



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

INSTITUTO DE INVESTIGACIONES EN MATEMÁTICAS
APLICADAS Y EN SISTEMAS

POSGRADO EN CIENCIA E INGENIERIA
DE LA COMPUTACIÓN

**RESOLUCIÓN DE LA ECUACION DE
KHON-SHAM EN TARJETAS GRÁFICAS
CON CUDA**

T E S I N A

PARA OBTENER EL GRADO DE:

**ESPECIALISTA EN CÓMPUTO DE
ALTO RENDIMIENTO**

PRESENTA

JOSÉ ANTONIO AYALA BARBOSA

DIRECTOR DE TESINA

DR. JOSÉ JESÚS CARLOS QUINTANAR SIERRA

Agradecimientos

El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi.

Índice

Agradecimientos	3
Índice	5
Introducción	8
Capítulo 1. Marco Teórico	10
1.1 Generealidades	10
1.1.1 Proceso	10
1.1.2 Estado	10
1.1.3 Memoria	10
1.1.4 Ejecución	10
1.1.5 Programa	10
1.2 CPU	10
1.3 GPU	11
1.3.1 Tipos de GPU	12
1.4 Memoria	13
1.4.1 Tipos de almacenamiento	13
1.4.2 Arreglos de memoria	14
Capitulo 2. Cómputo de Alto Rendimiento	15
2.1 High Performance Computing	15
2.2 Surgimiento del Multicore	15
2.3 Ejecución de instrucciones	15
2.4 Taxonomía de Flynn	16
2.4.1 SISD: Single Instruction, Single Data	16
2.4.2 SIMD: Single Instruction, Multiple Data	16
2.4.3 MISD: Multiple Instruction, Single Data	17
2.4.4 MIMD: Multiple Instruction, Multiple Data	17
2.5 Arreglo de procesadores	18
2.5.1 Procesador físico	18
2.5.2 Socket	18
2.5.3 Core o Núcleo	18
2.5.4 Procesador virtual:	18
2.5.5 Thread	18
2.5.6 Variable compartida	18
2.6 Topología de Procesos	19
2.7 Computación Paralela	19

2.7.1 Programación secuencial y paralela	20
2.7.2 Principales limitaciones del paralelismo	21
2.7.3 Métricas de desempeño.....	22
2.8 Cómputo en GPU	24
2.8.1 Diferencia entre CPU y GPU	25
2.8.2 Cómputo Heterogéneo.....	26
2.9 Tecnología de Gráficos.....	27
2.9.1 C/C++, Fortran, Python	28
2.9.2 Modelado molecular en GPU	28
2.10 CUDA.....	30
2.10.1 Arquitectura CUDA.....	30
2.10.2 Estructura CUDA.....	31
2.10.3 Ubicación de Threads	32
2.10.4 Kernels.....	32
2.10.5 Manejo de memoria	33
Capítulo 4. Fisicoquímica.....	36
4.1 Química	36
4.1.1 Átomo.....	36
4.1.2 Elemento	36
4.1.3 Molécula.....	36
4.1.4 Fisicoquímica	36
4.2 Física	37
4.2.1 Mecánica Clásica	37
4.2.2 Mecánica Cuántica	37
4.2.3 Constante de Planck	37
4.2.4 Principio de incertidumbre.....	38
4.2.6 Función de onda	38
4.3 Ecuación de Schrödinger.....	38
4.4 Teoría del Funcional Densidad (DFT)	39
4.4.1 Ecuación de Kohn-Sham	40
Capítulo 4. Algoritmo de Jacobi.....	42
4.1 Forma cuadrática	42
4.2 Eigensistemas.....	42
4.2.1 Transformación de coordenadas.....	43
4.2.2 Eigenvalores	44
4.2.3 Eigenvector.....	45
Capítulo 5. Implementación	47
Capítulo 6. Pruebas y resultados	48
Conclusiones	49

Tabla de ilustraciones.....	50
Bibliografía	51
Anexos.....	53

Introducción

La computación como herramienta que auxilia a la ciencia

Durante las últimas décadas ha incrementado el interés en la computación paralela, el principal objetivo de ella es el incrementar la velocidad de computación.

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 concurrentemente.

Actualmente el paralelismo se ha convertido en la corriente que todo el mundo de la programación está 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 cuando existen muchos datos que pueden ser operados al mismo tiempo, por ello es mejor distribuirlos entre los diversos procesadores.

La Teoría del Funcional Densidad (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 su lugar, la DFT calcula las propiedades electrónicas a partir de la densidad tridimensional de las nubes electrónicas del sistema. Esta simplificación ha ayudado a poner los cálculos cuánticos en manos de un gran número de investigadores, no sólo de teóricos puros y duros. 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 su lugar, la DFT calcula las propiedades electrónicas a partir de la densidad tridimensional de las nubes electrónicas del sistema. Esta simplificación ha ayudado a poner los cálculos cuánticos en manos de un gran número de investigadores, no sólo de teóricos puros y duros.

Como ya se observó, en la mecánica cuántica se requiere de un gran poder de cómputo.

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 la GPU para el cálculo, como se ha hecho tradicionalmente en la CPU.

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. La potencia de las GPUs se ha hecho

presente con sistemas computacionales relacionadas con el modelado de átomos en moléculas.

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

Capítulo 1. Marco Teórico

1.1 Generealidades

Para entrar en contexto, es necesario mencionar algunos conceptos generales sobre cuales son las bases de la computación

1.1.1 Proceso

Cambio de estado de la memoria por acción del procesador.

1.1.2 Estado

Valor instantáneo de una variable.

1.1.3 Memoria

Dispositivo que almacena datos.

1.1.4 Ejecución

Generación de señales digitales que realizan una instrucción.

1.1.5 Programa

Especificación de uno o varios procesos.

1.2 CPU

CPU son las siglas en ingles de Central Processing Unit (Unidad de Proceso Central). Es el componente que procesa todas las instrucciones y datos del software y el hardware motivo por el cual constituye el elemento más importante de la computadora.

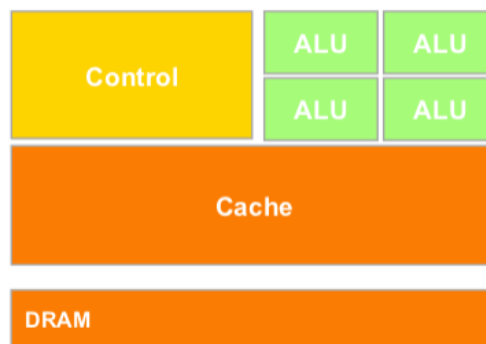
El funcionamiento de la CPU se basa en cuatro sencillos pasos: lectura, decodificación, ejecución y escritura. Está conformada principalmente por los siguientes componentes:

- Unidad aritmético-lógica (ALU): realiza, como su nombre lo indica, operaciones aritméticas y lógicas. Es el motor de cálculo que recibe un código y escoge la operación requerida para decodificarlo.
- Unidad de control: recibe los datos de entrada e instrucciones desde la memoria, ejecuta y envía la información una vez procesada.
- Caché: almacena datos temporalmente.
- Bus de direcciones: encargada de enviar las direcciones a la memoria y los periféricos para dar a conocer dónde se debe escribir o leer un dato.

- Bus de datos: permite que el procesador envíe o reciba datos de la memoria y periféricos.
- Registro de instrucción: almacena la instrucción que se está llevando a cabo en cada momento.
- Contador de programa: contiene la dirección de memoria que será la próxima instrucción que deberá ejecutarse.
- Registro de direcciones de memoria: almacena la siguiente dirección de memoria en que se va a leer o escribir un dato.

El CPU es un procesador de propósito general, lo que significa que puede hacer cualquier tipo de calculo, pero esta diseñada para realizar el cómputo en serie, o sea paso a paso. Aunque se pueden utilizar bibliotecas para realizar programación concurrente ya que el hardware *per se* no tiene esa implementación. Por otro lado, se requiere que una computadora tenga al menos dos CPU para realizar tareas paralelas.

La arquitectura, en términos generales, se puede representar como:

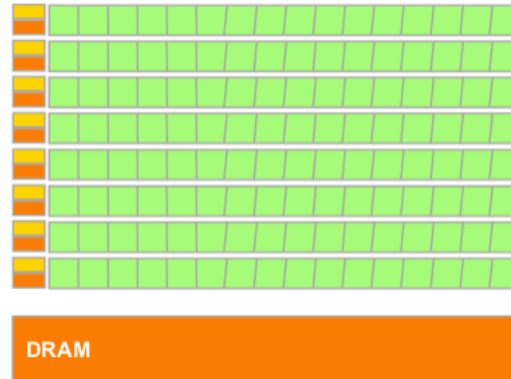


1.3 GPU

La GPU o Graphic Processing Unit (Unidad de Proceso Central), es un coprocesador muy parecido al CPU, solo que el tipo de procesamiento al que se dedica es al de gráficos. De modo que la carga de este tipo de operaciones puede redirigirse del CPU al GPU para aligerar la carga de información y poder dedicarse a realizar otras tareas.

