Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Herramienta PSE (problem solving experiment) para problemas financieros modelados con la integral de Wiener a través del método de Monte Carlo

Tesis que presenta

Ivonne Yanely Olmos Aquino

para obtener el Grado de

Maestra en Ciencias

en Computación

Director de Tesis

Dr. Sergio Chapa Vergara

Mexico, D.F. **Septiembre de 2014**

AGRADECIMIENTOS

Agradezco al Consejo Nacional de Ciencia y Tecnología por el apoyo económico recibido.

Agradezco al Centro de Investigación y de Estudios Avanzados (CINVESTAV) del IPN por los apoyos brindados a lo largo de esta travesía.

Agradezco al Dr. Sergio V. Chapa Vergara por el apoyo brindado para poder concluir el presente trabajo de tesis.

Agradezco a mis sinodales, el Dr. Jorge Buenabad chávez y el Dr Amilcar Meneses Viveros por el apoyo brindado, el tiempo y la atencion que tuvieron para revisar el presente trabajo de tesis.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivación

La simulación computacional de problemas complejos requiere del uso de computo de alto rendimiento. El computo de alto rendimiento hace uso de tecnologías como clústers, supercomputadoras, computo en Grid o el computo paralelo.

En los últimos años el computo paralelo ha adquirido una gran importancia pues es posible dividir problemas que parecen muy grandes y complejos en otros mas pequeños que más tarde se resolverán de manera simultanea. Para explotar está característica los equipos modernos cuentan con procesadores multinucleo y multi-procesador.

Más recientemente con el nuevo enfoque de las GPU (Graphics Processing Unit) que pasaron de ser sólo para procesamiento gráfico a convertirse en unidades de procesamiento general GPGPU (General Purpose Graphic Proccessing Unit), la mayoría de las computadoras de escritorio ya tienen incluida una tarjeta para el procesamiento gráfico. Los fabricantes de estas tarjetas se han esforzado por lograr que la comunidad científica use éstas para la resolución de sus problemas que requieren un gran poder de cómputo. Es así como apareció CUDA que es básicamente C con algunas sentencias añadidas, justo para facilitar el proceso de cambio de un lenguaje a otro de la comunidad científica.

En la resolución de problemas científicos complejos en áreas como física, química, biología, economía entre otras se requiere de un gran poder de computo y entre las opciones que se tienen en el computo de alto rendimiento se ha elegido el computo paralelo puesto que el método de Monte Carlo es inherentemente paralelo.

Muchos problemas de la vida real no tienen una solución exacta y la única manera de aproximar su solución es por medio de la simulación. Inicialmente el método de Monte Carlo fue usado en problemas especificos para la física, pero su aplicación no se limita a una ciencia, pues puede usarse en diferentes tipos de problemas de diferentes áreas. En la simulación de Monte-Carlo se intenta seguir el "tiempo de dependencia" de un modelo

para el cual los cambios o su crecimiento no procede de una forma predefinida sino mas bien tienen una naturaleza estocástica la cual depende de una secuencia de números aleatorios que se generan durante la simulación.

La precisión de Monte-Carlo depende de la minuciosidad con la cual es muestrado el espacio fase. A medida que se aumenta el número de pruebas El resultado es mas presiso. Cada nueva ejecución del cálculo que se haga derivará en diferentes resultados, pero se producirán valores los cuales se consideran con los ya obtenidos para obtener el error estático.

El método de Monte Carlo es ubicuo en aplicaciones en el mundo de las finanzas y la industria de las aseguradoras. Este método es algunas veces la única herramienta posible para la ingeniería financiera o para problemas económicos complejos que requieren ser calculados como riesgos, precios de acciones, precios de opciones etc. En años recientes se han visto muchos desarrollos de los métodos de Monte Carlo con un alto potencial para aplicaciones exitosas [1, 2, 3].

El tiempo y la memoria que consume la generación de números aleatorios y las trayectorias brownianas representan un problema pues en el pasado había problemas que podían tardar incluso años en ser resueltos por el método de Monte Carlo. La computación de alto rendimiento dio una posibilidad para resolver estos problemas pues ahora se puede usar cluster de computadoras para dividir el problema y resolver pequeñas partes en cada nodo. Debido a la capacidad paralela intrínseca del método de Monte-Carlo, esto fue una solución aceptable. Sin embargo, ahora se cuenta con los procesadores multi-núcleo que permiten hacer esta misma partición en pequeñas tareas para que no sea el procesador central quien realice todo el cómputo lo que resulta en un menor tiempo de ejecución.

1.2. Planteamiento del problema

1.2.1. Problema del método de Monte-Carlo

El método de Monte-Carlo para los problemas que se plantea resolver implica tres principales retos:

1. Generación de números aleatorios
2. Camino aleatorio
3. Integral de Wiener

Estos elementos a su vez representan cierta complejidad computacional, en cuanto al tiempo de ejecución que requiere cada uno y los recursos de los cuales tendremos la necesidad de disponer al abordas la solución para cada elemento

1.2.2. Generación de números aleatorios

El método de Monte Carlo depende fuertemente de la rápida y eficiente generación de números aleatorios. Los números aleatorios de la simulación de Monte Carlo son usualmente generados por un algoritmo numérico, el cual es determinista; por lo tanto los números son solo pseudo-aleatorios. No elegir un buen algoritmo para la generación de los números aleatorios puede dar como resultado valores completamente erróneos [4].

Las simulaciones estocásticas y especialmente el método de Monte-Carlo usa variables aleatorias. Por lo tanto la capacidad para generar números aleatorios con una distribución especifica es necesario. El principal problema es encontrar algoritmos que provean números con un largo periodo, es decir que la secuencia tarde mucho en volver a repetirse [5].

Hay dos maneras de analizar la calidad de una secuencia de números aleatorios:

- Examinando las propiedades matemáticas del generador analiticamente o
- Someterlo a una bateria de pruebas estadisticas.

1.2.3. Camino aleatorio

En algún momento todos hemos observado la trayectoria de un camino aleatorio. Un ejemplo simple son las partículas de polvo danzando sin un patrón aparente al contacto con los rayos solares. Las partículas se mueven de un lado a otro, de arriba abajo sin seguir trayectorias fijas, siendo completamente aleatorio su movimiento. En física al movimiento de estas partículas se le llama movimiento browniano y toma su nombre del botánico francés Robert Brown quien observó que pequeñas partículas de polen se desplazaban de manera aleatoria aparentemente sin razón alguna.

La generación de trayectorias aleatorias es otro de los puntos clave para tener una ejecución exitosa de la simulación de Monte Carlo, pero esto deriva en un mayor consumo de la memoria y por tanto debe hacerse un uso eficiente de ésta para lograr alcanzar un buen resultado [6].

