|  |  |  |
| --- | --- | --- |
| escudo unam | **UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO**  INSTITUTO DE INVESTIGACIONES EN MÁTEMATICAS  APLICADAS Y EN SISTEMAS | |
|  |  | |
|  | POSGRADO EN CIENCIA E INGENIERIA  DE LA COMPUTACIÓN | |
| **DIAGONALIZACIÓN DE LA MATRIZ DE KHON-SHAM CON TARJETAS GRÁFICAS** | |
|  | **TRABAJO FINAL** |  |
|  | QUE PARA OBTENER EL GRADO DE: |  |
|  | **ESPECIALISTA EN CÓMPUTO DE**  **ALTO RENDIMIENTO** |  |
|  |  |  |
|  | PRESENTA  **JOSÉ ANTONIO AYALA BARBOSA**  TUTOR  **DR. JOSÉ JESÚS CARLOS QUINTANAR SIERRA** |  |
|  |  | **Ciudad Universitaria, CDMX 2018** |  |

**Agradecimientos**

Índice

Índice 5

Introducción 7

Capítulo 1. Matriz de Khon-Sham 9

1.1 Ecuación de Schrödinger 9

1.2 Teoría del Funcional de la Densidad (DFT) 9

1.2.1 Ecuación de Kohn-Sham 10

1.3 Modelado molecular en GPU 12

Capítulo 2. Algoritmo de Jacobi 14

2.1 Forma cuadrática 14

2.2 Eigensistemas 14

2.2.1 Transformación de coordenadas 15

2.2.2 Eigenvalores 16

2.2.3 Eigenvector 17

Capítulo 3. Implementación en C, OpenACC y CUDA 18

3.1 Cómputo en GPU 18

3.1.1 Diferencia entre CPU y GPU 18

3.2 Caracteristicas del equipo 19

3.3 Implementación en C 20

3.3.1 Creación de matrices 20

3.3.2 Funciones auxiliares 20

3.3.3 Driver 21

3.3.4 Jacobi 22

3.3.5 Compilación 23

3.4 Implementación en OpenACC 24

3.4.1 Estándar OpenACC 24

3.4.2 JacobiACC 26

3.4.3 Compilación 27

3.5 Implementación en CUDA 28

3.5.1 Modelo de programación en CUDA 28

3.5.2 Driver 35

3.5.3 JacobiCUDA 35

3.5.4 Compilación 36

Capítulo 4. Pruebas y resultados 38

4.1 Métricas de desempeño 38

4.1.2 Runtime 38

4.1.2 Cost Factor 38

4.1.3 Speedup Factor 38

4.1.3 Aceleración 39

2.1.4 Efficiency 39

4.2 Resultados 40

4.2.1 Resultados de Runtime 40

4.2.2 Resultados de Cost Factor 41

4.2.3 Resultados de Speed Up 43

4.2.3 Resultados de Aceleración 44

4.2.4 Resultados de Efficiency 46

4.3 Resultados de una matriz 47

4.3.1 Matriz a evaluar 47

4.3.2 Resultado Serial 48

4.3.3 Resultado OpenACC 49

4.3.4 Resultado CUDA 50

Conclusiones 51

Tabla de Figuras 53

Bibliografía 54

CÓDIGO FUENTE 56

CF 1.1 creaMatrix.c 56

CF 1.2 auxFuncs.h 57

CF 1.3 driverA.c 58

CF 1.4 jacobiA.h 59

CF 1.5 driverA.cu 63

CF 1.6 jacobiACUDA.h 65

Introducción

Fisica y quimica computacional tratan de resolver problemas implementando métodos del análisis numérico para resolver con precisión problemas para los cuales ya existe una teoría cuantitativa [1]. Inicialmente se consideró como una subdisciplina de la física teórica, y como una herramienta que auxilia a la ciencia. Pero en la actualidad ya es considerada como una area de la fisica teorica. Por ejemplo en el Institute for Theoretical Physicas ETH Zurich [2] es una de las areas de investigacion, al lado de materia condensada, teoría cuántica del campo, partículas elementales y teoría de la información entre otros.

Tanto en la física como en la química computacional es necesario, entre otras muchas cosas, resolver sistemas lineales de ecuaciones, integrar, por ejemplo, las ecuaciones diferenciales de Newton en física clásica [3], o resolver la ecuación de Schrödinger en mecánica cuántica etc.; y para esto se pueden aplicar algoritmos como Runge Kuta, Monte-Carlo, y también en la mecánica cuántica un problema a resolver son las matrices Hermitianas de valores y vectores propios, (eigenmatrices, eigenvalores y eigenvectores) y uno de los algoritmos de elección es el método de diagonalización de Jacobi.

El problema de Eigensistemas aparece en uno de los métodos empleados para resolver la ecuación de Schrödinger o la ecuación de Khon-Sham. En la ecuación de Schrödinger su solución es conocida como la función de onda, mientras que las solución de la ecuación de Khon-Sham es el valor absoluto al cuadrado de la función de onda y se interpreta en la física como la densidad electrónica.

Durante las ultimas décadas ha incrementado el interés en la computación paralela, el principal objetivo de ella es ahorro de la energia y evitar el sobrecalientamiento de los circuitos. Desde una perspectiva de puros cálculos, la computación paralela puede definirse en esos muchos cálculos que se resuelven simultáneamente, y como atacan problemas muy largos y los dividen en unos mucho más pequeños que pueden ser resueltos paralela y concurrentemente [4].

Actualmente el paralelismo, por razones de ahorro de energias y por desempeño se ha convertido en la corriente que todo el mundo de la programación esta siguiendo. Existen dos principales tipos de paralelismo en aplicaciones la paralelización de tareas y la paralelización de datos. La primera surge cuando hay muchas tareas que pueden operar independientemente en paralelo, por ello el paralelismo de tareas se enfoca en la distribución de las funciones en los distintos núcleos de procesamiento. La segunda entra cunado existen muchos datos que pueden ser operados al mismo tiempo, siendo mejor distribirlos entre los diversos procesadores.

La unidad de procesamiento de gráficos (GPU) es uno de los componentes más importantes en los ordenadores modernos [5]. Es aquí donde se construyen los increíbles gráficos que podemos ver en los videojuegos modernos.

Dentro de la computación, CUDA es una plataforma de cómputo paralelo de uso general y un modelo de programación que aprovecha el motor de cómputo paralelo en las GPU de la marca NVIDIA para resolver muchos problemas computacionales complejos de una manera más eficiente. Usando CUDA, puede acceder a las tarjetas de video para el cálculo como se ha hecho tradicionalmente en la CPU.

Es inevitable no mencionar que CUDA al ofrecer un entorno de programación de bajo nivel, ha mantenido un poco alejado a los programadores que están fuera del desarrollo del cómputo científico, por ello se creó el estándar de programación OpenACC, que con solo algunas directivas se le indica al compilador que cierta sección de código podría ser paralelizable, y el este se encargará de verificar si existen dependencias de dato o si es posible realizarse.

Varias de esas aplicaciones están relacionadas con la solución de las ecuaciones de Kohn-Sham, dentro de la teoría de funcionales de la densidad, también se han creado diversas aplicaciones relacionadas con métodos basados en la función de onda que estiman la correlación electrónica o energías de ionización [6].

En marzo de 2014, el Departamento de Energía de Estados Unidos (DOE), otorgó a IBM la comisión para construir dos supercomputadoras. Sierra localizada en el Laboratorio Nacional Lawrence Livermore y Summit localizada en el Laboratorio Nacional Oak Ridge. En En el caso de Summit se apostó por equiparla con nodos de alto rendimiento y tarjetas gráficas, lo cual abrió el camino a utilizar está tecnología de forma masificada [7]. Summit representa un cambio importante con respecto a la forma en que se viene realizando el mundo del supercómputo, ya que, al combinar CPUs de alto rendimiento con GPUs de NVIDIA optimizadas para la AI, se puede ver que la computación se va decantando cada vez más por el cómputo heterogéneo.

La potencia de las GPUs se ha hecho presente en sistemas computacionales relacionados con el modelado de átomos en moléculas [8]. Debido a ésto, en este trabajo se paralelizará el método de Jacobi en tarjetas gráficas y se realizará una comparación entre su versión Secuencial realizada en lenguaje C, contra una versión paralela utilizando OpenACC y otra versión utilizando el modelo de CUDA.

Capítulo 1. Matriz de Khon-Sham

1.1 Ecuación de Schrödinger

A diferencia de la mecánica clásica que determina los elementos por su posición y velocidad, la mecánica cuántica calcula la probabilidad de encontrar partículas en una cierta posición física gracias a la ecuación de Schrödinger [9]. Pero para poder ser implementada se requiere de mucho poder de cómputo, y al ser calculada con métodos numéricos, la precisón depende de cuál método se elija. Por esta razón los resultados que se arrojan tendrán asociado un error.

La ecuación de Schrödinger para predecir el comportamiento de átomos y móleculas considera varios aspectos, los mas importantes son:

* La existencia de un núcleo atómico, en donde se concentra la mayoría de la masa del átomo.
* Los niveles energéticos donde se distribuyen los electrones según el principio de Aufbau.
* La interaccion electrón electrón.
* La interacción nucleo electrón.
* La dualidad onda-partícula.

Aunque con la mecánica cuántica queda claro que no se puede saber con certeza dónde se encuentra un electrón, la ecuacion de Shrödinger nos da la probabilidad de encontrarlo en un elemento de volumen permitido por el principio de incertidumbre de Heisenberg. Cada valor absoluto al cuadrado de la solución de la ecuación de Schrödinger Ψ(**r**), nos dá la probabilidad de encontra el electrón en una elemento de volumen centrado en **r**. El valor absoluto al cuadrado de la función de onda, |Ψ(**r**)|2, define la distribución de densidad electrónica alrededor del núcleo. Este concepto de densidad electrónica da la probabilidad de encontrar un electrón en una cierta región del átomo, llamada orbital atómico, concepto análogo al de órbita en el modelo de Bohr [10].

1.2 Teoría del Funcional de la Densidad (DFT)

Es un procedimiento alternativo a la ecuación de Schrödinger que permite una descripción muy exacta de los sistemas con muchas partículas, la DFT simplifica los cálculos, dejando a un lado el uso de la función de onda en la determinación del movimiento de electrones y átomos de una molécula. En el caso de la ecuación de Schrödinger esta solución se conoce como la función de onda , mientras que la solución de la ecuación Kohn-Sham [11] está dada por el valor absoluto al cuadrado de la función de la función de onda y esta cantidad se interpreta como la densidad electrónica . Una forma de obtener una solución de ambas ecuaciones es aplicando el método variacional, éste método se usa para determinar de una manera aproximada el nivel de energía más bajo del sistema; también conocida como la energía del estado base; éste método consiste en proponer una función tentativa que depende de varios parámetros los cuales se varían hasta que se obtenga una energía mínima.

La Teoria del funcional de la densidad obtiene la energía y distribución electrónica del estado fundamental [12], trabajando con la densidad electrónica en vez de la función de ondas. Una desventaja es que, salvo los casos más simples, no se conoce de manera exacta el funcional que relaciona esta densidad electrónica con la energía del sistema. En la práctica, se usan funcionales que se han comprobado que dan buenos resultados [13].

La primera aproximación que se hace al estudiar sistemas con más de un átomo es la aproximación de Born-Oppenheimer, en la cual los electrones en una molécula se mueven en un campo eléctrico producido por los núcleos fijos. Esta aproximación puede hacerse porque al ser la masa del núcleo mayor a la de los electrones, su velocidad es menor, por lo que para él, los electrones fungen como una nube de carga [14], y por otro lado para los electores, los núcleos parecen estar estáticos.

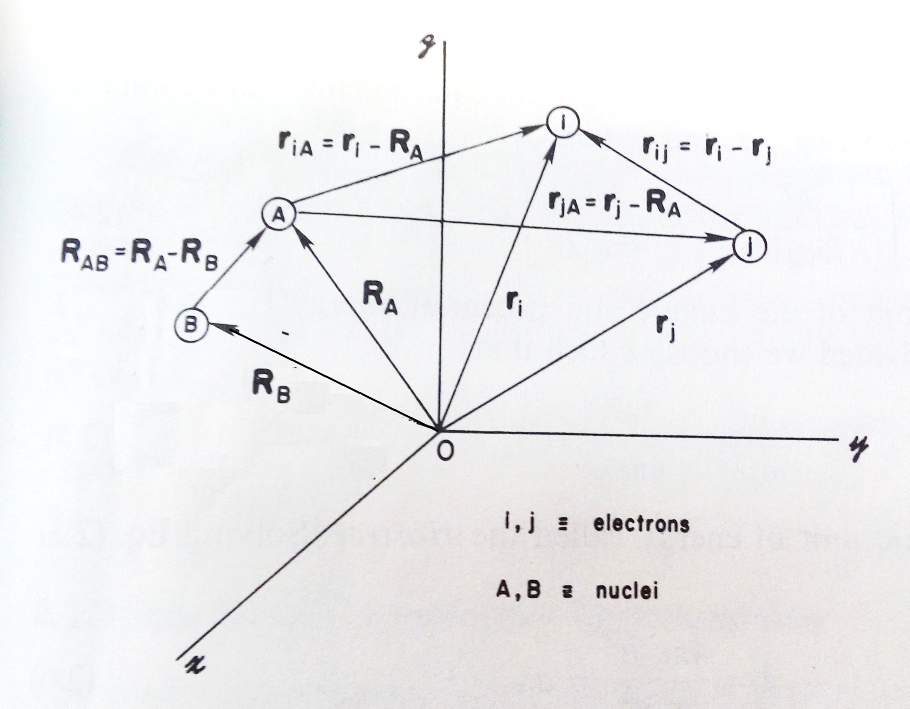


Figura 1 Sistema coordenado molecular: i,j=electrones y A, B= núcleos.

1.2.1 Ecuación de Kohn-Sham

La ventaja de que la densidad es una magnitud mucho más simple que la función de onda, simplificando las ecuaciones y bajando el costo computacional [15]. Por ello hasta el momento es el procedimiento preferido para abordar problemas a partir de una cierta complejidad.

La energía del sistema está dada por la suma de la energía:

Más propiamente dicho:

Al considerar que los núcleos son fijos, el termino de la energía cinética de los núcleos es eliminada y el termino de interacción de núcleo - núcleo se vuelve constante, y asi podemos cambiar el origen del eje y volverlo 0 para simplificar la ecuación [13].

Resultando el Hamiltoniano como:

El método de Hartree-Fock es una de las primeras implementaciones computacionales para resolver la ecuación de Schrödinger. La teoría del funcional de la densidad se desarrollo para resolver las ecuaciones de Khon-Sham. En ambos métodos se diagonalizan matrices Hermitianas.

