Universidad Autónoma Metropolitana
Unidad Azcapotzalco

División de Ciencias Básicas e Ingeniería

Licienciatura en Ingeniería en Computación

Implementación paralela de algoritmos evolutivos

Proyecto que presenta:
Bertrand Jesús Almeida Arrieta

para obtener el titulo de:
Ingeniero en Computación

Asesores:
M. en C. Hilda María Chablé Martínez
M. en C. Oscar Alvarado Nava

México, D.F. **Diciembre de 2013**

Resumen

Se presenta la paralelización de un algoritmo evolutivo que tiene como objetivo encontrar un mínimo global, por medio de funciones de evaluación de diferentes individuos. La implementación hace uso de un analizador de ecuaciones matemáticas, lo que permite que las funciones a evaluar puedan ser definidas por el usuario.

Los resultados obtenidos fueron una ganancia significativa en la aceleración del tiempo de ejecución en los los módulos de mutación y de evaluación del algoritmo evolutivo en la implementación paralela con respecto a la implementación secuencial.

Agradecimientos

- A la División de Ciencias Básicas e Ingeniería de la Universidad Autónoma Metropolitana, Unidad Azcapotzalco.

- Al Departamento de Electrónica y de Sistemas

Dedicatoria

Dedicatoria

Índice general

Resumen III

Agradecimientos V

Dedicatoria VII

Lista de Figuras X

Lista de Tablas XI

1. Introducción **1**
1.1. Objetivos 2
1.2. Organización del reporte 2

2. Algoritmos evolutivos **3**

3. Algoritmo Shunting Yard **7**

4. Sistemas paralelos y GPUs **9**
4.1. Sistemas paralelos. 9
4.2. Hilos en GPUs 13
4.3. Memoria de una GPU 13
4.4. Arreglo de procesadores 15

5. CUDA **17**
5.1. GPGPU 17
5.2. CUDA 17
5.2.1. Terminología de CUDA 18
5.2.2. Modelo de programación 18
5.2.3. Extensiones CUDA 20
5.2.4. Direccionamiento en CUDA 21

6. Desarrollo del proyecto **23**
6.1. Diseño del algoritmo evolutivo secuencial 23
6.1.1. Generación y representación de individuos 23

6.1.2. Mutación y reproducción 24
6.1.3. Evaluación de la función objetivo 24
6.1.4. Competencia y selección 24
6.2. Paralelización del algoritmo evolutivo 25
6.2.1. Generación y representación de individuos 25
6.2.2. Módulo de Mutación 25
6.2.3. Módulo de Evaluación 27
6.2.4. Competencia y selección 28

7. Resultados y conclusiones 29
7.1. Sistemas de cómputo. 29
7.2. Experimentos 30
7.3. Resultados 31
7.3.1. Factor de aceleración 35
7.4. Conclusiones 39

A. Código fuente 43
A.1. Bibliotecas comunes usadas en ambas implementaciones. 43
A.1.1. Bibliotecas Adicionales 48
A.2. Programas principales 49

Índice de figuras

2.1. Esquema de funcionamiento del algoritmo genético. 4

4.1. Comparación de GFLOPS teóricos entre varios modelos de GPUs y CPU. 11
4.2. Comparación de la arquitectura de la CPU vs GPU. 12
4.3. Modelo de memoria de la GPU. 14
4.4. Arquitectura de una GPU. 15

5.1. Direccionamiento de hilos. 19
5.2. Arreglo bidimensional de bloques e hilos. 22

6.1. Funcionamiento del módulo de mutación en paralelo 26
6.2. Funcionamiento del módulo de evaluación en paralelo. 28

7.1. Gráfica comparativa de la ecuación (6.3) 31
7.2. Gráfica comparativa de la ecuación (6.4) 32
7.3. Gráfica comparativa de la ecuación (6.5) 33
7.4. Gráfica comparativa entre el algoritmo secuencial y el paralelo usando la ecuación (6.3) 34
7.5. Gráfica comparativa entre el factor de aceleración entre GPU utilizando la ecuación (6.3) 36
7.6. Gráfica comparativa entre el factor de aceleración entre GPU utilizando la ecuación (6.4) 37
7.7. Gráfica comparativa entre el factor de aceleración entre GPU utilizando la ecuación (6.5) 38

Índice de tablas

4.1. Taxonomía de Flynn. 9
4.2. Sistemas actuales de acuerdo a la taxonomía de Flynn. 9

7.1. Sistemas de pruebas. 29
7.2. Tarjetas nVidia utilizadas. 29
7.3. Tiempos promedio de distintos tamaños de población de la ecuación (6.3) 31
7.4. Tiempos promedio de distintos tamaños de población de la ecuación (6.4) 32
7.5. Tiempos promedio de distintos tamaños de población de la ecuación (6.5) 33
7.6. Comparativa de desempeño entre el algoritmo secuencial y el paralelo de la ecuación (6.3) 34
7.7. Factor de aceleración en las pruebas con distintos tamaños de población de la ecuación (6.3) 36
7.8. Factor de aceleración en las pruebas con distintos tamaños de población de la ecuación (6.4) 37
7.9. Factor de aceleración en las pruebas con distintos tamaños de población de la ecuación (6.5) 38

Capítulo 1

Introducción

La tendencia de la tecnología multinúcleo (*multi-core*) que existe hoy en día y su capacidad de cómputo parelelo comienza a ser aprovechada para mejorar la eficiencia de las aplicaciones, siendo las unidades de procesamiento gráfico o GPUs (*Graphics Processing Unit*) las de mayor potencial.

La ejecución **SIMD** (*Single Instruction, Multiple Data*) es una técnica computacional empleada para conseguir paralelismo a nivel de datos. La técnica consiste en que las instrucciones se reciben en una única unidad de control la cual envía estas instrucciones a diferentes unidades de procesamiento, donde las instrucciones son ejecutadas en un proceso sobre diferentes conjuntos de datos. El concepto está altamente asociado a las GPUs por su gran escalabilidad debido al número de procesadores *stream*[1] que tienen integrados.

El cómputo en GPUs se ha convertido en una alternativa para paralelizar parcial o totalmente aplicaciones en las cuales se busca mayor rendimiento en cuanto a tiempo de procesamiento. Dados los enfoques que tiene la arquitectura de una GPU y una unidad central de procesamiento o CPU (*Central Processing Unit*), una integración de CPU-GPU resulta beneficiosa al momento de diseñar aplicaciones que puedan explotar las ventajas de ambas.

Una forma de aprovechar el desempeño de las nuevas generaciones de CPU son las aplicaciones paralelas, con el enfoque del paralelismo varios procesos pueden realizarse simultáneamente, basándose en el principio de dividir los problemas grandes para obtener varios problemas pequeños, que son posteriormente solucionados en paralelo.

[1] Un *stream* es un conjunto de registros que requieren un cálculo similar, estos *stream* proveen paralelismo a nivel de datos.

1.1. Objetivos

El objetivo general de este proyecto es implementar y paralelizar un algoritmo evolutivo que pueda ser ejecutado en GPUs.

A continuación se listan los objetivos específicos que se desarrollaron en este proyecto:

1. Implementar el algoritmo de Shunting Yard para la captura de las funciones de minimización del algoritmo evolutivo.
2. Modelar un algoritmo evolutivo para minimización de funciones.
3. Utilizar las características de una GPU para el cálculo y procesamiento en paralelo de un algoritmo evolutivo.
4. Realizar una comparativa de desempeño de la implementación del algoritmo secuencial en una CPU y la implementación paralela en una GPU.
5. Comprobar las ganancias de velocidad en los tiempos de ejecución de un algoritmo evolutivo aplicando técnicas de cómputo paralelo

1.2. Organización del reporte

El reporte está organizado de la siguiente forma:

- Algoritmos evolutivos: Breve introducción a los algoritmos evolutivos.
- Algoritmo Shunting Yard: La sección explica el funcionamiento y propósito de este algoritmo.
- Sistemas paralelos y GPUs: Esta sección da un acercamiento a los sistemas paralelos y la arquitectura de las GPUS.
- CUDA: Introducción a la plataforma CUDA.
- Desarrollo del proyecto: En esta sección se detalla la implementación del proyecto.

Capítulo 2

Algoritmos evolutivos

Los algoritmos evolutivos son métodos de optimización y búsqueda de soluciones basados en los postulados de la evolución biológica. En ellos se mantiene un conjunto de entidades que representan posibles soluciones, las cuales se mezclan, y compiten entre sí, de tal manera que las más aptas son capaces de prevalecer a lo largo del tiempo, evolucionando cada vez hacia mejores soluciones.

En general suele hablarse de tres paradigmas principales de la computación evolutiva:

- Programación Evolutiva.
- Estrategias Evolutivas.
- Algoritmos Genéticos.

La manera en la que se imita a la naturaleza decidirá cómo evoluciona el algoritmo. Existen varias formas de aproximarse a esta imitación, pero existen unos elementos comunes en la mayoría de los algoritmos evolutivos:

1. Un problema de optimización a resolver.

2. Individuos con código genético: cada individuo es una posible solución al problema, y esta solución se describe completamente con su código genético.

3. Una función de evaluación de cada individuo: la solución correspondiente a cada individuo puede ser más o menos buena. Esta función cuantifica que tan buena es la solución, de manera que se puedan elegir los mejores individuos.

4. Una forma de obtener nuevas generaciones: los mejores individuos tendrán descendencia y, para que exista variabilidad se introducen conceptos como mutación y reproducción sexual.

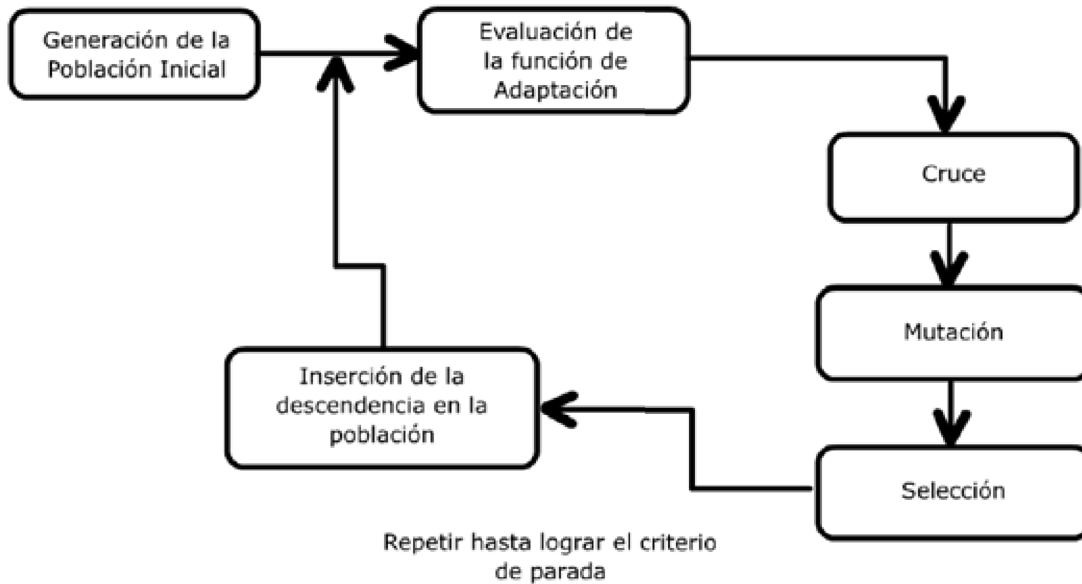


Figura 2.1: Esquema de funcionamiento del algoritmo genético.

En la Figura 2.1 se muestra un esquema del funcionamiento de un algoritmo genético. Siguiendo el esquema presentado, después de un cierto número (posiblemente grande) de generaciones, se espera obtener individuos (soluciones) adaptados al problema que se trata de resolver.

Los algoritmos evolutivos se componen de las siguientes operaciones básicas:

- Cruce.
- Mutación.
- Evaluación.
- Selección.

Cada operación se describe a continuación.

Cruce.

Se denomina operación de cruce a la forma de calcular el genoma del nuevo individuo en función del genoma de los padres. Ésta se obtiene operando sobre dos cromosomas a la vez para generar dos descendientes donde se combinan las características de ambos cromosomas padres.

Mutación.

Se define como una variación de la información contenidas en el código genético, usualmente un cambio de un gen a otro producido por algún factor exterior al algoritmo genético. La razón por la cual se hace mutación sobre los individuos es para incrementar el número de saltos evolutivos. La mutación permite explorar nuevos subespacios de soluciones, lo que enriquece la diversidad genética de la población, además de ser un mecanismo de prevención para evitar la degeneración de la población.

Función de evaluación.

La función se utiliza para evaluar a los individuos, en caso de ser seleccionados para la competencia de torneo, éstos serán filtrados dependiendo de su medida de adaptación o el equivalente de ser una posible solución al problema.

Selección.

Hay varias técnicas de seleccionar un subconjunto de la población

- Selección directa: toma elementos de acuerdo a un criterio objetivo, por ejemplo "los N mejores".
- Selección aleatoria: puede ser realizado por selección equiprobable o selección estocástica.
  - Selección equiprobable: todos tienen la misma probabilidad de ser escogidos.
  - Selección estocástica: la probabilidad de que un individuo sea escogido depende de una heurística. Los distintos procedimientos estocásticos son:
    - Selección por sorteo: cada individuo de la población tiene asignado un rango proporcional -o inversamente proporcional- a su adaptación. Se escoge un número aleatorio dentro del rango global, y el escogido es aquel que tenga dicho número dentro de su rango. La probabilidad de ser escogido es proporcional/inversamente proporcional al grado de adaptación del individuo.
    - Por ruleta: El comportamiento es similar al de una ruleta, donde se define un avance en cada tirada a partir de la posición actual. Tiene la ventaja de que no es posible escoger dos veces consecutivas el mismo elemento, y que puede ser forzado a que sea alta la probabilidad de que no sean elementos próximos en la población.

- Por torneo: Se escoge un subconjunto de individuos de manera aleatoria y de entre ellos selecciona el más adecuado por otra técnica determinística de tipo 'el mejor' o 'el peor'. Esta técnica tiene la ventaja de que permite un cierto grado de elitismo -el mejor nunca va a morir, además de tener más probabilidad de reproducirse.

Capítulo 3

Algoritmo Shunting Yard

El algoritmo Shunting Yard es un método para analizar las ecuaciones matemáticas especificadas en la notación infija, el cual es utilizado para producir la salida en la notación polaca inversa o RPN[1]. En el proyecto se utiliza el algoritmo para aceptar función definidas por parte del usuario, de manera que la aplicación sea flexible. La salida de dicho algoritmo será una estructura de pila que contendrá la RPN de la función, la cual será enviada a la GPU para que sea evaluada en paralelo en el módulo de evaluación de la función objetivo. Esta función es leída desde un archivo de texto previamente definida por el usuario.

A continuación se muestra un ejemplo de la representación polaca inversa. Dada la siguiente función:

$$X_1\text{\^{}}2 + X_2\text{\^{}}2 \tag{3.1}$$

la función 3.1 queda como:

$$RPN = X_1 2\text{\^{}} X_2 2\text{\^{}}+ \tag{3.2}$$

El pseudo código del algoritmo Shunting Yard se muestra a continuacii¿½n en el cuadro 1.

[1]RPN por sus siglas en inglés *Reverse Polish Notation*

```
while haya tokens a ser leídos do
    Lea un token;
    if el token es un número then
        agréguelo a la cola de salida;
    end
    if el token es un token de función then
        póngalo (push) sobre la pila;
    end
    if el token es un separador de un argumento de función then
        while que el token en el tope de la pila sea un paréntesis abierto do
            retirar (pop) de la pila a los operadores e insertarlos en la cola de salida;
            if no es encontrado ningún paréntesis abierto then
                los paréntesis fueron mal emparejados;
            end
        end
    end
    if el token es un operador, o1, then
        while que haya un operador, o2, en el tope de la pila y o1 es asociativo izquierdo y su precedencia
        es menor que (una precedencia más baja) o igual a la de o2, ó o1 es asociativo derecho y su
        precedencia es menor que (una precedencia más baja) que la de o2 do
            retirar (pop) de la pila el o2, e insertarlo en la cola de salida;
        end
        insertar o1 en el tope de la pila;
    end
    if el token es un paréntesis abierto then
        insertar en la pila;
    end
    if el token es un paréntesis derecho then
        while que el token en el tope de la pila sea un paréntesis abierto do
            retirar a los operadores de la pila e insertarlos en la cola de salida.;
        end
        Retirar el paréntesis abierto de la pila, pero no insertarlo en la cola de salida;
        if el token en el tope de la pila es un token de función then
            insertar en la cola de salida;
        end
        if la pila se termina sin encontrar un paréntesis abierto then
            hay parńtesis sin pareja;
        end
    end
end
if no hay más tokens para leer then
    while todavía haya tokens de operadores en la pila do
        if el token del operador en el tope de la pila es un paréntesis then
            hay parï¿½ntesis sin la pareja correspondiente ;
        end
        retirar al operador e insertarlo en la cola de salida;
    end
end
```

Algoritmo 1: Pseudocódigo del Algoritmo Shunting Yard.

Capítulo 4

Sistemas paralelos y GPUs

4.1. Sistemas paralelos.

Los sistemas paralelos buscan mejorar la velocidad de ejecución de los programas al utilizar múltiples sistemas de procesamiento. La taxonomía de Flynn se muestra en la Tabla 4.1. sistemas:

	Una instrucción	**Múltiples instrucciones**
Un dato	SISD	MISD
Múltiples datos	SIMD	MIMD

Tabla 4.1: Taxonomía de Flynn.

- **SISD** *Single Instruction, Single Data*
- **SIMD** *Single Instruction, Multiple Data*
- **MISD** *Multiple Instruction, Single Data*
- **MIMD** *Multiple Instruction, Multiple Data*

Flujo instrucciones	**Flujo de datos**	**Nombre**	**Ejemplos**
1	1	SISD	Máquina de Von Neumann
1	Múltiple	SIMD	Máquinas vectoriales
Múltiple	Múltiple	MIMD	Multiprocesador y arreglos

Tabla 4.2: Sistemas actuales de acuerdo a la taxonomía de Flynn.

Existen varios niveles de paralelismo en los sistemas de cómputo actuales:

- Paralelismo a nivel de microarquitectura.
  - Sistemas con *pipeline* y superescalares.
- Sistemas de memoria compartida
  - Sistemas multiprocesador
  - Sistemas multinúcleo
  - Sistemas en arreglo (GPU)
- Sistemas de memoria distribuida
  - Sistemas en cluster
  - Sistemas masivamente paralelos

La mayoría de los procesadores convencionales implementan una arquitectura superescalar, lo que les permite ejecutar más de una instrucción por ciclo de reloj, siempre y cuando las instrucciones no presenten algún tipo de dependencia, ya sea de control, estructural o de datos. Para evitar estos riesgos, los procesadores junto con el compilador recurren a técnicas de ejecución especulativa, lo cual causa que el manejo y control de instrucciones sea complejo [1]. Estos procesadores presentan el primer nivel de paralelismo a nivel de microarquitectura.

Usualmente un sistema multiprocesador consta mínimo de 2 procesadores convencionales conectados en el mismo bus del sistema, lo cual hace a los sistemas multiprocesador que no sean fácilmente escalables debido al problema de contención de memoria principal.

En la actualidad los procesadores convencionales cuentas con más de un núcleo, los cuales se duplican aproximadamente cada generación, comunicando todos los núcleos por medio de un sistema de bus interno al chip mejorando el problema de contención.