1.2.4. Integral de Wiener

Para entender mejor lo que es un proceso de Wiener pensemos en un individuo que tiene una riqueza inicial igual a W(0) unidades monetarias, un tiempo t más tarde el individuo extraerá una muestra de una variable aleatoria distribuida normalmente con media 0 y varianza 1 y el valor de la muestra será añadido o extraído de la riqueza inicial. Ahora su riqueza es igual a W(1) unidades monetarias. Volveremos a repetir el experimento extrayendo una nueva muestra a partir de una variable aleatoria con las mismas

características que la anterior. La muestra extraída se vuelve a añadir o sustraer de la de la riqueza anterior y ahora su riqueza es W(2). El experimento se repetirá sucesivamente hasta que al cabo de t periodos, la riqueza del individuo sea W(t). La riqueza acumulada en t periodos puede describirse con la siguiente expresión:

$$W(t-1) = W(t) + e_{t+1}; W(0) = W_0; e_t \sim N(0,1) \tag{1.1}$$

La función W(t) se denomina función aleatoria o bien, al ser el argumento de la función el tiempo es un proceso estocástico.

W(t) verifica las siguientes propiedades:

$$E_t[W(t+1) - W(t)] = E_t[e_(t+1)] = 0 \tag{1.2}$$

$$Var_t[W(t+1) - W(t)] = Var_t[e_(t+1)] = 1 \tag{1.3}$$

$$E_t[(W(t+1) - W(t))(W(t+j) - W(t+j-1))] = 0 \text{para todo} j > 0 \tag{1.4}$$

La propiedad [4] implica que las variaciones de W(t) que corresponden a dos intervalos de tiempo diferentes son independientes y esto se deriva del supuesto de que las variables W_t son variables aleatorias independientes.

Una propiedad importante de W(t) es el hecho de que el valor de W($t+1$) depende solamente de W(t), siendo totalmente independiente del camino que W haya seguido hasta llegar a t, es decir que la expectativa en t sobre el valor futuro de W no depende de cómo se llegó hasta W(t), de la historia pasada de W sino solo de su valor en t. Con esto, de cierto modo podríamos decir que el proceso estocástico no tiene memoria ya que los valores previos de W(t) no influirán en los posibles valores que pueda tomar en el futuro.

Esto se vuelve muy importante en la medida en que se vaya a utilizar este proceso estocástico para simular el comportamiento de una variable financiera (precio de una acción, un tipo de interés, el valor de un derivado, etc.) ya que esto supone admitir de forma implícita la hipótesis de los mercados eficientes[1].

En el ejemplo anterior no se especificó cuál sería el incremento del tiempo entre cada experimento. Ahora supongamos que el tiempo es un año, entonces los i ntervalos entre cada experimento serían de $1/n$. Siendo n un número mayor a 1, este nuevo proceso se puede describir como:

$$W(t + \Delta t) = W(t) + e_{t+\Delta t}$$

Siendo $\Delta t = 1/n; W(0) = W_0; e_{t+\Delta t} \sim N(0,1)$

Obtener una buena aproximación a la integral de Wiener es complejo y costoso de evaluar; por ello la aproximación se hace mediante métodos numéricos como el método de Monte Carlo que es un método no determinístico o estadístico numérico apropiado

[1]Hipótesis según la cual los precios de los activos reflejan la información relevante [7]

para aproximar la solución de este tipo de problemas. El método de Monte Carlo hace posible la realización de experimentos con muestreos de números pseudoaleatorios en una computadora. El error absoluto de este método decrece como $1/\sqrt{n}$ pero resolver esta integral a través del método de Montecarlo es costoso computacionalmente hablando.

1.3. Estructura de la tesis

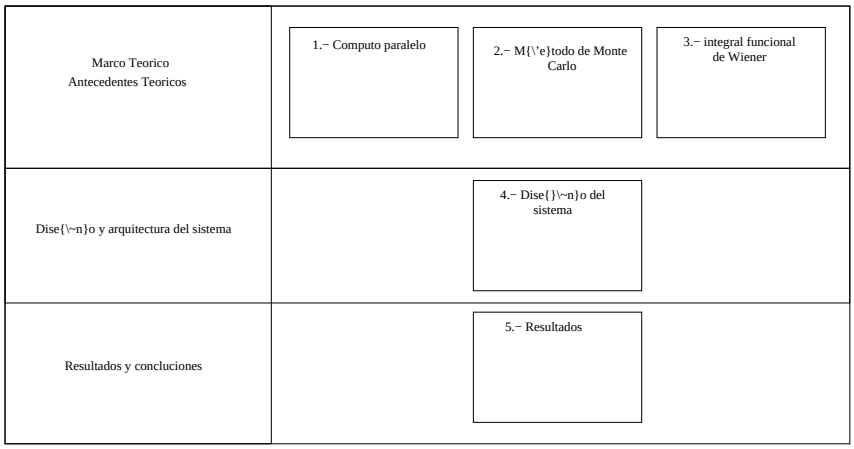

Figura 1.1: Esquema de Tesis

La tesis está organizada en 3 bloques a su vez se agrupan 6 capítulos. En el primer bloque se da a conocer el marco teórico que fundamenta esta tesis. Este bloque comprende los capítulos 2, 3 y 4. El siguiente bloque es dedicado a describir el desarrollo y es comprendido por el capítulo 5. Finalmente en el ultimo bloque se presentan las conclusiones y los resultados obtenidos (capitulo 6).

El capítulo 2: "Computo Paralelo" presenta diferentes enfoques para la realización de computo paralelo: clústers, multiprocesador, multinucleo, computo distribuido y GPUs. Ahondamos en la arquitectura general de las GPUs de la familia NVIDIA. Se da una visión general del compilador y las herramientas desarrolladas por NVIDIA para hacer posible la programación de propósito general sobre sus tarjetas gráficas. En este capitulo se pretende introducir al lector al computo paralelo y darle a conocer la importancia de su uso en problemas grandes y complejos.

El capítulo 3: "El método de Monte Carlo" Presenta la importancia que representa el uso del método de Monte Carlo, su naturaleza, las variaciones del método y la generación de las diferentes distribuciones de números aleatorios que Montecarlo requiere según la variación con la que se trabaja. Entre estos se encuentra la generación de números aleatorios continuos y la generación de número aleatorios discretos, por ultimo se mencionan

algunos ejemplos donde Monte Carlo juega un papel fundamental.

El capítulo 4: "Integral funcional de Wiener" presenta la integral funcional de Wiener, así como los algoritmos utilizados para la construcción de trayectorias, según sus propiedades. Por último se muestra el algoritmo en CUDA que fue utilizado para la generación de trayectorias en forma paralela.

El capítulo 5: "Diseño del sistema" presenta el diseño general de la arquitectura desarrollada para este proyecto. Se muestra el diseño realizado tanto en la versión secuencial como en la paralela y los requerimientos necesarios para la entrada y salida de datos.