Comparando la ecuación de Schrödinger con el DFT, puede observarse que son muy similares [12]:

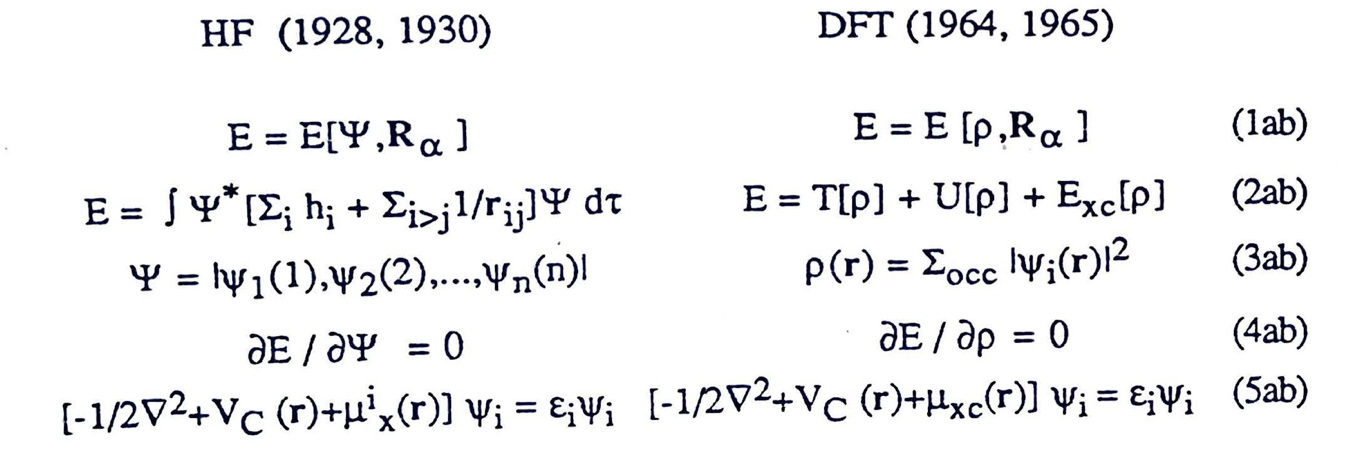


Figura 2 Comparación entre Ecuación de Schrödinger y Ecuación de Khon-Sham.

Donde para la ecuación de Schrödinger la energía se calcula en función de la función de onda para saber la posición de los electrones. Es necesario utilizar la sumatoria de las Hermitianas más el inverso de las posiciones en ij, por la función de onda. Y la función de onda se calcula con el valor absoluto de la sumatoria cada átomo. Dándonos así el principio variacional = 0. Con ello nos da como resultante la suma de la energía cinética más el potencial colombiano más el potencial de intercambio y correlación [12].

Por el lado del DFT, la energía esta en función de la densidad y la posición de los electrones. Y la energía se calcula con la energía cinética en función de la densidad [16], más la energía potencial también en función de la densidad y energía Exc (de intercambio y correlación). Es importante mencionar que la densidad esta dada en función de la posición, para obtenerla, debe realizarse la sumatoria de los niveles ocupados del valor absoluto del cuadrado de la función de onda. Y nuevamente el principio variacional es igual a 0. Entonces la ecuación del DFT resulta al sumar la energía cinética más la energía colombiana más la energía de intercambio y correlación.

Puede observarse que ambas ecuaciones dan como resultado el vector energía por la longitud de onda.

Para resolver la ecuación, en escaso se procederá a transformar la expresión polinomio a una matriz del tipo cuadrática, la cual tiene la peculiaridad de ser simétrica, por lo que es aconsejable resolverla por el método numérico de rotaciones de Jacobi. Con dicho método se podrán encontrar los valores característicos y finalmente dar solución a la ecuación de Khon-Sham.

1.3 Modelado molecular en GPU

El modelado molecular en GPUs es una técnica que usa el poder de procesamiento de las tarjetas gráficas para realizar simulaciones moleculares. Está mayormente utilizado para realizar cálculos y simulaciones en el campo de la fisicoquímica cuántica. Existen ya algunos sistemas que implementan esta tecnología.

2.9.2.1 TeraChem

Es un software de propósito general de química cuántica, está diseñado para correr en las GPU de NVIDIA, fue desarrollado en la Universidad de Stanford por el profesor Todd Martinez, quien fue pionero en el uso de la tecnología GPU para la química computacional.

TeraChem simula la dinámica y el movimiento de las moléculas, resolviendo la ecuación electrónica de Schrödinger para determinar las fuerzas entre los átomos, lo cual es denominado como “dinámica molecular ab initio”. Algunas de sus características son:

* Códigos de energía y gradiente de Kohn-Sham basados en grid y Hartree-Fock restringidos, no restringidos y restringidos.
* Cuadrículas DFT estáticas y dinámicas.
* Teoría funcional de la densidad dependiente del tiempo (TDDFT) y CI Singles (CIS).
* La optimización puede llevarse a cabo en coordenadas cartesianas o internas como se especifica en el archivo de inicio (todas las geometrías de entrada se proporcionan en cartesianos).
* Optimización restringida con átomos congelados, longitudes de enlaces restringidos, ángulos y diedros.
* Condiciones esféricas de frontera.
* Soporte completo para GPUs de NVIDIA.

2.9.2.3 deMon2k

Es un paquete de software para realizar cálculos con la Teoría del Funcional Densidad, el cual utiliza las combinaciones de orbitales de tipo Gausseano para dar solución a las ecuaciones Kohn-Sham [17].

La primera versión disponible de deMon apareció en 1992 en la Université de Montréal (UdM) como un proyecto de tesis de doctorado [[1]](#footnote-1).

deMon significa “densité de Montréal”.

Algunas características que contiene el programa son [18]:

* Ajuste variable del potencial de Coulomb.
* Optimización geométrica y búsqueda de estado de transición.
* Simulaciones dinámicas moleculares.
* Cálculo de propiedades como polarizabilidades, hiperpolarizabilidades, RMN, espectros e intensidades IR e Raman, datos termodinámicos.
* Código paralelo (MPI).
* Interfaces para software de visualización (Molden, Molekel, Vu).
* Portabilidad a varias plataformas informáticas y sistemas operativos.

El sistema esta pensado para correr en sistemas Unix y Linux, la compilación se realiza bajo FORTRAN90. El programa también esta pensado para correr en paralelo utilizando MPI (Message Passing Interface), pero únicamente si se tienen las bibliotecas instaladas.

La solución que se implementará, está basada en el módulo del cálculo de la energía por teoría del DFT de este paquete.

Capítulo 2. Algoritmo de Jacobi

2.1 Forma cuadrática

Todo polinomio de segundo grado homogéneo de la forma:

Dicho polinomio puede expresarse de una manera matricial de la forma:

Donde la matriz A asociada a la forma cuadrática es una matriz simétrica (Hermitiana) de orden n cuyos elementos de la diagonal principal son los coeficientes de los términos cuadráticos de la expresión polinómica, y los restantes elementos de la matriz son la mitad de los coeficientes de los términos no cuadráticos de dicha expresión. La matriz con forma cuadrática es simétrica, lo que significa que los términos de productos son semejantes(). Esta relación entre los elementos de una y otra expresión de la forma cuadrática, permite obtener fácilmente cada una de ellas a partir de la otra [19].

Ejemplo.

Se tiene el siguiente polinomio y se expresará la matriz de forma cuadrática.

2.2 Eigensistemas

En muchos problemas en el campo de las matemáticas aplicadas, es necesario el resolver ecuaciones lineales con la forma:

o de la forma:

donde A es una matriz de *nxn* con parámetros escalares λ conocidos como valores característicos o eigenvalores, x es una matriz columna de variables independientes también llamado *eigenvector*.

Obviamente cualquier múltiplo de un eigenvector x podrá considerarse como eigenvector, excepto para este caso el 0. El problema es encontrar λ y su correspondiente eigenvector.

2.2.1 Transformación de coordenadas

Para mostrar el siguiente concepto, tenemos una matriz x como vector columna con dos componentes y , si ponemos el los ejes de y con el mismo origen, tenemos que el vector tiene las dos diferentes componentes mencionadas [20]. Trigonométricamente tenemos:

o en la forma matricial:

A esto se le llama la relación de transformación, lo que da:

que conecta los dos sistemas y .

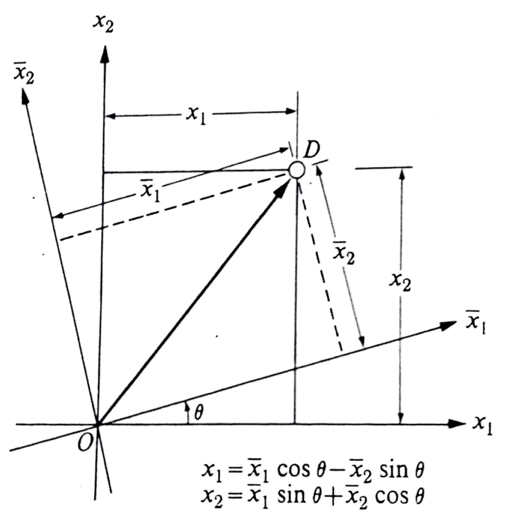


Figura 3 Giro de ejes de una matriz simétrica.

Lo importancia de la matriz T toma lugar cuando sustituimos la ecuación en la de los eigensistemas, dándonos:

pudiendo también:

recordando que , comprobándolo con:

por ende deducimos que:

2.2.2 Eigenvalores

Nuestro objetivo es que todos los elementos fuera de la diagonal se vuelvan cero, para ello es necesario seleccionar un ángulo de rotación que permita realizarlo.

Como queremos eliminar el elemento fuera de la diagonal y teniendo en cuenta que y son similares:

por identidades trigonométricas podemos concluir:

Dando como resultado esperado:

Con lo que al final del procedimiento podemos obtener que y son los eigenvalores deseados.

2.2.3 Eigenvector

Los eigenvectores son matrices columna de una matriz a los que les corresponde un eigenvalor . Para agrupar los eigenvectores en una matriz cuadrada únicamente se agrupan de izquierda a derecha. También es necesario agrupar los eigenvalores en una matriz diagonal . Así podemos expresar la igualdad:

Si aplicamos un poco de algebra:

Tomando en cuenta que :

Por ello podemos entender que la matriz cuadrada que representa los vectores es igual a las multiplicaciones sucesivas de las matrices .

Capítulo 3. Implementación en C, OpenACC y CUDA

Ahora bien, se implementará la resolución de la matriz de Khon-Sham en lenguaje C, posteriormente se realizará la programación en una tarjeta gráfica habilitada para OpenACC y para CUDA.

3.1 Cómputo en GPU

El cómputo en GPU o también llamado GPGPU (cómputo de propósito general en unidades de procesamiento de gráficos) es utilizado para acelerar el procesamiento realizado tradicionalmente por únicamente la CPU, donde la GPU actúa como un coprocesador que puede aumentar la velocidad del trabajo, donde la computación es más intensiva aprovechando la enorme potencia del procesamiento paralelo, mientras que el resto del código se ejecuta en la CPU [21].

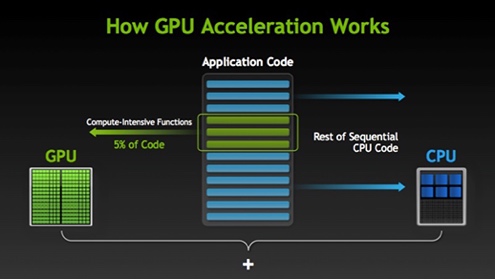


Figura 4 Aceleración de programas en GPUs.

3.1.1 Diferencia entre CPU y GPU

El CPU es un procesador de propósito general, lo que significa que puede hacer cualquier tipo de calculo, pero esta diseñada para realizar el procesamiento en serie. Aunque se pueden utilizar bibliotecas para realizar concurrencia y paralelismo, el hardware per se no tiene esa implementación [22].

El GPU es un procesador mucho más especializado para tareas que requieren de un alto grado de paralelismo. La tarjeta gráfica en su interior contiene miles de núcleos de procesamiento que son más pequeños y que por ende realizan un menor número de operaciones. Esto hace que la GPU esté optimizada para procesar cantidades enormes de datos pero con programas más específicos [23]. Lo más común al utilizar la aceleración por GPU es ejecutar una misma instrucción a múltiples datos para aprovechar su arquitectura.

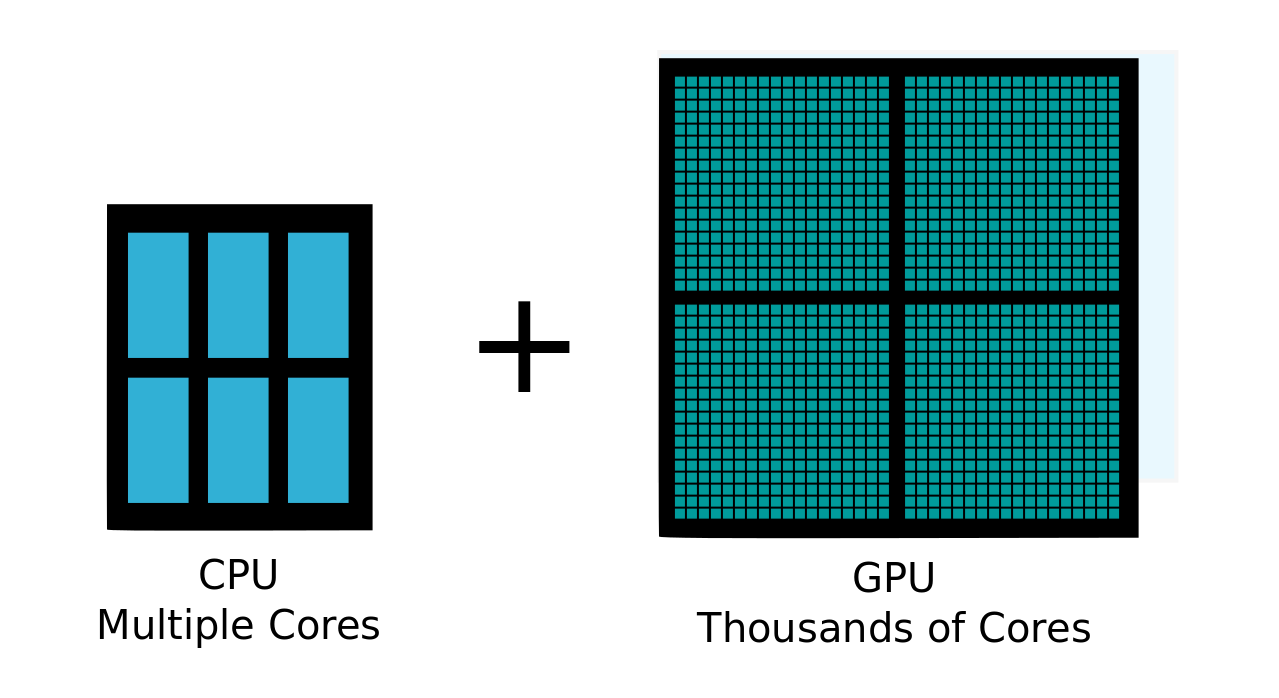


Figura 5 Comparación entre Multicore y Manycore.