Los arreglos de procesadores cuentan con una gran cantidad de núcleos pero mucho más simples que un núcleo de una CPU convencional. En todos los núcleos se ejecuta la mismas instrucciones pero con datos distintos. Es en esta categoría donde se ubican los GPUs.

Mientras que la mejora del rendimiento de los microprocesadores convencionales se ha desacelerado significativamente, las GPUs han seguido mejorando continuamente. Este fenómeno es ilustrado en la Figura 4.1, la cual muestra una comparación de los GigaFLOPS[1] que pueden alcanzar teóricamente varios modelos de GPUs y CPU.

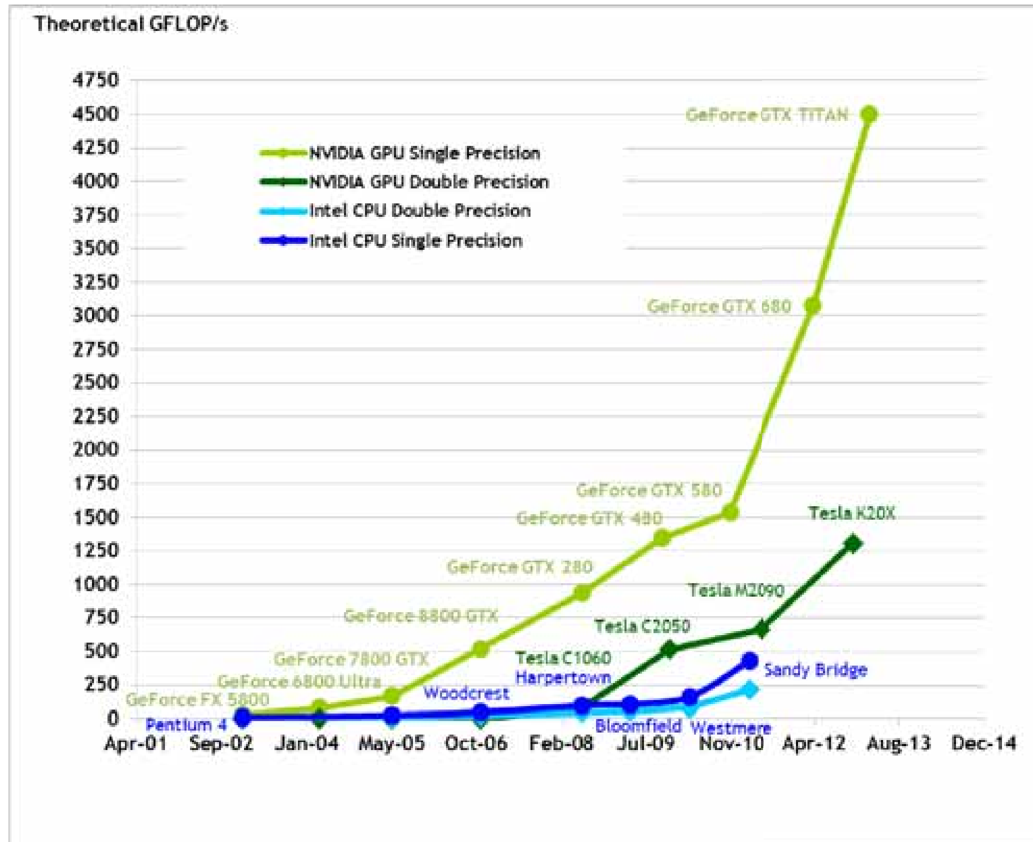


Figura 4.1: Comparación de GFLOPS teóricos entre varios modelos de GPUs y CPU.

La explicación de la brecha de rendimiento entre un sistema en arreglo (GPUs) y un sistema multinúcleo se encuentra en las filosofías de diseño fundamentales entre los dos sistemas. El diseño de un sistema multinúcleo está optimizado para el rendimiento del código secuencial. Para esto se hace uso de una lógica de control compleja para permitir que las instrucciones de un sólo hilo de ejecución sean ejecutadas en paralelo o incluso fuera de su orden secuencial, manteniendo la apariencia de una ejecución secuencial. Más importante aún, se proporcionan grandes memorias caché para reducir las latencias de instrucciones y acceso a datos de aplicaciones complejas. Ni la lógica de control ni las memorias caché contribuyen a la velocidad de procesamiento.

La filosofía de diseño de la GPU está determinada por el rápido crecimiento en la industria de los videojuegos, la cual ejerce una enorme presión económica a la capa-

[1] FLOPS = operación de punto flotante por segundo

cidad para realizar una enorme cantidad de cálculos de punto flotante por fotograma de vídeo. Esta demanda motiva a los vendedores de GPUs a buscar maneras de maximizar el área del chip y la energía dedicados a cálculos de punto flotante. La solución que prevalece hasta la fecha es optimizar el rendimiento de la ejecución de un número masivo de hilos. En la Figura 4.1 se muestra una comparativa de la arquitecturas de los CPUs y los GPUs.

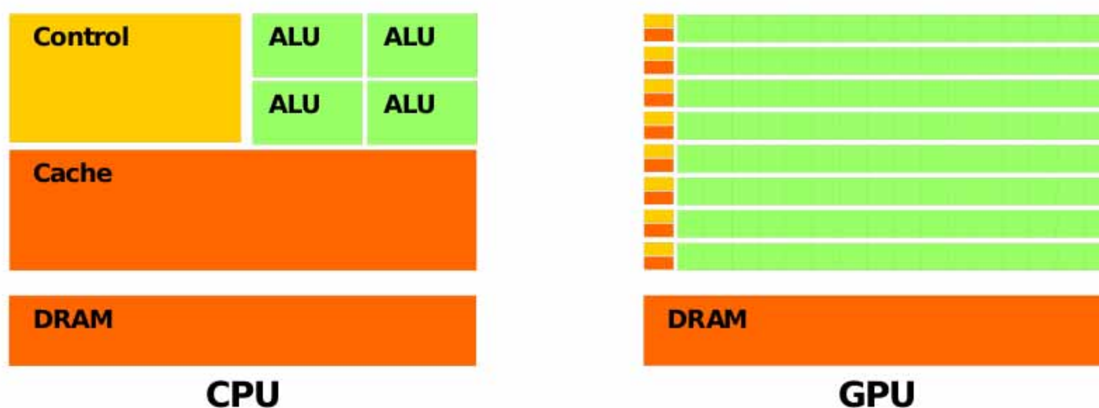


Figura 4.2: Comparación de la arquitectura de la CPU vs GPU.

En las GPUs el hardware toma ventaja de un gran número de hilos de ejecución para encontrar trabajo que ejecutar cuando algunos de ellos están a la espera de los accesos a memoria de gran latencia, minimizando la lógica de control necesaria para cada subproceso de ejecución. Las pequeñas memorias caché ayudan a controlar los requisitos de ancho de banda de estas aplicaciones para múltiples subprocesos que tienen acceso a los mismos datos de la memoria no tengan necesidad de ir todos a la DRAM. Como resultado de ello, mucho más área de chip se dedica para los cálculos de punto flotante.

Por otra parte el ancho de banda de memoria en las GPUs opera aproximadamente 10 veces que el ancho de banda de los CPUs. Debido a la manera en que varios programas de sistema, aplicaciones, y dispositivos entrada-salida esperan sus accesos a memoria para trabajar. Los procesadores de propósito general tienen que satisfacer tales requisitos, haciendo que el ancho el ancho de banda de memoria sea más difícil de aumentar [2]. En contraste, con los modelos de memoria más simples y con menos limitantes en el diseño de GPUs se pueden lograr más fácilmente un ancho de banda más alto.

4.2. Hilos en GPUs

Las CPUs manejan hilos "pesados" , mientras que las GPU usan hilos "ligeros". La ejecución de un hilo pesado puede requerir el cambio de un conjunto diferente de recursos contextuales. El contexto de un hilo es una parte equitativa de la CPU, un reloj de hardware genera interrupciones periódicamente. Esto permite al sistema operativo programar todos los procesos en la memoria principal para ejecutarlos en la CPU a intervalos iguales. Cada vez que una interrupción de reloj ocurre, la CPU (en el núcleo) recoge un proceso diferente para ejecutar. Cada interrupción de la CPU de un proceso a otro se denomina un cambio de contexto.Un ejemplo de un hilo de pesado son los procesos que genera y administra un sistema operativo, donde los procesadores pueden necesitar tener acceso a más recursos , y el tiempo de cambio de contexto puede ser mayor.

Un hilo ligero, también conocido como subproceso, es un proceso que puede compartir el espacio de direcciones y recursos con otros hilos, reduciendo el tiempo de cambio de contexto durante la ejecución. Los hilos generalmente son comparados en tiempo de ejecución, un hilo ligero toma menos tiempo en ejecutarse que un hilo pesado [3].

4.3. Memoria de una GPU

Los componentes principales en la arquitectura de una GPU son la jerarquía de memoria y el arreglo de procesadores.

La memoria en los GPUs tiene diferentes capacidades de almacenamiento y diferentes tiempos de acceso, generando la siguiente jerarquía:

- Memoria Global(*Global memory*): Es la memoria de mayor capacidad disponible y tiene una gran latencia. Ésta puede ser accedida por todos los hilos independientemente del bloque y el *grid* donde se encuentren.
- Memoria compartida (*Shared memory*): Es asignada por bloque y se puede acceder a todos los hilos en el bloque. Es de menor latencia que la memoria global.
- Memoria Constante: Éste es el lugar donde se almacenan las constantes y argumentos del kernel. El acceso es lento pero con facilidad de acceso desde *host*.
- Registros: Es la clase más rápida de la memoria y es privada para cada hilo.
- Memoria Local: Cada hilo tiene su memoria local, es lenta, pero en caché.

La memoria global es muy parecida a la RAM en una CPU, es accesible tanto por la propia GPU como la CPU. Actualmente cada GPU cuenta gigabytes de memoria de doble velocidad de datos Gráficos, GDDR[2] DRAM, conocida como *Global memory*.

La memoria global y la memoria constante pueden ser escritas o leídas por el *host* a través de una interfaz de programación o API[3], pero la memoria constante es sólo de lectura para la GPU cuando todos los hilos simultáneamente accesan a la misma locación de memoria.

Los registros y la memoria compartida son memorias en chip, las variables que residen en este tipo de memorias puede ser accesados a altas velocidades de manera paralela. Los registros se asignan individualmente a cada hilo, cada hilo puede accesar sólo a su propio registro. Una función *kernel* típicamente utiliza registros para mantener las variables de acceso frecuente que son privadas para cada hilo.

La memoria compartida es asignada para los hilos de un sólo bloque; todos los hilos en un bloque pueden acceder a variables en la memoria compartida, lo cual hace que sea más eficiente la cooperación entre hilos compartiendo sus datos de entrada y resultados intermedios de sus procesos [2]. En la Figura 4.3) se muestran los modelos de memoria de una GPU.

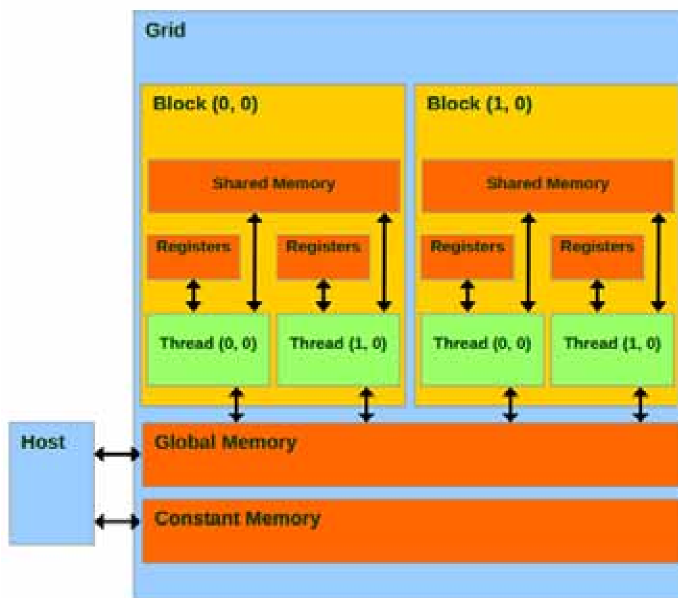


Figura 4.3: Modelo de memoria de la GPU.

[2]GDDR por sus siglas en inglés *Graphics Double Data Rate*
[3]API por sus siglas en inglés *Application Programming Interface*

4.4. Arreglo de procesadores

El segundo componente es el arreglo de procesadores o *stream*, los cuales están diseñados para ejecutar cientos de hilos simultáneamente. En la arquitectura Fermi cada multiprocesador *stream* cuenta con 32 núcleos CUDA en grupos de 16 (es decir 2 grupos de 16 núcleos CUDA por multiprocesador), funcionando a 700 MHz cada uno (véase Figura 4.4). Además de contar con 16 unidades de carga/almacenamiento que permiten operaciones de acceso a la memoria por cada 16 hilos por ciclo de reloj. Estos núcleos CUDA cuenta con una ALU de 32 bits con *pipeline* y una unidad de punto flotante (FPU) capaz de ejecutar una operación entera o de punto flotante en cada ciclo de reloj [4]. En la Figura 4.4 se muestra la arquitectura completa de los GPUs.

Un arreglo de procesadores está diseñado para ejecutar cientos de hilos simultáneamente. Para gestionar una gran cantidad de hilos tales se emplea una arquitectura única llamada SIMT (*Single-Instruction, Multiple-Thread*). Las instrucciones están en *pipeline* para aprovechar el paralelismo a nivel de instrucción dentro de un solo hilo, así como el uso paralelismo a nivel de hilo dentro del hardware. Todas las instrucciones se emiten en orden y no hay ninguna predicción de saltos y tampoco hay ejecución especulativa[5].

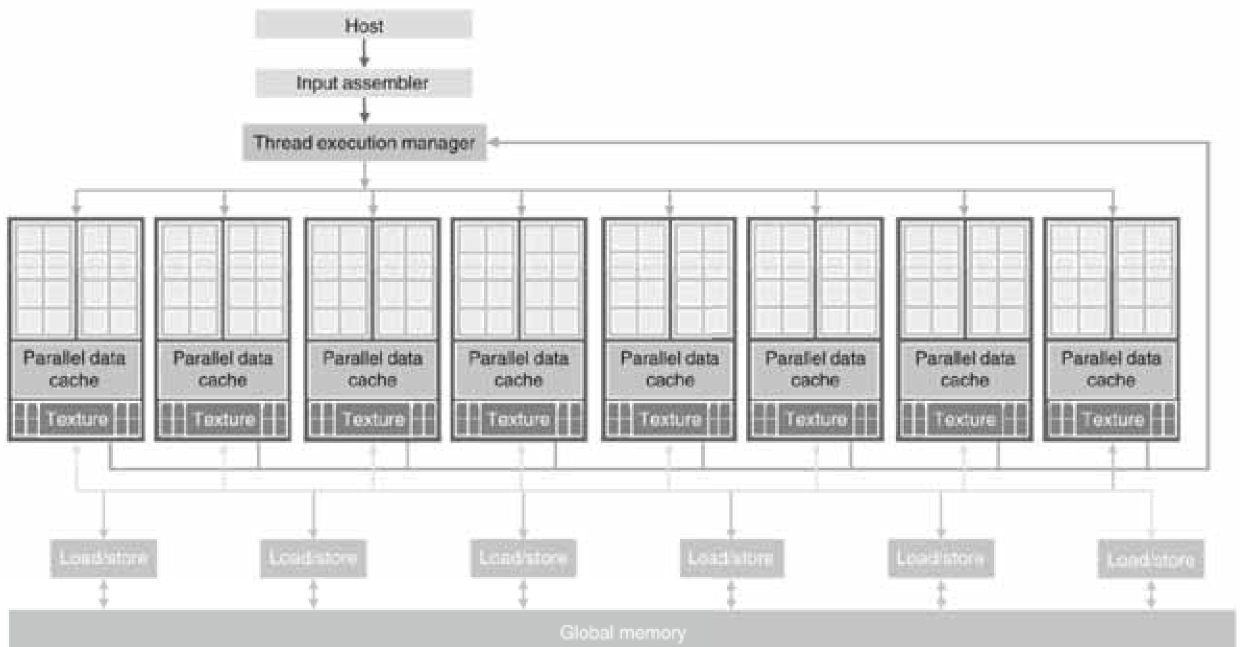


Figura 4.4: Arquitectura de una GPU.

Capítulo 5

CUDA

5.1. GPGPU

El procesamiento GPGPU[1] es la utilización de una GPU, que por lo general se encarga únicamente del cómputo en aplicaciones gráficas, para llevar a cabo cálculos de aplicaciones de propósito general como las que tradicionalmente son ejecutadas por un CPU. El GPGPU toma ventaja del gran número de elementos de cálculo con los que cuentan los GPU para generar múltiples hilos que podrían completar un cálculo en un menor tiempo. Así mismo se aprovecha la mayor capacidad de cómputo punto flotante con la que cuentan los GPUs generando un cálculo computacional intensivo, altamente paralelo las cuales se consideran altamente atractivas para su uso en aplicaciones especialmente en el ámbito científico y de simulación.

5.2. CUDA

CUDA[2] hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por la empresa nVidia que permiten a los programadores usar una variación del lenguaje de programación C/C++ para codificar algoritmos en GPUs de nVidia. Por medio de envolturas o *wrappers* se puede usar Python, Fortran y Java en vez de C/C++ y en el futuro también se añadirá FORTRAN, OpenGL y Direct3D. Funciona en todas las GPU nVidia de la serie G8X en adelante, incluyendo GeForce, Quadro, ION y la línea Tesla.

CUDA intenta explotar las ventajas de las GPU frente a las CPU de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneos.

[1]GPGPU por sus siglas en inglés *General-Purpose Computing on Graphics Processing Units*.
[2]CUDA por sus siglas en inglés *Compute Unified Device Architecture*

El primer SDK se publicó en febrero de 2007 en un principio para Windows, Linux, y más adelante en su versión 2.0 para Mac OS. Actualmente se ofrece para Windows XP/Vista/7, para Linux 32/64 bits y para Mac OS.

5.2.1. Terminología de CUDA

- ***Thread***: (Hilo) es la unidad más pequeña ejecución de una instrucción.
- ***Block***: (Bloque) contiene varios hilos.
- ***Warp***: Un grupo de hilos ejecutados físicamente en paralelo (por lo general se ejecuta la misma aplicación).
- ***Grid***: (Rejilla), contiene varias secuencias de bloques de hilos.
- ***Kernel***: una aplicación o programa que se ejecuta en cada hilo en la GPU.
- ***Device***:(dispositivo), se refiere a la GPU.
- ***Host***:(anfitrión) La CPU.

5.2.2. Modelo de programación

CUDA extiende C al permitir al programador definir funciones llamadas *kernels*, cuando se llama una función *kernel*, se ejecutan N veces en paralelo por diferentes hilos N CUDA, en lugar de una sola vez como funciones regulares en C/C++. Un kernel se define utilizando el especificador de declaración `__ global__` y el número de hilos CUDA que ejecutan en esa llamada de *kernel* se especifica mediante paréntesis angulares de la siguiente manera <<< ... >>>, la cual es la sintaxis de configuración de ejecución. Cada subproceso (hilo) que ejecuta el núcleo se le asigna un identificador único que es accesible dentro del kernel mediante la variable incorporada **threadIdx** [6].

Los hilos son la unidad mínima de ejecución, éstos se agrupan en bloques, los cuales a su vez se agrupan en *grids*, de manera que cada *grid* es ejecutado por un *kernel* (véase Figura 5.1).