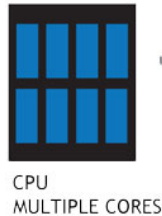
La diferencia entre ambos procesadores radica en la arquitectura de cada uno. Aunque están diseñados para funcionar de modo muy similar, las GPU están construidas para procesar miles de pequeñas operaciones como los gráficos requieran.

La arquitectura, en términos generales, se puede representar como:



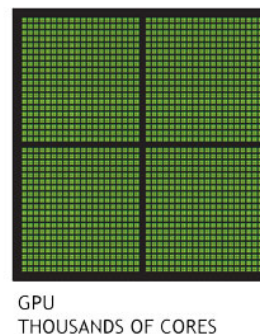
El CPU para operaciones secuenciales en cambio, Las GPU son diseñadas como computadoras especializadas para el cómputo numérico.

La diferencia mayor entre la CPU está en su filosofía de diseño:



Multicores: Mejorar el desempeño de soluciones secuenciales.

Manycores: Optimizar el throughput¹ de muchos threads ejecutados en paralelo.



1.3.1 Tipos de GPU

Actualmente existen tres grandes tipos de unidades de procesamiento gráfico. Más que por la arquitectura, estos difieren entre si por el modo en que son implementadas en términos de Hardware:

1.3.1.1 Tarjetas dedicadas

Este tipo de unidades gráficas son las que proporcionan mayor potencia. Como su nombre lo indica, tienen una serie de especificaciones y están expresamente diseñadas para cumplir

¹ Tasa de transferencia efectiva, es el volumen de trabajo o de información neto que fluye a través de un sistema.

con sus tareas específicas. Generalmente se suele entender que una tarjeta dedicada es aquella que se integra a la tarjeta madre mediante un puerto aparte y lo más importante, posee una memoria RAM independiente que solo podrá ser utilizada por el GPU.

1.3.1.2 GPU integrados

A diferencia de las unidades dedicadas, las integradas utilizan la memoria del sistema para realizar sus funciones. Este tipo de integrados son los más comunes en las computadoras modernas. El núcleo central de estas unidades puede estar en la tarjeta madre, pero recientemente las cosas han cambiado, y tanto los fabricantes como AMD o Intel suelen integrarlas dentro sus procesadores, dichas tecnologías han sido denominadas como AMD Accelerated Processing Unit e Intel HD Graphics respectivamente.

1.3.1.3 Tarjetas híbridas

Diseñadas para mantener precios contenidos y al mismo tiempo aseguran niveles de potencia adecuados. En esta clasificación de unidades gráficas híbridas también comparten la memoria del sistema, pero para disminuir el tiempo de latencia de esta última, integran una cantidad limitada de memoria propia que se encarga de realizar las labores inmediatas.

1.4 Memoria

La memoria es el dispositivo que almacena los datos por un periodo de tiempo. La velocidad y rendimiento de su sistema dependen de la cantidad de memoria que esté instalada en su computadora.

1.4.1 Tipos de almacenamiento

1.4.1.1 Almacenamiento primario

El almacenamiento primario es el que está directamente conectada a la CPU de la computadora y debe estar presente para que este procesador efectúe cualquier función.

Las computadoras actuales utilizan la Random Access Memory (RAM), o Memoria de Acceso Aleatorio, esta memoria primaria contiene los programas en ejecución y los datos con que operan, tiene la ventaja de que es muy rápida, pero por ello es costosa y es limitada.

1.4.1.2 Almacenamiento secundario

Este tipo de memoria requiere que la computadora use sus canales de entrada/salida para acceder a la información y se utiliza para almacenamiento a largo plazo de información persistente.

En las computadoras actuales se suele usar el disco duro para almacenar los datos, es mucho más grande, pero lento de acceso. Por este motivo, la memoria secundaria se ha ganado el acrónimo de almacenamiento masivo.

1.4.2 Arreglos de memoria

Existen dos grandes clasificaciones de la memoria, y esto corresponde a la manera en que serán accedidas las direcciones de memoria.

1.4.2.1 Memoria compartida

Las diferentes unidades de computo, por ejemplo el CPU o la GPU, comparten una memoria común a la cual todos tienen acceso en igualdad de condiciones.

1.4.2.2 Memoria distribuida

En esta denominación, las diferentes unidades de cálculo, tienen una memoria propia a la cual los demás procesadores no tienen acceso directo y deben tener implementados algunos protocolos para compartir la información.

Capítulo 2. Cómputo de Alto Rendimiento

2.1 High Performance Computing

El cómputo de alto rendimiento o cómputo de alto desempeño, es el conjunto de técnicas computacionales que solucionan problemas complejos, comúnmente científicos, de una manera más acelerada que usando sistemas probablemente más simples.

2.2 Surgimiento del Multicore

Debido a que el calor generado por el CPU es proporcional a la frecuencia de reloj, entre más se aumenta la velocidad del reloj, el calor aumenta y hace necesario preocuparse por implementar mejores sistemas de ventilación, lo que trae consigo el incremento del consumo energético, el aumento de tamaño de los equipos y lo que más impacto tiene aún, el aumento de costos.

Por ello surge la necesidad de buscar técnicas para que con la misma frecuencia pueda tener más potencia de cómputo. En este punto surge la integración de varios procesadores en una misma tarjeta o también llamado el multicore. Esto nos ayuda a bajar la frecuencia de cada procesador trabajando con más información en paralelo al mismo tiempo que bajamos el gasto energético.

Aunque el multicore, en principio surgió para la disipación, ahora aprovecha la existencia de muchos núcleos de procesamiento para realizar diferentes tareas al mismo tiempo y ayudar a resolver algunos problemas en menor tiempo.

2.3 Ejecución de instrucciones

Ahora que se conoce la manera en que está diseñada la arquitectura del CPU y el GPU, podemos hablar sobre la manera en que ocurre la ejecución de las tareas de un programa, tanto serial como paralela, donde la primera hace referencia a que las instrucciones realizadas de manera secuencial (una a la vez); la segunda, como su nombre lo da a entender, varias tareas de un mismo programa son hechas de forma simultánea.

Para realizar un procesamiento en paralelo es necesario identificar los tipos de memoria para así conocer donde se encuentran las variables a ocupar. Podemos identificar dos grandes tipos, la memoria compartida, donde todos los procesos pueden acceder a todas las direcciones de una memoria común, y la memoria distribuida cada procesador accede a su memoria local comunicándose con otros mediante dispositivos de entrada/salida.

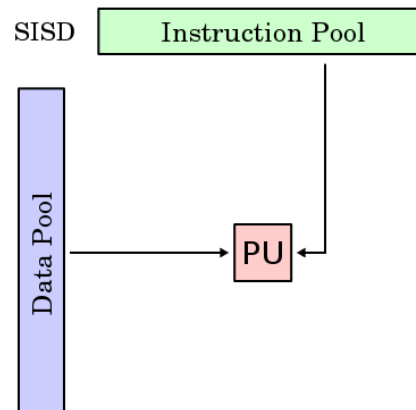
2.4 Taxonomía de Flynn

La taxonomía de Flynn es la clasificación más extendida sobre el paralelismo, en donde se distinguen el número de datos e instrucciones que puede procesar a la vez, dichos datos pueden ser simples (solamente uno) o múltiples a la vez.

		Datos	
		Simples	Múltiples
Instrucciones	Simples	SISD	SIMD
	Múltiples	MISD	MIMD

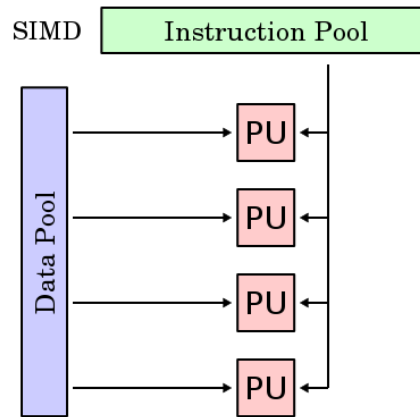
2.4.1 SISD: Single Instruction, Single Data

En esta primera categoría el procesador realiza únicamente una instrucción por cada ciclo de reloj, y dicha tarea solamente trabaja con un dato a la vez. Este es el modelo más antiguo de una computadora y el más extendido.