Es necesario destacar que los manycore y los multicore son utilizados para etiquetar a los CPU y los GPU, pero entre ellos existen diferencias. Un core de CPU es relativamente más pesado, está diseñado para realizar un control lógico muy complejo para buscar y optimizar la ejecución secuencial de programas. En cambio un core de GPU es más ligero y está optimizado para realizar tareas de paralelismo de datos como un control lógico simple enfocándose en el throughput de los programas paralelos.

Con aplicaciones computacionales intensivas, las secciones del programa a menudo muestran una gran cantidad de paralelismo de datos. Las GPU se usan para acelerar la ejecución de esta porción de paralelismo de datos. Cuando un componente de hardware que está físicamente separado de la CPU se utiliza para acelerar secciones computacionalmente intensivas de una aplicación, se lo denomina acelerador de hardware. Se puede decir que las GPU son el ejemplo más común de un acelerador de hardware.

3.2 Caracteristicas del equipo

Las pruebas se realizaron en el servidor NV que está instalado en la Facultad de Ciencias. Entre sus caracteristicas:

|  |  |
| --- | --- |
| Sistema Operativo | CentOS Linux release 7.3.1611 |
| Arquitectura | 64 bits |
| Dusco duro | 1TB |
| Procesador | Intel(R) Core(TM) i7-7700 @ 3.60GHz |
| Tarjeta de video | Nvidia GeForce GTX 1060 6GB |
| Arquitectura de tarjeta de video | Pascal |
| Máximo de threads por bloque | 1280 |
| Memoria | 6GB |

3.3 Implementación en C

Ahora empezaremos la descripción de la implementación del algoritmo Jacobi para resolver la matriz de Khon-Sham en lenguaje C.

3.3.1 Creación de matrices

Antes que nada se creó un programa llamado **creaMatrix.c**, el cual emula las matrices cuadráticas propias que se forman a partir de la ecuación de Khon-Sham, este programa genera archivos .txt con un formato especial, donde el primer elemento es el orden de la matriz, y los siguientes datos son los respectivos a cada posición de la matriz, y el formato del archivo a generar inicia con la cadena “matrix” y se le concatena el orden n de la matriz, concatenándosele al final la extensión “.txt”. Por ejemplo:

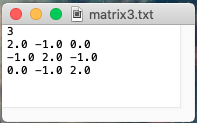


Figura 6 Matriz de 3 x 3.

3.3.2 Funciones auxiliares

Fue necesario crear una biblioteca llamada **auxFuncs.h** que contuviera algunas funciones auxiliares que se utilizarían a lo largo del programa completo. Está biblioteca se basó en la que contiene el libro de Numerical Recipes [24]. Aquí se almacenan las funciones que obtienen el tiempo del sistema y la transforman a segundos, también contiene funciones que reservan memoria para vectores de elementos de tipo entero y de tipo flotante, y por último una función que en caso de existir algún problema en la asignación de la memoria dispara el código de error. Por términos de compatibilidad entre las versiones de código en las tecnologías que hacen uso de las tarjetas gráficas, las matrices se verán en forma de vectores.

3.3.2.1 Matrices Vector

Lo más fácil para convertir una matriz a un arreglo es desdoblar cada renglón en serie.

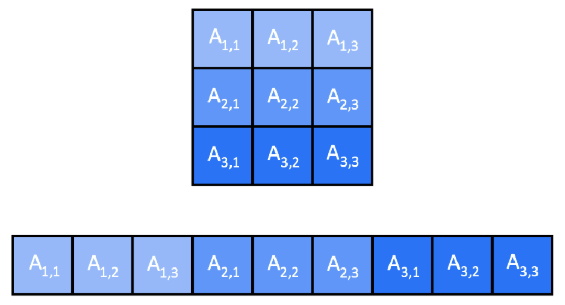


Figura 7 Conversión de matriz a arreglo.

Pasando así de *A[i][j] a Am[i\*n+j],* donde i representa el renglón, j la columna y n el orden de la matriz, pensando que sea cuadrada.

3.3.3 Driver

Para realizar las corridas del método numérico, se realizó un programa llamado **driverA.c** que contiene la función *main()*, aquí se deben agregar las bibliotecas **auxFuncs.h** y **jacobiA.h** que contienen las funciones necesarias para correr el código.

primero lee una matriz de un archivo que le fue pasado como parámetro al ejecutar el programa, dicho archivo debe contener el formato que ya sé especifico. Si existe el archivo lo va leyendo y guardando en una matriz c, si no, retorna 1 y el programa termina con un error. Posteriormente al haber leído la matriz, empieza a reservar la memoria de una matriz de operación, la matriz que contendrá a los eigenvectores y un arreglo que contendrá a los eigenvalores. Para mantener la integridad de la matriz leída, se copia la información en la matriz de operación y se inicializa la estructura de eigenvectores con la matriz identidad. Para verificar que matriz es la que se va a operar, se imprime la matriz de operación.

Ahora se inicia la función *jacobiMultip()*, pero para cronometrar el tiempo que tarda en obtener las soluciones del método se obtiene el tiempo del sistema antes y después de realizar la subrutina. La función tiene como parámetros:

|  |  |
| --- | --- |
| *jacobiMultip(mat,n,evec,eval,nrot);* | mat -> La matriz de operación. |
| n -> El orden de la matriz. |
| evec -> El eigenvector |
| eval -> El arreglo de eigvec |
| nrot -> Variable inicializada en 0 que da el nuemero de rotaciones que realizo el algoritmo |

Al momento de concluirse el cálculo se imprime la matriz de operación nuevamente, que ahora en su diagonal principal tendrá obligatoriamente en su diagonal principal los eigenvalores. Luego se imprime el eigenvalor con su eigenvector correspondiente.

Para finalizar el driver, se imprime tanto el número de rotaciones que se necesitaron para obtener el resultado final como el tiempo que tardo en llegar a la solución.

3.3.4 Jacobi

La biblioteca **jacobiA.h** contiene la función jacobiMultip(), la cual es el núcleo de operación del método numérico para resolver la matriz de Khon-Sham.

Dentro de esta matriz se debe reservar memoria para una matriz temporal que nos será de utilidad para auxiliar a los cálculos, una matriz T que será la que contendrá las transformaciones de las coordenadas y un pequeño vector que será el almacén de las coordenadas del elemento a eliminar de la matriz original.

Se empezará a iterar el método buscando el elemento máximo en el triangulo superior de la matriz y almacenando las coordenadas en el vector *piv\_element[]*. Las iteraciones las realizará hasta que se encuentre que el máximo elemento a eliminar es menor que el threshole (límite) recomendado por Numerical Recipies de 1x10-7 [24] o que haya superado las 50,000 iteraciones, en cualquiera de los casos termina el ciclo por que ha tomado el máximo elemento como |0|.

En caso de que aún existan elementos a eliminar, se realizan 5 funciones que hacen que el método numérico acerque los valores a cero. Las funciones son:

*new\_T\_mat(piv\_elem[0],piv\_elem[1],n,mat,T,mat\_temp);*

Se recibe como parametros la posicion en i y en j del elemento máximo, el orden de la matriz, y la matriz T donde se guardaran las transformaciones de coordenadas. Inicializa la matriz T con la identidad, calcula el seno, el coseno y la tangente y los agrega en las posiciones necesarias para ir eliminando los valores de la matriz de operación.

*mat\_mult(n,eigvec,T,mat\_temp);*

Esta función multiplica el eigenvector con la matriz T y la asigna en la matriz auxiliar. La multiplicación de matrices que se realiza es trivial ya que se utilizan 3 for para recorrer las matrices y almacenar el producto en la tercer matriz.

*copy\_mat(n,mat\_temp,eigvec);*

Se recibe como parámetro el orden de la matriz original, la matriz auxiliar y el eigenvector. Lo que se realiza aquí es el copiado elemento a elemento de lo obtenido en la función anterior a el eigenvector para que se mantenga actualizado con los últimos valores.

*mat\_mult\_tra(n,T,mat,mat\_temp);*

En esta sección es necesario realizar la premultiplicación y la postmultiplicación de la Matriz T por la matriz de operación, por lo que esta función lo que realiza es la operación de la traspuesta de T por mat, para evitar el calculo de la traspuesta, se modifica un poco la función de *mat\_mult()* y se cambia el orden de recorrido de los renglones de T, para que ahora recorra por columnas y la operación se guarda en la matriz auxiliar.

*mat\_mult(n,mat\_temp,T,mat);*

Por último se realiza la posmultiplicación de la matriz auxiliar por T y se almacena el resultado en la matriz de operación.

Lo anterior se realiza hasta que ya no exista un máximo, por lo que cuando termina el ciclo, la diagonal principal de la matriz se asigna a cada posición correspondiente de los eigenvalores.

Y termina la funcion *jacobiMultip()* devolviendonos al *main()* del driver.

3.3.5 Compilación

Para compilar el programa desde la terminal, es necesario unicamente tener el compilador GNU que está por defecto en los sistemas basados en UNIX.

La sintaxis de la compilacion debería ser:

*gcc –o jacobi driverA.c*

Y para correrlo, se le debe pasar como parametro el nombre de un archivo con la matriz a resolver.

*./jacobi file.txt*

Un ejemplo de la salida con una matriz de 3x3 es:

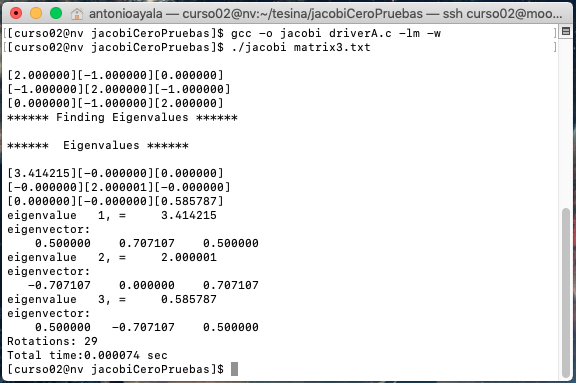


Figura 8 Salida de Jacobi con matriz de 3 x 3 en C.

3.4 Implementación en OpenACC

Se modificará el código para poder utilizar el estándar de OpenACC y paralelizar automáticamente en la tarjeta gráfica el algoritmo de Jacobi.

3.4.1 Estándar OpenACC

Como veremos posteriormente, CUDA al ofrecer un entorno de programación de bajo nivel, ha mantenido un poco alejado a los programadores que están fuera del desarrollo del cómputo científico, por ello se creó el estándar de programación OpenACC [25], que con solo algunas directivas indica al compilador que cierta sección de código podría ser paralelizable, y éste se encargaría de verificar si existen dependencias de dato o si es posible realizarse.

Al no manejarse completamente el comportamiento del programa a un bajo nivel, tiene la desventaja de que el rendimiento puede ser inferior al de CUDA, pero se acelera el desarrollo de los proyectos al no tenerse que modificar la lógica de programación.

Para poder utilizar el estándar es necesario tener el compilador PGI que tiene las bibliotecas de automatización del paralelismo.

3.4.1.2 Estructura de CUDA

Consta de cuatro principales componentes: gangs, workers y vectors [26].

|  |  |
| --- | --- |
| vectors | Es el elemento de granularidad más fina, su simil en SIMD es el thread. |
| workers | Son el intermediario entre gangs y vectors, permiten el mapeo de los vectors en el hardware. |
| gangs | Es un grupo o bloque de workers, en diferentes gangs, los workers pueden tener trabajos independientes. |

3.4.1.3 Paralelizar Bucles

Es necesario identificar los puntos estratégicos de paralelización como pueden ser las estructuras de bucle ya que aquí es donde se puede acelerar el procesamiento. En este caso no es necesario preocuparse del manejo de la memoria, únicamente es necesario verificar si se están obteniendo los datos correctos. La filosofía de OpenACC marca que es necesario asegurarse que primero se identifiquen los puntos estratégicos de paralelismo para después verificar cuál es la directiva más apropiada para el fragmento de código [26].

3.4.1.4 Directivas

Para darle los parámetros de la configuración de threads al compilador, debe usarse *#pragma acc parallel num\_gangs(#), vector\_length(#),* y por la version de nuestra tarjeta gráfica usaremos 32 gangs de 64 workers para poder operar matrices de máximo 2048 renglones. Una vez dados los parametros, debe indicarse que acción de OpenACC a utilizar.Sin profundizar mucho, la directiva más utilizada por su simplicidad es *#pragma acc loop,* al momento de identificar que una región puede ser paralelizable, se agrega la línea y automáticamente el compilador verificará si es posible realizar su paralelización.

En el ejemplo en que tenemos una multiplicación de matrices, por recomendación [26], se deben poner las directivas donde se realice el recorrido de los renglones y las columnas.

|  |
| --- |
| *#pragma acc parallel num\_gangs(32), vector\_length(64)*  *{*  *#pragma acc loop*  *for (i = 1 ; i <= n ; i++ ){*  *#pragma acc loop*  *for (j = 1 ; j <= n ; j++ ){*  *for (k = 1 ; k <= n ; k++ ){*  *A[i\*n+j] += B[k\*n+i] \* C[k\*n+j];*  *}*  *}*  *}*  *}* |

La siguiente recomendación [26] es que si existe un bloque en donde no se realice otra operación que asignación de números sin tanta precisión como lo pueden ser los enteros, es preferible dejar que el CPU utilice sus pipelines para acelerar el código y no utilizar la directiva.

3.4.1.5 Portabilidad

El estándar permite la portabilidad en el sentido que, si no se cuenta con el compilador PGI, puede utilizarse cualquier otro para C, y en el proceso de compilación simplemente se ignoraran las directivas.

3.4.2 JacobiACC

En la biblioteca **jacobiA.h** se buscan las posibles regiones que se puedan paralelizar.