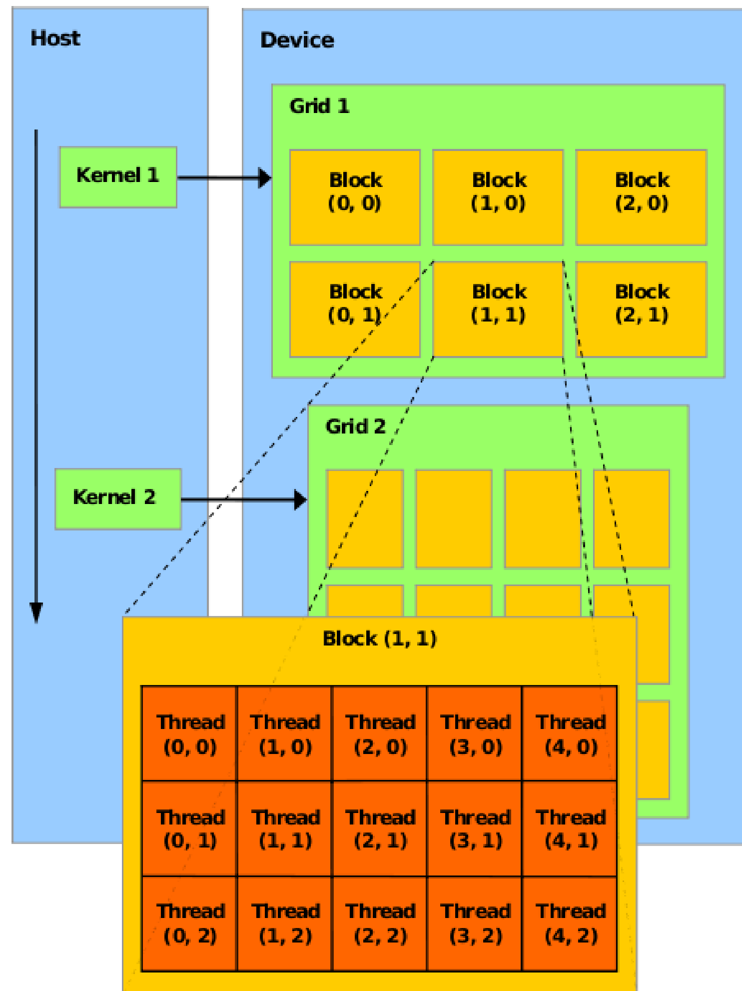


Figura 5.1: Direccionamiento de hilos.

5.2.3. Extensiones CUDA

Hay 4 extensiones principales en CUDA, éstas son :

Calificadores de Tipo Función: especifican si la ejecución es en el host o en el dispositivo. Hay 3 subtipos:

- *device*: la función se llama y se ejecuta en el dispositivo.
- *shared*: la función se llama desde el host y se ejecuta en el dispositivo.
- *host*: la función se llama y se ejecuta en el host.

Calificadores de Tipo Variable: especifica la localidad de memoria en el dispositivo y el tiempo de vida de la variable, hay 2 subtipos:

- *device*: la variable tiene el tiempo de vida de la aplicación y se almacena en la memoria global, lo que hace que sea accesible desde todos los bloques dentro del grid.
- *shared*: la variable tiene la vida útil del bloque y se almacena en la memoria compartida, por lo tanto, accesible desde todos los hilos dentro del bloque.

Calificadores de Tipo Configuración de Ejecución: especifica como ejecutar un kernel en el dispositivo, de la siguiente manera:

<<< **GridDimension, BlockDimension, Ns, S** >>>

Donde:
GridDimension: Dimensión de la rejilla.
BlockDimension: Dimensión de la bloque.
Ns (opcional): la cantidad de memoria asignar a ésta función.
S (opcional): especifica un *stream* asociada a esta función.

Calificadores de Tipo Variables Incorporadas: 4 variables especifican el tamaño del grid y bloques y los índices de los threads.

gridDim ->especifica la dimensión de la rejilla.
blockDim ->especifica la dimensión del bloque.
BlockIdx ->ID del bloque.
ThreadIdx ->ID del hilo.

5.2.4. Direccionamiento en CUDA

Hilos y bloques deben tener un identificador único para acceder a ellos, desde el programa. Es importante ya que son los principales componentes cuando se trata de escribir código paralelo eficiente. CUDA tiene variables predefinidas las cuales proporcionan el valor del índice del bloque en el que se encuentra, ya sea en una o dos dimensiones. Al momento de lanzar el *kernel*, se especifica N como el número de bloques paralelos como:

Kernel <<<N, k >>>

N es interpretado como unidimensional en este caso, con k hilos por cada bloque, hay que tener en cuenta que no debe exceder más de 65,535 bloques por dimensión (es decir si se lanza un *kernel* en 2 dimensiones el máximo de bloques sería de $65,535^2$), esto es por el límite propio del *hardware*. Igualmente el número de hilos por bloque está limitado, éste varía dependiendo de la GPU pero el más común es de 512 hilos por bloque.
Tanto hilos como bloques el máximo de dimensiones es de 3.

Para poder accesar a los hilos a traves de multiples bloques e hilos, el indexado se logra con variables predefinidas por CUDA, a continuación se muestra un ejemplo de como sería el indexado en bloques de una dimensión.

int tid = threadIdx.x + blockIdx.x * blockDim.x;

donde **tid** es identificador del hilo en el bloque donde está alojado, **blockDim** es una variable que define el tamaño de todos bloques en hilos, dado que son bloques unidimensionales solo se utiliza **blockDim.x**.
blockId.x es la posición del bloque actual. Una forma más sencilla de explicarlo es pensando en que los bloques son filas y los hilos son columnas (véase Figura 5.2.4).

Cuando se trata de bloques en 2D o 3D, el identificador de hilo se calcula de manera similar, como una función, el cual representa al hilo dentro del bloque. Dentro de un bloque 2D los hilos son posicionados en forma (0,0), (0,1), ... (n-1, n-1) .

La función para obtener el identificador del hilo en un bloque 2D sería :

x + y * Dimx

Donde 'x' y 'y' son los índices del hilo en x y y, Dimx es la dimensión x en el bloque.
En un bloque de 3D se aplica lo mismo con la diferencia de tratar con una dimensión más que se traduce en la función:

x + y * Dimx + z* Dimx * Dimy

Donde 'x', 'y' y 'z' son los índices del hilo (igual que para el bloque 2D) y Dimx y Dimy son las dimensiones x,y del bloque.

Bloque 0	Hilo 0	Hilo 1	Hilo 2	Hilo 3
Bloque 1	Hilo 0	Hilo 1	Hilo 2	Hilo 3
Bloque 2	Hilo 0	Hilo 1	Hilo 2	Hilo 3
Bloque 3	Hilo 0	Hilo 1	Hilo 2	Hilo 3

Figura 5.2: Arreglo bidimensional de bloques e hilos.

Capítulo 6

Desarrollo del proyecto

Este proyecto se implementó utilizando **programación evolutiva**, dado que esta técnica no requiere de un módulo de cruce. La operación de cruce de los cromosomas tiene una alta dependencia de datos, mientras que la mutación es completamente independiente con un alto potencial de paralelismo. La implementación de la propuesta se desarrollo en 2 fases:

- Diseño e implementación secuencial del algoritmo evolutivo en C++.
- Diseño e implementación paralela del algoritmo evolutivo en CUDA/ C++.

El módulo de Shunting Yard

Se utiliza el mismo módulo tanto en la implementación secuencial como en la paralela. Este módulo lee desde un archivo llamado `funcion.txt`, la función de evaluación que el usuario ingreso previamente. En la implementación secuencial este módulo contiene una función que se encarga de devolver los valores de $f(x_i)$ y $f(x_i + h)$ cada vez que se llama en la función de evaluación, en la implementación paralela esta misma función se adaptó para que fuese compatible con la GPU y así lograr que el proceso de la evaluación de la función sea paralelo en cada hilo en ejecución.

6.1. Diseño del algoritmo evolutivo secuencial

6.1.1. Generación y representación de individuos

La codificación del genoma es fundamental en un problema de algoritmos genéticos. En nuestro algoritmo genético, el número de genes es igual al número de variables declaradas en la función de evaluación con sus correspondiente factores de mutación. Un cromosoma está definido como:

$$< X_1, ..., X_n, \sigma_1, ..., \sigma_n > \tag{6.1}$$

donde $\overline{X}$ es un vector de números reales que representa los valores de cada gen en el cromosoma. y $\overline{\sigma}$ es un vector de números reales que representa el rango de exploración del operador de mutación. La generación de la población en el algoritmo se asignan valores aleatorios a cada uno de los cromosomas.

6.1.2. Mutación y reproducción

De manera general, la operación de mutación en el algoritmo se describe como:

$$\begin{aligned} X_i' &= X_i + \sigma_i'.N_i(0,1) \\ \sigma_i' &= \sigma_i.(1 + \alpha.N(0,1)) \end{aligned} \tag{6.2}$$

donde X_i' y σ_i' representan la descendencia de cada individuo. $N_i(0,1)$ representa el i-ésimo valor de un vector N de variables aleatorias con distribución normal (media cero y desviación estándar de 1) y N(0, 1) denota un número aleatorio de distribución normal.

6.1.3. Evaluación de la función objetivo

Este módulo mide la adaptación de cada individuo, este módulo consume la mayor parte del tiempo computacional. Se definieron tres funciones de evaluación diferentes para la implentación, éstas son:

$$X_1{}^2 + X_2{}^2 + X_3{}^2 + \cdots + X_n{}^2 \tag{6.3}$$

$$75(X_1 - X_2{}^2) + 75(X_2 - X_3{}^2) + \cdots + 75(X_{n-1} - X_n{}^2) \tag{6.4}$$

$$10\sin(X_1 + 10) + 10\sin(X_2 + 10) + \cdots + 10\sin(X_n + 10) \tag{6.5}$$

6.1.4. Competencia y selección

Este módulo selecciona un porcentaje aleatorio de la población, tanto como de la población de los padres como la de los hijos, este porcentaje no será menor a 30 % ni mayor a 70 %. Después se hace la competencia uno a uno, de manera que el individuo con mejor evaluación (la más cercana a 0) pasa a la siguiente ronda, esta comparación es hecha tomando el valor absoluto de ambas evaluaciones de los individuos. Esto se hace así en caso de que alguna evaluación sea de valor negativo. Después de 2 rondas, los individuos ganadores se incorporan a la población original. Si éste pertenecía a la población de los hijos, es decir la población mutada, entonces se sustituye según su índice en la población original.

6.2. Paralelización del algoritmo evolutivo

En la ejecución de cada módulo, cada vez que se manda a llamar una función a ejecutar en la GPU, ésta se divide en bloques, dependiendo del tamaño de la población se crearán tantos bloques como sea necesario, con 16 hilos en cada bloque. El por qué utilizar un grupo de 16 hilos o un *half-warp* es por la capacidad de la arquitectura para obtener 16 palabras de memoria simultáneamente en un solo ciclo de reloj por medio de los 16 hilos. Se realiza de este modo la ejecución de los *kernel* por cuestión de eficiencia, aunque los patrones dependen de la capacidad de cómputo de la GPU, generalmente son multiplos de 16. El agrupamiento de hilos en *warps* es importante para los accesos de memoria global, con el fin de realizar las menos transacciones como sea posible, ya que reduce tiempo y evita un cuello de botella al realizar númerosos accesos a memoria.

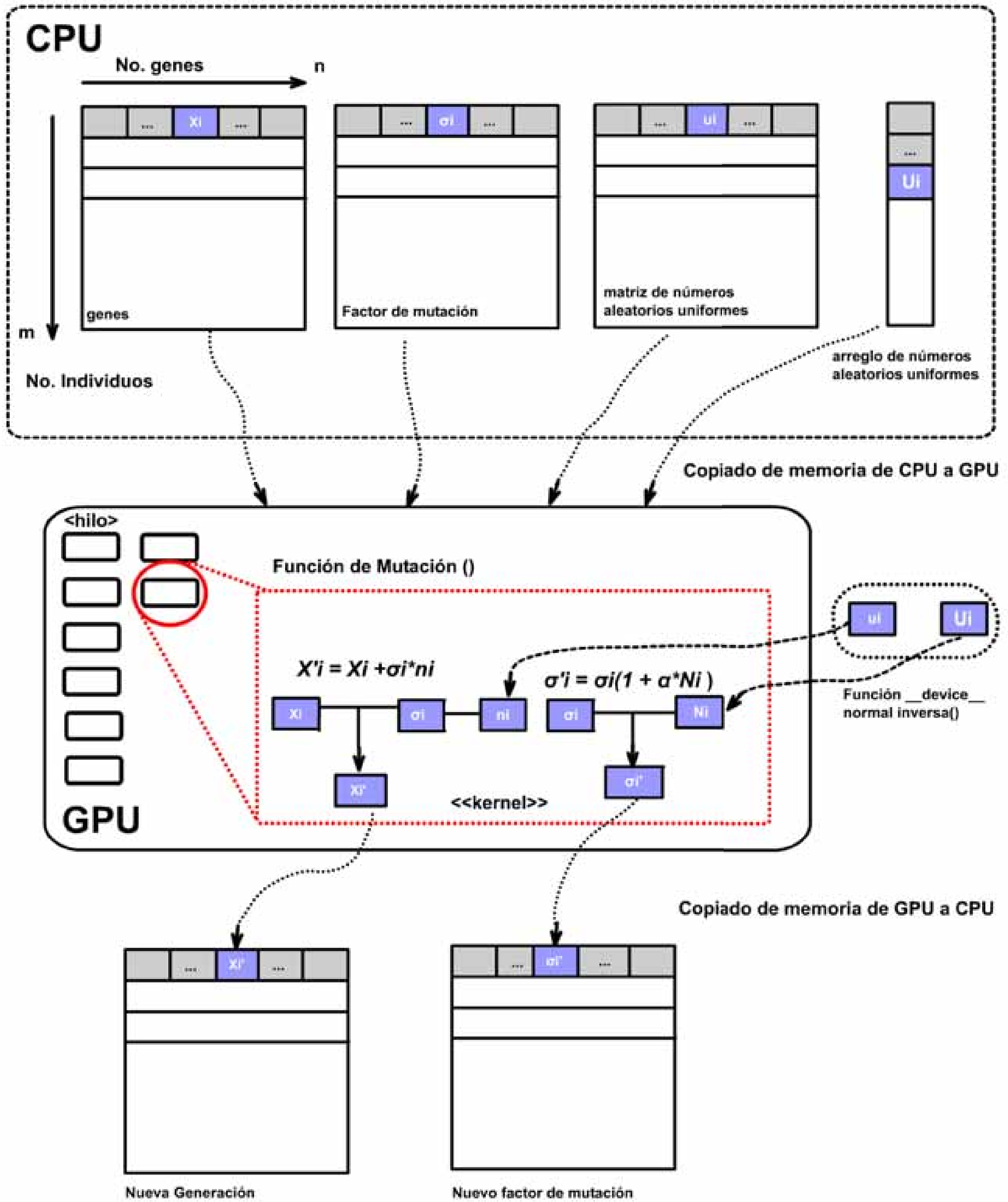
6.2.1. Generación y representación de individuos

Al adaptar el algoritmo secuencial a la plataforma CUDA en paralelo fue necesario hacer compatibles los datos de la implementación secuencial con datos GPU. Esto es, una representación matricial de arreglos de memoria dinámica. Una estrategia utilizada para los datos fueran compatibles entre la CPU y la GPU, fue separar los genes del cromosoma en 2 matrices, una para $\overline{X}$ y otra para $\overline{\sigma}$, de esta manera el manejo de datos entre la GPU y CPU es más sencillo. Una de las limitantes de la GPU es que no tiene capacidad de generar números aleatorios, de manera que la CPU se encarga de la generación de población en ambas implementaciones.

6.2.2. Módulo de Mutación

El proceso de mutación en el módulo en paralelo se realiza de la siguiente manera, se envía una copia de las matriz $\overline{X}$ y de la matriz $\overline{\sigma}$, una matriz $U_i(0,1)$ y un arreglo $u(0,1)$ con números aleatorios uniformes. Como se había mencionado anteriormente la GPU no puede generar números aleatorios, en la implementación secuencial estos números aleatorios son calculados en el momento del proceso de mutación, en la implementación paralela se envía la matriz $U_i(0,1)$ y el arreglo $u(0,1)$ generados en la CPU, en la misma función de mutación dentro de la GPU se normalizan estos números aleatorios uniformes llamando a una función de tipo *device* llamada normal inversa, logrando de esta manera el proceso de normalización en paralelo. Se utiliza la función de distribución normal acumulativa inversa por que su cálculo no requiere de realizar iteraciones. El proceso de mutación es realizado por cada gen, es decir, se lanzaron tantos hilos como genes hay en la población total, de manera que cada hilo se es responsable de mutar un gen del cromosoma (véase Figura 6.1).

Figura 6.1: Funcionamiento del módulo de mutación en paralelo



6.2.3. Módulo de Evaluación

La estructura de datos de cola (la cual guarda en notación polaca inversa la función de evaluación), es manejada por una biblioteca estandar `queue.h` de C++, la cual no permite acceder a las funciones de la clase ya que están definidas en el *host* y no en la plataforma, por lo que estas funciones no pueden ser llamadas desde el dispositivo GPU. Para poder acceder a estos métodos se tendría que redefinir la biblioteca, una solución más simple a este problema fue crear un arreglo de estructuras que contuviera la misma información en variables de datos de tipo simple de memoria estática.

De manera similar al módulo de mutación, se envía una copia de las matrices de la población original y su descendencia, además de un arreglo que contiene la notación polaca inversa de la función a evaluar en este módulo. Por ejemplo la ecuación 6.3 en notación polaca inversa es: $X_1 2 \wedge X_2 2 \wedge + X_3 2 \wedge + \cdots + X_{100} 2 \wedge$.

Originalmente es una cola pero por la arquitectura de CUDA, ésta fue convertida en un arreglo de estructuras para que pueda ser procesado correctamente por la GPU. En el mismo *kernel* se calcula la evaluación por individuo de la población original y su descendencia al mismo tiempo. Estas evaluaciones son guardadas de manera parcial en una variable temporal que almacena la suma de las derivaciones parciales de cada variable para obtener la derivada de la función de varias variables y poder encontrar su mínimo con ese conjunto de genes (véase Figura 6.2.3).

La derivación parcial se cálcula a partir de una función de tipo *device* en la GPU, esta función recibe la cola, y la matriz de genes, además de parametros adicionales como el desplazamiento en la matriz del individuo (refiriéndonos a si los genes pertenecen al individuo 1, individuo 2, etc.), h es un número pequeño el cual se usa para el cálculo de la derivada, el índice de la variable a derivar y el tamaño de la cola.

Esta función regresa el valor de $f(x_i)$ y $f(x_i + h)$, que en el módulo de evaluación se encarga de operar ambos resultados para poder cálcular $f'(x_i) = \lim_{h \to 0} \frac{f(x_i+h)-f(x_i)}{h}$.

El tamaño de cola es utilizado como el número máximo de iteraciones que deben realizarse para obtener el valor de la derivada parcial. En la función se almacena una variable local, un arreglo llamado pila de tipo flotante, el cual simula una pseudo pila. La simulación de la pila se logra a través de funciones tipo *device* push (insertar) y pop (sacar), las cuales reciben una variable entera llamada pos_pila, la cual se incrementa o decrementa según la operación realizada en la pila, esta variable pos_pila guarda la posición actual del ultimo valor en entrar o salir de la pila. Esta pila almacena los resultados parciales de las operaciones realizadas que se van leyendo de la cola con la función de evaluación, sustituyendo los valores de las variables según sea necesario.

