INSTITUTO TECNOLÓGICO DE APIZACO

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

"ALGORITMO HÍBRIDO PARALELO EVOLUTIVO PARA APLICACIONES EN TOMOGRAFÍA SÍSMICA"

TESIS

PRESENTA:

ING. EUSTOLIA CARREÓN ESTEBAN

PARA OBTENER EL TÍTULO DE:

MAESTRO EN SISTEMAS COMPUTACIONALES

ASESORES:

DR. JOSÉ FEDERICO RAMÍREZ CRUZ

DR. JOSÉ CRISPÍN HERNÁNDEZ HERNÁNDEZ

CO-ASESOR:

DR. MIGUEL O. ARIAS ESTRADA

Apizaco, Tlaxcala Enero de 2015

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO
Instituto Tecnológico de Apizaco

"2015, Año del Generalísimo José María Morelos y Pavón"

Apizaco, Tlax., 12 de Enero de 2015

ASUNTO: **Aprobación del trabajo de Tesis de Maestría.**

M.A.D. MA. A. ACELA DAVILA JIMENEZ
JEFA DE LA DIVISIÓN DE ESTUDIOS DE
POSGRADO E INVESTIGACIÓN.
P R E S E N T E.

Por este medio se le informa a usted, que los integrantes de la **Comisión Revisora** para el trabajo de tesis de maestría que presenta la **ING. EUSTOLIA CARREON ESTEBAN**, con número de control **M12370002** candidata al grado de **Maestra en Sistemas Computacionales** y egresada del **Instituto Tecnológico de Apizaco,** cuyo tema es "*ALGORITMO HÍBRIDO PARALELO EVOLUTIVO PARA APLICACIONES EN TOMOGRAFÍA SÍSMICA*"**,** fue:

A P R O B A D O

Lo anterior, al valorar el trabajo profesional presentado por la candidata y constatar que las observaciones que con anterioridad se le marcaron así como correcciones sugeridas para su mejora ya han sido realizadas.

Por lo que se avala se continúe con los trámites pertinentes para su titulación.

Sin otro particular por el momento, le envió un cordial saludo.

LA COMISIÓN REVISORA

DR. JOSE FEDERICO RAMIREZ CRUZ

DR. JOSE CRISPIN HERNANDEZ HERNANDEZ

DR. EDMUNDO BONILLA HUERTA

M.C. BLANCA ESTELA PEDROZA MENDEZ

C. p.- Interesada.

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO
Instituto Tecnológico de Apizaco

"2015, Año del Generalísimo José María Morelos y Pavón"

Apizaco, Tlax., 13 de Enero de 2015

No. de Oficio: DEPI/012/15

ASUNTO: **Se Autoriza Impresión de Tesis de Grado.**

ING. EUSTOLIA CARREON ESTEBAN
CANDIDATA AL GRADO DE MAESTRA EN
SISTEMAS COMPUTACIONALES
No. de Control: **M12370002**
P R E S E N T E.

Por este medio me permito informar a usted, que por aprobación de la Comisión Revisora asignada para valorar el trabajo, mediante la Opción: **I Tesis de Grado por Proyecto de Investigación**, de la **Maestría en Sistemas Computacionales**, que presenta con el tema: "***ALGORITMO HÍBRIDO PARALELO EVOLUTIVO PARA APLICACIONES EN TOMOGRAFÍA SÍSMICA***" y conforme a lo establecido en el Procedimiento para la Obtención del Grado de Maestría en el Instituto Tecnológico, la División de Estudios de Posgrado e Investigación a mi cargo le emite la:

AUTORIZACIÓN DE IMPRESIÓN

Debiendo entregar un ejemplar del mismo debidamente encuadernado y seis copias en CD en formato PDF, para presentar su Acto de Recepción Profesional a la brevedad.

Sin otro particular por el momento, le envío un cordial saludo.

A T E N T A M E N T E
PENSAR PARA SERVIR, SERVIR PARA TRIUNFAR®

M.A.D. MA. A. ACELA DAVILA JIMENEZ
JEFA DE LA DIVISIÓN DE ESTUDIOS DE
POSGRADO E INVESTIGACIÓN.

Secretaría de Educación Pública
Instituto Tecnológico de Apizaco
División de Estudios de Posgrado
e Investigación

MAADJ/mebr.

C.p. Expediente.

Resumen

En este trabajo de investigación se realiza la paralelización de un algoritmo de cobertura de rayos que forma parte de un algoritmo de tomografía sísmica, cuyo objetivo es trazar los rayos generados por 7 fuentes de energía sísmica artificiales (shotpoints) hacia cientos de dispositivos receptores (geófonos), sobre una Unidad de Procesamiento de Gráficos.

Al principio se hace la descripción del Algoritmo de Tomografía Sísmica (ATS) en su forma secuencial, mostrándolo como parte de la función objetivo en una Evolución Diferencial, cuya colaboración tiene como finalidad la generación de un modelo de velocidades de ondas sísmicas en un volumen de 414 414 m^3 en la corteza.

En la etapa de paralelización sobre la GPU, se evalúa en cada hilo a cada uno de los geófonos de manera independiente; se hace uso de la memoria global para almacenar la información necesaria, dado que la capacidad de la memoria compartida está restringida a 48 KB por bloque.

Los datos usados en este trabajo se obtuvieron de un experimento realizado en el campo volcánico El Potrillo, ubicado en el sur de Nuevo México, a cargo del Departamento de Ciencias Computacionales y del Departamento de Ciencias Geológicas de la Universidad de Texas, en El Paso.

Abstract

In this research a parallelization of an ray coverage algorithm as part of a seismic tomography algorithm is performed, which aims to trace rays generated from 7 sources of artificial seismic energy (shotpoints) to hundreds of receiving devices (geophones) on a Graphics Processing Unit. Initially, an description of the seismic tomography algorithm (ATS, Algoritmo de Tomografía Sísmica) is made in its secuential form, showing it as part of the objective function in a differential evolution, which cooperation is intended to generate a model of seismic wave velocities in a volume of 414,414 m textsuperscript 3 in the crustal.

During the stage of parallelization on the GPU, each geophone is evaluated in each thread in a independent way; the use of the global memory is needed to store information, due the shared memory capacity is limited to 48 KB per block. The data used in this study were obtained from an experiment conducted in the volcanic field called “El Potrillo”, located in southern New Mexico, by the Department of Computer Science and the Department of Geological Sciences, University of Texas at El Paso.

Índice general

Índice de figuras

Índice de tablas

DEDICATORIA

A Dios, por permitirme lograr esta meta. Por darme la paciencia y la fortaleza para seguir adelante; por mostrarme el camino y no dejarme desistir en los momentos difíciles.

A mi familia, por su apoyo incondicional y por el ser el motivo de mi superación.

A Marco, por todo el apoyo que me ha brindado; por su paciencia, su solidaridad y su cariño.

AGRADECIMIENTOS

A los compañeros de la maestría, por su amistad y por haber compartido conmigo gratas experiencias.

Al cuerpo de docentes del posgrado del Instituto Tecnológico de Apizaco, por ser una fuente de inspiración para continuar en este camino interminable de formación académica.

Al Dr. José Federico, por dirigir esta tesis; por todas y cada una de sus enseñanzas y por la confianza que ha tenido en mí.

Al CONACYT, por financiar mis estudios de posgrado.

Capítulo 1

INTRODUCCIÓN

1.1. INTRODUCCIÓN

Actualmente se desarrollan procesos de cómputo eficaces que alcanzan los resultados esperados por el usuario; sin embargo, ya no basta únicamente obtenerlos sino hacerlo en el menor tiempo posible.

El cálculo de operaciones en un procesador común es lento debido a que realiza las tareas de manera secuencial; es decir, requiere hacer una a la vez, y no es posible que sean varias al mismo tiempo. Y bien, este problema no se refleja en tareas de oficina sino en problemas que procesan grandes volúmenes de información, ya que al usar este tipo de procesadores la espera de los resultados puede ser de horas, o de días.

En el campo de la Tomografía Sísmica se realizan estudios que permiten obtener modelos de velocidades de ondas, propagadas a través de la corteza terrestre por fuentes de energía naturales, como terremotos; o artificiales, como explosiones, que a la vez ayudan a identificar los materiales o las sustancias existentes en diversas capas de la tierra; para ello se usan diversas técnicas de búsqueda, en este caso en particular se utilizan los Algoritmos Evolutivos (AEs); sin embargo, aunque son métodos eficaces para resolver problemas de gran magnitud como el que se aborda en este trabajo, tiene la inconveniencia de necesitar demasiados recursos

computacionales, y por lo tanto ralentizar el proceso por horas o días, dependiendo de la cantidad de los datos que se usen.

En este trabajo se hace uso de un AE y de un ATS, que tienen como objetivo generar un modelo de velocidades sísmicas de una sección de la corteza terrestre de acuerdo a los tiempos de llegada de las ondas, desde una fuente de energía hacia cada uno de los receptores que se encuentran distribuidos a lo largo de dicha sección. Para probar el algoritmo se utilizaron los datos proporcionados por la Universidad de Texas en El Paso, de un experimento realizado en el campo volcánico El Potrillo, ubicado en el sureste de Nuevo México, cuyo volumen es de 414 414 m^3. Parte de este algoritmo es paralelizado en una Unidad de Procesamiento de Gráficos (GPU, por sus siglas en inglés de Graphics Processing Unit) y bajo la arquitectura de CUDA, en el lenguaje C, con la finalidad de reducir el tiempo de ejecución.

1.2. DESCRIPCIÓN DEL PROBLEMA

El procesamiento de información y los cálculos que realiza una computadora para concluir una tarea ocasiona que el tiempo de cómputo requerido para desplegar los resultados correspondientes, sea la mayoría de las veces demasiado debido a que la Unidad Central de Procesos (CPU, por sus siglas en inglés de Central Processing Unit) debe culminar una actividad para pasar a otra, generando así lentitud, que se traduce en ineficiencia cuando se precisan grandes tareas.

Un estudio de Tomografía Sísmica permite obtener un modelo de velocidades de la estructura de la corteza terrestre bajo cierta región. El conocimiento de estas velocidades tanto en longitud, en amplitud, como en profundidad del volumen permite la localización de diferentes elementos que se encuentran bajo la tierra, por ejemplo: tipos de rocas, fluidos, gases, material de desecho, etc., debido a que cada uno de ellos presenta diferente velocidad de transmisión de ondas sísmicas de acuerdo a su densidad.

En la Universidad de Texas, Ramírez Cruz et al. (2013) desarrollaron un algoritmo para la obtención de modelo de velocidades de las ondas sísmicas en la corteza terrestre, el cual combina los AEs con un algoritmo basado en el gradiente de Tomografía Sísmica que trabaja de manera secuencial, lo cual implica demasiado tiempo de cómputo.

El método de Tomografía Sísmica basado en el gradiente necesita una buena aproximación inicial para hallar una solución óptima. Para resolver el problema inverso de encontrar un modelo de velocidades en tres dimensiones, la mayoría de los procedimientos necesitan un modelo de referencia inicial para desarrollar uno mejor. En general, los modelos de optimización estándar (como el basado en el gradiente) tienen la ventaja de ser más rápidos que los métodos de optimización global como los AEs.

Los AEs tienen la capacidad de buscar en espacios grandes y no son dependientes de una solución inicial aproximada a la óptima. Debido a la gran cantidad de parámetros que manipulan, estos algoritmos se han utilizado principalmente en modelos de una o dos dimensiones. Además, han sido implementados con una combinación de diferentes técnicas para obtener mejores resultados (Ramírez Cruz et al., 2013).

Una de las desventajas de la aplicación de AEs es su alto consumo de recursos computacionales: memoria y/o tiempo del procesador. La inclusión de técnicas paralelas en la implementación de algoritmos ha sido muy importante en mecanismos de búsqueda y optimización, debido a que reducen el tiempo de ejecución. De tal forma, la especial adecuación de estos procedimientos, para trabajar en problemas complejos, puede mejorar.

Una técnica simple para paralelizar Algoritmos Genéticos (AGs) es dividir las tareas de evaluación de la población entre varios procesadores. Las implementaciones paralelas de AGs son comunes, y en muchos casos, tienen éxito para reducir el tiempo requerido para encontrar soluciones aceptables (Cantú-Paz, 1997).

El modelo empleado para la tecnología de GPUs se basa en el uso combinado de una CPU y una GPU en un sistema de coprocesamiento heterogéneo. La parte secuencial de la aplicación se ejecuta en la CPU y las partes de mayor carga computacional se aceleran en la GPU. Esto hace que el tiempo usado para realizar el proceso, y los cálculos de información

se reduzcan en gran medida (Cantú-Paz, 1995).
En este trabajo se diseñará e implementará un algoritmo evolutivo paralelo sobre un modelo CPU-GPU, en la arquitectura CUDA y en lenguaje C, por la ventaja que ofrece el hardware en cuanto al coprocesamiento de datos a un bajo costo y con un alto desempeño, con la finalidad de reducir el tiempo de cálculo en un ATS que realiza el trazo de rayos sísmicos.

1.3. JUSTIFICACIÓN

Debido a que el algoritmo híbrido evolutivo desarrollado para la generación de modelos de Tomografía Sísmica de la Universidad de Texas emplea horas de cómputo en obtener resultados, es necesario reducir el tiempo usado en el proceso de datos.
La importancia de reducir el tiempo en los procesos de un algoritmo de Tomografía Sísmica mediante la paralelización de los mismos, obedece a la necesidad de hacer más eficientes los modelos de velocidades de ondas para el conocimiento de las propiedades y la composición de los materiales internos que constituyen la corteza terrestre, así como la determinación precisa de anomalías sísmicas o discontinuidades a escala local y global.
Los modelos de velocidades obtenidos con Tomografía Sísmica proporcionan información que puede ser utilizada para una amplia variedad de aplicaciones tales como investigaciones arqueológicas (Polymenakos and Papamarinopoulos, 2005), control de calidad y evaluación de proyectos de ingeniería (Liu and Guo, 2005), además del descubrimiento de depósitos de agua, aceite o material de desecho (Lanz et al., 1998; Göktürkler et al., 2008).
Finalmente, se propone el uso de Algoritmos Evolutivos paralelos, ya que son técnicas de búsqueda de gran alcance que se utilizan con éxito para resolver problemas en diferentes disciplinas. Los AEs paralelos prometen una mejora sustancial en el rendimiento (Cantú-Paz, 1998).
Con la combinación de Algoritmos Evolutivos y de GPUs, el tiempo requerido para el pro-

cesamiento de datos en aplicaciones de Tomografía Sísmica, tal como la generación de un modelo de velocidades de las ondas será mejorado en gran medida.

1.4. OBJETIVOS

1.4.1. General

Diseñar e implementar un Algoritmo Evolutivo Paralelo en una GPU para reducir el tiempo en los procesos de un Algoritmo de Tomografía Sísmica.

1.4.2. Específicos

- Implementar un Algoritmo Evolutivo Paralelo en una GPU.
- Analizar el algoritmo de cobertura de rayos, parte del Algoritmo de Tomografía Sísmica, para determinar su posible paralelización.
- Integrar el algoritmo de cobertura de rayos en la GPU, como parte de la función de aptitud en el Algoritmo Evolutivo.

1.5. PREGUNTA DE INVESTIGACIÓN

¿Cuánto es posible reducir el tiempo de ejecución en los procesos de un Algoritmo de Tomografía Sísmica, mediante la paralelización de Algoritmos Evolutivos en una GPU?

1.6. TRABAJOS RELACIONADOS

Los Algoritmos Evolutivos se han convertido en una propuesta de búsqueda robusta y de optimización para resolver problemas con los resultados de la competencia en varias disciplinas como: informática, matemáticas, sistemas sociales, ecología, finanzas y en los sistemas industriales. Han sido aplicados en la solución del problema de las N-reinas y en el del agente viajero, por mencionar algunos; como el componente de aprendizaje en sistemas clasificadores (Andalon-Garcia and Chavoya, 2009), haciendo uso de operadores aleatorios que actúan sobre una población de soluciones candidatas generando nuevas soluciones en el espacio de búsqueda, para posteriormente ser evaluadas y en base a su aptitud (grado de aportación a la solución del problema), determinar si reemplazan o no a la población actual; repitiendo este proceso hasta cumplir con un criterio de terminación, que puede ser un número de generaciones, o encontrar la solución deseada. Cabe mencionar que existen problemas difíciles que requieren una población más grande para que un algoritmo converja, y esta necesidad implica directamente mayores costos computacionales.

La motivación básica detrás de muchos de los primeros estudios de los Algoritmos Evolutivos Paralelos fue reducir el tiempo de procesamiento necesario para alcanzar una solución aceptable. Esto se llevó a cabo implementando Algoritmos Evolutivos en diferentes arquitecturas paralelas, además se notó que en algunos casos éstos encontraron mejores soluciones que los Algoritmos secuenciales.

Las implementaciones paralelas de Algoritmos Evolutivos son comunes, y en la mayoría de los casos tienen éxito para reducir el tiempo requerido y encontrar soluciones aceptables. El diseño de Algoritmos Evolutivos Paralelos involucra opciones como el uso de una o múltiples poblaciones. En ambos casos el tamaño de la(s) población(es) debe ser establecido cuidadosamente; además, las poblaciones pueden quedar aisladas o pueden comunicarse intercambiando individuos. La comunicación involucra un costo extra y decisiones adicionales sobre topologías que determinan cuántos individuos son intercambiados y sobre la frecuencia de las comunicaciones. El tamaño de la población es importante, sin lugar a dudas, debido

a que estudios previos muestran que determina ampliamente la calidad de la búsqueda y su duración.
Cantú-Paz (1999) hizo las siguientes contribuciones para la compresión de Algoritmos Evolutivos Paralelos:

- Calibrar la población exacta para AGs paralelos y simples. Un tamaño de población adecuado es necesario para AGs para ser efectivos y eficientes. Además de considerar que las poblaciones aisladas son pocos prácticas
- Escalabilidad equivalente de únicas o múltiples poblaciones. Sin tomar en cuenta el tipo de población, los AGs pueden integrar un gran número de procesadores.
- Límite inferior sobre paralelismo aceptable. Con cálculos muy simples, los usuarios pueden determinar los beneficios mínimos que ellos pueden esperar de las implementaciones paralelas.

Aunque los Algoritmos Evolutivos son muy efectivos en la resolución de muchos problemas prácticos, el tiempo de ejecución ha llegado a ser un factor limitante para algunos problemas gigantes debido a la gran cantidad de soluciones que deben ser evaluadas. Afortunadamente, la mayoría de estas evaluaciones que consumen mucho tiempo pueden desempeñarse independientemente por cada individuo en la población usando varios tipos de paralelización (Pospichal et al., 2010).
La idea básica detrás de muchos programas paralelos es dividir las grandes tareas en pequeñas secciones y resolverlas simultáneamente usando múltiples procesadores. Esta propuesta de divide y conquista puede ser aplicada de diferentes formas. La literatura muestra muchos ejemplos de implementaciones paralelas satisfactorias donde se emplean diversos métodos, como el uso de una población única, o la división de la población en varias subpoblaciones aisladas. Algunos métodos pueden explotar arquitecturas de computadora masivamente paralelas mientras otros son más adecuados en múltiples computadoras con un poco menos pero más poderosos elementos de procesamiento (Cantú-Paz, 1998). Andalon-Garcia and

Chavoya (2009) presentaron los resultados del análisis de rendimiento de los Algoritmos Evolutivos Paralelos, para cuatro funciones, sobre un clúster de 8 nodos y concluyeron que el tiempo de ejecución disminuyó conforme el número de procesadores aumentó; sin embargo, en las simulaciones con 50 procesadores y más, el rendimiento empeoró al incrementar el número de poblaciones, probablemente debido a que el tamaño de la subpoblación disminuyó y hubo menos variedad en los individuos. También afirmaron que la ejecución de los Algoritmos Paralelos tiene 2 componentes principales: el tiempo usado en la computación, y la comunicación de información entre procesadores. El primero está ampliamente determinado por el tamaño de la población mientras que el segundo depende directamente del número de procesadores y del hardware usado para ejecutar el algoritmo.

Por otra parte, durante muchos años en la industria informática ha habido una predicción en la potencia de los procesadores de las computadoras llamada Ley de Moore, la cual enuncia que el número de transistores en un microprocesador se duplica cada dieciocho meses. Sin embargo, se ha postulado que esta ley ya no es válida para CPUs porque en los últimos años el aumento de la potencia de éstas ha sido limitado. Por una parte, debido a los límites físicos de capacidad de procesamiento que alcanzan las láminas de silicio; y en otra, a que los principales consumidores de productos de informática son los trabajadores de oficina. El principal uso de los recursos informáticos del entorno de oficina está en la hoja de cálculo y aplicaciones de tratamiento de textos, para los cuales la potencia de los procesadores actuales es suficiente. Por lo tanto, si un producto no es demandado por el consumidor, es poco probable que se invierta en el desarrollo de éste; sin embargo, hay un aspecto de la computación que sigue estando impulsado por una necesidad de alta potencia: la industria de juegos, que requiere gran capacidad de procesamiento para los gráficos. Los juegos de computadora precisan tarjetas especiales para ofrecer las imágenes en 3D de vanguardia. Así, la continua exigencia de los jugadores por visualizar imágenes más realistas ha motivado el desarrollo de la generación de tarjetas gráficas a niveles más altos en potencia de cálculo (Chitty et al., 2007).

Los procesadores gráficos han evolucionado hasta llegar a ser muy potentes y flexibles. Las

GPUs modernas procesan cálculos de punto flotante mucho más rápido que las CPUs de hoy, por lo que en los últimos años estas características han atraído a los investigadores para utilizar este dispostivo en aplicaciones no sólo gráficas sino de propósito general (Chitty et al., 2007); además, con el desarrollo de herramientas de programación de GPUs, varios algoritmos han sido adaptados a este hardware satisfactoriamente y la plataforma de computación híbrida GPU - CPU, comparada con las implementaciones en CPUs han alcanzado aceleraciones importantes (Mu et al., 2013). En los últimos años se han implementado los AEs en GPUs y se ha visto que son capaces de mejorar decenas de veces el desempeño, sobre todo cuando se utilizan tamaños grandes de población. Por lo tanto, existe un aumento significativo en la velocidad de programación genética mediante el uso de este hardware (Chitty et al., 2007).

La Figura 1.1 muestra que la implementación de Yu et al. (2005) basada en GPU, es mucho más rápida que la CPU e incrementa su eficiencia mientras mayor sea el número de la población.

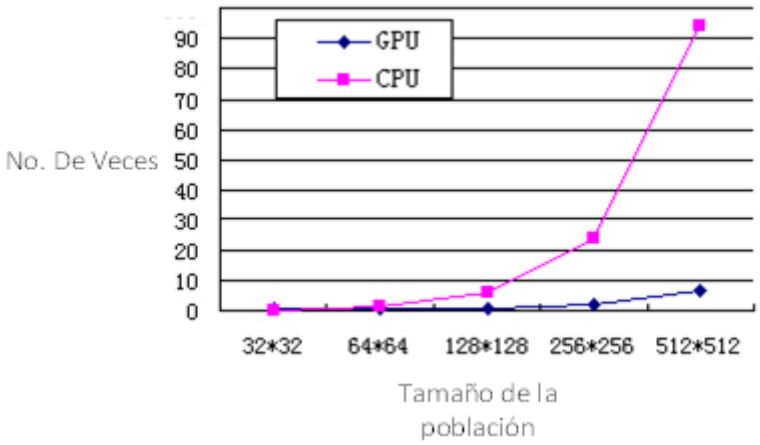

Figura 1.1: Comparación entre la CPU y la GPU

En esta investigación todos los operadores genéticos fueron implementados en la GPU. Las pruebas mostraron que en el mayor tamaño de la población se obtuvo la mejor acelera-

ción. Los autores optaron por esta plataforma de hardware debido a que es potente y de bajo costo y concluyeron que el uso de estas tarjetas es una solución prometedora para diversos algoritmos, ya que tienen mucho mayor rendimiento en comparación con las CPU.