La primera se encuentra en la función *max\_elem()* y la directiva se colocan al inicio del primer ciclo.

|  |
| --- |
| *max\_elem() {*  *…*  *#pragma acc parallel num\_gangs(32), vector\_length(64)*  *{*  *#pragma acc loop*  *for (r = 0; r < n-1; r++)*  *}*  …  } |

La siguiente region paralelizable, se encuentra en las fucniones *mat\_mult()* y *mat\_mult\_tra()*,Por lo que seguimos la recomendación y se colocan al inicio del primer y segundo for.

|  |
| --- |
| *mat\_mult(){*  *…*  *#pragma acc parallel num\_gangs(32), vector\_length(64)*  *{*  *#pragma acc loop*  *for (i = 0 ; i < n ; i++ ){*  *#pragma acc loop*  *for (j = 0 ; j < n ; j++ ){*  *C[i\*n+j] = 0.0;*  *for (k = 0 ; k < n ; k++ )*  *C[i\*n+j] += A[i\*n+k] \* B[k\*n+j];*  *}*  *}*  *…*  *}* |
| *mat\_mult\_tra(){*  *…*  *#pragma acc parallel num\_gangs(32), vector\_length(64)*  *{*  *#pragma acc loop*  *for (i = 0 ; i < n ; i++ ){*  *#pragma acc loop*  *for (j = 0 ; j < n ; j++ ){*  *C[i\*n+j] = 0.0;*  *for (k = 0 ; k < n ; k++ )*  *C[i\*n+j] += A[k\*n+i] \* B[k\*n+j];*  *}*  *}*  *}*  *…*  *}* |

Finalmente, se agrega la directiva en la funcion copy\_mat() al inicio del primer for.

|  |
| --- |
| *copy\_mat(){*  *…*  *#pragma acc parallel num\_gangs(32), vector\_length(64)*  *{*  *#pragma acc loop*  *for (i = 0; i < n; ++i)*  *for (j = 0; j < n; ++j)*  *B[i\*n+j]=A[i\*n+j];*  *}*  *…*  *}* |

Y así obtenemos nuestra versión de OpenACC a partir del código en C. Por lo que mantenemos el driver intacto.

3.4.3 Compilación

Para compilar, como ya se mencionó, es necesario tener el compilador PGI, en donde se activar la directiva de compilación *–acc* para indicar que se utilizará el estándar ahora indicaremos que automáticamente se le asigne los threads necesarios en cada matriz, utilizamos la directiva *-ta=multicore*.

La sintaxis de la compilacion debería ser:

*pgcc -acc -ta=multicore –o jacobiACC driverA.c*

Y para correrlo, se le debe pasar como parametro el nombre de un archivo con la matriz a resolver.

*./jacobiACC file.txt*

Nuevamente mostrando un ejempo con la misma matriz de 3x3:

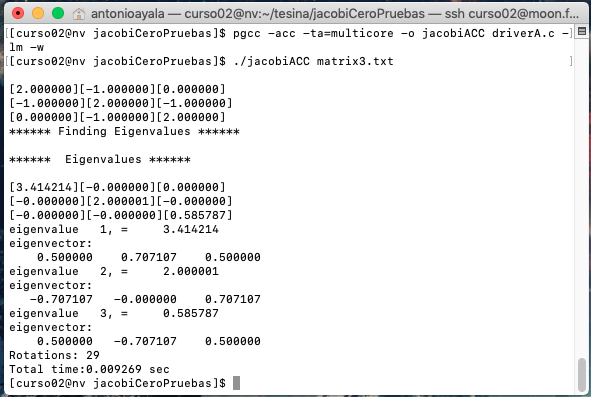


Figura 9 Salida de Jacobi con matriz de 3 x 3 en OpenACC.

3.5 Implementación en CUDA

3.5.1 Modelo de programación en CUDA

CUDA es el acrónimo en inglés de *Compute Unified Device Architecture*, el cual es una arquitectura de hardware y de software que permite ejecutar programas en las tarjetas gráficas de la marca NVIDIA [27]. Utiliza el modelo SIMT (Single Instruction, Multiple Thread) el cual está basado en el modelo SIMD [28], pero adecuado al la arquitecturas de las tarjetas gráficas.

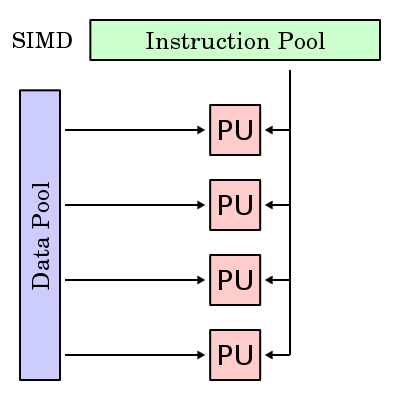


Figura 10 Diagrama de modelo SIMD.

La programación en CUDA es adecuada para resolver problemas que pueden ser expresados en paralelismo de datos, por la arquitectura que manejan.

Se explicará el caso particular de la extensión de CUDA en C, ya que es el lenguaje en el que la implementación del algoritmo se realizará.

3.5.1.1 Arquitectura CUDA

CUDA C es una extensión del estándar ANSI C con varios complementos del lenguaje para utilizar la programación heterogénea y también API sencillas para administrar los dispositivos, memoria y otras tareas. CUDA también es un modelo de programación escalable que permite a los programas trabajar transparentemente con un número variable de núcleos, aumentando o disminuyendo la cantidad de cores [29].

CUDA nos provee dos niveles de API para manejar el GPU y organizar los threads. El *API driver* es un API de nivel bajo y es un poco difícil de programar, pero provee un control sobre como el dispositivo (GPU) está siendo usado. El *API runtime* es un API a un nivel más alto, cada función de esté nivel se rompe en operaciones más básicas del API driver.

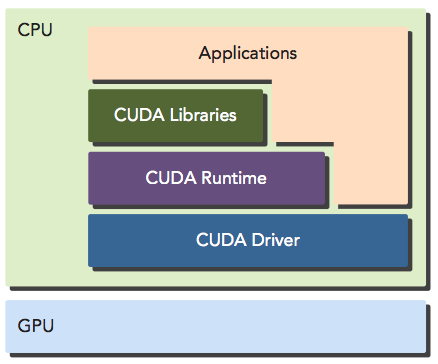


Figura 11 Arquitectura CUDA.

Un programa en CUDA consiste en la mezcla de dos códigos, el host code y el device code. El compilador de NVIDIA, *nvcc*, separa ambos códigos durante el proceso de compilación. Durante la etapa de enlace, las librerías de CUDA se agregan al procedimiento de las funciones que irán al device para poder manipular completamente la tarjeta.

El compilador nvcc esta basado en LLVM[[2]](#footnote-2), por lo que el lenguaje puede ser extendido creando nuevos soportes para el GPU, eso si, utilizando el CUDA Compiler SDK que nos comparte NVIDIA [29].

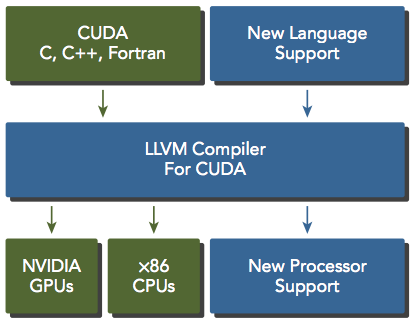


Figura 12 Jerarquía LLVM.

3.5.1.2 Estructura de CUDA

Consta de cuatro principales componentes: kernel, threads, bloques y grids.

|  |  |
| --- | --- |
| kernel | Son las funciones paralelas escritas en el programa que indican que operaciones se realizaran en el GPU. |
| thread | Un thread de un bloque ejecuta una instancia de un kernel. Tiene su propio id dentro de su bloque, su propio PC, registros, memoria privada, entradas y salidas. |
| bloque | Un thread de un bloque ejecuta una instancia de un kernel. Tiene su propio id dentro de su bloque, su propio PC, registros, memoria privada, entradas y salidas. |
| grid | Es un arreglo de bloques de threads que ejecutan el mismo kernel, leen entradas desde memoria global, escriben resultados en memoria global, y se sincronizan entre otras llamadas a kernels. |

Cada thread en CUDA tiene su propio contador de programa y registros. Todos los threads comparten un espacio de direcciones de memoria llamado “global memory”, el cual es la memoria con la que se comunican lo bloques entre sí. Todos aquellos que se encuentren dentro de un bloque comparten el acceso a una memoria muy rápida llamada “shared memory”, pero el tamaño es más limitado.

NVIDIA define las características y limitaciones de CUDA con las “compute capabilities”, por ello los threads deben estar agrupados en bloques, de los cuales pueden poseer cuando mucho, en las últimas versiones, de 512 ó 1024 threads, dependiendo de la versión del CUDA. Los bloques de threads pueden ser de una, dos o tres dimensiones [29].

Hay que tener en cuenta que aunque todos los threads en cada bloque van a trabajar, lo primero que harán es localizar si se encuentran dentro de la matriz, si lo están, realizan la instrucción, pero si no, inmediatamente finalizarán.

3.5.1.3 Mapeo de Estructuras

Los threads son necesarios de organizar para que realicen un procesamiento ordenado y no generen condición de competencia. Por lo anterios, es necesario diseñar la forma en la cual accederán a los datos.

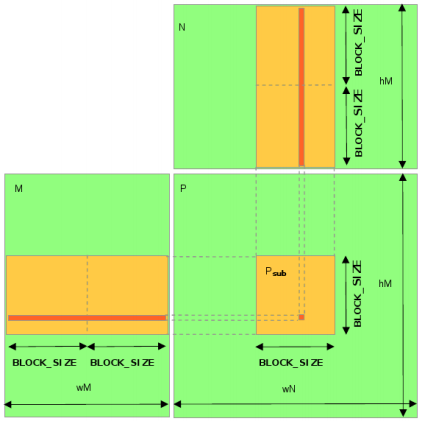


Figura 13 Mapeo de estructuras en CUDA.

Al momento de ejecutar el programa, se pueden obtener los id de las siguientes estructuras para verificar en que posición de procesamiento nos encontramos [30] [29].

|  |  |
| --- | --- |
| gridDim.x , gridDim.y , gridDim.z | Numero de block que se encuentran a lo largo de las dimensiones x,y,z en el Grid. |
| blockIdx.x , blockIdx.y , blockIdx.z | El índice del bloque en el que se encuentra. |
| blockDim.x , blockDim.y , blockDim.z | El numero de threads que se encuentran en un bloque . |
| threadIdx.x , threadIdx.y , threadIdx.z | El id del thread en que se está procesando. |

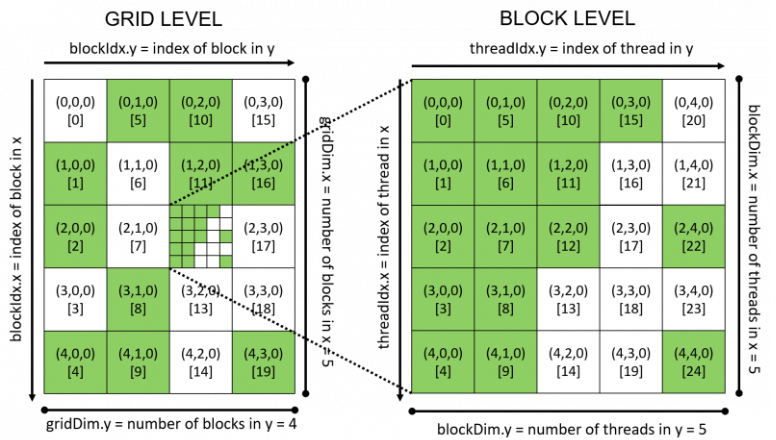


Figura 14 Dimensión de Grids y Blocks en CUDA.

3.5.1.4 Kernels

El modelo de programación CUDA permite ejecutar aplicaciones en sistemas heterogéneos simplemente anotando el código con un pequeño conjunto de extensiones para el lenguaje de programación C, pero es importante decir que tanto el host como el device tienen memoria independiente, por lo que para acceder a los datos del otro es necesario enviar la información. Para ayudar a diferenciar entre que variables están en el host y en el device, se recomienda empezar con *d\_* las del segundo.

Un componente clave del modelo de programación de CUDA es el kernel, ya que se puede expresar un kernel como un programa secuencial, pero en realidad, CUDA administra la programación de ellos y los asigna a los threads [29] . La nomenclatura de un kernel es parecida a la de una función, pero hay que agregar la directiva \_\_global\_\_ (doble guion bajo a cada lado. Para diferenciarlas de las funciones del host, se sugiere que empiecen con *kernel\_.* Un ejemplo de esto podría ser:

*\_\_global\_\_ void kernel\_helloFromGPU(argument list){}*

Para poder lanzar el kernel, se debe utilizar la nomenclatura siguiente:

*kernel\_name <<<grid, block>>>(argument list);*

El host puede operar independientemente del dispositivo, ya que, cuando se ha lanzado un kernel, el control se devuelve inmediatamente al host liberando a la CPU para realizar tareas adicionales complementadas por el código paralelo de datos que se ejecuta en el dispositivo [29]. El modelo de programación de CUDA es principalmente asincrónico, por lo que el cómputo del device puede superponerse con el procesamiento del dispositivo host y realizar actividades al mismo tiempo.

El proceso típico del flujo de datos de un programa en CUDA sigue el siguiente patrón:

Figura 15 Pipeline de ejecución de CUDA.

3.5.1.5 Manejo de memoria

Al diseñar un programa en paralelo con partición de datos, es necesario que dichos datos sean enviados a cada thread para que trabajen los datos asignados. Generalmente existen dos tipos de partición, la partición por bloques y la partición cíclica. En la partición por bloque muchos elementos consecutivos son empaquetados juntos, cada paquete es asignado a un thread en orden, por lo que cada thread procesa un paquete a la vez. En cambio la partición cíclica tiene menos datos empaquetados juntos, al terminar solicitan más trozos de información [29], resaultando así en que pueden ir saltando entre los datos.

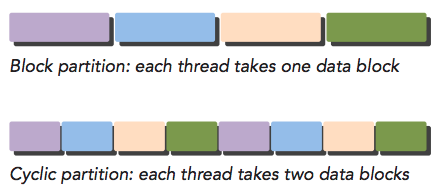


Figura 16 Partición de datos en CUDA.

Usualmente los datos están guardados en una dimensión, aunque el punto de vista del problema requiera una visualización multidimensional. Determinar el cómo distribuir datos entre threads está estrechamente relacionado con la forma en que los datos se almacenan físicamente, así como orden de ejecución de cada uno. La forma de organización de los threads tiene un efecto significativo en el rendimiento del programa.

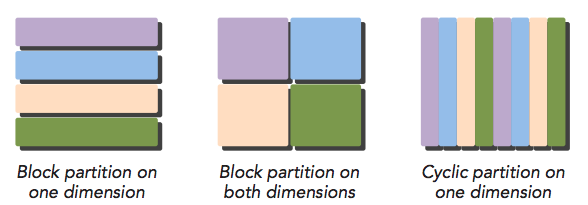


Figura 17 Partición de datos de una matriz en CUDA.

