Aceleración de algoritmos de computación científica basada en arquitecturas heterogéneas

Manuel Alejandro Tamayo Monsalve

Universidad Nacional de Colombia
Facultad de Ingeniería y Arquitectura
Departamento de Ingenierías Eléctrica, Electrónica y Computación
Manizales, Colombia
2014

Scientific computing algorithms acceleration based on heterogeneous architectures

Manuel Alejandro Tamayo Monsalve

National University of Colombia
Faculty of Engineering and Arquitecture
Electric, electronic and computer engineering department
Manizales, Colombia
2014

Aceleración de algoritmos de computación científica basada en arquitecturas heterogéneas

Manuel Alejandro Tamayo Monsalve

Tesis presentada como requisito parcial para optar al título de:
Magister en Ingeniería - Automatización Industrial

Director:
Gustavo Adolfo Osorio Londoño, Ph.D.

Línea de Investigación:
Análisis de sistemas dinámicos y análisis numérico
Grupo de Investigación:
Percepción y Control Inteligente (PCI)

Universidad Nacional de Colombia
Facultad de Ingeniería y Arquitectura
Departamento de Ingenierías Eléctrica, Electrónica y Computación
Manizales, Colombia
2014

”Mi querido hijo ya ni de aliento puede vivir nuestra unión.
Ya las olas han pasado y han dejado a su paso suelo infértil.
Solo queda en nuestro haber el recuerdo de un propósito.
Simplemente seguiré a tu lado para que mueras en mis manos.”

Agradecimientos

- En primer lugar a Dios por permitirme llegar a este punto.
- A mi familia y seres queridos por acompañarme durante todo este proceso que es la vida.
- A mi tutor el profesor Gustavo Osorio por el apoyo, confianza e inumerables contribuciones.
- A mis compañeros Rubelio Cardona y Luis Fernando Castaño por sus aportes.
- A mis compañeros del grupo PCI por su amistad y aportes.
- Al Departamento de Ingenierías Eléctrica, Electrónica y Computación, la DIMA y la Facultad de Ingeniería y Arquitectura de la Universidad Nacional de Colombia sede Manizales.
- A todas las personas que de una forma u otra hicieron posible este logro.

Resumen

En este trabajo se presenta el uso de arquitecturas heterogéneas para acelerar los cálculos pertinentes a varios algoritmos de computación científica como son los casos típicos de ecuaciones diferenciales parciales. Para poder lograr este objetivo es necesaria una similitud entre las estructuras de los problemas con las arquitecturas de computo, además de buscar formas de optimización tales como el manejo apropiado de los recursos disponibles y emplear características propias de las arquitecturas como la indexación bidimiensional de memoria. Esto es posible gracias a las diferentes configuraciones permitidas por el estándar de computación en paralelo Open Compute Language (OpenCL). Dentro de los diferentes problemas para analizar se encuentran las tres clasificaciones de ecuaciones diferenciales parciales de segundo grado, a saber: ecuación parabólica, elíptica e hiperbólica. Para estas ecuaciones se tomaron problemas clásicos de la literatura, en los que se obtuvo una aproximación de la solución mediante el método explícito. La ecuación de calor en una dimensión, Laplace en dos dimensiones y finalmente onda en una y dos dimensiones. Para comprobar los resultados presentados en este trabajo se realizan comparaciones entre las velocidades de respuesta de los diferentes algoritmos para procesos secuenciales en CPU y paralelos, utilizando procesador con múltiples núcleos y unidades de procesamiento gráfico (GPU); teniendo en cuenta las medidas de tiempo de los procesos de escritura en memoria principal, los tiempos de ejecución del proceso en los dispositivos aceleradores, y el tiempo exclusivo que tarda el kernel en ser ejecutado. Los resultados que aquí se muestran fueron realizados en un computador con procesador AMD black six de 6 núcleos con un reloj de 3300 MHz y 4Gb de memoria RAM; una GPU AMD HD 6700 de 10 unidades computacionales con 2^{22} elementos de proceso, un reloj de 850 MHz y 2Gb de memoria, los cuales son de uso comercial y de fácil acceso a la comunidad científica en general.

Palabras clave:
Computación Científica, Ecuaciones diferenciales parciales (EDP), Unidad de procesamiento gráfico (GPU), Open Compute Language (OpenCL), Computación en paralelo

Abstract

This study describes the use of heterogeneous architectures to accelerate the pertinent calculations for several scientific computing algorithms such as the typical cases of partial differential equations. To achieve this objective a similarity between the structures of the problems with the architecture of these devices is needed, besides searching ways of optimization such as the appropriate use of the available resources and use particular characteristics of the devices as bi-dimensional indexing; which can occur thanks to the different configurations allowed by the standard Open Compute Language (OpenCL). Within the different problems to be analyzed we have the three classifications of the second grade partial differential equations, which are parabolic, elliptic and hyperbolic equations. For this equations we took the classic problems of literature, being explicitly heat equation in one dimension, Laplace in two dimensions and additionally wave in one and two dimensions. To verify the results presented in this study comparisons of each problem were performed between the response rates of the different algorithms for sequential and parallel process, using multicores and GPU; taking into account the measurements of time of the processes of writing in the principal memory and without it, the execution times of the process in the accelerator devices, and the time it takes the exclusive kernel execution. The results shown here were achieved in a computer with a AMD black processing unit of 6 cores, with a 3300 MHz clock and a 4Gb ram memory; also an AMD HD 6700 GPU of 10 computing units with 2^{22} process elements was used, a 850 MHz clock and a 2Gb memory. Which in general terms are for commercial use and have easy access in general to the scientific community in general.

Keywords:

Computational science, partial differential equations (PDE), Graphics processing unit(GPU), Open Compute Language (OpenCL),parallel computing

Lista de Figuras

1-1. Espacio discretizado. 2

2-1. Kernel para la aproximación de la ecuación de calor en una dimensión. . . . 9
2-2. Número de iteraciones en escala logarítmica vs Tiempo para una barra de 10000 puntos. 13
2-3. Malla de resultados ecuación de calor para una barra de 10000 puntos. . . . 13

3-1. Kernel para la aproximación de la ecuación de Laplace en (2+1D) dimensiones. 16
3-2. Comparación de resultados entre el proceso secuencial, el procesador con 6 núcleos y la GPU. sin escritura en memoria principal. 19
3-3. Comparación del *Speed-up* en x entre el proceso y el kernel para la ecuación de Laplace. 20
3-4. Malla de resultados final de la ecuación de Laplace. 21

4-1. Kernel para la ecuación de onda 1 dimensión. 27
4-2. Kernel para la ecuación de onda 2 dimensiones. 28
4-3. Comparación de los tiempos de ejecución en los tres dispositivos para una cuerda con 100000 puntos en escala logarítmica. 31
4-4. Malla de resultados ecuación de onda en una dimensión para una cuerda con 100 puntos. 32
4-5. Comparación de los máximos, mínimos y promedios para los tiempos de ejecución en ms en una escala logarítmica, del procesador secuencial y la GPU, en diferentes tamaños de malla para la ecuación de onda en dos dimensiones. 34
4-6. Comparación del *Speed-up* en x entre el proceso y el kernel para la ecuación de onda en dos dimensiones. 34
4-7. Malla de resultados en un tiempo aleatorio para la ecuación de onda en dos dimensiones con constante de propagación de 500m/s. 35
4-8. Malla de resultados en un tiempo aleatorio para la simulación del estanque de agua con constante de propagación de 1493m/s. 36

Lista de Tablas

2-1. Tiempo de ejecución en ms de la ecuación de calor en una dimensión, para el proceso secuencial, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra. 11
2-2. Tiempo de ejecución en ms de la ecuación de calor en una dimensión, para el proceso con GPU utilizando memoria global , donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra 11
2-3. Tiempo de ejecución en ms de la ecuación de calor en una dimensión, para el proceso con GPU utilizando memoria local , donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra 12
2-4. *Speed-up* en x de la comparación entre el proceso secuencial y la GPU utilizando memoria global para la ecuación de calor en una dimensión, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra. 12
2-5. *Speed-up* en x de la comparación entre el proceso secuencial y la GPU utilizando memoria local para la ecuación de calor en una dimensión, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra. 12

3-1. Tiempo de ejecución en ms y *Speed-up* en x de la ecuación de Laplace en (2+1D) dimensiones con escritura en memoria principal. 18
3-2. Tiempo de ejecución en ms y *Speed-up* en x de la ecuación de Laplace en (2+1D) dimensiones sin escritura en memoria principal. 19
3-3. Tiempo de ejecución en segundos y *Speed-up* en x para la ecuación de Laplace en (2+1D) dimensiones comparando entre el proceso y el kernel mediante indexación 20

4-1. Tiempo de ejecución en ms de la ecuación de onda en una dimensión, para el proceso secuencial, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la cuerda. 30
4-2. Tiempo de ejecución en ms de la ecuación de onda en una dimensión, utilizando la GPU, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la cuerda. 30

4-3. Tiempo de ejecución en ms de la ecuación de onda en una dimensión, para el procesador utilizando los 6 núcleos, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la cuerda. 30
4-4. *Speed-up* en x de la comparación entre el proceso secuencial y la GPU para la ecuación de onda en una dimensión, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra. 31
4-5. *Speed-up* en x de la comparación entre el proceso secuencial y el procesador de 6 núcleos para la ecuación de onda en una dimensión, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra. 31
4-6. Tiempo de ejecución en ms y *Speed-up* en x de la ecuación de onda en dos dimensiones. 34
4-7. Tiempo de ejecución en segundos y *Speed-up* en x para la ecuación de onda en 2 dimensiones comparando entre el proceso y el kernel mediante indexación . 35

Contenido

Agradecimientos ix

Resumen xi

Lista de Figuras xv

Lista de Tablas xvii

1. Introducción **1**
1.1. Estado del arte 4

2. Aceleración de problemas en una dimensión usando GPU. Caso de estudio: Ecuación de calor **7**
2.1. Introducción 7
2.2. Ecuación de calor 7
2.3. Implementación en GPU 8
2.4. Experimento 10
2.5. Resultados 14
2.6. Conclusiones 14

3. Aceleración de problemas en (2+1D) dimensiones utilizando Indexación. Caso de estudio: Ecuación de Laplace **15**
3.1. Introducción 15
3.2. Ecuación de Laplace 15
3.3. Implementación mediante Indexación en GPU 16
3.4. Experimento 17
3.5. Resultados 22
3.6. Conclusiones 23

4. Aceleración de problemas en 1 y 2 dimensiones utilizando indexación y gestión de memoria. Caso de estudio: Ecuación de onda **25**
4.1. Introducción 25
4.2. Ecuación de onda 26
4.3. Implementación 28

4.4. Experimento 29
4.4.1. Una dimensión 29
4.4.2. Dos dimensiones 33
4.5. Resultados 37
4.6. Conclusiones 37

5. Conclusiones y recomendaciones **39**
5.1. Conclusiones 39
5.2. Recomendaciones 42

Bibliografía **43**

A. Anexo: Tutorial OpenCL **49**
A.1. Principios Básicos para crear un código usando OpenCL 50
A.1.1. Kernel 50
A.1.2. Host 51
A.1.3. Programa 52
A.1.4. Ejecución del kernel 52

B. Anexo: Programa OpenCL **54**
B.1. Ecuación de Calor 54
B.2. Ecuación de Laplace 62
B.2.1. Cambios para indexar 68
B.3. Ecuación de Onda 70
B.3.1. Una dimensión 70
B.3.2. Dos dimensiones 78

1. Introducción

Las estructuras de computación heterogéneas como los procesadores con múltiples núcleos y las unidades de procesamiento gráfico o GPU (de sus siglas en inglés *graphic process unit*), tienen una mayor capacidad de procesamiento en relación a las estructuras tradicionales basadas en arquitecturas tradicionales con un único procesador o CPU (de sus siglas en inglés *control process unit*) [1].

Se entiende como computación heterogénea el manejo simultáneo de distintos tipos de unidades computacionales [2]. Generalmente se usan combinaciones de unidades de procesamiento central (CPU) junto con aceleradores, donde los más comunes son: las arquitecturas de motor Célula de banda ancha (*Cell Broadband Engine Architecture CBEA*), las unidades de procesamiento gráfico (GPU) y arreglos de compuertas programables en campo (*Field Programmable Gate Array FPGA*). Estos son económicos y presentan mejores rendimientos en operaciones de punto flotante que las CPU tradicionales [3]. Por su flexibilidad, el manejo de estas unidades brinda la posibilidad de una mayor distribución de tareas en paralelo [59].

El dominio de unidades computacionales con diferentes tipos de estructuras no es una tarea fácil, y requiere de un amplio conocimiento de las mismas. Por este motivo, desde el 2008, los fabricantes de dispositivos buscaron crear el estándar OpenCL. Este es un protocolo de programación abierto, que dada su portabilidad no limita la aplicación a un determinado fabricante. Además OpenCL presenta grandes posibilidades de optimización y ventajas en el cambio rápido de aplicación [38], convirtiéndolo en una herramienta de gran utilidad en el campo de la computación científica.

Debido a la diversidad de unidades computacionales que pueden ser utilizadas, es necesario escoger la que posea una estructura más acorde al tipo de problema a solucionar, en este caso problemas modelados con ecuaciones diferenciales parciales. Las ecuaciones diferenciales parciales son una de las áreas de mayor uso en una de las disciplinas de la computación científica, la mecánica computacional [4]. Esta disciplina tiene gran interés en problemas de simulación de mecánica de fluidos, los cuales tienen la característica de necesitar millones de cálculos similares para tener una aproximación de los resultados [58].

En la mayoria de los casos no es fácil obtener una solución analítica en los problemas modelados con ecuaciones diferenciales parciales [60]. Por esto se han desarrollado diversos métodos

numéricos para la aproximación de soluciones. Estos métodos requieren grandes cantidades de cálculos para que sus resultados sean confiables, por lo que no son viables de realizar sin herramientas computacionales que los faciliten y aceleren.

Para lograr llevar estos problemas al campo de la computación se hace necesaria una discretización del mismo. Para este caso se utiliza el método de diferencias finitas, que discretiza las ecuaciones del problema [5, 6]. En la figura **1-1** se aprecia la discretización en el eje vertical del tiempo, y en el eje horizontal del espacio.

Finalmente, dado que la forma de los métodos utilizados para la aproximación de ecuaciones diferenciales parciales y la arquitectura interna de la GPU son muy similares, posiblemente esta sea la opción que brinde mejores tiempos de respuesta frente a los demás dispositivos aceleradores.

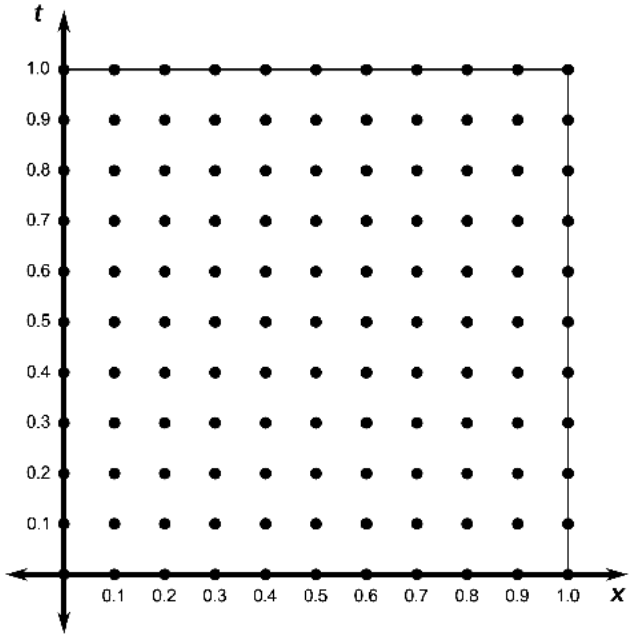

Figura 1-1.: Espacio discretizado.

Para comprobar la eficiencia que tienen las arquitecturas heterogéneas, en la solución número rica de este tipo de problemas, es necesario tomar medidas de desempeño apropiadas a las características que se desean mejorar que, para el caso de este trabajo, son: la aceleración de tiempo de ejecución y la exactitud. Para poder determinar qué tanto mejora el resultado y el tiempo de espera usando dispositivos aceleradores se tomarán medidas de tiempo en diferentes circunstancias, las cuales incluyen los tiempos de escritura en memoria principal, los tiempos de paso de información a los dispositivos, los tiempos de inicialización y finalmente los generados exclusivamente por los dispositivos; permitiendo un análisis más detallado

de las aceleraciones presentes. Para verificar todos los resultados se empleó un sistema con las siguientes características: Un computador con procesador AMD black six de 6 núcleos con un reloj de 3300 MHz y 4Gb de memoria RAM; una GPU AMD HD 6700 de 10 unidades computacionales con 2^{22} elementos de proceso, un reloj de 850 MHz y 2Gb de memoria.

Los experimentos realizados para determinar las mejoras de desempeño también tendrán como propósito analizar distintas formas de implementación que puedan ser utilizadas con el fin de mejorar los tiempos de respuesta, además de brindar una fuente de información mayor de las condiciones bajo las cuales los dispositivos aceleradores, son de mayor utilidad. La primera de estas implementaciones es la relacionada con el uso correcto de las distintas jerarquías de memoria, pasando de una memoria global que es más grande y lenta, a pequeños grupos de memoria local, que a pesar de ser menores en tamaño tiene menores tiempos de acceso. Otra de las alternativas está en una buena gestión de los datos luego de ser ingresados a los dispositivos aceleradores, para evitar procesos de pasos de información innecesarios y re-alimentar dichos datos. Por último y de una forma más trascendente para este trabajo se encuentra el caso de la implementación con indexación de memoria.

La indexación es una posibilidad de implementación que posee la GPU debido a la estructura interna de su memoria, esta capacidad permite organizar los datos en memoria de tal forma que la interacción que puedan tener sea más veloz. Esta herramienta ha sido utilizada en su mayoría para procesamiento de imágenes, pero dada la similitud que existe entre la representación matricial de las imágenes y la representación que se hace en los problemas de dos dimensiones, se logra un mejor uso de esta herramienta.