Wong et al. (2005) propusieron un algoritmo de Programación Evolutiva llamado Fast Evolutionary Programming (FEP) para resolver cinco funciones de prueba. En el algoritmo, el bucle principal y la selección de individuos se ejecutaron en la CPU; mientras que la evaluación, la mutación y reproducción se realizaron sobre la GPU, por no requerir información externa. En el caso de la reproducción fue necesaria la interacción de al menos dos individuos. A partir de ciertos tamaños de población obtuvieron una ganancia de velocidad de 5.02x (Nx representa la velocidad de x). Éste es un ejemplo de cómo se subdividen las tareas: las partes secuenciales y/o que necesitan comunicaciones suelen ejecutarse sobre la CPU, mientras que las partes con menos interacción entre ellas, sobre todo en cuanto acceso a datos, suelen paralelizarse para su ejecución sobre la GPU.

Wong and Wong (2006, 2009) presentaron un Algoritmo Genético Híbrido (AGH) y paralelo, donde todo el proceso evolutivo se ejecutó sobre la GPU; sólo la generación de números aleatorios, para la población inicial, se realizó en la CPU por limitaciones de las GPUs. Cada individuo se asignó a un procesador de la GPU y la selección la realizaron escogiendo a uno de sus vecinos de forma probabilista, para llevar a cabo la reproducción. Los autores compararon AGH con AG estándar ejecutándose sobre una CPU y el algoritmo FEP de Wong et al. (2005). Gracias al uso de un nuevo algoritmo de selección pseudo-determinístico en que se redujo la cantidad de números aleatorios transferidos entre la CPU y la GPU, AGH consiguió una mejora de velocidad de 5,30x respecto a la versión secuencial.

El modelo propuesto por Luong et al. (2010) se basó en un rediseño del modelo de islas. Propusieron tres esquemas diferentes: en el primero utilizó un AE de grano grueso que usó un esquema maestro-esclavo para ejecutar la evaluación sobre la GPU. En el segundo se distribuyó la población sobre los procesadores de la GPU. En el último se mejoró la segunda propuesta empleando la memoria más rápida disponible en la GPU. Las dos últimas propuestas consiguieron reducir la latencia entre la CPU y la GPU a costa de tener que ajustar la

mayoría de los parámetros del algoritmo. Finalmente se compararon las velocidades de las implementaciones secuencial y paralela obteniendo como resultado una ganancia de velocidad de 1757x entre el primer y el tercer esquema. En el Congreso mundial de Inteligencia Artificial, Kannan and Ganji (2010) mostraron su trabajo en el descubrimiento de fármacos, usando CUDA. Autodock es una herramienta de descubrimiento de fármacos que usa un algoritmo genético para encontrar la posición de acoplamiento óptimo ligado a una proteína. La aceleración de Autodock pudo ser alcanzada usando CUDA, ya que las características de ejecución resultantes fueron aceleradas aproximadamente 50 veces más en la evaluación de la función de aptitud, y de 10 a 47 veces dependiendo del tamaño de la población. También se presentó una propuesta CGPU (CPU+GPU) para la manipulación de la memoria en grandes proyectos.

En la investigación de Vidal and Alba (2010) ellos trabajaron sobre la arquitectura CUDA y almacenaron en la memoria global de la GPU los individuos, junto con sus valores de evaluación. Tanto la evaluación como los operadores genéticos fueron ejecutados sobre la GPU sin utilizar la CPU. Para la generación de números aleatorios aprovecharon un generador de números pseudoaleatorios proporcionado por el kit de desarrollo de CUDA basado en el método Mersenne twister. Sus experimentos incluyeron problemas de optimización discretos y continuos. Compararon la eficiencia de la versión paralela con respecto a la secuencial ejecutada sobre la CPU y obtuvieron aceleraciones desde 5 hasta 24.

Tsutsui and Fujimoto (2009) propusieron ejecutar un AG sobre una GPU para resolver el problema QAP (del inglés, Quadratic Assignment Problems), utilizando CUDA. Este es uno de los problemas de optimización de permutaciones más difíciles de resolver. Su modelo generaba la población inicial sobre la CPU y después lo copiaba a la memoria de la GPU. Después cada subpoblación se hacía evolucionar sobre la GPU; usaron una tarjeta NVIDIA GeForce GTX285. Tras un cierto número de generaciones las subpoblaciones intercambiaban individuos; los resultados experimentales mostraron ganancias de velocidad de entre 3x y 12x al resolver 8 instancias del problema, en comparación a un Intel i7 965.

Berger and Galea (2013) propusieron algunas estrategias de paralelización para el problema

de la mochila, uno de los 21 problemas NP-completos de Richard Karp y mejor conocido como KP (del inglés, Knapsack Problem), el cual consiste en la selección de un conjunto de elementos con un peso y un valor específicos, para llenar una mochila con capacidad determinada. Su método más eficaz no requirió realizar una transferencia de datos costosa entre la memoria principal y la memoria de la GPU. Además, los valores iniciales de la matriz que contenía el valor óptimo actual de la capacidad de la mochila se ponían a cero por la GPU, después los valores de los elementos (peso y ganancias) se pasaban como argumentos al llamar el kernel, y sólo el valor del último elemento de la matriz se copiaba de la memoria de la GPU a la memoria principal. Para este trabajo se hicieron pruebas en 3 plataformas de Hardware diferentes; la primera, con un procesador AMD Sempron 140 2.7 GHz y una tarjeta GeForce GTX460 de 336 núcleos; la segunda con Intel Xeon X5650 2.67 GHz y una Tesla C2075 de 448 núcleos; y la tercera con Intel Xeon E5-2620 2 GHz, una tarjeta GeForce GTX680 y 1536 núcleos. La aceleración para cada una de ellas fue: de 16.8x a 17.4x; de 13.2x a 15.2x y de 11.8x a 13.3x, respectivamente.

Ueno and Suzumura (2012) usaron la GPGPU para la detección de anomalías en los motores de automóviles. Ellos aceleraron una de las aplicaciones más importantes para el cálculo de datos: el algoritmo SST (del inglés, Singular Spectrum Transformation), cuya parte dominante es el cálculo de la matriz SVD (del inglés, Singular Value Decomposition) con complejidad computacional de $O(n^3)$, donde n es el tamaño de la matriz necesaria para determinar el tamaño de la ventana de los datos destino. En este trabajo, el cálculo de SVD se descompuso en varias etapas: primero la CPU envía la matriz de entrada a la GPU para hacer una bidiagonalización; posteriormente se envían los datos obtenidos a la CPU para calcular los valores simples; enseguida la información es mandada a la GPU para calcular los vectores simples y finalmente la GPU los envía de regreso a la CPU.

En el paralelismo basado en tareas se asignó una actividad en un bloque diferente y con esto se evitó hacer una sincronización, lo cual es relativamente costoso. También de esta manera los bloques se ejecutaron de manera independiente al no tener que hacer uso de los datos de otros bloques. La transferencia de datos de las tarjetas usadas (Tesla C1060 y GeForce 8800

GTS 512) dominaron desde 11.9 % (tamaño de la matriz: 32) al 3,73 % (tamaño de la matriz: 512). Los resultados mostraron una excelente ganancia de desempeño para matrices grandes, en comparación con matrices pequeñas, teniendo que el algoritmo SST para una matriz de tamaño 1000 fue calculada en 4.44 segundos, lo cual resultó 12.44 veces más rápida que una CPU.

El uso del hardware paralelo ha tenido, sin lugar a dudas, múltilples aplicaciones por su alto desempeño. Si bien es cierto que las CPUs modernas integran múltiples núcleos, las GPUs integran cientos de ellos, y su uso es adecuado para procesamiento de propósito general y no sólo gráfico. Desafortunadamente, el paso de una arquitectura secuencial a una paralela no es tan fácil ya que frecuentemente los algoritmos deben ser rediseñados desde cero para maximizar las operaciones que se pueden realizar en paralelo, y por ende aprovechar al máximo la potencia de procesamiento ofrecido por la plataforma. Esto es especialmente cierto para las GPUs, cuyos núcleos se pueden utilizar simultáneamente para realizar cálculos paralelos de datos.

1.7. DESCRIPCIÓN DEL CONTENIDO DEL DOCUMENTO

En la sección II se explican de manera general algunos conceptos de Tomografía Sísmica para entender el problema que se estudia en esta investigación; se definen los Algoritmos inmersos en el campo de la Computación Evolutiva y las características de cada uno de ellos; el significado de Cómputo Paralelo; una breve historia de las GPUs, así mismo el papel que actualmente desempeñan en la solución de problemas de propósito general y finalmente se hace una descripción de la arquitectura CUDA, sobre la cual se implementa la solución al problema

En el capítulo III se hace una descripción del funcionamiento del programa de cobertura de

rayos para realizar, en secciones posteriores, una comparación entre el desempeño de la versión secuencial, que es de la cual se parte, y la versión paralela, obtenida a lo largo de esta investigación.

La sección IV muestra la Evolución Diferencial realizada, y el desarrollo del modelo paralelo para el algoritmo de cobertura de rayos sísmicos. En el capítulo V se presentan las pruebas realizadas sobre el algoritmo, con diferentes tamaños de población; se hace un análisis de los resultados obtenidos después de hacer la paralelización y se realiza la comparación del rendimiento en cada una de las versiones, tanto secuencial como paralela. En el capítulo VI se presentan las conclusiones de este trabajo, así como los trabajos futuros para mejorar los resultados de esta investigación.

Capítulo 2

FUNDAMENTOS DE EVOLUCIÓN DIFERENCIAL Y PROGRAMACIÓN PARALELA CON CUDA

2.1. EVOLUCIÓN DIFERENCIAL

La evolución es un proceso de optimización donde el objetivo principal es mejorar la habilidad de un organismo, o sistema, para sobrevivir en ambientes competitivos y dinámicos (Engelbrecht, 2007).

La computación evolutiva es un área de las ciencias de la computación inspirada en los principales procesos de la evolución natural: reproducción, mutación, competencia y selección. La evolución natural se observa como un proceso de solución a problemas, los cuales están caracterizados por el caos, el azar, la temporalidad y la no linealidad; estas son las características de los problemas que han demostrado ser especialmente intratables por métodos clásicos. De esta manera, los algoritmos evolutivos ofrecen una opción de solución a situaciones donde los métodos clásicos no ofrecen un resultado satisfactorio dado un conjunto

limitado de recursos de cómputo.
Los algoritmos evolutivos están basados en procesos de aprendizaje colectivos dentro de una población de individuos, cada uno de los cuales representa un punto de búsqueda en el espacio de soluciones potenciales para un problema dado. La población es inicializada arbitrariamente y evoluciona hacia mejores regiones del espacio de búsqueda por medio de procesos aleatorios de selección, mutación y recombinación; de los cuales estos dos últimos son en algunas ocasiones completamente omitidos. Durante el proceso evolutivo se genera información de calidad (valor de aptitud) de los puntos de búsqueda, y la selección favorece a aquellos individuos con aptitud más alta para ser reproducidos con más frecuencia que los peores individuos. El mecanismo de recombinación o cruza permite la mezcla de información de padres que hereda a sus descendientes, y la mutación introduce diversidad a la población.

La ED surgió a partir de los intentos de Kenneth Price por resolver el problema del ajuste polinomial de Chebychev. Con el fin de mejorar las soluciones existentes, se emplearon vectores diferencia como mecanismo de perturbación y de esta idea surgió la ED. La idea principal es utilizar la diferencia entre vectores para perturbar a otro vector de la población. Este método ha demostrado converger más rápido y con más certeza que muchos otros métodos de optimización global; requiere de pocas variables de control, es robusto, fácil de implementar y de paralelizar (Méndez, 2008).
Dentro de las características principales se encuentran (Reyes, 2011):

- La capacidad de manejar funciones objetivo no lineales, no diferenciables y multimodales.
- El algoritmo es fácilmente paralelizable y resulta útil cuando la evaluación de la función objetivo es computacionalmente costosa.
- No se requiere predefinir distribuciones de probabilidad como en el caso de las estra-

tegias evolutivas.

- Emplea una codificación real y la precisión está determinada por el formato de punto empleado.
- Suele converger a un valor óptimo (posiblemente local) de manera consistente a lo largo de una secuencia de ejecuciones independientes.

En la ED, al igual que en el resto de los algoritmos evolutivos existen convencionalmente 3 operadores: mutación, recombinación o cruza y selección, pero también existen 3 parámetros de control que desempeñan un rol importante en esta técnica de optimización como: el tamaño de la población (NP, del inglés, Number Population), el factor de escala (F) y la tasa o probabilidad de cruza (CR, del inglés, Crossover Rate) (Qin et al., 2009; Ao and Chi, 2009).
En este algoritmo la recombinación es introducida para incrementar la diversidad de la población (Gao-yang and Ming-guang, 2010; Ao and Chi, 2009).

2.1.1. Mutación

El operador de mutación proporciona diversidad a la población a través de la introducción de nuevas soluciones, evitando la convergencia prematura en el algoritmo; es decir, la posibilidad de una rápida obtención no necesariamente del óptimo global. Para superar este problema es necesario conservar la diversidad en las generaciones, mediante un parámetro denominado factor de escala, elegido por el usuario para controlar la amplificación de la diferencia entre 2 individuos, así como para evitar el estancamiento de la búsqueda permitiendo la exploración de otras áreas en el espacio de búsqueda y evitando el estancamiento de la misma (Sun et al., 2012). Su valor es tomado del rango de 0 a 1 (Reyes, 2011); de 0 a hasta 1.2 (Bujok and Tvrdik, 2011); o bien es establecido a 0.5 (Ao and Chi, 2009).

A diferencia de los Algoritmos Genéticos donde el operador más importante es la cruza, y al igual que en las Estrategias Evolutivas, la mutación es el operador clave en el funcionamiento de una Evolución Diferencial.

2.1.2. Recombinación o Cruza

La recombinación o cruza es una de las operaciones evolutivas implicadas en la generación de nuevos individuos. Esta operación es llevada a cabo con la participación de dos o más padres, quienes heredan rasgos o características a sus descendientes mediante una mezcla de información que se da de manera aleatoria. Existe la posibilidad de que los padres sobrevivan para formar parte de la siguiente generación, esto se da en caso de que la aptitud sea mejor que la de los hijos.
Aunque este operador no es el principal en una ED, a diferencia de un AG, éste se implementa debido a que es parte de los principios originales de los AEs. Esta operación depende de un parámetro muy importante: la tasa de recombinación o cruza, que de acuerdo a la literatura consultada debe ser baja, por ejemplo 0.2.

Recombinación Binomial

En este tipo de cruza se reemplazan los elementos del vector x_i usando la regla 2.1.1:

$$Y_j = \begin{cases} v_j & \text{si } U_j \leq \text{CR} \quad \text{o } j = l \\ x_{ij} & \text{si } U_j > \text{CR} \quad \text{y } j \neq l \end{cases} \tag{2.1.1}$$

Donde l es un número entero elegido aleatoriamente desde 1 hasta n y $U_1, U_2, ...U_n$ son variables aleatorias independientes distribuidas uniformemente entre 0 y 1. CR es un parámetro que influye en el número de elementos a los que se le aplica este operador, y va de 0 a 1.

La ecuación 2.1.1 asegura que al menos un elemento de x_i sea cambiado aún si CR=0

(Tvrdik, 2008).

Recombinación Exponencial

Esta recombinación, según la muestra Reyes (2011), es semejante a la de un punto en los AGs, donde se elige a *p*, un número elegido aleatoriamente desde 1 hasta n y del vector padre1 se toman los valores desde 1 hasta *p* para formar la primera parte del hijo, y los valores desde p+1 hasta n en Padre2 para completar al descendiente.

2.1.3. Estrategias de la ED

La selección de las variantes de mutación y los parámetros son el tema más importante en la investigación de este algoritmo y constituyen sus diversas estrategias (Ao and Chi, 2009). Éstas difieren en la forma en que el vector objetivo es seleccionado, el número de vectores diferencia que es usado y la forma en que los puntos de cruza son determinados. Para caracterizar estas variaciones, una notación general fue adoptada en la literatura, llamada DE/x/y/z (Engelbrecht, 2007) en la que *x* se refiere al método de selección del vector objetivo, que puede ser aleatorio: rand; eligiendo al mejor: best; la combinación de las 2 anteriores: rand-to-best; o bien, aquel donde un vector diferencia es calculado a partir de un vector padre y el mejor, mientras el resto de los vectores son calculados usando vectores seleccionados aleatoriamente: current-to-best. La notación *y* indica el número de vectores diferencia y *z* indica el método de cruza usado (binomial o exponencial). De esta manera, se tienen como ejemplo las siguientes estrategias:

- DE/rand/1
- DE/best/1
- DE/best/2

2.2. HISTORIA DE LAS GPUs

Desde 2003, la industria de los semiconductores se ha asentado sobre dos trayectorias principales para el diseño de microprocesador. La trayectoria multinúcleo busca mantener la velocidad de ejecución de los programas secuenciales mientras se mueve en varios núcleos. Los multinúcleos comenzaron como procesadores de doble núcleo, y este número se duplica en cada generación. Un modelo actual es el microprocesador reciente Intel Core i7, que tiene cuatro núcleos. Éste admite *hyperthreading* con dos subprocesos de hardware y está diseñado para maximizar la velocidad de ejecución de programas secuenciales. En contraste, la trayectoria de muchos núcleos se centra más en el rendimiento de ejecución de aplicaciones paralelas. Comenzó como un gran número de núcleos muy pequeños, y de igual manera este número se duplica en cada generación. Un modelo actual es la GPU NVIDIA GeForce GTX 680 con 1536 núcleos, cada uno de los cuales es multiproceso; es decir, es un procesador de instrucciones simples ordenadas que comparten su control y la caché de instrucciones con otros siete núcleos. Los procesadores de muchos núcleos, especialmente las GPUs, han liderado la carrera de rendimiento de punto flotante desde 2003; este fenómeno se muestra en la Figura 2.1 (NVIDIA Corporation, 2014). Si bien la mejora en el rendimiento de los microprocesadores de propósito general se ha reducido de manera significativa, las GPUs han seguido mejorando. A partir de 2009 la relación entre GPUs de muchos núcleos y CPUs de múltiples núcleos, para un rendimiento de cálculo en coma flotante es de 10 a 1.

La gran diferencia de rendimiento entre la ejecución en paralelo y secuencial radica en las filosofías de diseño fundamentales entre los dos tipos de procesadores, como se muestra en la Figura 2.2 y ha motivado a muchos desarrolladores de aplicaciones a mover las piezas computacionalmente intensivas de su software a GPUs; como era de esperar, estas partes también son el blanco principal de la programación paralela, cuando hay más trabajo que hacer, hay más oportunidad de dividirlo entre los trabajadores para cooperar en paralelo.

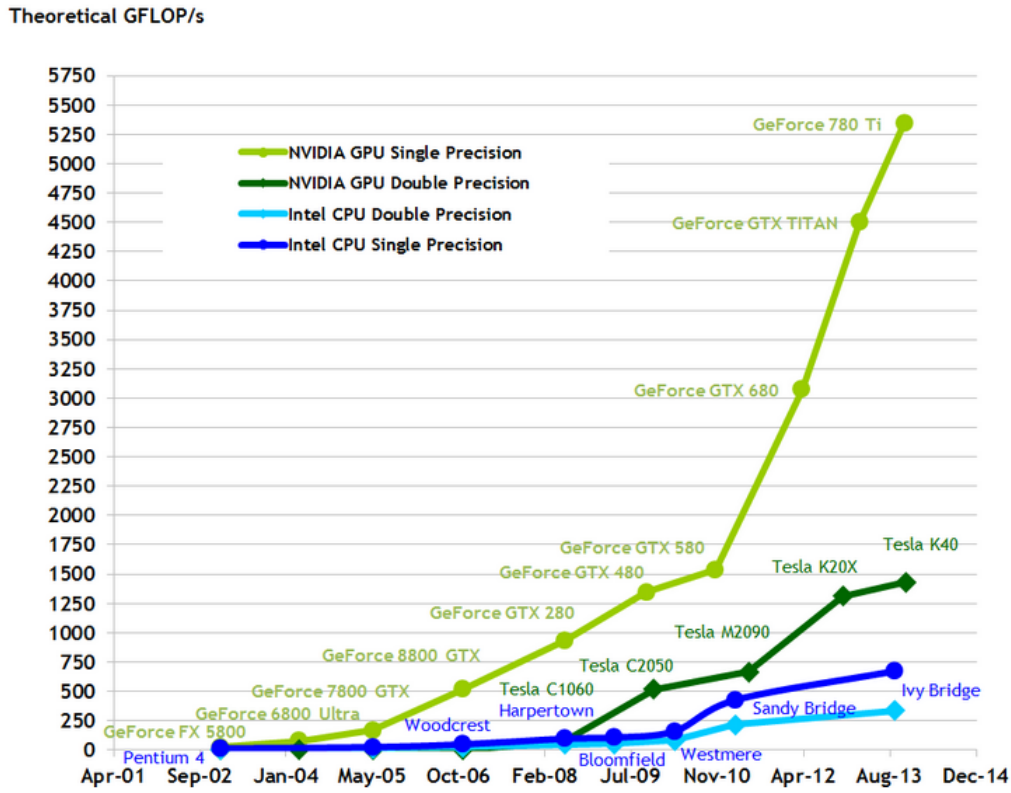

Figura 2.1: Desempeño entre GPUs y CPUs

2.3. CÓMPUTO PARALELO

El cómputo paralelo consiste en el uso activo de varias unidades de procesamiento de manera simultánea para resolver un problema dado. Una tarea es dividida en múltiples subtareas usando la técnica de divide y vencerás, y cada una de ellas es manejada en una unidad de procesamiento diferente.

Muchas aplicaciones actualmente requieren mucho mayor poder de cómputo del que una computadora secuencial es capaz de ofrecer. El cómputo paralelo ofrece la distribución del trabajo entre diferentes unidades de procesamiento, resultando mayor poder de cómputo y rendimiento del que se puede obtener mediante un sistema tradicional de un procesador. El desarrollo del cómputo paralelo ha sido influenciado e impulsado por diferentes factores, de entre los que destacan los siguientes (Kirk and Hwu, 2010):

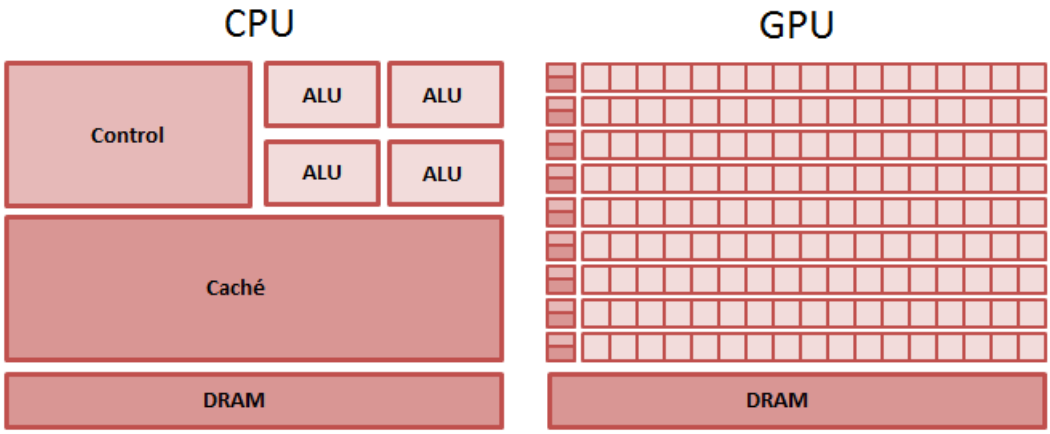

Figura 2.2: Diferencia del diseño de la CPU y la GPU

- Los requerimientos de cómputo son cada vez mayores, tanto en las aplicaciones de negocios, como en las aplicaciones científicas. Entre los problemas que precisan mayor poder de cómputo se encuentran aquellos relacionados con las ciencias de la vida, aeroespaciales, sistemas de información geográfica, análisis y diseño de ingeniería, etc..

- Las limitaciones físicas de las arquitecturas secuenciales, debido a que aunque cada vez se reduce más el tamaño de los transistores para obtener una mayor capacidad de cómputo, la industria se está acercando cada vez más al límite físico posible.

- El uso del cómputo paralelo se encuentra en un estado maduro y puede ser explotado comercialmente debido a que ya existe un avance significativo en la investigación y desarrollo de herramientas, y ambientes para este fin.

- El avance significativo en la tecnología de redes de interconexión, el cual permite comunicaciones cada vez más rápidas.