El modelo de programación CUDA supone un sistema compuesto por un host y un dispositivo propia memoria separada entre si. El tiempo de ejecución de CUDA proporciona funciones para asignar y liberar memoria del dispositivo y transferir datos entre la memoria del host y la memoria del dispositivo.

Aquí hay una tabla que muestra las funciones correspondientes a C y a CUDA C para administrar la memoria

|  |  |
| --- | --- |
| Funciones estándar de C | Funciones de CUDA C |
| *malloc* | *cudaMalloc* |
| *memcpy* | *cudaMemcpy* |
| *memset* | *cudaMemset* |
| *free* | *cudaFree* |

La función utilizada para transferir datos entre el host y el device es cudaMemcpy, pero es necesario indicar cual es la fuente y el destino.

*cudaMemcpyHostToDevice*

*cudaMemcpyDeviceToHost*

*cudaMemcpy(d\_a, h\_a, nBytes, cudaMemcpyHostToDevice);*

*cudaMemcpy(h\_b, d\_b, nBytes, cudaMemcpyDeviceToHost);*

Esta función tiene un comportamiento síncrono, ya que el host se bloquea hasta que la función regrese una bandera de que la transferencia se realizo con éxito [29].

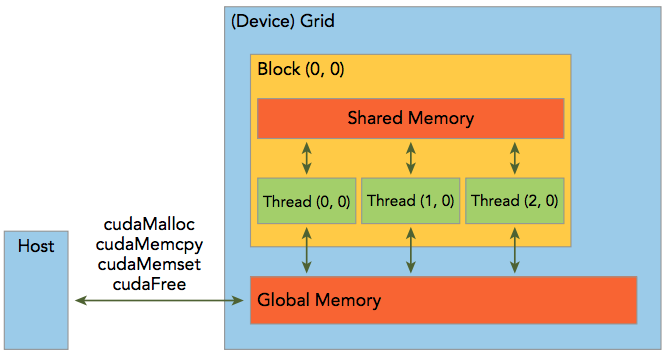


Figura 18 Transferencia de datos entre Host y Device en CUDA.

3.5.1.6 Trabajando con matrices

Cuando se trabaja con matrices en CUDA es necesario convertirlas a arreglos de una dimensión. Esto puede sonar un poco confuso, pero el problema es el lenguaje en si, ya que necesita saber el número de columnas que se tendrán antes de compilar el programa, lo que hace imposible cambiarlo en la mitad del código. Por ello no podemos utilizar la notación de *A[i][j]* si no que debemos utilizar una manera para indexar correctamente entre renglones y columnas [29]. Como ya habíamos visto anteriormente, lo más fácil para convertir una matriz a un arreglo es desdoblar cada renglón en serie, pasando así de *A[i][j] a Am[i\*n+j],* donde i representa el renglón, j la columna y n el numero total de columnas.

3.5.1.7 Id de Threads

Al trabajar con matrices, es necesario que verifiquemos que sección del threads ingresaran a la estructura de datos para evitar duplicidad de operaciones o, al contrario, estemos fuera del rango de la matriz.

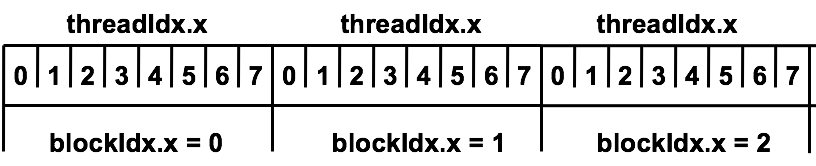


Figura 19 Orden de ID de los threads de CUDA.

3.5.2 Driver

Al código del driver, únicamente se le quitará la biblioteca **jacobiA.h** y se intercambiará por **jacobiACUDA.h**, y el siguiente cambio únicamente es el de cambiar la extensión **\*.c** por **\*.cu**

3.5.3 JacobiCUDA

En la biblioteca que contiene la función de Jacobi es donde se realizan los cambios mayores, ya que es necesario mover la información entre el host y el device.

Por cada una de las variables que se tienen originalmente es necesario crear su homologo en el device, por lo que, por buenas practicas, se debe iniciar el nombre con d\_{variable}.

Ahora es necesario indicar las dimensiones de los grid y bloques, por las características que se tienen en la computadora, se decidió tener una dimensión de grids con 32 bloques de 64 threads cada uno, esto nos permitirá realizar matrices máximo de 2048 x 2048.

Ahora es necesario reservar la memoria en el device para las variables que se operaran, para proceder a copiar la información del host al device. Una vez superado este paso las operaciones ser realizaran dentro del device, pero serán controladas por el CPU. Las funciones traducidas a CUDA son las siguientes:

*\_\_global\_\_ kernel\_max\_elem<<<1,1>>>(d\_piv\_elem,n,d\_mat);*

Ya que el elemento máximo unicamente se va a buscar en el triangulo superior, se pude utilizar un solo thread para realizar la operación.

*\_\_global\_\_ kernel\_set\_max\_elem<<<1,1>>>(d\_mat,n,d\_piv\_elem,d\_max\_elem);*

Como la operacion se realiza en el device, es necesario encontrar ese valor máximo y bajarlo al host para que el sea el que determine si ya se llegó al threshole o aun se necesita iterar.

*\_\_global\_\_ kernel\_new\_T\_mat<<<dimGrid, dimBlock>>>(d\_piv\_elem,n,d\_mat,d\_T);*

Cuando todavia hay necesidad de aplicar el método númerico, se debe calcular la matriz de transformación de coordenadas. Donde se checa que los threads que esten fuera del rango de la matriz terminen sin afectar la operación. A cada thead se le encarga que a la columna que les tocó les asignen ceros a todos sus renglones, posteriormente el elemento que esta dentro de la diagonal principal le da el valor 1, para formar la matriz identidad. Si a el thread le toca en su columna un elemento de la transformacion de coordenadas, debe calcular el seno, coseno y tangente y lo asigna a las posiciones que necesitan ser rotadas. Dichas fuciones ahora están unicamente disponibles para ser llamadas en el device porque se les antecede la directiva *\_\_device\_\_.*

*\_\_device\_\_ float cal\_tan()*

*\_\_device\_\_ float cal\_cos()*

*\_\_device\_\_ float cal\_sin()*

*\_\_global\_\_ kernel\_mat\_mult<<<dimGrid, dimBlock>>>(n,d\_eigvec,d\_T,d\_mat\_temp);*

En esta función, se le agrega el índice de los threads de la tarjeta, *y* a las columnas y el de las *x* a las columnas. Nuevamente se descartan los threas que caen fuera de la matriz. Por la naturaleza del índice de los threas, a cada thread se le asigna un renglón de la primer matriz a cada thread una columna de la matriz b. Y es necesario anidar un ciclo para ir recorriendo cada uno de los elementos y operarlos.

\_\_global\_\_ kernel\_copy\_mat<<<dimGrid, dimBlock>>>(n,d\_mat\_temp,d\_eigvec);

Esta operacion es trivial, ya que a cada thread se le asigna una columna de la matiz resultante y el elemento que se encuentra en la diagonal principal lo copia en el arreglo de eigenvalores.

*\_\_global\_\_ kernel\_mat\_mult\_tra<<<dimGrid, dimBlock>>>(n,d\_T, d\_mat ,d\_mat\_temp);*

Esta es la sección del codigo donde hay que poner más atención, ya que el algoritmo pide que se transponga la matriz T, pero para ahorrarnos operaciones, vamos a agregar un indice que nos permita recorrer ambas matrices por las columnas, pero que se conserve el orden de llenado de la matriz solución.

*\_\_global\_\_ kernel\_mat\_mult<<<dimGrid, dimBlock>>>(n,d\_mat\_temp,d\_T,d\_mat);*

Aqui, al igual que en el calculo del eigenvector, se multiplica renglon por columna de lo obtenido en la funcion anterior por la matriz de transformación T.

*\_\_global\_\_ kernel\_copy\_diag<<<dimGrid, dimBlock>>>(n,d\_mat,d\_eigval);*

Al superar el threshold, se rompe el ciclo y se copian los elementos de la diagonal principal al arreglo de eigenvalores.

Ahora que se tienen el resultado, es necesario copiarlo de regreso al host para que se pueda mostrar en pantalla los resultados cudaMemcpy(…,…,…,cudaMemcpyDeviceToHost). Y como ya se llegó al fin del método numérico, es necesario liberar la memoria en el device, esto con la sentencia de cudaFree() a cada una de las variables del dispositivo.

3.5.4 Compilación

Para compilar el código, es necesario tener el compilador NVCC de NVIDIA, y que los archivos de programa tengan terminación \*.cu.

La sintaxis de la compilacion debería ser:

*nvcc -acc –o jacobiCUDA driverA.cu*

Y para correrlo, se le debe pasar como parametro el nombre de un archivo con la matriz a resolver.

*./jacobiCUDA file.txt*

Ahora mostrando un ejempo con la misma matriz de 3x3:

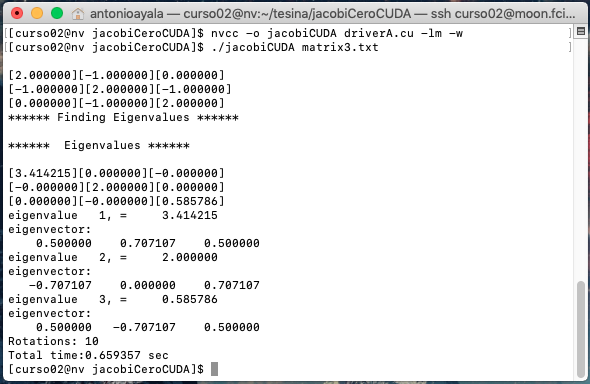


Figura 20 Salida de Jacobi con matriz de 3 x 3 en CUDA.

Capítulo 4. Pruebas y resultados

4.1 Métricas de desempeño

Existe una creciente demanda de poder de cómputo, y algunas áreas que requieren gran velocidad en los cálculos, mayormente están en el campo de la simulación científica o problemas de ingeniería, por ello los cálculos deben ser resueltos en un periodo “razonable” de tiempo. Aparte de la necesidad de incrementar la velocidad, el uso de varios procesadores ofrece comúnmente una solución más precisa al poder partir el problema en diversas subtareas.

Cuando se desarrollan soluciones en paralelo la primer pregunta que surge es “el cuán más rápido se resuelve el problema”, pero antes de saber cuanto se acelera en paralelo es necesario enfrentar el “mejor” algoritmo secuencial (en un solo procesador) para saber la aceleración y ver si compensa la inversión en el sistema paralelo, ya que si no se hace así es mejor buscar técnicas de optimización de código, ahorrándose costos y tiempo.

Ahora bien, al saber que se tiene la mejor versión de la solución en secuencial podemos compararla contra la paralela. Es necesario aclarar que las mediciones no son exactas, si no un promedio, ya que existen más factores que influyen como la asignación de recursos del sistema operativo o la cantidad de memoria que se posea, entre otras cosas [31].

4.1.2 Runtime

El Runtime o tiempo de ejecución, hace referencia al tiempo que pasa entre el inicio del programa y la finalización de todos los procesos.

4.1.2 Cost Factor

Representa la cantidad de trabajo realizado por el programa, ya que, aunque disminuye el tiempo al tener un sistema paralelo, los procesadores están trabajando al mismo tiempo, por ello muchas veces es necesario conocer la totalidad del costo que conlleva la utilización del algoritmo.

4.1.3 Speedup Factor

Es la medición relativa del rendimiento de un programa en paralelo, para calcularla se necesita medir el tiempo que tarda la versión secuencial entre la versión paralela. Hay que tener en cuenta que se debe calcular el tiempo con la misma cantidad de datos a procesar para que la medición sea justa.

4.1.3 Aceleración

Es la medición de la aceleración relativa de un programa en paralelo. Es similar al Speed Up Factor, pero da el porcentaje de la aceleración con respecto al programa en un solo procesador.

2.1.4 Efficiency

A veces es útil conocer cuanto tarda cada procesador en realizar su tarea, regularmente se expresa en porcentaje, por ejemplo, si se obtiene el 50% significa que se están utilizando la mitad del tiempo en promedio que en la versión secuencial.

4.2 Resultados

4.2.1 Resultados de Runtime

|  |  |  |  |
| --- | --- | --- | --- |
| RUNTIME [s] | | | |
| Orden | **Serial** | **OpenACC** | **CUDA** |
| 3 | 0.000074 | 0.009269 | 0.667713 |
| 5 | 0.079594 | 0.105867 | 0.667437 |
| 10 | 0.537204 | 0.277441 | 0.663814 |
| 16 | 2.272125 | 0.606796 | 0.693325 |
| 32 | 16.624936 | 6.420716 | 1.075765 |
| 64 | 127.455333 | 61.942851 | 6.336773 |
| 128 | 1127.73628 | 592.659655 | 87.821701 |
| 256 | 10417.17143 | 5429.11005 | 516.680462 |
| 512 | 76337.37146 | 46845.537 | 2495.80583 |
| 1024 | 381686.8573 | 93691.0741 | 12754.6959 |

|  |  |  |  |
| --- | --- | --- | --- |
| RUNTIME [min] | | | |
| Orden | Serial | OpenACC | CUDA |
| 3 | 0.000001 | 0.0002 | 0.0111 |
| 5 | 0.0013 | 0.0018 | 0.0111 |
| 10 | 0.0090 | 0.0046 | 0.0111 |
| 16 | 0.0379 | 0.0101 | 0.0116 |
| 32 | 0.2771 | 0.1070 | 0.0179 |
| 64 | 2.1243 | 1.0324 | 0.1056 |
| 128 | 18.7956 | 9.8777 | 1.4637 |
| 256 | 173.6195 | 90.4852 | 8.6113 |
| 512 | 1272.2895 | 780.7590 | 41.5968 |
| 1024 | 6361.4476 | 1561.5179 | 212.5783 |

Figura 21Gráfica de RUNTIME.

Figura 22 Gráfica del inicio de la divergencia del RUNTIME.

4.2.2 Resultados de Cost Factor

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| COST FACTOR | | | | | | |
| Orden | **Procesadores** | **Serial** | **Procesadores** | **OpenACC** | **Procesadores** | **CUDA** |
| 3 | 1 | 0.000074 | 32 | 0.296608 | 3 | 2.003139 |
| 5 | 1 | 0.079594 | 32 | 3.387744 | 5 | 3.337185 |
| 10 | 1 | 0.537204 | 32 | 8.878112 | 10 | 6.63814 |
| 16 | 1 | 2.272125 | 32 | 19.417472 | 16 | 11.0932 |
| 32 | 1 | 16.624936 | 32 | 205.462912 | 32 | 34.42448 |
| 64 | 1 | 127.455333 | 32 | 1982.17123 | 64 | 405.553472 |
| 128 | 1 | 1127.73628 | 32 | 18965.109 | 128 | 11241.1777 |
| 256 | 1 | 10417.1714 | 32 | 173731.522 | 256 | 132270.198 |
| 512 | 1 | 76337.3715 | 32 | 1499057.19 | 512 | 1277852.59 |
| 1024 | 1 | 381686.857 | 32 | 2998114.37 | 1024 | 13060808.6 |

Figura 23 Factor de Costo total.