En el capítulo 2 de este documento se encuentra el estudio de la ecuación de calor en una dimensión, en donde el problema a analizar es una barra aislada en sus extremos a la cual se le aplica un punto de calor y se observa el comportamiento de la misma. Para este caso se analiza el comportamiento del cambio de memoria global por memoria local y se miden las diferentes aceleraciones para distintos tamaños del problema consiguiendo aceleraciones de 2.1x para barras de 10000 puntos.

En el capítulo 3 está el estudio de la ecuación de Laplace en dos dimensiones, para este caso se analiza el estado estacionario de una placa plana a la que se le aplica una temperatura constante en sus aristas, y se observa la transferencia de temperatura a la placa. En este caso se miden los tiempos de aceleración para diferentes tamaños de malla cuadrada y se analiza la incidencia en la aceleración que tiene el proceso de escritura en memoria principal, logrando aumentar la aceleración de 2.1x hasta 4.1x. También se miden los cambios en el tiempo de ejecución que presenta la indexación, para el proceso de la GPU, y para los cálculos realizados en cada uno de los elementos computacionales (kernel).

En el capítulo 4 se analizan dos casos de ecuación de onda, en una y dos dimensiones, el problema para el primero de ellos es una cuerda elástica fija en los extremos, la cual se inicializa lejos de su estado de equilibrio; el segundo es un estanque de agua el cual es perturbado al arrojar un objeto en medio de este. Estos problemas presentan la facilidad de poder reutilizar la información dada anteriormente y separarla en un lugar de la memoria para no tener que recurrir a transferencias innecesarias de datos ya presentes; además para el caso de dos dimensiones se encuentra de nuevo la indexación como herramienta. Para estos dos casos se observan las diferentes aceleraciones en los tiempos de ejecución y los cambios que presenta la indexación en el proceso de la GPU y el kernel, observando cómo la presencia de este método genera aceleraciones superiores a 20x.

En el último capítulo se encuentran las conclusiones y recomendaciones del trabajo, además se presenta un corto tutorial para facilitar la comprensión de las implementaciones mediante OpenCL y finalmente los principales codigos de implementación en OpenCL utilizados en este trabajo. Finalmente en la sección de anexos se puede encontrar un tutorial bósico de como funciona OpenCL, ademas de los principales códigos con que se realizaron las simulaciones de las ecuaciones vistas en este documento.

1.1. Estado del arte

En la literatura se encuentra gran cantidad de documentos referentes a problemas de computación científica, en los cuales se buscan soluciones mediante el uso de arquitecturas heterogéneas y, muchos de estos utilizan casos de estudio con ecuaciones diferenciales parciales como en [37, 43, 52, 53]. Todos estos cambian su forma de ver la utilidad de los dispositivos aceleradores, ya sea midiendo la cantidad de operaciones que pueden realizar en un determinado tiempo (*FLOPS*) [30, 32], midiendo el gasto energético necesario para resolver un determinado problema [35, 45], o como en otros casos, buscando acelerar un problema para resolverlo en el menor tiempo posible [40, 42, 56].

Los trabajos enfocados a acelerar procesos dependen en gran parte de los equipos con los que cuente, por lo tanto en la mayoría de estos se utilizan medidas de desempeño relativas a los propios equipos sin utilizar los aceleradores, para comparar luego sus ventajas, como en [30, 39, 41, 50]. También, muy frecuentemente cambian las arquitecturas según se incremente la complejidad del problema o no consigan las aceleraciones requeridas [32, 42, 45, 51], pudiendo conseguir aceleraciones de más de 80 veces con varios aceleradores y hasta lograr superar las 2000 veces utilizando bancos de más de 50 dispositivos(*cluster*) [47, 51, 55, 57]. Pero en algunos otros casos se busca una mejor implementación de los algoritmos tratando de conseguir mejoras en el desempeño de los mismos con los recursos disponibles [31, 33, 36, 49, 52].

Una de las técnicas utilizadas para acelerar los procesos con GPU es conocida como indexación (*multigrid*) en la cual se asignan índices a los datos para tener una interacción mayor entre ellos y facilitar los procesos, pero este tipo de técnica se ha utilizado en mayor medida en problemas relacionados con procesamiento de imágenes [49, 56], aunque en algunos casos se encuentran en otro tipo de problemas que son más próximos a ecuaciones diferenciales parciales [35, 46, 65–67].

Estos problemas modelados con ecuaciones diferenciales parciales tienen una gran área de desarrollo en la simulación de tejidos como en [62–64]. Estos tienen la particularidad de necesitar miles de datos en forma de células, para poder tener una aproximación a cualquier tejido, y dada su complejidad son procesos que tardan un tiempo considerablemente alto brindando una motivación adicional a trabajos que buscan aceleración de algoritmos enfocados a la computación científica.

Normalmente, las comparaciones de aceleración se realizan mediante puntos de referencia (*benchmark*) [31,33], pero estos solamente permiten la comparación de los dispositivos para ciertos algoritmos, y dado que la cantidad de problemas que pueden encontrar solución en este campo es abundante, las comparaciones que se generan en estos últimos problemas se ejecutan frente a las mismas implementaciones de forma secuencial, ya que en la mayoría de los casos no se tienen super computadores o *benchmark* que permitan comparaciones.

2. Aceleración de problemas en una dimensión usando GPU. Caso de estudio: Ecuación de calor

2.1. Introducción

En este capítulo se presenta como caso de estudio, el análisis de la ecuación de calor. Se realiza la aproximación de la solución usando el método de diferencias finitas explícito, también se muestra cómo el uso de arquitecturas heterogéneas, tales como la GPU muestra considerables ventajas en cuanto a desempeño frente a la CPU tradicional, debido a la posibilidad de ejecutar el mismo patrón de cómputo en cada uno de sus núcleos.

La forma de programar e implementar estos algoritmos en los diversos dispositivos se facilita debido al uso del estándar de computación paralela llamado OpenCL. Este permite, no solo un rápido cambio entre plataforma y dispositivos, sino también la posibilidad de gestionar los recursos para mejorar el desempeño, de tal forma que según se conozcan las estructuras del problema y del dispositivo acelerador, se consigan mejores tiempos de respuesta en los resultados.

El experimento que se realiza está basado en un problema de disipación de temperatura en una barra con los extremos aislados. Las pruebas van cambiando en el número de iteraciones y en el tamaño de puntos que se toman de la barra. También se muestran los cambios que se producen cuando se utilizan memorias globales y memorias locales, donde finalmente se consiguen aceleraciones de hasta 2.1x.

2.2. Ecuación de calor

El primer caso a analizar es una ecuación clásica de problemas con ecuaciones diferenciales parciales parabólicas como es la propagación de calor [7], esta describe los cambios de temperatura en un determinado espacio a lo largo del tiempo. Para definir de una forma concreta el problema, se buscará cómo se disipa la temperatura en una barra homogénea y de longitud

L, previamente calentada y que está aislada en sus extremos. Donde la temperatura T es la solución a una distancia x en la barra después de un tiempo t, y el parámetro α es la difusividad térmica propia del material. Ver ecuación (2-1)

$$\frac{\partial T}{\partial t} - \alpha \frac{\partial^2 T}{\partial x^2} = 0, 0 < x < L, t > 0 \tag{2-1}$$

Existen varios métodos de diferencias finitas para obtener una aproximación numérica, ya sea mediante un esquema explícito, implícito o uno semi implícito que se obtiene al combinar los dos anteriores. Para este caso se utilizará el esquema explícito, que consiste en calcular los valores desconocidos de cada punto en un tiempo posterior con base en un promedio de los vecinos en el tiempo anterior, como se muestra en la figura **2-1**. El cálculo de este tipo de plantilla es ejecutado por cada núcleo o unidad de proceso del dispositivo, y es conocida como kernel [34]. La implementación de este kernel en la ejecución se hace mediante la ecuación (2-2) que aparece como resultado de utilizar el método de diferencias finitas explícito, en donde valores de la primera iteración son conocidos (valores iniciales).

$$T_2(i+1) = \beta T_1(i) + (1 - 2\beta) T_1(i+1) + \beta T_1(i+2) \tag{2-2}$$

Uno de los retos en la aproximación de la solución de ecuaciones diferenciales parciales en CPU es el tiempo de ejecución [8]. Las GPU son una excelente alternativa dada sus capacidades de manejar grandes cantidades de información, de procesarlas de una forma concurrente, y de buscar mejores velocidades en el tiempo de respuesta. Además presentan un consumo energético inferior al de la CPU por tener un reloj interno más lento.

La principal ventaja que tiene la GPU frente a la CPU es la cantidad de procesos que puede ejecutar de forma concurrente. Mientras la CPU puede estar formada por varios núcleos optimizados para el procesamiento en serie, la GPU consta de millares de núcleos más pequeños y eficientes diseñados para el trabajo en paralelo [9]. Por ende es posible realizar las operaciones en cada uno de estos núcleos. También la GPU genera una respuesta con la misma tolerancia de error que la CPU debido a que su estructura está diseñada para cumplir el estándar de representación de punto flotante ieee754 [10].

2.3. Implementación en GPU

Para la programación de las GPU era necesario disponer de los programas propios de cada fabricante [11], ya que existen varias empresas que fabrican estos equipos entonces se desa-

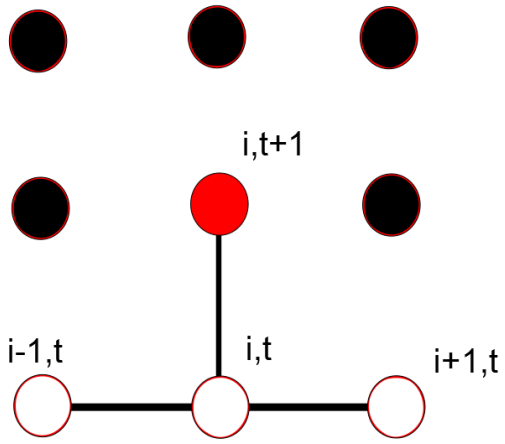

Figura 2-1.: Kernel para la aproximación de la ecuación de calor en una dimensión.

rrollaron varios programas con fines similares. Para unificar tantos criterios fue desarrollado OpenCL [12] como un estándar que no solo sirve para la programación de GPU sino también de otros aceleradores. No obstante, para poder sacar un mejor provecho del dispositivo se hace necesario un conocimiento bastante amplio de las arquitecturas internas de los aceleradores [13, 38].

Mediante el conocimiento de OpenCL y la estructura del problema, es posible hacer un uso más eficiente de los recursos de la GPU [14, 38]. Estos recursos pueden ser tales como el manejo de memoria, que genera mejores rendimientos. La memoria interna de la GPU también cambia según la arquitectura y el fabricante, pero en general cumple un mismo patrón de jerarquía; la memoria global, es la más grande y la más lenta. Luego aparece la memoria local que está presente en pequeñas agrupaciones de núcleos. Finalmente se tiene la memoria privada de cada núcleo.

Dados los arreglos de memoria y la estructura del problema, se hace uso de la memoria local [15, 48], la cual se gestiona luego de enviar los datos a la GPU, y sin la necesidad de

recurrir a nueva información que provenga de la CPU, ver anexo B.1. Esta permite almacenar el resultado de la primera iteración y utilizarlo como datos iniciales de la segunda, para mejorar el rendimiento, y no tener que enfrentar el mayor problema que tienen los dispositivos aceleradores, la comunicación con la CPU [16].

2.4. Experimento

Para poder verificar qué tan eficiente es un dispositivo como la GPU para acelerar ecuaciones diferenciales parciales, se realizaron pruebas en una CPU convencional y en una GPU con capacidad de usar OpenCL. También se realizaron pruebas mediante una mejor administración de los recursos de memoria. El experimento muestra el tiempo de respuesta en milisegundos que tardó cada uno de los dispositivos y su aceleración (*Speed-up*). Esta es la medida de aceleración utilizada para comparar el comportamiento secuencial al comportamiento paralelo. Las características del equipo con que son desarrolladas estas pruebas son: Un computador con procesador AMD black six de 6 núcleos con un reloj de 3300 MHz y 4Gb de memoria RAM; una GPU AMD HD 6700 de 10 unidades computacionales con 2^{22} elementos de proceso, un reloj de 850 MHz y 2Gb de memoria. Las implementaciones se realizaron mediante el estandar OpenCL como se ve en el anexo B.1 y fueron compiladas y ejecutadas mediante *Microsoft Visual C++* 2010 en Windows 7 Profesional.

Las siguientes tablas muestran el tiempo en milisegundos que tardó el proceso en ejecutarse y en guardar la información del resultado. Se realizó de una forma secuencial en CPU y de dos formas en GPU usando memoria global, y usando memoria local. Los datos faltantes en las tablas son debido a que el proceso no se completó por falta de memoria. En la tabla **2-1** se presentan los resultados del tiempo de ejecución utilizado por la CPU, en la tabla **2-2** los resultados de la GPU con memoria global, en la tabla **2-3** los resultados de la GPU con memoria local, y en las tablas **2-4** y **2-5** las respectivas aceleraciones con respecto al procesador secuencial.

El mejor rendimiento obtenido se presenta en el problema con 30000 puntos de la barra y 10000 iteraciones. Para este caso se obtuvo un tiempo de respuesta por parte del procesador secuencial de 687368ms y de la GPU de 326472ms, lo cual entrega una aceleración (*Speed-up*) de 2.1x. Para este caso en particular se muestra como a medida que aumentan las iteraciones se va creando una brecha más amplia entre los tiempos de respuesta entre un procesador secuencial CPU y la GPU con memoria local como se muestra en la figura **2-2** en escala logarítmica.

Finalmente es necesario buscar que el resultado no solo sea rápido sino también preciso. Para

realizar el proceso se comparan una a una las mallas de puntos resultantes, como la de la figura **2-3**, obteniendo resultados exactamente iguales. Esto indica que el manejo de GPU no genera un error adicional al determinado por el método numérico.

Ecuación de calor 1D - CPU secuencial

	10	100	1000	10000	100000	1000000
10	7	6	42	397	3751	33283
100	9	38	309	3052	31874	289179
1000	39	283	3068	30054	318957	
10000	380	2994	30039	347449		
100000	3859	32690	363052			
1000000	38390	424464				

Tabla 2-1.: Tiempo de ejecución en ms de la ecuación de calor en una dimensión, para el proceso secuencial, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra.

Ecuación de calor 1D - GPU con memoria global

	10	100	1000	10000	100000	1000000
10	367	359	566	626	3242	29031
100	395	414	631	2654	24597	
1000	706	981	3175	22277		
10000	4008	6950	28686			

Tabla 2-2.: Tiempo de ejecución en ms de la ecuación de calor en una dimensión, para el proceso con GPU utilizando memoria global , donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra

Ecuación de calor 1D - GPU con memoria local

	10	100	1000	10000	100000	1000000
10	69	64	91	302	2478	24443
100	87	109	339	2142	20551	
1000	341	562	2404	20226		
10000	2626	5381	23490			
100000	25873	55723				

Tabla 2-3.: Tiempo de ejecución en ms de la ecuación de calor en una dimensión, para el proceso con GPU utilizando memoria local , donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra

***Speed-up* - GPU con memoria global**

	10	100	1000	10000	100000	1000000
10	0,019073	0,016713	0,074204	0,634128	1,157001	1,146464
100	0,022784	0,091787	0,489698	1,149962	1,295849	
1000	0,055240	0,288481	0,966299	1,349104		
10000	0,094810	0,430791	1,047165			

Tabla 2-4.: *Speed-up* en x de la comparación entre el proceso secuencial y la GPU utilizando memoria global para la ecuación de calor en una dimensión, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra.

***Speed-up* - GPU con memoria local**

	10	100	1000	10000	100000	1000000
10	0,101449	0,093751	0,461538	1,314569	1,513720	1,361657
100	0,103448	0,348623	0,911504	1,424839	1,550970	
1000	0,11436	0,503558	1,276206	1,485909		
10000	0,144706	0,556402	1,278799			
100000	0,149151	0,586651				

Tabla 2-5.: *Speed-up* en x de la comparación entre el proceso secuencial y la GPU utilizando memoria local para la ecuación de calor en una dimensión, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra.

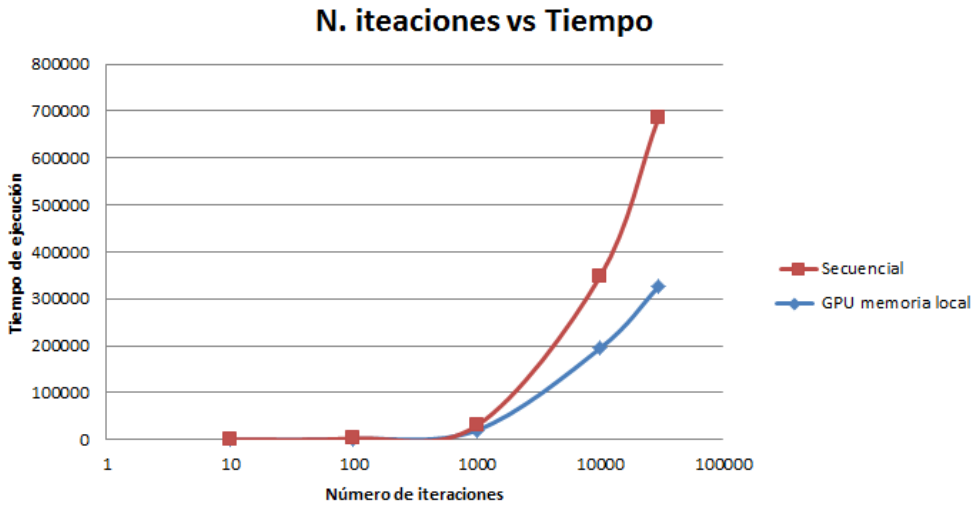

Figura 2-2.: Número de iteraciones en escala logarítmica vs Tiempo para una barra de 10000 puntos.