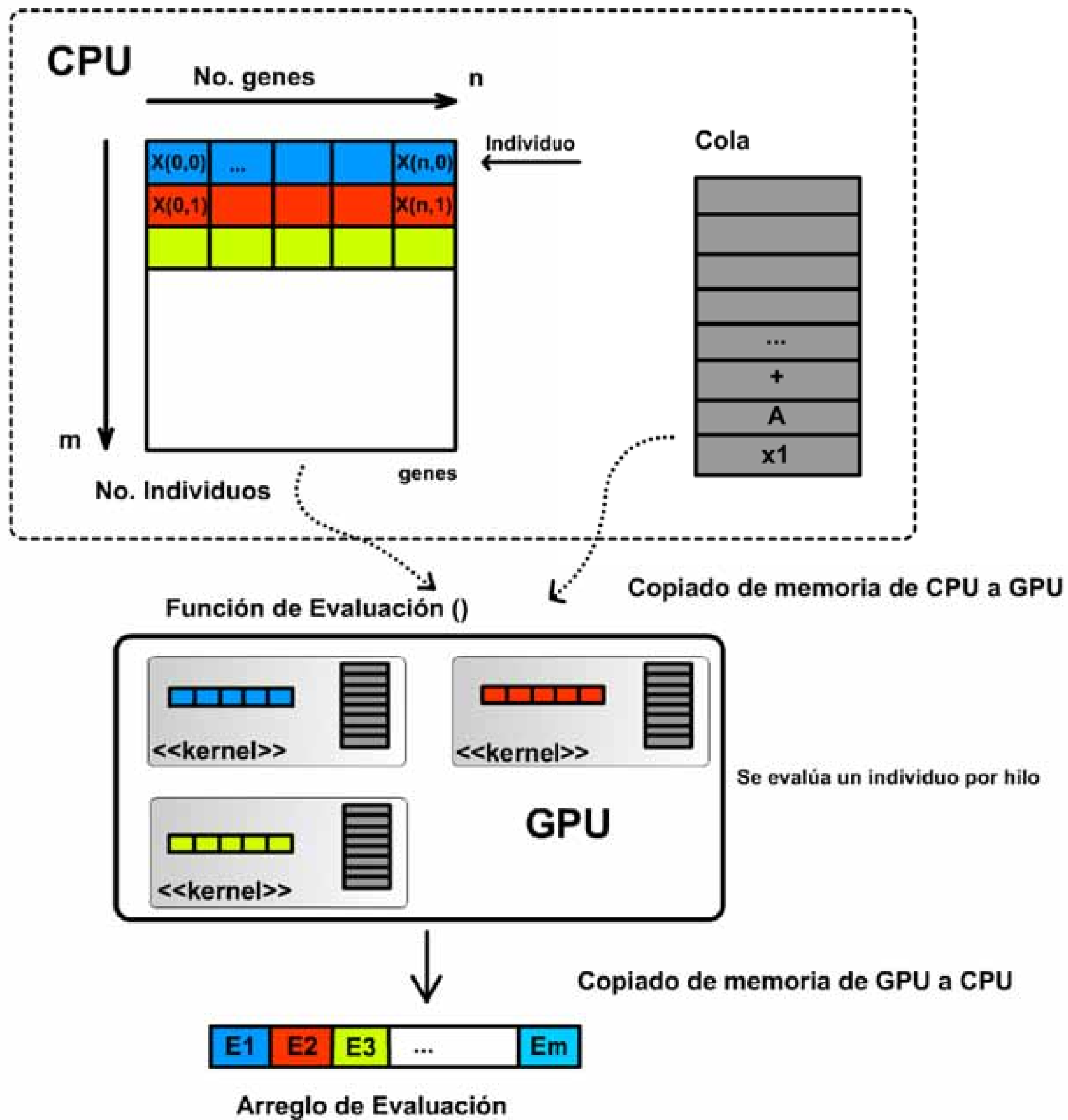


Figura 6.2: Funcionamiento del módulo de evaluación en paralelo.

6.2.4. Competencia y selección

El módulo de competencia y selección no se modificó ya que el tiempo de ejecución es inferior a milisegundos en todas las pruebas, de manera que no tiene caso paralelizarlo ya que no habría una ganancia significativa.

Capítulo 7

Resultados y conclusiones

7.1. Sistemas de cómputo.

Se utilizaron dos ambientes de pruebas, ambos contaban con procesadores convencionales y una tarjeta nVidia con GPUs. La Tabla 7.1 muestran las características de ambos.

	Sistema 1	**Sistema 2**
Procesador	Intel Core i3	Intel Xeon
Gráficas	GeForce GT 630	Tesla M2090
RAM	2 G	24 G
Sistema operativo	Windogs XP SP3	Winbugs Multipoint Server

Tabla 7.1: Sistemas de pruebas.

En la Tabla 7.1 se muestran las características de las tarjetas nVidia utilizadas.

	GeForce GT 630	**Tesla M2090**
Núcleos CUDA	96	512
Frecuencia de reloj de memoria	900 MHz	1.85 GHz
Frecuencia de reloj de GPU	810 MHz	1.3GHz
Capacidad de memoria	1Gb GDDR5	6 Gb GDDR5
Ancho de banda máximo	51.02 Gb/s	177 Gb/s
Rendimiento de punto flotante	311 GFLOPS	1331 GFLOPS

Tabla 7.2: Tarjetas nVidia utilizadas.

La versión de CUDA para el desarrollo en paralelo utilizada fue la 4.0., el ambiente de desarrollo usado fue Micosoft Visual Studio 2010 Express para los algoritmos en paralelo y Code Blocks con MinGW para el secuencial.

7.2. Experimentos

Se realizaron distintas pruebas de desempeño con diferentes tamaños de población inicial, probando con las 3 ecuaciones propuestas. Usando tamaños de población de 500, 1000, 5000 y 10000 para tomar tiempos promedio de cada uno de los módulos con pocas generaciones.

Para hacer una comparación más exhaustiva de la disminución de tiempo entre implementaciones del secuencial contra el paralelo, se creó un programa especial que iterara 900 generaciones y 5000 individuos en la población, aunque no se pudo terminar la ejecución del programa debido a fallas con la energía electrica en el servidor, solo se pudo obtener resultados de 400 generaciones.

Un problema que se presentó en la primera versión del módulo de evaluación en el ambiente de desarrollo, fue al crear poblaciones con más de 4800 individuos, lo cual provocaba un error código 6 en CUDA. Este error indica que el *kernel* tardo más tiempo en ejecutarse que el intervalo de espera máximo, resultando en la terminación de la ejecución del módulo como medida de seguridad.

La solución a este problema fue envíar partes de la población a la GPU y copiar los resultados parciales correspodientes en el arreglo de evaluacion de población. Lo cual causó un ligero aumento en el tiempo de ejecución del modulo, pero de cualquier manera la reducción del tiempo del programa secuencial a paralelo seguía siendo considerable.

De manera que la versión del módulo de evaluación que se utilizó en la GPU GT 630 es diferente a la que se utilizó en la GPU Tesla M2090, en la cual se ejecutó la versión original de la implementación sin problema alguno.

En la GPU Tesla M2090 no tiene este problema debido a que su poder de procesamiento es mayor que la GPU utilizada en la fase de desarrollo, aunque tiene la misma medida de seguridad que la GPU GT 630, ésta no llega al limite.

7.3. Resultados

Como se puede apreciar en las Figuras 7.1, 7.2 y 7.3 hay una reducción significativa en los tiempos de ejecución en paralelo, en particular se muestran los resultados de los tiempos promedio que tarda en procesar una generación (iteración) con distintos valores del tamaño de población de las 3 ecuaciones (6.3) (6.4) (6.5) . En las gráficas se puede observar la diferencia de tiempos de procesamiento entre la CPU y la GPU, especialmente se puede notar que en la GPU Tesla M2090 (la cual es utilizada para cómputo científico) la gran reducción del tiempo de procesamiento.

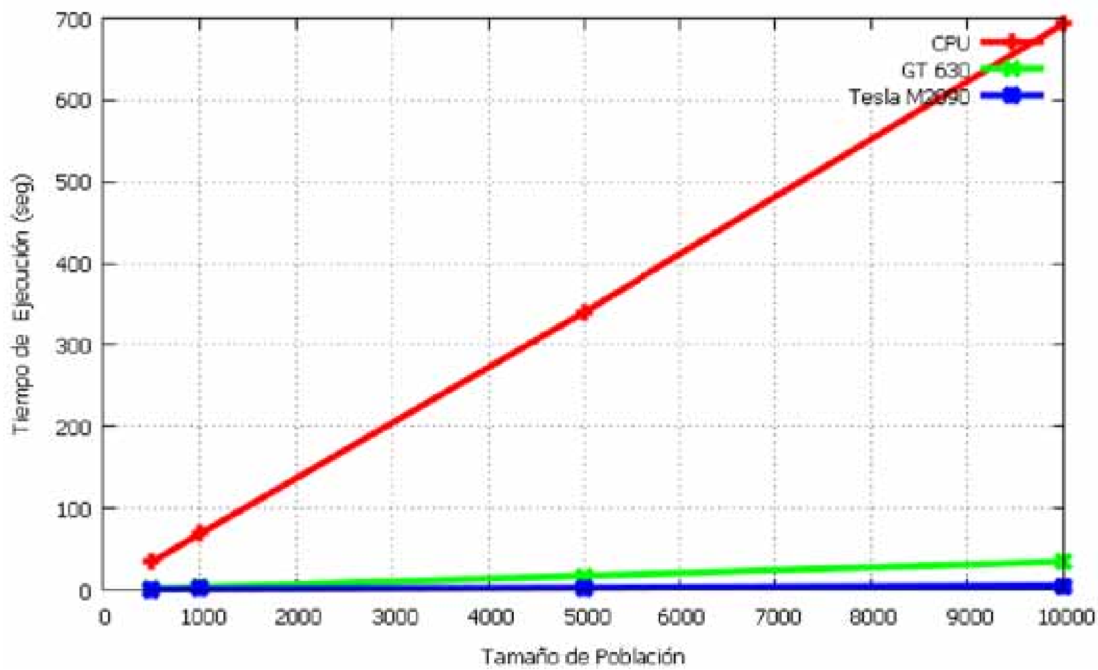


Figura 7.1: Gráfica comparativa de la ecuación (6.3)

Tamaño de Población	CPU (seg)	GPU GT 630 (seg)	GPU Tesla M2090 (seg)
500	34.072	1.73754	1.0495422
1,000	68.579	3.46759	1.1115833
5,000	340.563	17.2449	2.61937731
10,000	693.546	34.55	5.07180784

Tabla 7.3: Tiempos promedio de distintos tamaños de población de la ecuación (6.3)

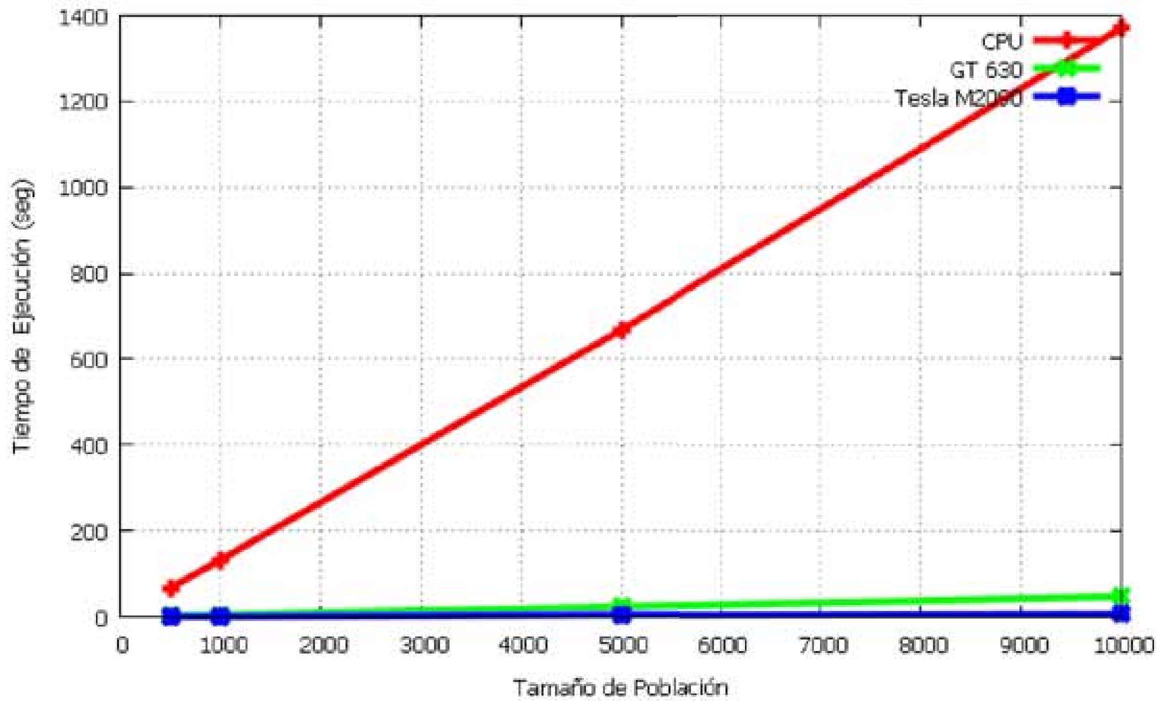


Figura 7.2: Gráfica comparativa de la ecuación (6.4)

Tamaño de Población	CPU (seg)	GPU GT 630 (seg)	GPU Tesla M2090 (seg)
500	67.469	2.37621	1.44172429
1,000	132.906	4.75172	1.52319408
5,000	668.14	23.7501	4.6080544
10,000	1370.642	47.4803	7.20389034

Tabla 7.4: Tiempos promedio de distintos tamaños de población de la ecuación (6.4)

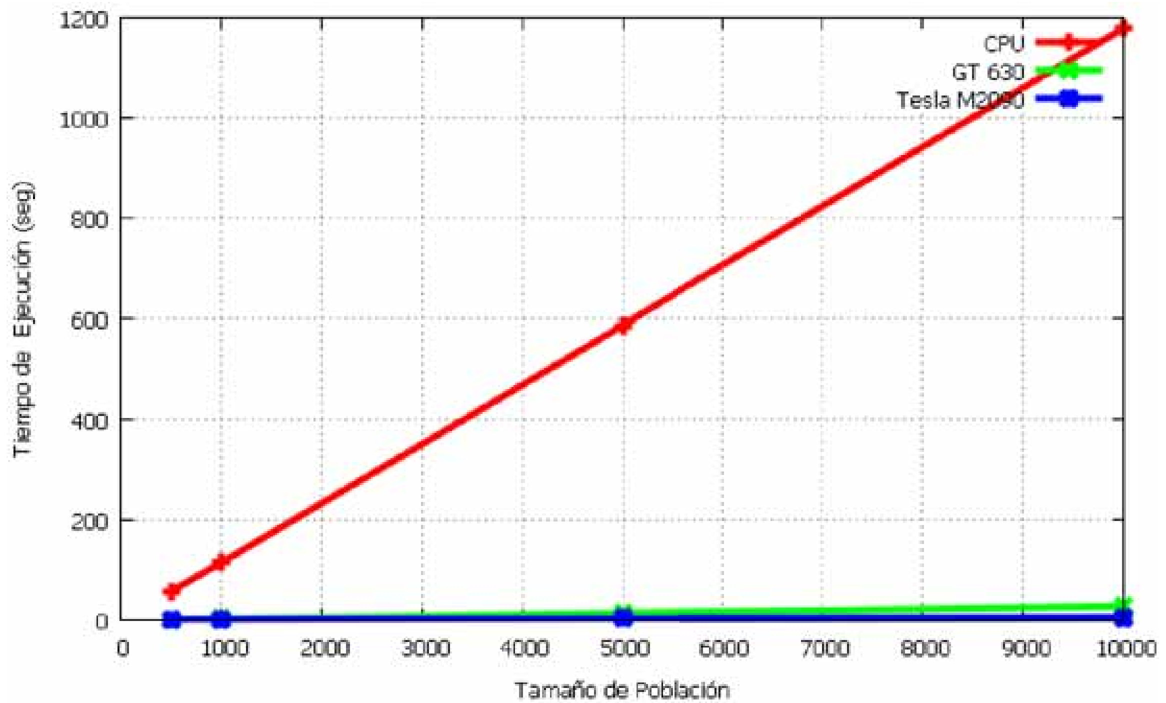


Figura 7.3: Gráfica comparativa de la ecuación (6.5)

Tamaño de Población	CPU (seg)	GPU GT 630 (seg)	GPU Tesla M2090 (seg)
500	57.687	1.35915	0.84191404
1,000	115.078	2.7144	0.99451504
5,000	587.391	13.5287	2.73158632
10,000	1178.607	27.0634	4.5692779

Tabla 7.5: Tiempos promedio de distintos tamaños de población de la ecuación (6.5)

Los siguientes resultados son la comparación de la ejecución de ambos algoritmos (Ver Figura 7.4), con un tamaño de población inicial de 5,000 individuos y 400 generaciones, utilizando la ecuación (6.3) , los tiempos son tomados cada 50 generaciones calculadas. Originalmente estaba planeado 900 generaciones pero por problemas técnicos en el servidor, solo se pudo completar un máximo de 400 generaciones en la implementación secuencial, en la implementación paralela se obtuvieron los tiempos resultantes de las 900 generaciones.

