionamiento.
- *Device*: Corresponde a la tarjeta gráfica alojada en el Host, que en este caso, corresponde a una tarjeta gráfica del fabricante NVIDIA.

[1] Proceso en el cual una imagen descrita en un formato vectorial se convierte en un conjunto de pixeles y es mostrada en un medio de salida digital como una pantalla.

## 3.1 Arquitectura GPU

El modelo de programación de NVIDIA llamado CUDA (de Compute Unified Device Architecture) busca como se mencionó anteriormente, explotar todo el potencial de las tarjetas gráficas NVIDIA en los distintos segmentos del mercado, ya sea en la serie *GeForce* para juegos de video, *Quadro* para edición y generación de gráficos en 3D profesionales y *TESLA* para computo de alta precisión como en áreas de desarrollo científico.

CUDA depende de la arquitectura del GPU en el cual se estén ejecutando los programas, por lo cual, se detallaran a continuación los componentes a nivel de hardware más importantes.

## 3.2 Kepler

Los GPUs actuales de NVIDIA están fabricados utilizando la arquitectura Kepler, que busca ser una mejora notable en comparación  su antecesora Fermi, a nivel de consumo energético y rendimiento.

Un GPU Kepler está compuesto de diferentes *Graphics Processing Clusters* (GPCs), *Streaming Multiprocessors* (SMs) y distintos controladores de memoria. La serie Geforce está formada de cuatro GPCs, ocho *next-generation Streaming Multiprocessors* (SMX) y cuatro controladores de memoria, como se puede observar en la Figura:

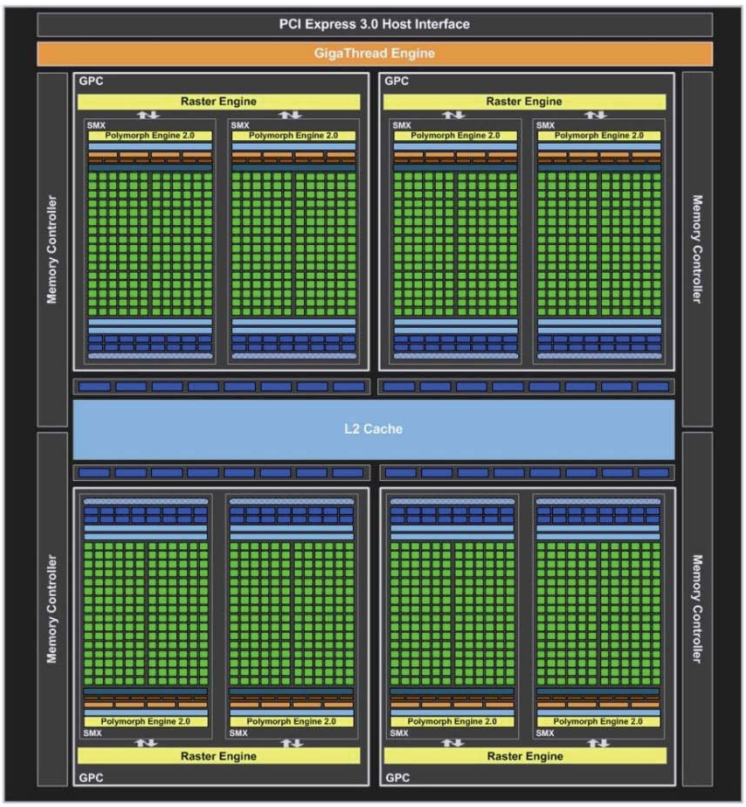



Figura 3.1 GPU Arquitectura Kepler

### 3.2.1 SMX

Un SMX es el corazón de los GPU de NVIDIA en la arquitectura Kepler. La mayoría de los componentes clave de hardware que permiten el funcionamiento de CUDA residen en él.

Al interior de cada SMX residen 192 CUDA *Cores*, los cuales se encargan de realizar los cálculos aritméticos. Además se encuentran unidades *load/store* que se encargan de cargar y guardar los resultados de las operaciones de los CUDA Cores en memoria. En la Figura 3.2 se pueden observar los componentes antes mencionados:

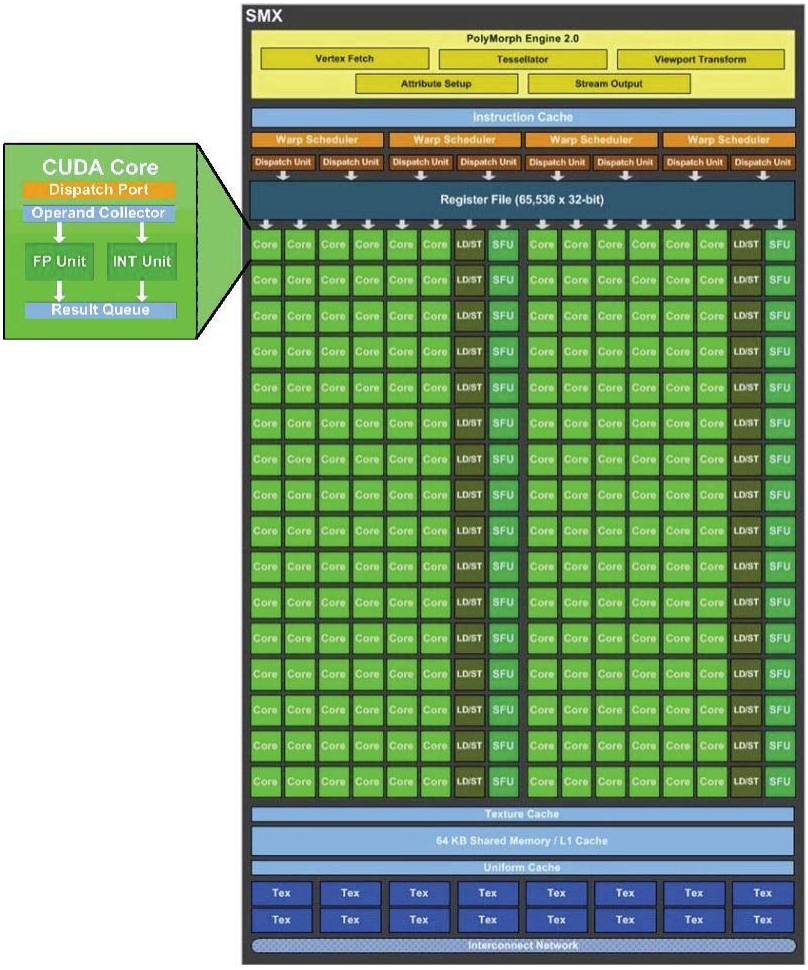



Figura 3.2 Diagrama SMX

## 3.3 Modelo de Programación

Aprovechando las ventajas de un lenguaje de alto nivel como C, CUDA permite al programador desarrollar funciones las cuales serán ejecutadas en paralelo en el *device*, estas funciones reciben el nombre de *Kernel*, las cuales son ejecutadas N veces en paralelo para N *Threads* (Hebras) diferentes. En el modelo de programación de CUDA se definen los siguientes conceptos:

- *Threads*: Una hebra es la unidad básica de todo *kernel*. Cada una de las hebras que son ejecutadas en un *kernel* realiza trabajo en uno de los elementos de un objeto, por ejemplo, en un arreglo, cada hebra ejecutaría la tarea en cada elemento de ese arreglo.
- *Blocks*: Los Bloques corresponden a una agrupación de hebras que se pueden comunicar entre sí a través de la memoria compartida.
- *Warps:* Es la unidad mínima de planificación y está formada por 32 hebras.
- *Grid:* Es la división inicial que se aplica sobre el objeto. Cada *grid* está formada por bloques. Todas las hebras de un *grid* se comunican entre sí por medio de una memoria global.