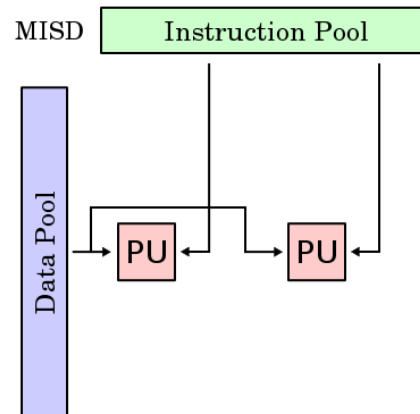
2.4.2 SIMD: Single Instruction, Multiple Data

Es necesario tener varios procesadores, aquí todos ellos ejecutan la misma instrucción, pero cada uno procesa un dato diferente, cabe destacar que todas las unidades procesan simultáneamente.



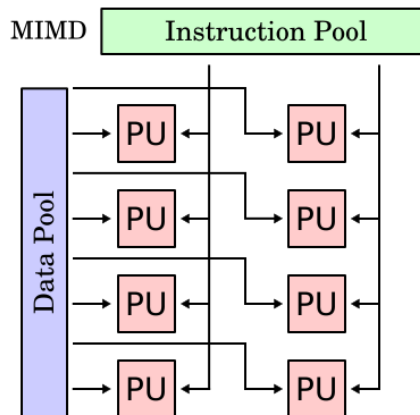
2.4.3 MISD: Multiple Instruction, Single Data

Cada unidad de procesamiento ejecuta una instrucción distinta pero aplicándolas al mismo dato. Esta categoría tiene una aplicación muy limitada en la vida real.



2.4.4 MIMD: Multiple Instruction, Multiple Data

Cada unidad ejecuta una instrucción distinta, y admite que los procesadores operen con diversos datos, lo cual ayuda a acortar tiempo al realizar diversas tareas con diferentes datos simultáneamente.



2.5 Arreglo de procesadores

Al momento de hablar sobre programación paralela, es necesario entender la forma en que están arreglados los procesadores y así comprender de que tipo de interacción tienen.

2.5.1 Procesador físico

Aquí se tiene un único chip de considerable tamaño que realiza las operaciones con las señales digitales en una computadora.

2.5.2 Socket

Referida a la placa base donde se conecta el procesador, por usos y costumbres se le conoce así también al conjunto de procesadores físicos dentro de un chip.

2.5.3 Core o Núcleo

Son, individualmente, los procesadores físicos que contiene un Socket.

2.5.4 Procesador virtual:

Con ayuda del software se puede hacer que un core funcione como dos o más concurrentemente, haciendo creer que existen varios.

2.5.5 Thread

También llamado con los nombres de hilo, hebra, fibra, lienzo, proceso ligero. Es un subproceso que se genera al partir una tarea en varios pedazos, y cada uno de esos pedazos realizara una parte del total de la tarea de manera simultanea.

Cada hilo tiene:

- Contador de programa.
- Pila de ejecución.
- El estado del procesador (incluyendo el valor de los registros).

Los hilos de ejecución que comparten los mismos recursos, son en conjunto conocidos como un proceso.

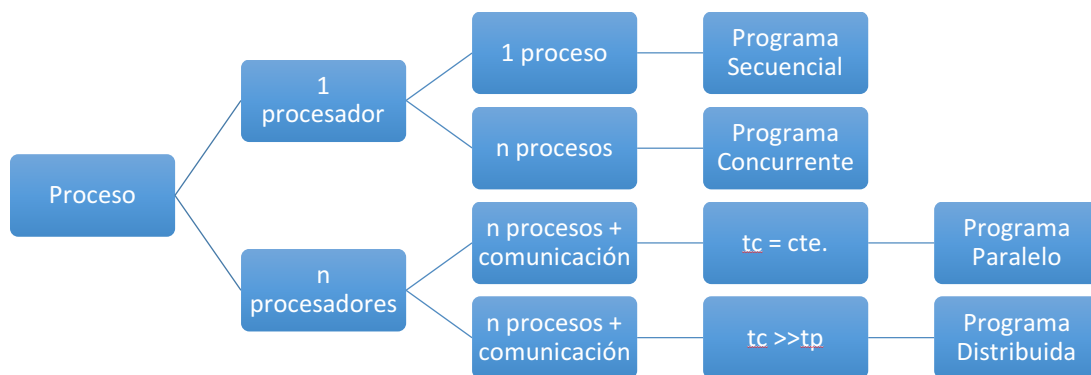
Cuando un hilo modifica un dato en la memoria, los otros hilos pueden acceden a ese dato modificado inmediatamente.

2.5.6 Variable compartida

La variable compartida o también llamada variable global, es aquella que todos los procesos concurrentes pueden ver y escribir, por lo que es necesario tener mucho cuidado sobre quien esta utilizándola en que momento.

2.6 Topología de Procesos

Podemos expresar los tipos de paralelismo con una clasificación de el numero de procesos que se realizan y la forma en que lo hacen. Si se tiene un solo procesador y se realiza un proceso a la vez, tenemos un programa secuencial. Si se tiene un solo procesador, pero es posible realizar varios procesos a la vez, eso sí, dando un corto tiempo de procesamiento a cada uno, tenemos un programa concurrente. Por otra parte si se utilizan n procesadores para realizar n procesos, caemos en dos clasificaciones que tienen al tiempo de comunicación como diferenciadora. El primer caso es llamado como programación paralela, aquí el tiempo de comunicación es aproximadamente constante, mayormente ocurre cuando se utilizan los cores de un mismo socket. El segundo caso entra cuando el tiempo de comunicación no es constante y muchas veces es mayor al tiempo de procesamiento, ya que la mayoría de las veces se lleva a cabo por una red y esta sujeta a los protocolos y ancho de banda de la misma.



2.7 Computación Paralela

Durante las ultimas décadas ha incrementado el interés en la computación paralela, el principal objetivo de ella es el incrementar la velocidad de computación.

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 concurrentemente.

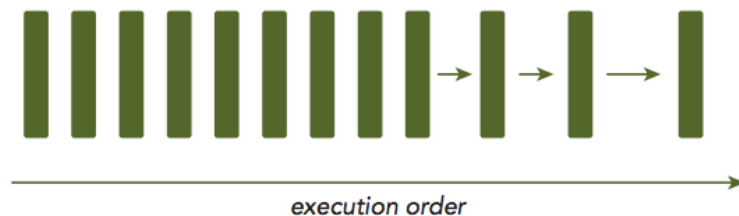
Los aspectos de software y de hardware están muy interrelacionados y pueden distinguirse dos áreas, por un lado en cuanto a hardware tenemos la arquitectura de la computadora, y por el otro en el campo del software tenemos la programación paralela.

La arquitectura de la computadora se encarga de proveer el espacio para que las cosas paralelas funcionen, dota de los dispositivos para que se puedan utilizar en cierto momento. Mientras que la programación paralela se encarga completamente de resolver los problemas utilizando el poder que se provee en la arquitectura computacional.

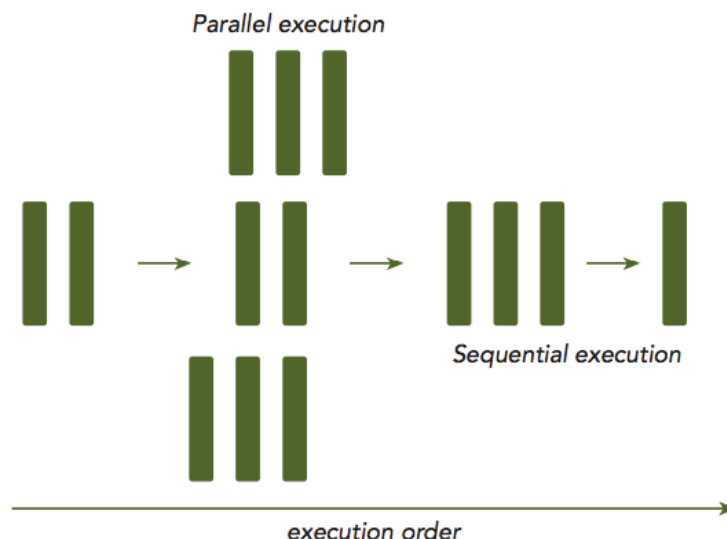
Cuando se implementan algoritmos secuenciales, no es necesario comprender los detalles de la arquitectura de la computadora para escribir programas correctos, sin embargo, cuando se implementan algoritmos para máquinas multicore es necesario conocer las características de las máquinas para saber con qué recursos cuentan para poder realizar programas tanto correctos como eficientes.

2.7.1 Programación secuencial y paralela

Cuando se resuelven problemas con programas de cómputo, lo más natural es dividir el problema en una serie de cálculos, llamadas tareas, para ejecutarlas paso a paso.



Un programa básicamente consiste de dos elementos principales, las instrucciones y los datos. Cuando un problema está dividido en pequeñas tareas es necesario saber si existe alguna dependencia de datos entre ellas y agruparlas. El análisis de las dependencias es una característica principal a tomar en cuenta cuando se quieren realizar algoritmos paralelos. Cuando se tengan identificadas todas las dependencias se puede proceder a realizar la paralelización correcta y empezar a utilizar los recursos de hardware en donde se piensa implementar.