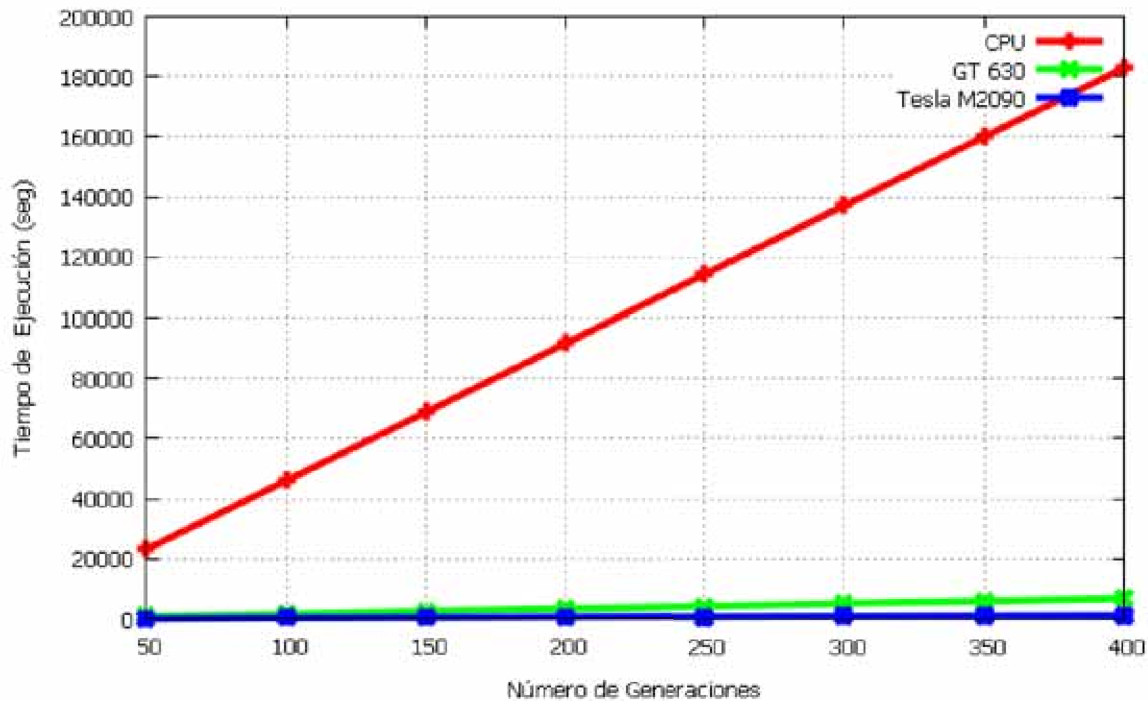


Figura 7.4: Gráfica comparativa entre el algoritmo secuencial y el paralelo usando la ecuación (6.3)

No. de generaciones	Secuencial CPU servidor(seg)	Paralelo GT 630 (seg)	Paralelo Tesla M2090 (seg)
50	23244.50	879.534	15.918
100	46034.10	1740.6	316.238
150	68834.00	2602.3	473.223
200	91647.80	3464.13	630.292
250	114466.00	4325.56	787.241
300	137281.00	5187.32	944.27
350	160097	6066.854	1101.33
400	182921	6927.92	1258.25

Tabla 7.6: Comparativa de desempeño entre el algoritmo secuencial y el paralelo de la ecuación (6.3)

7.3.1. Factor de aceleración

El beneficio potencial del cómputo en paralelo se mide por el tiempo que se tarda para completar una tarea en un solo procesador en comparación con el tiempo que se tarda en completar la misma tarea en N procesadores en paralelo. El aumento de velocidad debido a la utilización de N procesadores en paralelo se define por:

$$Factor_{acceleracion} = \frac{Tiempo_{secuencial}(1)}{Tiempo_{paralelo}(N)} \tag{7.1}$$

donde $Tiempo_{secuencial}(1)$ es el tiempo de procesamiento en solo procesador y $Tiempo_{paralelo}(N)$ es el tiempo de procesamiento en los N procesadores en paralelo[7], de manera que haciendo un promedio con los datos obtenidos, se obtiene un factor de aceleración 145.498314 para la ecuación (6.3) en comparación con la CPU del servidor y la GPU Tesla M2090 con 5,000 individuos y 400 generaciones y un factor de aceleración de 26.437822 con la GPU GT 630 utilizada en el ambiente de desarrollo.

A continuación se presentan los factores de aceleración para las pruebas con distintos tamaños de población de la 3 ecuaciones (6.3) (6.4) (6.5):

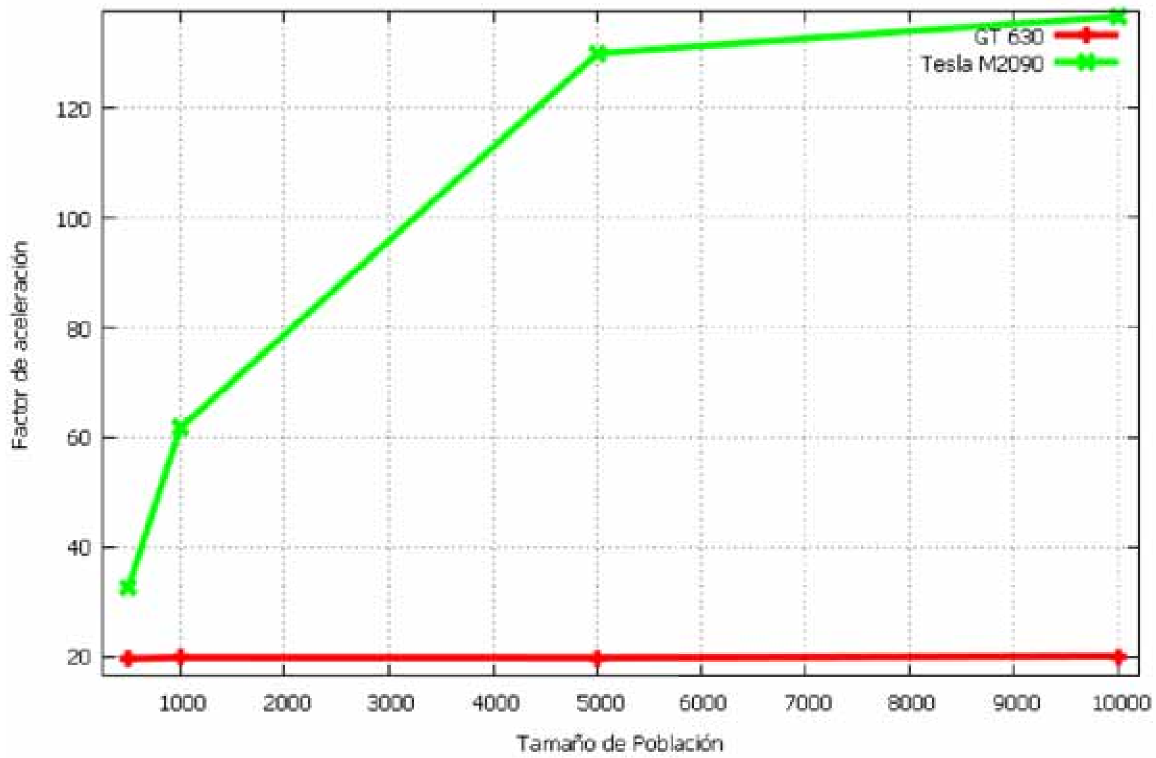


Figura 7.5: Gráfica comparativa entre el factor de aceleración entre GPU utilizando la ecuación (6.3)

Tamaño de Población	Factor de acceleración en GT 630	Factor de acceleración en Tesla M2090
500	19.6093327348	32.4636779731
1,000	19.777136282	61.6948815262
5,000	19.7486213315	130.0167786824
10,000	20.0736903039	136.7453227487

Tabla 7.7: Factor de aceleración en las pruebas con distintos tamaños de población de la ecuación (6.3)

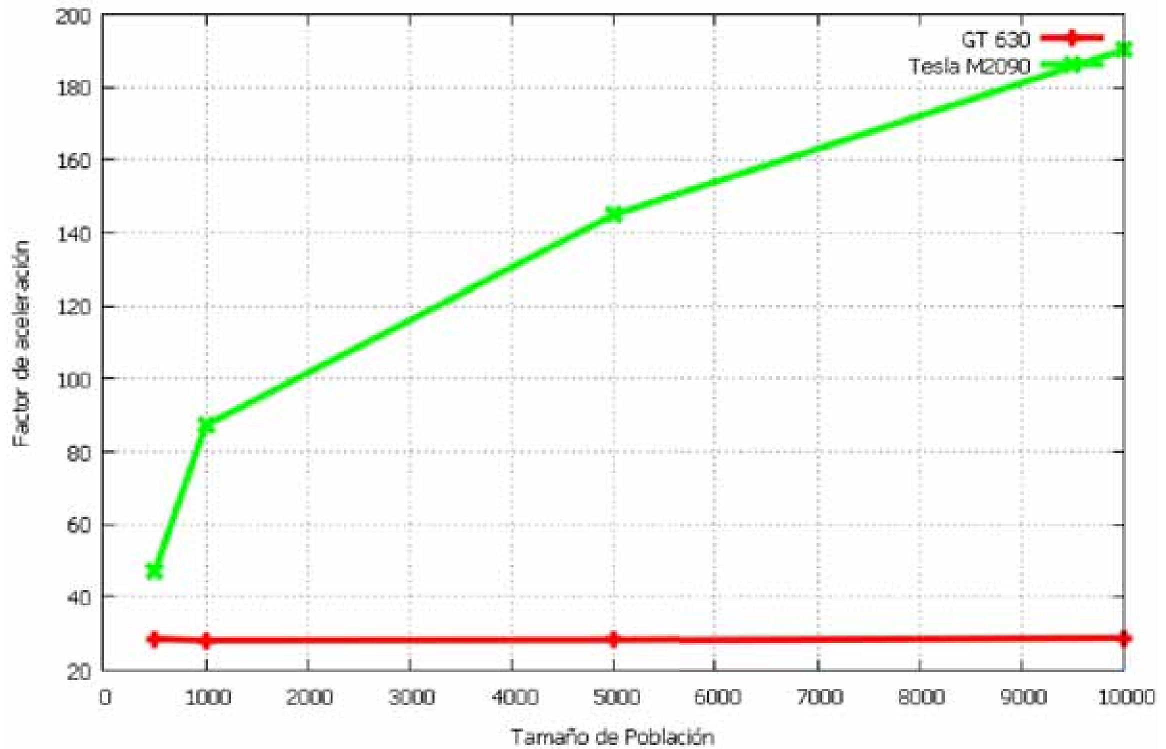


Figura 7.6: Gráfica comparativa entre el factor de aceleración entre GPU utilizando la ecuación (6.4)

Tamaño de Población	Factor de acceleración en GT 630	Factor de acceleración en Tesla M2090
500	28.3935342415	46.7974358676
1,000	27.9700824123	87.2548034063
5,000	28.1320920754	144.9939479881
10,000	28.8675935072	190.264139973

Tabla 7.8: Factor de aceleración en las pruebas con distintos tamaños de población de la ecuación (6.4)

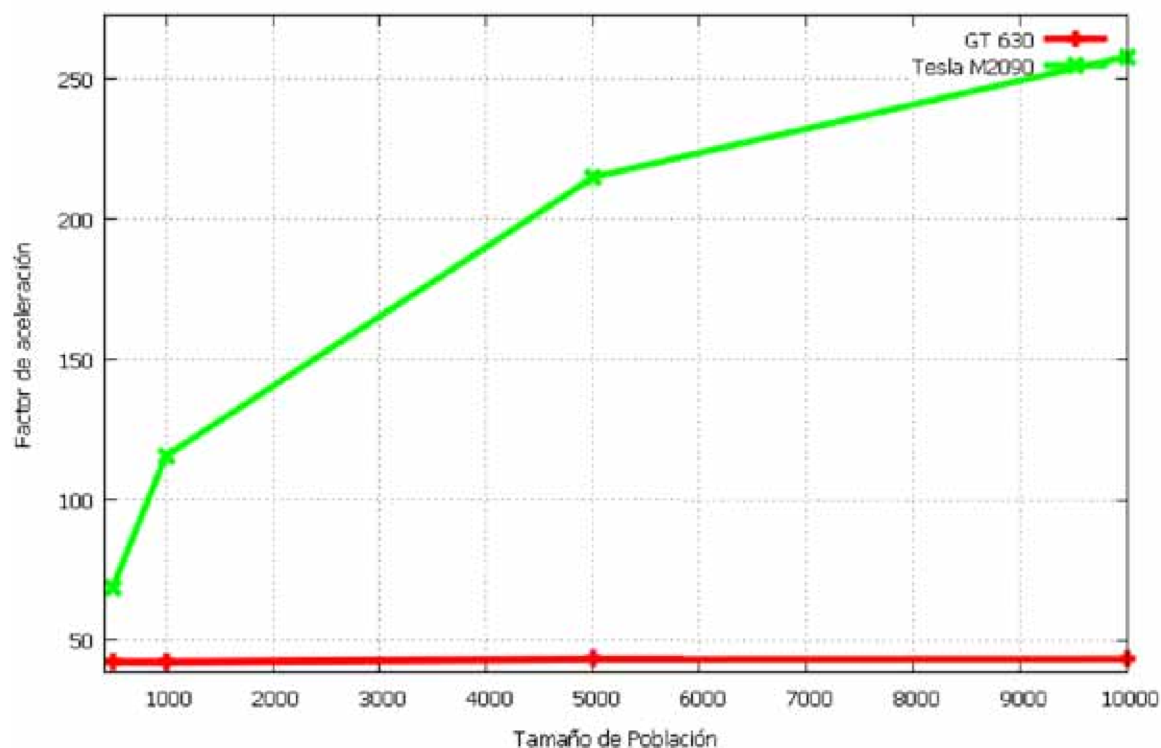


Figura 7.7: Gráfica comparativa entre el factor de aceleración entre GPU utilizando la ecuación (6.5)

Tamaño de Población	Factor de acceleración en GT 630	Factor de acceleración en Tesla M2090
500	42.443438914	68.5188715941
1,000	42.3953728264	115.7126794181
5,000	43.4181406935	215.0365872384
10,000	43.5498496124	257.9416323091

Tabla 7.9: Factor de aceleración en las pruebas con distintos tamaños de población de la ecuación (6.5)

Como se puede apreciar los valores de factor de aceleración para la GPU GT 630 hay una diferencia poco significativa en comparación con los factores de aceleración obtenidos en la GPU Tesla M2090 los cuales van en aumento conforme el tamaño de población crece.

7.4. Conclusiones

La ventaja que ofrece CUDA como lenguaje de programación de alto nivel, es la accesibilidad del desarrollo de aplicaciones en la GPU, la cual puede ejecutar miles de subprocesos (hilos) de forma simultánea para procesar en paralelo problemas matemáticos de alta carga computacional a gran velocidad. Esto posibilita la integración de la CPU-GPU en la creación de aplicacións que puedan explotar al máximo las ventajas de ambas arquitecturas.

El uso de la GPU para aumentar la eficiencia en el procesamiento computacional es superior por la filosofía con la cual fue diseñada su arquitectura, aunque está limitado en cuanto al tipo de datos que pueden ser envíados. Lo cual puede representar un problema de compatibilidad entre el *host* y la plataforma. Es importante que al desarrollar aplicaciones en paralelo y en especial si es una migración de una aplicación secuencial, tener en cuenta que las limitaciones de CUDA incluyen:

- No recursividad.
- Punteros a funciones.
- Variables estáticas dentro de funciones.
- Funciones con número de parámetros variable.
- Números desnormalizados o NaNs (*Not a Number*).

Ciertas características, por ejemplo la recursión, en tarjetas GPU de generación Fermi es soportada bajo ciertas condiciones, como usar la recursión solo en funciones de tipo *device*. Estas limitantes no son tan relevantes cuando se usa la ventaja más notable de la GPU, que es el parelelismo masivo para el cálculo aritmético intenso. Muchos problemas computacionales pueden obtener una mejoría notable en la eficiencia de procesamiento y adaptarse perfectamente al estilo de cómputo de GPU.

Bibliografía

[1] Mikko H. Lipasti John Shen. *Modern Processor Design: Fundamentals of Superscalar Processors*. Editorial Springer, 2011.

[2] Wen-mei W. Hwu David B. Kirk. *Programming Massively Parallel Processors A Hands-on Approach*. Editorial Morgan Kaufmann, 2010.

[3] Peter Baer Galvin Abraham Silberschatz, Greg Gagne. *Operating System Concepts*. Editorial Wiley, 2012.

[4] András Vajda. *Programming Many-Core Chips*. Editorial Wiley, 2011.

[5] Alan Tatourian. *NVIDIA GPU Architecture and CUDA Programming Environment*. 2013. [En Línea] Disponible en: http://code.msdn.microsoft.com/vstudio/NVIDIA-GPU-Architecture-45c11e6d.

[6] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2011. [En Línea] Disponible en: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[7] Fayez Gebali. *Algorithms and Parallel Computing*. Editorial Wiley, 2011.

[8] Edward Kandrot Jason Sanders. *CUDA by example: an introduction to general-purpose GPU programming*. Editorial Addison Wesley, 2011.

[9] Shane Cook. *CUDA Programming A Developers Guide to Parallel Computing with GPUs*. Editorial Morgan Kaufmann, 2011.

[10] Jürgen Branke. *Evolutionary Optimization in Dynamic Environments*. Editorial Springer, 2002.

Apéndice A

Código fuente

A.1. Bibliotecas comunes usadas en ambas implementaciones.

Archivo file.h, carga la función en un string para poder ser procesado posteriormente en la implementación del shunting yard.

```
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <string.h>
#include <sstream>


using namespace std;

string archivo() {

  ifstream file;

  stringstream ss;

  string line;

  file.open ("funcion.txt",ios::in);
  //Lectura del archivo  con la funcion de evaluacion
 if (file.is_open())
  {
    while ( file.good() )
    {
      getline (file,line);
    }
    file.close();
    return line;
  }

  else cout << "Error al abrir el archivo...";

  return 0;

}
```

src/secuencial/file.h

Archivo shun-yard.h, se encarga de analizar las ecuaciones y las guarda en una pila en notación polaca inversa.

```
#include<iostream>
#include <stdlib.h>
#include<string.h>
#include<sstream>
#include<cmath>
#include<stack>
#include<queue>
#include <vector>
#include "file.h"

#define PI 3.14159265

using namespace std;

bool is_func(string);
bool is_op(string);
bool is_left(char);
bool is_var(string);

int get_prec(char);


 float operate(float, string);
 float operate(float, float, char);




typedef std::vector< std::vector<float> > matriz;

struct torneo {

    int indice;//indice del individuo en poblacion
    float val;//valor de la evaluacion
    bool p;//si de poblacion prima

void inicio(int n){indice = n;  p=false;};

    };



  struct output_data {

      bool is_var;// es variable
      int ind; // indice de variable

      float num;// num cte
      string opf; // operador - funcion
      bool is_num;//es constante

  // guarda numero
  output_data(float n){ num = n; is_num = true; is_var = false; ind=0; opf =" ";}
  //guarda funcion u operador
  output_data(string o) { opf = o; is_num = false; is_var = false; ind=0;  num =0;}
  //guarda una variable
  output_data(bool x,int i) { ind = i; is_var = true; is_num = true; opf =" "; num =0;}
  output_data(){};
  };

   struct data {

      bool is_var;// es variable
      int ind;  // indice de variable

      float num;// num cte
      char opf[6]; // operador - funcion
      bool is_num;//es constante

  //copiado de datos de la estructura outpu_data
  data(struct output_data d){
  num = d.num;
  is_num = d.is_num;
  is_var = d.is_var;
  ind=d.ind;

  std::copy( d.opf.begin(), d.opf.end(), opf);

   opf[d.opf.size()]='\0';

  }

  data(){};
  };

   queue<output_data> parser_file(int &ind) {