### 3.3.1 Kernel

CUDA al ser una extensión de C permite al programador utilizar la sintaxis del lenguaje agregando aspectos específicos como son los *Kernel*.

Los *Kernel* son funciones que al ser llamadas son ejecutadas N veces en paralelo por N diferentes hebras. Estas funciones son invocadas desde el host, pero son ejecutadas en el *device*.

Al definir un Kernel se debe hacer utilizando la declaración `__global__` [7] y además se debe especificar el número de hebras a ejecutar utilizando una *configuración de ejecución*, que se utiliza asignando parámetros al operador especial <<<…>>>. Como primer parámetro corresponde a la organización y tamaño del *grid* de bloques y como segundo parámetro corresponde al tamaño y la organización (número de hebras) que va a tener cada bloque. El tipo de retorno de los kernel debe ser obligatoriamente *void*.

### 3.3.2 Jerarquía de hebras

En CUDA cada hebra puede ser identificada utilizando la variable `threadIdx`, la cual corresponde a un vector de tres componentes (x, y, z), el cual puede representar una hebra en un bloque de una, dos o tres dimensiones. Esto permite que los elementos sean relacionados de manera natural como en el caso de un vector, matriz o para representar datos con volúmenes.

Las hebras dentro de un bloque pueden cooperar entre ellas compartiendo datos a través de la memoria compartida y sincronizándose a través de la ejecución coordinada de accesos a memoria.

Para la eficiencia de la cooperación, se cuenta con que la memoria compartida es de una baja latencia cercana al núcleo del GPU y se espera que todas las hebras de un bloque residan en el mismo núcleo.

Por restricciones de la arquitectura del GPU, el número de hebras por bloque está limitado, ya que se espera que todas las hebras se encuentren en el núcleo del GPU lo cual implica que dependan de los recursos limitados por hardware. El número máximo de hebras por bloque dependen de la arquitectura del GPU de NVIDIA que ejecuta el código. En la última arquitectura lanzada por NVIDIA (y la utilizada en esta investigación) llamada *Kepler* se pueden ejecutar un máximo de 1024 hebras. Sin embargo, para hacer frente a esta restricción, un kernel puede ser ejecutado por múltiples bloques de hilos de igual forma. De esta manera, el número total de hilos en ejecución será igual al número de bloques multiplicado por el número de hilos de cada bloque.

Los bloques son organizados en grids de una, dos o tres dimensiones al igual que las hebras dentro de un bloque, en donde cada bloque del grid está identificado por la variable dentro del kernel llamada `blockIdx`. También hay una variable llamada `blockDim` la cual indica la dimensión de un bloque. Para acceder a una hebra dentro del grid se debe utilizar la siguiente metodología:

- Para un bloque unidimensional, el ID de la hebra es el mismo ID.
- Para un bloque bidimensional de tamaño (Dx, Dy), el ID de la hebra de índice (x, y) es (x + y Dx).
- Para un bloque tridimensional de tamaño (Dx, Dy, Dz), el ID de la hebra con índice (x, y, z) es (x + y Dx + z Dx Dy).

Otra consideración que hay que tener en cuenta, es que como se mencionó anteriormente existe otra subdivisión de las hebras en cada bloque llamadas *Warps*. Cada warp está formado por 32 hebras, en donde varios warps pueden residir en cada multiprocesador (SMX), pero solo uno está en ejecución en cada momento.

### 3.3.3 Jerarquía de memoria

En lo que respecta a memorias, CUDA cuenta con distintos tipos que pueden ser utilizadas según sea la necesidad y la naturaleza de la aplicación que se desee implementar.

Cada hebra tiene un espacio de memoria privada llamada *Local Memory* (Memoria Local). También existe un tipo de memoria a nivel de bloques llamada *Shared Memory* (Memoria Compartida) la cual es visible para todas las hebras del bloque y que tiene el mismo tiempo de vida definido que el bloque especifico al cual pertenece. Además existe una memoria común para todas las hebras llamadas *Global Memory* (Memoria Global).

#### 3.3.3.1 Memoria Local

Los accesos a memoria local solo ocurren para algunas variables automáticas. El espacio de memoria local no tiene caché, por lo que es más importante aún el modelo de acceso correcto para maximizar el ancho de banda, dado lo costoso que son los accesos a la memoria del dispositivo.

#### 3.3.3.2 Memoria Compartida

Como se explicó anteriormente, se trata de una memoria común a cada bloque de hilos. Está construida *on-chip*, con lo cual un acceso a ella es mucho más rápido que uno a memoria global. De hecho, para todos los hilos de un warp, acceder a la memoria compartida es tan rápido como acceder a un registro, siempre y cuando no haya conflictos

de bancos entre las hebras. Para obtener un mayor ancho de banda, la memoria compartida de cada SMx se encuentra dividida en módulos de memoria de igual tamaño llamados bancos, que pueden ser accedidos simultáneamente. De esta forma, la memoria puede atender con éxito un pedido de lectura o escritura constituido por *n* direcciones que pertenezcan a *n* bancos distintos. Así se consigue un ancho de banda *n* veces mayor al de un banco simple. Ahora bien, si dos direcciones de un pedido caen en el mismo banco de memoria, hay un conflicto y el acceso tiene que ser serializado. El hardware entonces divide el acceso a memoria en tantos accesos libres de conflicto como sean necesarios, perdiendo ancho de banda. Para conseguir el máximo rendimiento, es importante conocer la organización de los bancos de memoria, y cómo se organiza el espacio de direcciones. Así se podrán programar los accesos a memoria compartida de manera que se minimicen los conflictos.

#### 3.3.3.3 Memoria de Acceso Global

Dentro de las memorias de acceso global existen tres espacios de memorias los cuales son la *memoria global, memoria de texturas* y la *memoria de constantes*, las cuales son accesibles por todas las hebras. La memoria de texturas y de constantes son espacios de solo lectura.

La memoria global, constantes y texturas son espacios de memorias optimizados para múltiples propósitos y que son persistentes a través de los múltiples kernel lanzados en la misma aplicación.

El espacio de memoria global no es cacheado. El espacio de constantes está cacheado, el valor solicitado se lee desde la caché de constantes y sólo se lee la memoria global en un fallo de caché. Leer de la caché de constantes puede ser tan rápido como leer de un registro, siempre que todas las hebras de un warp lean la misma dirección. El espacio de texturas, al igual que el de constantes, está cacheado, pero con la diferencia de que está optimizado para localidad espacial 2D. De modo que las hebras pertenecientes a un mismo warp que lean direcciones que estén cercanas conseguirán un mejor rendimiento. Además tiene un tiempo de latencia constante, independientemente de si el dato se lee de caché o de memoria; la única diferencia es que reduce el ancho de banda utilizado con la memoria. De esta manera, resulta ventajoso leer datos del espacio de texturas en lugar de hacerlo del espacio de memoria global o de constantes.