El capitulo 6: "Resultados y conclusiones" presenta los resultados obtenidos de las ejecuciones paralelas y secuenciales y las conclusiones a las que se llego sobre este trabajo.

Capítulo 2

Computo paralelo

2.1. Introducción

El computo paralelo es una tendencia que ha cobrado gran importancia en los últimos años. Problemas que antes podían tardar días o meses en ser resueltos ahora con una correcta paralelización pueden reducir su tiempo de ejecución a unas cuantas horas. Anteriormente soló instituciones gubernamentales, empresas o grandes universidades podían tener acceso a supercomputadoras para ejecutar ahí cálculos y simulaciones intensivas. Hoy en día se cuenta con una gran cantidad de opciones para hacer computo intensivo. Las tarjetas gráficas o GPUs (Graphics Processing Unit) por sus siglas en ingles brindan la posibilidad de hacer computo de propósito general. Para fomentar el uso de las GPUs en la comunidad científica los fabricantes de éstas han desarrollado lenguajes de programación fáciles de aprender como CUDA (Compute Unified Device Architecture). CUDA comprende un conjunto de herramientas de desarrollo y un compilador que fue desarrollado por NVIDIA[1].

2.2. Computo paralelo

La complejidad de los problemas que se intentan resolver a través de las computadoras es cada vez mayor. Anteriormente se tenían problemas que podían consumir mucho tiempo en resolverse. La computación de hoy en día se ha dado a la tarea de buscar la manera de desarrollar herramientas que ayuden a la reducción del tiempo de ejecución que involucra la solución de problemas complejos. Entre las aplicaciones en las que la reducción de el tiempo ha resultado útil se encuentran la predicción del clima, la exploración petrolera, el procesamiento de imágenes entre muchas otras.

[1]Empresa multinacional especializada en el desarrollo de unidades de procesamiento gráfico y tecnologías de circuitos integrados para estaciones de trabajo, ordenadores personales y dispositivos móviles

El paralelismo explota la idea de dividir un problema grande en varios mas pequeños y resolver estos al mismo tiempo. Se puede realizar una paralelización a nivel de bit, de instrucción, de datos o de tarea. El tipo de paralelización a llevar a cabo dependerá del tipo de problema que se tenga. Las capacidades de hardware que se tienen deben de permitir llevar a cabo este tipo de procesamiento, pero también es importante contar con software adecuado que soporte la ejecución y coordinación de procesos paralelos.

El paralelizmo puede ser introducido a varios niveles [8]. En el nivel mas bajo se puede añadir paralelismo a nivel del chip a través de pipeline y diseños superescalares con multiples unidades funcionales. tambien podria ser añadido teniendo palabras muy largas de instruccion con un paralelismo implicito. Carateisticas especiales podrian ser añadidas a una CPU para permitirle manejar multiples hilos de control a la vez. Finalmente muchas CPUs pueden ser puestas en el mismo chip. Estas caracteristicas podrian mejorar el rendimiento en un factor de 10 sobre los diseños puramente secuenciales.

En el siguiente nivel las CPUs adicionales con capacidades de procesamiento adicional podrían ser añadidas. generalmente estas CPUs conectables tienen funciones especializadas tales como el procesamiento de paquetes de red, el procesamiento multimedia o la criptografía. Sin embargo, para ganar un factor de cien o mil o un millón, es necesario replicar CPUs enteras y hacerlas trabajar juntas de manera eficiente. Esta idea conduce a grandes multiprocesadores y multicomputadoras (clúster).

Por último, ahora es posible conjuntar a organizaciones enteras a trav'es de Internet para formar redes de cómputo muy débilmente acoplados. Cuando dos CPU o elementos de procesamiento están muy juntos, tienen un alto ancho de banda y bajo retardo entre ellos y los componentes interactuan unos con otros se dice que están fuertemente acoplados. de modo opuesto cuando están muy separados, tienen un ancho de banda bajo y alto retardo y son computacionalmente independientes, se dice que están débilmente acoplados. La figura 2.1 muestra las graficamente las opciones que se han descrito.

2.2.1. Paralelismoa nivel de chip

Una forma de aumentar el rendimiento de un chip es haciendo mas cosas a la vez. En otras palabras, explotar el paralelismo. En esta seción se abordatran algunas de las maneras que existen para explotar el paralelismo a nivel de chip. Se verá el paralelismo a nivel de instrucción, multihilo, y poner más de una CPU en el chip. Las tecnicas anteriores son muy diferentes, pero cada una de diferente manera ayuda a lograr hacer mas cosas al mismo tiempo.

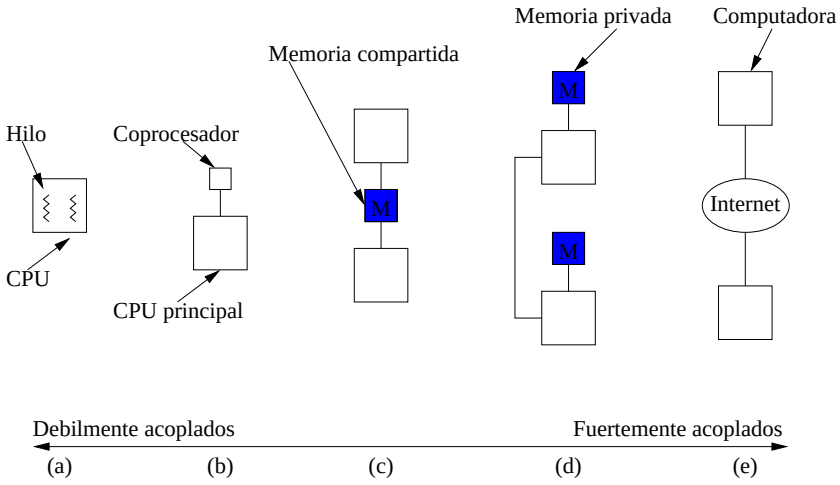

Figura 2.1: (a) Paralelismo a nivel de chip. (b) Un coprocesador. (c) Multiprocesador. (d) Una multicomputadora. (e) Computo en Grid

2.2.1.1. Paralelismoa a nivel de instrucción

Una manera de conseguir el paralelismo es emitir múltiples instrucciones por ciclo de reloj . CPUs de múltiples temáticas vienen en dos variedades: los procesadores superescalares y procesadores VLIW. CPUs superescalares son capaces de emitir múltiples instrucciones a las unidades de ejecución en un sólo ciclo de reloj. El número de las instrucciones que en realidad se emiten depende tanto del diseño del procesador ycomo de las circuntancias en el momento de la emision. El hardware limita el número máximo que se puede emitir. Sin embargo, si una instrucción necesita una unidad funcional que no está disponible o un resultado que todavía no se ha calculado, la instrucción no será emitida.