Hay que estar conscientes de que aunque se quiera paralelizar completamente el algoritmo, siempre habrá una sección serial que inevitablemente estará presente, y es imposible

quitarla, por ello es necesario optimizar lo que más se pueda todas las secciones del código para acelerar la resolución del problema.

Existen dos principales formas de descomponer los problemas, esto dependerá de la naturaleza propia de los problemas.

2.7.1.1 Descomposición funcional

En la descomposición funcional o paralelismo de tareas el problema es dividido en un gran número de partes más pequeñas y dichas subtareas son asignadas a los procesadores disponibles. Para así ayudar a procesar los diferentes requerimientos por separado. Tan pronto como un procesador termina una subtask, recibe otra hasta que queden resueltas todas. La desventaja de este tipo de descomposición radica en que es necesario que la comunicación sea no buffereada² y así esperar hasta que se reciba completamente el dato para enviar la nueva información.

2.7.1.2 Descomposición de dominio

También llamada paralelismo de datos, divide los datos en partes de aproximadamente el mismo tamaño y las cuales son asignadas a diferentes procesadores para así trabajar sólo con la parte de datos que le toca. Esta estrategia puede ayuda a dividir los datos con el mismo algoritmo para burlar los cuellos de botella que puedan provocar cada uno de los lotes de información, ya que cada uno puede ser distinto o el tipo de dato requiere más o menor tiempo en ser resuelto. En tal caso el tiempo de ejecución es determinado por el proceso que tarde más, sin importar que los demás procesadores permanezcan en estado idle o de ocio.

2.7.2 Principales limitaciones del paralelismo

Un problema muy recurrente es la limitación de la aceleración de resoluciones, es la barrera de potencia y que es necesario recalcar que la frecuencia del reloj no puede ser incrementada sin exceder el sistema de enfriamiento, o el equipo puede ser destruido por la gran cantidad de calor que se genera.

La siguiente limitante es a la que nos enfrentamos: la barrera de memoria, donde el acceso a los datos, ya sea la memoria o el bus de datos, es muy limitado por temas de materiales de conducción de la energía.

La dependencia de datos ocurre cuando una instrucción consume datos que son producidos por una instrucción anterior, por lo que si no esta sincronizada la ejecución puede suponer un cuello de botella al necesitar esperar hasta que se tenga el recurso necesario para continuar con el procedimiento.

² Que contiene un espacio de memoria temporal.

Por último, tenemos un problema que no puede ser solucionado de manera física, la barrera de instrucciones. Está limitante es más bien matemática ya que plantea que todas las estrategias paralelas a nivel instrucción están en uso, lo que quiere expresar que el Algoritmo que se está implementando ya no da para más, se ha llegado al tope de opciones paralelas.

Teniendo en cuenta lo anterior, ahora pasemos a las limitaciones en cuanto a software. Aquí surge un concepto llamado condición de competencia, el cual ocurre cuando dos o más procesos buscan modificar una variable compartida, por ello hay que tener mecanismos de protección para evitar que se corrompa la información u ocurra el llamado deadlock, y esto sucede cuando los procesos se bloquean entre si, y ambos se quedan esperando un estado que nunca sucederá, como por ejemplo que ninguno tenga el recurso pero detecten que el otro lo está ocupando y aguarden a que se libere.

Es necesario garantizar a los procesos una eventual entrada a la solución de la tarea, sino no tendría sentido realizarla. Es muy importante asegurar la vitalidad de todos los procesos, lo que significa que todos ellos deben tener la oportunidad de consumir los recursos para poder avalar que el programa terminará y no se quedara en espera infinita.

2.7.3 Métricas de desempeño

Existe una gran 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 u 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 en ofrece comúnmente una solución más precisa al poder partir el problema en más subtareas.

Cuando se desarrollan soluciones en paralelo la primer pregunta que surge es el cuán más rápido 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 exacta, 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.

2.7.3.1 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.

$$t(p)$$

2.7.3.2 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.

$$C(p) = \text{runtime con } p \text{ procesadores} * \text{número de procesadores} *$$

$$C(p) = p * t_p$$

2.7.3.2 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.

$$S(p) = \frac{\text{runtime en un sólo procesador}}{\text{runtime con } p \text{ procesadores}}$$

$$S(p) = \frac{t_s}{t_p}$$

2.7.3.1 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.

$$E = \frac{\text{runtime en un sólo procesador}}{\text{runtime con } p \text{ procesadores} \times \text{número de procesadores}}$$

$$E = \frac{t_s}{t_p * p} 100\%$$

$$E = \frac{t_s}{C(p)} 100\%$$

$$E = \frac{S(p)}{p} 100\%$$

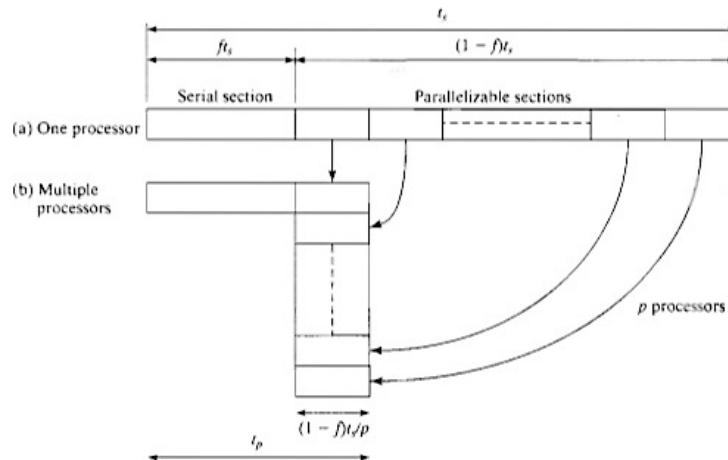
2.7.3.1 Speedup máximo

Al momento de paralelizar un programa aparecen nuevos factores que pueden mermar Speedup de la solución, ya sea porque en algunos procesadores que pueden realizar trabajo están en estado idle, o aparezcan cálculos que no están contemplados en la versión secuencial como lo son las banderas y sincronización de procesos o simplemente la comunicación entre procesos. Hay ocasiones en las que un problema por más que se quiera no se puede paralelizar en ciertas partes ya que es crítico que se realice en serie, por ello no puede pensarse en que la versión será completamente paralela.

Al existir una sección secuencial, deben tomarse en cuenta en el análisis de rendimiento tanto la sección paralela como la sección secuencial debido a que es una parte que siempre va a mantenerse constante. Esta situación está contemplada en la ecuación conocida como la “Ley de Amdahl”. Está definida como:

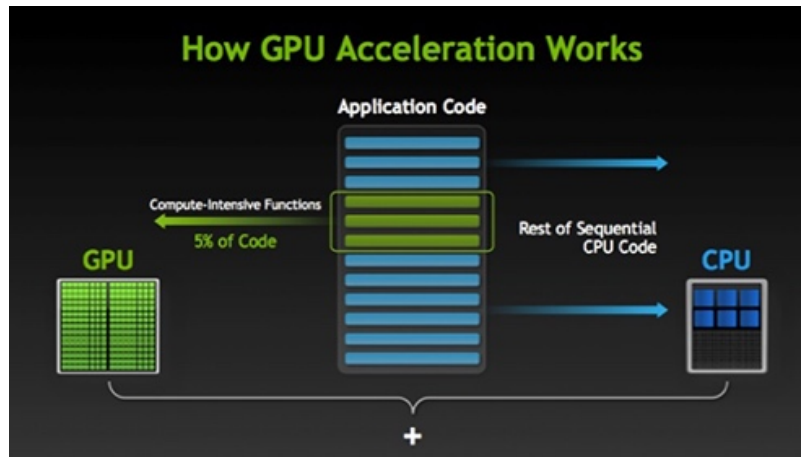
$$S(p) = \frac{t_s}{f t_s + \frac{(1-f)t_s}{p}} = \frac{p}{1 + (p-1)f}$$

Donde f expresa la fracción de procesamiento que no puede ser dividido en tareas paralelas. Por lo que podemos observar, entre mayor sea sección serial, menor será el Speedup y no tendrá caso paralelizar las tareas.