## 3.4 Modelo de Ejecución

La arquitectura del GPU de NVIDIA está construida alrededor de un arreglo escalable de *next-generation Streaming Multiprocessors* (SMx). Cuando un programa CUDA en el CPU del host invoca un grid de kernels, los bloques del grid se enumeran y se distribuyen en multiprocesadores (SMx) con capacidad de ejecución disponible. Las hebras de un bloque se ejecutan concurrentemente en un multiprocesador. Cuando los bloques de hebras terminan, nuevos bloques son lanzados en los multiprocesadores libres. Un multiprocesador consiste en 192 núcleos de procesadores escalables (SP, también llamados CUDA Cores) [8] y un chip de memoria compartida. Los multiprocesadores crean, manejan y ejecutan las hebras concurrentes en hardware sin ninguna planificación anterior.

A nivel de ejecución, se puede establecer una sincronización de las hebras utilizando `__syncthreads()__` como una simple instrucción. Esto proporciona sincronización rápida por barrera con baja carga de creación de hilos y ninguna planificación anterior.

### 3.4.1 SIMT

Para manejar cientos de hilos corriendo distintos programas, el multiprocesador emplea la arquitectura llamada SIMT (Single Instruction, MultipleThread (Instrucción Simple, Hebras Múltiples)). El multiprocesador asigna cada hebra a un núcleo escalar de procesador, y cada escalar ejecuta una hebra independientemente con sus propias direcciones de instrucción y registro de estado. La unidad de multiprocesador SIMT crea, maneja, planifica y ejecuta en grupos de 32 hebras paralelas llamados warps. Hebras individuales componiendo un SIMT warp empiezan en la misma dirección de programa a la vez pero son libres en todo lo demás de bifurcarse y ejecutarse libremente.

Cuando un multiprocesador recibe uno o más bloques de hebras que ejecutar, los divide en warps que son planificados por la unidad del SIMT. El modo en que se divide un bloque en warps es siempre el mismo; cada warp contiene hebras consecutivas, incrementando las IDs de las hebras con el primer warp conteniendo la hebra 0.

Cada momento de lanzamiento de instrucciones, la unidad SIMT selecciona un warp que esté listo para ser ejecutado, lanza la siguiente instrucción del grupo de hebras del warp activo y planifica la siguiente instrucción. Un warp ejecuta una instrucción común a la vez, así que la eficiencia total se realiza cuando las 32 hebras de un warp convergen en el mismo camino de ejecución. Si las hebras de un warp divergen, el warp ejecuta en serie cada rama tomada, deshabilitando las hebras que no están en esa rama, y cuando se completan todas las ramas, las hebras convergen de nuevo en el mismo camino de ejecución. Las divergencias de camino solo ocurren dentro de un warp, diferentes warps ejecutan independientemente sin importar si están ejecutando flujos de códigos comunes o distintos.

La arquitectura SIMT es semejante a la organización vectorial SIMD (Single Instruction, Multiple Data (Instrucción Simple, Datos Múltiples)) en una única instrucción que controla múltiples elementos a procesar. La diferencia clave es que la organización vectorial SIMD expone el ancho de SIMD al software, mientras que las instrucciones SIMT especifican el comportamiento de ejecución y ramificación de una única hebra. En contraste con máquinas vectoriales SIMD, SIMT permite a los programadores escribir código paralelo a nivel de hebras para hebras independientes, escalares, tanto como código de datos paralelos para hebras coordinadas.

Con el propósito de la corrección, los programadores pueden esencialmente ignorar el comportamiento del SIMT, aunque se pueden dar mejoras substanciales cuando se cuida que las hebras de un warp no diverjan en su camino.

La cantidad de bloques que puede procesar un multiprocesador de una vez depende de la cantidad de memoria compartida que se necesite. Si no hay suficientes registros o memoria compartida por multiprocesador para procesar como mínimo un bloque, el kernel fallará en su ejecución. Un SMx puede ejecutar hasta 8 hebras de un bloque de forma concurrente.

# 4 Set-Covering

El *Set-Covering Problem* es un problema clásico el cual consiste en encontrar un conjunto de soluciones que permitan cubrir un conjunto de necesidades al menor costo posible de acuerdo a un conjunto de restricciones. Un conjunto de necesidades corresponde a las filas, y el conjunto solución es la selección de columnas que cubren en forma óptima al conjunto de filas. Existen muchas aplicaciones de este tipo de problemas, siendo las principales la localización de servicios, selección de archivos en un banco de datos, simplificación de expresiones booleanas, balanceo de líneas de producción, entre otros [16][17].

El problema del Set-Covering se puede definir como sigue:

Sea:

| | |
|---|---|
| $A = (a_{ij})$ | Una matriz binaria (0,1) de dimensiones $m\ x\ n$ |
| $C = (c_j)$ | Un vector *n* dimensional de enteros |
| $M = \{1, \ldots, m\}$ | Conjunto de filas |
| $N = \{1, \ldots, n\}$ | Conjunto de columnas |

El valor $c_j (j \in N)$ representa el costo de la columna $j$, y podemos asumir que, $c_j > 0$ para $j \in N$. Así, decimos que una columna $j \in N$ cubre una fila $i \in M$ si $a_{ij} = 1$. El SCP busca un subconjunto de columnas de costo mínimo $S \subseteq N$, tal que cada fila $i \in M$está cubierta por al menos una columna $j \in S$.

Así, el problema de Set-Covering se puede plantear de la siguiente manera:

$$mín(z) = \sum_{j=1}^{n} c_j x_j$$

Sujeto a:

$$\sum_{j=1}^{n} a_{ij} x_j \geq 1 \qquad \forall_i \in M$$

$$x_j = \begin{cases} 1 & si\ j \in S \\ 0 & sino \end{cases} \qquad \forall_j \in N$$

# 5 Artificial Bee Colony Algorithm

El Artificial Bee Colony Algorithm es una metaheurística *population-based* de enjambres basada en el comportamiento de las abejas en su proceso de búsqueda y selección de fuentes de alimento, que se utiliza para la resolución de problemas complejos de optimización.

El ABC Algorithm está compuesto por tres grupos de abejas que representan a su vez los comportamientos: las abejas empleadas o abejas recolectoras que están asociadas con fuentes específicas de alimento, las abejas observadoras las cuales como su nombre lo dice "observan" como las abejas empleadas realizan la *Danza de las Abejas* que les entrega la información de las fuentes de alimentos y las abejas exploradoras que se dedican a buscar al azar nuevas fuentes de alimentos. Las abejas observadoras y exploradoras también son llamadas abejas desempleadas. Inicialmente, todas las fuentes de alimento son descubiertas por abejas exploradoras. A partir de entonces, el néctar de las fuentes de alimento es explotado por las abejas empleadas y por las abejas observadoras, y hasta última instancia, hacer que se agoten. Entonces, la abeja empleada que explota la fuente de alimento una vez que esta se agota se convierte en una abeja exploradora en busca de nuevas fuentes de alimento, una vez más. En otras palabras, la abeja propia cuya fuente de alimento se ha agotado se convierte en una abeja exploradora. En ABC Algorithm, la posición de una fuente de alimento representa una posible solución al problema y la cantidad de néctar de una fuente de alimento corresponde a la calidad (*fitness*) de la solución asociada. El número de abejas empleadas es igual al número de fuentes de alimentos (soluciones) ya que cada abeja empleada está asociada con una y sólo una fuente de alimento.