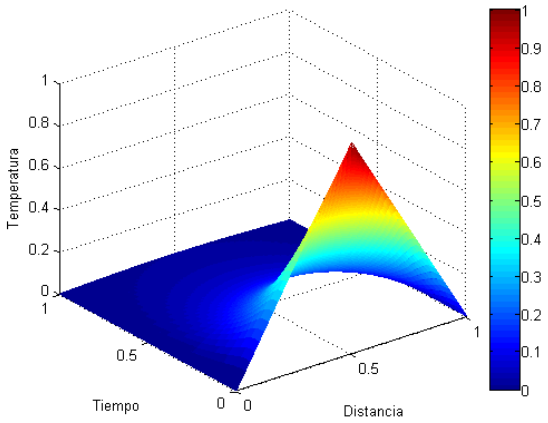

Figura 2-3.: Malla de resultados ecuación de calor para una barra de 10000 puntos.

2.5. Resultados

Al revisar los resultados obtenidos se puede encontrar que no es posible obtener aceleraciones significativas para problemas que contengan poca cantidad de datos usando la GPU. En algunos casos solo el tiempo de la inicialización de los datos en la GPU es mayor que el tiempo que tarda la CPU en ejecutar el proceso completo. Cuando el número de datos a procesar supera la barrera de inicialización, se empiezan a notar las ventajas de utilizar este tipo de dispositivos aceleradores.

También es importante tener en cuenta que solo el hecho de utilizar la GPU no es suficiente para conseguir los resultados óptimos, y se requiere de un conocimiento un poco más amplio para poder aprovechar la estructura del dispositivo en la solución del problema. Esto se evidencia en que la GPU usando memoria global no logró completar los problemas con mayor cantidad de datos. Igualmente es válido tener en cuenta que utilizar cualquier dispositivo con velocidades de reloj inferior a la de la CPU generan menores consumos de energía, lo que es una medida de rendimiento que no es considerada en este trabajo.

Evitando la transferencia de información entre la GPU y la CPU para los datos que son necesarios en futuras iteraciones, se consiguen resultados que evidencian aceleraciones. Este proceso solo se logra gracias al uso correcto de las características de la GPU, en este caso la gestión de memoria local.

Finalmente se logra obtener un pico de aceleración (*Speed-up*) de 2.1 X en una barra con 30000 puntos y 10000 iteraciones. Esta muestra una aceleración considerable del problema planteado, teniendo presente que se trabajó con una GPU comercial que no está específicamente diseñada para el manejo de números, sino a trabajar con procesamiento de imágenes.

2.6. Conclusiones

Luego de realizar las diferentes pruebas para varios tamaños de barra y varias cantidades de iteraciones se lograron aceleraciones cercanas al doble de velocidad de un procesador secuencial. Se observa como la GPU obtiene su mejor desempeño cuando se superan los 100.000 datos, pero teniendo precaución de no superar la capacidad máxima de memoria o de unidades computacionales. También se pudo observar cómo los datos obtenidos con la GPU cuando se utilizó manejo de memoria local fueron en todos los casos superiores a los datos obtenidos mediante la memoria global, lo que comprueba que para acelerar este algoritmo no solo fue suficiente con utilizar un dispositivo acelerador sino también tener un buen manejo de los recursos que posee.

3. Aceleración de problemas en (2+1D) dimensiones utilizando Indexación. Caso de estudio: Ecuación de Laplace

3.1. Introducción

En este capítulo se trabajará la ecuación de Laplace. El problema en concreto a resolver es el calentamiento de una placa plana a la cual se le aplica calor en sus bordes, y en donde se busca hallar la temperatura final de la placa luego de que el calor se distribuya por la misma. Este problema será resuelto mediante el método de Gauss Seidel [18], en donde será utilizado un kernel basado en el promedio de los vecinos espaciales.

Para implementar este algoritmo de una forma más eficiente y orientada a mejores desempeños se utilizará una técnica conocida como indexación [21], la cual consiste en organizar los datos de tal forma que generen una mejor interacción entre sus vecinos. Esto permite que los procesos de saltos de memoria realizados para cada cálculo se ejecuten en menor tiempo.

En el experimento se realizan pruebas de tiempos para diferentes tamaños de mallas cuadradas y con criterios de paradas definidos por una cantidad de iteraciones. También se busca establecer la diferencia que impone la escritura de los datos en memoria principal luego de ser procesados, por lo que se miden tiempos con y sin escritura en memoria principal. De igual forma se busca presentar los beneficios que trae utilizar la indexación, por lo que se realizan cálculos de los tiempos de ejecución del proceso, que consiste en el tiempo de escritura en la GPU, realizar las operaciones correspondientes (kernel) y luego sacar la información del resultado; y el tiempo de ejecución para el proceso de la GPU, y para los cálculos realizados en cada uno de los elementos computacionales (kernel).

3.2. Ecuación de Laplace

El segundo caso de estudio también es una ecuación clásica, pero ahora con ecuaciones diferenciales parciales de segundo orden elípticas como es la ecuación de Laplace [17]. Esta

modela el comportamiento final de una variable en estado estacionario para una región determinada. Problemas relacionados con campos vectoriales tales como la temperatura y los campos magnéticos o gravitatorios son en los que la ecuación de Laplace juega un papel importante. Para este caso se buscará encontrar la temperatura en estado estacionario de una placa calentada de grosor irrelevante, la placa está aislada excepto en sus bordes donde posee una temperatura preestablecida y constante, además no hay pérdidas de calor por lo que la función se iguala a 0. La función es independiente del tiempo por ende el cambio de temperatura T solo depende de la distancia en X y en Y. Ver ecuación (3-1)

$$\Delta T = \frac{(\partial^2 T)}{\partial x^2} + \frac{(\partial^2 T)}{\partial y^2} = 0 \tag{3-1}$$

Para resolver este tipo de ecuaciones que son independientes en el tiempo también existe una variedad de métodos que permiten una aproximación numérica, y para este caso va a ser utilizado el método de Gauss Seidel que cuando se aplica en ecuaciones diferenciales parciales es conocido como método de Liebmann [70]. Este consiste en calcular cada punto de la malla como un promedio de sus vecinos espaciales como se muestra en la ecuación (3-2), siempre que la malla sea cuadrada, y luego iterar cada punto mediante el kernel de la figura **3-1** hasta que no se produzca un cambio considerable o hasta que se llegue a una tolerancia de error deseada.

$$T_2(i,j) = \frac{T_1(i-1,j) + T_1(i+1,j) + T_1(i,j-1) + T_1(i,j+1)}{4} \tag{3-2}$$

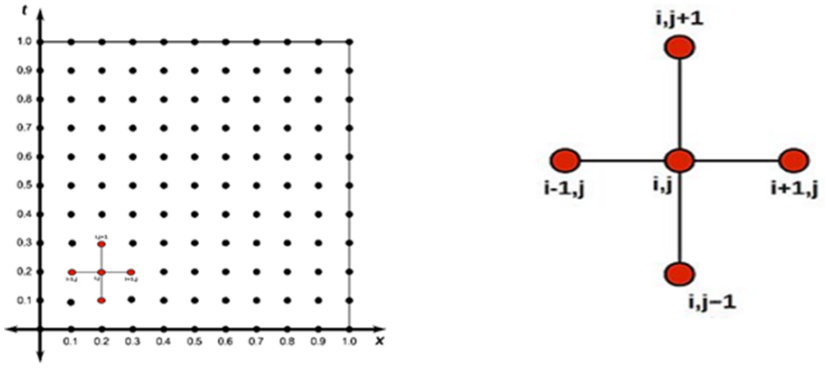

Figura 3-1.: Kernel para la aproximación de la ecuación de Laplace en (2+1D) dimensiones.

3.3. Implementación mediante Indexación en GPU

La indexación es una característica de la GPU que puede ser empleada en aplicaciones que tengan una representación matricial, como es el caso de problemas de algebra lineal, proce-

samiento de imágenes y ecuaciones diferenciales parciales [19]. Este proceso consiste en usar la estructura interna que posee la GPU para procesar la información, de modo tal que la interacción de cualquier elemento con sus vecinos sea más eficiente [20, 44].

Mientras que el proceso normal de almacenamiento en memoria se realiza en forma unidimensional, la estructura interna de la GPU puede distribuir la información de forma bidimensional [21]. Al permitirse una distribución bidimensional, es posible mediante índices de posicionamiento horizontal y vertical lograr que las operaciones que involucran vecinos espaciales se realicen de una forma más rápida .

Para realizar este proceso se hace uso de las herramientas que proporciona OpenCL [22]. La primera es elevar a 2 el índice del número de dimensiones que va a utilizar el programa, para determinar que se hará uso de las características bidimensionales; la segunda consiste en definir los tamaños de mallas con las cuales se piensa trabajar, teniendo precaución con no superar el límite máximo de memoria que posee cada dimensión; finalmente el tercer y último paso consiste en cambiar las características del kernel para utilizar un segundo índice y trabajar las operaciones en forma matricial en lugar de la forma vectorial. Para la implementación en OpenCL ver anexo B.2.

3.4. Experimento

Para verificar los beneficios que brinda el proceso de indexación en la aceleración de este tipo de ecuaciones diferenciales se realizaron pruebas del problema planteado anteriormente en diferentes dispositivos, primero el proceso se realizó ejecutando los algoritmos de forma secuencial para tener el tiempo de comparación, luego se ejecuta en el mismo procesador pero ahora utilizando OpenCL para sacar provecho de los múltiples núcleos, y finalmente se usa la GPU con su capacidad de indexación bidimensional.

En este experimento se analiza la aceleración (*Speed-up*), pero en este caso también se mide la aceleración en procesos aislados de la escritura final en memoria principal, para comparar no solo los tiempos del procedimiento total sino los exclusivos de los aceleradores computacionales. También se plantea las diferencias que implica el utilizar o no la característica de indexación que posee la GPU mediante la comparación realizada entre los tiempos de ejecución del kernel y del tiempo de proceso total de la GPU, que incluye lectura y escritura de los datos en el dispositivo. Por último se comprueba la precisión de los resultados finales entre los diferentes dispositivos.

En la tabla **3-1** se puede observar el comportamiento del tiempo de respuesta en diferentes tamaños de malla. Para todas las ejecuciones el criterio de parada de las iteraciones es

igual a un error de tolerancia de 6 cifras significativas, lo que mostraba una cantidad de iteraciones muy cercana al tamaño de una dimensión en cada una de las mallas; en estos tiempos se incluyen los invertidos en escritura de los datos en memoria principal de todas las iteraciones. También se observa en esta tabla como generalmente las velocidades de la GPU superan las otras arquitecturas, en especial cuando están por debajo de la mitad de su capacidad máxima de elementos de proceso, que para la GPU utilizada es 4096 en cada dimensión. En la tabla **3-2** se observa cómo la aceleración (*Speed-up*) que proporciona utilizar los múltiples núcleos va decreciendo entre mayor cantidad de datos, mientras que la GPU llega a su estado óptimo de la misma forma que en el caso anterior, pero mostrando una diferencia más amplia con respecto a los otros dispositivos, debido a que para este caso no se considera el tiempo de escritura en memoria principal. En la figura **3-2** se muestra, en escala logarítmica, cómo la mejora de velocidad que brindan los múltiples núcleos desaparece y en la GPU se incrementa mientras no sobrepase sus límites de elementos de procesamiento.

Finalmente en la tabla **3-3** se pueden comparar los beneficios que ofrece la indexación de la GPU, en cuanto a los tiempos de ejecución del proceso y del kernel, para solo cinco iteraciones, además de la aceleración que se genera en la GPU por utilizar esta herramienta. Esto se puede observar más claramente en la figura **3-3**. En la malla de resultados de la figura **3-4**, se puede observar la distribución de la temperatura en la placa y que al compararla bit a bit con los resultados presentes en los demás dispositivos se encuentra que son exactamente iguales.

Ecuación de Laplace en (2+1D) dimensiones con escritura en memoria principal

Tamaño	Secuencial	GPU	*Speed-up*	Multinúcleo	*Speed-up*
127*127	305	201	1.51	213	1.43
255*255	992	700	1.41	725	1.36
511*511	4012	2231	1.79	2843	1.41
1023*1023	18821	8636	2.17	11205	1.67
2047*2047	56935	40760	1.39	No Aplica	

Tabla 3-1.: Tiempo de ejecución en ms y *Speed-up* en x de la ecuación de Laplace en (2+1D) dimensiones con escritura en memoria principal.

Ecuación de Laplace en (2+1D) dimensiones sin escritura en memoria principal

Tamaño	Secuencial	GPU	*Speed-up*	Multinúcleo	*Speed-up*
127*127	54	45	1.1	22	2.4
255*255	290	152	1.9	155	1.8
511*511	2202	570	3.8	1433	1.5
1023*1023	18961	4582	4.1	17438	1.1
2047*2047	159818	48117	3.3	No Aplica	
4095*4095	1196996	533216	2.2	No Aplica	

Tabla 3-2.: Tiempo de ejecución en ms y *Speed-up* en x de la ecuación de Laplace en (2+1D) dimensiones sin escritura en memoria principal.

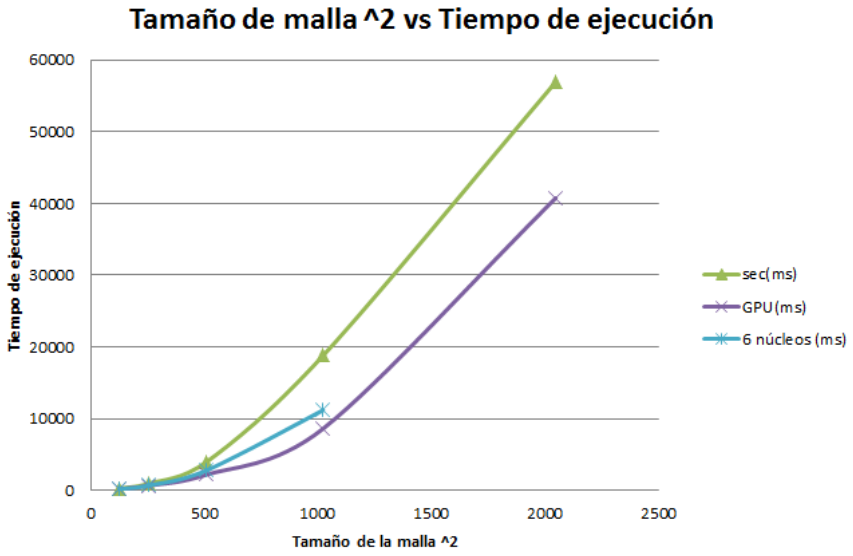

Figura 3-2.: Comparación de resultados entre el proceso secuencial, el procesador con 6 núcleos y la GPU. sin escritura en memoria principal.

Ecuación de Laplace - Comparación del proceso y el kernel mediante indexación en GPU

Tamaño	Proceso sin index	Proceso con index	*Speed-up*
127*127	8E-03	5E-03	1.6
255*255	2.5E-02	9E-03	2.77
511*511	6.2E-02	2.1E-02	2.95
1023*1023	1.65E-01	6.4E-02	2.57
2047*2047	6.02E-01	2.7E-01	2.22
	Kernel sin index	Kernel con index	*Speed-up*
127*127	3.29E-04	2.08E-05	15.83
255*255	8.81E-04	7.50E-05	11.74
511*511	2.54E-03	1.93E-04	13.17
1023*1023	1.00E-02	1.46E-03	6.87
2047*2047	4.01E-02	1.16E-02	3.46

Tabla 3-3.: Tiempo de ejecución en segundos y *Speed-up* en x para la ecuación de Laplace en (2+1D) dimensiones comparando entre el proceso y el kernel mediante indexación .

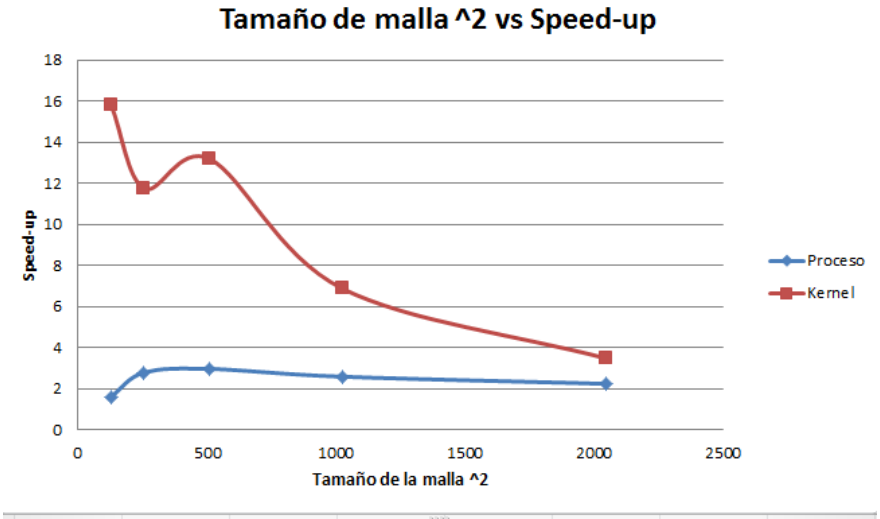

Figura 3-3.: Comparación del *Speed-up* en x entre el proceso y el kernel para la ecuación de Laplace.