```

```
89        ind = 0;
90        string in = archivo();
91
92         stringstream str (in);
93         queue<output_data> output;
94
95         stack<string> opfs;
96         string part;
97
98         while(str >> part) {
99
100       //variables y constantes
101
102             if(part==" "){
103               //ignorar espacios
104             }  else if(!is_func(part) && !is_op(part)) {
105
106                     if(!is_var(part)){//es una constante
107
108                     output_data temp ((float)atof(part.c_str()));
109                     output.push(temp);
110
111                     }else{//es variable
112
113                     string aux_ind = part.substr (1,part.length()-1);
114                     output_data temp (true,(int)atoi(aux_ind.c_str()));
115                     output.push(temp);
116
117                     if(ind < (int)atoi(aux_ind.c_str()))
118
119                     ind = (int)atoi(aux_ind.c_str());
120
121                         }
122
123                }
124
125         else if(is_func(part)) {//funcion
126                 opfs.push(part);
127        } else {
128    //operadores
129           char op = part[0];
130           int prec = get_prec(op);
131           bool left = is_left(op);
132
133        if(op != '(' && op != ')') {
134
135            while(!opfs.empty() && (left && prec <= get_prec(opfs.top()[0])) ||
136           (!opfs.empty()&& (!left && prec < get_prec(opfs.top()[0]))) )
137      {
138
139               output.push(opfs.top());
140                opfs.pop();
141      }
142         string temp (1, op); opfs.push(temp);
143    }
144
145         if(opfs.size() == 0 || op == '(') {
146             string temp (1, op);
147             opfs.push(temp);
148         }
149
150         if(op == ')') {
151
152             while(opfs.top()[0] != '(') {
153                    output.push(opfs.top());
154                    opfs.pop();
155         }
156
157         opfs.pop();
158
159             if(is_func(opfs.top())) {
160                     output.push(opfs.top());
161                    opfs.pop();
162                   }
163             }
164         }
165    }
166
167  while(!opfs.empty()) {
168    output.push(opfs.top());
169    opfs.pop();
170  }
171
172
173         return output;
174    }
175
176  /*****************Evalaucion de la cola************************/
177
```

```
178 float func_eval(queue<output_data> output,vector<float> x,int index, float h){
179
180 stack<float> eval; // guarda el resultado parcial
181
182   while(!output.empty()) {
183
184       output_data out = output.front();
185       output.pop();
186
187        if(out.is_num && !out.is_var) {
188        //es una constante meter el valor a pila
189                eval.push(out.num);
190
191
192        }else if(out.is_num && out.is_var){
193
194           //es una variable , sustituir y meter el valor a pila
195             if((out.ind-1)== index)
196               eval.push(x[out.ind-1]+h);//literal a derivar
197               else
198                eval.push(x[out.ind-1]);
199            }
200
201      else if(is_func(out.opf))  {
202    //funciones sin cos etc
203          float num = eval.top();
204          eval.pop();
205
206          float res = operate(num, out.opf);
207          eval.push(res);
208
209          }
210
211        else {// operaciones
212
213        float num1 = eval.top();
214        eval.pop();
215        float num2 = eval.top();
216        eval.pop();
217        float res = operate(num2, num1, out.opf[0]);
218        eval.push(res);
219
220
221          }
222
223        }
224
225       return eval.top();
226 }
227
228    //revisa si es variable
229
230     bool is_var(string check) {
231
232        if(check.length() > 1 && (check[0]=='x'  || check[0]=='X'))
233            return true;
234
235             return false;
236
237    }
238
239
240
241     //revisa funcion
242
243    bool is_func(string check) {
244
245        if(check == "sin" || check == "cos" || check == "log" || check == "e")
246        return true;
247
248        return false;
249    }
250
251
252         //revisa si es operador
253         bool is_op(string check) {
254
255             if(check.length() > 1)
256             return false;
257
258             switch(check[0]) {
259                  case '+':
260                  case '-':
261                  case '*':
262                  case '/':
263                  case '^':
264                  case '(':
265                  case ')':
266                   return true;
```

```
267                    default: return false; }
268                 }
269
270         bool is_left(char op) {
271
272              switch(op) {
273
274                  case '+':
275                  case '-':
276                  case '*':
277                  case '/':
278
279                  return true;
280
281                  default: return false;
282                  }
283             }
284
285         int get_prec(char op) {
286
287             switch(op) {
288
289                 case '^':
290
291                 return 2;
292
293                 case '*':
294                 case '/':
295
296                 return 1;
297
298                 case '+':
299                 case '-':
300
301                 return 0;
302
303                 default: return -1;
304                 }
305             }
306
307         //evalua una funcion
308         float operate(float num, string func) {
309
310             if(func == "sin")
311
312                 return sin(num * PI/180.0);
313
314             else if(func == "cos")
315
316                 return cos(num * PI/180.0);
317
318             else if(func == "log")
319
320                 return log10(num);
321
322            else if(func == "e")
323
324                 return exp (num);
325
326             return 0; }
327
328     //evalua una operacion
329     float operate(float num1, float num2, char op) {
330
331         switch(op) {
332             case '+':
333             return num1 + num2;
334
335             case '-':
336             return num1 - num2;
337
338             case '*':
339             return num1 * num2;
340
341             case '/':
342             return num1 / num2;
343
344             case '^':
345             return pow(num1, num2);
346              default: return 0;
347              }
348
349         }
```

src/secuencial/shun–yard.h

A.1.1. Bibliotecas Adicionales

La biblioteca que calcula la distribución normal inversa fue creada por Peter John Acklam, programada originalmente en Java, la versión en linea adaptada en Javascript fue por Alankar Misra, se adaptó para que compilara lenguaje C++.

Fuente: http://home.online.no/ pjacklam/notes/invnorm/impl/misra/normsinv.html

Archivo normsinv.h

```
#include <math.h>
#include <iostream>
#include <ctime>

using namespace std;
//distribucion normal inversa
float normsinv(double p)
{
    // coeficientes
    double a[] = {-3.969683028665376e+01,  2.209460984245205e+02,
                  -2.759285104469687e+02,  1.383577518672690e+02,
                  -3.066479806614716e+01,  2.506628277459239e+00};

   double  b[] = {-5.447609879822406e+01,  1.615858368580409e+02,
                  -1.556989798598866e+02,  6.680131188771972e+01,
                  -1.328068155288572e+01 };

    double  c[] = {-7.784894002430293e-03, -3.223964580411365e-01,
                  -2.400758277161838e+00, -2.549732539343734e+00,
                  4.374664141464968e+00,  2.938163982698783e+00};

   double  d[] ={7.784695709041462e-03, 3.224671290700398e-01,
                   2.445134137142996e+00,  3.754408661907416e+00};

    float plow  = 0.02425;
    float phigh = 1 - plow;

    // aproximacion racional para region inferior:
    if ( p < plow ) {
             double q  = sqrt(-2*log(p));
             return (((((c[0]*q+c[1])*q+c[2])*q+c[3])*q+c[4])*q+c[5]) /
                                   ((((d[0]*q+d[1])*q+d[2])*q+d[3])*q+1);
    }

    // aproximacion racional para region superior:
    if ( phigh < p ) {
             double q  = sqrt(-2*log(1-p));
             return -(((((c[0]*q+c[1])*q+c[2])*q+c[3])*q+c[4])*q+c[5]) /
                                    ((((d[0]*q+d[1])*q+d[2])*q+d[3])*q+1);
    }

    // Aproximacion racional para region central:
    float q = p - 0.5;
    float r = q*q;

    return (((((a[0]*r+a[1])*r+a[2])*r+a[3])*r+a[4])*r+a[5])*q /
                    (((((b[0]*r+b[1])*r+b[2])*r+b[3])*r+b[4])*r+1);
}
```

src/secuencial/normsinv.h

A.2. Programas principales

Archivo evolutivo_secuencial.cpp, es la implentación secuencial del algoritmo evolutivo.

Evolutivo secuencial

```
#include<iostream>
#include<stdlib.h>
#include <ctime>
#include <algorithm>

#include<cmath>
#include<stack>
#include<queue>


#include "shun-yard.h"
#include "normsinv.h"


const int tam_pob = 5000;
const int no_gen =400;

using namespace std;

//funcion de distribucion uniforme
float unifRand()
{
   // srand( time( NULL ) );
    return rand() / double(RAND_MAX);
}

class alg_evo{

public:

int tam;// numero de variables



   vector<float> temp_x,temp_f;

    matriz x,sigma;
    matriz x_p,sigma_p;

    float eval[tam_pob];
    float eval_p[tam_pob];

alg_evo(int); //genera poblacion
~alg_evo();

void eval_ind( queue<output_data> ecuacion);
void mut_rep();
void comp_sel();
void imp();
void print();
    };


void alg_evo::imp(){

      for(int j=0;j<tam_pob;j++){ // individuos

        if(eval[j]<1 && eval[j]>-1){
              cout<<j<<":: eval: "<<eval[j]<<endl;
              cout<<"individuo:\n";
              for(int i=0;i<tam;i++)
              cout<<x[i][j]<<"   ";

        cout<<endl<<"----------"<<endl;
        }
      }

    }

void alg_evo::print(){

      for(int j=0;j<tam_pob;j++){ // individuos
   for(int i=0;i<tam;i++){ //cromosomas

```

```
75          cout<<x[i][j]<<" ";
76
77            }
78            cout<<*";;"<<endl;
79
80          }
81
82      }
83
84  //constructor de la clase genera la poblacion inicial
85  alg_evo::alg_evo(int n){
86
87      tam = n;
88
89
90      srand( time( NULL ) );
91
92          for(int j=0;j<tam_pob;j++){ // individuos
93
94              for(int i=0;i<tam;i++){ //cromosomas
95
96
97                temp_x.push_back((float) pow(-1.0,(rand()%4))*rand()+rand()/100000.0);
98                temp_f.push_back((float)pow(-1.0,(rand()%4))*rand()/10000.0);
99
100
101             }
102
103             x.push_back(temp_x);
104             sigma.push_back(temp_f);
105
106             temp_x.clear();
107             temp_f.clear();
108
109         }
110
111
112       //  x_p=x;
113       //  sigma_p=sigma;
114
115
116 }
117
118 alg_evo::~alg_evo(){
119
120     x.clear();
121     sigma.clear();
122
123     x_p.clear();
124     sigma_p.clear();
125
126
127
128     }
129
130 void alg_evo::mut_rep(){
131 //para la implementacion paralela es necesario crear matriz (para N(0,1)para cromosomas)
132 //y un arreglo N(0,1) para sigma
133
134  float   ran_m = 0.2 * normsinv(unifRand());
135
136 //limpiar valores anteriores para nueva mutacion
137     x_p.clear();
138     sigma_p.clear();
139
140
141             temp_x.clear();
142             temp_f.clear();
143
144
145         float x_mu,fac_mu;
146
147  for(int j=0;j<tam_pob;j++){ // individuos
148
149                 ran_m =  normsinv(unifRand());
150
151             for(int i=0;i<tam;i++){ //cromosomas
152
153                 x_mu =(float)(x[i][j] +  normsinv(unifRand())* sigma[i][j]);
154
155                 fac_mu = (float)(sigma[i][j] * (1 + ran_m ));
156
157
158                   temp_f.push_back(x_mu);
159                   temp_x.push_back(fac_mu);
160
161             }
162
163             x_p.push_back(temp_x);
```

```
164             sigma_p.push_back(temp_f);
165 
166             temp_x.clear();
167             temp_f.clear();
168 
169         }
170 
171 
172     }
173 //*********************************/
174 
175 void alg_evo::eval_ind( queue<output_data> ecuacion){
176 
177 
178 float  eval_min,eval_min_p,h,k;
179 queue<output_data> cola = ecuacion;
180 
181     for( int i = 0;i < tam_pob;i++){//individuo
182 
183      eval_min_p=eval_min=0;// contendra la suma de la derivadas parciales
184 
185     //[cromosoma][#individuo] = individuo
186 
187  vector<float> crm1,crm2;
188 
189      crm1=x[i];//copia de cromosoma
190      crm2=x_p[i];
191 
192         for(int j=0;j<tam;j++){//cromosoma
193 
194             h = unifRand()/100.0; //limite
195             k = unifRand()/100.0;//limte para la cromosoma prima
196 
197             eval_min += ( func_eval(cola,crm1,j,h) - func_eval(cola,crm1,j,0.0))/h;//derivacion parcial y suma
198             eval_min_p += ( func_eval(cola,crm2,j,k) - func_eval(cola,crm2,j,0.0))/k;//derivacion parcial y suma
199 
200         }
201 
202         eval[i] = eval_min;//guardar eval
203         eval_p[i] = eval_min_p;
204     }
205 
206 }
207 
208 
209 void alg_evo::comp_sel(){
210 
211 //***comparacion**//
212 
213    srand( time( NULL ) );
214 
215 int k;
216 int q=((rand() % 100) + 1);
217 
218 
219     if(q<30)
220 
221         q+=30;
222 
223     else if(q>70)
224 
225         q-=30;
226 
227     q = tam_pob*q/100;
228 
229     if((q%2)==1)//que no queden impares los participantes del torneo
230 
231         q++;
232 
233 
234    torneo ronda_1[q];
235 //**********Inicio ronda_1***************/
236     for(k=0;k<q;k++){
237 
238        ronda_1[k].inicio(rand()%tam_pob);//asignacion de indice
239        bool padre =   rand()%2;
240       if(padre == false){
241 
242             ronda_1[k].p = true;
243             ronda_1[k].val = eval_p[ronda_1[k].indice];
244 
245             }else{
246 
247                  ronda_1[k].val = eval[ronda_1[k].indice];
248 
249                 }
250        }
251 
252 
```

```
253     torneo ronda_2[(q/2)];
254
255  for(k=0;k<(q/2);k++){
256
257 //   <      >
258 //  [o]   [ o+1]
259     if(ronda_1[k].val<ronda_1[k+1].val)
260
261            ronda_2[k]= ronda_1[k];
262
263            else
264
265            ronda_2[k]= ronda_1[k+1];
266
267        }
268
269
270  torneo ronda_3[(q/4)];
271
272  for(k=0;k<(q/4);k++){
273
274 //   <      >
275 //  [o]   [ o+1]
276     if(ronda_2[k].val<ronda_2[k+1].val)
277
278            ronda_3[k]= ronda_2[k];
279
280            else
281
282            ronda_3[k]= ronda_2[k+1];
283
284        }
285
286
287 //****seleccion***//
288
289 //*******insercion de los ganadores a la poblacion**************************************//
290 int i;
291
292             temp_x.clear();
293             temp_f.clear();
294
295  for(k=0;k<(q/4);k++){
296
297     if(ronda_3[k].p == true){
298
299       i = ronda_3[k].indice;
300
301
302       vector<float> tmp_x,tmp_f;
303
304         tmp_x =x_p[i];
305         tmp_f = sigma_p[i];
306
307         x[i]=tmp_x;
308         sigma[i]=tmp_f;
309
310         tmp_x.clear();
311         tmp_f.clear();
312
313         }
314
315  }
316
317
318
319 }
320
321 int main(){
322
323 int n;//numero de literales en la expresion
324
325 //cargar la cola con la expresion
326   double elapsed_secs,prom_mut,prom_eval,prom_comp;
327    clock_t begin,end,t_ini,t_fin;
328     queue<output_data> ecuacion = parser_file(n);
329
330    begin = clock();
331    t_ini=clock();
332     alg_evo prueba(n);
333
334      end = clock();
335      elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
336      cout<<"generacion: " <<elapsed_secs <<endl<<endl;
337      //inicializacion de tiempos promedios
338      prom_mut=prom_eval=prom_comp=0;
339
340   for(int g=0;g<no_gen;g++){
341
```

```
342       begin = clock();
343
344        prueba.mut_rep();
345
346        end = clock();
347
348       elapsed_secs = double (end - begin) / CLOCKS_PER_SEC;
349       prom_mut+=elapsed_secs;
350     //  cout<<"mutacion tiempo: " <<elapsed_secs<<endl <<endl;
351
352         begin = clock();
353
354        prueba.eval_ind(ecuacion);//<-
355         end = clock();
356
357       elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
358       prom_eval+=elapsed_secs;
359     //  cout<<"evaluacion tiempo: " <<elapsed_secs<<endl <<endl;
360
361          begin = clock();
362
363        prueba.comp_sel();//<-
364          end = clock();
365
366       elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
367       prom_comp+=elapsed_secs;
368    //   cout<<"Seleccion " <<elapsed_secs<<endl <<endl;
369
370      }
371      t_fin=clock();
372      elapsed_secs = double(t_fin - t_ini) / CLOCKS_PER_SEC;
373 cout<<"prom mut: "<<prom_mut/no_gen<<"  prom eval: "<<prom_eval/no_gen<<"  prom sel:"<<prom_comp/no_gen<<endl<<"
        tiempo total: "<<elapsed_secs;
374 // prueba.imp();
375 //prueba.print();
376 prueba.~alg_evo();
377   // system("pause");
378     return 0;
379 }
```

src/secuencial/evolutivo_secuencial.cpp

Archivo parTesla.cu, es la versión de la implentación paralela del algorirmo evolutivo para la Tesla M2090.

Evolutivo paralelo Tesla

```
#include <cuda.h>
#include <iostream>
#include <stdlib.h>
#include <ctime>
#include <algorithm>

#include<cmath>

#include "shun_yard.h"

#include<stdio.h>

 #define MAX 10

const int tam_pob = 5000;
const int no_gen =900;

int tam_crom;

 float *x, *sigma, *x_p, *sig_p, *eval, *eval_p;
 float *dev_x, *dev_sig, *dev_x_p, *dev_sig_p, *dev_eval,*dev_eval_p;

 int *dev_tam, *dev_t_cr, *dv_tmp; //tamaño cola - tamaño de cromosoma - tmp

 data *cola, *dev_cola;


float unifRand(){

    return rand() / double(RAND_MAX);

}

/*--------------FUNCIONES DEVICE ----------------------------------*/
   __device__   float operate_1(float num,  const char *func) {



           if(func[0] == 's' && func[1] == 'i' && func[2] == 'n' )

            return sinf(num * PI/180.0);

        else if(func[0] == 'c' && func[1] == 'o' && func[2] == 's'  )

            return cosf(num * PI/180.0);

        else if(func[0] == 'l' && func[1] == 'o' && func[2] == 'g'  )

            return logf(num);

       else if(func[0] == 'e')

            return expf (num);

            return 0; }
            //evaluates an operator.

 __device__   float operate_2(float num1, float num2, char  op) {

        switch(op) {
            case '+':
            return num1 + num2;

            case '-':
            return num1 - num2;

            case '*':
            return num1 * num2;

            case '/':
            return num1 / num2;

            case '^':
            return pow(num1, num2);
             default: return 0;
             }