## 5.1 Versión Binaria del ABC Algorithm

De acuerdo a la literatura revisada en [2] y [8], la versión básica del ABC Algorithm permite resolver problemas de optimización en el espacio continuo de soluciones. Si la solución se estructura de manera binaria, la versión básica del ABC Algorithm debe ser modificada para resolver este tipo de problemas. Este informe, se basará en la versión de [9], llamada disABC, la cual mediante una modificación a los comportamientos definidos en [2], permite utilizar la representación binaria en el espacio de soluciones discretas.

## 5.2 Algoritmo General

Un esquema general del ABC Algorithm se puede definir como sigue:

```
ABC Algorithm
Inicio
 Fase de Inicialización
 Repetir
       Fase de Abejas Empleadas
       Fase de Abejas Observadoras
       Fase de Abejas Exploradoras
       Memorizar la mejor solución lograda hasta ahora
 Hasta (Ciclo = Máximo número de ciclos o tiempo máximo de CPU)
Fin
```

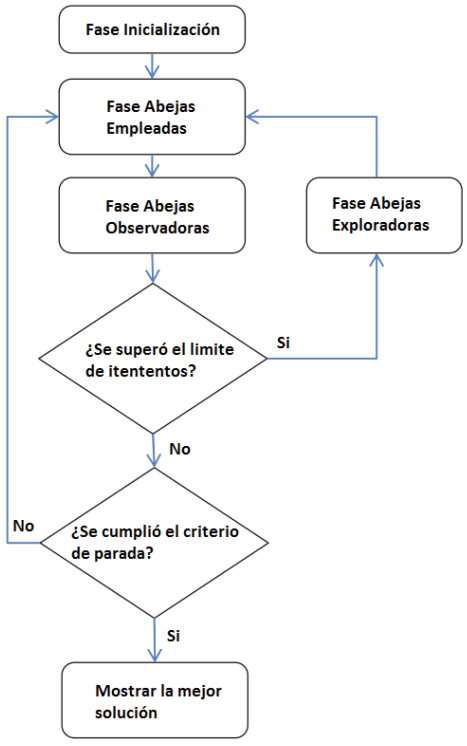



Figura 5.1 Diagrama de Flujo de ABC Algortihm.

A continuación se detallarán cada una de las fases que participan en el ABC Algorithm.

### 5.2.1 Fase de Inicialización

En la *fase de inicialización* todos los vectores de la población de fuentes de alimento, ($X_i$) son inicializados ( $i = 1 \dots SN, SN: tamaño\ de\ la\ poblacion$ ) por las abejas exploradoras y los parámetros del ABC Algorithm son establecidos. Cada fuente de alimento, ($X_i$), es un vector solución para el problema de optimización, donde cada vector ($X_i$) está formado por *n* variables, ( $X_{ij}, j = 1 \dots n$ ), las cuales deben ser optimizadas a fin de minimizar la función objetivo.

Para cada bit del vector solución, se genera un numero aleatorio $r_{ij}$ en el rango [0,1]. Si el numero generado es menor que 0.5, entonces el bit toma el valor de 0, en otro caso un 1.

La definición anterior se formula como sigue:

$$X_{mi} = \begin{cases} 0\ si\ r_{ij} < 0.5 \\ 1\ en\ otro\ caso \end{cases} \qquad (1)$$

### 5.2.2 Fase de Abejas Empleadas

Las abejas empleadas buscan nuevas fuentes de alimento ($V_i$) en el vecindario que tengan más néctar que la fuente de alimento ($X_i$) en su memoria. Encontrando una fuente de alimento en el vecindario se evalúa su calidad (fitness).

Para generar una nueva solución, se utilizará el método definido en [9], donde se evalúa una *medida de Similitud* entre las distintas fuentes de alimento, con el fin de medir que tan distantes se encuentran una de otra. Para realizar esto, se utiliza la siguiente definición:

$$Disimilitud\ (X_i, X_k) = 1 - Similitud\ (X_i, X_k) = 1 - \frac{M_{11}}{M_{01}+M_{10}+M_{11}} \qquad (2)$$

Donde $X_i$ es la $i^{esima}$ abeja empleada y $X_k$ es la abeja vecina seleccionada de la población de abejas. $M_{01}$ , $M_{10}$ y $M_{11}$ se define como sigue:

- $M_{01}$ representa el número total de bits donde $X_{ij}$ = 0 y $X_{kj}$ = 1, donde $j = 1,2, \dots, D$ y D es la dimensión del problema.

$$M_{01} = \sum_{j=1}^{D} F\left(X_{ij} = 0,\ X_{kj} = 1\right)$$

• $M_{10}$ representa el número total de bits donde $X_{ij}$ = 1 y $X_{kj}$ = 0, donde $j = 1,2,\dots,D$ y D es la dimensión del problema.

$$M_{10} = \sum_{j=1}^{D} F(X_{ij} = 1,\ X_{kj} = 0)$$

• $M_{11}$ representa el número total de bits donde $X_{ij}$ = 1 y $X_{kj}$ = 1, donde $j = 1,2,\dots,D$ y D es la dimensión del problema.

$$M_{11} = \sum_{j=1}^{D} F\left(X_{ij} = 1,\ X_{kj} = 1\right)$$

Como se mencionó anteriormente, la versión del ABC Algorithm está formulada para trabajar en el espacio continuo de soluciones, siendo necesario modificar la ecuación para el cálculo de vecindario para hacerla compatible con el espacio de búsqueda continuo. Según [2], para calcular una fuente de alimento en el vecindario se debe utilizar $V_{ij} = X_{ij} + \Phi_{ij}(X_{ij} - X_{kj})$, donde $V_{ij}$ es la nueva fuente de alimento, $X_{ij}$ es la fuente de alimento para calcular la vecina, $X_k$ es una fuente de alimento seleccionada aleatoriamente, $i$ es un parámetro elegido aleatoriamente y $\Phi_{ij}$ es un número aleatorio en el rango [-$a$, $a$].

La ecuación de cálculo de vecindario debe ser modificada de la siguiente manera:

$$\textit{Disimilitud}\ (V_i, X_i) \approx \vartheta\ \textit{x Disimilitud}\ (X_i, X_k) \qquad (3)$$

Donde $V_i$ es la solución candidata para $X_i$, $\vartheta$ es un factor de escalada positivo, y ≈ es un operador de semejanza. Para construir $V_i$, se deben determinar tres variables que se figuran a continuación:

• $M_{01}$ representa el número total de bits donde $V_{ij}$ = 0 y $X_{ij}$ = 1, donde $j = 1,2,\dots,D$ y D es la dimensión del problema.

$$M_{01} = \sum_{j=1}^{D} F\left(V_{ij} = 0,\ X_{ij} = 1\right)$$

• $M_{10}$ representa el número total de bits donde $V_{ij}$ = 1 y $X_{ij}$ = 0, donde $j = 1,2,\dots,D$ y D es la dimensión del problema.

$$M_{10} = \sum_{j=1}^{D} F(V_{ij} = 1,\ X_{ij} = 0)$$

• $M_{11}$ representa el número total de bits donde $V_{ij}$ = 1 y $X_{ij}$ = 1, donde $j = 1,2,\dots,D$ y D es la dimensión del problema.

$$M_{11} = \sum_{j=1}^{D} F\left(V_{ij} = 1,\ X_{ij} = 1\right)$$

Además, se debe desarrollar:

$$\min \left| \left(1 - \frac{M_{11}}{M_{01} + M_{10} + M_{11}}\right) - \vartheta \, x \, \left(1 - \frac{M_{11}}{M_{01} + M_{10} + M_{11}}\right) \right| \qquad (4)$$

Sujeto a:

$$M_{11} + M_{01} = n_1$$

$$M_{01} \leq n_0$$

$$M_{01}, M_{10}, M_{11} \geq 0 \text{ y entero.}$$

Aquí, $n_1$ y $n_0$ son números con valores de 1 y 0 en el vector vinario $X_i$ respectivamente. Para determinar los valores de $M_{01}$ , $M_{10}$ y $M_{11}$, se debe resolver (4).