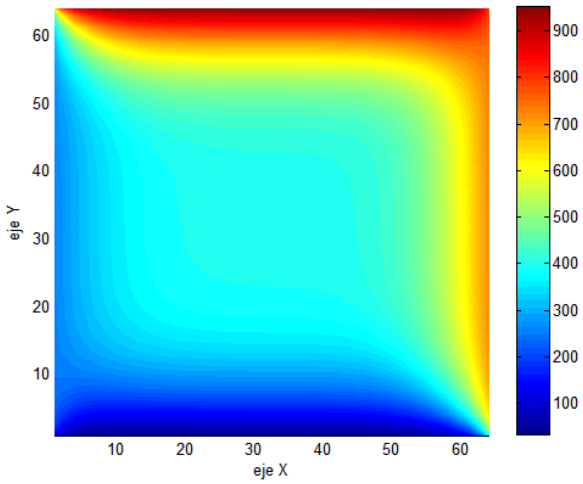

Figura 3-4.: Malla de resultados final de la ecuación de Laplace.

3.5. Resultados

Los resultados obtenidos en este experimento muestran igualmente como el uso de la GPU brinda una mejora considerable en el tiempo, especialmente cuando la cantidad de datos a procesar es alta, pero inferior a la mitad de la capacidad máxima de ésta. Esto es una clara fuente de información acerca de la cantidad de datos que puede llegar a procesar y las ventajas referentes a tiempo que puede llegar a conseguir, dado que sobrepasar la mitad de estos recursos influye considerablemente en el desempeño. Este fenómeno se debe principalmente a dos posibles causas: Primero, la arquitectura de la GPU que fue utilizada tiene un espacio de memoria reservado para manejo de gráficos, incluyendo la salida de vídeo, y este espacio es de alrededor de la mitad de su memoria máxima, como sucede en [68]. Segundo, la gestión de memoria necesita de igual forma de elementos computacionales que realicen esta labor, por ende al utilizar la mitad de la misma es necesario del resto de ella para gestionarla, como en [69].

Al comparar los resultados de la GPU con otro acelerador computacional como es la programación con múltiples núcleos se puede ver como la GPU demuestra un gran problema en principio, debido al tiempo gastado en el proceso de inicialización del dispositivo y en la primera entrada de datos al mismo, mientras que esta dificultad no es tan evidente en el procesador de múltiples núcleos. De igual forma este problema es compensado en gran parte cuando la cantidad de datos aumenta, ya que la GPU realiza los cálculos de una forma más rápida gracias a la indexación, cualidad que no posee el procesador de múltiples núcleos.

También es preciso notar la gran dependencia que sigue existiendo del procesador como tal, ya que al eliminar el tiempo de escritura en memoria principal se puede observar cómo el proceso de aceleración se incrementa y se ve una mayor ventaja al usar la GPU.

El análisis de comparar únicamente los tiempos de ejecución del proceso de la GPU y del kernel es más complejo, pero muestra una gran fortaleza en la implementación mediante indexación. En el tiempo de proceso se puede observar cómo al utilizar esta técnica se obtienen mejoras de desempeño, pero al comparar ambas aceleraciones se puede percibir cómo éste debería ser mayor al que se muestra en los resultados. El problema que se puede observar se debe al mayor tiempo que tarda la memoria de la GPU en organizar los datos de forma indexada, problema que se soluciona con una memoria de mayor velocidad. Esta solución igualmente mejoraría el desempeño de todos los procesos en general, pero al utilizar la indexación se obtendrían comparativamente aún mejores resultados.

3.6. Conclusiones

Luego de realizar las diferentes etapas del experimento se puede observar cómo la GPU genera aceleraciones de hasta 2.17 veces mientras dependa de la escritura en memoria principal, pero que se incrementa hasta 4.1 veces cuando se ignora este procedimiento. También se muestra cómo la cantidad de datos sigue siendo muy importante en el momento de conseguir mejoras de desempeño y que esta vez se establece en un punto inferior a la cantidad de elementos de proceso que posee la GPU.

Por último, vale la pena notar cómo la comparación de los tiempos de ejecución del kernel muestran una aceleración de más de 13x, demostrando las grandes posibilidades que tiene esta implementación. De hecho existen GPU específicas que están destinadas a manejo de información y a cálculo científico, las cuales poseen una memoria dedicada de acceso rápido de datos que pueda realizar la indexación de una forma más eficiente.

4. Aceleración de problemas en 1 y 2 dimensiones utilizando indexación y gestión de memoria. Caso de estudio: Ecuación de onda

4.1. Introducción

En este capítulo se estudiará la implementación de la ecuación hiperbólica de onda, la tercera clasificación de ecuaciones diferenciales parciales de segundo orden. Para analizar el problema se utilizarán dos casos. Primero se define una cuerda elástica fija en sus extremos la cual es inicializada con una onda sinusoidal para observar la evolución de la misma. Segundo se define un estanque de agua cuadrado que es perturbado por un objeto que se arroja en medio del mismo. Estos dos problemas también son resueltos por el método de diferencias finitas y constituyen el análisis de la ecuación en una y dos dimensiones.

Para la implementación de este par de ecuaciones se utilizarán métodos vistos anteriormente, como son el manejo de memoria y la indexación para el caso de dos dimensiones. Además de utilizar una asignación mayor de memoria dadas las características propias que posee este tipo de ecuaciones.

Para el experimento en una dimensión se realizan pruebas con diferentes tamaños de cuerda y diferente número de iteraciones para analizar las aceleraciones que muestran los diferentes dispositivos. Para el caso de dos dimensiones se realizan pruebas con mallas cuadradas de distintos tamaños pero con el número de iteraciones fijo en 200 como ejemplo. También se hacen análisis de los tiempos exclusivos del proceso realizado por la GPU, que incluyen lectura y escritura en el dispositivo, y el tiempo del kernel para determinar las mejoras que brinda el proceso de indexación.

4.2. Ecuación de onda

Como último caso de análisis se encuentra la ecuación de onda, que pertenece al grupo de ecuaciones diferenciales parciales hiperbólicas y describe el comportamiento de la propagación de una onda en un medio [23]. Para este caso se van a utilizar dos tipos diferentes de problemas y así analizar esta ecuación tanto en su forma unidimensional como bidimensional [24].

Para modelar el comportamiento de la propagación de onda en un medio, se usará la ecuación (4-1) en donde Δ es el Laplaciano de U y c es la constante de velocidad de propagación de la onda en el medio [25]. El primer problema consiste en un pulso que viaja a través de una línea unidimensional que tiene los extremos fijos, de esta forma la ecuación de este problema sería la ecuación (4-2) en donde U indica la posición en la que se encuentra la línea en cada instante de tiempo, para este caso se utilizará una cuerda elástica fija en los extremos, la cual se inicializa lejos de su estado de equilibrio. El segundo problema se basa en una membrana elástica plana y cuadrada con los bordes fijos, y que es perturbada por un estímulo en la parte central [26], que para efectos prácticos se trabajara como un estanque de agua cuadrado. En esta ecuación (4-3) la posición de la membrana se indica por la función U que para este caso depende de x, y, t.

$$\frac{\partial^2 U}{\partial t^2} = c^2 \Delta U \tag{4-1}$$

$$\frac{\partial^2 U}{\partial t^2} = c^2 \frac{\partial^2 U}{\partial x^2} \tag{4-2}$$

$$\frac{\partial^2 U}{\partial t^2} = c^2 (\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}) \tag{4-3}$$

Resolver estas ecuaciones tiene un grado mayor de complejidad que las anteriores, ya que no solo involucran en la función una dependencia del tiempo sino también que requieren de información de un instante de tiempo más que los ejemplos anteriores, lo cual conlleva a mayor cantidad de datos y de operaciones, que se hacen necesarias para llegar a una respuesta [28].

Para resolver la ecuación en una dimensión se utiliza un esquema explícito, donde el kernel (ver figura **4-1**), es un promedio de la posición anterior de los vecinos espaciales ponderado con la posición de él mismo en un instante de tiempo atrás, como se muestra en la ecuación (4-4). Además de los valores iniciales se necesita calcular una aproximación de un instante de tiempo adicional, para ejecutar el kernel completamente, esta se basa igualmente en el promedio de los vecinos espaciales. Para el caso con dos dimensiones se utiliza la ecuación (4-5) que se basa en las mismas condiciones que para el caso unidimensional solo que teniendo presente que ahora los vecinos espaciales son cuatro y no dos como en el caso anterior, como se muestra en la figura **4-2**.

$$T_2(i+1) = \gamma^2 T_1(i) + T_1(i+2) + 2(1-\gamma)T_1(i+1) - T_0(i+1) \tag{4-4}$$

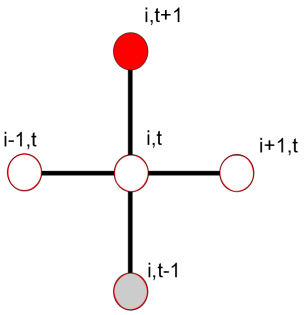

Figura 4-1.: Kernel para la ecuación de onda 1 dimensión.

$$\begin{aligned} T_2(i,j) = \gamma^2(T_1(i,j+1) - (2T_1(i,j)) + T_1(i,j-1)) + \\ \gamma^2(T_1(i+1,j) - (2T_1(i,j)) + T_1(i-1,j)) + \\ 2T_1(i,j) - T_0(i,j) \end{aligned} \tag{4-5}$$

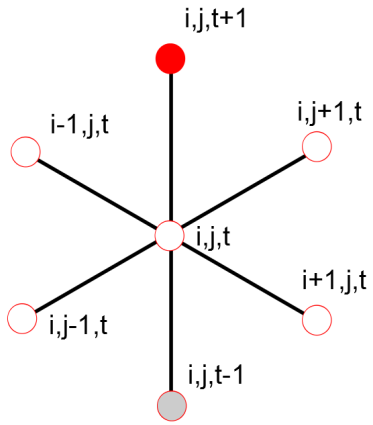

Figura 4-2.: Kernel para la ecuación de onda 2 dimensiones.

4.3. Implementación

Según sean las características del problema se pueden utilizar estrategias vistas en los capítulos anteriores para conseguir mejores formas de implementar este tipo de ecuaciones. Debido a que estas son un poco más avanzadas que las vistas anteriormente generarían un avance sólido hacia ecuaciones de mayor grado de complejidad, dado que este último tipo se acerca más a los modelos reales de simulación [29].

La implementación para la ecuación en una sola dimensión se hizo muy similar a la de distribución de calor vista en el primer capítulo, se realizó una gestión de memoria local con el fin de utilizar la información de la GPU para futuras iteraciones, con el agregado de tener que utilizar información relevante a dos estados de tiempo anterior, lo que lo convierte en un proceso más lento y con mayor necesidad de datos. Para solucionar este nuevo problema de tiempo se realiza una asignación mayor de memoria en el dispositivo y se vuelve a asignar los datos que serán utilizados en la próxima iteración [53]. De esta forma se consigue ahorrar tiempo y se logra evitar nuevas asignaciones de datos para las siguientes iteraciones.

Para la implementación de la ecuación en dos dimensiones se hace uso de la misma estrategia de la ecuación de Laplace, la característica de indexación que posee la GPU, aparte de esto se combina con la asignación de memoria que se usó en la implementación anterior. La

combinación de estos procesos permite un mejor desempeño en cuestiones de tiempo pero afecta en gran medida la cantidad de cálculos que se pueden hacer debido a la abundante información que requiere esta ecuación.

4.4. Experimento

4.4.1. Una dimensión

Para este caso se analizará una ecuación de onda de una dimensión. Ésta describe cómo se transfiere un pulso a través de una cuerda elástica con sus extremos fijos. La cuerda será inicializada por una onda sinusoidal y se observará el efecto oscilante que tiene a través del tiempo en la posición de la cuerda, además se medirá el tiempo de respuesta del programa y se realizará la comparación de aceleración entre la implementación secuencial y otros dispositivos aceleradores, como la GPU y la implementación en múltiples núcleo.

En la tabla **4-1** se muestra el tiempo en milisegundos empleado por el procesador secuencial para dar solución a la ecuación de onda para distintos tamaños de la cuerda y diferente cantidad de iteraciones. La tabla **4-2** muestra el mismo desempeño que la primera tabla, pero esta vez utilizando la GPU con las implementaciones descritas anteriormente. En la tabla **4-3** los tiempos del procesador trabajando con los múltiples núcleos, y en la figura **4-3** se puede observar mejor la diferencia entre los tres procedimientos en una escala logarítmica. Las tablas **4-4** y **4-5** indican las comparaciones de tiempo conseguidas entre el proceso secuencial y las diferentes arquitecturas midiendo en ellas la aceleración (*Speed-up*).

Finalmente se muestra la malla de resultados en la figura **4-8** que indica la posición de la cuerda a través del tiempo, y que al comparar los resultados con los demás dispositivos aceleradores se encuentra una total similitud de los mismos, dejando el mismo índice de exactitud.

Ecuación de onda 1D - CPU Secuencial

	10	100	1000	10000	100000	1000000
10	1	10	30	270	2491	24715
100	2	50	240	2300	23471	224677
1000	40	250	2330	21812	214303	
10000	230	2330	22230	228953		
100000	2020	22281	222066			
1000000	20374	223324				

Tabla 4-1.: Tiempo de ejecución en ms de la ecuación de onda en una dimensión, para el proceso secuencial, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la cuerda.

Ecuación de onda 1D - GPU

	10	100	1000	10000	100000	1000000
10	60	60	90	310	2470	23370
100	100	110	300	2140	18692	200163
1000	430	680	2690	18231	198899	
10000	3751	6211	26801	204667		
100000	36102	61655	257803			
100000	359654	618478				

Tabla 4-2.: Tiempo de ejecución en ms de la ecuación de onda en una dimensión, utilizando la GPU, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la cuerda.

Ecuación de onda 1D - CPU 6 núcleos

	10	100	1000	10000	100000	1000000
10	176	169	205	509	3131	28680
100	183	240	438	2773	23916	237597
1000	267	534	3012	26157	232606	
10000	1195	3765	27995	236675		
100000	9521	35760	260988			
100000	96735	349370				

Tabla 4-3.: Tiempo de ejecución en ms de la ecuación de onda en una dimensión, para el procesador utilizando los 6 núcleos, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la cuerda.

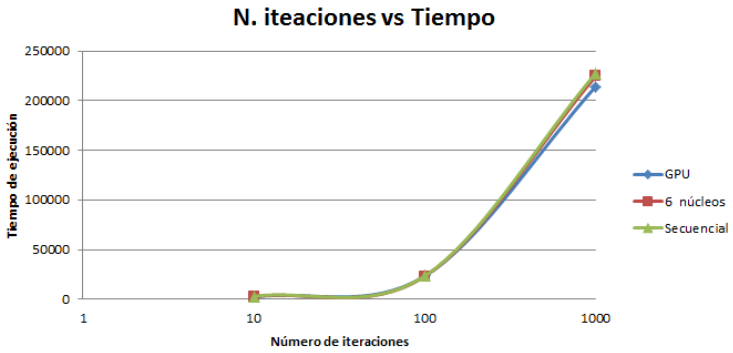

Figura 4-3.: Comparación de los tiempos de ejecución en los tres dispositivos para una cuerda con 100000 puntos en escala logarítmica.

***Speed-up* - GPU**

	10	100	1000	10000	100000	1000000
10	0,016667	0,166667	0,333333	0,870968	1,008502	1,057552
100	0,020000	0,454545	0,800000	1,074766	1,255671	1,122470
1000	0,093023	0,367647	0,866171	1,196424	1,077446	
10000	0,061317	0,375141	0,829447	1,118661		
100000	0,055953	0,361382	0,861379			
100000	0,056649	0,361086				

Tabla 4-4.: *Speed-up* en x de la comparación entre el proceso secuencial y la GPU para la ecuación de onda en una dimensión, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra.

***Speed-up* - CPU 6 núcleos**

	10	100	1000	10000	100000	1000000
10	0,005682	0,059172	0,146341	0,530452	0,795592	0,861750
100	0,010929	0,208333	0,547945	0,829427	0,981393	0,945622
1000	0,149813	0,468165	0,773572	0,833888	0,921313	
10000	0,192469	0,618858	0,794070	0,967373		
100000	0,212163	0,623070	0,850867			
100000	0,210617	0,639219				

Tabla 4-5.: *Speed-up* en x de la comparación entre el proceso secuencial y el procesador de 6 núcleos para la ecuación de onda en una dimensión, donde las filas indican las iteraciones del proceso y las columnas el número de puntos tomados en la barra.

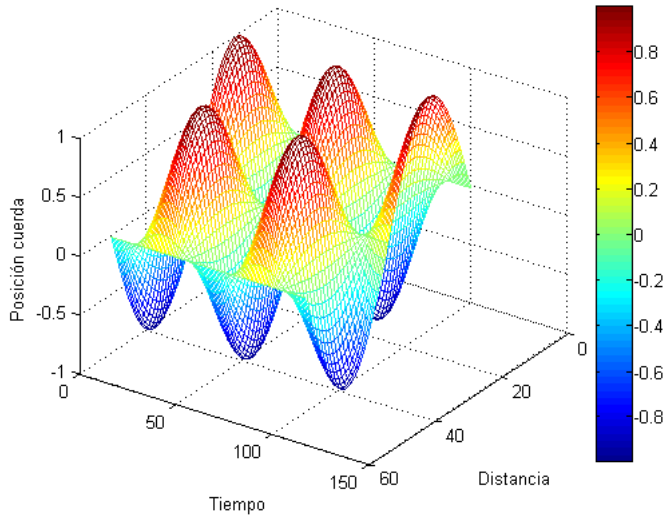

Figura 4-4.: Malla de resultados ecuación de onda en una dimensión para una cuerda con 100 puntos.

4.4.2. Dos dimensiones

Para el caso bidimensional el problema a definir será el comportamiento de un objeto arrojado a un estanque con agua, lo cual define la transmisión de una onda en un espacio plano. Para este caso se supone el medio de propagación un estanque de agua cuadrado, y el objeto arrojado en la mitad del mismo será la perturbación. En este problema se medirán los tiempos de repuesta del proceso secuencial, de la GPU y del procesador de múltiples núcleos para diferentes tamaños de mallas, pero con el mismo criterio de parada de 200 iteraciones. También se analizarán las implicaciones del proceso de indexación realizado para este experimento, sumándole el tiempo correspondiente a la asignación de memoria que se utilizó debido a las características que presenta este problema.