Figura 24 Factor de Costo hasta matriz de 32 x 32.

4.2.3 Resultados de Speed Up

|  |  |  |  |
| --- | --- | --- | --- |
| Speed UP | | | |
| Orden | Serial | OpenACC | CUDA |
| 3 | 100.00% | 0.80% | 0.01% |
| 5 | 100.00% | 75.18% | 11.93% |
| 10 | 100.00% | 193.63% | 80.93% |
| 16 | 100.00% | 374.45% | 327.71% |
| 32 | 100.00% | 258.93% | 1545.41% |
| 64 | 100.00% | 205.76% | 2011.36% |
| 128 | 100.00% | 190.28% | 1284.12% |
| 256 | 100.00% | 191.88% | 2016.17% |
| 512 | 100.00% | 162.96% | 3058.63% |
| 1024 | 100.00% | 407.39% | 2992.52% |

Figura 25 Speed up total.

Figura 26 Speed up de las primeras matrices.

4.2.3 Resultados de Aceleración

|  |  |  |  |
| --- | --- | --- | --- |
| Aceleración | | | |
| Orden | Serial | OpenACC | CUDA |
| 3 | 0.00% | -99.20% | -99.99% |
| 5 | 0.00% | -24.82% | -88.07% |
| 10 | 0.00% | 93.63% | -19.07% |
| 16 | 0.00% | 274.45% | 227.71% |
| 32 | 0.00% | 158.93% | 1445.41% |
| 64 | 0.00% | 105.76% | 1911.36% |
| 128 | 0.00% | 90.28% | 1184.12% |
| 256 | 0.00% | 91.88% | 1916.17% |
| 512 | 0.00% | 62.96% | 2958.63% |
| 1024 | 0.00% | 307.39% | 2892.52% |

Figura 27 Aceleración total.

Figura 28 Aceleración de las primeras matrices.

4.2.4 Resultados de Efficiency

|  |  |  |  |
| --- | --- | --- | --- |
| Efficiency | | | |
| Orden | Serial | OpenACC | CUDA |
| 3 | 100.00% | 0.02% | 0.00% |
| 5 | 100.00% | 2.35% | 2.39% |
| 10 | 100.00% | 6.05% | 8.09% |
| 16 | 100.00% | 11.70% | 20.48% |
| 32 | 100.00% | 8.09% | 48.29% |
| 64 | 100.00% | 6.43% | 31.43% |
| 128 | 100.00% | 5.95% | 10.03% |
| 256 | 100.00% | 6.00% | 7.88% |
| 512 | 100.00% | 5.09% | 5.97% |
| 1024 | 100.00% | 12.73% | 2.92% |

Figura 29 Eficiencia.

Figura 30 Eficiencia sin los valores del programa en C.

4.3 Resultados de una matriz

4.3.1 Matriz a evaluar

Imagen que contiene captura de pantalla



Descripción generada automáticamente

Figura 31 Matriz de 10 x 10.

4.3.2 Resultado Serial

Imagen que contiene texto



Descripción generada automáticamente

Figura 32 Resultados de Jacobi en Serial de una matriz de 10 x 10.

4.3.3 Resultado OpenACC

Imagen que contiene texto



Descripción generada automáticamente

Figura 33 Resultados de Jacobi en OpenACC de una matriz de 10 x 10.

4.3.4 Resultado CUDA

Imagen que contiene texto



Descripción generada automáticamente

Figura 34 Resultados de Jacobi en CUDA de una matriz de 10 x 10.

Conclusiones

Normalmete los GPUs son utilizados para la computación gráfica y videojuegos, pero una alternativa para la optimización de los programas que requieren un calculo intensivo, como el que se realizó en este trabajo, es la implementación del hardware gráfico en investigaciones cientificas.

El cómputo cientifico hace uso de un gran poder de procesamiento, sumado a que necesita que se obtengan resultados en un perido relativamente corto de tiempo. Por eso es de vital importancia encontrar en las nuevas tecnologias, formas que ayuden a la solución de problemas que antes eran impensables resolver.

Al momento de paralelizar el trabajo empiezan a surgir ciertos errores en el cálculo de resultados, ya que puede existir condicion de carrera y muchas veces los threads chocan entre si, o en el caso de CUDA, al no contar con un dispositivo que esté contruido para cálculos de cómputo cientifico, los núcleos de la tarjeta gráfica tienen una precisión de punto flotante menor, entonces surgen errores de truncamiento y/o redondeo.

Por ejemplo en la matriz más pequeña que se probó (3x3):

|  |  |  |  |
| --- | --- | --- | --- |
| Eigenvalores | | | |
|  | **Serial** | **OpenACC** | **CUDA** |
| 1 | 3.414215 | 3.414214 | 3.414215 |
| 2 | 2.000001 | 2.000001 | 2.000000 |
| 3 | 0.585787 | 0.585787 | 0.585786 |
| nrot | 29 | 29 | 10 |

Ahora con una de 5x5:

|  |  |  |  |
| --- | --- | --- | --- |
| Eigenvalores | | | |
|  | **Serial** | **OpenACC** | **CUDA** |
| 1 | 5.630339 | 5.630339 | 5.630342 |
| 2 | -0.277626 | -0.277625 | -0.277601 |
| 3 | -0.391462 | -0.391460 | -0.390448 |
| 4 | -0.786500 | -0.786441 | -0.784438 |
| 5 | -4.177851 | -4.177852 | -4.177853 |
| nrot | 50001 | 50001 | 32 |

Finalmente una de 10 x 10:

|  |  |  |  |
| --- | --- | --- | --- |
| Eigenvalores | | | |
|  | **Serial** | **OpenACC** | **CUDA** |
| 1 | 20.431740 | 20.431728 | 20.431736 |
| 2 | 20.431759 | 20.431734 | 20.431734 |
| 3 | 2.425922 | 2.425918 | 2.425921 |
| 4 | 2.425919 | 2.425920 | 2.425920 |
| 5 | 1.001439 | 1.001523 | 1.000001 |
| 6 | 1.001456 | 1.001410 | 1.000000 |
| 7 | 0.630436 | 0.630444 | 0.629809 |
| 8 | 0.630389 | 0.630422 | 0.629808 |
| 9 | 0.512564 | 0.512567 | 0.512543 |
| 10 | 0.512567 | 0.512565 | 0.512543 |
| nrot | 50001 | 50001 | 158 |

Se puede encontrar que a pocos elementos en las matrices, practicamente el único resultado que varia es el que se obtiene con CUDA, pero una vez que vamos incrementando el orden de las matrices el resultado de OpenACC también va divergiendo de nuestro algoritmo en Serial. Debemos ver qué es lo que nos importa más, obtener un resultado preciso o un resultado más rápido. En este caso en particular, los digitos que difieren son casí insignificantes, y podemos estar dispuestos a perder esa pequeña aproximación.

Pudimos observar que operar las matrices con pocos elementos en paralelo es una perdida de recursos, ya que no varía mucho el tiempo de ejecucion y, en el caso de CUDA, hasta aumenta considerablemente por el desplazamiento de datos entre el host y el device. Pero ahora en el caso contrario, al operar matrices muy grandes, el Runtime varía inmensamente y es evidente el ver que se compensa la pérdida de precisión con la de la aceleración del cálculo.

Considerando que, si bien, la programación serial ha estado presente desde el inicio de la computación, los problemas del campo de la ciencia y la tecnología cada día requieren de un mayor poder de procesamiento y precisión. Por ello es necesario la busqueda e implementacion tecnologias que coadyuden en las investigaciones.

EL conocer OpenACC y el entender su capacidad de paralelismo comparado con el Serial, así como las características de velocidad de CUDA darán a los investigadores la opción de cualquiera de los tres métodos de acuerdo con sus necesidades científicas.

Para un programador que se encuentra fuera del área del cómputo científico, le será fácil la implementación de OpenACC en los sistemas ya creados, ya que con pocas directivas se le indica al compilador las posibles secciones paralelizables. En cambio CUDA requiere de un conocimiento especializado de la arquitectura de la computadora y de la tarjeta gáfica, así como para el análisis del algoritmo. Tambien es importante recordar que para utilizar el estándar CUDA, es indispensable el contar con las GPUs de la marca NVIDIA, en cambio OpenACC puede utilizar cualquier tarjeta gráfica.

Tabla de Figuras

[Figura 1 Sistema coordenado molecular: i,j=electrones y A, B= núcleos. 11](#_Toc528616132)

[Figura 2 Comparación entre Ecuación de Schrödinger y Ecuación de Khon-Sham. 12](#_Toc528616133)

[Figura 3 Giro de ejes de una matriz simétrica. 16](#_Toc528616134)

[Figura 4 Aceleración de programas en GPUs. 18](#_Toc528616135)

[Figura 5 Comparación entre Multicore y Manycore. 19](#_Toc528616136)

[Figura 6 Matriz de 3 x 3. 20](#_Toc528616137)

[Figura 7 Conversión de matriz a arreglo. 21](#_Toc528616138)

[Figura 8 Salida de Jacobi con matriz de 3 x 3 en C. 24](#_Toc528616139)

[Figura 9 Salida de Jacobi con matriz de 3 x 3 en OpenACC. 28](#_Toc528616140)

[Figura 10 Diagrama de modelo SIMD. 28](#_Toc528616141)

[Figura 11 Arquitectura CUDA. 29](#_Toc528616142)

[Figura 12 Jerarquía LLVM. 30](#_Toc528616143)

[Figura 13 Mapeo de estructuras en CUDA. 31](#_Toc528616144)

[Figura 14 Dimensión de Grids y Blocks en CUDA. 32](#_Toc528616145)

[Figura 15 Pipeline de ejecución de CUDA. 33](#_Toc528616146)

[Figura 16 Partición de datos en CUDA. 33](#_Toc528616147)

[Figura 17 Partición de datos de una matriz en CUDA. 34](#_Toc528616148)

[Figura 18 Transferencia de datos entre Host y Device en CUDA. 35](#_Toc528616149)

[Figura 19 Orden de ID de los threads de CUDA. 35](#_Toc528616150)

[Figura 20 Salida de Jacobi con matriz de 3 x 3 en CUDA. 38](#_Toc528616151)

[Figura 21Gráfica de RUNTIME. 42](#_Toc528616152)

[Figura 22 Gráfica del inicio de la divergencia del RUNTIME. 42](#_Toc528616153)

[Figura 23 Factor de Costo total. 43](#_Toc528616154)

[Figura 24 Factor de Costo hasta matriz de 32 x 32. 43](#_Toc528616155)

[Figura 25 Speed up total. 44](#_Toc528616156)

[Figura 26 Speed up de las primeras matrices. 45](#_Toc528616157)

[Figura 27 Aceleración total. 46](#_Toc528616158)

[Figura 28 Aceleración de las primeras matrices. 46](#_Toc528616159)

[Figura 29 Eficiencia. 47](#_Toc528616160)

[Figura 30 Eficiencia sin los valores del programa en C. 48](#_Toc528616161)

[Figura 31 Matriz de 10 x 10. 48](#_Toc528616162)

[Figura 32 Resultados de Jacobi en Serial de una matriz de 10 x 10. 49](#_Toc528616163)

[Figura 33 Resultados de Jacobi en OpenACC de una matriz de 10 x 10. 50](#_Toc528616164)

[Figura 34 Resultados de Jacobi en CUDA de una matriz de 10 x 10. 51](#_Toc528616165)

Bibliografía

|  |  |
| --- | --- |
| [1] | J. M. Thijssen, Computational Physics, New York: Cambridge University Press, 2007. |
| [2] | «Institute for Theoretical Physicas ETH Zurich,» [En línea]. Available: http://www.itp.phys.ethz.ch/research.html. |
| [3] | [En línea]. Available: https://chemistry.stanford.edu/research/research-areas/theoretical-chemistry. [Último acceso: 22 junio 2018]. |
| [4] | G. W. George Hager, Introduction to High Performance Computing for Scientists and Engineers, T. &. F. Group, Ed., 2011. |
| [5] | [En línea]. Available: https://www.amd.com/es-xl/products/graphics/server/gpu-compute. [Último acceso: 23 junio 23]. |
| [6] | [En línea]. Available: http://www.ub.edu/web/ub/es/recerca\_innovacio/recerca\_a\_la\_UB/instituts/institutspropis/iqtcub.html. [Último acceso: 22 junio 2018]. |
| [7] | [En línea]. Available: https://www.ibm.com/thought-leadership/summit-supercomputer/. [Último acceso: 23 junio 2018]. |
| [8] | [En línea]. Available: https://chemistry.stanford.edu/research/research-areas/physical-chemistry. [Último acceso: 22 junio 2018]. |
| [9] | S. l. e. d. Schrödinger. [En línea]. Available: http://www.ugr.es/~jllopez/Cap3-Sch.pdf. [Último acceso: 14 junio 2018]. |
| [10] | [En línea]. Available: https://www.nucleares.unam.mx/~vieyra/node26.html. [Último acceso: 23 junio 2018]. |
| [11] | P. e. d. K.-S. p. s. f. f. Kohn-Sham. [En línea]. Available: http://ricabib.cab.cnea.gov.ar/519/1/1Benitez\_Moreno.pdf. [Último acceso: 14 junio 2018]. |
| [12] | J. K. Labanoswski, Density Functional Methods in Chemestry, New York: Springer-Velarg, 1991. |
| [13] | D. S. SHOLL, DENSITY FUNCTIONAL THEORY. A Practical Introduction., National Energy Technology Laboratory: Georgia Institute of Technology, 2009. |
| [14] | T. d. f. d. l. densidad electrónica. [En línea]. Available: http://www.fis.cinvestav.mx/~daniel/thELA.pdf. [Último acceso: 12 junio 2018]. |
| [15] | E. Urriolabeitia, «La teoría del funcional de la densidad (DFT) va por el mal camino,» *divulgame.org,* 9 enero 2017. |
| [16] | T. D. F. D. L. DENSIDAD. [En línea]. Available: http://depa.fquim.unam.mx/amyd/archivero/DensityFunctionalTheory\_21556.pdf. [Último acceso: 14 junio 2018]. |
| [17] | [En línea]. Available: http://www.demon-software.com/public\_html/index.html. [Último acceso: 23 junio 2018]. |
| [18] | [En línea]. Available: http://www.demon-software.com/public\_html/program.html#branches. [Último acceso: 23 junio 2018]. |
| [19] | F. J. Ayres, MATRICES, Teoria y problemas resueltos, McGraw-Hill, 1962. |
| [20] | S. K. Shan, Numerical Methods and Computers, Massachusetts: Addison-Wesley, 1966. |
| [21] | [En línea]. Available: http://la.nvidia.com/object/what-is-gpu-computing-la.html. [Último acceso: 23 junio 2018]. |
| [22] | [En línea]. Available: https://www.profesionalreview.com/2017/06/21/diferencia-la-cpu-la-gpu/. [Último acceso: 12 mayo 2018]. |
| [23] | [En línea]. Available: https://www.profesionalreview.com/2017/06/21/diferencia-la-cpu-la-gpu/. [Último acceso: 23 junio 2018]. |
| [24] | W. H. Press, Numerical Recipes in C. The Art of Scientific Computing, United States of America: Cambridge University Pres, 2002. |
| [25] | «OpenACC: El API GPGPU Post-CUDA,» 12 2011. [En línea]. Available: https://www.fayerwayer.com/2011/12/openacc-el-api-gpgpu-post-cuda/. |
| [26] | openacc.org, OpenACC Programming and Best Practices Guide, 2015. |
| [27] | «Monday-Programming-GPUs-OpenACC,» [En línea]. Available: http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0517B-Monday-Programming-GPUs-OpenACC.pdf. |
| [28] | [En línea]. Available: https://es.wikipedia.org/wiki/SIMD. |
| [29] | J. Cheng, Professional CUDA C Programming, Indianapolis: Jhon Wiley & Sons, 2014. |
| [30] | [En línea]. Available: https://www.pelagos-consulting.com/?p=503. |
| [31] | M. A. Barry Wilkinson, Parallel Programming: Techniques and Aplications Using Networked Workstations an Parallel Computers, Segunda Edicion ed., Pearson, 2005. |