Clasificación

Existen diferentes clasificaciones de arquitecturas paralelas. Una de las más conocidas es la clasificación propuesta por Flynn (Jadhav, 2007), quien se basa en el número de instrucciones que se ejecutan y en el número de datos sobre los que se aplican estas instrucciones:

- SISD (Single Instruction, Single Data). En este caso se habla de una computadora secuencial en la que no hay paralelismo en las instrucciones, ni en el flujo de los datos. Un ejemplo son las computadoras tradicionales con un solo procesador.
- MISD (Multiple Instruction, Single Data). Esta arquitectura es poco común, en ella, múltiples instrucciones se aplican a un solo flujo de datos. Esta arquitectura se usa en situaciones de paralelismo redundante, en donde se requieren sistemas tolerantes a fallas.
- SIMD (Single Instruction, Multiple Data). La arquitectura SIMD es aquella en la que una misma instrucción es ejecutada sobre un conjunto de datos distintos. En esta arquitectura se cuenta con varios procesadores capaces de ejecutar la misma instrucción simultáneamente, sobre un segmento de datos que previamente fue asignado a cada procesador. Esta arquitectura es muy utilizada en las aplicaciones científicas debido a que en ellas suele requerirse un gran número de operaciones sobre matrices y vectores. Ejemplos de este tipo de arquitectura son las GPUs o las máquinas vectoriales.
- MIMD (Multiple Instruction, Multiple Data). En las computadoras MIMD se realiza la ejecución simultánea de un grupo de instrucciones sobre distintos flujos de datos. Los sistemas distribuidos se ubican dentro de este tipo de arquitectura. A su vez, este tipo de arquitectura puede subdividirse en MIMD de memoria compartida, en caso de que todos los procesadores puedan acceder a todas las regiones de la memoria disponible; o MIMD de memoria distribuida, en caso de que cada procesador posea su propia región

de memoria y no pueda acceder directamente a los datos contenidos en la memoria de otro procesador (Mattson et al., 2004).

Lenguajes de programación paralela y modelos

Muchos lenguajes de programación paralela y modelos han sido propuestos en las últimas décadas. Los que son más ampliamente utilizados son los Message Passing Interface (MPI) para la computación del clúster escalable y OpenMP de memoria compartida para sistemas multiprocesador. MPI es un modelo en el que los nodos en un clúster de computación no comparten memoria, todo el intercambio de datos y la interacción debe hacerse a través de paso de mensajes explícitos. MPI ha tenido éxito en el dominio de la computación de alto rendimiento científico. Las aplicaciones escritas en MPI se han distinguido por ejecutarse con éxito en los sistemas de grupo con más de 100 000 nodos. La cantidad de esfuerzo necesario para trasladar una aplicación en MPI, sin embargo, puede ser extremadamente alto debido a la falta de memoria compartida a través de nodos de computación. CUDA, por otro lado, proporciona la memoria compartida para la ejecución en paralelo en la GPU para hacer frente a esta dificultad. En cuanto a la comunicación entre la CPU y la GPU, CUDA proporciona capacidad limitada de memoria compartida, por tal razón los programadores necesitan, para gestionar la transferencia de datos entre la CPU y la GPU, el paso de mensajes, una función cuya ausencia en MPI ha sido históricamente considerado como una debilidad importante.

OpenMP da soporte de memoria compartida, por lo que ofrece las mismas ventajas que CUDA en los esfuerzos de programación; sin embargo, no ha sido capaz de escalar más allá de un par de cientos de nodos de computación, debido a la administración de hilos y a los requerimientos de hardware para la caché. CUDA consigue una escalabilidad mucho mayor con una administración sencilla, hilos de baja sobrecarga y sin requisitos de hardware para caché; y aunque no da soporte a una amplia gama de aplicaciones como OpenMP debido a estos intercambios de escalabilidad, muchas súper aplicaciones encajan bien en el modelo

simple de administración de hilos de CUDA y así aprovechan la escalabilidad y el rendimiento.
Los aspectos de CUDA son similares tanto a MPI como a OpenMP en que el programador maneja las construcciones de código paralelos, aunque los compiladores OpenMP hacen más de la automatización en la gestión de la ejecución en paralelo. Varios esfuerzos de investigación en curso pretenden añadir una mayor automatización de la gestión de paralelismo y optimización del rendimiento de la cadena de herramientas CUDA. Los desarrolladores que tienen experiencia con MPI y OpenMP encontrarán CUDA fácil de aprender. Especialmente, muchas de las técnicas de optimización del rendimiento son comunes entre estos modelos (Kirk and Hwu, 2010).

2.4. CUDA

CUDA es un lenguaje y conjunto de herramientas de programación desarrollado por la empresa NVIDIA para programar sus GPUs. Consta de un conjunto de aplicaciones y un compilador que permite a los programadores usar una extensión del lenguaje C y C++ para programar algoritmos que se ejecutarán sobre una GPU NVIDIA; además CUDA puede utilizarse también con los lenguajes Python, Fortran y Java.
Por medio de CUDA un bloque de código puede aislarse como una función y compilarse para ser ejecutado en la tarjeta gráfica por uno o más hilos.

En noviembre de 2006, NVIDIA dio a conocer su tarjeta gráfica GeForce 8800 GTX, que fue la primera GPU que soportaba DirectX 10. La GeForce 8800 GTX fue también la primera GPU en ser construida con la arquitectura CUDA de Nvidia. Esta arquitectura incluyó nuevos componentes diseñados exclusivamente para el cómputo con GPU y con el objetivo de resolver las limitaciones que impedían a los procesadores gráficos anteriores ser útiles para el cómputo de propósito general.

La arquitectura de NVIDIA está construida en torno a un conjunto escalable de bloques de procesadores (SMs, del inglés Streaming Multiprocessors) multihilos. Cuando un programa de CUDA en el Host invoca una cuadrícula kernel (grid), los bloques de la cuadrícula son enumerados y se distribuyen entre los multiprocesadores con capacidad de ejecución disponible. Los hilos de un bloque se ejecutan simultáneamente en un multiprocesador, y múltiples bloques de hilos pueden ejecutarse simultáneamente en un multiprocesador. Cuando un conjunto de bloques terminan de procesarse, nuevos bloques son lanzados sobre los multiprocesadores desocupados. Un multiprocesador está diseñado para ejecutar cientos de hilos simultáneamente. Para gestionar una gran cantidad de hilos, se emplea la arquitectura SIMT (Single Instruction, Multiple Thread) donde el multiprocesador crea, gestiona, programa, y ejecuta las hilos en grupos paralelos de 32 llamados warps (NVIDIA Corporation, 2014). Cada warp consta de 32 hilos con identificadores consecutivos, por ejemplo: los hilos con identificador (ThreadId) del 0 al 31 forman el warp 1, del 32 al 63 forman el warp 2, y así sucesivamente (Kirk and Hwu, 2010). El número de warps que pueden residir y ser procesados en un SM depende de la capacidad de cómputo de la GPU. El número total de warps por bloque es calculado mediante la fórmula 2.4.1 (NVIDIA Corporation, 2014):

$$n = \lceil (T/W, 1) \rceil \tag{2.4.1}$$

Donde T es el número de hilos por bloque, W es el tamaño del warp, el cual es 32 y *ceil(x,y)* es x redondeada al número inmediato superior múltiplo de y.

Debido a que NVIDIA buscó que esta nueva familia de procesadores gráficos fuera utilizada para cómputo de propósito general, cada una de las Unidades Aritméticas Lógicas (ALUs) fue construida cumpliendo los requisitos de la IEEE para aritmética de punto flotante de precisión simple, y fueron diseñadas para usar un conjunto de instrucciones para cómputo de propósito general y no específicamente para los gráficos. Además, las unidades de ejecución de la GPU pueden acceder arbitrariamente a la lectura y escritura de la memoria, así como a una caché conocido como memoria compartida (Sanders and Kandrot, 2010).

Para entender el entorno CUDA es necesario tener claros algunos conceptos (Orocio et al., 2012):

- HOST: Unidad Central de Procesamiento (CPU).
- DEVICE: Unidad de Procesamiento de Gráficos (GPU).
- KERNEL: función que se ejecuta de forma paralela en la GPU.
- THREAD o HILO: unidad mínima de ejecución en la GPU.
- BLOCK o BLOQUE: conjunto de hilos agrupados, el número de hilos dentro de un bloque es limitado y está definido por las características de la GPU. Cada hilo dentro de un bloque mantiene un identificador único.
- GRID o CUADRÍCULA: es a su vez un conjunto de bloques, el tamaño de un GRID es también limitado, pero permite que una gran cantidad de threads sean ejecutados en una sola llamada a un Kernel. Un bloque dentro de un grid es también identificado mediante un ID.

En la Figura 2.3, se puede apreciar un esquema de organización de threads en la tecnología CUDA.

Los Threads durante su ejecución pueden acceder a distintos espacios de memoria:

1. La memoria local (Local Memory). Es aquella que se le asigna a cada hilo. Este tipo de memoria se encuentra físicamente en la memoria DRAM del dispositivo.
2. Memoria compartida (Shared Memory). Es aquella que se le asigna a cada hilo y se encuentra físicamente en la tarjeta de video.

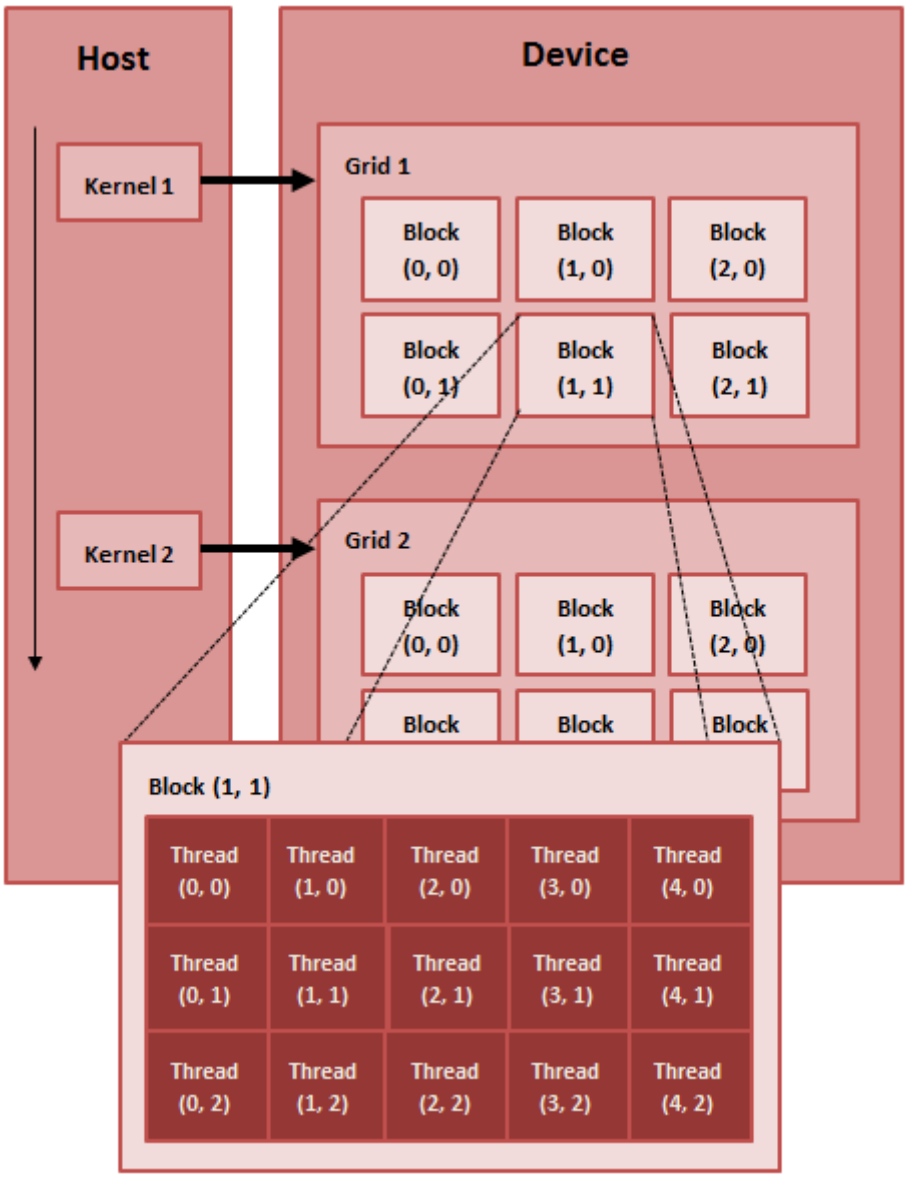

Figura 2.3: Organización de hilos en CUDA

3. Memoria global (Global Memory). Es aquella que sirve para comunicar a los hilos de todos los bloques y la CPU. Al igual que los hilos, esta memoria es accesible por el Host.

4. Memoria constante (Constant Memory) y Memoria de textura (Texture Memory). Son utilizadas para el procesamiento y visualización de imágenes y son sólo de lectura para

los hilos. Este par de memorias son especializadas, pues son del tipo memoria caché.

Toda esta jerarquía de memoria se puede apreciar en la Figura 2.4.

En CUDA se tienen una serie de calificadores que indican cierta particularidad para sus fun-

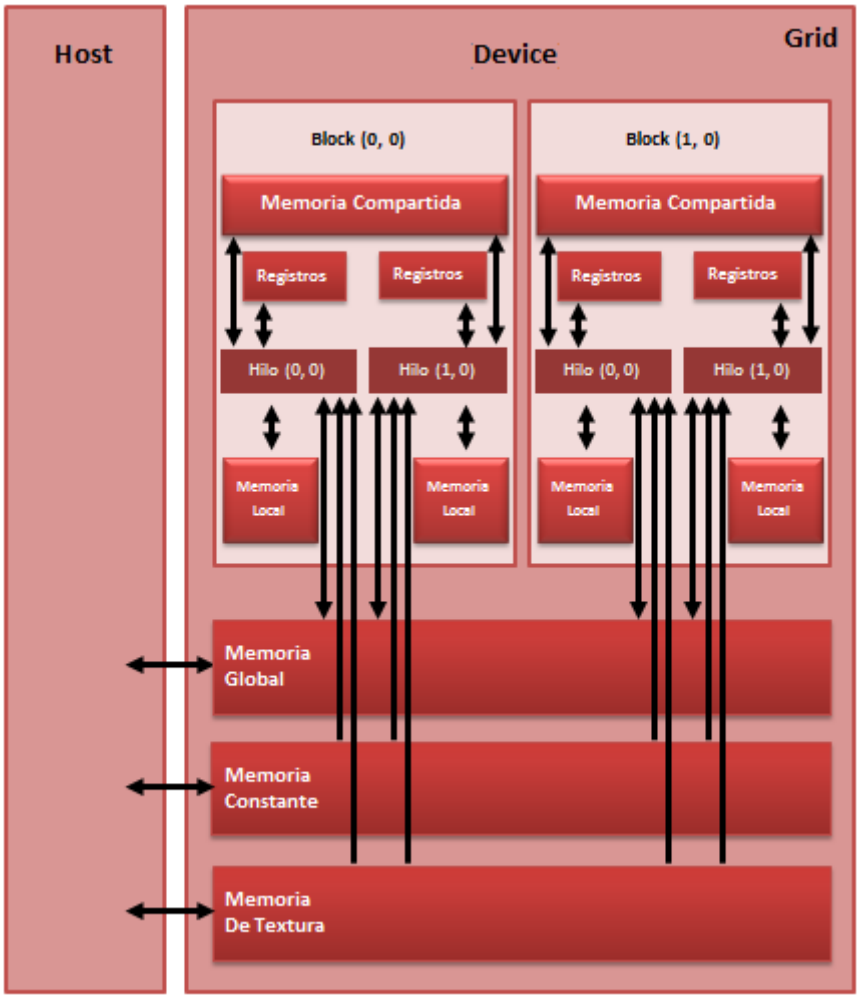

Figura 2.4: Organización de los diferentes tipos de memoria en una GPU NVIDIA, y sus diferentes niveles de acceso

ciones; éstas se deben anteponer a la declaración de dicha función y definen las siguientes características:

- _device_ : indica que una función es invocada y ejecutada desde la GPU.
- _global_ : indica que dicha función será llamada desde el Host y ejecutada en el GPU; se utiliza este calificador para caracterizar a las funciones Kernel en un programa CUDA.
- _host_ : indica que la función es invocada y ejecutada desde el Host.

Una parte importante en los programas realizados en CUDA es la transferencia de información entre la memoria del CPU y la memoria global del GPU. Para ello se incluye en el lenguaje ciertas estructuras que realizan la copia de datos en diferentes flujos.

- CudaMemcpyHostToHost: transfiere datos desde el Host al mismo Host.
- CudaMemcpyDeviceToDevice: transfiere datos desde el Device al mismo Device.
- CudaMemcpyHostToDevice: transfiere datos desde el Host hacia el Device.
- CudaMemcpyDeviceToHost: transfiere datos desde el Device hacia el Host.

Capítulo 3

DESCRIPCIÓN DEL ALGORITMO DE TOMOGRAFÍA SÍSMICA

En este capítulo se describe el concepto de Tomografía Sísmica y se explica, de manera general, el funcionamiento del ATS a paralelizar en este trabajo de investigación.

3.1. TOMOGRAFÍA SÍSMICA

La tomografía sísmica es una técnica de imagen que procesa observaciones del movimiento de la tierra, recolectadas con sismómetros, para mejorar los modelos estructurales, y ha sido uno de los medios más efectivos para obtener imágenes del interior del planeta en las últimas décadas (Lee et al., 2013). En esta área se analiza la variación de los tiempos de llegada de las ondas sísmicas generadas por fuentes naturales o artificiales, y registradas por los sismógrafos; consiguiendo un modelado en 3D de la distribución de sus velocidades en la tierra, para lo cual se precisa una programación compleja y una elevada potencia de cálculo. Las ondas sísmicas tienen información importante sobre las propiedades físicas de las estructuras del subsuelo a través del cual se propagan. El tiempo de viaje de una onda está en

función de la distribución de velocidades en su trayectoria. Si se conoce la distribución de velocidades en la Tierra en capas concéntricas, se establece un modelo que predice los tiempos de viaje de las ondas sísmicas a cualquier distancia epicentral. Si la velocidad es mucho mayor, el tiempo de llegada es mucho menor, y viceversa; así se clasifican las llegadas como tempranas o tardías (Instituto Andaluz de Geofísica, 2012).

Gran parte del conocimiento que se tiene de la estructura interna de la Tierra ha sido proporcionado mediante el análisis de las ondas sísmicas. Las estructuras de los materiales que conforman el planeta son reflejadas en el campo de ondas que se propaga en su interior al ocurrir un terremoto o al generarse un campo de ondas mediante una fuente artificial. Las imágenes tridimensionales obtenidas mediante la tomografía sísmica han contribuido de forma espectacular a un mayor conocimiento de las propiedades y composición de los materiales que constituyen la corteza, manto y núcleo terrestres, así como a la determinación precisa de anomalías sísmicas o discontinuidades a escala local y global (Óscar Pintos, 2012).

El problema de tomografía sísmica es frecuentemente formulado como un problema de optimización, en el que se busca un modelo óptimo de la estructura de la tierra que minimice una función objetivo donde se cuantifica la diferencia entre los campos de ondas observados y los campos de ondas sintéticos, usando un modelo de la estructura de la tierra como referencia (Lee et al., 2013).

3.2. ALGORITMO DE COBERTURA DE RAYOS SÍSMICOS

Para iniciar, es importante mencionar que el volumen para este caso de estudio, cuyas dimensiones son 231 Km., 26 Km. y 69 Km. de longitud, amplitud y profundidad, respectivamente, es discretizado en espacios de 1 km. con la finalidad de generar una rejilla por donde los rayos atraviesan determinadas celdas hasta llegar a los receptores. En la figura 3.1 se aprecia una representación del modelo discreto.

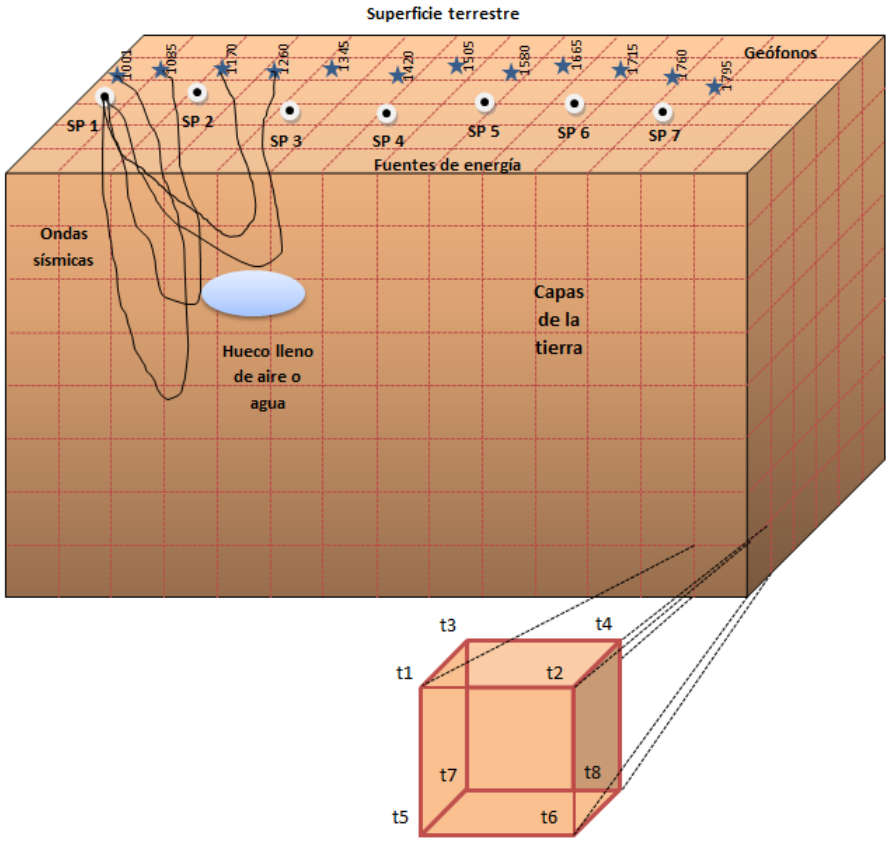

Figura 3.1: Modelo discreto del volumen de la corteza terrestre

Los objetivos del ATS son: obtener la diferencia entre el tiempo de llegada observado por un geólogo y el calculado por el programa (dt), misma que se busca minimizar en medida de lo posible; encontrar la ruta de los rayos generados a partir de cada fuente de energía hacia los dispositivos receptores, dadas las ubicaciones en 3 dimensiones para cada uno de ellos; y contabilizar el número de rayos que pasan en las celdas. En el algoritmo se lleva a cabo, en primer lugar, la lectura de datos como: las dimensiones del volumen; la posición de las fuentes de energía y de los geófonos; y los tiempos de llegada observados por el experto. Se hallan las celdas que contienen a las fuentes de energía y a los geófonos, mediante sus datos de ubicación en las 3 dimensiones. A continuación se calcula el tiempo de llegada del frente de onda desde la fuente al receptor, mediante interpolación trilineal con los tiempos

de llegada del frente de onda de cada uno de los 8 vértices de la celda donde se encuentra el geófono; una vez que se ha obtenido este dato, se puede realizar el cálculo del tiempo residual mediante la fórmula 3.2.1:

$$dt=t_o\text{-}t_c \tag{3.2.1}$$

Para encontrar la ruta de los rayos generados por el frente de onda, es necesario hacer el trazo desde el receptor hacia la fuente de energía. Es preciso calcular el gradiente del rayo, con respecto al tiempo, en las 3 dimensiones como lo muestra la fórmula 3.2.2:

$$\nabla t = \frac{\partial t}{\partial x}, \frac{\partial t}{\partial y}, \frac{\partial t}{\partial z} \tag{3.2.2}$$

Para realizar el cálculo es necesario tomar en cuenta los 8 vértices de cada celda por donde esté pasando el rayo. Para cada una de las dimensiones del modelo discreto se aplica la ecuación 3.2.3:

$$\frac{1}{4h}\sum_{i=1}^{4}\Delta t \tag{3.2.3}$$

Donde: $\Delta t = t_i - t_{i-1}$.