La otra forma de paralelismo a nivel de instrucción se encuentra en los procesadores VLIW (Very Long Instruction Word). En la forma original, las máquinas VLIW efectivamente tuvieron palabras largas que contienen instrucciones que utilizan múltiples unidades funcionales. Consideremos, Por ejemplo, el pipeline de la figura 2.2, si la máquina cuenta con cinco unidades funcionales y puede realizar dos operaciones de enteros , una operación de punto flotante , una carga y un almacenamiento de forma simultánea. Una instrucción VLIW para esta máquina contendría cinco códigos de operación y cinco pares de operandos, uno de código de operación y otro par de operandos por unidad funcional. Con 6 bits por código de operación, 5 bits por registro, y 32 bits por dirección de memoria, las instrucciones podrían ser fácilmente 134 bits.

sin embargo este diseño tiene la desventaja de que no todas las instrucciones son capases de usar todas las unidades funcionales a la vez por lo que habría que rellenar con no operación y dependiendo de cada instrucción se desaprovecharían unidades funcionales.

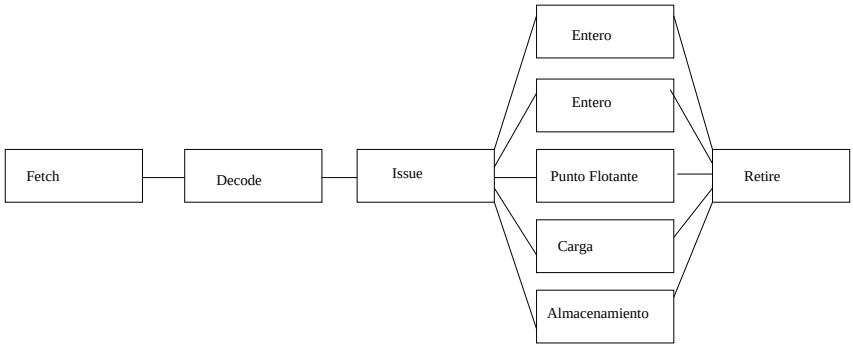

Figura 2.2: Una CPU pipeline

Las maquinas VLIW modernas delegan al compilador la responsabilidad de conjuntar paquetes de instrucciones que puedan ocupar todas las unidades funcionales.

2.2.1.2. Multithilo

Todas las modernas CPUs pipeline tienen un problema inherente: cuando se hace una referencia de memoria a un elemento que no esta en el nivel 1 ni el nivel 2 de caché, hay una larga espera hasta que la palabra solicitada (y su línea caché asociada) se cargan en la memoria caché. Un enfoque para hacer frente a esta situación, es llamado multihilo, ésto permite a la CPU manejar múltiples hilos de control, al mismo tiempo, en un intento de ocultar estas paradas. En resumen, si el hilo 1 está bloqueado, la CPU todavía tiene una oportunidad de mandar a ejecutar al hilo 2 con el fin de mantener el hardware totalmente ocupado.

Aunque la idea básica es bastante simple, existen múltiples variantes. multihilo de grano fino, de grano grueso y simultaneo. Los procesadores multihilo contienen hardware de estado (contador de programa y registros) para varios hilos. Dado un ciclo, el procesador ejecuta instrucciones de uno de los hilos. En el ciclo siguiente, cambia a un contexto diferente y ejecuta instrucciones del nuevo hilo. Pero aun así, este tipo de procesadores siguen limitados por el paralelismo de instrucciones en un único hilo.

En una arquitectura multihilo de grano grueso, en cada ciclo se ejecutan instrucciones de un hilo, en el momento en que este hilo sufre una parada, debida quizás a un fallo de caché, es cuando se da un cambio de contexto y entra a ejecutarse un nuevo hilo.

un procesador multihilo simultaneo puede seleccionar múltiples instrucciones de múltiples hilos cada ciclo desde un único pipeline. Un diseño de éste tipo explota el paralelismo a nivel de instrucción al seleccionar instrucciones de varios hilos [9].

2.2.1.3. Multiprocesadores en un solo chip

El enfoque multihilo proporciona importantes mejoras de rendimiento a bajo costo, pero en algunas aplicaciones se requiere una ganancia de rendimiento mucho mayor. Para obtener esta ganancia, se están desarrollando los chips multiprocesador. Dos áreas en las que estos chips, que contienen dos o más CPUs, son de interés es en servidores de gama alta y en electrónica de consumo. Ahora vamos a hablar brevemente de cada uno de ellos.

2.2.1.3.1. Multiprocesadores homogéneos en un chip Hoy en día es posible tener dos o más potentes CPU en un solo chip . Dado que estas CPUs a menudo comparten la misma caché de nivel 2 y la memoria principal, el beneficiario es un multiprocesador. Para los pequeños multiprocesadores de un solo chip, dos diseños son frecuentes . En el primero, en realidad sólo hay un chip, pero tiene una segunda pipeline, que podría duplicar la tasa de ejecución de la instrucción. En el segundo, hay núcleos separados sobre el chip. Un núcleo es un gran circuito, tal como una CPU, controladores de E/S o caché, que pueden ser colocados en un chip de forma modular, por lo general junto a otros núcleos.

2.2.1.3.2. Multiprocesadores heterogéneos en un chip Un área de aplicación completamente diferente de los multiprocesadores de un solo chip está en los sistemas embebidos, especialmente en la electrónica de consumo audiovisual, tales como televisores, reproductores de DVD, videocámaras, consolas de videojuegos, teléfonos celulares, etc. Estos sistemas tienen requisitos de rendimiento y fuertes exigentes restricciones. Aunque estos dispositivos tienen un aspecto diferente, más y más de ellos son simplemente computadoras pequeñas, con una o más CPUs, memorias y controladores de E/S.

2.2.2. Coprocesadores

la Computadora puede ser acelerada mediante la adición de un segundo procesador, especializado. Estos coprocesadores vienen en muchas variedades. Incluso un chip de DMA (Direct Memory Access) puede ser visto como un coprocesador. En algunos casos, la CPU da al coprocesador una instrucción o conjunto de instrucciones para su ejecución; en otros casos, el coprocesador es más independiente y se ejecuta prácticamente él mismo.

Físicamente los coprocesadores pueden ir desde un gabinete separado a un área en el chip principal. En todos los casos, lo que los distingue es el hecho de que algún otro procesador es el procesador principal y los coprocesadores están ahí para ayudarlo. Ahora vamos a examinar tres áreas en lascuales es posible lograr una aceleración a través de coprocesadores: procesamiento de red, multimedia, y de la criptografía.

2.2.2.1. procesamiento de red

La mayoría de los ordenadores en la actualidad están conectados a una red o a Internet. Como resultado de los avances tecnológicos en hardware de red, las redes son ahora tan rápidas que se ha vuelto cada vez más difícil de procesar todos los datos entrantes y salientes en software. Como consecuencia de ello, los procesadores de red especiales se han desarrollado para manejar el tráfico y muchos ordenadores de gama alta ahora tienen uno de estos procesadores.