```

```
78          }
79
80
81     __device__ bool func_op(const char *check) {
82
83          if(check[0] == 's' || check[0] == 'c' || check[0] == 'l' || check[0] =='e')
84          return true;
85      else
86          return false;
87      }
88
89
90  __device__ void push(float pila[],float val,int &pos_pila){
91
92
93      if(pos_pila==MAX-1)
94      {
95         // printf("Stack full\n");
96          return;
97      }else{
98
99
100         pos_pila++;
101         pila[pos_pila]=val;
102
103
104     }
105
106 }
107
108 __device__ float pop(float pila[],int &pos_pila)
109 {
110       float t;
111      if(pos_pila==-1)
112      {
113           //printf("Stack empty\n");
114           return 0;
115      }else{
116      t=pila[pos_pila];
117      pos_pila=pos_pila-1;
118      return t;
119      }
120 }
121
122
123 //dist normal inv
124 __device__ float normsinv(float  p) {
125
126      float a[] = {-3.969683028665376e+01,  2.209460984245205e+02,
127                        -2.759285104469687e+02,  1.383577518672690e+02,
128                        -3.066479806614716e+01,  2.506628277459239e+00};
129
130    float  b[] = {-5.447609879822406e+01,  1.615858368580409e+02,
131                        -1.556989798598866e+02,  6.680131188771972e+01,
132                        -1.328068155288572e+01 };
133
134     float  c[] = {-7.784894002430293e-03, -3.223964580411365e-01,
135                        -2.400758277161838e+00, -2.549732539343734e+00,
136                        4.374664141464968e+00,  2.938163982698783e+00};
137
138    float  d[] ={7.784695709041462e-03, 3.224671290700398e-01,
139                         2.445134137142996e+00,  3.754408661907416e+00};
140
141
142     // Define break-points.
143     float plow  = 0.02425;
144     float phigh = 1 - plow;
145
146
147     // Rational approximation for lower region:
148     if ( p < plow ) {
149
150              float q  = sqrtf ( -2* logf (p)) ;
151              return   (((((c[0]*q+c[1])*q+c[2])*q+c[3])*q+c[4])*q+c[5]) /
152                                         ((((d[0]*q+d[1])*q+d[2])*q+d[3])*q+1);
153     }
154
155     // Rational approximation for upper region:
156     if ( phigh < p ) {
157              float q  = sqrtf(-2*logf(1-p));
158           return      -(((((c[0]*q+c[1])*q+c[2])*q+c[3])*q+c[4])*q+c[5]) /
159                                          ((((d[0]*q+d[1])*q+d[2])*q+d[3])*q+1);
160     }
161
162     // Rational approximation for central region:
163     float q = p - 0.5;
164     float r = q*q;
165
166         return   (((((a[0]*r+a[1])*r+a[2])*r+a[3])*r+a[4])*r+a[5])*q /
```

```
167                             (((((b[0]*r+b[1])*r+b[2])*r+b[3])*r+b[4])*r+1);
168
169
170
171
172   }
173
174
175
176   __device__ float funcion(data *cola,float *x,int pos_individuo,
177             float h, int index, int *tam_q){
178
179
180  float pila[MAX],num,num1,num2,res;
181 int pos_pila=-1;
182  int i;
183
184     for(i=0; i<*tam_q;i++){//hasta llegar al final de la cola
185
186
187          if(cola[i].is_num && !cola[i].is_var) {//cte
188
189
190           push(pila, cola[i].num, pos_pila);
191
192       }else if(cola[i].is_num && cola[i].is_var){//var
193
194
195
196           int in_dev = ( pos_individuo) + (cola[i].ind-1);
197
198           if(index != (cola[i].ind - 1))
199
200           push(pila, x[in_dev ], pos_pila);
201
202           else{
203
204             float f = (x[ in_dev ] + h);
205
206           push(pila, f , pos_pila);
207           }
208             }
209
210       else if(func_op(cola[i].opf))  { //funciones sin cos etc
211
212           num = pop(pila,pos_pila);
213
214           // cout<<"-->>"<<num<<endl;
215
216            res = operate_1(num, cola[i].opf);
217
218           push(pila, res, pos_pila);
219           }
220
221       else {// operandos
222
223                 num1 = pop(pila,pos_pila);
224          //  cout<<"-->>"<<num1<<endl;
225
226          num2 = pop(pila,pos_pila);
227          //  cout<<"-->>"<<num2<<endl;
228
229
230             res = operate_2(num2, num1, cola[i].opf[0]);
231
232                 push(pila, res, pos_pila);
233
234
235           }
236
237   }//tamaño cola
238
239   return pila[pos_pila];
240
241
242
243
244   }//fin device
245
246 /*--------------FUNCIONES DEVICE FIN ----------------------------------*/
247
248
249 /*--------------FUNCIONES GLOBAL ----------------------------------*/
250 //Mutacion de individuos
251 __global__ void mutacion( float *x, float *sig, float *x_p,
252         float *sigma_p, float * ranUni , float * randUni2, int *tam_crom) {
253
254     //calculo de la posicion del hilo actual
255       int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

```
256
257     if (tid < (tam_pob*(*tam_crom))){
258
259         x_p[tid] = x[tid] + normsinv(ranUni[tid])*sig[tid];
260     sigma_p[tid]= sig[tid] * (1 +  normsinv(randUni2[(int)(tid/tam_pob)]) );
261
262     }
263 }
264
265 //Evaluacion de individuos
266
267 __global__ void eval_gpu(data *cola, float *x, float *x_p,
268         float *eval, float *eval_p, int *tam_q, int *tam_crom){
269
270 //calculo de la posicion del hilo actual
271  int tid = blockIdx.x * blockDim.x + threadIdx.x;
272
273  float eval_tmp=0;
274  float eval_tmp_p=0;
275
276  int j;
277  int pos_individuo = *tam_crom*tid ;
278
279   if(tid < tam_pob){
280
281
282   for(j=0;j < *tam_crom ;j++){
283
284      eval_tmp += (funcion(cola,x,pos_individuo, 0.00001 ,j,tam_q)
285           + funcion(cola,x,pos_individuo, 0.0 ,j,tam_q))/0.00001;
286
287      eval_tmp_p += (funcion(cola,x_p,pos_individuo, 0.00001 ,j,tam_q)
288           + funcion(cola,x_p,pos_individuo, 0.0 ,j,tam_q))/0.00001;
289
290      }
291
292       eval[tid]= eval_tmp;
293       eval_p[tid]= eval_tmp_p;
294
295
296   }//if
297
298 }
299
300 /*--------------FUNCIONES GLOBAL FIN-----------------------------------*/
301 float mutar();
302 float evaluacion(int);
303 void seleccion();
304 void imp();
305
306 int main( void ) {
307
308     clock_t begin,end;
309     double elapsed_secs;
310
311     ofstream myfile;
312
313   float prom_tm[2]={0.0,0.0};
314
315   queue<output_data> ecuacion = parser_file(tam_crom);
316     int tmp=ecuacion.size();
317
318    cola = new data [ecuacion.size()];
319
320   x = new float [tam_pob*tam_crom];
321   sigma = new float [tam_pob*tam_crom];
322
323   x_p = new float [tam_pob*tam_crom];
324   sig_p = new float [tam_pob*tam_crom];
325
326   eval = new float [tam_pob];
327   eval_p = new float [tam_pob];
328
329   //copiado de la pila a un arreglo
330
331   for(int i=0;ecuacion.empty()!=true;i++){
332
333     data  ds(ecuacion.front());
334
335     cola[i]=ds;
336     ecuacion.pop();
337
338     }
339
340
341     srand( time( NULL ) );
342
343 //Generacion de poblacion inicial
344
```

```
345 begin = clock();
346 
347   for(int i=0; i < (tam_pob*tam_crom); i++){
348 
349         x[i] = (float)(pow(-1.0,(rand()%4))*rand()+rand()/100000.0);
350         sigma[i] = (float)(pow(-1.0,(rand()%4))*rand()/10000.0);
351 
352   }
353 
354 
355   for(int g=0;g<no_gen;g++){
356 
357     prom_tm[0] += mutar();
358     prom_tm[1] += evaluacion(tmp);
359 
360     seleccion();
361 
362   if(g%50==0)  {
363 
364      stringstream ss;
365     ss << g;
366     string s="file_cuda" + ss.str() + ".txt";
367     const char* chr = s.c_str();
368 
369     myfile.open (chr);
370     myfile << "\npromedio por iteracion: "<<(prom_tm[0]+prom_tm[1])<<endl;
371     myfile.close();
372     }
373   }
374 
375   end = clock();
376 
377   elapsed_secs = double (end - begin) / CLOCKS_PER_SEC;
378 
379   //impresion de resultados
380   imp();
381 
382 
383 
384    stringstream ss;
385    ss << tam_pob;
386 
387   string s = "file_cuda" + ss.str() + ".txt";
388   const char* chr = s.c_str();
389 
390   myfile.open (chr);
391   myfile << "\npromedio mutacion: "<<float(prom_tm[0]/no_gen)<<"\npromedio eval: "
392       <<float(prom_tm[1]/no_gen)<<"\ntiempo total: "<<elapsed_secs<<endl;
393   myfile.close();
394 
395  cout << "\npromedio mutacion: "<<float(prom_tm[0]/no_gen)<<"\npromedio eval: "
396     <<float(prom_tm[1]/no_gen)<<"\ntiempo total: "<<elapsed_secs<<endl;
397 
398    delete[] x;
399    delete[] sigma;
400 
401    delete[] x_p;
402    delete[] sig_p;
403 
404 
405   system("pause");
406     return 0;
407 }
408 
409 
410 
411 float mutar(){
412 
413 cudaEvent_t start, stop;
414 float tiempo;
415 cudaEventCreate(&start);
416 cudaEventCreate(&stop);
417 
418 /*******************/
419 //1 matriz - 2 arreglo
420   float *tmp1,*tmp2;
421 
422   float *ranUni, * ranUni2;
423 
424   int i,j,*tam_cr,*tmp_3;
425 
426     int t =(int)(tam_pob/16);
427         // launch kernel  4  -- 4
428     if(t==0)
429        t = 1;
430     else if (tam_pob%16 != 0)
431       t++;
432 
433 
```

```
434   ranUni = new float [tam_pob*tam_crom];
435   ranUni2 = new float [tam_pob];
436 
437   srand( time( NULL ) );
438 //generacion de matriz de numeros uniformes aletorios
439   for(i=0;i<(tam_pob); i++){
440 
441        ranUni2[i] = unifRand();
442 
443     for(j=0;j<(tam_crom); j++)
444 
445       ranUni[(i*tam_crom)+j] = unifRand();
446   }
447  tmp_3= &tam_crom;
448 
449 cudaEventRecord(start, 0);
450 /********************MUTACION*************************************/
451    // allocate the memory on the GPU
452     cudaMalloc( (void**)&dev_x, (tam_pob*tam_crom) * sizeof(float) ) ;
453     cudaMalloc( (void**)&dev_sig, (tam_pob*tam_crom) * sizeof(float) ) ;
454 
455     cudaMalloc( (void**)&dev_x_p, (tam_pob*tam_crom) * sizeof(float) ) ;
456     cudaMalloc( (void**)&dev_sig_p, (tam_pob*tam_crom) * sizeof(float) ) ;
457 
458   cudaMalloc( (void**)&tmp1, (tam_pob*tam_crom) * sizeof(float) ) ;
459     cudaMalloc( (void**)&tmp2, (tam_pob) * sizeof(float) ) ;
460 
461   cudaMalloc( (void**)&tam_cr,  sizeof(int) ) ;
462 
463 
464 ///copiado de memoria cpu a gpu
465 
466 
467   cudaMemcpy( dev_x, x, (tam_pob*tam_crom) * sizeof(float), cudaMemcpyHostToDevice  );
468     cudaMemcpy( dev_sig, sigma, (tam_pob*tam_crom) * sizeof(float), cudaMemcpyHostToDevice  );
469 
470   cudaMemcpy( tmp1, ranUni, (tam_pob*tam_crom) * sizeof(float), cudaMemcpyHostToDevice  );
471     cudaMemcpy( tmp2, ranUni2, (tam_pob) * sizeof(float), cudaMemcpyHostToDevice  );
472 
473   cudaMemcpy( tam_cr, tmp_3,  sizeof(int), cudaMemcpyHostToDevice  );
474 
475 
476 
477 
478   mutacion<<<t, 16>>>( dev_x, dev_sig, dev_x_p,dev_sig_p,tmp1,tmp2,tam_cr);
479 
480   // copiar resultado
481 
482      cudaMemcpy( x_p, dev_x_p, (tam_pob*tam_crom) * sizeof(float), cudaMemcpyDeviceToHost  );
483      cudaMemcpy( sig_p, dev_sig_p, (tam_pob*tam_crom) * sizeof(float), cudaMemcpyDeviceToHost  );
484 
485    cudaEventRecord(stop, 0);
486   cudaEventSynchronize(stop);
487 
488 cudaEventElapsedTime(&tiempo, start, stop);
489 
490 //cout<<"mutacion: ms)"<< tiempo<<endl;
491 
492 
493     // free the memory allocated on the GPU
494     cudaFree( dev_x );
495     cudaFree( dev_sig );
496 
497     cudaFree( dev_x_p );
498     cudaFree( dev_sig_p );
499 
500 
501 
502   ranUni2=ranUni=NULL;
503 
504     tmp1=NULL;
505   tmp2=NULL;
506 
507   return tiempo;
508 }
509 
510 
511 float evaluacion(int tmp){
512 
513 cudaEvent_t start, stop;
514 float tiempo;
515 cudaEventCreate(&start);
516 cudaEventCreate(&stop);
517 
518 //tamaño de en bytes de la estructura
519 int numBytes = tmp * sizeof(data);
520 int *dt=&tmp;
521   dv_tmp = &tam_crom;
522 
```

```
523   //calculo de hilo segun el tamaño de poblacion
524
525   int t =(int)(tam_pob/16);
526         // launch kernel  4  -- 4
527     if(t==0)
528        t = 1;
529     else if (tam_pob%16 != 0)
530       t++;
531
532  //reservacion de memoria
533
534     cudaEventRecord(start, 0);
535     cudaMalloc((void**)&dev_cola, numBytes);
536
537     cudaMalloc( (void**)&dev_x, (tam_pob*tam_crom) * sizeof(float) ) ;
538     cudaMalloc( (void**)&dev_x_p, (tam_pob*tam_crom) * sizeof(float) ) ;
539
540    cudaMalloc( (void**)&dev_eval, tam_pob * sizeof(float) ) ;
541    cudaMalloc( (void**)&dev_eval_p, tam_pob * sizeof(float) ) ;
542
543     cudaMalloc( (void**)&dev_tam,  sizeof(int) ) ;
544     cudaMalloc( (void**)&dev_t_cr,  sizeof(int) ) ;
545
546
547   //movemos los datos a la gpu
548
549   cudaMemcpy( dev_cola, cola,  numBytes, cudaMemcpyHostToDevice  );
550   cudaMemcpy( dev_x, x, (tam_pob*tam_crom) * sizeof(float), cudaMemcpyHostToDevice  );
551     cudaMemcpy( dev_x_p, x_p, (tam_pob*tam_crom) * sizeof(float), cudaMemcpyHostToDevice  );
552
553
554   cudaMemcpy( dev_tam, dt, sizeof(int), cudaMemcpyHostToDevice  );
555     cudaMemcpy( dev_t_cr, dv_tmp, sizeof(int), cudaMemcpyHostToDevice  );
556
557
558
559   eval_gpu<<<t,16>>>(dev_cola, dev_x, dev_x_p, dev_eval, dev_eval_p, dev_tam, dev_t_cr);
560
561      // copiar los resultados a la cpu
562
563   cudaMemcpy( eval, dev_eval, tam_pob * sizeof(float), cudaMemcpyDeviceToHost  );
564     cudaMemcpy( eval_p, dev_eval_p, tam_pob * sizeof(float), cudaMemcpyDeviceToHost  );
565
566   cudaEventRecord(stop, 0);
567     cudaEventSynchronize(stop);
568
569   cudaEventElapsedTime(&tiempo, start, stop);
570
571
572     cudaFree(dev_cola);
573     cudaFree(dev_x);
574   cudaFree(dev_x_p);
575   cudaFree(dev_eval);
576   cudaFree(dev_eval_p);
577   cudaFree(dev_tam);
578   cudaFree(dev_t_cr);
579
580   //regresa el tiempo de ejecucion
581 return tiempo;
582
583 }
584
585
586 void seleccion(){
587
588
589
590 //***comparacion**//
591   torneo *ronda_1,*ronda_2,*ronda_3;
592    srand( time( NULL ) );
593
594 int k;
595 int q=((rand() % 100) + 1);
596
597
598     if(q<30)
599
600         q+=30;
601
602     else if(q>70)
603
604         q-=30;
605
606     q = tam_pob*q/100;
607
608     if((q%2)==1)//que no queden impares los participantes del torneo
609
610         q++;
611
```

```
612
613   ronda_1 = new  torneo  [q];
614 //**********Inicio ronda_1****************/
615     for(k=0;k<q;k++){
616
617        ronda_1[k].inicio(rand()%tam_pob);//asignacion de indice
618      bool padre =   rand()%2;
619
620       if(padre == false){
621
622             ronda_1[k].p = true;
623             ronda_1[k].val = eval_p[ronda_1[k].indice];
624
625             }else{
626
627                  ronda_1[k].val = eval[ronda_1[k].indice];
628
629                 }
630        }
631
632
633
634   ronda_2 = new  torneo  [q/2];
635
636  for(k=0;k<(q/2);k++){
637
638
639     if(abs(ronda_1[k].val)<abs(ronda_1[k+1].val))
640
641            ronda_2[k]= ronda_1[k];
642
643            else
644
645            ronda_2[k]= ronda_1[k+1];
646
647        }
648
649
650
651   ronda_3 = new  torneo  [q/4];
652
653  for(k=0;k<(q/4);k++){
654
655
656     if(abs(ronda_2[k].val)<abs(ronda_2[k+1].val))
657
658            ronda_3[k]= ronda_2[k];
659
660            else
661
662            ronda_3[k]= ronda_2[k+1];
663
664        }
665
666
667 //****seleccion***//
668
669 //******insercion de los ganadores a la poblacion****************//
670 int i;
671
672
673  for(k=0;k<(q/4);k++){
674
675     if(ronda_3[k].p == true){
676
677       i = ronda_3[k].indice;
678
679
680         x[i] =x_p[i];
681         sigma[i] = sig_p[i];
682
683         }
684
685  }
686
687  ronda_1=NULL;
688  ronda_2=NULL;
689  ronda_3=NULL;
690
691 }
692
693 /*imprime los indivuduos con la mejor evaluacion más cercana a 0*/
694 void imp(){
695
696       for(int j=0;j<tam_pob;j++){ // individuos
697
698         if(eval[j]<1 && eval[j]>-1){
699
700               cout<<j<<":: eval: "<<eval[j]<<endl;
```

```
701             cout<<"individuo:\n";
702             for(int i=0;i<tam_crom;i++)
703             cout<<x[(j*tam_crom)+i]<<"   ";
704 
705       cout<<endl<<"----------"<<endl;
706      }//if
707 
708    }//for
709 
710    }//funcion
```

src/tesla/parTesla.cu

Archivo parGT630.cu, es la versión de la implentación paralela del algoritrmo evolutivo para la GPU GT 630.

Evolutivo paralelo GT 630

```
1  #include <cuda.h>
2  #include <iostream>
3  #include <stdlib.h>
4  #include <ctime>
5  #include <algorithm>
6  
7  #include<cmath>
8  
9  #include "shun_yard.h"
10 
11 #include<stdio.h>
12 
13  #define MAX 10
14 
15 const int tam_pob = 5000;
16 const int no_gen =900;
17 
18 int tam_crom;
19 
20  float *x, *sigma, *x_p, *sig_p, *eval, *eval_p;
21  float *dev_x, *dev_sig, *dev_x_p, *dev_sig_p, *dev_eval,*dev_eval_p;
22 
23  int *dev_tam, *dev_t_cr, *dv_tmp; //tamaño cola - tamaño de cromosoma - tmp
24 
25  data *cola, *dev_cola;
26 
27 
28 float unifRand(){
29 
30     return rand() / double(RAND_MAX);
31 
32 }
33 
34 /*--------------FUNCIONES DEVICE ---------------------------------*/
35 __device__   float operate_1(float num,  const char *func) {
36 
37 
38 
39            if(func[0] == 's' && func[1] == 'i' && func[2] == 'n' )
40 
41             return sinf(num * PI/180.0);
42 
43         else if(func[0] == 'c' && func[1] == 'o' && func[2] == 's'  )
44 
45             return cosf(num * PI/180.0);
46 
47         else if(func[0] == 'l' && func[1] == 'o' && func[2] == 'g'  )
48 
49             return logf(num);
50 
51        else if(func[0] == 'e')
52 
53             return expf (num);
54 
55             return 0; }
56             //evaluates an operator.
57 
58  __device__   float operate_2(float num1, float num2, char  op) {
59 
60         switch(op) {
61             case '+':
62             return num1 + num2;
```