2.8 Cómputo en GPU

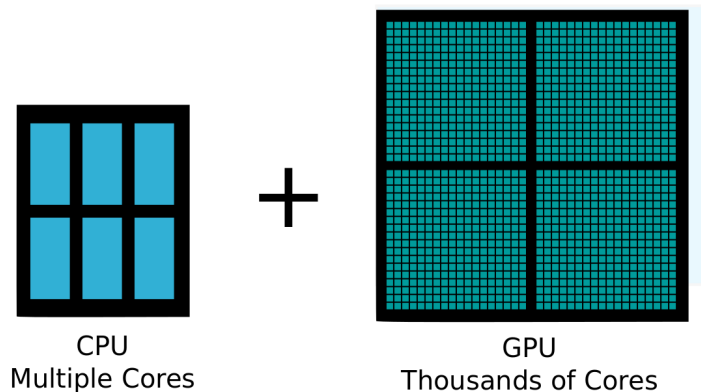
El cómputo en GPU o también llamado GPGPU (cómputo de propósito general en unidades de procesamiento de gráficos) es utilizada para acelerar el cómputo realizado tradicionalmente por únicamente la CPU, donde la GPU actúa como un coprocesador que puede acelerar el trabajo donde la computación es más intensiva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



2.8.1 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.

El GPU es un procesador mucho más especializado para tareas que requieren de un alto grado de paralelismo. La tarjeta grafica 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. Lo más común al utilizar la aceleración por GPU es ejecutar una misma instrucción a múltiples datos para aprovechar su arquitectura.



Es necesario destacar que los manycore y los multicore son utilizados para etiquetar a los CPU y los GPU, pero no entre ellos son un poco diferentes. 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.

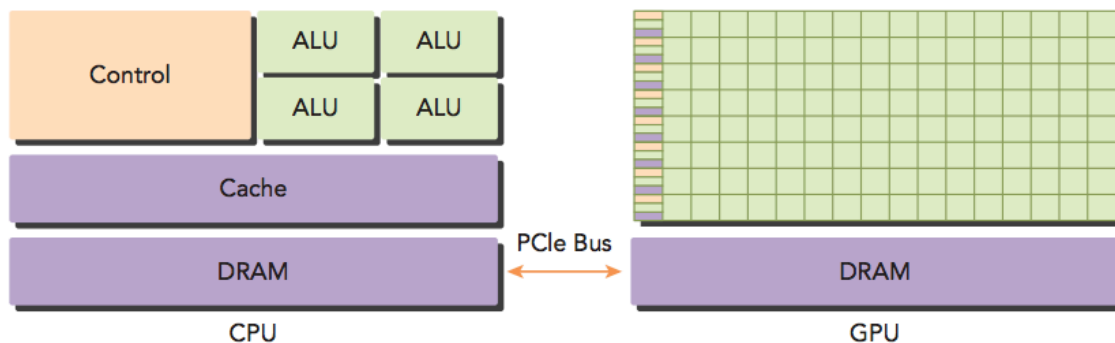
2.8.2 Cómputo Heterogéneo

El cambio de sistemas homogéneos a sistemas heterogéneos es un paso muy importante en la historia del cómputo de alto rendimiento. La computación homogénea utiliza uno o más procesadores de la misma arquitectura para ejecutar una aplicación. En cambio, el cómputo heterogéneo usa un conjunto de procesadores con diversas arquitecturas para ejecutar un programa, aplicando tareas a las arquitecturas que son más adecuadas, produciendo una mejora en el rendimiento como resultado.

Aunque los sistemas heterogéneos proporcionan ventajas significativas en comparación con el alto rendimiento tradicional, el uso efectivo de tales sistemas está actualmente limitado por el aumento de complejidad en el diseño de una solución.

Normalmente los CPU y los GPU están conectados por el bus de PCI-Express en un nodo de computo, aquí a la tarjeta gráfica se les conoce como Device, y al CPU al ser el que se encarga de dar las instrucciones al GPU, se lo conoce como Host. Un nodo de procesamiento heterogéneo típico consiste en dos socket de CPU multicore y uno o más GPU manycore.

Actualmente, una GPU no es una plataforma independiente sino un coprocesador de un CPU. Por lo tanto, las GPU deben operar en conjunto con un host basado en CPU a través de un bus PCI-Express



Una aplicación heterogénea consiste de dos partes, el código Host que corre en el CPU, y el código Device que corre en el GPU. Un programa que corre en una plataforma heterogénea es indudablemente iniciada por el CPU, el código del CPU es el responsable de manejar el ambiente, código y datos para el device antes de realizar las tareas intensivas en la tarjeta gráfica.

2.9 Tecnología de Gráficos

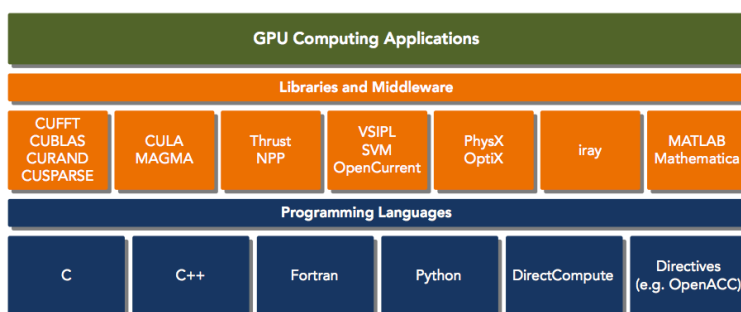
Las tarjetas gráficas modernas son procesadores cuyo principal objetivo es realizar cálculos para realizar el renderizado³ de multimedia. Pero por las necesidades crecientes del cómputo científico y la capacidad de realizar tareas en paralelo de las GPU, se ha buscado explotar este tipo de arquitecturas.

Actualmente existen dos grandes estándares de programación, CUDA y OpenCL. Primero tenemos a CUDA la cual es el estándar de programación de las GPU de la marca NVIDIA, al ser esto así la empresa ofrece soporte, constantes actualizaciones a bibliotecas y completo control del hardware, con lo que podemos acelerar las aplicaciones en un grado muy alto. OpenCL por un lado es un API abierta que permite acceder a la potencia de procesamiento de la programación GPGPU en cualquier tarjeta gráfica, pero por el otro, al tener toda una gama de posibilidades en las que puede trabajar, no está optimizada. Esto hace que sea una alternativa si no se cuenta con Tarjetas de la marca NVIDIA.

El cómputo científico busca la resolución de problemas muy grandes en el menor tiempo posible, por ello es recalcada la búsqueda de nuevas soluciones, una de ellas es el computo sobre tarjetas gráficas, y por ello mismo es justificable la utilización de CUDA.

Los GPU, según la taxonomía de Flynn, están en el campo del paralelismo del tipo MIMD (Multiple Instruction, Multiple Data) y en el de SIMD: Single Instruction, Multiple Data. EL GPU y el CPU no comparten el mismo ancestro, ya que históricamente los GPU son coprocesadores aceleradores de gráficos, pero recientemente éstos están evolucionando para ser más poderosos y de propósito general, son completamente programables y están ganando su lugar en los problemas de computo paralelo masivo.

Se puede acceder al cómputo con la plataforma CUDA a través de bibliotecas aceleradas propias de CUDA, directivas de compilación, interfaces de programación de aplicaciones y extensiones de lenguajes de programación estándar de la industria, incluidos los más importantes en el desarrollo del cómputo científico, como lo son C, C++, Fortran y Python.



³ Término utilizado para referirse al proceso de generar una imagen partiendo de un modelo matemático.

2.9.1 C/C++, Fortran, Python

Como ya se observó C/C++, Fortran y Python tienen extensión de CUDA, pero ¿cuál elegir?, la elección natural en el campo científico sería Fortran al tener tanta popularidad y estar presente en la gran mayoría de estos sistemas, pero con C/C++ tenemos mayor soporte en cuanto al manejo de memoria y estructuras de datos, por lo que es más viable su utilización, además de que ofrece mayor escalabilidad en un futuro al tener la posibilidad de implementar orientación a objetos. Python tiene la ventaja de ser un lenguaje de alto nivel que al tener un diseño simple ayuda al desarrollo de código rápido, pero sacrifica el rendimiento por completo, por lo que aún no es una opción.

2.9.2 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.

La primera versión disponible de deMon apareció en 1992 en la Université de Montréal (UdM) como un proyecto de tesis de doctorado ⁴.

deMon significa “densité de Montréal”.

Algunas características que contiene el programa son:

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

La solución que se pretende mostrar en el **Capítulo 5**, está basada en el módulo del cálculo de la energía por teoría del DFT de este paquete.

2.9.2.3 Summit

En marzo de 2014, el Departamento de Energía de Estados Unidos (DOE), otorgó a IBM la comisión para construir dos superordenadores. Sierra localizada en el Laboratorio Nacional Lawrence Livermore y Summit localizada en el Laboratorio Nacional Oak Ridge.

La misión era desarrollar una máquina de 5 a 10 veces más rápida que su predecesor, Titán. Durante cuatro años, el DOE trabajó con un equipo de IBM para superar innumerables barreras tecnológicas, construyendo un sistema con la fuerza de 200 petaflops (aproximadamente cuatrillones de operaciones por segundo) y capacidad de Deep Learning con neuronas de AI. Esto significa que Summit es más de 1 millón de veces más poderosa que la computadora personal más rápida existente, lo que le permite crear modelos, simulaciones y examinar miles y miles de variables que pueden ayudar a los investigadores a encontrar respuestas a problemas cada vez más complejos.