En la tabla **4-6** se muestran los tiempos de respuesta en milisegundos, del proceso completo, que presentaron los diferentes dispositivos. También están sus respectivas medidas de aceleración (*Speed-up*) que muestran cómo los mayores resultados se consiguen para los tamaños de malla más bajos, debido al mayor grado de complejidad que tienen las ecuaciones. En la figura **4-5** se pueden observar los diferentes tiempos de respuesta para el proceso secuencial y la GPU, estos son los tiempos mayores y menores que fueron obtenidos luego de repetir el proceso 20 veces. Para la tabla **4-7** se pueden observar los tiempos de respuesta (para una iteración) del kernel y del proceso exclusivo de la GPU, estos fueron analizados con y sin indexación para determinar las ventajas que se puedan encontrar en el uso de la misma. En la gráfica **4-6** se puede apreciar cómo el pico de aceleración (*Speed-up*) se genera por debajo de la capacidad máxima de elementos de proceso para esta GPU, y que a partir de ese punto empieza a decrecer, del mismo modo que ocurre cuando se hace la comparación frente a procesos secuenciales.

Finalmente en la figura **4-7** se puede ver la malla de resultados de la posición de la membrana con una constante de propagación de 500 metros por segundo en un tiempo determinado, para compararla bit a bit frente a los demás dispositivos y corroborar la exactitud presente en todos ellos. Además con la figura **4-8** que posee una constante de propagación más cercana al física del agua (1493 metros por segundo) se puede comprobar de forma gráfica la simulación del fenómeno físico, mediante el uso de una correcta aproximación a la solución de ecuaciones diferenciales parciales.

Ecuación de onda 2D

Tamaño	Secuencial	GPU	*Speed-up*	Multinúcleo	*Speed-up*
127*127	5990	2571	2.32	3370	1.77
255*255	25542	8571	2.98	9841	2.59
511*511	65302	40622	1.60	42740	1.52
1023*1023	229979	184012	1.24	204004	1.12
2047*2047	808325	810645	0.99	No Aplica	

Tabla 4-6.: Tiempo de ejecución en ms y *Speed-up* en x de la ecuación de onda en dos dimensiones.

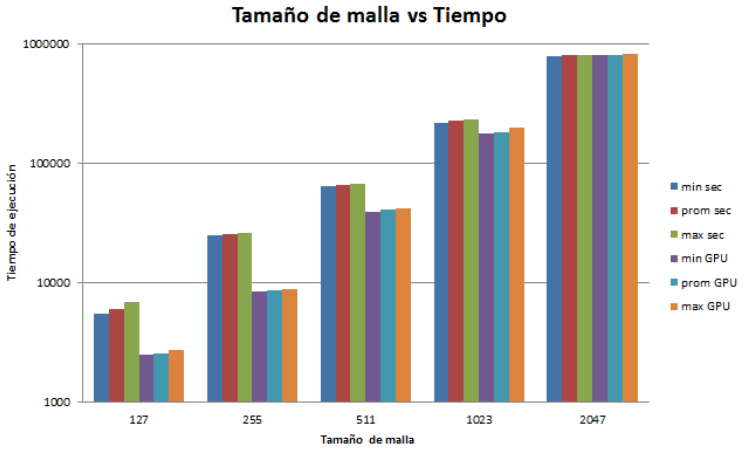

Figura 4-5.: Comparación de los máximos, mínimos y promedios para los tiempos de ejecución en ms en una escala logarítmica, del procesador secuencial y la GPU, en diferentes tamaños de malla para la ecuación de onda en dos dimensiones.

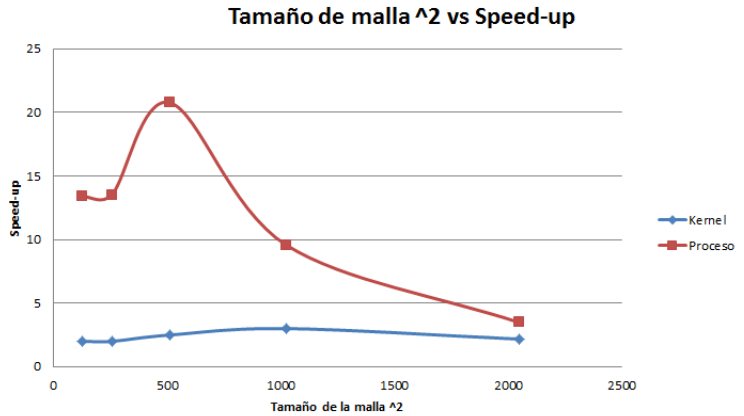

Figura 4-6.: Comparación del *Speed-up* en x entre el proceso y el kernel para la ecuación de onda en dos dimensiones.

Ecuación de onda en 2 dimensiones - Comparación proceso y kernel mediante indexación

Tamaño	Proceso sin index	Proceso con index	*Speed-up*
127*127	2E-02	1E-02	2
255*255	4E-02	2E-02	2
511*511	1E-01	4E-02	2.5
1023*1023	3E-01	1E-01	3
2047*2047	1.09	5	2.18

	Kernel sin index	Kernel con index	*Speed-up*
127*127	4.37E-04	3.26E-05	13.39
255*255	1.25E-03	9.23E-05	13.52
511*511	4.99E-03	2.40E-04	20.75
1023*1023	2.00E-02	2.09E-03	9.56
2047*2047	7.99E-02	2.30E-02	3.47

Tabla 4-7.: Tiempo de ejecución en segundos y *Speed-up* en x para la ecuación de onda en 2 dimensiones comparando entre el proceso y el kernel mediante indexación .

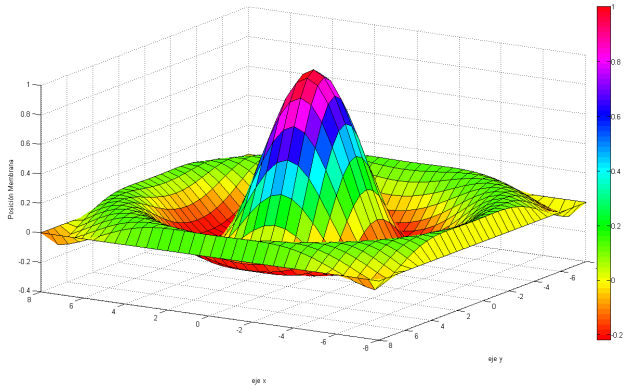

Figura 4-7.: Malla de resultados en un tiempo aleatorio para la ecuación de onda en dos dimensiones con constante de propagación de 500m/s.

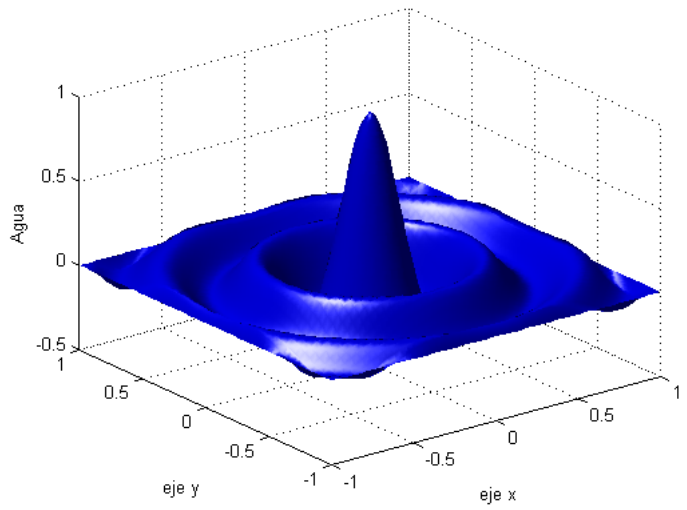

Figura 4-8.: Malla de resultados en un tiempo aleatorio para la simulación del estanque de agua con constante de propagación de 1493m/s.

4.5. Resultados

Los resultados obtenidos para estos dos últimos experimentos muestran cómo según sean las características del problema se pueden implementar una o varias estrategias para sacar el mayor provecho tanto de la estructura del problema como del dispositivo previsto para dar solución al mismo.

También se puede analizar mediante los anteriores experimentos cómo la GPU obtiene su verdadero potencial cuando la cantidad de datos a procesar supera los diez mil elementos y por lo menos cien iteraciones, dado que los tiempos de inicialización y entrada de datos hace de este un proceso lo suficientemente largo para ser considerado inservible en menor cantidad de información. También se realizaron pruebas que permitieran observar la aceleración en ausencia de la escritura en memoria principal, dando como mejor resultado una aceleración de 3.7x para la malla cuadrada de 255 elementos, considerando tiempos de 627ms frente a 168ms para el proceso secuenciales y en GPU respectivamente.

Por último las tablas presentadas para este problema de dos dimensiones contiene la mejor aceleración (*Speed-up*) mostradas en este trabajo, lo cual indica las posibilidades de mejoras de desempeño que ofrecen las arquitecturas heterogéneas para resolver problemas con ecuaciones diferenciales parciales, en especial las que poseen la posibilidad de indexación por tener información en dos ejes espaciales.

4.6. Conclusiones

Luego de realizar ambos experimentos se puede ver nuevamente cómo los dispositivos aceleradores cobran real importancia cuando la cantidad de datos a procesar es superior a 100.000 datos para esta GPU, y que dadas las características del problema en una dimensión solamente se logran aceleraciones de 1.2x utilizando la misma. Pero cuando se accede a un problema más acorde a la arquitectura de la GPU se pueden lograr mejores desempeños, como en el caso de dos dimensiones, que la aceleración presente se acerca a 3x teniendo en cuenta el tiempo de escritura en memoria principal, y superando 3.7x cuando se salta este proceso.

También se puede ratificar mediante la comparación del proceso de la GPU y el kernel cómo la indexación brinda grandes ventajas de desempeño que superan las 20x, pero que no son evidenciadas en los tiempos totales de ejecución debido a la velocidad baja que posee la memoria la GPU con que se realizaron las pruebas, frente a otros dispositivos aceleradores que están diseñados con fines de acelerar algoritmos de computación científica.

5. Conclusiones y recomendaciones

5.1. Conclusiones

Las comparaciones realizadas en este trabajo demuestran la importancia de tener un conocimiento amplio sobre las arquitecturas heterogéneas, para poder lograr beneficios importantes en los tiempos de respuesta de cualquier algoritmo que se quiera implementar en este tipo de dispositivos. Se puede notar cómo la GPU proporciona mejoras de desempeño cuando la cantidad de datos es relativamente alta, y presenta su pico de calidad cuando supera los 100.000 datos para la GPU utilizada, pero teniendo precaución de no superar la capacidad máxima de memoria o de unidades computacionales. No es suficiente contar con una cantidad de datos elevada para conseguir las aceleraciones buscadas, también es necesario contar con una similitud entre los dispositivos aceleradores y la estructura del problema.

Buscando diferentes formas de acelerar los procesos se encontraron métodos que incluyen manejos de memoria local y global, reservas de espacio en las mismas para evitar transferencias de datos innecesarias, evitar procesos de escritura en memoria principal y el proceso de indexación.

Para el primer caso se estudió la ecuación de calor para la cual se realizaron pruebas con distintos tamaños de mallas enfocados a determinar los tiempos de respuesta de esta ecuación para la ejecución secuencial; y poder compararlo con una implementación en GPU. Debido a que las aceleraciones presentadas por la GPU no fueron de un orden muy alto, se realizaron pruebas de comparación utilizando como alternativa el uso de la memoria local de la GPU, logrando mejoras de tiempo en todos los experimentos mediante el uso de la misma. Todos los resultados empleando memoria local fueron mejores a los datos obtenidos mediante la memoria global, aumentando la aceleración de 1.3x hasta llegar a los 2.1x, lo que comprueba la necesidad de una buen gestión de los recursos que poseen los aceleradores como la GPU.

Para el siguiente problema se trabajó la ecuación de Laplace, para este caso se analizaron diferentes tamaños de mallas cuadradas y las comparaciones de velocidad y aceleración presentes entre el procesador secuencial, el multinúcleo y la GPU. Aquí se pudo observar cómo la GPU lograba generar aceleraciones de hasta 2.17x mientras que el procesador multinúcelo solo conseguía aceleraciones de 1.67x. Para este proceso se descartó el tiempo de escritura en memoria principal, con el fin de acercarse un poco más a la aceleración propia de los

aceleradores, incrementando hasta más de 4 veces la aceleración cuando se ignora este procedimiento de escritura.

También hay que tener presente que este problema involucra una segunda dimensión espacial, lo que permite utilizar herramientas como la indexación, un proceso más acordes a la arquitectura de la GPU, y para determinar la importancia de esta forma de implentación se tomaron los tiempos de respuesta del proceso de la GPU y del kernel en presencia y ausencia de la misma, mostrando aceleraciones cercanas a 3x en el proceso completo de la GPU y aceleraciones superiores a 15x cuando se ejecuta solo el kernel. Esta diferencia se debe principalmente a los bajas velocidades que presenta la memoria de la GPU, que no permiten apreciar una aceleración considerable en el proceso completo. Lo anterior demuestra cómo la implementación del proceso de indexación es bastante útil pero que dados los dispositivos con que se realizaron las pruebas no evidencia mejoras significativas, lo cual se solucionaría fácilmente con el cambio de dispositivo. Este cambio supondría mejoras en todos los casos, pero en los relacionados con indexación cobraría mayor importancia.

Para la última clasificación de ecuaciones diferenciales parciales encontramos la ecuación de onda, en este caso se analizaron problemas en una y dos dimensiones. El primero de ellos consiste en una cuerda atada en los extremos que es perturbada por una onda seno, en este problema se compararon las distintas aceleraciones que presentaban tanto el múltiple núcleo como la GPU, pero debido a que esta ecuación tiene un componente adicional en el tiempo obliga a los dispositivos aceleradores a utilizar mayor cantidad de información que en los problemas anteriores, pero a su vez se podía reservar una cantidad de memoria mayor y evitar transferencias de información innecesarias y de esta forma lograr tiempos de ejecución más cortos. Utilizar una mayor cantidad de memoria en estos dispositivos no evidenció aceleraciones significativas, ya que la GPU solo logró aceleraciones de 1.25x y el procesador con múltiples núcleos no consiguió ejecutar en ningún caso el algoritmo en menor tiempo que el procesador secuencial, pero siguió dando una fuente importante de información referente a la capacidad máxima que tiene la GPU y bajo qué cantidad de información resulta útil.

Finalmente, para el caso de la ecuación de onda en dos dimensiones se pudo combinar un poco de los últimos métodos, como son el proceso de reutilización de memoria que se trabajó en la ecuación de onda en una dimensión junto a la indexación vista en la ecuación de Laplace. Para comprobar la eficiencia de estas implementaciones se midieron los tiempos de ejecución para diferentes tamaños de mallas cuadradas, teniendo presente el tiempo de escritura en memoria principal; consiguiendo aceleraciones cercanas a los 3x, y llegando a su mejor desempeño siempre por debajo de la capacidad máxima de memoria y de elementos de proceso. Además para este caso, que representa un problema de mayor complejidad dada su dependencia del tiempo y la necesidad de utilizar más datos, se consideró nuevamente la influencia de la indexación en la GPU. Al realizar comparaciones del proceso de la GPU y el

kernel contando con la indexación y en ausencia de la misma, se mostraron grandes ventajas de desempeño que superan las 20x de aceleración , pero que no son evidenciadas en los tiempos totales de ejecución debido a la baja velocidad que posee la memoria de la GPU con que se realizaron las pruebas, frente a otros dispositivos aceleradores que están diseñados con fines de acelerar algoritmos de computación cietífica. Lo que deja a la indexación como una gran alternativa de implementación en problemas que tengan una estructura similar a la de los dispositivos aceleradores, que además proporcionaría mejoras significativamente mayores en el caso de utilizar una GPU con memoria de acceso rápido y que tenga una estructura diseñada para computación científica.

5.2. Recomendaciones

Las posibilidades que brinda OpenCL para mejorar los algoritmos son más amplias según se cuente con los dispositivos apropiados para implementarlas, o la posibilidad de utilizar bancos de aceleradores, que mejorarían las opciones cuando la velocidad sea lo más importante. También se pueden buscar diferentes métodos para resolver ecuaciones diferenciales parciales, de modo que se puedan acomodar de una mejor forma las características del problema a las arquitecturas de los dispositivos con que se cuenten.

La opción de continuar con los experimentos referentes a ecuaciones diferenciales parciales sigue estando presente, ya sea incrementando los problemas a casos tridimensionales o buscando modelos matemáticos con mayor impacto en problemas de computación científica. O de una forma más simple realizando pruebas que determinen la eficiencia de las implementaciones realizadas en este trabajo en dispositivos con mejor rendimiento en cálculo numérico.

También la posibilidad de continuar con la primera motivación con la que se empezó a realizar este trabajo, la cual está relacionada con la simulación del potencial de acción del tejido cardíaco. Debido a la complejidad y cantidad de datos que maneja este problema, los tiempos de respuesta para mallas relativamente pequeñas es muy alto, por este motivo existe la necesidad de lograr aceleraciones significativas. Igualmente las dificultades que presenta solucionar un problema de simulación mayor están latentes, ya que la estructura de los datos no siempre es similar a la de los dispositivos, los problemas de convergencia y estabilidad de los métodos numéricos se vuelve un reto mayor que implica el cambio de los mismos, y finalmente las implicaciones de tener una ecuación de mayor grado de dificultad con tres dimensiones y que requiera visualización brindan un campo de investigación que puede llegar a dar grandes frutos.

Bibliografía

[1] Q.Liu1. W. Luk. (2012) Heterogeneous Systems for Energy Efficient Scientific Computing. Department of Computing, Imperial College London. London, UK.

[2] N.Rodriguez. M.Murazzo. D.Villafañe. M.Alves. D.Medel. (2013) Integración de Computación Heterogénea con Hadoop para Cloud Computing XV workshop de investigadores en ciencia de la computación.