2.2.2.2. Procesadores grafios

Una segunda área en la que se utilizan los coprocesadores es para manejar el procesamiento de gráficos de alta resolución, como la representación 3D. CPUs ordinarios no son especialmente buenos en los cálculos masivos necesarios para procesar la gran cantidad de datos necesarios para estas aplicaciones. Por esta razón, la mayoría de las PC y muchos procesadores futuros estarán equipados con GPUs (Grafic Unit Processing) a los que se pueden descargar una gran parte de procesamiento general.

2.2.2.3. criptoprocesadores

Una tercera área en la que los coprocesadores son populares es la seguridad, especialmente la seguridad de red. Cuando se establece una conexión entre un cliente y un servidor, en muchos casos, primero deben autenticarse entre sí. A continuación, una conexión encriptada y segura ha de establecerse entre ellos para que los datos pueden ser transferidos de una forma segura para frustrar cualquier intento de intrusión que pueden aprovechar la línea.

El problema de la seguridad es que para lograrlo, se debe usar la criptografía y la está requiere de cómputo intensivo. Criptografía viene en dos formas generales , llamados criptografía de clave simétrica y la criptografía de clave pública. El primero se basa en mezclar los bit muy a fondo, más o menos el equivalente de lanzar un mensaje en una licuadora eléctrica . Este último se basa en la multiplicación y exponenciación de un gran número (por ejemplo, 1024 bits) y es extremadamente lento.

Para manejar el cálculo necesario para cifrar datos de forma segura para la transmisión o el almacenamiento y luego descifrarlos más tarde, varias empresas han producido coprocesadores criptográficos, a veces como tarjetas enchufables de bus PCI . Estos coprocesadores tienen hardware especial que les permite realizar las operaciones de criptografía necesaria mucho más rápido de lo que una CPU normal puede hacerlo.

2.2.3. Multiprocesadores

Un ordenador paralelo en el que todas las CPU comparten una memoria común se llama un multiprocesador, como se indica. Todos los procesos que trabajan juntos en un multiprocesador pueden compartir un único espacio de direcciones virtual asignado a la memoria común . Cualquier proceso puede leer o escribir una palabra de memoria con sólo ejecutar una instrucción LOAD o STORE. No se necesita nada más. El hardware se encarga del resto. Dos procesos pueden comunicarse simplemente si uno de ellos escribe datos en la memoria y el otro los lee después de esta escritura.

La capacidad de los dos (o más) procesos para comunicarse con sólo leer y escribir en la memoria. Es un modelo fácil para los programadores y es aplicable a una amplia gama de problemas.

Debido a que todas las CPU en un multiprocesador ven la misma imagen de memoria, sólo hay una copia del sistema operativo . En consecuencia, sólo hay una página del mapa de memoria y una tabla de procesos. Cuando un proceso se bloquea, la CPU guarda su estado en las tablas del sistema operativo, a continuación, busca otro proceso para que se ejecute. Es esta imagen de sistema único que distingue a un multiprocesador de una multicomputadora, en el que cada equipo tiene su propia copia del sistema operativo.

Un multiprocesador, al igual que todos los equipos, debe contar con dispositivos de E/S, tales como discos, adaptadores de red y otros equipos. En algunos sistemas multiprocesador, sólo ciertos CPUs tienen acceso a los dispositivos de E/S, y por lo tanto tienen una función especial de E/S . En otros, cada CPU tiene igual acceso a todos los dispositivos de E/S . Cuando cada CPU tiene igual acceso a todos los módulos de memoria y todos los dispositivos de E/S, y se tratan como intercambiables con los demás por el sistema operativo , el sistema se llama un SMP (Symmetric multiprocesador).

2.2.4. Multicomputadoras

El segundo diseño posible para una arquitectura paralela es una en la que cada CPU tiene su propia memoria privada , accesible sólo a sí mismo y no a cualquier otra CPU. Tal diseño se llama un multicomputadora, o, a veces un sistema de memoria distribuida. El aspecto clave de una multicomputadora que lo distingue de un multiprocesador es que cada CPU en una multicomputadora tiene su propia memoria privada, local que se puede acceder con sólo ejecutar las instrucciones de LOAD y STORE, pero que ninguna otra CPU puede acceder mediante las instrucciones de LOAD y STORE.

Los multiprocesadores tienen un único espacio de direcciones físico compartido por todas las CPUs , mientras que las multicomputadoras tienen un espacio de direcciones físicas por CPU. Dado que las CPUs en una multicomputadora no se pueden comunicar

con sólo leer y escribir la memoria común , necesitan un mecanismo de comunicación diferente. Lo que hacen es pasar mensajes de un lado a otro usando la red de interconexión. Como ejemplos de multicomputadoras podemos mencionar la IBM BlueGene/L [10].

Multicomputadoras se pueden dividir en dos categorías. la primera contiene los MPPs(procesadores masivamente paralelos), que son las supercomputadoras costosas que constan de muchas CPU fuertemente acopladas por una red de interconexión de alta velocidad.

La otra categoría consiste en PCs normales, estaciones de trabajo o servidores, posiblemente montados en anaqueles, y se conectan mediante la tecnología de interconexión comercial. Lógicamente, no hay mucha diferencia, pero enormes supercomputadoras que cuestan muchos millones de dólares se utilizan de manera diferente a las redes de ordenadores ensamblados por los usuarios a una fracción del precio de un MPP. Estas máquinas son conocidas por diversos nombres, entre ellos NOW (Red de Estaciones de Trabajo) y COW(cúmulo de estaciones de trabajo), o a veces sólo clúster.

2.2.5. Computo en grid

Muchos de los retos actuales de la ciencia, la ingeniería, la industria, el medio ambiente, y otras áreas son interdiciplinarios y de gran escala. Resolverlos requiere expertos, habilidades, conocimientos, instalaciones, software, y datos de varias organizaciones, a menudo en diferentes países.

Algunos de las tareas en las que se trabaja en conjunto son a largo plazo, otros son más a corto plazo, pero todos comparten el denominador común de exigir a organizaciones separadas trabajar en conjunto con sus propios recursos y procedimientos para lograr un objetivo común.

Con diferentes organizaciones, con diferentes ordenadores, sistemas operativos , bases de datos y protocolos trabajando juntos para compartir recursos y datos ha sido muy difícil. Sin embargo, la creciente necesidad de cooperación entre organizaciones a gran escala ha llevado al desarrollo de sistemas y tecnología para la conexión de computadoras muy distantes entre sí en lo que es llamado la grid.