Según la representación de una celda en la figura 3.2, y de acuerdo a la ecuación 3.2.3, el valor del gradiente en el eje X, se calcula mediante la diferencia de los valores de los vértices de la arista derecha y la izquierda y haciendo posteriormente un promedio entre los vértices, que son multiplicados previamente por el tamaño de la celda; el valor del gradiente en el eje Y, sigue el mismo proceso sólo que las aristas que se consideran son la de atrás y la de adelante; y finalmente, para el valor del gradiente en el eje Z, se consideran las aristas de arriba y de abajo de la celda.

Posteriormente, se analizan cada uno de los valores obtenidos con la finalidad de determinar de qué celda proviene el rayo, y cuál será la siguiente a evaluar. Así mismo, se van contabilizando las celdas pertenecientes a la ruta del rayo, y si el número es mayor al límite

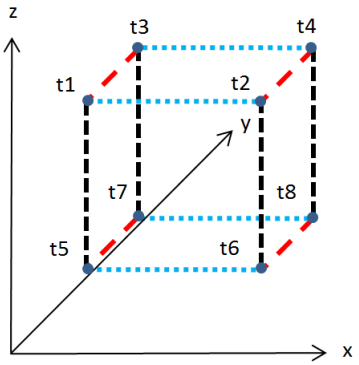

Figura 3.2: Representación de una celda por donde está pasando un rayo sísmico

que se considera que un rayo puede recorrer, se concluye que éste salió de las dimensiones del volumen, por lo que no se puede continuar trazando, y por ende es descartado en el algoritmo. Una vez que se ha terminado la tarea anterior, se eleva al cuadrado el valor de *dt* y se va acumulando, así mismo se va contando el número de rayos completos. Este proceso se lleva a cabo en cada uno de los geófonos, para cada una de las fuentes de energía, y cuando finalmente se han evaluado todos, se obtiene el valor cuadrático medio (RMS, del inglés, Root Mean Square) mediante la fórmula 3.2.4:

$$\sqrt{\frac{1}{N}\sum_{j=1}^{m}\sum_{n=1}^{n_j}(t_o - t_c)^2} \tag{3.2.4}$$

Donde: N es el Total de geófonos, m es el número de shotpoints, y n_j es el número de geófonos evaluados por el shotpoint.

Capítulo 4

ED APLICADA EN TOMOGRAFÍA SÍSMICA

En esta sección se describe el Algoritmo de Evolución Diferencial usado para el problema de estudio, el método para la paralelización del algoritmo de cobertura de rayos sobre la GPU, y la integración de ambos.

4.1. ALGORITMO DE EVOLUCIÓN DIFERENCIAL

Se desarrolló un AE, debido a que este tipo de métodos de búsqueda tienen la ventaja de no depender de una condición inicial; la cual si es mala, puede afectar el desempeño y la eficiencia como en el caso de un método basado en el gradiente. De manera particular se eligió una Evolución Diferencial, técnica robusta de búsqueda, precisa y relativamente rápida, capaz de tratar funciones objetivo de gran dimensión con una proporción relativamente pequeña en los tamaños de las poblaciones. El método de Evolución Diferencial posee ventajas sobre el resto de los AEs, tales como simplicidad y versatilidad (González, 2011); además de que, aunque es una técnica de búsqueda reciente en el campo de la Computación Evolutiva,

ha tenido buen desempeño (Sun et al., 2012; Gao-yang and Ming-guang, 2010). En la figura 4.1 se muestra el diagrama de flujo de una ED.

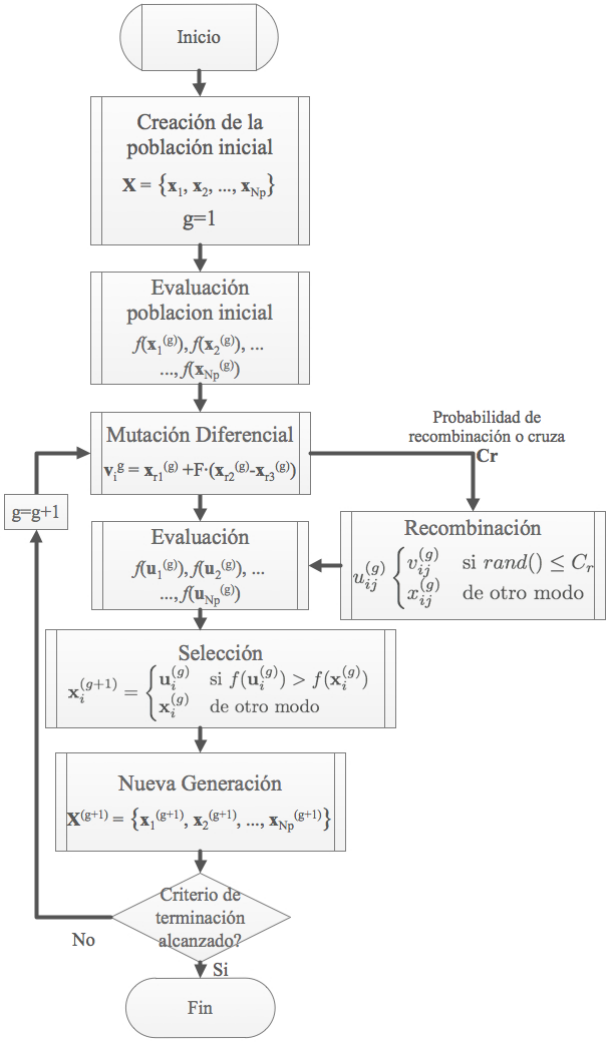

Figura 4.1: Diagrama de flujo de una ED.

4.1.1. Representación de los individuos

En el algoritmo, cada individuo de la población representa un conjunto de profundidades y velocidades en diferentes capas de la tierra, para este caso se tomaron 8 puntos de

la profundidad debido a que anteriormente, con el uso de Estrategias Evolutivas, ha dado mejores resultados en comparación con otros valores (Ramírez Cruz et al., 2013); sin embargo, este número puede variar de acuerdo a elección de los usuarios. En la Figura 4.2 se representa un individuo, el cual al ser visto como un vector está dividido en dos partes. En la primera, los valores 1 y 69 ocupan las posiciones 0 y 7, respectivamente, correspondientes al mínimo y máximo de la profundidad en kms, y las 6 posiciones intermedias son datos de tipo entero, diferentes entre sí, y que deben ser ordenados de menor a mayor. En la segunda parte del individuo se asignan las velocidades en un rango de 3 a 8 km/s; que cumplen con las mismas restricciones de la primera pero son valores reales. Las velocidades para las 61 capas restantes de las 69 en total se calculan mediante interpolación lineal.

La representación de cada individuo muestra que a cada valor de profundidad le corresponde una velocidad, debido a que la finalidad fue generar un modelo de capas planas, donde cualquier punto dentro de toda la superficie de una capa en determinado valor de profundidad tiene la misma velocidad; y aunque se sabe que en la realidad no es así, es usado de esta manera obtener un modelo inicial de velocidades y realizar posteriormente una simulación de frente de onda.

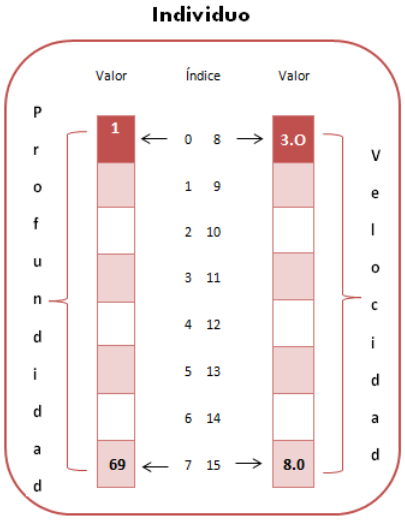

Figura 4.2: Representación de un individuo

4.1.2. Mutación

Para el algoritmo desarrollado en este trabajo se asignó a F un valor aleatorio entre 0 y 1. La mutación se implementó mediante la siguiente ecuación 4.1.1

$$\vec{v}_j^i = \vec{x}_{r1}^i + F \cdot (\vec{x}_{r2}^i - \vec{x}_{r3}^i) \tag{4.1.1}$$

Donde *r1, r2, r3* son valores enteros desde 0 hasta el número de individuos de la población, diferentes entre sí y diferentes del índice en funcionamiento. En este trabajo el 80 % del total de la población fue generada mediante este operador.

4.1.3. Recombinación o Cruza

Para llevar a acabo esta operación la tasa de recombinación o cruza se estableció en 0.2. Se eligieron 2 padres al azar y se generó 1 hijo mediante la ecuación 4.1.2:

$$Hijo_i = \begin{cases} Padre1_i & \text{si } rand \in (0,1) \geq 0.5 \\ Padre2_i & \text{en caso contrario} \end{cases} \tag{4.1.2}$$

En este algoritmo el 20 % del total de la población fue generada mediante este operador. En la figura 4.3 se representa un ejemplo del proceso de recombinación.

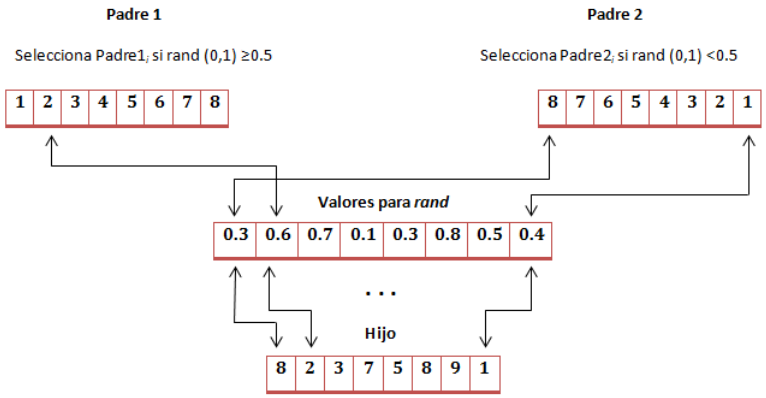

Figura 4.3: Proceso de recombinación

4.1.4. Selección

Cada vez que un individuo x_i' fue generado, se evaluó su modelo de velocidades para determinar si formaría parte de la siguiente generación o no. Debido a que el problema en cuestión es de minimización, si el valor de la evaluación $f(x_i')$ era menor que el de su padre $f(x_i)$, este individuo pasaba a la siguiente generación; de lo contrario, el padre era elegido x_i.

4.1.5. Función de Evaluación

La función de evaluación de este algoritmo consistió en medir los tiempos residuales (diferencia entre el tiempo de llegada del rayo sísmico a cada geófono calculado y el tiempo

observado). Para obtener el tiempo de llegada de cada uno de los geófonos se realizó una simulación de la propagación del frente de onda dentro del volumen de estudio, empleando el modelo de Vidale (1990). Se leyeron los datos de las ubicaciones de las 7 fuentes de energía, de los cientos de receptores distribuidos y de los tiempos de llegada de la onda sísmica, en base a un experimento de campo realizado por un geólogo. Para calcular el tiempo de llegada dentro de la celda donde se encontraba el geófono, se usó una interpolación tri-lineal utilizando los tiempos de llegada del frente de onda a los ocho vértices de la celda. Estos dos procesos son parte del ATS propuesto por Vidale (1990) y Hole (1992). De acuerdo a los valores iniciales de la velocidad y la profundidad (representadas en cada uno de los individuos), se generó mediante el método Vidale (1990) un modelo inicial de tiempos de llegada del frente de onda y a continuación se trazaron las trayectorias de los rayos sísmicos siguiendo la dirección del gradiente de frente de onda en cada celda, desde el receptor a la fuente. Después de que se generaron los tiempos de viaje, se calculó el tiempo residual mediante la diferencia del tiempo observado y el calculado. Una vez realizadas estas operaciones para cada rayo que atravesaba el modelo desde el geófono hasta la fuente, se repitió el proceso para el resto de las fuentes y de los receptores y se obtuvo el RMS.

4.2. PARALELIZACIÓN DEL ALGORITMO DE COBERTURA DE RAYOS E INTEGRACIÓN DEL ALGORITMO EVOLUTIVO EN LA GPU

Como se vio en el capítulo tres, el algoritmo de cobertura de rayos consiste en calcular el tiempo de llegada del frente de onda desde las fuentes de energía hacia cada uno de los geófonos localizados en una capa cercana a la superficie terrestre a evaluar (tc), y hallar la diferencia entre éste y el tiempo observado en el experimento (dt). Además, el algoritmo

verifica que, según el modelo de velocidades propuesto por la evolución diferencial, cada uno de los rayos sísmicos llegue hasta los geófonos, cruzando un conjunto de celdas internas del modelo. En el algoritmo secuencial, el tiempo de llegada al geófono, así como el cálculo de la trayectoria entre el geófono y el shotpoint es realizado geófono por geófono, ralentizando el proceso de cálculo, debido a que éste debe terminar antes de empezar con el siguiente.

En este trabajo, se hizo la distribución del cálculo de dt y la trayectoria de los rayos sísmicos sobre un hilo de procesamiento dentro de una GPU, permitiendo la ejecución de este proceso de manera paralela. Se creo la mínima cantidad de hilos posible dentro de cada bloque con el propósito de que cada uno de ellos fuera asignado a un solo procesador dentro del bloque y se lleve a cabo una ejecución realmente paralela debido a que en la arquitectura de CUDA se procesan 32 hilos.
El modelo paralelo usado en este trabajo de investigación se muestra en la figura 4.4.

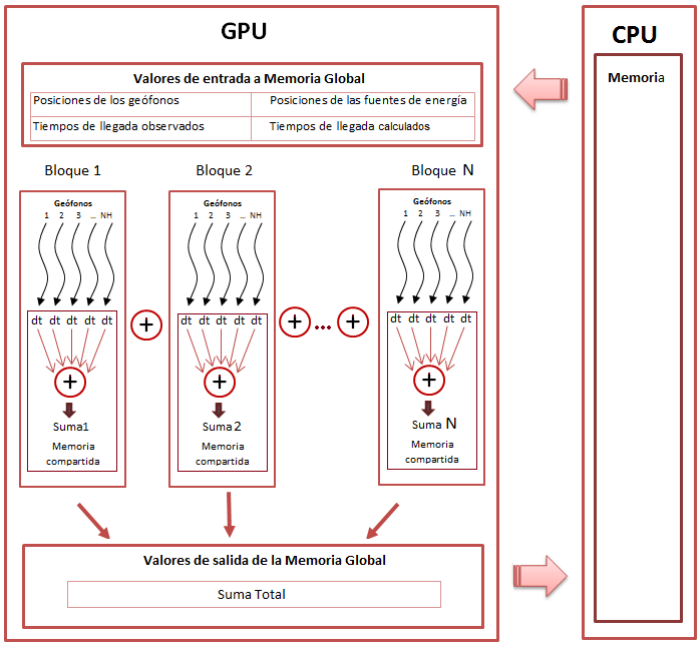

Figura 4.4: Modelo de paralelización para algoritmo de cobertura de rayos

Donde cada hilo realiza el proceso que se muestra a continuación.

```
Leer posicion de geofono (x0, y0, z0)
Leer posicion de SP (xs, ys, zs)
Leer dimension de la celda (h)
Leer dimension de la superficie (NX,NY,NZ)
Leer tiempo de llegada observado (to)
  Generar tiempos con simulacion de frente de onda (tc)
  Encuentra la celda que contiene al SP (i, j, k)
  dt= tc-to
    Mientras posicion del rayo sea diferente de SP
      Calcular gradiente(t)= [dt/dx, dt/dy, dt/dz]
      Encuentra direccion dentro del modelo
      Desplaza la posicion del rayo una celda en la direccion hallada
      Almacena la trayectoria del rayo
    fin_mientras
  return dt
```

La información correspondiente a las ubicaciones de las fuentes de energía y de los receptores, así como la de los tiempos de llegada fue transferida desde la CPU a la GPU, con la finalidad de tener acceso a ésta de manera más rápida. El algoritmo de evolución diferencial implementado en la CPU-GPU adoptó el esquema maestro-esclavo de una sola población, en donde el nodo maestro estuvo representado por la CPU mientras que los esclavos fueron cada uno de los procesadores dentro de la GPU. En la arquitectura CUDA los hilos se encuentran dentro de bloques. El número de bloques para este trabajo fue calculado mediante la ecuación 4.2.1:

$$nBloques = \lceil NGEOS/NMAXHILOS \rceil \tag{4.2.1}$$

Donde *NGEOS* representa a los 4440 geófonos que fueron evaluados en total, ya que si bien es cierto que el número de receptores distribuidos en el volumen fue de 793, no todos pueden ser considerados en los 7 shotpoints y tampoco tuvieron la misma información con

respecto a las fuentes de energía, debido a sus distintas ubicaciones. *NMAXHILOS* identifica al número de hilos por bloque, que puede ser desde 32 hasta 512 ó 1024, dependiendo del modelo de la GPU. En este trabajo se usó NMAXHILOS= 64, para un mejor rendimiento.

Como puede notarse en la ecuación, el número bloques es el entero inmediato superior al resultado de la división debido a que, como pueden variar el número de receptores y también el número de hilos por bloque, es posible que el número de bloques resultante de esta operación no sea suficiente para ejecutar los hilos que sean necesarios, por lo que realizando el cálculo de esta manera no se dará este caso; por el contrario, es posible que sobren, pero cuando suceda a estos se les asigna un valor de 0 para ser considerados en el proceso pero sin afectar el resultado final.

La función kernel está integrada por los siguientes parámetros:

- *NG*: representa el valor total de geófonos, calculado anteriormente
- *ptrpix*: contiene tanto los identificadores de los geófonos, como los tiempos de llegada observados por el geólogo; desde las 7 fuentes de energía hacia cada uno de ellos.
- *ptrstat*: almacena las ubicaciones de las 7 fuentes de energía, en amplitud, longitud y profundidad del volumen de estudio.
- *dev_t*: guarda los tiempos de llegada desde cada una de las fuentes de energía hacia todos los puntos del modelo 3D; estos son calculados previamente en la simulación del frente de onda.
- *temp2*: contiene el resultado de la suma de los tiempos residuales al cuadrado (dt^2) de todos los geófonos.
- *dev_inicioarreglo*: contiene las posiciones del primero y último geófono que se evaluaron en cada fuente, como lo muestra la tabla 4.1. Fue necesario obtener estos datos

debido a que los 4440 receptores fueron almacenados dentro de un vector unidimensional, además de que cada fuente genera un frente de onda diferente; es decir, 7 * 414 414 tiempos de llegada como se muestra en la figura 4.5, y son transferidos a la GPU. Cuando el algoritmo calcula el gradiente dentro de una celda, necesita los 8 tiempos de llegada de los vértices de esa celda; es por ello que el kernel debe saber a qué fuente pertenece, para realizar el cálculo de la posición en dicho tiempo.

Inicio	**Fin**
0	487
488	1104
1105	1838
1839	2435
2436	3123
3124	3791
3792	4439

Tabla 4.1: Posiciones de inicio y fin de geófonos, para cada fuente de energía

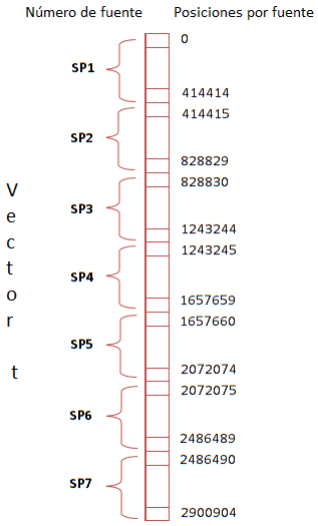

Figura 4.5: Acceso al vector de tiempos

- *dev_saltado*: Valor que corresponde a un shotpoint que puede ser omitido en el experimento; sin embargo, es tomado en cuenta asignando 0 a los tiempos de los hilos que evalúen los geófonos de este número de fuente, para no afectar el proceso paralelo y el resultado final.

- *nshots*: Identifica al número total de fuentes de energía.

En este trabajo de investigación la memoria compartida es usada para realizar la suma de los tiempos residuales de los geófonos en un bloque; por lo que se declara un arreglo de tipo *__shared__* dentro del kernel, cuyo tamaño depende del número de hilos por bloque que deseen usarse (NMAXHILOS); en este caso se tomó el 64 debido a que tuvo un mejor desempeño en comparación con 16, 32, 128, 256 y 512.

Es necesario que haya un identificador del geófono dentro de un bloque, y este es calculado con la ecuación 4.2.2:

$$I = NMAXHILOS * blockIdx.x + threadIdx.x \tag{4.2.2}$$

Donde: blockIdx.x es el identificador del bloque y threadIdx.x es el identificador del hilo dentro de cada bloque.

Pueden haber identificadores de geófonos que sobrepasen el número total de receptores a evaluar, en consecuencia éstos no deben procesar ningún dato. Si este es el caso, al igual que se hace en la fuente de energía omitida, automáticamente se le asigna el valor de 0 al resultado de dt, que almacena el tiempo residual cuadrático.

Para la evaluación de cada hilo y debido a que la información de cada geófono es distinta, de acuerdo al número de fuente; se debe calcular, en base a I y a SP, el valor que se usará para desplazamiento en el vector que contiene los 414 414 tiempos de llegada de todas las fuentes de energía; con la finalidad de acceder al valor correspondiente para el hilo en

ejecución. Este proceso se realiza verificando si I está dentro de los rangos contenidos en *dev_inicioarreglo*, entre el primer valor y el último de cada SP.

Una vez que todos los hilos ejecutan sus procesos para encontrar el tiempo residual, se debe asegurar que todos terminen para continuar con los cálculos finales. La función que sincroniza todos los hilos de cada bloque para garantizar la finalización de sus procesos es:

```
1 __syncthreads();
```

Posteriormente, se realiza la suma de los valores de dt^2 de todos los hilos en cada bloque, y se almacenan en *sum*. El proceso lo lleva a cabo sólo uno de los hilos y en este trabajo es el *threadIdx.x==0*. La ecuación 4.2.3 se usó para llevar a cabo este proceso.

$$sum_b = \sum_{i=1}^{NMAXHILOS-1} dt^2 \tag{4.2.3}$$

Debido a que se hizo uso de datos almacenados en memoria compartida, el acceso a ellos es mucho más rápido en comparación con la memoria global.

Para finalizar la función kernel, se hizo la suma total de los tiempos residuales pero ahora de todos los bloques, haciendo uso de la ecuación 4.2.4:

$$temp2 = \sum_{b=0}^{nBloques} sum_b \tag{4.2.4}$$

Donde *nBloques* es el número total de bloques y sum_b contiene la suma de los hilos en cada bloque, y *temp2* la suma de todos los bloques.

```
1 atomicAdd(temp2, sum);
```

4.2.1. Llamada al kernel

Para que la función kernel pueda ser ejecutada sobre la GPU, es necesario hacer un llamado desde el Host e indicar el número de bloques e hilos, además de hacer la reservación

de memoria; la transferencia de datos, de la CPU a la GPU y viceversa, así como la liberación de memoria.

La instrucción para la ejecución del kernel en este trabajo fue:

```
1  kernel<<<nbloques,NMAXHILOS>>>(NG, dev_pix, dev_stat, dev_t, temp2,
       dev_inicioarreglo, saltado, nshots);
```

Este kernel es ejecutado de manera paralela, devolviendo la suma total de los valores de dt^2.

4.2.2. Reservación de memoria en la GPU

Para implementar el algoritmo en CUDA, es necesario reservar el espacio de memoria para almacenar los datos requeridos por el kernel en la GPU, con la finalidad de tener acceso a los datos requeridos por el algoritmo, entre la CPU y la GPU; ya que este proceso es la principal causa de que ocurra un "cuello de botella" en las aplicaciones en GPUs, ocasionando en consecuencia un bajo rendimiento.

El espacio que se reserva es en la región de la memoria global de la GPU, para que los datos se mantengan durante toda la ejecución del programa. Los datos se almacenarán de forma continua, de tal manera que la información de cada geófono se encontrará contigua en la memoria.

Para llevar a cabo el proceso anterior se hace uso de la función *cudaMalloc*. Su prototipo es el siguiente:

```
1  cudaStatus = cudaMalloc((void**)devPtr, size);
```

Donde *devPtr* es el apuntador a la región de memoria en la GPU; *size* es el tamaño requerido en bytes y *cudaStatus* es una variable de error que es comparada con *cudaSuccess*; en caso de ser igual significa que ha terminado de manera exitosa, de lo contrario indica que ha ocurrido un error de tipo *cudaErrorMemoryAllocation*.