CÓDIGO FUENTE

CF 1.1 creaMatrix.c

1. /\*
2. \* creaMatrix.c
3. \*
4. \*
5. \* José Antonio Ayala Barbosa
6. \* Oct, 2018
7. \*
8. \*
9. \*/
11. #include <stdio.h>
12. #include <string.h>
13. #include <stdlib.h>
15. #define NP 16
16. int main(int argc, char const \*argv[]){
18. float m[NP][NP];
19. int i,j,k;
20. char nombreArchivo[100];
21. FILE \*archivo;
22. char tempChar[100];
24. for ( i = 0; i < NP; ++i)
25. {
26. for (j = i,k=NP/2; j < NP; ++j, k--)
27. {
28. m[i][j]=m[j][i]= k;
29. }
30. }
32. [strcpy](http://www.opengroup.org/onlinepubs/009695399/functions/strcpy.html)(nombreArchivo, "matrix");
33. [sprintf](http://www.opengroup.org/onlinepubs/009695399/functions/sprintf.html)(tempChar, "%d", NP);
34. [strcat](http://www.opengroup.org/onlinepubs/009695399/functions/strcat.html)(nombreArchivo, tempChar);
35. [strcat](http://www.opengroup.org/onlinepubs/009695399/functions/strcat.html)(nombreArchivo, ".txt");
37. if ([fopen](http://www.opengroup.org/onlinepubs/009695399/functions/fopen.html)(nombreArchivo, "r") == NULL){ // Si no existe archivo crea un archivo con los numeros
38. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Creating File**\n**");
39. archivo = [fopen](http://www.opengroup.org/onlinepubs/009695399/functions/fopen.html)(nombreArchivo, "w"); // abre archivo de escritura
40. [fprintf](http://www.opengroup.org/onlinepubs/009695399/functions/fprintf.html)(archivo,"%d",NP);
41. [fprintf](http://www.opengroup.org/onlinepubs/009695399/functions/fprintf.html)(archivo,"%c",'**\n**');
43. for (i = 0; i < NP; i++){
44. for (j = 0; j < NP; j++){
46. [fprintf](http://www.opengroup.org/onlinepubs/009695399/functions/fprintf.html)(archivo,"%.1f ",m[i][j]); //Escribe numeros aleatorios
47. }
48. [fprintf](http://www.opengroup.org/onlinepubs/009695399/functions/fprintf.html)(archivo,"%c",'**\n**');
49. }
50. [fclose](http://www.opengroup.org/onlinepubs/009695399/functions/fclose.html)(archivo); // Cierra el archivo
52. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("File created**\n**");
53. }
55. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**");
56. return 0;
57. }

CF 1.2 auxFuncs.h

1. /\*
2. \* auxFuncs.h
3. \* Auxiliary functions for timing and allocating data.
4. \*
5. \* José Antonio Ayala Barbosa
6. \* Oct, 2018
7. \*
8. \*
9. \*/
11. #include <stdio.h>
12. #include <stddef.h>
13. #include <stdlib.h>
14. #include <sys/time.h>
16. void nrerror(char error\_text[]){
17. /\* Numerical Recipes standard error handler \*/
18. [fprintf](http://www.opengroup.org/onlinepubs/009695399/functions/fprintf.html)(stderr,"Numerical Recipes run-time error...**\n**");
19. [fprintf](http://www.opengroup.org/onlinepubs/009695399/functions/fprintf.html)(stderr,"%s**\n**",error\_text);
20. [fprintf](http://www.opengroup.org/onlinepubs/009695399/functions/fprintf.html)(stderr,"...now exiting to system...**\n**");
21. [exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(1);
22. }
24. void get\_walltime\_(double\* wcTime) {
25. struct timeval tp;
26. gettimeofday(&tp, NULL);
27. \*wcTime = (double)(tp.tv\_sec + tp.tv\_usec/1000000.0);
28. }
30. void get\_walltime(double\* wcTime) {
31. get\_walltime\_(wcTime);
32. }
34. float \*vectorF(int n){
35. float \*v;
36. v=(float \*) [malloc](http://www.opengroup.org/onlinepubs/009695399/functions/malloc.html)((size\_t)(n\*sizeof(float)));
37. if (!v) nrerror("allocation failure in vector()");
38. return v;
39. }
41. int \*vectorI(int n){
42. int \*v;
43. v=(int \*) [malloc](http://www.opengroup.org/onlinepubs/009695399/functions/malloc.html)((size\_t)(n\*sizeof(int)));
44. if (!v) nrerror("allocation failure in vector()");
45. return v;
46. }

CF 1.3 driverA.c

1. /\*
2. \* driverA.c
3. \* Driver for function JACOBI.
4. \*
5. \* José Antonio Ayala Barbosa
6. \* Oct, 2018
7. \*
8. \*
9. \*/
11. #include <stdio.h>
12. #include <sys/time.h>
13. #include <stdlib.h>
14. #include "auxFuncs.h"
15. #include "jacobiA.h"
17. int main(int argc, char \*\*argv)
18. {
20. int n;
21. char \*nombreArchivo=argv[1];
22. double S,E;
23. int i, j, k, nrot=0;
24. FILE \*archivo;
25. float \*c,\*mat,\*eval,\*evec;
27. if ([fopen](http://www.opengroup.org/onlinepubs/009695399/functions/fopen.html)(nombreArchivo, "r") == NULL){
28. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("File not found**\n**");
29. return 1;
30. }else{
31. archivo = [fopen](http://www.opengroup.org/onlinepubs/009695399/functions/fopen.html)(nombreArchivo, "r");
32. [fscanf](http://www.opengroup.org/onlinepubs/009695399/functions/fscanf.html)(archivo, "%d", &n);
33. c = vectorF(n\*n);
34. for (i = 0; i < n; ++i){
35. for (j = 0; j < n; ++j){
36. [fscanf](http://www.opengroup.org/onlinepubs/009695399/functions/fscanf.html)(archivo, "%f", &c[i\*n+j]);
37. }
38. }
39. [fclose](http://www.opengroup.org/onlinepubs/009695399/functions/fclose.html)(archivo);
40. }
42. mat=vectorF(n\*n);
43. evec=vectorF(n\*n);
44. eval=vectorF(n);
46. for (i = 0; i < n; ++i)
47. for (j = 0; j < n; ++j)
48. mat[i\*n+j]=c[i\*n+j];
50. for (i = 0; i < n; i++){
51. for (j = 0; j < n; j++)
52. evec[i\*n+j] = 0.0;
53. evec[i\*n+i] = 1.0;
54. }
56. for (i = 0; i < n; ++i){
57. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**");
58. for (j = 0; j < n; ++j)
59. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("[%f]",mat[i\*n+j]);
60. }
62. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**\*\*\*\*\*\* Finding Eigenvalues \*\*\*\*\*\***\n**");
63. get\_walltime(&S);
65. jacobiMultip(mat,n,evec,eval,&nrot);
67. get\_walltime(&E);
69. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**\*\*\*\*\*\*  Eigenvalues \*\*\*\*\*\***\n**");
70. for (i = 0; i < n; ++i){
71. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**");
72. for (j = 0; j < n; ++j)
73. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("[%f]",mat[i\*n+j]);
74. }
75. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**");
76. for (i = 0; i < n; ++i){
77. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("eigenvalue %3d, = %12.6f**\n**",i+1,eval[i]);
78. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("eigenvector:**\n**");
79. for (j=0,k=1;j<n; j++,k++) {
80. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("%12.6f",evec[i\*n+j]);
81. if ((k%5) == 0) [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**");
82. }
83. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**");
84. }
86. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Rotations: %d**\n**",nrot );
87. ("Total time:%f sec**\n**", E-S);
88. free(c);
89. free(mat);
90. free(evec);
91. free(eval);
93. return 0;
94. }

CF 1.4 jacobiA.h

1. #include <stdio.h>
2. #include <math.h>
3. #include <stdlib.h>
4. #include <stdbool.h>
5. #include <math.h>
6. #include"auxFuncs.h"
8. void max\_elem(int \*piv\_elem,int n,float \*mat){
9. int r,c;
10. int max\_i = 0;                               //first coordenate i
11. int max\_j = 1;                               //first coordenate j
13. #pragma acc parallel num\_gangs(32), vector\_length(64)
14. {
15. #pragma acc loop
16. for (r = 0; r < n-1; r++)
17. for (c = r+1; c < n; c++)
18. if([fabs](http://www.opengroup.org/onlinepubs/009695399/functions/fabs.html)(mat[r\*n+c]) > [fabs](http://www.opengroup.org/onlinepubs/009695399/functions/fabs.html)(mat[max\_i\*n+max\_j])){ //if exists a higher element
19. max\_i = r;                          //replace new coor
20. max\_j = c;
21. }
22. }
23. piv\_elem[0] = max\_i;                           //store new coordenates
24. piv\_elem[1] = max\_j;
26. }
28. float cal\_tan(int max\_i,int max\_j,float \*mat, int n){
29. float num;
30. float den;
31. float a1;
32. float a2;
33. float a3;
35. num = 2 \* (mat[max\_i\*n+max\_j]);
36. if(mat[max\_i\*n+max\_i] < mat[max\_i\*n+max\_i])
37. num = -num;
39. a1 = mat[max\_i\*n+max\_i] - mat[max\_j\*n+max\_j];
40. a2 = a1\*a1;
41. a3 = 4 \* mat[max\_i\*n+max\_j]\*mat[max\_i\*n+max\_j];
42. den = a2 + a3;
43. den = [sqrt](http://www.opengroup.org/onlinepubs/009695399/functions/sqrt.html)(den);
44. den = [abs](http://www.opengroup.org/onlinepubs/009695399/functions/abs.html)(a1) + den;
45. return num/den;
46. }
48. float cal\_cos(float tang){                //cos = 1/√(1+tan^2)
49. float cose;
50. cose = 1 + (tang \* tang);
51. cose = [sqrt](http://www.opengroup.org/onlinepubs/009695399/functions/sqrt.html)(cose);
52. cose = 1 / cose;
53. return cose;
54. }
56. float cal\_sin(float cose, float tang){       //sin = cos\*tan
57. float sino;
58. sino = cose\*tang;
59. return sino;
60. }
62. void new\_T\_mat(int max\_i, int max\_j,int n,float \*mat,float \*T,float \*mat\_temp){
63. float tang, cose, sino;
64. int c,r;
66. tang = cal\_tan(max\_i,max\_j,mat,n);
67. cose = cal\_cos(tang);
68. sino = cal\_sin(cose,tang);
70. for (r = 0; r < n; r++){            //Generate identity matrix
71. for (c = 0; c < n; c++)
72. T[r\*n+c] = 0.0;
73. T[r\*n+r] = 1.0;
74. }
75. //T Rotating matrix
76. T[max\_i\*n+max\_i] = cose;
77. T[max\_j\*n+max\_j] = cose;
78. T[max\_i\*n+max\_j] = -sino;             //Element to eliminate
79. T[max\_j\*n+max\_i] = sino;
81. }
83. void mat\_mult(int n,float \*A, float \*B,float \*C){
84. int i,j,k;
85. #pragma acc parallel num\_gangs(32), vector\_length(64)
86. {
87. #pragma acc loop
88. for (i = 0 ; i < n ; i++ ){
89. for (j = 0 ; j < n ; j++ ){
90. C[i\*n+j] = 0.0;
91. #pragma acc loop
92. for (k = 0 ; k < n ; k++ ){
93. C[i\*n+j] += A[i\*n+k] \* B[k\*n+j];
94. }
95. }
96. }
97. }
98. }
100. void mat\_mult\_tra(int n,float \*A, float \*B,float \*C){
101. int i,j,k;
102. #pragma acc parallel num\_gangs(32), vector\_length(64)
103. {
104. #pragma acc loop
105. for (i = 0 ; i < n ; i++ ){
106. for (j = 0 ; j < n ; j++ ){
107. C[i\*n+j] = 0.0;
108. #pragma acc loop
109. for (k = 0 ; k < n ; k++ ){
110. C[i\*n+j] += A[k\*n+i] \* B[k\*n+j];
111. }
112. }
113. }
114. }
115. }
117. void copy\_mat(int n,float \*A, float \*B){
118. int i,j;
119. #pragma acc parallel num\_gangs(32), vector\_length(64)
120. {
121. #pragma acc loop
122. for (i = 0; i < n; ++i)
123. for (j = 0; j < n; ++j)
124. B[i\*n+j]=A[i\*n+j];
125. }
126. }