El objetivo de la grid es proporcionar una infraestructura técnica para permitir que un grupo de organizaciones que comparten un objetivo común formen una organización virtual. Esta organización virtual tiene que ser flexible, con muchos y cambiantes miembros, permitiendo a los miembros trabajar juntos en áreas que consideren oportunas, mientras se les permite mantener el control sobre sus propios recursos al grado que deseen.

2.2.6. Metricas de Desempeño

La razón por la que se contruyen computadoras paralelas es lograr tener un mejor desempeño que el que se tiene con el modelo uniprocesador. Esta mejora en la velocidad de los procesos tambien debe ser alcansada con eficiencia de costos. Un problema que se resuelve en una maquina que logra hacerlo dos veces mas rapido, pero que cuesta 100 veces más tal vez no seria rentable. Mencionaremos algunos de los aspectos de desempeño asociados con las arquitecturas paralelas.

2.2.6.1. metricas de hardware

Desde el punto de vista del hardware las metricas que interesan son la velocidad de la CPU y de E/S. las velocidad de la CPU y de E/S son las mismas que en el uniprocesador, de modo que los parametros de interes en un sistema paralelo seran los asociados con la interconexión. los dos elementos clave son la latencia y el ancho de banda. La latencia de viaje redondo es el tiempo que tarda una CPU en enviar un paquete y recibir una respuesta. si el paquete es enviado a una memoria entonces ésta mide el tiempo que toma leler o escribir una palabra o un bloque de palabras. si el paquete es enviado a otra CPU entonces se mide el tiempo de comunicacion interprocesador para paquetes con ese tamaño.

La latencia comprende varios factores y es diferente para las interconexiones de comunicación de circuitos, de almacenamiento y reenvío, de corte virtual y de enrrutamiento por tunel. En el caso de la conmutación de circuitos la latencia es la suma del tiempo de transmisión y el tiempo de preparación. Para preparar un circuito es necesario enviar un paquete de "sondeo" que se encargara de reservar los recursos y de informar de los resultados de su envío. Después se debe armar un paquete de datos y una vez listo entonces es enviado. Si el tiempo de preparación es t_s, el tamaño del paquete el p bits y el ancho de banda en de b bits/s, la latencia en un sentido sera $T_s + p/b$. si el circuito permite que los paquetes fluyan en ambas direcciones entonces sin haber tiempo de preparación para la respuesta, la latencia mínima para enviar un paquete y recibir la respuesta sería $T_s + 2p/b$ segundos.

En el caso de la conmutación de paquetes no es necesario enviar un paquete de sondeo. El tiempo que lleva armar el paquete es T_a entonces el tiempo de transmisión en un sentido sería $T_a + p/b$, pero éste es sólo el tiempo que tarda en llegar al primer conmutador. Llamaremos T_d al retardo finito que hay dentro de cada conmutador. Si hay n conmutadores, la latencia total en un sentido sería $T_d + n(p/b + T_d) + p/b$ el último término es debido al envió del último conmutador al destino.

Las latencias en un sentido para el corte virtual y el enrutamiento a través del túnel en el mejor de los casos son cercanas a $T_a + p/b$ debido a que no hay que enviar un paquete

de sondeo y no existen los retrasos asociados al almacenamiento y reenvío. Básicamente la latencia consiste en el tiempo de preparación del paquete de datos mas el tiempo que le toma llegar a su destino, en todos los casos se debe sumar el retraso de propagación, pero éste es generalmente pequeño.

La otra métrica asociada al hardware es el ancho de banda. El número de bits que el sistema es capaz de transmitir es crítico para el desempeño. Existen diferentes métricas para medir el ancho de banda.
Para calcular el ancho de banda bisectriz se debe dividir de forma conceptual la red en dos partes iguales pero disconexas eliminando un conjunto de aristas de su grafo. Se calcula el ancho de banda de las aristas eliminadas . Puede haber muchas distintas formas de dividir la red. El ancho de banda bisectriz es el mínimo de todas las divisiones posibles. El ancho de banda agregado es la suma de las capacidades de todos los enlaces. Lo que nos proporciona la máxima cantidad de bits que pueden estar en transito en un determinado momento. También existe el ancho de banda medio en la salida de cada CPU.

Es muy importante tener baja latencia y un buen ancho de banda aunque los programadores prefieren reducir la latencia primero y después preocuparse por como aumentar el ancho de banda.

2.2.6.2. Métricas de software.

Un usuario está interesado en que tanto va a aumentar la velocidad de la ejecución de sus programas con respecto a la versión ejecutada en un uniprocesador. La métrica importante para ellos es la aceleración. La aceleración es el indicador de cuantas veces mas rápido se ejecuta un programa en un sistema de n procesadores en comparación de uno con un procesador.

Parte de la razón por la cual es casi imposible lograr una aceleración perfecta es que casi todos los programas cuentan con una parte secuencial, como la fase de inicialización, la lectura de los datos, la reunión de los resultados. En estos caso contar con muchas CPUs o coprocesares especializados no es de gran ayuda. Supongamos que un programa se ejecuta durante T segundos en un uniprocesador, y que una fracción f de este tiempo corresponde a un código secuencial y una fracción $1 - f$ es potencialmente paralelizable. Si este ultimo código se puede ejecutar en n CPU sin gasto extra su tiempo de ejecución podrá reducirse de $(1 - f)T$ a $(1 - f)T/n$ en el mejor de los casos. Esto deriva en un tiempo de ejecución total $fT + (1 - f)T/n$. la aceleración no es mas que el tiempo de ejecución del programa original, T, dividido entre el nuevo tiempo de ejecución.

$$Aceleracion = n/1 + (n - 1)f \tag{2.1}$$

Con $f = 0$ se puede obtener una aceleración lineal, pero con $f > 0$ no se puede lograr la aceleración perfecta debido a la parte secuencial. Este resultado se conoce como la ley de

Amdahl.

Otra ley importante considerar que ademas está en estrecha relación con la ley de Amdahl es la ley de Gustafson [11]. Si usamos s y p para representar el tiempo secuencial y el tiempo paralelo en un sistema paralelo el procesador secuencial requería $s + ps$ para desarrollar la tarea. Este razonamiento da una pauta a la ley de Amdahl.

$$Aceleracion = (s + pn)/(s + p) = s + pn = n + (1 - n)s \tag{2.2}$$

La ley de Gustafson supone que la cantidad total de trabajo que se hará en paralelo varía linealmente con el número de procesadores. Amabas leyes asumen que el tiempo secuencial es independiente del numero de procesadores.

Es importante considerar que estas leyes no son los únicos aspectos que limitan la aceleración, también están el ancho de banda y la latencia. Estos aspectos deben de ser considerados por un programador a la hora de buscar acelerar cualquier algoritmo.