4.2.3. Transferencia de datos CPU-GPU y GPU-CPU

Posterior a la reservación de memoria, se hace la transferencia de datos mediante la función *cudaMemcpy*, cuya estructura es:

```
1 cudaStatus = cudaMemcpy(dev_t, t, size);
```

Donde *dev_t* es la variable donde se almacenan los valores en el destino y *size* indica el tamaño requerido en bytes, que para este caso es:

```
1 nshots*(NXNYNZ+10)*sizeof(float)
```

Los valores que son transferidos son: las posiciones de los geófonos y de las fuentes de energía; así como los tiempos de llegada observados y calculados, desde cada una de las fuentes de energía hacia los geófonos.

4.2.4. Liberación de memoria en la GPU

La memoria reservada tanto en el Host, como en la GPU debe ser liberada mediante las instrucciones:

```
1 cudaFree(ptr); //En CUDA C
2 free(ptr); //En C
```

De esta manera se llevó a cabo la paralelización del algoritmo de cobertura de rayos sísmicos en la Unidad de Procesamiento de Gráficos. En la siguiente sección se describirán las pruebas realizadas, así como los resultados obtenidos en la paralelización.

Capítulo 5

PRUEBAS Y RESULTADOS

En este capítulo se presentan las pruebas aplicadas al algoritmo de cobertura de rayos sísmicos, y el análisis del mismo en las versiones secuencial y paralela.

5.1. ANÁLISIS DE TIEMPOS

El algoritmo fue ejecutado 5 veces con poblaciones de 10, 20, 30, 40 y 50 individuos en cada una, sobre una tarjeta NVIDIA GeForce GT 430, cuyas especificaciones se muestran en la tabla 5.1. Las versiones secuencial y paralela fueron evaluadas de acuerdo al tiempo que tardaron en realizar el cálculo de la suma total de la diferencia al cuadrado de los tiempos de llegada a cada uno de los geófonos, con la restricción de que se genere un rayo sísmico entre la fuente y cada receptor. .

Si bien es cierto que la medición de tiempos en una versión secuencial se hace en la CPU, y con la función *clock()*, el algoritmo usado en este trabajo se ejecutó en un hilo dentro de un bloque sobre la GPU, y se usó la función *cudaRecord* para efectos de igualdad en cuanto a las unidades de medida con la versión paralela; sin embargo, debido a que en la arquitectura CUDA la GPU tiene un tiempo máximo de cómputo por proceso, ésta no soportó la ejecución de la versión secuencial para los 4 440 geófonos puesto que el tiempo de cómputo

Características	**Valor**
Núcleos de procesamiento (2 Multiprocesadores * 48 núcleos CUDA)	96 núcleos CUDA
Versión de la capacidad de cálculo	2.1
Memoria global	2048 MB
Velocidad de reloj de la GPU	1.4 GHz
Velocidad de reloj de memoria	800 MHz
Memoria compartida por bloque	48 KB
Tamaño de warp	32
Número máximo de hilos por multiprocesador	1536
Número máximo de hilos por bloque	1024
Tamaño máximo de cada dimensión por bloque	1024 * 1024 * 64
Tamaño máximo de cada dimensión por cuadrícula	65535 * 65535 * 65535
Límite de tiempo de ejecución en el kernel	Sí

Tabla 5.1: Especificaciones de la tarjeta NVIDIA GEFORCE GT 430

requerido era mayor al permisible, por lo que sólo fue posible registrar el tiempo de 1 070 de ellos antes de que la computadora terminara el proceso paralelo, y mediante el método regresión lineal se buscó un modelo en 2D que aproximara los valores de los receptores registrados para posteriormente calcular la tendencia de estos datos para 4 440 geófonos. La figura 5.1 muestra con línea delgada el tiempo de los geófonos, y con línea gruesa el modelo de regresión lineal según los datos de los 1 070 receptores. Como puede apreciarse, el comportamiento del tiempo tiende a incrementarse a medida que el número de geófonos aumenta. Según el cálculo realizado en este modelo, para 4 440 geófonos el tiempo estimado es de 8 919.6 milisegundos.

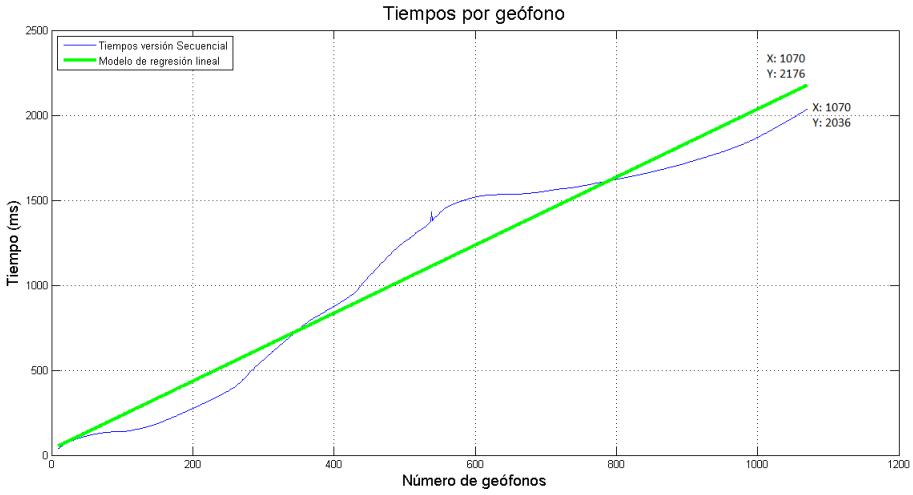

Figura 5.1: Modelo de Regresión Lineal para la medición de tiempos en la versión secuencial

En la figura 5.2 se graficaron los tiempos de ejecución del algoritmo de cobertura de rayos, en horas, en función del tamaño de población por cada versión. La línea con 'o' identifica el comportamiento del algoritmo secuencial y la línea con '+' hace referencia al algoritmo paralelizado. Puede observarse que, el tiempo en la primera versión incrementa de manera constante, en contraste con la segunda versión donde se aprecia que el valor de la pendiente disminuye mientras el tamaño de la población crece, por lo que se deduce que si el tamaño de la población sigue incrementándose las líneas de la gráfica se separarán mucho más, remarcando la diferencia entre ambas versiones, donde la versión paralela superará en gran medida a la versión secuencial.

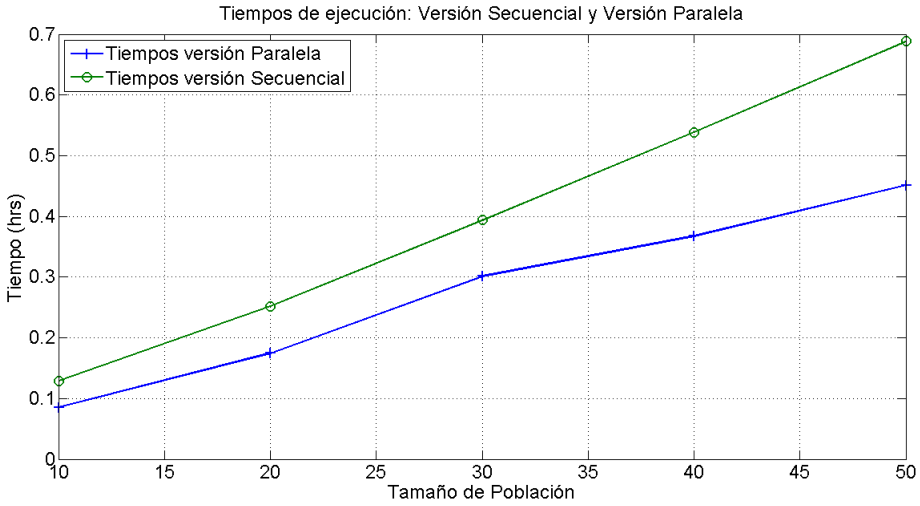

Figura 5.2: Comparación de tiempos de la Versión Secuencial y Paralela

Al desplazarse un rayo sísmico desde un shotpoint hasta un geófono, éste atraviesa varias celdas del modelo tridimensional, el tiempo de cómputo promedio de tránsito del rayo por una celda se calculó utilizando la expresión 5.1.1:

$$Ttpc = Tpng/Cpng \tag{5.1.1}$$

Donde *Ttpc* es el tiempo de cómputo para el cálculo de transición del rayo en una celda; *Tpng* es el tiempo de cómputo por número de geófonos y *Cpng* es el número de geófonos.

La gráfica superior de la figura 5.3 muestra el crecimiento del tiempo de cómputo del algoritmo secuencial cada vez que se aumenta el número de geófonos desde 10 hasta 1070 (máximo número permitido por la tarjeta gráfica); en la gráfica central se observa el número de celdas que se van acumulando al aumentar el número de geófonos, y como consecuencia, en la gráfica inferior se muestra la razón entre el tiempo del algoritmo secuencial y el número de celdas por geófonos, y utilizando una regresión lineal se puede observar que el tiempo promedio por celda se aproxima a 0.0275 ms.

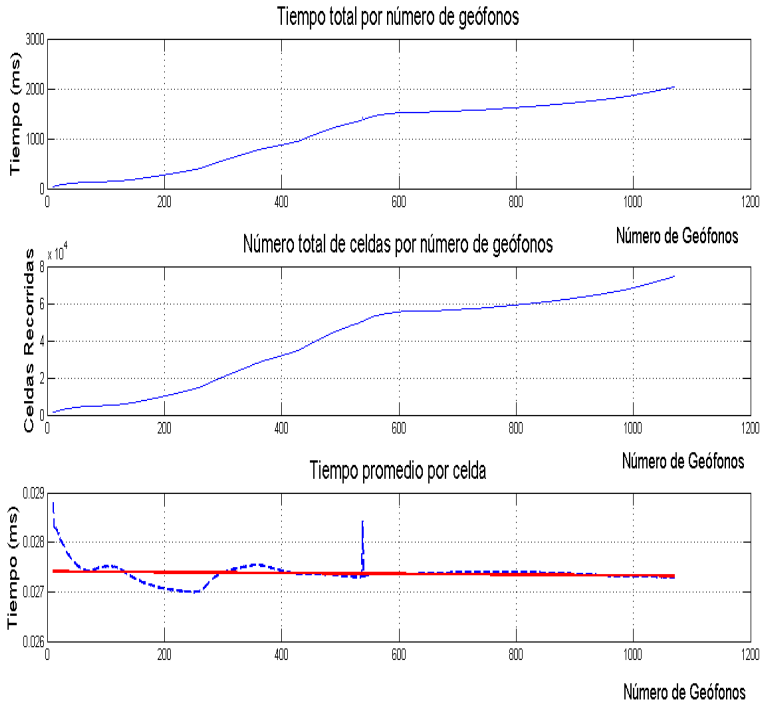

Figura 5.3: Tiempo del cálculo de transición del rayo sísmico

5.2. ANÁLISIS DE APTITUD

En las figuras 5.4,5.6, 5.8, 5.10 y 5.12 se visualizan las gráficas del desempeño del algoritmo de cobertura de rayos en base a la aptitud registrada durante las 5 ejecuciones, haciendo una comparación entre las versiones paralela y secuencial en cada una de ellas. En estas gráficas las líneas con los símbolos: '+', 'o' y '*' son usadas para hacer referencia a máximos, promedios y mínimos, respectivamente, del algoritmo secuencial; y aquellas con símbolo: '×', '□' y '◇' hacen referencia a máximos, promedios y mínimos, respectivamente, del algoritmo en CUDA. En las figuras 5.5, 5.7, 5.9, 5.11 y 5.13 se muestran los modelos de velocidades 1D iniciales, generados por los mejores individuos de la ED, por cada tamaño de población evaluado por el algoritmo en la versión paralela. La línea punteada en cada una

de las gráficas representa el individuo con que empieza a evolucionar el algoritmo, y la línea continua muestra el mejor individuo después de haber realizado el proceso evolutivo por 10 generaciones.

En la figura 5.4 se aprecia la gráfica de aptitudes cuando se evalúa el algoritmo con un tamaño de población de 10 individuos, y se nota que, debido a que la población inicial es generada de forma aleatoria, el valor de la aptitud en las primeras generaciones es alto, pero a medida que los individuos van evolucionando durante las generaciones, el resultado va mejorando en ambas versiones. Si bien es cierto que en las primeras 4 generaciones se nota un valor máximo, en la versión de CUDA, que está muy por encima del resto de los valores, es importante notar que a partir de la quinta generación hay una mejora significativa que incluso llega a superar a la versión secuencial durante el resto del proceso evolutivo. La figura 5.5 muestra el modelo de velocidades generado por el individuo inicial y el individuo final de la versión paralela.

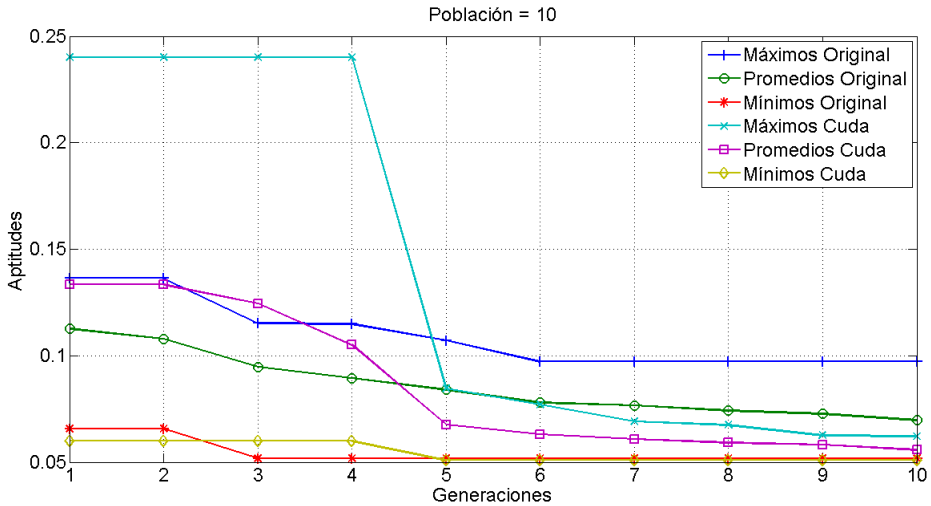

Figura 5.4: Aptitud con tamaño de población 10

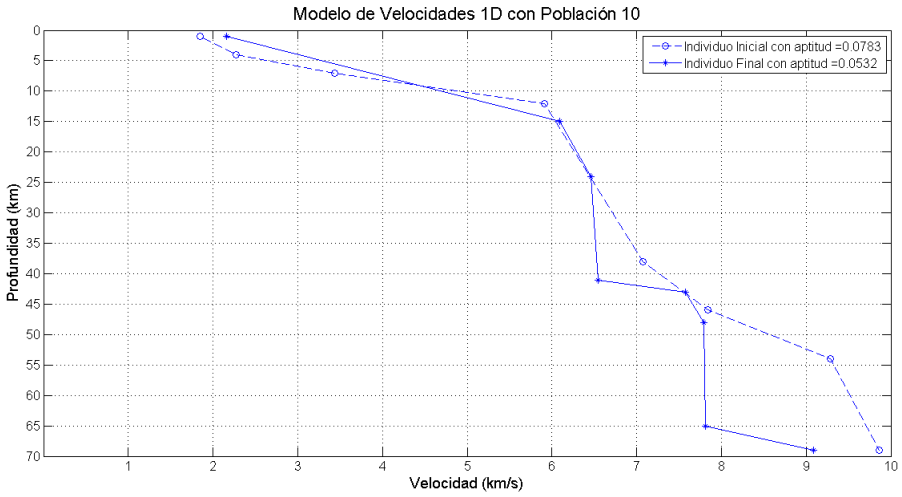

Figura 5.5: Modelo de velocidades inicial con tamaño de población 10

En la figura 5.6 se muestra la gráfica de las aptitudes cuando el algoritmo de cobertura de rayos es evaluado con un tamaño de población de 20 individuos. Puede notarse que al haber mayor diversidad en comparación con la figura anterior, el valor de la aptitud mejora de manera más rápida, pero es importante notar que la aptitud máxima de la versión de CUDA se mantiene siempre arriba de la versión secuencial durante las 10 generaciones. La figura 5.7 muestra el modelo de velocidades obtenido por el individuo inicial y el individuo final de la versión en CUDA.

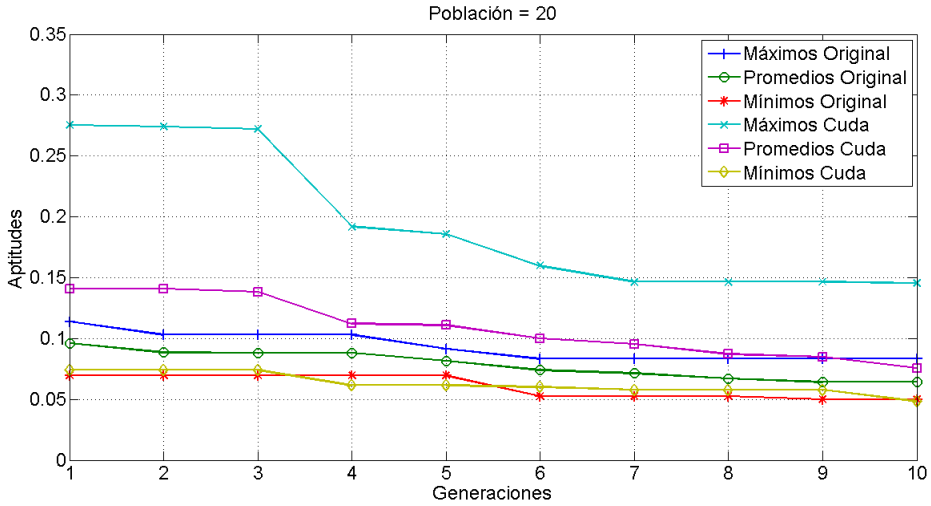

Figura 5.6: Aptitud con tamaño de población 20

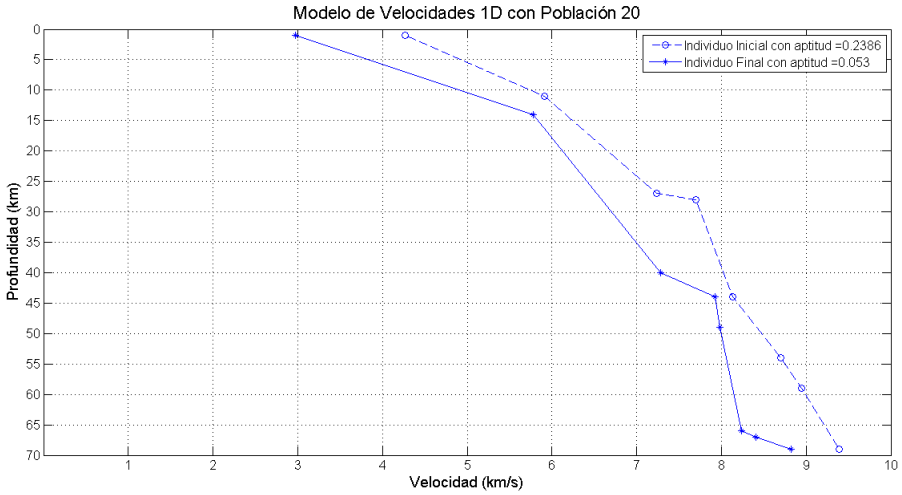

Figura 5.7: Modelo de velocidades inicial con tamaño de población 20

La figura 5.8 muestra los mejores valores para ambas versiones del algoritmo, las cuales se dan cuando el tamaño de la población es de 30 individuos. El valor de la aptitud disminuye considerablemente con respecto a las poblaciones de 10 y 20 individuos, debido a que existe mayor diversidad en la población. La figura 5.9 muestra el modelo de velocidades generado por el individuo inicial y el individuo final de la versión paralela.

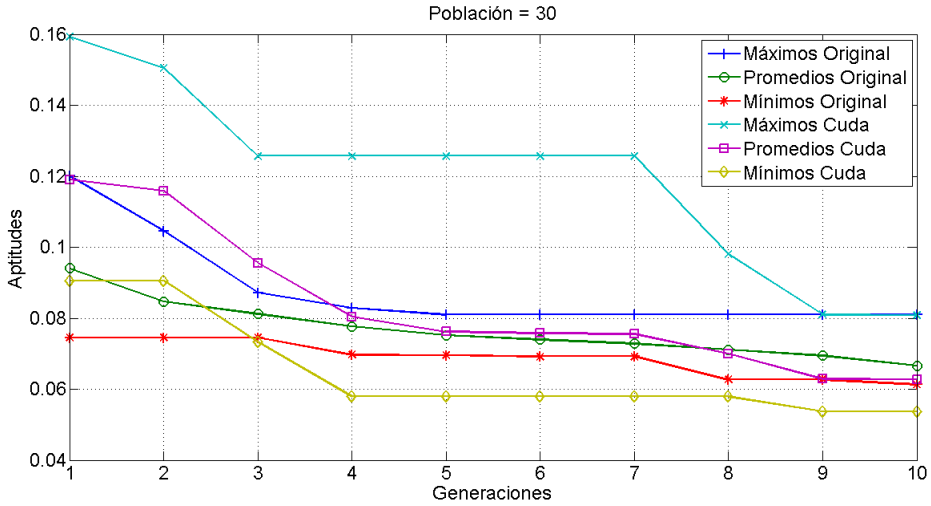

Figura 5.8: Aptitud con tamaño de población 30

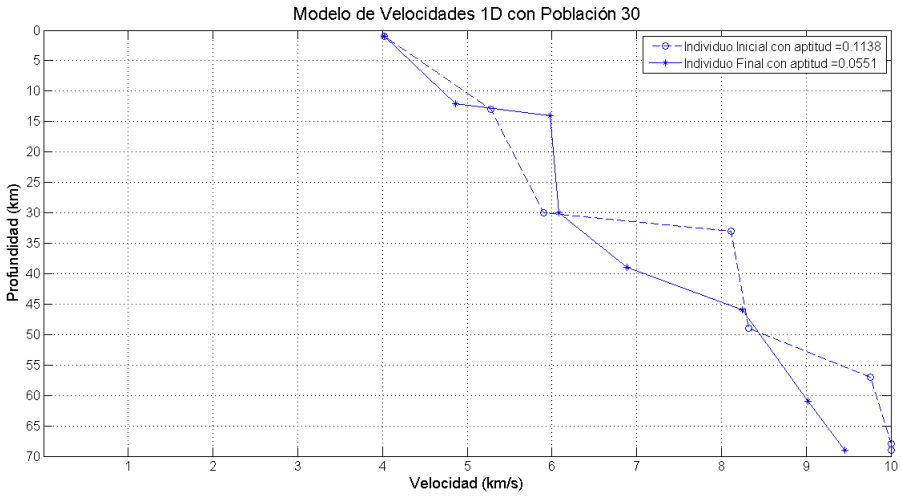

Figura 5.9: Modelo de velocidades inicial con tamaño de población 30

Cuando se incrementa el tamaño de la población a 40 individuos el valor de la aptitud, como se muestra en la figura 5.10 es más alto al iniciar el algoritmo, en comparación a la población de 30 individuos; sin embargo, disminuye de forma rápida desde la segunda generación obteniendo en la sexta el mejor valor de aptitud, y a partir de ésta se mantienen los

valores hasta terminar el proceso. La figura 5.11 muestra el modelo de velocidades obtenido por el individuo inicial y el individuo final de la versión en CUDA.