```
63
64              case '-':
65              return num1 - num2;
66
67              case '*':
68              return num1 * num2;
69
70              case '/':
71              return num1 / num2;
72
73              case '^':
74              return pow(num1, num2);
75               default: return 0;
76               }
77
78          }
79
80
81     __device__ bool func_op(const char *check) {
82
83          if(check[0] == 's' || check[0] == 'c' || check[0] == 'l' || check[0] =='e')
84          return true;
85      else
86          return false;
87      }
88
89
90
91 __device__ void push(float pila[],float val,int &pos_pila){
92
93
94      if(pos_pila==MAX-1)
95      {
96         // printf("Stack full\n");
97          return;
98      }else{
99
100
101         pos_pila++;
102         pila[pos_pila]=val;
103
104
105     }
106
107 }
108
109 __device__ float pop(float pila[],int &pos_pila)
110 {
111       float t;
112      if(pos_pila==-1)
113      {
114           //printf("Stack empty\n");
115           return 0;
116      }else{
117      t=pila[pos_pila];
118      pos_pila=pos_pila-1;
119      return t;
120      }
121 }
122
123
124 //dist normal inv
125 __device__ float normsinv(float  p) {
126
127      float a[] = {-3.969683028665376e+01,  2.209460984245205e+02,
128                   -2.759285104469687e+02,  1.383577518672690e+02,
129                   -3.066479806614716e+01,  2.506628277459239e+00};
130
131    float  b[] = {-5.447609879822406e+01,  1.615858368580409e+02,
132                   -1.556989798598866e+02,  6.680131188771972e+01,
133                   -1.328068155288572e+01 };
134
135     float  c[] = {-7.784894002430293e-03, -3.223964580411365e-01,
136                   -2.400758277161838e+00, -2.549732539343734e+00,
137                   4.374664141464968e+00,  2.938163982698783e+00};
138
139    float  d[] ={7.784695709041462e-03, 3.224671290700398e-01,
140                    2.445134137142996e+00,  3.754408661907416e+00};
141
142
143     // Define break-points.
144     float plow  = 0.02425;
145     float phigh = 1 - plow;
146
147
148     // Rational approximation for lower region:
149     if ( p < plow ) {
150
151              float q  = sqrtf ( -2* logf (p)) ;
```

```
152             return   (((((c[0]*q+c[1])*q+c[2])*q+c[3])*q+c[4])*q+c[5]) /
153                                               ((((d[0]*q+d[1])*q+d[2])*q+d[3])*q+1);
154     }
155
156     // Rational approximation for upper region:
157     if ( phigh < p ) {
158             float q  = sqrtf(-2*logf(1-p));
159           return      -(((((c[0]*q+c[1])*q+c[2])*q+c[3])*q+c[4])*q+c[5]) /
160                                                      ((((d[0]*q+d[1])*q+d[2])*q+d[3])*q+1);
161     }
162
163     // Rational approximation for central region:
164     float q = p - 0.5;
165     float r = q*q;
166
167         return   (((((a[0]*r+a[1])*r+a[2])*r+a[3])*r+a[4])*r+a[5])*q /
168                              (((((b[0]*r+b[1])*r+b[2])*r+b[3])*r+b[4])*r+1);
169
170
171
172
173   }
174
175
176
177   __device__ float funcion(data *cola,float *x,int pos_individuo,
178               float h, int index, int *tam_q, int *tam_crom){
179
180 // el arreglo pila simula una pila
181 //num1,num2 y num son se guardan temporalmente los valores de la pila
182
183  float pila[MAX],num,num1,num2,res;
184 int pos_pila=-1;
185 int i;
186
187     for(i=0; i<*tam_q;i++){//hasta llegar al final de la cola
188
189
190          if(cola[i].is_num && !cola[i].is_var) {//cte
191
192
193           push(pila, cola[i].num, pos_pila);
194
195       }else if(cola[i].is_num && cola[i].is_var){//var
196
197
198           int in_dev = ( pos_individuo) + (cola[i].ind-1);
199
200           if(index != (cola[i].ind - 1))
201
202           push(pila, x[in_dev ], pos_pila);
203
204           else{
205
206             float f = (x[ in_dev ] + h);
207           push(pila, f , pos_pila);
208
209           }
210             }
211
212       else if(func_op(cola[i].opf))  { //funciones sin cos etc
213
214           num = pop(pila,pos_pila);
215           res = operate_1(num, cola[i].opf);
216
217           push(pila, res, pos_pila);
218           }
219
220       else {// operandos
221
222                num1 = pop(pila,pos_pila);
223          num2 = pop(pila,pos_pila);
224
225             res = operate_2(num2, num1, cola[i].opf[0]);
226
227                 push(pila, res, pos_pila);
228
229
230           }
231
232   }//tamaño cola
233
234   return pila[pos_pila];
235
236
237
238
239   }//fin device
240
```

```
241 /*--------------FUNCIONES DEVICE FIN ----------------------------------*/
242
243
244 /*--------------FUNCIONES GLOBAL ----------------------------------*/
245 //Mutacion de individuos
246 __global__ void mutacion( float *x, float *sig, float *x_p, float *sigma_p, float * ranUni ,
247               float * randUni2, int *tam_crom) {
248
249       //calculo de la posicion del hilo actual
250     int tid = threadIdx.x + blockIdx.x * blockDim.x;
251
252
253     if (tid < (tam_pob*(*tam_crom))){
254
255         x_p[tid] = x[tid] + normsinv(ranUni[tid])*sig[tid];
256     sigma_p[tid]= sig[tid] * (1 +  normsinv(randUni2[(int)(tid/tam_pob)]) );
257
258     }
259 }
260
261 //Evaluacion de individuos
262
263 __global__ void eval_gpu(data *cola, float *x, float *x_p,
264             float *eval, float *eval_p, int *tam_q, int *tam_crom){
265
266
267   //calculo de la posicion del hilo actual
268  int tid = blockIdx.x * blockDim.x + threadIdx.x;
269
270  float eval_tmp=0;
271  float eval_tmp_p=0;
272
273  int j;
274  int pos_individuo = *tam_crom*tid ;
275
276   if(tid < 500){
277
278
279   for(j=0;j < *tam_crom ;j++){
280
281      eval_tmp += (funcion(cola,x,pos_individuo, 0.00001 ,j,tam_q,tam_crom)
282                 + funcion(cola,x,pos_individuo, 0.0 ,j,tam_q,tam_crom))/0.00001;
283
284      eval_tmp_p += (funcion(cola,x_p,pos_individuo, 0.00001 ,j,tam_q,tam_crom)
285                 + funcion(cola,x_p,pos_individuo, 0.0 ,j,tam_q,tam_crom))/0.00001;
286
287      }
288
289       eval[tid]= eval_tmp;
290       eval_p[tid]= eval_tmp_p;
291
292
293   }//if
294
295 }
296
297 /*--------------FUNCIONES GLOBAL FIN----------------------------------*/
298 float mutar();
299 float evaluacion(int);
300 void seleccion();
301 void imp();
302
303 int main( void ) {
304
305     clock_t begin,end,t_gen,t_fin;
306     double elapsed_secs;
307     ofstream myfile;
308
309   float prom_tm[2]={0.0,0.0};
310
311   queue<output_data> ecuacion = parser_file(tam_crom);
312     int tmp=ecuacion.size();
313
314    cola = new data [ecuacion.size()];
315
316   x = new float [tam_pob*tam_crom];
317   sigma = new float [tam_pob*tam_crom];
318
319   x_p = new float [tam_pob*tam_crom];
320   sig_p = new float [tam_pob*tam_crom];
321
322   eval = new float [tam_pob];
323   eval_p = new float [tam_pob];
324
325   //copiado de la pila a un arreglo
326
327   for(int i=0;ecuacion.empty()!=true;i++){
328
329     data  ds(ecuacion.front());
```

```
330
331     cola[i]=ds;
332     ecuacion.pop();
333
334     }
335
336
337     srand( time( NULL ) );
338
339 //Generacion de poblacion inicial
340
341 t_gen=begin = clock();
342
343   for(int i=0; i < (tam_pob*tam_crom); i++){
344
345         x[i] = (float)(pow(-1.0,(rand()%4))*rand()+rand()/100000.0);
346         sigma[i] = (float)(pow(-1.0,(rand()%4))*rand()/10000.0);
347
348   }
349
350 t_fin=clock();
351
352 cout<< "tiempo de generacion: ms) "<<( t_fin-t_gen)<<endl;
353
354
355   for(int g=0;g<no_gen;g++){
356
357     prom_tm[0] += mutar();
358     prom_tm[1] += evaluacion(tmp);
359
360     seleccion();
361
362   }
363
364   end = clock();
365
366   elapsed_secs = double (end - begin) / CLOCKS_PER_SEC;
367
368   //impresion de resultados
369   imp();
370
371
372    stringstream ss;
373    ss << tam_pob;
374
375   string s="file_cuda" + ss.str() + ".txt";
376   const char* chr = s.c_str();
377
378 //guarda los tiempos totales en un archivo
379   myfile.open (chr);
380   myfile << "\npromedio mutacion: "<<float(prom_tm[0]/no_gen)<<"\npromedio eval: "<<float(prom_tm[1]/no_gen)<<"\
        ntiempo total: "<<elapsed_secs<<endl;
381
382   myfile.close();
383
384    delete[] x;
385    delete[] sigma;
386
387    delete[] x_p;
388    delete[] sig_p;
389
390
391   system("pause");
392     return 0;
393 }
394
395
396
397 float mutar(){
398
399 cudaEvent_t start, stop;
400 float tiempo;
401 cudaEventCreate(&start);
402 cudaEventCreate(&stop);
403
404
405 //tmp1 matriz de numeros aletorios
406 //tmp2 de numeros aletorios arreglo
407
408   float *tmp1,*tmp2;
409
410   float *ranUni, * ranUni2;
411
412   int i,j,*tam_cr,*tmp_3;
413   int t = ((tam_pob*tam_crom)/16);
414
415   ranUni = new float [tam_pob*tam_crom];
416   ranUni2 = new float [tam_pob];
417
```

```
418   srand( time( NULL ) );
419 
420 //generacion de matriz de numeros uniformes aleatorios
421 
422   for(i=0;i<(tam_pob); i++){
423 
424        ranUni2[i] = unifRand();
425 
426     for(j=0;j<(tam_crom); j++)
427 
428       ranUni[(i*tam_crom)+j] = unifRand();
429   }
430 
431 tmp_3= &tam_crom;
432 
433 cudaEventRecord(start, 0);
434 /********************MUTACION*****************/
435    // reservacion de memoria en la gpu
436 
437     cudaMalloc( (void**)&dev_x, (tam_pob*tam_crom) * sizeof(float) ) ;
438     cudaMalloc( (void**)&dev_sig, (tam_pob*tam_crom) * sizeof(float) ) ;
439 
440     cudaMalloc( (void**)&dev_x_p, (tam_pob*tam_crom) * sizeof(float) ) ;
441     cudaMalloc( (void**)&dev_sig_p, (tam_pob*tam_crom) * sizeof(float) ) ;
442 
443   cudaMalloc( (void**)&tmp1, (tam_pob*tam_crom) * sizeof(float) ) ;
444     cudaMalloc( (void**)&tmp2, (tam_pob) * sizeof(float) ) ;
445 
446   cudaMalloc( (void**)&tam_cr,  sizeof(int) ) ;
447 
448 
449 ///copiado de memoria cpu a gpu
450 
451 
452   cudaMemcpy( dev_x, x, (tam_pob*tam_crom) * sizeof(float), cudaMemcpyHostToDevice  );
453     cudaMemcpy( dev_sig, sigma, (tam_pob*tam_crom) * sizeof(float), cudaMemcpyHostToDevice  );
454 
455   cudaMemcpy( tmp1, ranUni, (tam_pob*tam_crom) * sizeof(float), cudaMemcpyHostToDevice  );
456     cudaMemcpy( tmp2, ranUni2, (tam_pob) * sizeof(float), cudaMemcpyHostToDevice  );
457 
458   cudaMemcpy( tam_cr, tmp_3,  sizeof(int), cudaMemcpyHostToDevice  );
459 
460 
461 
462 
463   mutacion<<<t, 16>>>( dev_x, dev_sig, dev_x_p,dev_sig_p,tmp1,tmp2,tam_cr);
464 
465 
466   // copiar resultado
467 
468      cudaMemcpy( x_p, dev_x_p, (tam_pob*tam_crom) * sizeof(float), cudaMemcpyDeviceToHost  );
469      cudaMemcpy( sig_p, dev_sig_p, (tam_pob*tam_crom) * sizeof(float), cudaMemcpyDeviceToHost  );
470 
471   cudaEventRecord(stop, 0);
472   cudaEventSynchronize(stop);
473 
474 cudaEventElapsedTime(&tiempo, start, stop);
475 
476 cout<<"mutacion: ms)"<< tiempo<<endl;
477 
478 
479     // free the memory allocated on the GPU
480     cudaFree( dev_x );
481     cudaFree( dev_sig );
482 
483     cudaFree( dev_x_p );
484     cudaFree( dev_sig_p );
485 
486 
487 
488   ranUni2=ranUni=NULL;
489 
490     tmp1=NULL;
491   tmp2=NULL;
492 
493   return tiempo;
494 }
495 
496 
497 float evaluacion(int tmp){
498 
499 cudaEvent_t start, stop;
500 float tiempo,tiempo_total;
501 
502 cudaEventCreate(&start);
503 cudaEventCreate(&stop);
504 
505 //tamaño de en bytes de la estructura (pila)
506 int numBytes = tmp * sizeof(data);
```

```
507   int *dt=&tmp;
508   dv_tmp = &tam_crom;
509
510
511   //se divide para poder ser procesado
512   int pob_parc = tam_pob/500;
513
514   int t =(int)(500/16);
515
516     if(t==0)
517
518      t = 1;
519
520     else if(t%16!=0)
521
522       t++;
523
524
525   for(int i=0;i<pob_parc;i++) {
526
527        //poblacion parcial
528     float *cr_x, *cr_x_p;
529     float *ev,*ev_p;
530
531     cr_x = new float [500*tam_crom];
532     cr_x_p = new float [500*tam_crom];
533
534     ev = new float [500];
535     ev_p = new float [500];
536
537        int pos =(i*500);
538      //copia de evaluacion
539      for(int j=0;j<500;j++){
540
541      ev[j]= eval[pos+j];
542      ev_p[j]=eval_p[pos+j];
543      }
544
545     //copiado parcial de la poblacion
546     for(int j=0;j<500;j++){
547
548       for(int k=0;k<tam_crom;k++){
549
550        cr_x[(j*tam_crom) + k]= x[pos +(j*tam_crom) + k];
551        cr_x_p[(j*tam_crom) + k]= x_p[pos +(j*tam_crom) + k];
552         }
553     }
554
555
556   cudaEventRecord(start, 0);
557
558   //reservado de memoria
559     cudaMalloc((void**)&dev_cola, numBytes);
560
561     cudaMalloc( (void**)&dev_x, (500*tam_crom) * sizeof(float) ) ;
562     cudaMalloc( (void**)&dev_x_p, (500*tam_crom) * sizeof(float) ) ;
563
564    cudaMalloc( (void**)&dev_eval, 500 * sizeof(float) ) ;
565    cudaMalloc( (void**)&dev_eval_p, 500 * sizeof(float) ) ;
566
567     cudaMalloc( (void**)&dev_tam,  sizeof(int) ) ;
568     cudaMalloc( (void**)&dev_t_cr,  sizeof(int) ) ;
569
570   //copiado a la gpu
571
572   cudaMemcpy( dev_cola, cola,  numBytes, cudaMemcpyHostToDevice  );
573   cudaMemcpy( dev_x, cr_x, (500*tam_crom) * sizeof(float), cudaMemcpyHostToDevice  );
574     cudaMemcpy( dev_x_p, cr_x_p, (500*tam_crom) * sizeof(float), cudaMemcpyHostToDevice  );
575
576
577   cudaMemcpy( dev_tam, dt, sizeof(int), cudaMemcpyHostToDevice  );
578     cudaMemcpy( dev_t_cr, dv_tmp, sizeof(int), cudaMemcpyHostToDevice  );
579
580
581
582   eval_gpu<<<t,16>>>(dev_cola, dev_x, dev_x_p, dev_eval, dev_eval_p, dev_tam, dev_t_cr);
583
584
585     // copiar los resultados de vuelta a la cpu
586
587   cudaMemcpy( ev, dev_eval, 500 * sizeof(float), cudaMemcpyDeviceToHost  );
588     cudaMemcpy( ev_p, dev_eval_p, 500 * sizeof(float), cudaMemcpyDeviceToHost  );
589
590   cudaEventRecord(stop, 0);
591     cudaEventSynchronize(stop);
592
593   // int pos =(i*500);
594
595   for(int j=0;j<500;j++){
```

```
596
597      eval[pos+j] = ev[j] ;
598      eval_p[pos+j] = ev_p[j] ;
599      }
600
601
602   cudaEventElapsedTime(&tiempo, start, stop);
603   tiempo_total=tiempo;
604
605     cudaFree(dev_cola);
606     cudaFree(dev_x);
607   cudaFree(dev_x_p);
608   cudaFree(dev_eval);
609   cudaFree(dev_eval_p);
610   cudaFree(dev_tam);
611   cudaFree(dev_t_cr);
612
613   }//fin for
614   cout<<"tiempo eval: "<<tiempo_total<<endl;
615 return tiempo_total;
616
617 }
618
619 void seleccion(){
620
621
622
623 //***comparacion**//
624   torneo *ronda_1,*ronda_2,*ronda_3;
625    srand( time( NULL ) );
626
627 int k;
628 int q=((rand() % 100) + 1);
629
630
631     if(q<30)
632
633         q+=30;
634
635     else if(q>70)
636
637         q-=30;
638
639     q = tam_pob*q/100;
640
641     if((q%2)==1)//que no queden impares los participantes del torneo
642
643         q++;
644
645
646   ronda_1 = new  torneo  [q];
647 //**********Inicio ronda_1****************/
648     for(k=0;k<q;k++){
649
650        ronda_1[k].inicio(rand()%tam_pob);//asignacion de indice
651      bool padre =   rand()%2;
652
653       if(padre == false){
654
655             ronda_1[k].p = true;
656             ronda_1[k].val = eval_p[ronda_1[k].indice];
657
658             }else{
659
660                  ronda_1[k].val = eval[ronda_1[k].indice];
661
662                 }
663        }
664
665
666
667   ronda_2 = new  torneo  [q/2];
668
669  for(k=0;k<(q/2);k++){
670
671
672     if(abs(ronda_1[k].val)<abs(ronda_1[k+1].val))
673
674            ronda_2[k]= ronda_1[k];
675
676            else
677
678            ronda_2[k]= ronda_1[k+1];
679
680        }
681
682
683
684   ronda_3 = new  torneo  [q/4];
```

```
685
686  for(k=0;k<(q/4);k++){
687
688
689     if(abs(ronda_2[k].val)<abs(ronda_2[k+1].val))
690
691            ronda_3[k]= ronda_2[k];
692
693            else
694
695            ronda_3[k]= ronda_2[k+1];
696
697        }
698
699
700 //****seleccion***//
701
702 //******insercion de los ganadores a la poblacion****************//
703 int i;
704
705
706  for(k=0;k<(q/4);k++){
707
708     if(ronda_3[k].p == true){
709
710       i = ronda_3[k].indice;
711
712
713         x[i] =x_p[i];
714         sigma[i] = sig_p[i];
715
716         }
717
718  }
719
720  ronda_1=NULL;
721  ronda_2=NULL;
722  ronda_3=NULL;
723
724 }
725
726 /*imprime los indivuduos con la mejor evaluacion más cercana a 0*/
727 void imp(){
728
729       for(int j=0;j<tam_pob;j++){ // individuos
730
731         if(eval[j]<1 && eval[j]>-1){
732
733               cout<<j<<":: eval: "<<eval[j]<<endl;
734               cout<<"individuo:\n";
735               for(int i=0;i<tam_crom;i++)
736               cout<<x[(j*tam_crom)+i]<<"   ";
737
738         cout<<endl<<"----------"<<endl;
739        }//if
740
741     }//for
742
743     }//funcion
```

src/GT630/parGT630.cu