Luego de resolver (4) y obtener los valores óptimos de $M_{01}$ , $M_{10}$ y $M_{11}$ , la solución candidata $V_i$ puede ser obtenida obteniendo el *generador de soluciones binarias* (NBSG) definido en [9]. Los pasos del NBSG son los siguientes:

- Paso 1: Calcular el valor de A por medio de $A = Dissimilarity\ (X_i, X_k)$ y úselo en el modelo de programación matemático (4) para obtener $M_{01}$ , $M_{10}$ y $M_{11}$. Inicialice el vector binario $V_i$ con ceros.
- Paso 2.1: (Fase de Herencia) Seleccione $M_{11}$ bits aleatoriamente de $V_i$ y cambie los 0 por 1.
- Paso 2.2: (Fase de Desheredamiento) Seleccione $M_{10}$ bits aleatoriamente de $V_i$ y cambie los 0 por 1.

Después de producir la nueva fuente de comida ($V_i$) se calcula su fitness y se realiza una comparación entre ($V_i$) y ($X_i$). Para calcular el valor del fitness de la solución $Fit_i(V_i)$, se utilizó la siguiente fórmula de [2]:

$$Fit_i(V_i) = \frac{1}{1 + f_i(X_i)} \qquad (5)$$

Donde $f_i(X_i)$ es el valor solución de la función objetivo ($X_i$).

### 5.2.3 Fase de Abejas Observadoras

Las abejas desempleadas se componen de dos grupos: las observadoras y las exploradoras. Las abejas empleadas comparten su información con las abejas observadoras que esperan en el nido y entonces las observadoras escogen probabilísticamente sus fuentes de alimento dependiendo de esta información. En ABC, una abeja observadora escoge una fuente de alimento dependiendo de los valores probabilísticos calculados usando los valores del fitness provistos por las abejas empleadas. Para este propósito, se puede utilizar un fitness basado en técnicas de selección, tales como lo son el método de selección tipo ruleta.

El valor probabilístico $P_i$ con el cual $X_i$ es escogido por una abeja observadora puede ser calculado usando la expresión dada en la ecuación de [2]:

$$P_i = \frac{fit_m(X_i)}{\sum_{i=1}^{SN} fit_i(X_i)} \qquad (6)$$

Después de que una fuente de comida $X_i$ para una abeja observadora se elige probabilísticamente, una fuente vecina $V_i$ utilizando (2) y se calcula su valor de fitness utilizando (5). Al igual que en la fase de abejas empleadas, se realiza una comparación entre $V_i$ y $X_i$. Entonces, mas abejas observadoras son reclutadas a fuentes más ricas, y surge un comportamiento de retroalimentación positivo.

### 5.2.4 Fase de Abejas Exploradoras

Las abejas desempleadas quienes escogen sus fuentes de alimento aleatoriamente se llaman “exploradoras” (Scout). Estas son las abejas empleadas cuyas soluciones no pueden ser mejoradas ante un predeterminado número de intentos. Especificadas por el usuario de ABC Algorithm y llamado “limite” o “criterio de abandono”, se vuelven exploradoras y sus soluciones son abandonadas. Entonces, las exploradoras convertidas empiezan a registrar nuevas soluciones, aleatoriamente. Por ejemplo, si la solución $X_i$ ha sido abandonada, la nueva solución descubierta por la exploradora quien fue la abeja empleada de $X_i$ puede ser definida por (1). Entonces aquellas fuentes las cuales inicialmente eran pobres o fueron hechas pobres por la exploración son abandonadas y un comportamiento de retroalimentación negativo surge para balancear la retroalimentación positiva.

### 5.2.5 Reparación de Soluciones

Con el objetivo de alcanzar valores que se acerquen, o mejor, lleguen al valor optimo en los problemas abordados, y debido al carácter estocástico de las soluciones arrojadas por el ABC Algorithm, es que se opta por reparar las soluciones generadas, el cual es un procedimiento que toma una solución infactible y revalida sus restricciones para que se convierta en una solución factible.

Esto se realiza de la siguiente manera.

```
Inicio
 SolucionReparada := 0
 Para i:= 0 Hasta NumColumnas y SolucionReparada=0 Hacer
       Si SolucionInfactible[i] = 0
             Reparar solucion SolucionInfactible[i]
             Evaluar SolucionInfactible
       Si SolucionInfactible es Factible
             SolucionReparada:= 1
Fin
```

# 6 Versión Paralela de ABC Algorithm en CUDA.

Durante el desarrollo de este proyecto, se desarrollaron variadas versiones del ABC tanto en CPU como en GPU. Cada versión del algoritmo se fue complementando mediante distintas técnicas obtenidas en la literatura de CUDA, ABC Algorithm y del SCP. En resumen, se obtuvieron tres principales versiones que corresponden a las distintas estrategias desarrolladas que buscan resolver el SCP mediante la metaheurística de las abejas:

## 6.1 Versión en CPU

La versión en CPU corresponde a la aplicación secuencial del ABC Algorithm obtenida del estado del arte de la metaheuristica. También a esta versión se le agregó un procedimiento de reparación de las soluciones para obtener mejores resultados sin desperdiciar tiempo de ejecución. Hay que recalcar que durante la ejecución de esta versión del algoritmo no interviene la tarjeta gráfica.

La versión en CPU fue desarrollada para poder obtener un punto de comparación entre la versión secuencial (CPU) y la versión paralelizada con CUDA.

## 6.2 Versión en GPU (Primera Versión)

La primera versión en GPU del ABC Algorithm (en adelante GPU 1) fue desarrollada utilizando los principios antes mencionados en CUDA. Esta versión busca obtener todo el potencial del GPU aplicando paralelismo a nivel de datos para la resolución del SCP.