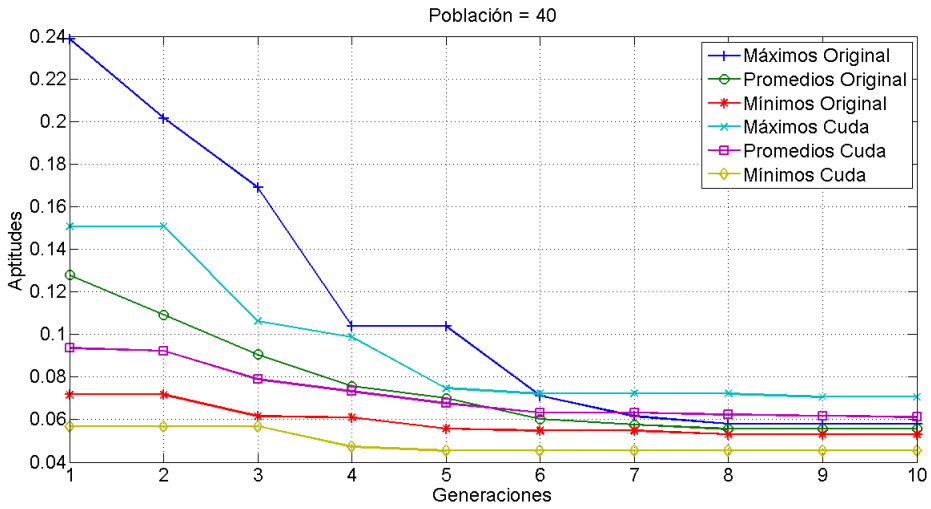

Figura 5.10: Aptitud con tamaño de población 40

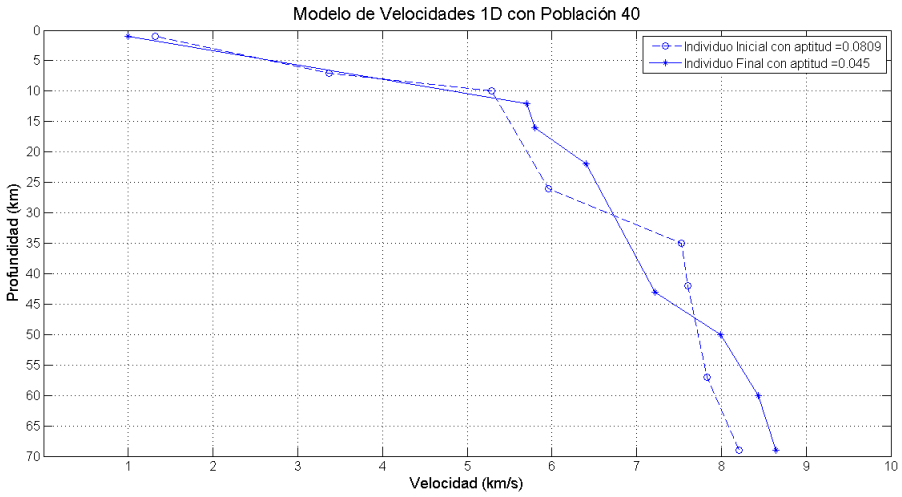

Figura 5.11: Modelo de velocidades inicial con tamaño de población 40

La figura 5.12 muestra los valores de aptitud para el tamaño de población de 50 individuos, el cual aunque mejora los resultados en pocas generaciones, no dio mejores aptitudes

que la población anterior.

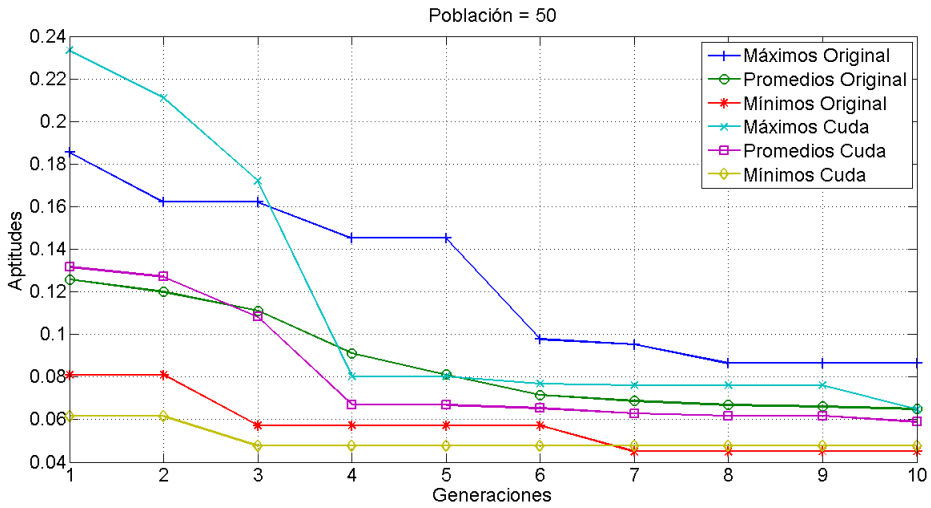

Figura 5.12: Aptitud con tamaño de población 50

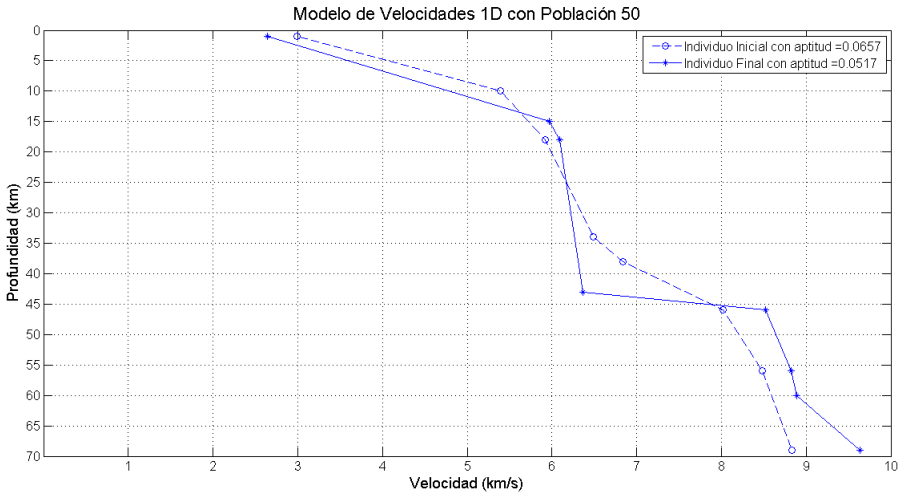

Figura 5.13: Modelo de velocidades inicial con tamaño de población 50

En general puede notarse que, de acuerdo al tamaño de la población, la aptitud mejora en ambas versiones, a través de 10 generaciones y tiende a ser similar al final de cada evaluación; esto debido a que los procesos llevados a cabo son iguales en ambas versiones. En

la figura 5.14 se aprecia la gráfica con los mínimos valores de aptitud por tamaño de población. Como puede observarse, las aptitudes que corresponden al tamaño población 40 son, de cierta manera, las mejores en comparación con el resto de las poblaciones.

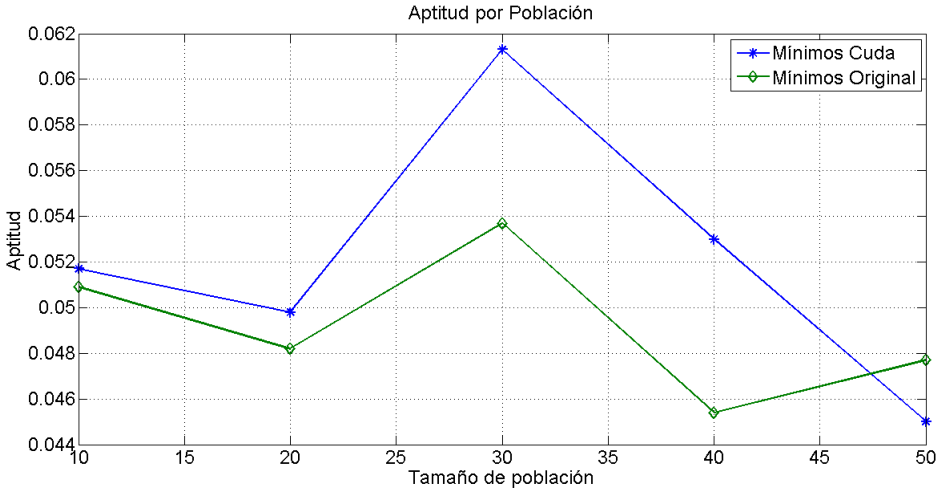

Figura 5.14: Aptitud mínima por tamaño de población

Estas fueron las pruebas realizadas para comparar el desempeño del algoritmo de cobertura de rayos, tanto en la versión secuencial como en la versión paralela en CUDA, ejecutadas desde Visual Studio 2010. En el siguiente capítulo se mostrarán las conclusiones y las propuestas para mejorar este proyecto de investigación, de acuerdo al trabajo desarrollado.

Capítulo 6

CONCLUSIONES Y TRABAJOS FUTUROS

Después de haber diseñado e implementado el modelo paralelo para el algoritmo de cobertura de rayos, que obtiene el RMS de los tiempos de llegada desde las fuentes de energía hacia cada uno de los geófonos y haberlo integrado en una Evolución Diferencial, en esta sección se comentan las conclusiones y los trabajos a futuro.

6.1. CONCLUSIONES

Actualmente, en diversas áreas se están usando las tarjetas gráficas para acelerar los procesos de algoritmos que requieren grandes recursos computacionales, como son las técnicas de búsqueda pertenecientes a la computación evolutiva, las cuales al usar individuos que a través de un proceso van intercambiando información entre si para modificarse y mejorar durante cierto número de generaciones, demandan gran cantidad de recursos computacionales como memoria y tiempo que una computadora convencional no ofrece. La arquitectura de las Unidades de Procesamiento de Gráficos, en contraste con las Unidades Centrales de Procesos, están diseñadas para ejecutar programas estructurados por el programador como

cálculos que pueden realizarse de manera simultánea al seguir el modelo: misma instrucción, múltiples datos.

En este trabajo de investigación se presentó la implementación de una Evolución Diferencial, donde la función objetivo es un algoritmo de cobertura de rayos sísmicos, cuya ejecución se llevó a cabo en una GPU. Si bien es cierto que la paralelización de los algoritmos evolutivos puede aplicarse en todo el proceso (generación de población inicial, operadores de selección, mutación y cruza, y la función de evaluación) la mayoría de los trabajos, incluyendo este, se enfocan únicamente a la paralelización de la función de evaluación debido a que es la que consume mayor tiempo de cómputo.

En el modelo original se debe tomar una sola fuente de energía, generar un frente de onda y, de acuerdo a los tiempos de llegada generados, hacer los cálculos para los cientos de geófonos correspondientes, considerando que tales procesos se llevan uno a la vez; es decir, se realizan las operaciones para un geófono y una vez que éste termina, se inicia el mismo proceso con el siguiente receptor, así sucesivamente hasta evaluar a todos los geófonos del shotpoint; además de que estas tareas se deben realizar por cada una de las fuentes de energía y en consecuencia, el tiempo de ejecución incrementa de manera considerable.

Una alternativa para mejorar el proceso anterior fue, de forma similar a la anterior, tomar una fuente de energía y generar un frente de onda pero, en contraste con la versión original, hacer los cálculos para cada geófono de manera simultánea asignando un receptor en cada hilo de CUDA y una vez terminada la ejecución para el shotpoint en evaluación, continuar uno a uno con el resto de ellos; sin embargo, el modelo paralelo diseñado en este trabajo, genera 7 frentes de onda (uno por cada fuente de energía), de tal manera que los 4 440 geófonos acceden simultáneamente a los tiempos de llegada. Esta solución mostró un mejor desempeño en comparación con la versión secuencial original (y aunque no se realizó la versión alternativa mencionada anteriormente, ésta tampoco hubiese dado mejores resultados), al distribuir la evaluación de los geófonos de manera simultánea, en lugar en hacerlo uno detrás de otro. La diferencia entre la versión secuencial y la versión paralela se notó en la disminución de tiempos de ejecución de la segunda mientras mayor fue el tamaño de la población, en con-

traste con la primera que incrementaba los tiempos de ejecución a medida que el tamaño de la población crecía.

En cuanto a la aptitud, en ambas versiones tiende a ser similar, y es de esperarse debido a que en los dos modelos se realizan exactamente las mismas operaciones. La variación que llega a apreciarse se debe a que la población inicial se genera de manera aleatoria, pero a través de las generaciones la diferencia entre dicha variación llega a ser mínima. Cabe mencionar que el análisis de la aptitud sólo se realizó para verificar que el funcionamiento en el algoritmo paralelo no se viera afectado por los cambios que se realizaron durante el diseño del modelo paralelo, pero la parte de mayor relevancia en este trabajo es el diseño del modelo paralelo, donde se asignó la evaluación de un geófono en cada hilo de CUDA y se formaron grupos de la mínima cantidad posible de hilos por cada bloque ya que la ejecución de estos dentro de un bloque se da por cada 32; y aunque este valor representa la agrupación de hilos más pequeña, en este caso se usaron 64 ya que este número mostró un mejor desempeño para la reducción del tiempo de ejecución del algoritmo de cobertura de rayos, al ser procesado sobre la GPU.

6.2. TRABAJOS FUTUROS

El desarrollo de esta tesis muestra la solución a uno de los módulos que conforman un algoritmo de tomografía sísmica, donde se requiere procesar una gran cantidad de información recolectada por dispositivos dentro de un volumen de la corteza terrestre, y aún hay mucho trabajo que realizar.

En cuanto al área de computación evolutiva existen varias propuestas, desde únicamente modificar los parámetros de la ED para realizar diversas pruebas con valores distintos y analizar la influencia que tiene cada uno de ellos en esta técnica de búsqueda, hasta usar algún otro algoritmo evolutivo como es el caso de los algoritmos genéticos o las estrategias evolutivas, técnicas que ya se han usado para la solución de este problema en su forma secuencial.

Por la parte de tomografía sísmica, se planea diseñar otro modelo paralelo donde se evalúen los geófonos mediante capas (una a la vez), haciendo uso de la memoria compartida para almacenar los valores de los tiempos de llegada desde la fuente de energía sísmica hacia cada uno de los receptores correspondientes a esa capa; de esta manera el acceso a los datos será mucho más rápido en comparación con el modelo propuesto en esta tesis, donde se utiliza la memoria global para almacenar los 414 414 valores de los tiempos debido a que la memoria compartida está limitada a 48 KB por bloque, y los hilos de distintos bloques no pueden colaborar entre si, y la memoria compartida en este trabajo sólo se usa para almacenar los valores de los tiempos residuales alojados en cada bloque de la GPU, utilizados a su vez para obtener la suma cuadrática de tales tiempos.

Aunado a lo anterior, como ya se ha mencionado, el algoritmo de cobertura de rayos sísmicos es sólo un pequeño módulo que, en conjunto con otros, conforman uno más complejo, por lo que las actividades a realizar en un futuro son el análisis del algoritmo de simulación de frente de onda para su posible paralelización, ya que es uno de los módulos que consume gran cantidad de recursos computacionales y al lograr distribuirlo en la GPU se reducirá aún más el tiempo de ejecución.

Apéndice A

ANEXOS

A.1. ESTANCIA

Mtro. Felipe Pascual Rosario Aguirre
Director
Instituto Tecnológico de Apizaco

Tonantzintla, Puebla, 28 de enero de 2015

ASUNTO: Carta de liberación

Estimado Mtro. Rosario Aguirre:

La presente es para informar que la estudiante de maestría **Eustolia Carreón Esteban**, del Instituto Tecnológico de Apizaco, realizó una estancia de trabajo en la Coordinación de Ciencias Computacionales del INAOE, con el tema ALGORITMO HIBRIDO PARALELO PARA APLICACIONES EN TOMOGRAFIA SISMICA. La estancia inició el 10 de enero de 2014 y originalmente terminaría el 28 de abril, pero se le agregó una extensión de tiempo del 29 de abril al 28 de julio, que se acordó para explorar mas alternativas de paralelización del algoritmo en cuestión. La estancia y trabajo de la estudiante fue satisfactoria.

Si requiere información adicional, por favor no dude en contactarme.

Atentamente

Dr. Miguel O. Arias Estrada
Investigador Titular B
Laboratorio de Cómputo Reconfigurable y de Alto Desempeño
Departamento de Ciencias Computacionales - INAOE
ariasmo@inaoep.mx
Tel: +52 222 266-3100 ext 8316

A.2. CIIAII 2014

ALGORITMO EVOLUTIVO PARALELO PARA TOMOGRAFÍA SÍSMICA

Eustolia Carreón Esteban

Instituto Tecnológico de Apizaco, Av. Instituto Tecnológico s/n, 01 (241) 417 2010 ext 138

euce_79@hotmail.com

José Federico Ramírez Cruz

Instituto Tecnológico de Apizaco, Av. Instituto Tecnológico s/n, 01 (241) 417 2010 ext 138

federico_ramirez@yahoo.com.mx

Sergio Palafox Ugarte

Instituto Tecnológico de Apizaco, Av. Instituto Tecnológico s/n, 01 (241) 417 2010 ext 138

serggio.23@gmail.com

Resumen

En este trabajo de investigación se realiza la paralelización de un algoritmo de tomografía sísmica, cuyo objetivo es trazar los rayos generados por 7 fuentes de energía sísmica artificiales (SP) hacia cientos de dispositivos receptores (geófonos) y obtener el tiempo mínimo residual, utilizando la arquitectura CUDA. El cálculo del tiempo residual es realizado en la GPU y se obtiene con la diferencia del tiempo observado (registrado por un geólogo), y el calculado previamente en el algoritmo. El proceso de cálculo del tiempo residual y el trazo de la trayectoria del rayo sísmico que viaja desde un SP hasta un geófono es asignado a una unidad mínima de ejecución en la GPU (hilo), que para la aplicación realizada en este trabajo fueron 4,439 ejecutándose en paralelo. Los datos usados en este trabajo se obtuvieron de un experimento realizado en el campo volcánico El Potrillo, ubicado en el sur de Nuevo México, a cargo del Departamento de Ciencias Computacionales y del Departamento de Ciencias Geológicas de la Universidad de Texas, en El Paso. Las pruebas fueron realizadas en una GPU NVIDIA GeForce GTX 480 con 448 núcleos CUDA y con una CPU Intel Core I7 de 3.6 GHz.

Palabra(s) Clave(s): cómputo paralelo, CUDA C, tomografía sísmica, algoritmos evolutivos.

1. Introducción

La tomografía sísmica es una técnica de imagen que asimila observaciones del movimiento de la tierra, recolectadas con sismómetros, para mejorar los modelos estructurales, y ha sido uno de los medios más efectivos para obtener imágenes del interior del planeta en las últimas décadas [1]. Los modelos de velocidades de ondas, propagadas a través de la corteza terrestre por fuentes de energía naturales, como terremotos; o artificiales, como explosiones, proporcionan información que puede ser utilizada para una amplia variedad de aplicaciones tales como investigaciones arqueológicas, control de calidad y evaluación de proyectos de ingeniería, además del descubrimiento de depósitos de agua, aceite o material de desecho [2], para ello se usan diversas técnicas de búsqueda, en este caso en particular se utilizan los Algoritmos Evolutivos (AEs), ya que tienen la capacidad de buscar en espacios grandes y no son dependientes de una solución inicial aproximada a la óptima. La Computación Evolutiva (CE) es reconocida como un método eficaz para resolver problemas de optimización difíciles; sin embargo, conlleva grandes costos computacionales, ya que generalmente se evalúa a todos los candidatos de soluciones en una población para todas las generaciones [3] y por lo tanto ralentiza el proceso por horas o días, dependiendo de la cantidad de los datos que se usen. Debido a que muchas aplicaciones actualmente requieren mayor poder de cómputo del que una computadora secuencial es capaz de ofrecer, el cómputo paralelo ofrece la distribución del trabajo entre diferentes unidades de procesamiento, resultando mayor poder de cómputo y rendimiento del que se puede obtener mediante un sistema tradicional de un procesador [4]. Con el desarrollo de herramientas de programación de Unidades de Procesamiento Gráfico (GPU’s, por sus siglas en inglés), varios algoritmos han sido adaptados a este hardware satisfactoriamente y la plataforma de computación híbrida GPU - CPU, comparada con las implementaciones en CPU's ha alcanzado aceleraciones importantes [5]

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

En los últimos años se han implementado los AEs en GPU's y se ha visto que son capaces de mejorar decenas de veces el desempeño mediante el uso de este hardware, sobre todo cuando se utilizan tamaños grandes de población [6].

En este trabajo se hace uso de un AE y de un Algoritmo de Tomografía Sísmica, que tienen como objetivo generar un modelo de velocidades de acuerdo a los tiempos de llegada de las ondas, desde una fuente de energía (SP por sus siglas en inglés Shot Point) hacia cada uno de los receptores (geófonos). Los datos utilizados fueron producto de un estudio en la región del campo volcánico El Potrillo, ubicado en el sureste de Nuevo México, cuyo volumen a evaluar fue de 414 414 m^3. Se realiza además la paralelización en una GPU y bajo la arquitectura de CUDA, en el lenguaje C, con la finalidad de reducir el tiempo de ejecución.

En la sección 2 se describirán los algoritmos ED, de tomografía sísmica y el modelo de paralelización sobre la GPU. En la sección 3 se presentan resultados experimentales obtenidos y se hace una comparación entre el desempeño de las versiones secuencial y paralela. En la sección 4 se mencionan las conclusiones y los trabajos futuros

2. Desarrollo

2.1 Algoritmo de Evolución Diferencial (ED)

El algoritmo de Evolución Diferencial es uno de los más potentes algoritmos de optimización actualmente usados [7, 8]. Para la generación del modelo de velocidades se implementa una ED, el cual mediante los operadores de selección, cruza, mutación y evaluación mejora con en el transcurso de decenas de generaciones.

El pseudocódigo del algoritmo evolutivo empleado en este trabajo se muestra en la Fig. 1.

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

```
1  G ← 0;
2  Inicializar(P_g ← {x⃗_1, ··· , x⃗_np})
3  while Criterio de Terminación NO satisfecho do
4      for i ← {1, ··· , np} do
5          r_1, r_2, r_3 ∈ {1, ··· , np} aleatoriamente seleccionados,
6                                  donde r_1 ≠ r_2 ≠ r_3 ≠ i ;
7          j_rand ∈ {1, ··· , n} aleatoriamente seleccionado ;
8          for j ← {1, ··· , n} do
9              if U_j[0, 1] < CR or j = j_rand then
10                 u_{i,j} ← x_{r3,j,G} + F(x_{r1,j,G} − x_{r2,j,G}) ;
11             else
12                 u_{i,j} ← x_{i,j,G} ;
13         if f(u⃗_i) ≤ f(x⃗_{i,G}) then
14             x⃗_{i,G+1} ← u⃗_i ;
15     G ← G + 1;
```

Fig. 1. Pseudocódigo de una Evolución Diferencial

2.1.2 Representación de los individuos

En este algoritmo cada individuo de la población representa un conjunto de profundidades y velocidades en 8 diferentes capas de la tierra.

En la Fig. 2 se representa un individuo, el cual al ser visto como un vector, está dividido en dos partes. En la primera, los valores 1 y 69 ocupan las posiciones 0 y 7, respectivamente, que son el mínimo y el máximo de la profundidad en kms, y las posiciones intermedias son ocupadas por valores de tipo entero que no pueden repetirse y que deben ser ordenados de menor a mayor. En la segunda parte del individuo se asignan las velocidades en un rango de 3 a 8 km/s; que cumplen con las mismas restricciones de la primera pero son valores reales.

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

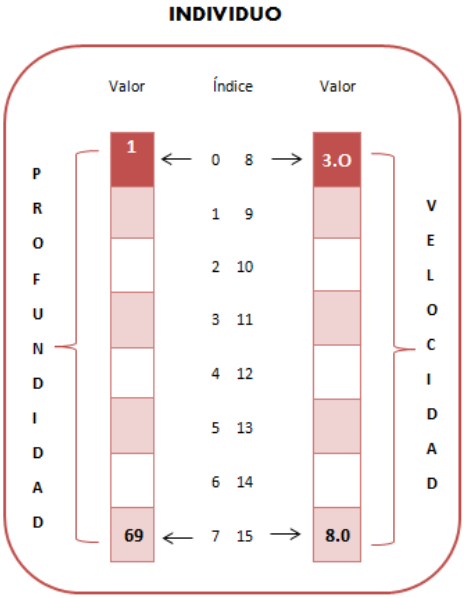

Fig. 2. Representación de un individuo

2.1.3 Recombinación o cruza

La recombinación o cruza es una de las operaciones evolutivas implicadas en la generación de nuevos individuos. Esta operación es llevada a cabo con la participación de dos o más padres, quienes heredan rasgos o características a sus descendientes mediante una mezcla de información que se da de manera aleatoria. También existe la posibilidad de que los padres sobrevivan para formar parte de la siguiente generación, esto se da en caso de que la aptitud sea mejor que la de los hijos.