Las especificaciones de Summit contiene 250 PBytes de almacenamiento, por nodo 500 GBytes de memoria coherente y 800 GBytes de RAM no volátil, 9216 CPUs POWER9 con 22 core cada uno, que se conectan con 27,648 GPUs Nvidia Tesla V100 por medio de NVLink de NVIDIA para acelerar la conexión entre ellos y eliminar en gran medida la latencia.

⁴ Alain St-Amant, Ph.D. Thesis, Université de Montréal, 1992.

Summit representan 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.

2.10 CUDA

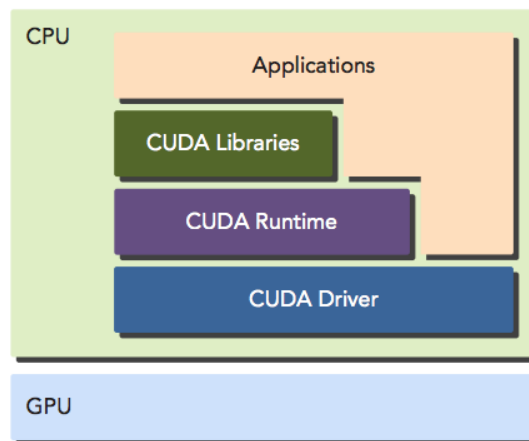
CUDA es el acrónimo en inglés de *Compute Unified Device Architecture*, el cual es una arquitectura de hardware y de software que permite ejecutar programas en las tarjetas gráficas de la marca NVIDIA. Utiliza el modelo SIMT (Single Instruction, Multiple Thread) el cual está basado en el modelo SIMD, véase 2.4.2, pero adecuado al la arquitecturas de las tarjetas gráficas. 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.

2.10.1 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.

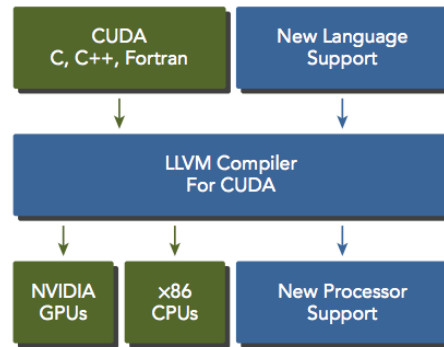
CUDA nos provee dos niveles de API para manejar el GPU y organizar los threads. El *API driver* es un API de nivel bajo y es un poco difícil de programar, pero provee un control sobre como el dispositivo (GPU) está siendo usado. El *API runtime* es un API a un nivel más alto, cada función de esté nivel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



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 LLVM⁵, 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.



2.10.2 Estructura CUDA

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

kernel	Son las funciones paralelas escritas en 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.

⁵ 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.

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 512 ó 1024 threads, dependiendo de la versión del CUDA. Los bloques de threads pueden ser de una, dos o tres dimensiones.

Hay que tener en cuenta que aunque todos los 512 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.

2.10.3 Ubicación de Threads

threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2							

2.10.4 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 los nombres con *h_* para las del primero y *d_* para el 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. 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){}
```

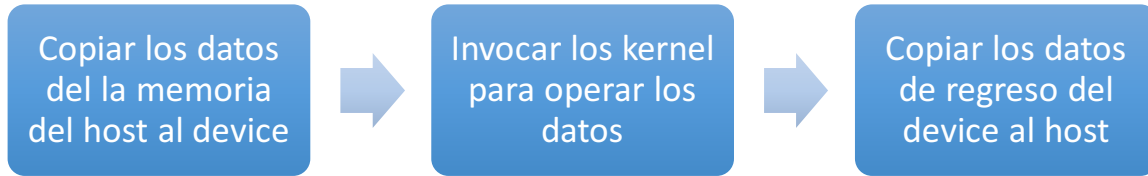
Para poder lanzar el kernel, se debe utilizar la nomenclatura siguiente:

```
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. El modelo de programación de CUDA es principalmente asíncronico, 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:



2.10.5 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 con ese pedazo de datos. 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 pedazos de información, por ello pueden ir saltando entre los datos.

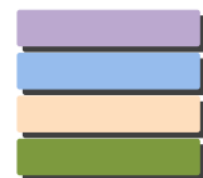


Block partition: each thread takes one data block



Cyclic partition: each thread takes two data blocks

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.



Block partition on one dimension



Block partition on both dimensions



Cyclic partition on one dimension

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

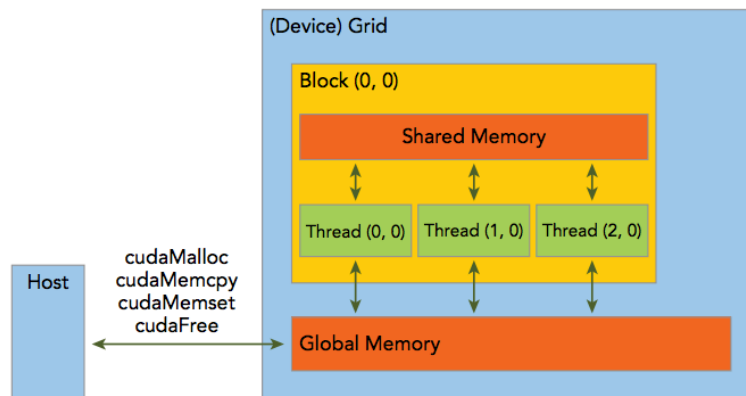
Funciones estándar de C	Funciones de CUDA C
<i>malloc</i>	<i>cudaMalloc</i>
<i>memcpy</i>	<i>cudaMemcpy</i>
<i>memset</i>	<i>cudaMemset</i>
<i>free</i>	<i>cudaFree</i>

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.



2.10.5.1 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 numero 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.

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

$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

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.

Capítulo 4. Fisicoquímica

4.1 Química

La química es la ciencia que estudia la estructura, composición y las propiedades de la materia, así como también las transformaciones que se experimentan al realizarse reacciones en ella.

Los cuerpos están formados por partículas, y estas a su vez por moléculas y átomos. Las moléculas pueden estar formadas por átomos de la misma naturaleza (elementos) o diferente naturaleza (compuestos).

4.1.1 Átomo

La partícula más pequeña que compone a la materia, esta a su vez en subpartículas llamadas protones (+) y neutrones (+/-) que se encuentran en el núcleo y por los electrones (-) que se encuentran girando alrededor de este.

4.1.2 Elemento

La sustancia de la cual no se puede, por medios químicos, obtener otra más sencilla. Es la menor cantidad de átomos que puede existir en libertad.

4.1.3 Molécula

Es la unión estable de varios átomos (iguales o diferentes), mediante las fuerzas de valencia, siendo la porción más simple de materia que conserva sus propiedades y se representa con formulas químicas. Es electrónicamente neutra ya que se comporta como una sola partícula. Las moléculas se pueden descomponer para separar los elementos que las componen por métodos químicos.

4.1.4 Fisicoquímica

La química como una ciencia física, es única al examinar y crear estructuras moleculares, ya que controla los sistemas moleculares a través del diseño y desarrollo de herramientas de estudio del comportamiento, tanto de los átomos, como de las moléculas, o a mayor escala como sistemas moleculares tan complejos como se requieran.

Las investigaciones en fisicoquímica se realizan por medio del enfoque de la energía de las estructuras químicas y su transformación, también provee las bases moleculares para la investigación todas las tecnologías de que permiten la conversión de energía.

Al haber avances en la ciencia tomando en cuenta este enfoque, se requieren realizar modelos que permitan realizar simulaciones atomísticas⁶ del comportamiento molecular, aunque dichas simulaciones pueden ser desde la interacción entre átomos, no se descarta que se requiera representar dinámica molecular en seres vivos. Por dicho motivo, es necesario abordar la problemática con ayuda de tecnologías computacionales para, así, obtener resultados mucho más rápida y fácilmente, teniendo como consecuencias adjunta el poder formular nuevas direcciones en cuanto a las reacciones químicas, en menor tiempo.

4.2 Física

Es la ciencia que estudia las propiedades de los cuerpos y las leyes que rigen las transformaciones que afectan a su estado y a su movimiento, sin alterar su naturaleza.

4.2.1 Mecánica Clásica

En el campo de la física, la mecánica clásica es uno de las dos principales ramas de estudio en la ciencia de la mecánica.

Tiene que ver con el conjunto de leyes físicas que rigen y la matemática que describe los movimientos de un cuerpos y las distribuciones geométricas dentro de un límite determinado por la acción de un sistema de fuerzas.