[3] A.R.Brodtkorb. (2010) Scientic Computing on Heterogeneous Architectures. Faculty of Mathematics and Natural Sciences, University of Oslo.

[4] D.Egloff. (2010). High Performance Finite Difference PDE Solvers on GPUs. QuantAlea GmbH.

[5] Modelos matemáticos en mecánica, consultado en septiembre 14. 2013. En `http://www.math.udel.edu/~fjsayas/MMM.pdf`

[6] M.A.Caro. B.Lora. V.García. (2008) Solución Numérica de la Ecuación de Calor por el Método de las Diferencias Finitas Departamento de Matemáticas de la Universidad del Atlántico, Colombia.

[7] J.Xiaohui. C.Tangpei. C.Dandan. L.W.Qun. (2012). Solving Large-Scale Three-Dimensional Heat Equations on CUDA. 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE).

[8] IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)

[9] L.M.Itu. C.Suciu. F.Moldoveanu. A.Postelnicu. (2011) GPU Optimized Computation of Stencil Based Algorithms. Transilvania Univeristy of BraÅŸov

[10] Que es el GPU Computing consultado en noviembre 9. 2013 `http://www.nvidia.es/object/gpu-computing-es.html`

[11] AMD SDK for OpenCL Applications 2013,`http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/`

[12] The OpenCL Specification versión 1.2, consultado en junio 12. 2013. En `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`

[13] B.K.Bergen. M.G.Daniels. P.M.Weber (2010). A Hybrid Programming Model for Compressible Gas Dynamics using OpenCL. 39th International Conference on Parallel Processing Workshops

[14] J.A.Herdman. W.P.Gaudin. S.McIntosh-Smith. M.Boulton. D.A.Beckingsale. A.C.Mallinson. S.A.Jarvis. (2012). Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. SC Companion: High Performance Computing, Networking Storage and Analysis.

[15] D.L.Foster. (2011).GPU Acceleration of Solving Parabolic Partial Differential Equations Using Difference Equations. The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications.

[16] L.M.Itu. C.Suciu. F.Moldoveanu. A.Postelnicu. (2011)GPU Accelerated Simulation of Elliptic Partial Differential Equations The 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications.

[17] J.J.Ramírez. C.A. Vanegas. A.M. Villegas. (2006) Método de Diferencias Finitas para la Solución de Ecuaciones en Derivadas Parciales. Universidad Eafit, Medellín, Colombia.

[18] N.Alias. (2010) Some parallel numerical method in solving parial differential equations. Universiti Teknologi Malaysia.

[19] J. A. Herdman and W. P. Gaudin, S. McIntosh-Smith and M. Boulton, D. A. Beckingsale, A. C. Mallinson and S. A. Jarvis. (2012) Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. SC Companion: High Performance Computing, Networking Storage and Analysis

[20] N.Bell. S.Dalton. L. N.Olson. (2011) Exposing fine-grained parallelism in algebraic multigrid methods. NVIDEA technical report

[21] J. Cohen. (2013) High Performance Algebraic Multigrid for Commercial Applications. GPU technology conference.

[22] M. Wagner. K. Rupp. J. Weinbub. (2012) A Comparison of Algebraic Multigrid Preconditioners using Graphics Processing Units and Multi-Core Central Processing Units Institute for Microelectronics, TU Wien.

[23] A. R. Brodtkorb.(2010) Scientic Computing on Heterogeneous Architectures.

[24] A.Casasus. J.J.Benito. F.Ureña. L.Gavate. (2009) Resolución de la ecuación de ondas en 2-D y 3-D utilizando diferencias finitas generalizadas. Consistencia y estabilidad. XXI Congreso de Ecuaciones Diferenciales y Aplicaciones. Ciudad real.

[25] F.S. Guzmán. (2010) Solución de la ecuación de onda como un problema de valores iniciales usando diferencias finitas. Revista mexicana de física 56. México.

[26] E.Zuazua. (2009) Métodos Numéricos de Resolución de Ecuaciones en Derivadas Parciales. Basque center for applied mathematics.Bilbao, España.

[27] I. Yanovsky. (2005) Partial Differential Equations: Graduate Level Problems and Solutions. University of California. Los Angeles.

[28] E. Miersemann (2012) Partial Differential Equations. Departament of mathematics. Leipzig university.

[29] F.Gamboa. H. Gonzales. S. E. Guevara. M. Lasprilla. (2008) método de diferencias finitas dominio temporal de cuarto orden con malla variable para simular la propagació n de ondas sísmicas en fracturas. Mecánica computacional vol XXVII. Argentina.

[30] B. Cloutier. B.K.Muite. P.Rigge. (2012) Performance of FORTRAN and C GPU Extensions for a Benchmark Suite of Fourier Pseudospectral Algorithms.Application Accelerators in High Performance Computing (SAAHPC). Chicago.

[31] P.Benner. P.Ezzatti. H.Mena. E.S.Quintana. A.Remon.(2011) Solving differential Riccati equations on multi-GPU platforms. CMMSE 2011. España.

[32] H.Anzt. T.Hahn. V.Heuveline. B.Rocker. (2010) GPU Accelerated Scientific Computing: Evaluation of the NVIDIA Fermi Architecture; Elementary Kernels and Linear Solvers. Engineering Mathematics and Computing Lab. Karlsruhe.

[33] S.Che. M.Boyer. J.Meng. D.Tarjan. J.W.Sheaffer. S.Lee. K.Skadron. (2009) Rodinia: A Benchmark Suite for Heterogeneous Computing. IISWC 2009 .Carolina del norte.

[34] J.A.Stratton. C.Rodrigues. I.Sung. N.Obeid. L.Chang. N. Anssari. G.D.Liu. W.W. Hwu. (2012) Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. illinois.

[35] J.Cabezas. M. Araya-Poloy. I.Geladoz. N.Navarroz E.Moranchoz. Jose.M.Cela. (2009) High-Performance Reverse Time Migration on GPU. Barcelona supercomputing center. Barcelona.

[36] J.R.Seatona. J.C.Sprotta (2011) GPU Accelerated Numerical Solutions to Chaotic PDEs. University of Wisconsin. Madison.

[37] K.Datta. M.Murphy. V.Volkov. S.Williams. J.Carter. L.Oliker. D.Patterson. J.Shalf. K.Yelick. (2008) Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. University of California. Berkeley.

[38] D.Demidov. K.Ahnert. K.Rupp. P.Gottschling. (2013) PROGRAMMING CUDA AND OPENCL: A CASE STUDY USING MODERN C++ LIBRARIES. cornell university. Harford

[39] C.D.Marcottea. R.O.Grigorieva. (2013) Implementation of PDE models of cardiac dynamics on GPUs using OpenCL. Georgia Institute of Technology. Atlanta.

[40] T.Bednarz. L.Domanski. J.A.Taylor. (2011) Computational Fluid Dynamics using OpenCL a Practical Introduction. CSIRO Mathematics Informatics and Statistics. Sydney Australia.

[41] C.Sayin (2012) solving linear equations with conjugate gradient method on opencl platforms. Kadir has university. Turquía .

[42] A.Crestetto. P.Helluy. J.Jung. (2012) Numerical resolution of conservation laws with OpenCL. Universite de Strasbourg. Francia.

[43] V.Rao. N.Agrawal S.Maity. (2012) C-DAC's Efforts - Application Kernels on HPC Cluster with GPU Accelerators. Computational Resource Centre Singapore. Singapore.

[44] P.Collingbourne. C.Cadar. P.H.J. Kelly. (2011) Symbolic Testing of OpenCL Code. Department of Computing Imperial College. London.

[45] P.Dua. R.Webera. P.Luszczeka. S.Tomova. G.Petersona. J.Dongarraa. (2010) From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. University of Tennessee. Knoxville.

[46] B.Gaster. T.Mattson. (2010) OpenCL. An introduction for programmers. Advanced micro devices.

[47] N.K.Choudhary. S.Navada (2010) an exploration of opencl on multiple hardware platforms for a numerical relativity application. Department of Electrical and Computer Engineering. North Carolina State University.

[48] Intel. (2010) IntelÂ® SDK for OpenCL* - 3D Fluid Simulation SampleIntelÂ® SDK for OpenCL* - 3D Fluid Simulation Sample.

[49] J.Bolz. I.Farmer. E.Grinspun. P.Schroder. (2003) Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. ACM 2003. New york.

[50] H.Anzt. S.Tomov. M.Gates. J.Dongarra. V.Heuveline. (2012) Block-asynchronous Multigrid Smoothers for GPU-accelerated Systems. Proceedings of the International Conference on Computational Science, ICCS 2012.

[51] J.Eaton. (2013) GPU-Accelerated Algebraic Multigrid for Commercial Applications. Manager NVAMG CUDA Library. NVIDIA.

[52] V.Heuveline. D.Lukarski. N.Trost. J.Weiss. (2011) Parallel Smoothers for Matrix-based Multigrid Methods on Unstructured Meshes Using Multicore CPUs and GPUs. Engineering Mathematics and Computing Lab (EMCL).

[53] H.Knibbe. C.W.Oosterlee. C.Vuik. (2011) GPU implementation of a Helmholtz Krylov solver preconditioned by a shifted Laplace multigrid method. Journal of Computational and Applied Mathematics.

[54] M.Liebmann. (2011) Algebraic Multigrid Methods on GPU-Accelerated Hybrid Architectures. Institute for Mathematics and Scienti

c Computing University of Graz. Austria.

[55] Z.Feng. Z.Zeng. (2010) Parallel Multigrid Preconditioning on Graphics Processing Units (GPUs) for Robust Power Grid Analysis. Department of ECE Michigan Technological University Houghton. California.

[56] H.Kostler. D.Ritter. C.Feichtinger. U.Rude. (2010) Performance of Multigrid Solvers on GPUs. University Erlangen-Nurnberg. Sydney.

[57] M.Clark. R.Browser. M.Cheng. (2012) Hierarchical algorithms on heterogeneous architectures. Nvidia.

[58] T. John. (2007) Transport, Reaction and Mixing in fluid flows.Universty of California. Santa Barbara.

[59] M.Hegland. (2011) Numerical solution of PDEs with hybrid and heterogeneous computing models. Fujitsu Laboratories Europe (FLE).

[60] H.Topcuoglu. S.Hariri. M.Y. Wu (2003) Performance effective and low complexity task scheduling for heterogeneous computing. IEEE transactions on parallel and distributed system.

[61] C.Lee. W.W.Ro. (2012) Cooperative Heterogeneous Computing for Parallel Processing on CPU/GPU Hybrids. Interaction between Compilers and Computer Architectures (INTERACT). New Orleans, LA.

[62] E.Bartocci. E.M.Cherry. J.Glimm. R.Grosu. S.A.Smolka. F.H. Fenton. (2011) Toward Real-Time Simulation of Cardiac Dynamics. Proceedings of the 9th ACM International Conference on Computational Methods in Systems Biology. Paris, France.

[63] F.Lionetti. A.McCulloch. B.S. Baden. (2009) GPU Accelerated Solvers for ODEs Describing Cardiac Membrane Equations. University of California. San Diego.

[64] F.Lionetti. (2010) GPU Accelerated Cardiac Electrophysiology. University of California. San Diego.

[65] C.Harley. (2012) Numerical Simulation of the Frank-Kamenetskii PDE: GPU vs. CPU Computing. University of the Witwatersrand South afirca.

[66] D.M.Dang. C.C.Christara. K.R.Jackson. (2009) A parallel implementation on GPUs of adi finite difference methods for parabolic pdes with applications in finance. Canadian applied mathematics quarterly.

[67] A.Gaikwad. I.M.Toke. (2009) GPU based Sparse Grid Technique for Solving Multidimensional Options Pricing PDEs. WHPCF. Portland. Oregon.

[68] S.Asselin. A.West. (2014) Graphics Acceleration in View Virtual Desktops. VMware Horizon 6 with View

[69] S.Kato. M.McThrow. C.Maltzahn. S.Brandt. (2012) Gdev: First-Class GPU Resource Management in the Operating System. Department of Computer Science. UC Santa Cruz.

[70] S.C.Chapra. RP.Canale (2006) Métodos Numéricos para Ingenieros. Mcgraw-Hill.

A. Anexo: Tutorial OpenCL

Se busca dar una breve introducción a la principal herramienta computacional con que se cuenta para el desarrollo de este trabajo. OpenCL es un estándar para programación en paralelo que generaliza el uso tanto de multiprocesadores como de dispositivos aceleradores. Este estándar fue creado por APPLE pero desarrollado en conjunto con AMD, IBM, INTEL y NVIDIA y fue llevado al grupo Khronos para convertirlo en un estándar abierto y libre de derechos.

El uso de OpenCL permite la implementación en una gran variedad de aceleradores computacionales, facilitando de esta forma la comparación entre los mismos, y aprovechar según la estructura del problema las ventajas que presenta los diferentes dispositivos. La programación de OpenCL está principalmente compuesta de 3 partes: una es un lenguaje de programación de kernel basado en C99; otra una potente interfaz de aplicación de programación y un tiempo de ejecución eficiente de kernels en CPU o GPU [1].

Como primera medida para hacer uso de esta herramienta hay que determinar para qué dispositivo se quiere instalar el OpenCL, ya que es necesario saber que kit de desarrollo de software (SDK) se debe descargar, ya que cada compañía de hardware brinda sus propias aplicaciones para cada sistema operativo.

Estos SDK determinan la compatibilidad de los dispositivos aceleradores con cualquiera de las versiones de OpenCL e instala la versión adecuada; además se encarga de crear un directorio raíz con las librerías necesarias para la compilación del programa.

Luego según el sistema operativo se utiliza un entorno de programación para lenguaje C, que en el caso de Windows se utilizará el Visual Studio. Teniendo listo el entorno de programación y el SDK instalado, el paso a seguir es crear un proyecto vacío, a continuación seguir los siguientes pasos:

1. Ir a la pestaña proyecto.
2. Propiedades de proyecto, se abre una ventana.
3. En esta ventana seleccionar propiedades de configuración en la lista de la izquierda.
4. Cambiar en el menú desplegable donde esta Active(Debug) por la opción Todas las configuraciones.

5. Nuevamente en la lista de la izquierda seleccionar: C/C++ General.
6. Agregar en "Directorios de inclusión adicionales" el siguiente comando: $(variable de entorno/include. En el caso de utilizar un procesador Intel por ejemplo el comando sería de la siguiente forma : "$(INTELOCLSDKROOT)/include"

7. En la lista de la izquierda seleccionar: Vinculador General.
8. Agregar en "directorios de bibliotecas adicionales" el siguiente comando: $(variable de entorno)/lib/x86.
9. En la lista de la izquierda seleccionar: Vinculador Entrada.
10. Agregar al final de "dependencias adicionales" el siguiente comando: OpenCL.lib. Ahora este proyecto ya está listo para poder compilar con OpenCL.

A.1. Principios Básicos para crear un código usando OpenCL

A.1.1. Kernel

En primer lugar debemos buscar en que forma podemos paralelizar el problema de modo que la herramienta de OpenCL sea de ayuda, para poder tener una mejor visión de esto se utilizara el problema clásico de sumar dos vectores. Generalmente para resolver este problema en CPU se utilizarían los dos vectores donde se encuentra la información, un tercero para almacenar el resultado, y finalmente una variable que indique la posición de los vectores. Este proceso se realiza secuencialmente, realizando primero la suma de la primera posición y así sucesivamente hasta completar el tamaño de los vectores, pero al ser el resultado de la suma actual independiente de los resultados anteriores se puede empezar a pensar como llevarlo al paralelismo.

Dicho lo anterior la manera más fácil de resolver este problema es independizando las sumas de modo que cada una pueda hacerse de forma concurrente y así obtener un resultado en menor tiempo. Es aquí donde entramos a una de las primeras partes para generar un código en OpenCL, el Kernel, que es la función que se ejecutará en el dispositivo.

Para el ejemplo de la suma de dos vectores el kernel se crearía utilizando tres parámetros que serían los dos vectores de entrada de la información y el vector de salida, además internamente para determinar qué posición del vector será sumada se le solicita mediante una función cual es el número de identificación del hilo que está corriendo, y ya que OpenCL genera tantos hilos como disponga el dispositivo cada uno estará encargado de realizar una suma.

```
__kernel void vector_add_gpu {
__global const float* a,
                        __global const float* b,
                        __global float* r)
int idx = get_global_id(0);
r[idx] = a [idx] +b[idx];
}
```

A.1.2. Host

El siguiente paso es configurar el host o anfitrión, para esto definiremos unos conceptos necesarios dentro del mismo y poder entender mejor el procedimiento necesario para ejecutar el kernel:

-Plataforma: la plataforma es la lista de dispositivos que OpenCL puede gestionar, en donde se puede compartir recursos y ejecutar kernel.
-Dispositivo: un dispositivo es un conjunto de unidades de computo, generalmente CPU múltiple núcleo y GPU.
-Contexto: el contexto es el entorno donde se incluye el kernel, donde se gestiona la memoria y donde se sincroniza la ejecución.
-Lista de comando: es un objeto que contiene los comandos que se ejecutarán en un dispositivo específico.
-Memoria: es un espacio reservado en el contexto para guardar la información enviada o recibida del dispositivo.
-Programa: Un programa de OpenCL consiste en un conjunto de núcleos. Los programas también pueden contener funciones auxiliares convocadas por las funciones kernel y los datos constantes.
A continuación se procede a crear el programa en C para ejecutar el código.
1. Incluir las librerías necesarias incluyendo CL/cl.h.
2. Obtener la lista de plataformas disponibles usando el comando clGetPlatformIDs que utiliza como parámetros el número de plataformas a buscar, un puntero donde se guardará la información de tipo cl platform id y por último una bandera de error.
3. Obtener la información de dispositivos disponibles en la plataforma usando clGetDeviceIDs que necesita la plataforma, el tipo del dispositivo, el número de dispositivos a buscar, un puntero donde guardar la información tipo cl device id y de nuevo la bandera de error.
4. Crear las propiedades para generar el contexto donde principalmente se debe referenciar la plataforma donde se va a crear.
5. Crear el contexto mediante clCreateContext que requiere las propiedades creadas en el paso anterior, el número de dispositivos dentro del contexto, la información del dispositivo,

2 parámetros que se usan para extraer información de errores durante la ejecución pero que pueden ser omitidos, y la tradicional bandera de error.
6. Generar la cola de comandos para el dispositivo usando clCreateCommandQueue que usa el contexto, el dispositivo, un opcional de propiedades que pueden incluir si la cola se ejecuta fuera de orden y también la creación de un perfil de la cola de comandos, y la bandera de error.