Aunque la cruza no es el operador principal en una ED, a diferencia de un Algoritmo Genético, éste se implementa debido a que es parte de los principios originales de los AEs, y depende de un parámetro muy importante: la tasa de recombinación o cruza, que de acuerdo a la literatura consultada está en un rango de [0 a 1]. Para este caso se estableció en 0.8, y el tipo de cruza es discreta binomial como se representa en la Fig. 3.

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

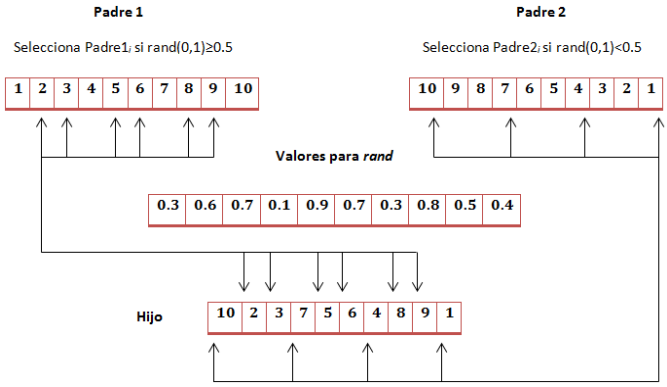

Fig. 3. Cruza Discreta Binomial

2.1.4 Mutación

El operador de mutación proporciona diversidad a la población a través de la introducción de nuevas soluciones, lo cual evita la convergencia prematura en el algoritmo; es decir, la posibilidad de una rápida obtención no necesariamente del óptimo global. Para superar este problema es necesario conservar la diversidad en las generaciones, lo cual permite la exploración de otras áreas en el espacio de búsqueda.

Para llevar a cabo la operación de mutación, por cada individuo de la población x_i, se eligen aleatoriamente tres individuos de población, $x_{r1,}\, x_{r2,}\, y\, x_{r3}$, con la condición de que sean diferentes entre sí y diferentes de x_i, el operador de mutación se calcula como la diferencia de dos de ellos multiplicada por un factor $F \in [0..1]$, y el resultado es sumado al tercer individuo, esto es, $x_i' = x_{r3} + F(x_{r2} - x_{r1})$, como se muestra en la Fig. 4. El valor de F puede ser establecido por el diseñador o puede ser un número aleatorio.

2.1.5 La Función de evaluación

Cada vez que un individuo x_i' es generado, debe ser evaluado para determinar si forma parte de la siguiente generación, en dado caso que el valor de la evaluación $f(x_i')$ sea mayor que la de su padre $f(x_i)$, este individuo pasa a la siguiente generación, de lo contrario se queda el padre x_i.

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

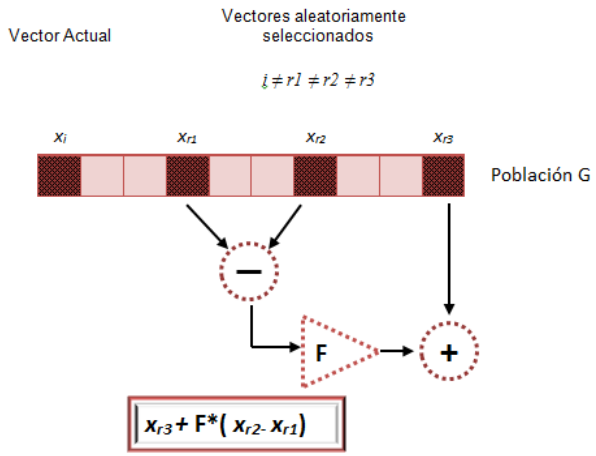

Fig. 4. Operación de Mutación en la ED

La función de evaluación de este algoritmo consiste en medir los tiempos residuales (diferencia entre el tiempo de llegada del rayo sísmico a cada geófono y el tiempo observado). Para calcular el tiempo de llegada del rayo sísmico se realiza una simulación de la propagación del frente de onda sísmica dentro del volumen a evaluar utilizando el modelo de Vidale [9]. Para calcular la posición del rayo sísmico dentro de la celda donde se encuentra el geófono se utiliza una interpolación tri-lineal utilizando los tiempos de llegada del frente de onda a los ocho vértices de la celda. Estos dos procesos son parte de un Algoritmo de Tomografía Sísmica (ATS) propuesto por Vidale and Holes [9, 10]. En este trabajo, el proceso del cálculo de del tiempo residual es paralelizado sobre una GPU.

El ATS inicia leyendo los datos de las ubicaciones tanto de las 7 fuentes de energía, de los cientos de receptores distribuidos, y de los tiempos de llegada en base al análisis de estudios sísmicos anteriores realizados por un geólogo. Se crea una red uniforme en las tres dimensiones del volumen de estudio, como se muestra en la Fig. 5. En este caso el tamaño de cada una de las celdas es de 1 km^3. De acuerdo a la velocidad y a la profundidad (representadas en cada uno de los individuos), se genera mediante el método de interpolación un modelo inicial de tiempos de llegada de los frentes de onda para rastrear las trayectorias de los rayos a lo largo de la

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

pendiente más pronunciada (la línea perpendicular a cada frente de onda) de un receptor a la fuente.

Después de que se han generado los tiempos de viaje se calcula el tiempo residual mediante la diferencia del observado y el calculado. Una vez que se han realizado estas operaciones para cada rayo que atraviesa el modelo desde el geófono hasta la fuente, se repite el proceso para el resto de las fuentes y de los receptores y se obtiene un promedio.

El modelo de velocidades tiene que ir mejorando haciendo cada vez mínimo el tiempo residual a través de las generaciones.

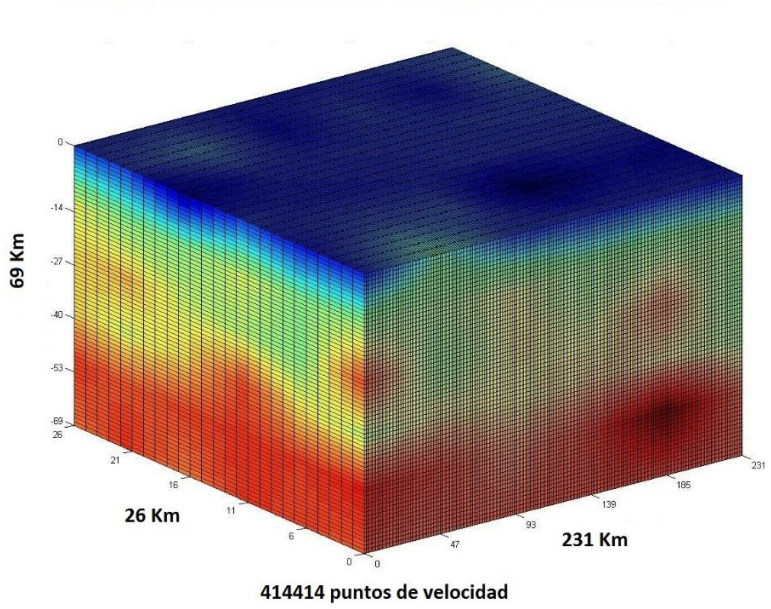

Fig. 5. Modelo de velocidades de la corteza terrestre

2.2 Modelo de paralelización

El modelo paralelo que se propone en este trabajo consiste en hacer el trazo de cada uno de los rayos de todas las fuentes, en forma independiente, debido a que no necesitan información de cálculos previos al que se está evaluando.

La información correspondiente a las ubicaciones de las fuentes de energía y a los receptores, así como la de los tiempos de llegada es transferida desde la CPU a la

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

GPU, con la finalidad de tener acceso de manera más rápida, ya que el pase de datos de un dispositivo a otro es el inconveniente que tiene el uso de este hardware.

El algoritmo de evolución diferencial implementado en la GPU adopta el esquema maestro-esclavo de una sola población, en donde el nodo maestro se encuentra representado por la CPU mientras que los esclavos serán cada uno de los procesadores dentro de la GPU.

En la arquitectura CUDA los hilos se encuentran dentro de bloques. El número de hilos por bloque (NH) puede ser desde 16 hasta 512 o 1024 dependiendo del modelo. En este caso se hace la prueba con NH= 64, para un mejor rendimiento. El número de bloques es calculado mediante la siguiente ecuación: $n = Ngeofonos/NH$.

En la Fig. 6 se muestra de manera general el modelo paralelo implementado.

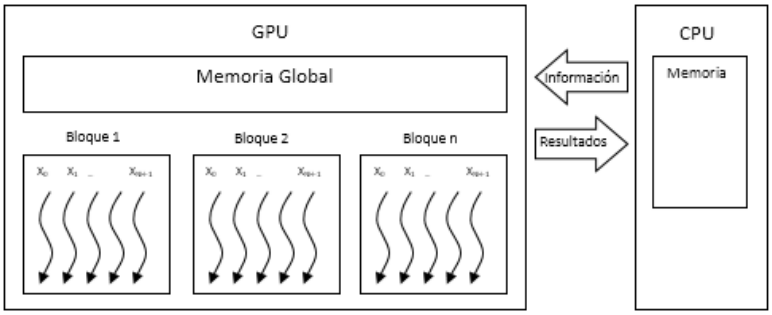

Fig. 6. Modelo paralelo

2.1.2 Reservación de memoria y transferencia de datos

Para la transferencia de datos desde la CPU a la GPU es necesario reservar un espacio en la memoria global mediante la instrucción cudaMalloc, que incluye como parámetros el número de valores a recibir multiplicado por el tipo de dato y un puntero para ir almacenándolos. Posteriormente se realiza la copia de la información con la función cudaMemcpy la cual es caracterizada por un parámetro que indica la fuente y el destino para realizar la transferencia, y que pueden ser:

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

cudaMemcpyHostToDevice, para copiar desde la CPU a la GPU; o cudaMemcpyDeviceToHost, para copiar desde la GPU a la CPU.

3. Resultados

En la versión secuencial se leen datos para hacer el trazo de rayos de un sólo un geófono. Se sigue la trayectoria celda por celda, desde el receptor hasta la fuente de energía, pero no se puede continuar con el siguiente sino hasta que termina el actual; en contraste con el modelo paralelo propuesto, donde se hace el cálculo simultáneamente para todos los geófonos hacia todas las fuentes de energía, ya que en cada uno de ellos se usan valores diferentes por lo que cada hilo toma los valores necesarios.

La versión secuencial realiza el cálculo de los tiempos residuales en un tiempo aproximado de 100 ms; sin embargo, la versión paralela registra un tiempo menor a 1 ms sin considerar la trasmisión de datos de la memoria del CPU al GPU.

5. Conclusiones

El concepto básico en un esquema de procesamiento paralelo es la división de un proceso en varios sub-procesos, los cuales son resueltos simultáneamente empleando múltiples procesadores.

En este trabajo se realizó la paralelización de la función objetivo de un algoritmo de evolutivo (ED), que consiste en el cálculo del promedio cuadrático de los tiempos residuales de un algoritmo de tomografía sísmica.

Se calculó el tiempo residual de 4440 los geófonos de forma paralela utilizando una GPU, cada hilo de la GPU calcula el tiempo y la trayectoria del rayo sísmico que va desde 7 fuentes de energía (SP) hasta cada uno de los geófonos.

En este trabajo se utilizó la memoria global del dispositivo para pasar la información de los cuadrados de los tiempos residuales de la GPU al CPU, en un trabajo futuro se utilizará la memoria compartida del dispositivo para realizar en la tarjeta GPU el promedio cuadrático de los tiempos residuales, se asume que este proceso acelerará el cálculo.

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

6. Referencias

1. En-Jui , L., He, H., John M., D., Po, C., & Liqiang, W. An optimized parallel LSQR algorithm for seismic tomography. *Computers & Geosciences*, 2013, pp. 184 - 197
2. J. F. Ramírez Cruz, O. Fuentes, R. Romero, & A. Velasco, A Hybrid Algorithm for Crustal Velocity Modeling. *In Advances in Computational Intelligence*. Springer Berlin Heidelberg, 2013, pp. 329-337
3. M. Oiso, Y. Matsumura, T. Yasuda and K. Ohkura. Implementing genetic algorithms to CUDA environment using data parallelization. *Technical Gazette, Hrcak Portal of scientific journals of Croatia*, 2011, Vol.18, No.4, 2011 pp. 511 – 517
4. Kirk, D. B., & Hwu, W.-m. W. Programming Massively Parallel Processors: A Hands-on Approach. San Francisco, CA, USA: Elsevier, 2010.
5. M, Dawei, P. Chen, & L. Wang, Accelerating the discontinuous Galerkin method for seismic wave propagation simulations using multiple GPUs with CUDA and MPI. Earthquake Science, Vol 26, No. 6, 2013, pp. 377-393
6. Chitty, Darren M. "A data parallel approach to genetic programming using programmable graphics hardware." Proceedings of the 9th annual conference on Genetic and evolutionary computation GECCO 07. ACM, 2007, pp. 1566-1573.
7. S. Das, and P. N. Suganthan, Differential Evolution: A Survey of the State-of-the-Art. IEEE Transactions on Evolutionary Computation, Vol. 15, No. 1, 2011, pp. 4-31
8. R. Storn and K. V. Price, “Differential evolution: A simple and efficient adaptive scheme for global optimization over continuous spaces,” ICSI, USA, Tech. Rep. TR-95-012, 1995 [Online]. Available: http://icsi.berkeley.edu/~storn/litera.html
9. J E. Vidale, Finite-difference calculation of traveltimes in three dimensions, Geophysics, vol. 55, No. 5, 1990, pp 521-526.

10. J. A. Hole, Nonlinear high-resolution three-dimensional seismic travel time tomography: Journal of Geophysical Research, Vol. 97, No. 85, 1992, pp. 6553–6562.

7. Autores

Ing. Eustolia Carreón-Esteban. Graduada de la Ingeniería en Sistemas Computacionales en 2009 en el Instituto Tecnológico Superior de Libres. Actualmente, estudiante de la Maestría en Sistemas Computacionales en el Instituto Tecnológico de Apizaco. Áreas de interés: Cómputo Paralelo y Algoritmos Evolutivos.

Dr. José Federico Ramírez-Cruz, Graduado de la ingeniería Industrial en Electrónica en el Instituto Tecnológico de Puebla en 1993, Graduado de la Maestría en Ciencias en la especialidad de Electrónica en el Instituto Nacional de Astrofísica y Óptica en 1994, Graduado del Doctorado en Ciencias, con especialidad en el area de Ciencias Computacionales en el Instituto Nacional de Astrofísica y Óptica en 2003, Realizó una estancia postdoctoral en la Universidad de Texas, en El Paso, en 2011, Es docente de tiempo completo del Departamento de Sistemas y Computación del Instituto Tecnológico de Apizaco. Áreas de interés: Algoritmos Evolutivos, Procesamiento Paralelo y Aprendizaje Automático.

Lic. Sergio Palafox-Ugarte. Graduado de la Licenciatura en Informática en 2006 en el Instituto Tecnológico de Apizaco. Actualmente estudiante de la Maestría en Sistemas Computacionales en el Instituto Tecnológico de Apizaco. Áreas de interés: Cómputo Paralelo y Optimización Evolutiva.

20 Años DE EXCELENCIA

Tec MISANTLA

INSTITUTO TECNOLOGICO SUPERIOR DE MISANTLA

ciiai 2014 Congreso Internacional de Informática Aplicada e Ingeniería Industrial

Imaginación: Soluciones con Ingeniería

otorga el presente:

RECONOCIMIENTO

A la C. Eustolia Carreón Esteban

Por su destacada participación con la ponencia
"ALGORITMO EVOLUTIVO PARALELO PARA LA TOMOGRAFÍA SÍSMICA"
dentro del Congreso Internacional de Informática Aplicada e Ingeniería Industrial celebrado en la ciudad de Misantla, Ver. Los días 13, 14 y 15 de noviembre de 2014.

SEP SEV INSTITUTO TECNOLOGICO SUPERIOR DE MISANTLA
DIRECCION
30EIT0002T

DR. JOSÉ ALBERTO GAYTÁN GARCÍA
DIRECTOR GENERAL

A.3. CUMBRE INTERNACIONAL DE LAS INGENIERÍAS

Principios de programación paralela sobre arquitecturas GPUs con tecnología CUDA en aplicaciones de propósito general GPGPU

S. Palafox-Ugarte, J. F. Ramírez-Cruz, E. Carreón-Esteban y B.E. Pedroza-Méndez

***Resumen*-- CUDA, es una tecnología de programación paralela de aparición relativamente reciente (2006), permite escribir códigos paralelos de propósito general (GPGPU) y ejecutarlos en unidades de procesamiento gráfico (GPUs) conocidas como tarjetas gráficas, relativamente económicas que prácticamente están presentes, frecuentemente en computadoras personales (PCs) convencionales. Así, es posible aprovechar el gran poder computacional de las tarjetas gráficas, mediante la paralelización de algoritmos secuenciales, de una manera simple, sobre todo para los programadores familiarizados con el lenguaje C. En este artículo se exponen los principios básicos para utilizar la tecnología CUDA en la programación paralela de dispositivos GPUs de NVIDIA, y se dan recomendaciones prácticas, tanto para garantizar el máximo aprovechamiento del GPU como para justificar el empleo del mismo, así como la configuración de un sistema de cómputo de alto rendimiento al combinar CPU (Host) + GPU (Device) para la ejecución de códigos paralelos en aplicaciones de propósito general.**

***Índice de Términos*— Arquitecturas GPUs, CUDA, Programación paralela.**

I. INTRODUCCIÓN

Recientemente las capacidades de las Unidades de Procesamiento Gráfico (del inglés Graphics Processing Units, abreviado GPUs), comúnmente llamados chips o tarjetas de gráficos han dejado de crecer linealmente para crecer exponencialmente, de manera específica las tarjetas gráficas a las cuales tiene acceso el consumidor común. Este tipo de tarjetas están avanzando mucho más rápido que la ley de Moore, la cual estipula que aproximadamente cada dos años se duplica el número de transistores en un circuito integrado y en consecuencia el poder de procesamiento del hardware, característica que se ha incrementado más de 10 veces de este periodo en las GPUs.

En los últimos años las unidades de procesamiento gráfico (GPU), que eran originalmente un dispositivo para aplicaciones gráficas, han atraído mucho interés en el campo de la investigación desde el punto de vista de computación de alto rendimiento de bajo costo. Las GPUs se han convertido en dispositivos altamente paralelos con una gran cantidad de núcleos de procesamiento y gran poder de cómputo. En comparación con los sistemas convencionales multiprocesador (workstations, clusters, grids y supercomputadoras) En la actualidad las GPUs son un medio alternativo mucho más accesible para la obtención de un gran poder computacional, brindando el mismo poder de cómputo que los sistemas multiprocesador más recientes, a una décima parte del precio y consumiendo una vigésima parte de la energía. Sin embargo, el rendimiento de las GPU rápidamente ha ido mejorando en los últimos años de tal manera que su máximo rendimiento es mucho mayor que la de las CPUs. El concepto de usar la GPU no sólo para aplicaciones gráficas, sino también para aplicaciones de propósito general se llama computación de propósito general en GPU (GPGPU) [1].

La idea consiste en utilizar la alta concurrencia (ejecutar varias tareas de forma simultanea) de las tarjetas gráficas para aplicaciones de cómputo general como pueden ser simulaciones de tráfico, simulaciones físicas, algoritmos de búsqueda, cálculos con vectores o matrices de grandes tamaños, comparación de secuencias, procesamiento de imágenes médicas y exploración del subsuelo por mencionar algunos propósitos; desde este punto de vista, tanto los fabricantes como los desarrolladores, han considerado esta nueva aplicación de la computación paralela como una prometedora área de investigación, sobre todo, por la amplia variedad de posibles aplicaciones que se pueden aprovechar en el paralelismo disponible en los actuales dispositivos GPUs.

II. ARQUITECTURAS

A. GPU NVIDIA

La tendencia actual, observada tanto en el desarrollo como en la aplicación de los dispositivos GPU ofrecidos por el fabricante NVIDIA, permite acompañar y contribuir a la consolidación del nuevo modelo de programación paralela soportado por la herramienta de programación CUDA (del inglés Compute Unified Device Architecture), donde la GPU, además de ofrecer una mayor capacidad de cálculo paralelo, tiene un papel importante como administrador de múltiples hilos. Esta GPU proporciona una arquitectura unificada, tanto para gráficos como para cálculos, que es fundamentalmente un arreglo escalable de multiprocesadores (MP) multi-hilo, donde cada MP consiste en 'n' unidades de procesamiento gráfico a los cuales se conoce como núcleos (CUDA Cores) cuya

actividad primordial es la manipulación de pixeles, porque las GPUs normalmente procesan un conjunto complejo de polígonos, renderizado de escenas, donde aplica texturas a los polígonos y luego se realiza el sombreado, así como los cálculos de iluminación; debido a esta función es común referirse a los núcleos de un MP como sombreador de pixel (del inglés pixel shader, abreviado PS). Uno de los pasos importantes fue el desarrollo de PS programables, estos fueron efectivamente pequeños programas que la GPU procesó para calcular diferentes efectos. Básicamente el modelo de hardware en las arquitecturas de las GPUs (o tarjetas de video) contienen 'N' MP, y cada uno de estos 'M' procesadores (núcleos) junto con memoria compartida (muy rápida y pequeña), caches de constantes y de texturas (sólo lectura) y finalmente una memoria global, todo dentro del circuito integrado como se muestra en la Fig. 1. La cantidad de MP es directamente proporcional a la capacidad de cómputo del GPU. Cada MP se encarga de la creación, manejo y ejecución de los hilos que están físicamente activos en el dispositivo GPU, teóricamente hasta 512 con el esquema denominado por NVIDIA como "una instrucción y múltiples hilos" (del inglés Single Instruction, Multiple Thread, abreviado SIMT). Debido a que un multiprocesador asigna cada uno de sus hilos a un núcleo y que cada hilo es ejecutado independientemente de los demás, con su propia dirección de instrucción y sus propios registros de estado, las herramientas de programación de NVIDIA ofrecen algunas funciones que son enfocadas precisamente al manejo y optimización de múltiples hilos, como la posibilidad de sincronizar el proceso que realizan todos los hilos en un bloque de código específico.

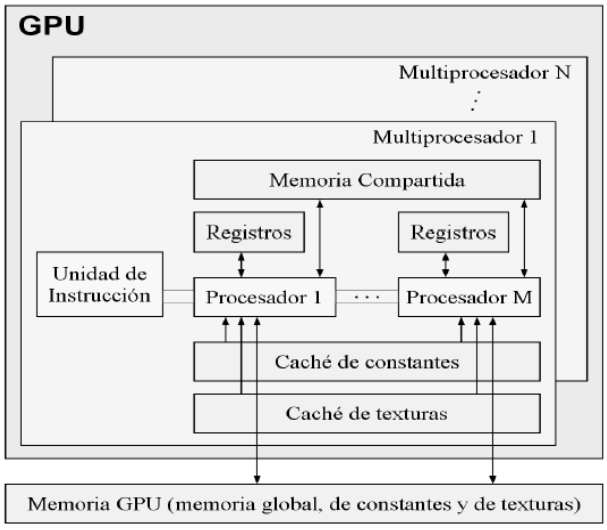

Fig. 1. Arquitectura GPU NVIDIA.

B. CUDA

La arquitectura CUDA, surgió en 2006, cuando NVIDIA lanzó sus tarjetas GeForce 8800 GTX [2], las que por primera vez incluyeron elementos dirigidos específicamente a posibilitar la solución de problemas de propósito general. Poco después, NVIDIA lanzó el compilador CUDA C, el primer lenguaje de programación para desarrollar aplicaciones de propósito general sobre una GPU; con la finalidad de captar la mayor cantidad de programadores que pudieran adoptar esta arquitectura. Esta herramienta promete a la amplia comunidad de desarrolladores lograr aceleraciones importantes durante procesamiento de datos mediante el uso de la tecnología CUDA, la cual se puede ejecutar en tarjetas gráficas NVIDIA para los principales sistemas operativos Linux y Windows. El paralelismo natural de la computación en la GPU se puede aprovechar en técnicas de optimización como lo son los algoritmos genéticos [3]. CUDA C es un lenguaje muy similar a C, al cual se le agregaron un conjunto de instrucciones que harían posible la programación en paralelo en un sólo equipo de cómputo. Se ha evolucionado al grado de tener que "engañar" al GPU utilizando para otros propósitos sus instrucciones para el manejo de pixeles, hasta contar con las interfaces de programación específicas que existen actualmente, por ejemplo CUDA, la cual está diseñada para soportar el esquema SIMT, de tal manera que múltiples hilos pueden ser ejecutados sobre muchos datos.