4.2.2 Mecánica Cuántica

La otra rama es la mecánica cuántica, también conocida como la física cuántica o la teoría cuántica, es la disciplina que estudia y brinda una descripción del movimiento de partículas a escalas espaciales muy pequeñas.

Delimita matemáticamente una gran parte del área, ya que trabaja con el comportamiento dual de partícula y de onda, similar a como son las interacciones de la energía y la materia.

Dentro de la mecánica cuántica, existe la mecánica cuántica relativista, y es la generalización que ayuda a comprender el comportamiento de las partículas que alcanzan velocidades cercanas a la luz, donde la ecuación de Schrödinger deja de ser efectiva.

4.2.3 Constante de Planck

La constante de Planck es una de las constantes fundamentales de la física, que fue propuesta para explicar la radiación de un cuerpo negro. Si se acepta la suposición de que la materia sólo puede tener estados de energía discretos y no continuos, entonces la energía irradiada no puede tomar cualquier tipo de valores sino únicamente valores

⁶ **Atomístico**, relativo a atomismo, donde el atomismo es la doctrina que explica la formación del mundo por la concurrencia fortuita de los átomos.

múltiples enteros de un quantum⁷ de energía. La constante de Planck vincula el valor de la energía a la frecuencia de la radiación:

$$E = hf$$

E = energía de la frecuencia
 $h = 6.63 \times 10^{-34}$ [Js] (cte de Planck)
f = frecuencia de onda

4.2.4 Principio de incertidumbre

Principio de incertidumbre o también llamada la relación de indeterminación de Heisenberg, establece que es imposible medir simultáneamente, y con precisión absoluta, el valor de la posición y la cantidad de movimiento de una partícula. Por ello define una de las diferencias fundamentales entre física clásica y física cuántica al no tener un símil en el campo de lo clásico.

La posición y la cantidad de movimiento de una partícula, respecto de uno de los ejes de coordenadas, son magnitudes complementarias que solo se pueden medir probabilísticamente con un límite fijado en la constante de Planck.

$$\Delta x \cdot \Delta p_x \geq \frac{h}{4\pi}$$

$\Delta x =$ indeterminación de la posición
 $\Delta p_x =$ indeterminación de la cantidad de movimiento
 $h =$ constante de Planck

4.2.5 Aproximación de Born-Oppenheimer

Una de las aproximaciones fundamentales de la mecánica cuántica es el desacoplamiento de los movimientos electrónicos y nucleares. Al ser la masa del núcleo mayor a la de los electrones, su velocidad es menor, por lo que para él los electrones funcionan como una nube de carga, y por otro lado para los electrones, los núcleos parecen estar estáticos.

4.2.6 Función de onda

La función de onda Ψ es una forma de representar el estado físico de un sistema de partículas, es una función compleja que da las coordenadas espaciales de cada una de las partículas.

4.3 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. Pero para poder ser implementada se requiere de mucho poder de cómputo, y al ser calculada con métodos numéricos, depende de cuál se elija, los resultados que se arrojan pueden ser algo imprecisos.

⁷ Un quantum o cuanto es la menor cantidad de energía que puede transmitirse en cualquier longitud de onda.

La ecuación de Schrödinger considera varios aspectos para poder ser implementada, dentro de los fundamentales están:

- La existencia de un núcleo atómico, en donde se concentra la mayoría del volumen del átomo.
- Los niveles energéticos donde se distribuyen los electrones según su energía.
- La dualidad onda-partícula.
- La probabilidad de encontrar al electrón.

Aunque con la mecánica cuántica queda claro que no se puede saber dónde se encuentra un electrón, sí define la región en la que puede encontrarse en un momento dado. Cada solución de la ecuación de ondas de Schrödinger, Ψ , describe un posible estado del electrón. El cuadrado de la función de onda, Ψ^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.

$$\frac{\partial^2 \Psi}{\partial x^2} + \frac{8\pi^2 m}{h^2} (E - V) \Psi = 0$$

Ψ = función de onda
 m = masa del electrón
 h = constante de Planck
 E = energía total del electrón
 V = energía potencial del electrón

4.4 Teoría del Funcional 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.

Con esta teoría el potencial efectivo está determinado por la densidad electrónica del sistema, se calculan las propiedades electrónicas a partir de la densidad tridimensional de las nubes electrónicas. (Urriolabeitia 2017)

La aproximación DFT se basa en la estrategia de introducir la correlación electrónica usando funcionales⁸ de la densidad electrónica. Estos métodos usan los teoremas de Hohenberg-Kohn, en el que se demuestra la existencia de un solo funcional que determina la energía del estado fundamental y la densidad electrónica exactamente.

- *Primer teorema de Hohenberg-Kohn.*

⁸ Funciones de funciones.

Existe una correspondencia de uno a uno entre el estado base de la densidad de un sistema de muchos electrones y el potencial externo que se genera al suponer la aproximación de Born-Oppenheimer. Una consecuencia inmediata es que el valor esperado del estado base de cualquier observable⁹ \hat{O} es único funcional de la densidad electrónica exacta del estado base:

$$\langle \Psi | \hat{O} | \Psi \rangle \geq O[\rho]$$

- *Segundo teorema de Hohenberg-Kohn.*

El funcional de Hohenberg-Kohn es universal para todo sistema de muchos electrones y alcanza su valor mínimo, igual a la energía total del estado base, para la densidad del estado base correspondiente a la energía del potencial.

La universalidad del funcional implica que al ser un sistema de muchas partículas, no depende de ninguna variable nuclear, es decir que contiene información únicamente de los electrones del sistema. Por ello se puede definir el operador densidad como:

$$\rho(\vec{r}) = \sum_{i=1}^N (\vec{r}_i - \vec{r})$$

\vec{r} = punto dónde probablemente se encuentre un electrón

4.4.1 Ecuación de Kohn-Sham

Sin embargo, el teorema no da la forma del funcional. Las aportaciones de la ecuación de Kohn-Sham a la DFT que emplea a la densidad electrónica como variable básica en lugar de la función de onda electrónica. La energía electrónica se divide en varios términos:

$$\begin{aligned} E &= E^T + E^V + E^J + E^{XC} \\ E^T &= \text{energía cinética electrónica} \\ E^V &= \text{energía de atracción núcleo - electrón} \\ E^J &= \text{energía de repulsión electrón - electrón} \\ E^{XC} &= \text{energía potencial de correlación e intercambio} \end{aligned}$$

Más propiamente podemos expresar de la siguiente manera:

$$\overline{H_{KS}} = -\frac{1}{2} \nabla_i^2 + \frac{e^2}{4\pi} \int \frac{\rho(\vec{r})}{|\vec{r}_i - \vec{r}|} d\vec{r} + E^J + E^{XC}$$

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. Por ello hasta el momento es el procedimiento preferido para abordar problemas a partir de una cierta complejidad.

⁹ Propiedad del estado de un sistema que puede ser determinada ("observada") por alguna secuencia de operaciones físicas.

A partir de la ecuación de Kohn-Sham se puede hallar la densidad exacta del estado base tomando en cuenta dos teoremas.

4.4.1.1 Teorema de Kohn-Sham

La densidad exacta del estado base $\rho(\vec{r})$ de un sistema de N electrones es

$$\rho(\vec{r}) = \sum_{i=1}^N \Psi^*(\vec{r}_i) - \Psi(\vec{r})$$

donde las funciones de onda electrónicas son las N soluciones de menor energía de la ecuación de Kohn-Sham, simplificando podemos obtener:

$$\overline{H_{KS}}\Psi_i = \epsilon_i\Psi_i$$

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 ello 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 Kohn-Sham.

Capítulo 4. Algoritmo de Jacobi

4.1 Forma cuadrática

Todo polinomio de segundo grado homogéneo de la forma:

$$q = X'AX = \sum_{i=1}^n \sum_{j=1}^n a_{ij}x_i x_j$$

$$q(x_1, x_2, \dots, x_n) = (a_{11}x_1^2 + a_{22}x_2^2 + 2a_{12}x_1x_2 + \dots + 2a_{n-1n}x_{n-1}x_n)$$

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

$$q(x_1, x_2, \dots, x_n) = (x_1, x_2, \dots, x_n) \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = X'AX$$

Donde la matriz A asociada a la forma cuadrática es una matriz simétrica 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 ($a_{ij} = a_{ji}$). 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.

Ejemplo.

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

$$q = x_1^2 + 2x_2^2 - 7x_3^2 - 4x_1x_2 + 8x_1x_3$$

$$q = X' \begin{bmatrix} 1 & -2 & 4 \\ -2 & 2 & 0 \\ 4 & 0 & -7 \end{bmatrix} X$$

4.2 Eigensistemas

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

$$\begin{bmatrix} a_{11} - \lambda & a_{12} & a_{13} \\ a_{21} & a_{22} - \lambda & a_{23} \\ a_{31} & a_{32} & a_{33} - \lambda \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

o de la forma

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

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

$$[A] \cdot [x] = \lambda[x]$$

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.