Como se mencionó en el capítulo 3 “Arquitectura GPU y Modelo de Programación CUDA”, lo que busca NVIDIA con su modelo de programación es que el GPU sea utilizado como una “unidad de apoyo” al CPU, con lo que paralelizando secciones del código desarrollado originalmente para CPU se pueda mejorar la potencia de cálculo general de la aplicación. Las secciones de código modificadas para ser ejecutadas en GPU son las siguientes:

- **Paralelización de Cálculo de Función Objetivo**: El cálculo de la función objetivo es una multiplicación de muchos datos, por lo tanto es un procedimiento apto de paralelizar (una instrucción sobre múltiples datos). Esta estrategia de paralelismo utiliza la solución a la cual se le desea calcular su valor optimo y reserva memoria correspondiente a esa solución en la GPU, en la cual se realiza la multiplicación de todas sus columnas por sus costos asociados (los cuales fueron reservados en memoria de la GPU al inicio del algoritmo), retornando el resultado de la multiplicación a la CPU para que continúe con el algoritmo.
- **Paralelización de Cálculo de Probabilidades:** El cálculo de Probabilidades no está muy alejado del cálculo de la función objetivo, su única diferencia es que realiza una división sobre múltiples datos, por lo que es un procedimiento apto de paralelizar. Esta estrategia de paralelismo utiliza el vector de fitness y realiza divisiones paralelas para calcular la probabilidad de cada valor de fitness de ese vector, una vez finalizado retorna un vector de probabilidades calculadas a la CPU para que continúe con la ejecución del algoritmo.

- **Reparación de Soluciones:** En la versión GPU 1 del ABC, se agregó una función para reparar las soluciones encontradas durante la ejecución de la aplicación. Hay que mencionar que si bien la reparación de las soluciones se agregó en la versión del código de GPU 1, la tarea en si es realizada exclusivamente en CPU.

## 6.3 Versión en GPU (Segunda Versión)

La segunda versión en GPU del ABC Algorithm (en adelante GPU 2) fue desarrollada utilizando la base de la versión GPU 1, a la cual se le hicieron mejoras posteriores a nivel de paralelismo. Entre las mejoras al modelo se destacan:

- **Reparación de Soluciones:** Se delegó el trabajo de la reparación de soluciones a la GPU, con lo cual, se obtiene mejoras de rendimiento.
- **Suma por reducción**: Una reducción es una combinación de todos los elementos de un vector en un valor único, utilizando para ello algún tipo de operador asociativo. Las implementaciones paralelas aprovechan esta asociatividad para calcular operaciones en paralelo. Al utilizar suma por reducción, se "prescinde" del CPU de la tarea de sumar vectores, reduciendo las transferencias CPU-GPU, con lo cual, se mejora el rendimiento general del algoritmo. Hay que mencionar que la suma por reducción se aplicó al cálculo de la función objetivo, al cálculo de probabilidades y a la reparación de soluciones (en GPU).

# 7 Experimentos y Resultados

Los experimentos realizados hasta la fecha utilizan las distintas instancias que otorga OR-Library [10] para el Set-Covering Problem publicados en su sitio web, estos resultados se obtienen utilizando un equipo que cuenta con el siguiente hardware:

- Procesador Intel Core i3 @ 3.3 Ghz
- Tarjeta de Video NVIDIA 660 Ti
- 8 GB de memoria RAM

Los parámetros del ABC Algorithm ejecutados en las pruebas son los siguientes:

- Población de abejas (Empleadas y Observadoras): 20
- Numero de fuentes de alimento: 10
- Numero de Iteraciones: 100.000
- Límite de intentos antes de abandonar una solución: 10.000

Las pruebas fueron ejecutadas 10 veces, obteniendo para cada batería de pruebas:

- Promedio de las 10 ejecuciones.
- Mejor resultado obtenido CPU y GPU (costo).
- Peor resultado obtenido CPU y GPU (costo).
- Tiempo de la mejor solución CPU y GPU.

Los resultados obtenidos con sus respectivos gráficos de comparación se detallan a continuación:

Tabla 7.1 Resultados CPU

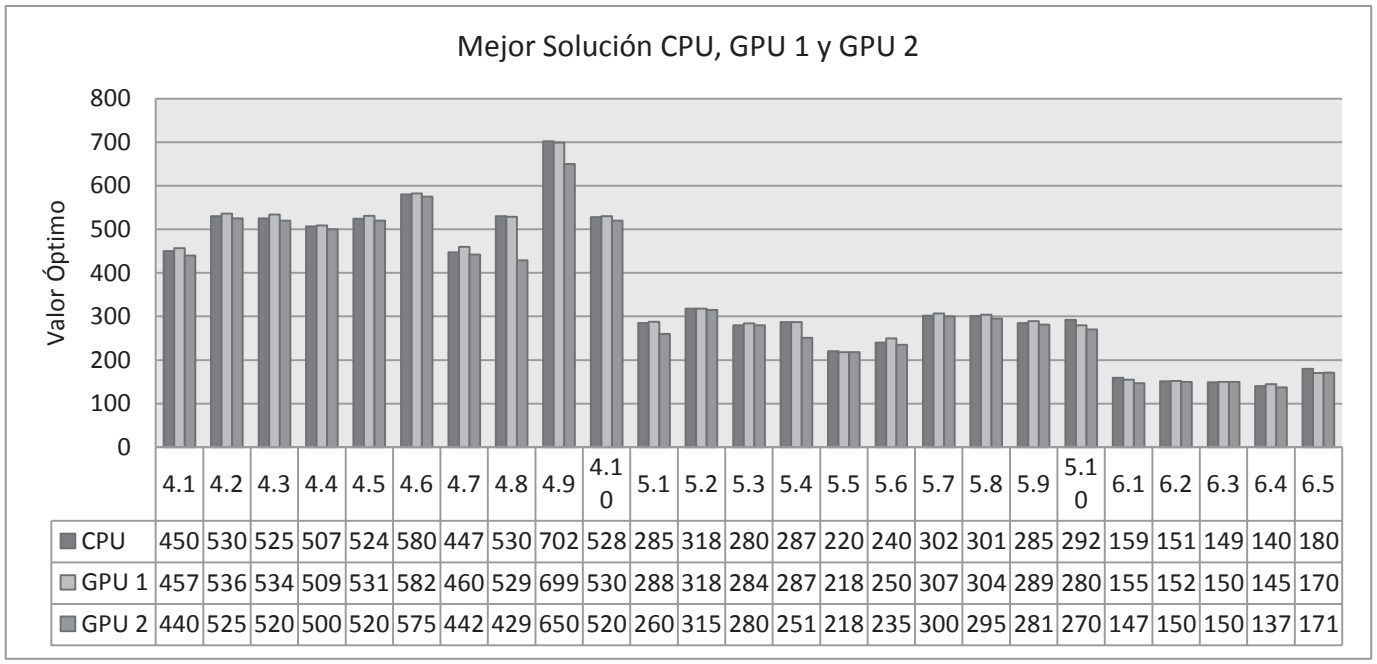
| INSTANCIA | | CPU | | | |
|---|---|---|---|---|---|
| **Problema** | **Valor Óptimo** | **Mejor Solución** | **Peor Solución** | **Promedio** | **Tiempo** |
| **4.1** | 429 | 450 | 470 | 463 | 32 |
| **4.2** | 512 | 530 | 541 | 537 | 35 |
| **4.3** | 516 | 525 | 570 | 562 | 34 |
| **4.4** | 494 | 507 | 515 | 510 | 33 |
| **4.5** | 512 | 524 | 561 | 537 | 34 |
| **4.6** | 560 | 580 | 592 | 583 | 37 |
| **4.7** | 430 | 447 | 460 | 451 | 34 |
| **4.8** | 492 | 530 | 551 | 537 | 35 |
| **4.9** | 641 | 702 | 717 | 709 | 36 |
| **4.10** | 514 | 528 | 542 | 532 | 35 |
| **5.1** | 253 | 285 | 306 | 305 | 40 |
| **5.2** | 302 | 318 | 331 | 319 | 37 |
| **5.3** | 226 | 280 | 289 | 286 | 45 |
| **5.4** | 242 | 287 | 301 | 292 | 43 |
| **5.5** | 211 | 220 | 261 | 246 | 46 |
| **5.6** | 213 | 240 | 268 | 252 | 42 |
| **5.7** | 293 | 302 | 318 | 314 | 40 |
| **5.8** | 288 | 301 | 340 | 330 | 41 |
| **5.9** | 279 | 285 | 302 | 292 | 43 |
| **5.10** | 265 | 292 | 345 | 300 | 46 |
| **6.1** | 138 | 159 | 169 | 164 | 49 |
| **6.2** | 146 | 151 | 191 | 180 | 51 |
| **6.3** | 145 | 149 | 175 | 161 | 50 |
| **6.4** | 131 | 140 | 160 | 151 | 53 |
| **6.5** | 161 | 180 | 201 | 198 | 48 |