La herramienta CUDA permite que los programadores escriban el código paralelo, usando lenguaje C estándar más algunas extensiones de NVIDIA, permitiendo organizar el paralelismo en un sistema jerárquico de tres niveles: malla, bloque, e hilo. El proceso comienza cuando el procesador anfitrión (CPU Host) invoca una función para el dispositivo (GPU Device), llamada kernel, enseguida se crea una malla (o arreglo) con bloques de múltiples hilos, para distribuirla en algún MP disponible. Con CUDA, dentro del programa se arranca la ejecución de los kernels paralelos mediante la siguiente sintaxis extendida de llamada a función:

kernel <<<dimGrid, dimBlock>>> (...parameters...);

Donde dimGrid y dimBlock son parámetros especializados que especifican, respectivamente, la dimensión (en bloques) de la malla de procesamiento paralelo y la dimensión (en hilos) de cada uno de los bloques. Prácticamente un Kernel en CUDA es una función en C que se ejecutará 'N' veces en paralelo por 'n' hilos.

El modelo de programación CUDA permite a los programadores lanzar bloques de software para ejecutarse en paralelo especificando el número de hilos por bloque que deben ser generados por los multiprocesadores de la GPU [4].

III. MODELOS DE MEMORIA

Durante la ejecución del kernel, los hilos tienen acceso a seis tipos de memoria dentro del dispositivo GPU, de acuerdo a los siguientes niveles de acceso predefinidos por NVIDIA:

- Memoria global. Es una memoria de lectura/escritura y se localiza en la tarjeta del GPU.
- Memoria para constantes. Es una memoria rápida (cache) de lectura y se localiza en la tarjeta del GPU.
- Memoria para texturas. Es una memoria rápida (caché) de lectura y se localiza en la tarjeta del GPU.

- Memoria local. Es una memoria de lectura/escritura para los hilos y se localiza en la tarjeta del GPU.
- Memoria compartida. Es una memoria de lectura/escritura para los bloques y se localiza dentro del circuito integrado del GPU.
- Memoria de registros. Es la memoria más rápida, de lectura/escritura para los hilos y se localiza dentro del circuito integrado del GPU.

Es importante especificar la ubicación de los diferentes tipos de memoria en la arquitectura del GPU para comprender el flujo y acceso de datos entre (CPU Host) y (GPU Device). La memoria compartida es la más rápida, pero su tamaño está limitado porque está ubicada dentro del circuito integrado [5], la memoria global es la encargada de recibir y retornar los datos a procesar en la GPU. Los threads son capaces de acceder a los espacios de memoria durante su periodo de vida. En la Fig. 2 se describe el acceso a la memoria local por parte de cada thread.

Los bloques de threads tienen un espacio de memoria compartida a la cual pueden acceder todos los threads dentro del bloque. Así mismo, todos los threads tienen acceso a la memoria global de la GPU durante la ejecución de un kernel. Estos espacios de memoria tienen un periodo de vida igual al tiempo de ejecución del kernel que ha sido invocado por los threads.

Tanto la memoria constante como la memoria compartida son recursos escasos, 16 KB y 64 KB, respectivamente, por lo que su uso debe ser limitado de acuerdo a las necesidades particulares de cada aplicación.

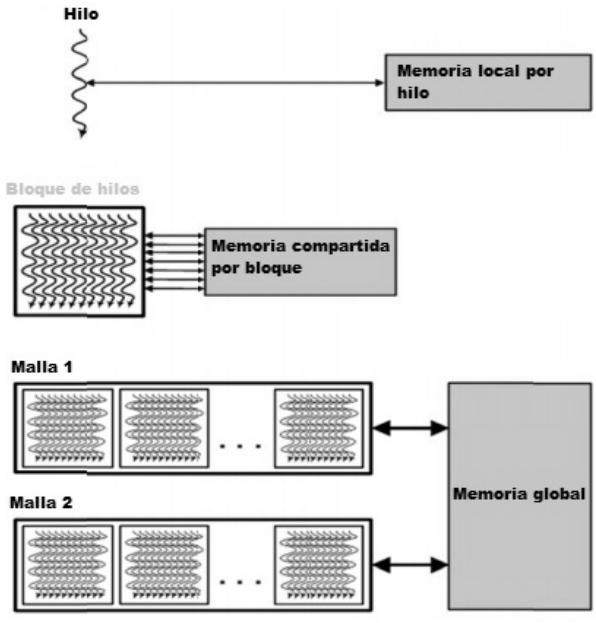

Fig. 2. Niveles de memoria en GPUs NVIDIA.

En la práctica es muy importante hacer énfasis en que, debido a los tiempos relativamente grandes, de retardo (latencia) y al bajo ancho de banda en las trasferencias de memoria, entre la computadora anfitrión [6] (CPU Host) y el dispositivo (GPU Device), es altamente recomendable dividir a la aplicación, de tal manera que cada parte del sistema (hardware) haga únicamente el trabajo que mejor realiza. El uso de la GPU se recomienda si:

- La complejidad de las operaciones justifica el costo de mover datos, hacia el dispositivo GPU. El ejemplo ideal es aquel en el que muchos hilos ejecutan una cantidad considerable de trabajo, ya que las transferencias deben ser minimizadas, los datos deberían mantenerse en le GPU tanto como sea posible.
- La aplicación tiene numerosos datos que pueden ser calculados simultáneamente en paralelo. Esto frecuentemente involucra operaciones aritméticas sobre un gran conjunto de datos, donde la misma operación puede ser realizada sobre miles de elementos al mismo tiempo.
- La aplicación puede ser dividida en operaciones simples, que pueden ser asignadas a gran número de hilos ejecutándose en paralelo.

Con estas consideraciones en mente, lo primero que se tiene que hacer es determinar cuál es la parte del código secuencial que se puede paralelizar mejor usando el GPU. Generalmente se elige como posibles candidatos a todos los segmentos de código que son especialmente demandantes de recursos computacionales (tiempo de procesamiento y memoria). Finalmente, se otorga al GPU aquellos segmentos de código que cumplen con las recomendaciones prácticas antes mencionadas Fig. 3.

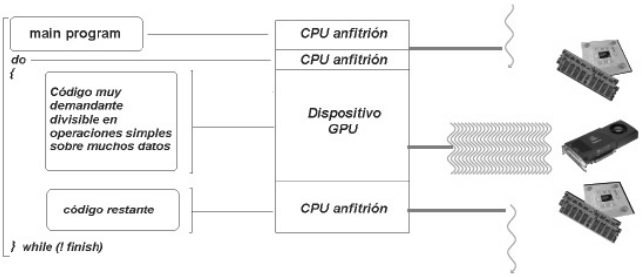

Fig. 3. Estrategia de paralelización.

IV. CODIFICACIÓN

Generalmente, una aplicación codificada en CUDA debe incluir los siguientes pasos:

1. El CPU Host llama al cuerpo principal del programa (main()).
2. Se reserva memoria dentro del dispositivo GPU.
3. Se copian los datos del CPU al dispositivo GPU.
4. El CPU llama a la función kernel.
5. El dispositivo GPU ejecuta el código paralelamente.
6. Se copian los resultados nuevamente a la memoria

del CPU principal.

7. Se libera la memoria reservada dentro del dispositivo GPU.

V. METODOLOGÍA

A. Selección de Características del GPU Device

Las principales características a considerar para seleccionar la GPU son: en primer lugar el uso que se le dará a la tarjeta gráfica; en este caso será para realizar programación paralela de propósito general, concepto utilizado frecuentemente hoy en día en las comunidades científicas por estudiantes e investigadores, pues han encontrado una excelente opción para ejecutar aplicaciones fuera de los gráficos por computadora, desarrollando técnicas para la implementación de simulaciones, bases de datos y algoritmos que requieren computación de alto rendimiento.

En segundo lugar existe una de las características que ha logrado aumentar exponencialmente el uso de GPUs como recurso de cómputo de alto rendimiento y es el bajo precio en relación a su potencia de cálculo, gran paralelismo, optimización para cálculos en coma flotante y sumando el hecho de trabajar en conjunto con el CPU de una computadora de escritorio común, se puede obtener un sistema de cómputo de alto rendimiento, discreto y eficiente sin la necesidad de recurrir al uso de ambientes monumentales como un clúster de computadoras. Con millones de GPU NVIDIA vendidas hasta la fecha, los desarrolladores de software, científicos e investigadores encuentran usos más amplios para la computación de la GPU con CUDA.

En base a las características antes descritas se ha seleccionado la tarjeta GeForce GT 640 que se muestra en la Fig. 4, tomando en cuenta la capacidad de cómputo y por supuesto bajo precio, así como el fabricante (ASUS). Esta tarjeta de video proporciona características acordes para incursionar en el campo de programación paralela con CUDA.

Fig. 4. Tarjeta gráfica seleccionada en relación desempeño-precio.

B. Instalación y Configuración GPU Device

Las consideraciones previas para la instalación son principalmente el tipo de puerto, buses de memoria y consumo de potencia que necesita la GPU para realizar interfaz con la CPU, especificaciones que debe satisfacer la tarjeta madre. En el caso de una computadora de escritorio para conectar una tarjeta de video generalmente se utilizan dos puertos: AGP y PCI, siendo mayormente utilizado este último, pues proporciona prestaciones superiores en cuanto a transferencia de datos. El PCI se encuentra actualmente en la versión 3.0, los buses de memoria actuales son DDR3 y DDR5 y la fuente de energía que suministra la potencia en PC´s de escritorio oscila entre los 500w y 650w. Continuando con las consideraciones previas, dos en especial se deben cuidar a detalle la potencia y la temperatura de la tarjeta gráfica, cuyas especificaciones en este aspecto se suman al consumo de todos los dispositivos que integran el CPU. La GPU tiene una relación potencia-watts mayor que la CPU, para determinar correctamente la potencia total que necesitan funcionando en conjunto CPU + GPU se encuentra disponible un calculador de potencia en la página web de ©ASUSTeK Computer Inc. [7], el cual permite especificar los componentes instalados y como resultado muestra el suministro total de energía en watts. Para finalizar la configuración se instalan los controladores y software proporcionados por el fabricante de la tarjeta gráfica en este caso ASUS, el cual provee la utilidad GPU Tweak, usada para monitorear en tiempo real el estado de la GPU y optimizar su configuración para conseguir un rendimiento óptimo de la tarjeta gráfica.

C. Software CUDA

Para usar la arquitectura de programación paralela CUDA en un sistema operativo Windows, es necesario:

- Tarjeta gráfica compatible con CUDA, en este caso la tarjeta seleccionada es la GeForce GT 640 habilitada con 384 núcleos CUDA.
- Controlador de la tarjeta gráfica, en el caso de GPUs NVIDIA el controlador es general.
- Microsoft Windows XP, Vista, 7 y 8 en arquitecturas de 32 o 64 bits.
- NVIDIA CUDA Toolkit, la versión a instalar de este software es CUDA 5.

Microsoft Visual Studio 2008 o 2010, donde se utilizará como entorno de desarrollo el editor para C++ [8].

VI. IMPLEMENTACIÓN

Para mostrar las ventajas del desempeño de un algoritmo paralelizado en la tecnología CUDA, en comparación con la programación secuencial usada tradicionalmente, se realiza una operación donde se toman dos vectores (v1 y v2) de N elementos de tipo entero. Posteriormente se inicializa v1 con una secuencia ascendente, desde 0 hasta N-1, y v2 con la suma de los valores de v1 en las posiciones i e i+1 elevadas al cubo. Es importante mencionar que esta operación se ha ejecutado tanto de forma paralela (GPU), como de forma secuencial (CPU) para hacer notar el rendimiento en cada uno de los dispositivos.

Se realizaron 10 ejecuciones del programa con diferente número de elementos para los vectores de prueba. Como puede notarse, la aceleración es directamente proporcional al número de elementos a procesar; es decir, mientras mayor sea la cantidad de datos a procesar menor será el tiempo de ejecución.

Cabe mencionar que cuando el número de elementos es menor que 50 millones, el tiempo de proceso en ambos dispositivos es similar, y por ende la aceleración no es

significativa. Esto se debe a que en la transferencia de datos desde la GPU a la CPU se consume mucho tiempo, y esta operación puede resultar innecesaria si la cantidad de datos a procesar es relativamente pequeña.

TABLA I
COMPARACIÓN DE LOS TIEMPOS REGISTRADOS EN AMBOS DISPOSITIVOS

Número de elementos (Millones)	Tiempo (s) CPU	Tiempo (s) GPU	Aceleración
50	0.59	0.418	x1.411
75	0.858	0.562	x1.526
100	1.734	1.093	x1.586
125	2.028	1.248	x1.625
150	1.482	0.889	x1.667
175	1.186	0.702	x1.689
192	2.246	1.311	x1.713

VII. CONCLUSIONES

En este artículo se han expuesto los conceptos básicos, el potencial y las restricciones de los dispositivos de procesamiento de gráficos (GPUs), al emplearse como procesadores paralelos de propósito general. En particular, se ha dado énfasis a las recomendaciones prácticas que procuran el mejor aprovechamiento de esta poderosa herramienta de cómputo.

Finalmente es importante aclarar que debido a la creciente demanda de los gráficos 3D de alta definición en aplicaciones interactivas, las GPUs han evolucionado en arquitecturas de procesamiento masivamente paralelas. La arquitectura de las GPUs modernas es muy eficiente, con la capacidad para realizar cómputo de propósito general ejecutando ciertos algoritmos de forma más rápida y eficiente que una CPU; siendo necesario considerar que no todos los problemas complejos pueden ser resueltos eficientemente en una GPU, la cual es adecuada para resolver problemas complejos que puedan ser expresados principalmente como el cálculo de datos en paralelo.

VIII. REFERENCES

[1] Oiso, Masashi and Matsumura, Yoshiyuki and Yasuda, Toshiyuki and Ohkura, Kazuhiro, “Implementing genetic algorithms to CUDA environment using data parallelization”, Tehni{\v{c}}ki vjesnik, vol.18, pp. 511-517,2011.

[2] Technical Brief NVIDIA GeForce 8800 GPU Architecture Overview November 2006 TB-02787-001_v01

[3] P. Pospichal, J. Jaros and Josef Schwarz, ”Parallel Genetic Algorithm on the CUDA Architecture”, Springer, pp. 442 - 451, 2005.

[4] Sarnath Kannan and Raghavendra Ganji, Porting Autodock to CUDA, 2010.

[5] NVIDIA CUDA C Programming Guide, Versión 4.2, 2012

[6] CUDA C Best Practices Guide, Version 5.0, 2012

[7] http://support.asus.com/PowerSupply.aspx

[8] Installation and Verification on Windows DU-05349-001_v5.0 | October 2012

IX. BIOGRAFÍAS

Sergio Palafox Ugarte
Graduado de la Licenciatura en Informática en 2006 en el Instituto Tecnológico de Apizaco.
Actualmente estudiante de la Maestría en Sistemas Computacionales en el Instituto Tecnológico de Apizaco.
Áreas de interés: Cómputo Paralelo y Estrategias de Optimización.

José Federico Ramírez Cruz
Graduado de la ingeniería Industrial en Electrónica en el Instituto Tecnológico de Puebla en 1993.
Graduado de la Maestría en Ciencias en la especialidad de Electrónica en el Instituto Nacional de Astrofísica y Óptica en 1994.
Graduado del Doctorado en Ciencias, con especialidad en el área de Ciencias Computacionales en el Instituto Nacional de Astrofísica y Óptica en 2003.
Realizó una estancia postdoctoral en la Universidad de Texas, en El Paso, en 2011.
Es docente de tiempo completo de la División de Estudios de Posgrado e Investigación del Instituto Tecnológico de Apizaco
Áreas de interés: Algoritmos Evolutivos, Procesamiento Paralelo y Aprendizaje Automático.

Eustolia Carreón Esteban. Graduada de la Ingeniería en Sistemas Computacionales en 2009 en el Instituto Tecnológico Superior de Libres. Actualmente, estudiante de la Maestría en Sistemas Computacionales en el Instituto Tecnológico de Apizaco.
Áreas de interés: Cómputo Paralelo y Algoritmos Evolutivos

M.C. Blanca Estela Pedroza Méndez. Estudió la licenciatura en Matemáticas Aplicadas en la Universidad Autónoma de Tlaxcala y se graduó en 1993. Posteriormente se graduó como Maestro en Ciencias Computacionales en la Benemérita Universidad Autónoma de Puebla en 1998. Es profesora de tiempo completo de la División de Estudios de Posgrado e Investigación del Instituto Tecnológico de Apizaco y Coordinadora de la Maestría en Sistemas Computacionales. Áreas de interés: Procesamiento de Lenguaje Natural, Procesos Estocásticos y Tutoriales Inteligentes.

IEEE
Advancing Technology for Humanity

ITESCO

SEP S·E·C

INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS VERACRUZ SECTION
INSTITUTO TECNOLÓGICO SUPERIOR DE COATZACOALCOS
IEEE ITESCO STUDENT BRANCH

GRANT THIS

RECOGNITION

TO: ING. EUSTOLIA CARREON ESTEBAN.

FOR YOUR PARTICIPATION WITH INVESTIGATION ARTICLE: **"PRICIPIOS DE PROGRAMACION PARALELA SOBRE ARQUITECTURAS GPUS CON TECNOLOGIA CUDA EN APLICACIONES DE PROPOSITO GENERAL GPGPU"** IN THE 2013 IEEE INTERNATIONAL ENGINEERING SUMMIT HELD FROM 29 TO 31 OCTOBER 2013, IN COATZACOALCOS, VERACRUZ, MEXICO.

CÉSAR PINEDA MORENO
CONFERENCE CHAIRMAN

RICARDO OROZCO ALOR
ITESCO DIRECTOR

TADEO URBINA GAMBOA
TECNHICAL COMITE CHAIRMAN

Bibliografía

Andalon-Garcia, I. R. and Chavoya, A. (2009). Performance comparison of three topologies of the island model of a parallel genetic algorithm implementation on a cluster platform.

Ao, Y. and Chi, H. (2009). Experimental study on differential evolution strategies. In *, 2009. GCIS '09. WRI Global Congress on Intelligent Systems*, volume 2, pages 19–24.

Berger, K.-E. and Galea, F. (2013). An efficient parallelization strategy for dynamic programming on gpu. *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 1797–1806.

Bujok, P. and Tvrdik, J. (2011). Parallel migration models applied to competitive differential evolution. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2011 13th International Symposium on*, pages 306–312.

Cantú-Paz, E. (1995). A summary of research on parallel genetic algorithms.

Cantú-Paz, E. (1997). Designing efficient master slave parallel genetics algorithms.

Cantú-Paz, E. (1998). A survey of parallel genetic algorithms. *CALCULATEURS PARALLELES, RESEAUX ET SYSTEMS REPARTIS*, 10.

Cantú-Paz, E. (1999). *Designing Efficient and Accurate Parallel Genetic Algorithms (Parallel Algorithms)*. PhD thesis, Champaign, IL, USA. AAI9952979.

Chitty, D. M., Malvern, Q., and Ps, W. (2007). A data parallel approach to genetic programming using programmable graphics hardware. In *GECCO 07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1566–1573. ACM Press.

Engelbrecht, A. (2007). *Computational Intelligence: An Introduction*. Wiley.

Gao-yang, L. and Ming-guang, L. (2010). The summary of differential evolution algorithm and its improvements. In *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, volume 3, pages V3–153–V3–156.

Göktürkler, G., Balkaya, Ç., and Erhan, Z. (2008). Geophysical investigation of a landslide: The altındağ landslide site, izmir (western turkey). *Journal of Applied Geophysics*, 65(2):84–96.

González, S. J. D. (2011). Implementación de un algoritmo de evolución diferencial paralelo basado en unidades de procesamiento gráfico. Master's thesis, Universidad Michoacana de San Nicolás de Hidalgo.

Hole, J. (1992). Nonlinear high-resolution three-dimensional seismic travel time tomography. *Journal of Geophysical Research: Solid Earth (1978–2012)*, 97(B5):6553–6562.

Instituto Andaluz de Geofísica (2012). Tomografía sísmica.

Jadhav, S. (2007). *Advanced Computer Architecture Computing*. Technical Publications, 2nd edition.

Kannan, S. and Ganji, R. (2010). Porting autodock to cuda.

Kirk, D. B. and Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Lanz, E., Maurer, H., and Green, A. G. (1998). Refraction tomography over a buried waste disposal site. *Geophysics*, 63(4):1414–1433.

Lee, E.-J., Huang, H., Dennis, J. M., Chen, P., and Wang, L. (2013). An optimized parallel {LSQR} algorithm for seismic tomography. *Computers Geosciences*, 61:184 – 197.

Liu, L. and Guo, T. (2005). Seismic non-destructive testing on a reinforced concrete bridge column using tomographic imaging techniques. *Journal of Geophysics and Engineering*, 2(1):23.

Luong, T. V., Melab, N., and Talbi, E.-G. (2010). GPU-based Island Model for Evolutionary Algorithms. In *Genetic and Evolutionary Computation Conference (GECCO)*, Portland, United States.

Mattson, T., Sanders, B., and Massingill, B. (2004). *Patterns for parallel programming*. Addison-Wesley Professional, first edition.

Méndez, A. M. (2008). Algoritmo híbrido para resolver problemas de optimización con restricciones. Master's thesis, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional.

Mu, D., Chen, P., and Wang, L. (2013). Accelerating the discontinuous galerkin method for seismic wave propagation simulations using multiple gpus with cuda and mpi. *Earthquake Science*, 26(6):377–393.

NVIDIA Corporation (2014). *NVIDIA CUDA C Programming Guide*.

Orocio, L. A. R., Palacios, J. C. B., Fabián, J. L. Q., García, M. A. C., and Cornejo, M. A. (2012). Configuración y programación de un clúster de gpus.

Polymenakos, L. and Papamarinopoulos, S. (2005). Exploring a prehistoric site for remains of human structures by three-dimensional seismic tomography. *Archaeological Prospection*, 12(4):221–233.

Pospichal, P., Jaros, J., and Schwarz, J. (2010). Parallel genetic algorithm on the cuda architecture. pages 442–451. Springer.

Qin, A. K., Huang, V. L., and Suganthan, P. (2009). Differential evolution algorithm with strategy adaptation for global numerical optimization. *IEEE Transactions on Evolutionary Computation*, 13(2):398–417.

Ramírez Cruz, J., Fuentes, O., Romero, R., and Velasco, A. (2013). A hybrid algorithm for crustal velocity modeling. In Batyrshin, I. and Mendoza, M., editors, *Advances in Computational Intelligence*, volume 7630 of *Lecture Notes in Computer Science*, pages 329–337. Springer Berlin Heidelberg.

Reyes, J. V. (2011). Propuesta de evolución diferencial para optimización de espacios restringidos. Master's thesis, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional.

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

Óscar Pintos (2012). Geofísica.

Sun, C., Zhou, H., and Chen, L. (2012). Improved differential evolution algorithms. In *Computer Science and Automation Engineering (CSAE), 2012 IEEE International Conference on*, volume 3, pages 142–145.

Tsutsui, S. and Fujimoto, N. (2009). Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study. In *GECCO (Companion)*, pages 2523–2530.

Tvrdik, J. (2008). Adaptive differential evolution and exponential crossover. In *Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on*, pages 927–931.

Ueno, K. and Suzumura, T. (2012). Gpu task parallelism for scalable anomaly detection. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10.

Vidal, P. and Alba, E. (2010). Cellular genetic algorithm on graphic processing units. In González, J., Pelta, D., Cruz, C., Terrazas, G., and Krasnogor, N., editors, *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, volume 284 of *Studies in Computational Intelligence*, pages 223–232. Springer Berlin Heidelberg.

Vidale, J. E. (1990). Finite-difference calculation of traveltimes in three dimensions. *Geophysics*, 55(5):521–526.

Wong, M. and Wong, T. (2009). Implementation of parallel genetic algorithms on graphics processing units. *Intelligent and Evolutionary Systems*, page 197–216.

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

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

Yu, Q., Chen, C., and Pan, Z. (2005). Parallel genetic algorithms on programmable graphics hardware. In *Lecture Notes in Computer Science 3612*, page 1051. Springer.