129. void jacobiMultip (float \*mat, int n,float \*eigvec, float \*eigval,  int \*nrot){
131. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*+\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
132. //On input
133. //mat: Contains the matrix to be diagonalized.
134. //n: Order of matrix a.
136. //On output
137. //eigvec: eigenvectors to be computed v
138. //eigval: Contains the eigenvalues in ascending order
139. //nrot: Number of Jacobi rotations.
140. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*+\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
142. int i,j;
143. int \*piv\_elem;             //Keep coordenates of an elemnt i,j
144. bool min = **false**;
145. float EPS = .0000001;
146. float \*T;                  //Contains the ratation matrix
147. float \*mat\_temp;           //A temporal matrix
149. mat\_temp = vectorF(n\*n);
150. T = vectorF(n\*n);
151. piv\_elem = vectorI(2);
153. for (\*nrot = 0; min == **false** ; ++\*nrot){
154. max\_elem(piv\_elem,n,mat);  //Search for max element in tringle up
156. if([fabs](http://www.opengroup.org/onlinepubs/009695399/functions/fabs.html)(mat[piv\_elem[0]\*n+piv\_elem[1]]) < EPS || \*nrot >= 50000 ) //if max element doesnt exist more
157. min=**true**;
159. else{
160. new\_T\_mat(piv\_elem[0],piv\_elem[1],n,mat,T,mat\_temp); //Calculate T
161. mat\_mult(n,eigvec,T,mat\_temp);               //Eigenvect
162. copy\_mat(n,mat\_temp,eigvec);
163. mat\_mult\_tra(n,T,mat,mat\_temp);
164. mat\_mult(n,mat\_temp,T,mat);
165. }
167. for (i = 0; i < n; ++i)
168. eigval[i]=mat[i\*n+i];
169. }
170. }

CF 1.5 driverA.cu

1. /\*
2. \* driverA.cu
3. \* Driver for function JACOBI in CUDA.
4. \*
5. \* José Antonio Ayala Barbosa
6. \* Oct, 2018
7. \*
8. \*
9. \*/
11. #include <stdio.h>
12. #include <sys/time.h>
13. #include <stdlib.h>
14. #include "auxFuncs.h"
15. #include "jacobiACUDA.h"
17. int main(int argc, char \*\*argv)
18. {
20. int n;
21. char \*nombreArchivo=argv[1];
22. double S,E;
23. int i, j, k, nrot=0;
24. FILE \*archivo;
25. float \*c,\*mat,\*eval,\*evec;
27. if ([fopen](http://www.opengroup.org/onlinepubs/009695399/functions/fopen.html)(nombreArchivo, "r") == NULL){
28. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("File not found**\n**");
29. return 1;
30. }else{
31. archivo = [fopen](http://www.opengroup.org/onlinepubs/009695399/functions/fopen.html)(nombreArchivo, "r");
32. [fscanf](http://www.opengroup.org/onlinepubs/009695399/functions/fscanf.html)(archivo, "%d", &n);
33. c = vectorF(n\*n);
34. for (i = 0; i < n; ++i){
35. for (j = 0; j < n; ++j){
36. [fscanf](http://www.opengroup.org/onlinepubs/009695399/functions/fscanf.html)(archivo, "%f", &c[i\*n+j]);
37. }
38. }
39. [fclose](http://www.opengroup.org/onlinepubs/009695399/functions/fclose.html)(archivo);
40. }
42. mat=vectorF(n\*n);
43. evec=vectorF(n\*n);
44. eval=vectorF(n);
46. for (i = 0; i < n; ++i)
47. for (j = 0; j < n; ++j)
48. mat[i\*n+j]=c[i\*n+j];
50. for (i = 0; i < n; i++){
51. for (j = 0; j < n; j++)
52. evec[i\*n+j] = 0.0;
53. evec[i\*n+i] = 1.0;
54. }
56. for (i = 0; i < n; ++i){
57. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**");
58. for (j = 0; j < n; ++j)
59. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("[%f]",mat[i\*n+j]);
60. }
62. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**\*\*\*\*\*\* Finding Eigenvalues \*\*\*\*\*\***\n**");
63. get\_walltime(&S);
65. jacobiMultip(mat,n,evec,eval,&nrot);
67. get\_walltime(&E);
69. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**\*\*\*\*\*\*  Eigenvalues \*\*\*\*\*\***\n**");
70. for (i = 0; i < n; ++i){
71. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**");
72. for (j = 0; j < n; ++j)
73. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("[%f]",mat[i\*n+j]);
74. }
75. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**");
76. for (i = 0; i < n; ++i){
77. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("eigenvalue %3d, = %12.6f**\n**",i+1,eval[i]);
78. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("eigenvector:**\n**");
79. for (j=0,k=1;j<n; j++,k++) {
80. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("%12.6f",evec[i\*n+j]);
81. if ((k%5) == 0) [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**");
82. }
83. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**");
84. }
86. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Rotations: %d**\n**",nrot );
88. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Total time:%f sec**\n**", E-S);
89. free(c);
90. free(mat);
91. free(evec);
92. free(eval);
93. return 0;
94. }

CF 1.6 jacobiACUDA.h

1. /\*
2. \* jacobiACUDA.h
3. \*
4. \*
5. \* José Antonio Ayala Barbosa
6. \* Oct, 2018
7. \*
8. \*
9. \*/
11. #include <cuda.h>
12. #include <stdio.h>
13. #include <math.h>
14. #include <stdlib.h>
15. #include <stdbool.h>
16. #include <math.h>
17. #include "auxFuncs.h"
19. \_\_global\_\_ void kernel\_max\_elem(int \*piv\_elem,int n,float \*mat){
20. int r,c;
21. int max\_i = 0;                     //first coordenate i
22. int max\_j = 1;                        //first coordenate j
24. for (r = 0; r < n-1; r++)
25. for (c = r+1; c < n; c++)
26. if([fabs](http://www.opengroup.org/onlinepubs/009695399/functions/fabs.html)(mat[r\*n+c]) > [fabs](http://www.opengroup.org/onlinepubs/009695399/functions/fabs.html)(mat[max\_i\*n+max\_j])){ //if exists a higher element
27. max\_i = r;               //replace new coor
28. max\_j = c;
29. }
30. piv\_elem[0] = max\_i;              //store new coordenates
31. piv\_elem[1] = max\_j;
33. }
35. \_\_global\_\_ void kernel\_set\_max\_elem(float \*mat,int n, int \*piv\_elem,float \*max\_elem){
36. \*max\_elem=mat[piv\_elem[0]\*n+piv\_elem[1]];
37. }
39. \_\_device\_\_ float cal\_tan(int max\_i,int max\_j,float \*mat, int n){
40. float num;
41. float den;
42. float a1;
43. float a2;
44. float a3;
46. num = 2 \* (mat[max\_i\*n+max\_j]);
47. if(mat[max\_i\*n+max\_i] < mat[max\_i\*n+max\_i])
48. num = -num;
50. a1 = mat[max\_i\*n+max\_i] - mat[max\_j\*n+max\_j];
51. a2 = a1\*a1;
52. a3 = 4 \* mat[max\_i\*n+max\_j]\*mat[max\_i\*n+max\_j];
53. den = a2 + a3;
54. den = [sqrt](http://www.opengroup.org/onlinepubs/009695399/functions/sqrt.html)(den);
55. den = [abs](http://www.opengroup.org/onlinepubs/009695399/functions/abs.html)(a1) + den;
56. return num/den;
57. }
59. \_\_device\_\_ float cal\_cos(float tang){   //cos = 1/√(1+tan^2)
60. float cose;
61. cose = 1 + (tang \* tang);
62. cose = [sqrt](http://www.opengroup.org/onlinepubs/009695399/functions/sqrt.html)(cose);
63. cose = 1 / cose;
64. return cose;
65. }
67. \_\_device\_\_ float cal\_sin(float cose, float tang){        //sin = cos\*tan
68. float sino;
69. sino = cose\*tang;
70. return sino;
71. }
73. \_\_global\_\_ void kernel\_new\_T\_mat(int \*piv\_elem,int n,float \*mat,float \*T){
74. float tang, cose, sino;
75. int c,r;
76. int max\_i = piv\_elem[0];
77. int max\_j = piv\_elem[1];
79. int row = blockIdx.y \* blockDim.y + threadIdx.y;
80. int col = blockIdx.x \* blockDim.x + threadIdx.x;
82. if(row >= n|| col >= n)
83. return;
85. for (row = row; row < n; ++row)
86. T[row\*n+col] = 0.0;
87. T[col\*n+col]=1.0;
89. if(max\_j==col){
90. tang = cal\_tan(max\_i,max\_j,mat,n);
91. cose = cal\_cos(tang);
92. sino = cal\_sin(cose,tang);
94. T[max\_i\*n+max\_i] = cose;
95. T[max\_j\*n+max\_j] = cose;
96. T[max\_i\*n+max\_j] = -sino;      //Element to eliminate
97. T[max\_j\*n+max\_i] = sino;
98. }
99. }
101. \_\_global\_\_ void kernel\_mat\_mult(int n,float \*A, float \*B,float \*C){
103. float Cvalue;
104. int row = blockIdx.y \* blockDim.y + threadIdx.y;
105. int col = blockIdx.x \* blockDim.x + threadIdx.x;
106. int e;
108. if(row >= n|| col >= n)
109. return;
111. for (row = row; row < n; ++row){
112. Cvalue = 0.0;
113. for (e = 0; e < n; e++)
114. Cvalue += (A[row \* n + e]) \* (B[e \* n + col]);
115. C[row \* n + col] = Cvalue;
116. }
117. }
119. \_\_global\_\_ void kernel\_mat\_mult\_tra(int n,float \*A, float \*B,float \*C){
121. float Cvalue;
122. int row = blockIdx.y \* blockDim.y + threadIdx.y;
123. int col = blockIdx.x \* blockDim.x + threadIdx.x;
124. int e,i;
126. if(row >= n|| col >= n)
127. return;
129. for (i = row; i < n; i++){
130. Cvalue = 0.0;
131. for (e = 0; e < n; e++)
132. Cvalue += (A[e \* n + i]) \* (B[e \* n + col]);
133. C[i \* n + col] = Cvalue;
134. }
135. }
137. \_\_global\_\_ void kernel\_copy\_mat(int n,float \*A, float \*B){
139. int row = blockIdx.y \* blockDim.y + threadIdx.y;
140. int col = blockIdx.x \* blockDim.x + threadIdx.x;
141. int e;
143. if(row >= n|| col >= n)
144. return;
146. for (e = 0; e < n; e++)
147. B[col\*n+e]=A[col\*n+e];
148. }
150. \_\_global\_\_ void kernel\_copy\_diag(int n,float \*A, float \*B){
152. int row = blockIdx.y \* blockDim.y + threadIdx.y;
153. int col = blockIdx.x \* blockDim.x + threadIdx.x;
155. if(row >= n|| col >= n)
156. return;
158. B[col]=A[col\*n+col];
159. }
161. void jacobiMultip (float \*mat, int n,float \*eigvec, float \*eigval,  int \*nrot){
163. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
164. //On input
165. //mat: Contains the matrix to be diagonalized.
166. //n: Order of matrix a.
168. //On output
169. //eigvec: eigenvectors to be computed v
170. //eigval: Contains the eigenvalues in ascending order
171. //nrot: Number of Jacobi rotations.
172. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
173. int i,j;
174. bool min = **false**;
175. float EPS = .0000001;
176. float \*max\_elem;
177. float max\_e;
179. float \*d\_mat;
180. float \*d\_eigvec;
181. float \*d\_eigval;
182. float \*d\_T;
183. float \*d\_mat\_temp;
184. int \*d\_piv\_elem;
185. float \*d\_max\_elem;
187. size\_t size = (n+1) \* (n+1) \* sizeof(float);
188. dim3 dimGrid( 32 );         // 32 x 1 x 1
189. dim3 dimBlock( 64 );       // 64 x 1 x 1
191. cudaMalloc(&d\_mat, size);
192. cudaMalloc(&d\_eigvec, size);
193. cudaMalloc(&d\_eigval, n\*sizeof (int));
194. cudaMalloc(&d\_T, size);
195. cudaMalloc(&d\_mat\_temp, size);
196. cudaMalloc(&d\_piv\_elem, 2\*sizeof (int));
197. cudaMalloc(&d\_max\_elem,sizeof (float));
199. cudaMemcpy(d\_mat, mat, size, cudaMemcpyHostToDevice);
200. cudaMemcpy(d\_eigvec, eigvec, size, cudaMemcpyHostToDevice);
201. cudaMemcpy(d\_eigval, eigval, n\*sizeof (int), cudaMemcpyHostToDevice);
203. for (\*nrot = 0; min == **false** ; ++\*nrot){
205. kernel\_max\_elem<<<1,1>>>(d\_piv\_elem,n,d\_mat);   //Search for max element in upper tringle
206. kernel\_set\_max\_elem<<<1,1>>>(d\_mat,n,d\_piv\_elem,d\_max\_elem);
207. cudaMemcpy(&max\_e, d\_max\_elem, sizeof(float), cudaMemcpyDeviceToHost);
209. if([fabs](http://www.opengroup.org/onlinepubs/009695399/functions/fabs.html)(max\_e) < EPS || \*nrot >= 50000 ) //if max element doesnt exist more
210. min=**true**;
212. else{
213. kernel\_new\_T\_mat<<<dimGrid, dimBlock>>>(d\_piv\_elem,n,d\_mat,d\_T);
214. kernel\_mat\_mult<<<dimGrid, dimBlock>>>(n,d\_eigvec,d\_T,d\_mat\_temp);
215. kernel\_copy\_mat<<<dimGrid, dimBlock>>>(n,d\_mat\_temp,d\_eigvec);
216. kernel\_mat\_mult\_tra<<<dimGrid, dimBlock>>>(n,d\_T,d\_mat,d\_mat\_temp);
217. kernel\_mat\_mult<<<dimGrid, dimBlock>>>(n,d\_mat\_temp,d\_T,d\_mat);
218. }
219. }
221. kernel\_copy\_diag<<<dimGrid, dimBlock>>>(n,d\_mat,d\_eigval);
223. cudaMemcpy(mat, d\_mat, size, cudaMemcpyDeviceToHost);
224. cudaMemcpy(eigvec, d\_eigvec, size, cudaMemcpyDeviceToHost);
225. cudaMemcpy(eigval,d\_eigval,n\*sizeof(float),cudaMemcpyDeviceToHost);
227. cudaFree(d\_mat);
228. cudaFree(d\_eigvec);
229. cudaFree(d\_eigval);
230. cudaFree(d\_T);
231. cudaFree(d\_mat\_temp);
232. cudaFree(d\_piv\_elem);
233. cudaFree(d\_max\_elem);
234. }

1. Alain St-Amant, Ph.D. Thesis, Université de Montréal, 1992. [↑](#footnote-ref-1)
2. LLVM (Low Level Virtual Machine) es una plataforma para desarrollar compiladores, que está diseñada para optimizar el tiempo de compilación, el tiempo de enlazado y el tiempo de ejecución en cualquier lenguaje de programación. [↑](#footnote-ref-2)