Tabla 7.2 Resultados GPU 1

| INSTANCIA | | GPU 1 | | | |
|---|---|---|---|---|---|
| **Problema** | **Valor Óptimo** | **Mejor Solución** | **Peor Solución** | **Promedio** | **Tiempo** |
| **4.1** | 429 | 457 | 473 | 461 | 31 |
| **4.2** | 512 | 536 | 545 | 538 | 33 |
| **4.3** | 516 | 534 | 538 | 536 | 30 |
| **4.4** | 494 | 509 | 520 | 511 | 31 |
| **4.5** | 512 | 531 | 567 | 545 | 32 |
| **4.6** | 560 | 582 | 592 | 584 | 34 |
| **4.7** | 430 | 460 | 470 | 466 | 31 |
| **4.8** | 492 | 529 | 551 | 532 | 33 |
| **4.9** | 641 | 699 | 732 | 702 | 32 |
| **4.10** | 514 | 530 | 550 | 537 | 31 |
| **5.1** | 253 | 288 | 306 | 291 | 37 |
| **5.2** | 302 | 318 | 342 | 323 | 36 |
| **5.3** | 226 | 284 | 290 | 287 | 43 |
| **5.4** | 242 | 287 | 301 | 296 | 40 |
| **5.5** | 211 | 218 | 267 | 251 | 44 |
| **5.6** | 213 | 250 | 260 | 258 | 42 |
| **5.7** | 293 | 307 | 320 | 318 | 41 |
| **5.8** | 288 | 304 | 342 | 331 | 38 |
| **5.9** | 279 | 289 | 296 | 291 | 40 |
| **5.10** | 265 | 280 | 344 | 292 | 45 |
| **6.1** | 138 | 155 | 175 | 163 | 47 |
| **6.2** | 146 | 152 | 196 | 180 | 49 |
| **6.3** | 145 | 150 | 180 | 162 | 50 |
| **6.4** | 131 | 145 | 159 | 150 | 50 |
| **6.5** | 161 | 170 | 210 | 180 | 47 |

Tabla 7.3 Resultados GPU 2

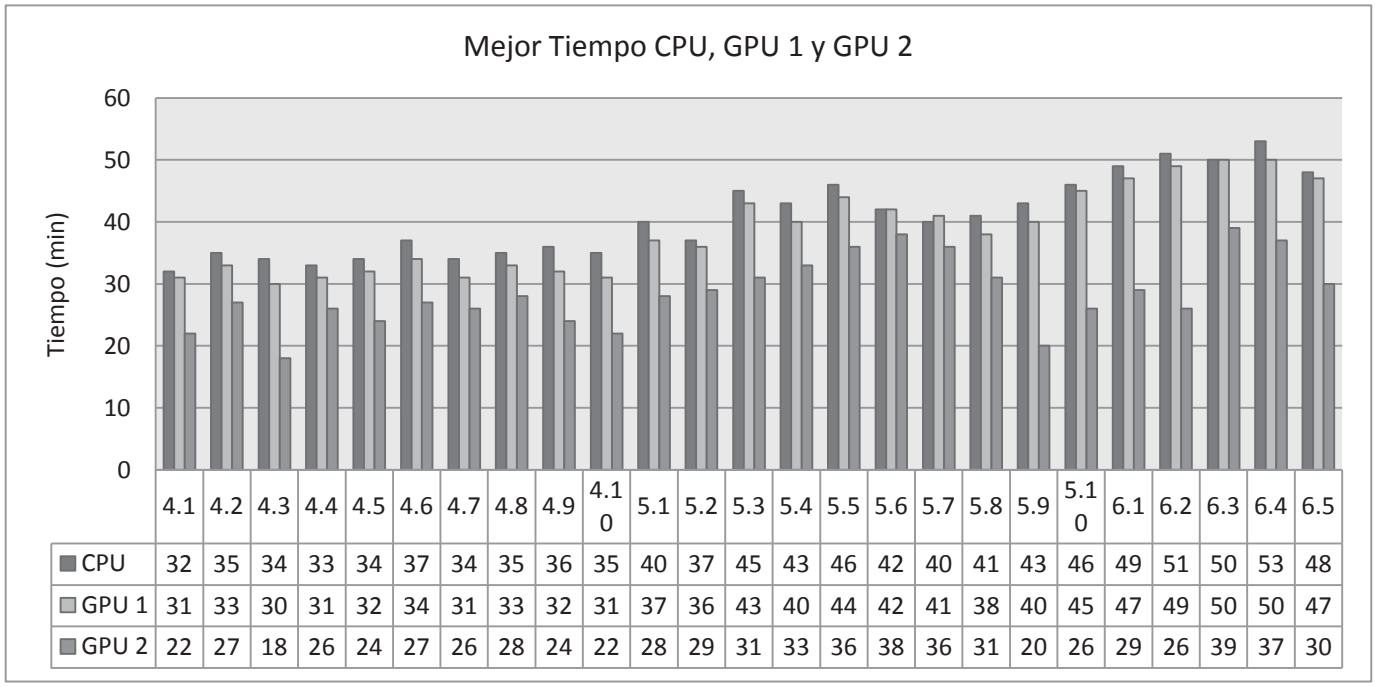
| INSTANCIA | | GPU 2 | | | |
|---|---|---|---|---|---|
| **Problema** | **Valor Óptimo** | **Mejor Solución** | **Peor Solución** | **Promedio** | **Tiempo** |
| **4.1** | 429 | 440 | 460 | 452 | 22 |
| **4.2** | 512 | 525 | 532 | 529 | 27 |
| **4.3** | 516 | 520 | 527 | 524 | 18 |
| **4.4** | 494 | 500 | 540 | 525 | 26 |
| **4.5** | 512 | 520 | 534 | 527 | 24 |
| **4.6** | 560 | 575 | 580 | 577 | 27 |
| **4.7** | 430 | 442 | 461 | 450 | 26 |
| **4.8** | 492 | 429 | 470 | 456 | 28 |
| **4.9** | 641 | 650 | 680 | 671 | 24 |
| **4.10** | 514 | 520 | 550 | 531 | 22 |
| **5.1** | 253 | 260 | 274 | 268 | 28 |
| **5.2** | 302 | 315 | 318 | 316 | 29 |
| **5.3** | 226 | 280 | 292 | 284 | 31 |
| **5.4** | 242 | 251 | 299 | 267 | 33 |
| **5.5** | 211 | 218 | 262 | 240 | 36 |
| **5.6** | 213 | 235 | 255 | 241 | 38 |
| **5.7** | 293 | 300 | 315 | 310 | 36 |
| **5.8** | 288 | 295 | 340 | 312 | 31 |
| **5.9** | 279 | 281 | 294 | 285 | 20 |
| **5.10** | 265 | 270 | 300 | 284 | 26 |
| **6.1** | 138 | 147 | 165 | 150 | 29 |
| **6.2** | 146 | 150 | 184 | 166 | 26 |
| **6.3** | 145 | 150 | 161 | 154 | 39 |
| **6.4** | 131 | 137 | 159 | 142 | 37 |
| **6.5** | 161 | 171 | 201 | 191 | 30 |