2.3. clasificación de las computadoras paralelas

Clasificar las computadoras paralelas no ha sido una labor fácil, sin embargo investigadores lo han intentado y logrado con cierto éxito. El esquema mas usado es el que propuso Michael Flynn en 1975 [12]. La taxonomía de Flynn se basa en los flujos de instrucciones y en los flujos de datos. El flujo de instrucciones corresponde a un contador de programa y el flujo de datos en una serie de instrucciones. Se pueden considerar independientes los flujos de datos y de instrucciones por lo cual es posible encontrar las clasificaciones de la Figura 2.3

Flujo de instrucciones	Flujo de datos	Nombre
1	1	SISD
1	Multiple	SIMD
Multiple	1	MISD
Multiple	Multiple	MIMD

Figura 2.3: Taxonomia de Flynn de las computadoras paralelas.

2.3.1. Máquina SISD

éste es el modelo tradicional de computación secuencial visto como un simple flujo de instrucciones que se ejecutan una tras otra sobre los mismos datos. la figura 2.4 muestra un esquemas de la maquina SISD.

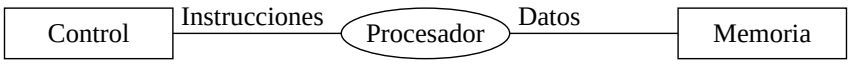

Figura 2.4: Máquina SISD

Un ejemplo sencillo sería pensar en la suma de todos los elementos de un vector A de N elementos $A_1, A_2, ...A_N$. Es necesario que el procesador acceda a memoria N veces para obtener los datos y después realice N-1 sumas para obtener el resultado final.

2.3.2. Máquina SIMD

En este tipo de maquinas solo existe una unidad de control que es la enargada de enviar las instrucciones a los diferentes procesadores o ALUs, para que cada uno ejecute la misma serie de instrucciones, pero sobre diferentes conjuntos de datos. Su posible representación se muestra en la Figura 2.5. Este tipo de maquinas son utilizadas para cálculos científicos.

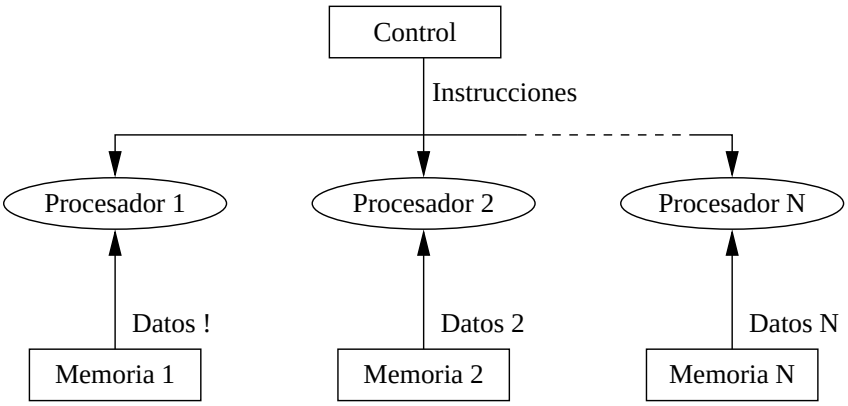

Figura 2.5: Máquina SIMD

Como ejemplo pensemos en dos vectores A y B de N elementos y queremos sumar $A_1 + B_1$, $A_2 + B_2$, hasta $A_N + B_N$ para dar origen a un nuevo vector C. Supongamos que también contamos con N procesadores, estoces cada procesador seria encargado de generar un nuevo elemento del vector C, pues a cada procesador le correspondería realizar una suma y en el tiempo de una instrucción se realizarían las N requeridas.

2.3.3. Máquina SDMI

En esta maquina Varias instrucciones operan sobre el mismo conjunto de datos y su representación se ilustra en la Figura 2.6. Está maquina no es muy usada en la practica.

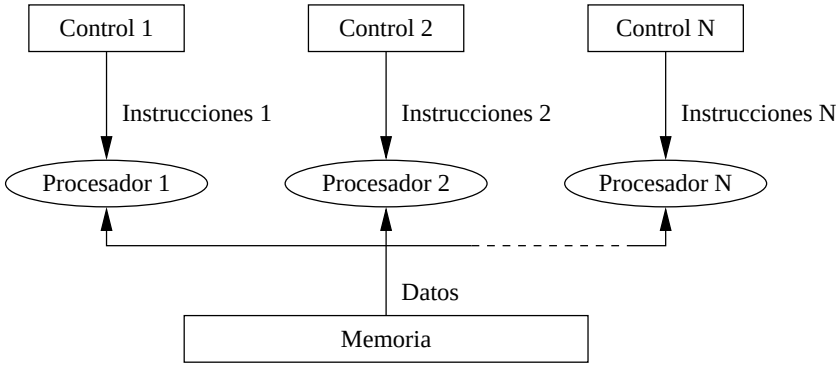

Figura 2.6: Máquina SDMI

Un ejemplo sencillo seria un par de matrices X y Y de tamaño NxN donde se debe hacer la suma, multiplicación y obtener la inversa de cada matriz. una serie de diferentes instrucciones deben operar sobre los mismos datos entonces cada procesador ejecutaría una operación diferente sobre las mismas matrices.

2.3.4. Máquina MDMI

Por ultimo se tienen la maquinas MIMD que se pueden ver como múltiples procesadores independientes que forman parte de un sistema mas grande. En la figura 2.7 se muestra un ejemplo.

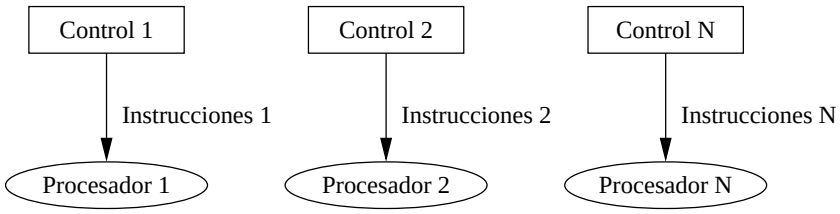

Figura 2.7: Máquina MDMI

en este caso cada procesador es libre de ejecutar su propio flujo de instrucciones con su propio flujo de datos simultÃ¡neamente.

2.4. GPUs y su arquitectura

2.4.1. Historia de la computacion sobre GPUs

Los procesadores centrales se desarrollaron sobre las velocidades de reloj y número de núcleos. Mientras tanto, el estado de procesamiento de gráficos se sometió a una dramática revolución. A finales de la década de 1980 y principios de la década de 1990, el crecimiento de la popularidad de los sistemas operativos impulsados gráficamente como Microsoft Windows ayudó a crear un mercado para un nuevo tipo de procesador. A principios de 1990, los usuarios comenzaron a comprar los aceleradores de visualización 2D para sus ordenadores personales. Estos aceleradores de visualización de mapa de bits ofrecen operaciones asistidas por hardware para ayudar en la visualización y la facilidad de uso de los sistemas operativos gráficos.