A.1.3. Programa

En la creación del programa se hace necesario leer y compilar el kernel dentro del contexto para poder unirlos y ejecutarlos, para esto la opción más adecuada es seguir los siguientes pasos:

1. Crear el kernel dentro del código original como una constante de caracteres para facilitar la lectura.
2. Creamos el programa mediante el la función clCreateProgramWithSource que utiliza como parámetros el contexto, el kernel como constante de caracteres, el tamaño del kernel si se desea y la bandera de error.
3. Compilar el kernel mediante clBuildProgram que solo utiliza principalmente el programa generado anteriormente.
4. Ahora se genera el kernel dentro de la plataforma usando clCreateKernel que utiliza el programa, necesita el nombre dado al kernel y contiene la bandera de error; esta función debe ser asignada a una variable tipo cl kernel.
5. Ahora se crean los objetos de memoria en el dispositivo mediante clCreateBuffer que necesita el contexto, una descripción del uso de esta memoria, el tamaño de esta memoria, un puntero de la información en el host que va a compartir este objeto de memoria y la bandera de error; esta función debe ser asignada a una variable tipo cl mem.

A.1.4. Ejecución del kernel

Finalmente se juntan los últimos elementos para ejecutar el kernel en el dispositivo, para luego extraer la información del mismo y por último borrar todo el espacio reservado en él.

1. Definir los argumentos en el kernel con clSetKernelArg que necesita el kernel, el índice del argumento, el tamaño del mismo, y el valor que va a tomar; estos argumentos están definidos como los objetos de memoria.
2. Poner en cola la ejecución del kernel, para que este se ejecute en el dispositivo, mediante clEnqueueNDRangeKernel que necesita como parámetros la cola de comandos, el kernel, el número de dimensiones a utilizar, un offset que generalmente se declara nulo, el tamaño del grupo de trabajo global, el tamaño del grupo de trabajo local, cantidad de eventos a usar, y

un puntero donde se ubiquen los eventos.
3. Ahora se necesita leer la información generada por el dispositivo con clEnqueueReadBuffer que usa la cola de comandos, el objeto de memoria que se va a leer, un comando de bloqueo que detiene el proceso mientras lee, el tamaño del desfase para el objeto de memoria, el tamaño del dato que se va a leer, un puntero de la información en el host, cantidad de eventos a usar, y un puntero donde se ubiquen los eventos.
4. Al final la información quedará almacenada en variables locales del host que podrán ser utilizadas normalmente por comandos propios del lenguaje C, pero es necesario liberar todo el espacio utilizando como prefijo clRelese y completándolo con el tipo de elemento que se desea borrar, y como único parámetro el nombre asignado a las variables de este tipo.

Para terminar se adjunta el código para sumar dos vectores de forma concurrente, funcional y con ciertos comentarios y señales de error para facilitar el debug, además de los procesos básicos de programación que son necesarios igualmente para hacerlo funcional pero no detallados en este tutorial, como son la creación de los vectores y demás.

B. Anexo: Programa OpenCL

B.1. Ecuación de Calor

```
#include "CL/cl.h"
#include "CL/cl_ext.h"
#include "iostream"
#include <string>
#include <time.h>
#include <math.h>
#include <stdio.h>
#include <fstream>

#define __NO_STD_VECTOR // Usar cl::vector
#define __CL_ENABLE_EXCEPTIONS // necesario para excepciones

const float ini=0;
const float fin=0;

#define PROFILING // tiempo de kernel

using namespace std;

// Kernel
const char *KernelSource = "\n" \
"__kernel void hello_kernel(                                          \n" \
"   __global float* a,\n" \
" __global float* d, __global float* e )     \n" \
"{                                                                    \n" \
"   int in = get_global_id(0);                                         \n" \
"   float r = e[0]; \n" \
"   d[in+1] = r*a[in] + (1-(2*r))*a[in+1]+ r*a[in+2];  \n" \
" \n" \
"}                                                                    \n" \
"\n";
```

```
int main(int argc, char **argv)
{

int arg;
cout <<"ingrese ARRAY_SIZE: ";
cin >> arg;
int ARRAY_SIZE=arg;

int arg1;
cout <<"ingrese M: ";
cin >> arg1;
int M=arg1;

remove("C:\\Users\\MANUEL ALEJANDRO\\Desktop\\archivo.txt");

FILE * Alfa = fopen ("C:\\Users\\MANUEL ALEJANDRO\\Desktop\\archivo.txt", "a+t");


clock_t t_ini, t_fin;
double secs;

t_ini = clock(); //medici\'{o}n del tiempo

cl_platform_id firstPlatformId;
cl_int errNum;
cl_context context=NULL;

//obtener la lista de plataformas disponibles
errNum=clGetPlatformIDs(1, &firstPlatformId, NULL);

//definir las propiedades del contexto
cl_context_properties contextProperties[] ={
CL_CONTEXT_PLATFORM,
(cl_context_properties)firstPlatformId,
0
};

cl_device_id device_id;
cl_uint numDevices;
```

```
clGetDeviceIDs( firstPlatformId, CL_DEVICE_TYPE_GPU, 1, &device_id, &numDevices);



//creaci\'{o}n contexto para el primer dispositivo GPU disponible
//context= clCreateContextFromType(contextProperties,CL_DEVICE_TYPE_CPU,NULL,NULL,&errNum)

//cl_device_id *devices;
cl_command_queue commandQueue= NULL;


//obtener el tama\~{n}o del buffer del dispositivo
context= clCreateContext(contextProperties,1,&device_id,NULL,NULL,&errNum);

errNum = clGetContextInfo(context,CL_CONTEXT_DEVICES,0,NULL,NULL);

//Reservar memoria para el buffer del dispositivo
//devices = new cl_device_id[deviceBufferSize / sizeof(cl_device_id)];

// obtenemos el arreglo de dispositivos disponibles
//errNum = clGetContextInfo(context,CL_CONTEXT_DEVICES,deviceBufferSize,devices,NULL);

// crear cola de comandos para el primer dispositivo disponible

commandQueue = clCreateCommandQueue(context,device_id,CL_QUEUE_PROFILING_ENABLE,NULL);

cl_program program;

// Crear un objeto de programa del kernel
program= clCreateProgramWithSource(context,1,(const char **)&KernelSource,NULL,NULL);

////////////////////////////////////////////
//iniciaci\'{o}n de variables
const int N = ARRAY_SIZE-2;
float NN=N;
float dx = 1 / (NN+1);

//float  x[N+1];
float *x;
```

```
x=(float*)malloc((N+1) * sizeof(float));
for (int i=0; i< N+1; i++)
{
x[i]=i*dx;
}
float MM=M;
float dy= 1/(MM+1);

//float t[M+1];
float *t;
t=(float*)malloc((M+1) * sizeof(float));
for (int i=0; i< M+1; i++)
{
t[i]=i*dy;
}

float l= dy/(dx*dx);
 //l=0.4;

fprintf(Alfa,"%d %d\n",ARRAY_SIZE,M);


// compilar el kernel
errNum= clBuildProgram(program,0,NULL,NULL,NULL,NULL);

// crear instancia del kernel
cl_kernel kernel;
kernel = clCreateKernel(program,"hello_kernel",&errNum);

// crear los arreglos de entrada salida para el kernel en la CPU
float *a;
a=(float*)malloc((ARRAY_SIZE) * sizeof(float));
//float d[ARRAY_SIZE];
float *d;
d=(float*)malloc((ARRAY_SIZE) * sizeof(float));

float e[1];
e[0]=(float)l ;

for (int i=1; i< ARRAY_SIZE-1; i++)
```

```
{
if (i <= ((ARRAY_SIZE/2)-1))
a[i] = (float)2*x[i];
else
a[i] = (float)2*(1-x[i]);
}
////////////////////////////////////////////////////

a[0]=ini;
a[ARRAY_SIZE-1]=fin;

// crear los objetos de memoria en la GPU
cl_mem memObjects[3] = {0,0,0};
memObjects[0] = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
sizeof(float) * ARRAY_SIZE, a, NULL);


memObjects[1] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
sizeof(float) * ARRAY_SIZE, NULL, NULL);



memObjects[2] = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
sizeof(float)*2,e ,NULL);



// definir los argumentos del kernel
errNum = clSetKernelArg(kernel,0,sizeof(cl_mem), &memObjects[0]);

errNum = clSetKernelArg(kernel,1,sizeof(cl_mem), &memObjects[1]);

errNum = clSetKernelArg(kernel,2,sizeof(cl_mem), &memObjects[2]);



size_t global[1] = {ARRAY_SIZE};
size_t loc;

 errNum = clGetKernelWorkGroupInfo(kernel, device_id,
```

```
 CL_KERNEL_WORK_GROUP_SIZE,
 sizeof(loc), &loc, NULL);



size_t local[1]={1};

// poner en cola la ejecuci\'{o}n del kernel
errNum = clEnqueueNDRangeKernel(commandQueue,kernel,1,NULL,global,local,
0,NULL,NULL);

for (int j=0;j<ARRAY_SIZE; j++)
{
fprintf(Alfa,"%f",a[j]);
fprintf(Alfa," ");

}
fprintf(Alfa,"\n");

// copiar el buffer de salida del dispositivo y devolverlo a la CPU
errNum = clEnqueueReadBuffer(commandQueue, memObjects[1],CL_TRUE,0,
ARRAY_SIZE * sizeof(float), a ,0,NULL,NULL);

a[0]=0;
a[ARRAY_SIZE-1]=0;

for (int j=0;j<ARRAY_SIZE; j++)
{

fprintf(Alfa,"%f",a[j]);
fprintf(Alfa," ");
}
fprintf(Alfa,"\n");
////////////////////////////////////////////////////////////////

//repetir el proceso seg\'{u}n n\'{u}mero de iteraciones
for (int k=0 ; k < M ; k++)
{

errNum = clEnqueueWriteBuffer(commandQueue, memObjects[0],CL_TRUE,0,
```

```
ARRAY_SIZE * sizeof(float), a ,0,NULL,NULL);

errNum = clEnqueueNDRangeKernel(commandQueue,kernel,1,NULL,global,local,
0,NULL,NULL);



// copiar el buffer de salida del dispositivo y devolverlo a la CPU
errNum = clEnqueueReadBuffer(commandQueue, memObjects[1],CL_TRUE,0,
ARRAY_SIZE * sizeof(float), a ,0,NULL,NULL);

a[0]=0;
a[ARRAY_SIZE-1]=0;

for (int j=0;j<ARRAY_SIZE; j++)
{
fprintf(Alfa,"%f ",a[j]);
}
fprintf(Alfa,"\n");

}

//medir el tiempo final
t_fin = clock();

  secs = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
  printf("\n%.16g milisegundos\n", secs * 1000.0);

fprintf(Alfa,"\n%.16g milisegundos\n", secs * 1000.0);
   cin >> errNum;
   fclose (Alfa);
   /* Liberar memoria */
   clReleaseKernel(kernel);
   clReleaseMemObject(memObjects[0]);
   clReleaseMemObject(memObjects[1]);
   clReleaseMemObject(memObjects[2]);
   clReleaseCommandQueue(commandQueue);
   clReleaseProgram(program);
   clReleaseContext(context);
return 0;
```

```
}
```

B.2. Ecuación de Laplace

```
\clearpage
#include "CL/cl.h"
#include "CL\cl_ext.h"
#include "iostream"
#include <string>
#include <time.h>
#include <math.h>
#include <stdio.h>

#define __NO_STD_VECTOR
#define __CL_ENABLE_EXCEPTIONS

#define PROFILING

const float ini=0;
const float fin=0;

using namespace std;

const char *KernelSource = "\n" \
"__kernel void hello_kernel(                                                \n" \
"   const int Mdim,const int Ndim,const int Pdim,\n" \
" __global float* A, __global float* C )     \n" \
"{                                                                          \n" \
" int i,j,k; \n" \
"   i = get_global_id(0);                  \n" \
"   C[i]=(A[i-1]+A[i+1]+A[i-Ndim]+A[i+Ndim])/4;  \n" \
" }                                                                          \n" \
"\n";


int main(int argc, char **argv){

cl_event event;

int Mdim;
cout <<"ingrese dimension de la matriz: ";
cin >> Mdim;
```

```
int M;
cout <<"ingrese numero de iteracions: ";
cin >> M;

remove("C:\\Users\\MANUEL ALEJANDRO\\Desktop\\archivo.txt");

FILE * Alfa = fopen ("C:\\Users\\MANUEL ALEJANDRO\\Desktop\\archivo.txt", "a+t");

clock_t t_ini, t_fin;
double secs;

cl_platform_id firstPlatformId;
cl_int errNum;
cl_context context=NULL;
int Ndim,Pdim;
Ndim=Mdim;

errNum=clGetPlatformIDs(1, &firstPlatformId, NULL);
if (errNum != CL_SUCCESS)
    {
        printf("Error: Failed to get a platform group!\n");
        return EXIT_FAILURE;
    }

cl_context_properties contextProperties[] ={
CL_CONTEXT_PLATFORM,
(cl_context_properties)firstPlatformId,
0
};

cl_device_id device_id;
cl_uint numDevices;
clGetDeviceIDs( firstPlatformId, CL_DEVICE_TYPE_GPU, 1, &device_id, &numDevices);


context= clCreateContext(contextProperties,1,&device_id,NULL,NULL,&errNum);

cl_command_queue commandQueue= NULL;
```

```
if (errNum != CL_SUCCESS)

    {
        printf("Error: Failed to get a context info!\n");
        return EXIT_FAILURE;
    }

if (errNum != CL_SUCCESS)
    {
        printf("Error: Failed to get a context info!\n");
        return EXIT_FAILURE;
    }

commandQueue = clCreateCommandQueue(context,device_id,CL_QUEUE_PROFILING_ENABLE,NULL);
cl_program program;


program= clCreateProgramWithSource(context,1,(const char **)&KernelSource,NULL,&errNum);
if (errNum != CL_SUCCESS)
    {
        printf("Error: Failed to get a program info!\n");
        return EXIT_FAILURE;
    }

errNum= clBuildProgram(program,0,NULL,NULL,NULL,NULL);

cl_kernel kernel;
kernel = clCreateKernel(program,"hello_kernel",&errNum);
if (!kernel || errNum != CL_SUCCESS)
    {
        printf("Error: Failed to create compute kernel!\n");
        exit(1);
    }


t_ini = clock();

float *A;
A=(float*)calloc((Ndim*Mdim),sizeof(float));
```

```
A[0]=0;
for (int i=1;i<Ndim;i++)
{
A[i]=0;
A[Ndim*i]=75;
A[Ndim*i+(Mdim-1)]=50;
A[(Mdim-1)*Ndim+i]=100;

}

cl_mem memObjects[2] = {0,0};
memObjects[0] = clCreateBuffer(context,CL_MEM_READ_ONLY| CL_MEM_COPY_HOST_PTR,
sizeof(float)*Ndim*Mdim, A, NULL);


memObjects[1] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
sizeof(float)*Ndim*Mdim, NULL ,NULL);



errNum = clSetKernelArg(kernel,0,sizeof(int), &Mdim);
if (errNum != CL_SUCCESS)
    {
        printf("Error: Failed to set arguments in kernel1!\n");
        return EXIT_FAILURE;
    }
errNum = clSetKernelArg(kernel,1,sizeof(int), &Ndim);
if (errNum != CL_SUCCESS)
    {
        printf("Error: Failed to set arguments in kernel2!\n");
        return EXIT_FAILURE;
    }
errNum = clSetKernelArg(kernel,2,sizeof(int), &Pdim);

if (errNum != CL_SUCCESS)
    {
        printf("Error: Failed to set arguments in kernel3!\n");
        return EXIT_FAILURE;
    }
```

```
errNum = clSetKernelArg(kernel,3,sizeof(cl_mem), &memObjects[0]);

if (errNum != CL_SUCCESS)
    {
        printf("Error: Failed to set arguments in kernel4!\n");
        return EXIT_FAILURE;
    }

errNum = clSetKernelArg(kernel,4,sizeof(cl_mem), &memObjects[1]);

if (errNum != CL_SUCCESS)
    {
        printf("Error: Failed to set arguments in kernel6!\n");
        return EXIT_FAILURE;
    }

size_t global[1] = {(Ndim*Mdim)};
size_t local[1] = {1};


for (int y=0;y<M; y++)
{

errNum = clEnqueueNDRangeKernel(commandQueue,kernel,1,NULL,global,NULL,
0,NULL,&event);
//errNum = clEnqueueTask(commandQueue,kernel,0,NULL,NULL);
if (errNum == CL_INVALID_WORK_GROUP_SIZE)
{
printf("Error: work dim!\n");

}


if (errNum != CL_SUCCESS)
{
printf("Error: Failed to enqueue kernel!\n");
return EXIT_FAILURE;
}
```

```
errNum = clEnqueueReadBuffer(commandQueue, memObjects[1],CL_TRUE,0,
Ndim*Mdim*sizeof(float), A ,0,NULL,NULL);
A[0]=0;



for (int i=1;i<Ndim;i++)
{
A[i]=0;
A[Ndim*i]=75;
A[Ndim*i+(Mdim-1)]=50;
A[(Mdim-1)*Ndim+i]=100;

}

for (int i=0;i<Ndim; i++)
{
for (int j=0;j<Mdim; j++)
{
fprintf(Alfa,"%f ",A[i*Ndim+j]);

}
fprintf(Alfa,"\n");
}
fprintf(Alfa,"\n");


errNum = clEnqueueWriteBuffer(commandQueue, memObjects[0],CL_TRUE,0,
Ndim*Mdim*sizeof(float), A ,0,NULL,NULL);

if (errNum != CL_SUCCESS)
{
printf("Error: Failed to read buffer!\n");
return EXIT_FAILURE;
}
```

```
}


printf("\n\n ");
#ifdef PROFILING
cl_ulong start,end;
clGetEventProfilingInfo(event,CL_PROFILING_COMMAND_START,sizeof(cl_ulong),&start,NULL);
clGetEventProfilingInfo(event,CL_PROFILING_COMMAND_END,sizeof(cl_ulong),&end,NULL);
double time = 1.e-9 * (end-start);
cout << "Time for kernel to execute " << time << endl;
#endif


t_fin = clock();
secs = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
printf("\n%.16g milisegundos\n", secs * 1000.0);

clReleaseKernel(kernel);
clReleaseMemObject(memObjects[0]);
clReleaseMemObject(memObjects[1]);

clReleaseCommandQueue(commandQueue);
clReleaseProgram(program);
clReleaseContext(context);
cin>>errNum;
free (A);


return 0;


}
```