| | 4.1 | 4.2 | 4.3 | 4.4 | 4.5 | 4.6 | 4.7 | 4.8 | 4.9 | 4.10 | 5.1 | 5.2 | 5.3 | 5.4 | 5.5 | 5.6 | 5.7 | 5.8 | 5.9 | 5.10 | 6.1 | 6.2 | 6.3 | 6.4 | 6.5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CPU | 450 | 530 | 525 | 507 | 524 | 580 | 447 | 530 | 702 | 528 | 285 | 318 | 280 | 287 | 220 | 240 | 302 | 301 | 285 | 292 | 159 | 151 | 149 | 140 | 180 |
| GPU 1 | 457 | 536 | 534 | 509 | 531 | 582 | 460 | 529 | 699 | 530 | 288 | 318 | 284 | 287 | 218 | 250 | 307 | 304 | 289 | 280 | 155 | 152 | 150 | 145 | 170 |
| GPU 2 | 440 | 525 | 520 | 500 | 520 | 575 | 442 | 429 | 650 | 520 | 260 | 315 | 280 | 251 | 218 | 235 | 300 | 295 | 281 | 270 | 147 | 150 | 150 | 137 | 171 |

Figura 7.1 Gráfico Mejores Soluciones

Como se puede observar en el gráfico la figura 7.1, si bien, en la versión GPU 2 se nota levemente mejoras en los resultados de valores óptimos, estos de manera bien general siguen siendo parejos, esto se debe a que la base del algoritmo es la misma y las mejoras a nivel de paralelismo buscan reducir los tiempos de ejecución.




| | 4.1 | 4.2 | 4.3 | 4.4 | 4.5 | 4.6 | 4.7 | 4.8 | 4.9 | 4.10 | 5.1 | 5.2 | 5.3 | 5.4 | 5.5 | 5.6 | 5.7 | 5.8 | 5.9 | 5.10 | 6.1 | 6.2 | 6.3 | 6.4 | 6.5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CPU | 32 | 35 | 34 | 33 | 34 | 37 | 34 | 35 | 36 | 35 | 40 | 37 | 45 | 43 | 46 | 42 | 40 | 41 | 43 | 46 | 49 | 51 | 50 | 53 | 48 |
| GPU 1 | 31 | 33 | 30 | 31 | 32 | 34 | 31 | 33 | 32 | 31 | 37 | 36 | 43 | 40 | 44 | 42 | 41 | 38 | 40 | 45 | 47 | 49 | 50 | 50 | 47 |
| GPU 2 | 22 | 27 | 18 | 26 | 24 | 27 | 26 | 28 | 24 | 22 | 28 | 29 | 31 | 33 | 36 | 38 | 36 | 31 | 20 | 26 | 29 | 26 | 39 | 37 | 30 |

Figura 7.2 Gráfico Mejores Tiempos

En lo que respecta a tiempos de ejecución se pueden observar mejoras importantes. En la figura 7.2 se muestran los tiempos obtenidos correspondientes al valor óptimo de las diez ejecuciones del programa. Como se puede observar entre la versión en CPU y la versión en GPU 1 existen diferencias que si bien son significativas, siguen siendo bajas con respecto al potencial que se puede obtener de CUDA. Esto se debe a los numerosos intercambios de memoria CPU-GPU realizados en el algoritmo, debido a que existen tareas específicas (como la suma de vectores) que se delegan exclusivamente al CPU. En cambio, entre las versiones GPU 1 y GPU 2 se puede notar una gran reducción en los tiempos de ejecución, esto se debe a la mejora realizada en la suma de valores de los vectores, lo que implica una reducción en las transferencias GPU-GPU.

# 8 Conclusiones

A lo largo del presente informe se mostraron las características principales del modelo de programación CUDA, demostrando que el uso de esta tecnología basada en el uso de GPUs como unidades de apoyo a la CPU, permite mejorar el rendimiento en problemas que cuentan con un gran paralelismo.

El uso de CUDA no busca reemplazar el uso de un CPU en las aplicaciones, sino utilizar la gran cantidad de núcleos de una tarjeta gráfica para realizar las tareas aritméticamente intensivas y con esto, minimizar los tiempos de cómputo en distintos programas como en áreas científicas o ingenieriles.

En lo que respecta a las soluciones obtenidas en las múltiples pruebas se puede apreciar que si bien, ninguna llega al óptimo en 100.000 iteraciones, se encuentran bastante cercanas, lo que implica que la ejecución del algoritmo con otras configuraciones podría mejorar aún más los resultados.

También se pudo demostrar que NVIDIA mediante CUDA, puede mejorar los tiempos de ejecución de distintos algoritmos, como en este caso se realizó desarrollando una aplicación para resolver un problema conocido de optimización.

# 9 Referencias


[1] NVIDIA Corporation, "CUDA C Programming Guide Version 5.0", 2012.

[2] Broderick Crawford, Ricardo Soto, Eric Monfroy, Carlos Castro, Wenceslao Palma, Fernando Paredes, "A Hybrid Soft Computing Approach for Subset Problems", 2013

[3] Nysret Musliu, "Local search algorithm for unicost Set-Covering Problem" Vienna University of Technology, Vienna, Austria.

[4] Pablo Itaim Ananias, "Resolución del Problema del Set-Covering utilizando un Algoritmo Genético", Valparaíso, 20 de junio del 2005.

[5] NVIDIA Corporation, "What is GPU Computing", http://www.nvidia.com/object/what-is-gpu-computing.html, revisada por última vez 18 de abril del 2013.

[6] GPGPU.org, "General Purpose Computation on Graphics Hardware", http://gpgpu.org/about Revisada por última vez 18 de abril del 2013.

[7] NVIDIA Corporation, "CUDA API Reference Manual Version 4.2", 2012.

[8] NVIDIA Corporation, "Whitepaper NVIDIA GeForce GTX 680 ", 2012.

[9] Mina Husseinzadeh Kashan, Nasim Nahavandi, Ali Husseinzadeh, Ali Husseinzadeh Kashan, "DisABC: A new artificial bee colony algorithm for binary optimization", Department of Industrial Engineering, Faculty of Engineering, Tarbiat Modares University, 2011.

[10] J.E. Beasley, "OR-Library: distributing test problems by electronic mail", Journal of the Operational Research Society 41(11) (1990).

[11] Audrey Delévacq, Pierre Delisle, Marc Gravel, Michael Krajecki, "Parallel Ant Colony Optimization on Graphics Processing Units", Départamentd' Informatique et de Mathématique, Université du Québec à Chicoutimi, Saguenay, Canada, 2012.