4.2.1 Transformación de coordenadas

Para mostrar el siguiente concepto, tenemos una matriz x como vector columna con dos componentes x_1 y x_2 , si ponemos el los ejes de \bar{x}_1 y \bar{x}_2 con el mismo origen, tenemos que el vector \overline{OD} tiene las dos diferentes componentes mencionadas. Trigonométricamente tenemos:

$$\begin{aligned} x_1 &= \bar{x}_1 \cos \theta - \bar{x}_2 \sin \theta \\ x_2 &= \bar{x}_1 \sin \theta + \bar{x}_2 \cos \theta \end{aligned}$$

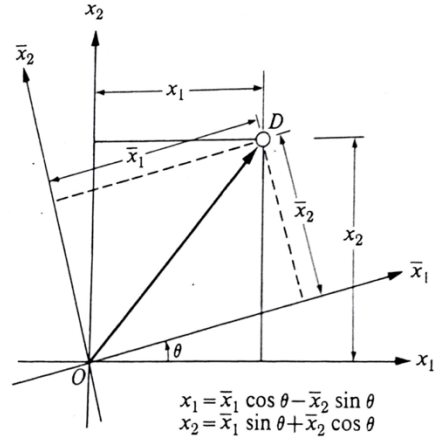
o en la forma matricial:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \end{bmatrix}$$

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

$$[x] = [T][\bar{x}]$$

que conecta los dos sistemas x y \bar{x} .



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

$$[A][T][\bar{x}] = \lambda[T][\bar{x}]$$

pudiendo también:

$$[T]'[A][T][\bar{x}] = \lambda[T]'[T][\bar{x}]$$

recordando que $T'T = I$, comprobándolo con:

$$\begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

por ende deducimos que:

$$\begin{aligned} [T]'[A][T][\bar{x}] &= \lambda[I][\bar{x}] \\ &= \lambda[\bar{x}] \end{aligned}$$

4.2.2 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.

$$\begin{aligned} B &= \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \\ &= \begin{bmatrix} a_{11} \cos^2 \theta + 2a_{12} \sin \theta \cos \theta + a_{22} \sin^2 \theta & a_{12}(\cos^2 \theta - \sin^2 \theta) + \sin \theta \cos \theta (a_{22} - a_{11}) \\ a_{12}(\cos^2 \theta - \sin^2 \theta) + \cos \theta \sin \theta (a_{22} - a_{11}) & a_{11} \sin^2 \theta - 2a_{12} \sin \theta \cos \theta + a_{22} \cos^2 \theta \end{bmatrix} \end{aligned}$$

Como quereos eliminar el elemento b_{12} fuera de la diagonal y teniendo en cuenta que b_{12} y b_{21} son similares:

$$a_{12}(\cos^2\theta - \sin^2\theta) + \cos\theta\sin\theta(a_{22} - a_{11}) = 0$$

por identidades trigonométricas podemos concluir:

$$\tan 2\theta = \frac{2a_{12}}{a_{11} - a_{22}}$$

Dando como resultado esperado:

$$B = \begin{bmatrix} a_{11}\cos^2\theta + 2a_{12}\sin\theta\cos\theta + a_{22}\sin^2\theta & 0 \\ 0 & a_{11}\sin^2\theta - 2a_{12}\sin\theta\cos\theta + a_{22}\sin^2\theta \end{bmatrix}$$

Con lo que al final del procedimiento podemos obtener que b_{11} y b_{22} son los eigenvalores deseados.

4.2.3 Eigenvector

Los eigenvectores son matrices columna $[\vec{x}]$ de una matriz $[A]$ a los que les corresponde un eigenvalor λ . Para agrupar los eigenvectores en una matriz cuadrada V únicamente se agrupan de izquierda a derecha. También es necesario agrupar los eigenvalores en una matriz matriz diagonal $[\lambda]$. Así podemos expresar la igualdad:

$$[A][V] = [V][\lambda]$$

Si aplicamos un poco de algebra:

$$[V]^{-1}[A][V] = [V]^{-1}[V][\lambda]$$

Tomando en cuenta que $[V]^{-1}[V] = [I]$:

$$[V]^{-1}[A][V] = [\lambda]$$

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

$$[V] = [T_1][T_2] \cdots [T_m]$$

4.3 Algoritmo en C

Ya que algoritmo que se pretende paralizar debe ser el más eficiente, se usara el que está en el *"Numerical Recipes in C. The Art of Scientific Computing"*.

Es necesario realizar algunas modificaciones menores al código para que en la versión paralela pueda correr con la misma disposición de memoria. Por ello las matrices que en la versión se expresaban de la forma $A[i][j]$ se cambiaran a la forma $A[i * N + j]$, véase sección 2.10.5.1.

Capítulo 5. Implementación

Como queremos que nuestro código corra en todas las configuraciones de cómputo de las tarjetas gráficas, usaremos el estándar de bloques de 16×32 para que contengan 512 threads. Todos los bloques deben tener la misma dimensión y se deben lanzar los suficientes bloques para cubrir por completo la matriz. Aunque si las dimensiones de la matriz no son múltiplos de 16, algunos de los threads estarán en estado idle.

Capítulo 6. Pruebas y resultados

El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi.

Conclusiones

El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi.

Tabla de ilustraciones

No se encuentran elementos de tabla de ilustraciones. Esta es una tabla de contenidos automática. Para usarla, aplique los estilos de encabezado (en la ficha Inicio) al texto que va en la tabla de contenidos y después actualice la tabla.

Si quiere escribir sus propias entrada, use una tabla de contenidos manual (en el mismo menú que la automática).

Bibliografía

- <https://chemistry.stanford.edu/research/research-areas/physical-chemistry> (último acceso: 22 de junio de 2018).
- <https://chemistry.stanford.edu/research/research-areas/theoretical-chemistry> (último acceso: 22 de junio de 2018).
- http://www.ub.edu/web/ub/es/recerca_innovacio/recerca_a_la_UB/instituts/institutspropis/iqtcub.html (último acceso: 22 de junio de 2018).
- http://www.demon-software.com/public_html/index.html (último acceso: 23 de junio de 2018).
- http://www.demon-software.com/public_html/program.html#branches (último acceso: 23 de junio de 2018).
- <https://www.ibm.com/thought-leadership/summit-supercomputer/> (último acceso: 23 de junio de 2018).
- <https://www.amd.com/es-xl/products/graphics/server/gpu-compute> (último acceso: 23 de junio de 2018).
- <http://la.nvidia.com/object/what-is-gpu-computing-la.html> (último acceso: 23 de junio de 2018).
- <https://www.profesionalreview.com/2017/06/21/diferencia-la-cpu-la-gpu/> (último acceso: 23 de junio de 2018).
- <https://www.nucleares.unam.mx/~vieyra/node26.html> (último acceso: 23 de junio de 2018).
- <http://www.eis.uva.es/~qgintro/atom/tutorial-10.html> (último acceso: 9 de julio de 2018).
- <https://tecnologia-informatica.com/la-memoria-ram/> (último acceso: 27 de mayo de 2018).
- <https://www.profesionalreview.com/2017/06/21/diferencia-la-cpu-la-gpu/> (último acceso: 12 de mayo de 2018).
- Ayres, Frank Jr. *MATRICES, Teoría y problemas resueltos*. McGraw-Hill, 1962.
- densidad electrónica, Teoría del funcional de la.
- <http://www.fis.cinvestav.mx/~daniel/thELA.pdf> (último acceso: 12 de junio de 2018).
- DENSIDAD, TEORIA DE FUNCIONES DE LA.
- http://depa.fquim.unam.mx/amyd/archivero/DensityFunctionalTheory_21556.pdf (último acceso: 14 de junio de 2018).
- Kohn-Sham, Potencial exacto de Kohn-Sham para sistemas finitos fuertementecorrelacionados. Luis Antonio Benítez Moreno.
- http://ricabib.cab.cnea.gov.ar/519/1/1Benitez_Moreno.pdf (último acceso: 14 de junio de 2018).
- Schrödinger, Sobre la ecuación de. <http://www.ugr.es/~jllopez/Cap3-Sch.pdf> (último acceso: 14 de junio de 2018).
- SHOLL, DAVID S. *DENSITY FUNCTIONAL THEORY. A Practical Introduction*. National Energy Technology Laboratory: Georgia Institute of Technology, 2009.
- Urriolabeitia, Esteban. «La teoría del funcional de la densidad (DFT) va por el mal camino.» *divulgame.org*, enero 2017.

Anexos

El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi. El veloz murciélago hindú comía feliz cardillo y kiwi.