B.2.1. Cambios para indexar

```
//Kernel
const char *KernelSource1 = "\n" \
```

```
"__kernel void hello_kernel(                                                \n" \
"   const int Mdim,const int Ndim,const int Pdim,\n" \
" __global float* A, __global float* C )     \n" \
"{                                                                          \n" \
" int i,j,k; \n" \
"   i = get_global_id(0);                  \n" \
"   j = get_global_id(1); \n" \

"C[i*Ndim+j]=(A[((i-1)*Ndim)+j]+A[((i+1)*Ndim)+j]
+A[(i*Ndim)+(j-1)]+A[(i*Ndim)+(j+1)])/4;  \n" \
" //}; \n" \
" }                                                                          \n" \
"\n";

// Tama\~{n}o de las dimensiones

size_t global[2]= {(size_t)Ndim,(size_t)Mdim};
size_t local[1]={1};

// Indice de dimensiones

errNum = clEnqueueNDRangeKernel(commandQueue,kernel,2,NULL,global,NULL,
0,NULL,&event);
```

B.3. Ecuación de Onda

B.3.1. Una dimensión

```
#include "CL/cl.h"
#include "CL/cl_ext.h"
#include "iostream"
#include <string>
#include <time.h>
#include <math.h>
#include <stdio.h>
#include <fstream>
#include <dos.h>
#include <conio.h>
#include <windows.h>

#define PI 3.14159265
#define __NO_STD_VECTOR
#define __CL_ENABLE_EXCEPTIONS

#define PROFILING

using namespace std;

const char *KernelSource = "\n" \
"__kernel void hello_kernel(                                                 \n" \
"   __global float* a,__global float* b,\n" \
" __global float* d, __global float* e )     \n" \
"{                                                                           \n" \
"   int in = get_global_id(0);                                               \n" \
"   float r = e[0]; \n" \

"      \n" \
" d[in+1] = (r*r)*(b[in]+b[in+2]) + 2*(1-(r*r))*b[in+1] - a[in+1]; \n" \
" a[in+1] = b[in+1];                                                         \n" \
"}                                                                           \n" \
"\n";
```

```
int main(int argc, char **argv){



int arg;
cout <<"ingrese ARRAY_SIZE: ";
cin >> arg;
int ARRAY_SIZE=arg;

int arg1;
cout <<"ingrese M: ";
cin >> arg1;
int M=arg1;

remove("C:\\Users\\MANUEL ALEJANDRO\\Desktop\\archivo.txt");

FILE * Alfa = fopen ("C:\\Users\\MANUEL ALEJANDRO\\Desktop\\archivo.txt", "a+t");


clock_t t_ini, t_fin;
double secs;

t_ini = clock();

cl_platform_id firstPlatformId;
cl_int errNum;
cl_context context=NULL;

errNum=clGetPlatformIDs(1, &firstPlatformId, NULL);

cl_context_properties contextProperties[] ={
CL_CONTEXT_PLATFORM,
(cl_context_properties)firstPlatformId,
0
};

cl_device_id device_id;
cl_uint numDevices;

clGetDeviceIDs( firstPlatformId, CL_DEVICE_TYPE_GPU, 1, &device_id, &numDevices);
```

```
cl_command_queue commandQueue= NULL;

context= clCreateContext(contextProperties,1,&device_id,NULL,NULL,&errNum);



errNum = clGetContextInfo(context,CL_CONTEXT_DEVICES,0,NULL,NULL);


commandQueue = clCreateCommandQueue(context,device_id,CL_QUEUE_PROFILING_ENABLE,NULL);



cl_program program;

program= clCreateProgramWithSource(context,1,(const char **)&KernelSource,NULL,NULL);

t_ini = clock();

const int N = ARRAY_SIZE-2;
float NN=N;
float dx = 1 / (NN+1);


float *x;
x=(float*)malloc((N+1) * sizeof(float));
for (int i=0; i< N+1; i++)
{
x[i]=i*dx;
}
float MM=M;
float dy= 1/(MM+1);

float *t;
t=(float*)malloc((M+1) * sizeof(float));
for (int i=0; i< M+1; i++)
{
t[i]=i*dy;
}
```

```
float l= 2*dy/dx;


 l=1;

fprintf(Alfa,"%d %d %f %f %f\n",ARRAY_SIZE,M,dx,dy,l);

errNum= clBuildProgram(program,0,NULL,NULL,NULL,NULL);

cl_kernel kernel;
kernel = clCreateKernel(program,"hello_kernel",&errNum);

float *a;
a=(float*)malloc((ARRAY_SIZE) * sizeof(float));
float *b;
b=(float*)malloc((ARRAY_SIZE) * sizeof(float));

float *d;
d=(float*)malloc((ARRAY_SIZE) * sizeof(float));

float e[1];
e[0]=(float)l ;



for (int i=1; i< ARRAY_SIZE-1; i++)
{
a[i]= sin(PI*x[i]);

}

a[0]=ini;
a[ARRAY_SIZE-1]=fin;

for (int i=1; i< ARRAY_SIZE-1; i++)
{
b[i]=(l*l)*((a[i-1]+a[i+1])/2)+(1-(l*l))*a[i];// +K*gi
}
```

```
b[0]=ini;
b[ARRAY_SIZE-1]=fin;

cl_mem memObjects[4] = {0,0,0,0};
memObjects[0] = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
sizeof(float) * ARRAY_SIZE, a, NULL);

memObjects[1] = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
sizeof(float) * ARRAY_SIZE, b, NULL);

memObjects[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
sizeof(float) * ARRAY_SIZE, NULL, NULL);



memObjects[3] = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
sizeof(float)*2,e ,NULL);


errNum = clSetKernelArg(kernel,0,sizeof(cl_mem), &memObjects[0]);

errNum = clSetKernelArg(kernel,1,sizeof(cl_mem), &memObjects[1]);

errNum = clSetKernelArg(kernel,2,sizeof(cl_mem), &memObjects[2]);

errNum = clSetKernelArg(kernel,3,sizeof(cl_mem), &memObjects[3]);



size_t global[1] = {ARRAY_SIZE};
size_t loc;

 errNum = clGetKernelWorkGroupInfo(kernel, device_id,
 CL_KERNEL_WORK_GROUP_SIZE,
 sizeof(loc), &loc, NULL);



  size_t local[1]={1};
```

```
errNum = clEnqueueNDRangeKernel(commandQueue,kernel,1,NULL,global,local,
0,NULL,NULL);

for (int j=0;j<ARRAY_SIZE; j++)
{
fprintf(Alfa,"%f",a[j]);
fprintf(Alfa," ");

}
fprintf(Alfa,"\n");


for (int j=0;j<ARRAY_SIZE; j++)
{
fprintf(Alfa,"%f",b[j]);
fprintf(Alfa," ");

}
fprintf(Alfa,"\n");

for (int i=1; i< ARRAY_SIZE-1; i++)
{
a[i]=b[i];

}

errNum = clEnqueueReadBuffer(commandQueue, memObjects[2],CL_TRUE,0,
ARRAY_SIZE * sizeof(float), b ,0,NULL,NULL);




b[0]=ini;
b[ARRAY_SIZE-1]=fin;

for (int j=0;j<ARRAY_SIZE; j++)
{

fprintf(Alfa,"%f",b[j]);
```

```
fprintf(Alfa," ");
}
fprintf(Alfa,"\n");



for (int k=0 ; k < M-1 ; k++)
{


errNum = clEnqueueWriteBuffer(commandQueue, memObjects[1],CL_TRUE,0,
ARRAY_SIZE * sizeof(float), b ,0,NULL,NULL);

errNum = clEnqueueNDRangeKernel(commandQueue,kernel,1,NULL,global,local,
0,NULL,NULL);

errNum = clEnqueueReadBuffer(commandQueue, memObjects[2],CL_TRUE,0,
ARRAY_SIZE * sizeof(float), b ,0,NULL,NULL);




b[0]=ini;
b[ARRAY_SIZE-1]=fin;

for (int j=0;j<ARRAY_SIZE; j++)
{

fprintf(Alfa,"%f ",b[j]);

}
fprintf(Alfa,"\n");

}

t_fin = clock();

  secs = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
  printf("\n%.16g milisegundos\n", secs * 1000.0);
```

```
fprintf(Alfa,"\n%.16g milisegundos\n", secs * 1000.0);

   fclose (Alfa);

   clReleaseKernel(kernel);
   clReleaseMemObject(memObjects[0]);
   clReleaseMemObject(memObjects[1]);
   clReleaseMemObject(memObjects[2]);
   clReleaseCommandQueue(commandQueue);
   clReleaseProgram(program);
   clReleaseContext(context);


return 0;

}
```

B.3.2. Dos dimensiones

```
#include "CL/cl.h"
#include "CL\cl_ext.h"
#include "iostream"
#include <string>
#include <time.h>
#include <math.h>
#include <stdio.h>

#define __NO_STD_VECTOR
#define __CL_ENABLE_EXCEPTIONS

#define PROFILING

const float ini=0;
const float fin=0;

using namespace std;

const char *KernelSource = "\n" \
"__kernel void hello_kernel(                                                  \n" \
"   const int Ndim, __global float* e,\n" \
" __global float* A, __global float* B, __global float* C )     \n" \
"{                                                                            \n" \
" int i,j; \n" \
"   i = get_global_id(0);                 \n" \
"   j = get_global_id(1); \n" \
" float kM = e[0]; \n" \
" float kN = kM; \n" \

"   C[i*Ndim+j]=((kM*kM)*(B[i*Ndim+(j+1)] - (2*B[i*Ndim+j]) + B[i*Ndim+(j-1)]) +
 (kN*kN)*(B[(i+1)*Ndim+j] - (2*B[i*Ndim+j]) + B[(i-1)*Ndim+j])
  + (2*B[i*Ndim+j]) - A[i*Ndim+j]);  \n" \

" A[i]=B[i]; \n" \

" }                                                                            \n" \
"\n";
```

```
int main(int argc, char **argv){

cl_event event;

int Mdim;
cout <<"ingrese dimension de la matriz: ";
cin >> Mdim;

int M;
cout <<"ingrese numero de iteracions: ";
cin >> M;

remove("C:\\Users\\MANUEL ALEJANDRO\\Desktop\\archivo.txt");

FILE * Alfa = fopen ("C:\\Users\\MANUEL ALEJANDRO\\Desktop\\archivo.txt", "a+t");

clock_t t_ini, t_fin;

double secs;

cl_platform_id firstPlatformId;
cl_int errNum;
cl_context context=NULL;
int Ndim,Pdim;
Ndim=Mdim;

errNum=clGetPlatformIDs(1, &firstPlatformId, NULL);


cl_context_properties contextProperties[] ={
CL_CONTEXT_PLATFORM,
(cl_context_properties)firstPlatformId,
0
};

cl_device_id device_id;
cl_uint numDevices;
clGetDeviceIDs( firstPlatformId, CL_DEVICE_TYPE_GPU, 1, &device_id, &numDevices);

context= clCreateContext(contextProperties,1,&device_id,NULL,NULL,&errNum);
```

```
cl_command_queue commandQueue= NULL;

commandQueue = clCreateCommandQueue(context,device_id,CL_QUEUE_PROFILING_ENABLE,NULL);

cl_program program;

program= clCreateProgramWithSource(context,1,(const char **)&KernelSource,NULL,&errNum);

errNum= clBuildProgram(program,0,NULL,NULL,NULL,NULL);

cl_kernel kernel;
kernel = clCreateKernel(program,"hello_kernel",&errNum);


t_ini = clock();
const int N = Mdim-2;
float NN=N;
float dx = 1 / (NN+1);


float *x;
x=(float*)malloc((N+1) * sizeof(float));
for (int i=0; i< N+1; i++)
{
x[i]=i*dx;
}
float MM=M;
float dy= 1/(MM+1);


float *t;
t=(float*)malloc((M+1) * sizeof(float));
for (int i=0; i< M+1; i++)
{
t[i]=i*dy;
}


float l= 0.4;
```

```
float *A;
A=(float*)calloc((Ndim*Mdim),sizeof(float));

A[0]=0;
A[Ndim*Mdim/2]=1;
for (int i=1;i<Ndim;i++)
{
A[i]=0;
A[Ndim*i]=0;
A[Ndim*i+(Mdim-1)]=0;
A[(Mdim-1)*Ndim+i]=0;

}

float *B;
B=(float*)calloc((Ndim*Mdim),sizeof(float));

for (int i=1;i<Ndim-1;i++)
for (int j=1;j<Ndim-1;j++)
{
B[i*Ndim+j]=(l*l)*((A[i*Ndim+j-1]+A[i*Ndim+j+1] +A[(i-1)*Ndim+j]+A[(i+1)*Ndim+j])/4)
+(1-(l*l))*A[i*Ndim+j];// +K*gi
}

for (int i=1;i<Ndim;i++)
{
B[i]=0;
B[Ndim*i]=0;
B[Ndim*i+(Mdim-1)]=0;
B[(Mdim-1)*Ndim+i]=0;

}


float e[1];
e[0]=(float)l ;


cl_mem memObjects[4] = {0,0,0,0};
```

```
memObjects[0] = clCreateBuffer(context,CL_MEM_READ_ONLY| CL_MEM_COPY_HOST_PTR,
sizeof(float)*2, e, NULL);

memObjects[1] = clCreateBuffer(context,CL_MEM_READ_ONLY| CL_MEM_COPY_HOST_PTR,
sizeof(float)*Ndim*Mdim, A, NULL);

memObjects[2] = clCreateBuffer(context,CL_MEM_READ_ONLY| CL_MEM_COPY_HOST_PTR,
sizeof(float)*Ndim*Mdim, B, NULL);


memObjects[3] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
sizeof(float)*Ndim*Mdim, NULL ,NULL);


 clSetKernelArg(kernel,0,sizeof(int), &Mdim);


 clSetKernelArg(kernel,1,sizeof(cl_mem), &memObjects[0]);


 clSetKernelArg(kernel,2,sizeof(cl_mem), &memObjects[1]);


 clSetKernelArg(kernel,3,sizeof(cl_mem), &memObjects[2]);


 clSetKernelArg(kernel,4,sizeof(cl_mem), &memObjects[3]);



size_t globalWorkSize[1] = {(Ndim*Mdim)};
size_t localWorkSize[1] = {1};

size_t global[2]= {(size_t)Ndim,(size_t)Mdim};
size_t local[1]={1};


for (int i=0;i<Ndim; i++)
{
```

```
for (int j=0;j<Mdim; j++)
{
fprintf(Alfa,"%f ",B[i*Ndim+j]);

}
fprintf(Alfa,"\n");
}
fprintf(Alfa,"\n");

t_ini = clock();
for (int y=0;y<M; y++)
{


 clEnqueueNDRangeKernel(commandQueue,kernel,2,NULL,global,NULL,
0,NULL,&event);

 clEnqueueReadBuffer(commandQueue, memObjects[3],CL_TRUE,0,
Ndim*Mdim*sizeof(float), B ,0,NULL,NULL);
B[0]=0;



for (int i=1;i<Ndim;i++)
{
B[i]=0;
B[Ndim*i]=0;
B[Ndim*i+(Mdim-1)]=0;
B[(Mdim-1)*Ndim+i]=0;

}
 clEnqueueWriteBuffer(commandQueue, memObjects[1],CL_TRUE,0,
Ndim*Mdim*sizeof(float), A ,0,NULL,NULL);

clEnqueueWriteBuffer(commandQueue, memObjects[2],CL_TRUE,0,
Ndim*Mdim*sizeof(float), B ,0,NULL,NULL);

}
```

```
t_fin = clock();

#ifdef PROFILING
cl_ulong start,end;
clGetEventProfilingInfo(event,CL_PROFILING_COMMAND_START,sizeof(cl_ulong),&start,NULL);
clGetEventProfilingInfo(event,CL_PROFILING_COMMAND_END,sizeof(cl_ulong),&end,NULL);
    double time = 1.e-9 * (end-start);
    cout << "Time for kernel to execute " << time << endl;
#endif

 secs = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
  printf("\n%.16g milisegundos\n", secs * 1000.0);
   clReleaseKernel(kernel);
   clReleaseMemObject(memObjects[0]);
   clReleaseMemObject(memObjects[1]);
      clReleaseMemObject(memObjects[2]);
   clReleaseMemObject(memObjects[3]);
   clReleaseCommandQueue(commandQueue);
   clReleaseProgram(program);
   clReleaseContext(context);
cin>>errNum;
free (A);
free (B);

return 0;


}
```