Por la misma época, en el mundo del computo profesional, una empresa con el nombre de Silicon Graphics pasó la década de 1980 popularizando el uso de gráficos en tres dimensiones en una variedad de mercados, incluyendo aplicaciones gubernamentales y de defensa y la visualización científica y técnica, así como proporcionar las herramientas para crear efectos cinematográficos asombrosos. En 1992, Silicon Graphics abre la interfaz de programación de su hardware mediante la liberación de la biblioteca OpenGL. Silicon Graphics destinó OpenGL a ser utilizado como un método estandarizado, independiente de la plataforma para escribir aplicaciones gráficas en 3D. Al igual que con el procesamiento y la CPU en paralelo, sólo sería cuestión de tiempo antes de que las tecnologías encontraran su camino en las aplicaciones de consumo general.

A mediados de la década de 1990, la demanda de aplicaciones de consumo que emplean gráficos 3D se había intensificado rápidamente, preparando el escenario para dos desarrollos bastante significativos. En primer lugar, el lanzamiento de juegos en primera persona tales como Doom, Duke Nukem 3D, Quake y ayudó a iniciar una búsqueda para crear progresivamente entornos 3D más realistas para juegos de PC. Aunque los gráficos 3D eventualmente trabajarían en casi todos los juegos de ordenador, la popularidad del género de los videojuegos de diparos aceleraría significativamente la adopción de los gráficos 3D en la computación de consumo. Al mismo tiempo, empresas como NVIDIA, ATI Technologies, y 3Dfx Interactive comenzaron a liberar los aceleradores de gráficos que eran lo suficientemente costeables y atractivos como para atraer la atención general.

El lanzamiento de la NVIDIA GeForce 256 empujó aún más las capacidades de hardware de gráficos de consumo. Por primera vez, los cálculos de transformación e iluminación se podrían realizar directamente en el procesador de gráficos, mejorando de este modo el potencial para incluso más aplicaciones visualmente interesantes. Desde transformación e iluminación ya eran parte integrante de la pipeline de gráficos OpenGL, la GeForce 256

marcó el comienzo de una progresión natural en el que cada vez más de la pipeline de gráficos se aplicaría directamente en el procesador gráfico.

La liberación de NVIDIA de la serie GeForce 3 en 2001 representa sin duda el avance más importante en la tecnología de GPU. La serie GeForce 3 fue el primer chip de la industria de la computación para poner en práctica el entonces nuevo estándar de Microsoft DirectX 8.0. Esta norma requiere que el hardware compatible contenga tanto vértices programables y las etapas de sombreado de píxeles programables. Por primera vez, los desarrolladores tenían algún control sobre los cálculos exactos que se pueden realizar en sus GPU.

2.4.2. Las GPUs como computadoras paralelas

Desde 2003, la industria de los semiconductores se ha asentado sobre dos trayectorias principales para el diseño de microprocesadores. La trayectoria multicore busca mantener la velocidad de ejecución de los programas secuenciales mientras se mueve en múltiples núcleos. Los multicores comenzaron como procesadores de doble núcleo, con el número de núcleos de aproximadamente el doble con cada generación de procesos de semiconductores. Un ejemplar actual es el reciente microprocesador core i7 de Intel. El microprocesador hace uso del hyperthreading con dos hilos de hardware y está diseñado para maximizar la velocidad de ejecución de los programas secuenciales. En contraste el enfoque many-core se enfoca en el rendimiento en la ejecución de programas paralelos. Estos comenzaron como un gran número de núcleos mucho más pequeñas y el número de núcleos se duplica con cada generación. Un ejemplar actual es la unidad de procesamiento gráfico (GPU) GeForce GTX 280 de NVIDIA con 240 core. Procesadores many-core especialmente las GPUs, han llevado la carrera del rendimiento en punto flotante desde 2003. Las mejoras en el desempeño de los microprocesadores de propósito general se ha reducido de manera significativa mientras que las GPUs han seguido mejorando sin cesar.

Esta significativa diferencia ya ha hecho que muchos desarrolladores cambien la ejecución de las partes de cálculo complejas a GPUs. No es sorprendente que estas partes de cálculo complejas también son el blanco principal de la programación paralela cuando hay más trabajo por hacer, hay más oportunidad de dividir el trabajo entre elementos que cooperan de forma paralela.

La diferencia que existe entre los procesadores multicore y los many-core está en sus diferentes filosofías de diseño fundamentales entre los dos tipos de procesadores, como se ilustra en la Figura 2.8 El diseño de una CPU está optimizado para el rendimiento del código secuencial. Hace uso de la sofisticada lógica de control para permitir que las instrucciones de un solo hilo se ejecuten en paralelo o incluso fuera de su orden secuencial, manteniendo la apariencia de ejecución secuencial. Más importante aún, las grandes memorias caché se proporcionan para reducir las latencias de acceso a las instrucción y

de acceso a datos de grandes aplicaciones complejas. Ni la lógica de control ni memorias caché contribuyen a la velocidad de cálculo. A partir de 2009, los nuevos microprocesadores de propósito general multicore suelen tener cuatro grandes núcleos de procesamiento diseñados para ofrecer un fuerte rendimiento del código secuencial.

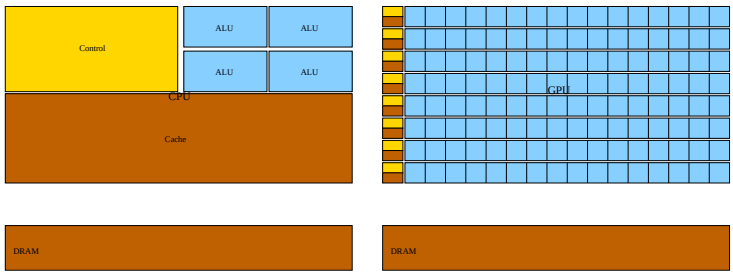

Figura 2.8: Diferencia de las filosofias de diseño de las GPUs y las CPUs

El ancho de banda de memoria es otro factor importante. Chips gráficos han estado operando a aproximadamente 10 veces el ancho de banda de los chips de CPU disponibles simultáneamente. A finales de 2006, la GeForce 8800 GTX, o simplemente G80, fue capaz de mover datos a cerca de 85 gigabytes por segundo (GB/s) dentro y fuera de su principal memoria dinámica de acceso aleatorio (DRAM).

La filosofía de diseño de las GPUs es seguida por el rápido crecimiento de la industria de los videojuegos, que ejerce una enorme presión económica para la capacidad de realizar una enorme cantidad de cálculos de punto flotante por frame de vídeo en los juegos avanzados. Esta demanda motiva a los vendedores de la GPU para buscar formas de maximizar el área del chip y el presupuesto de alimentación dedicada a cálculos de punto flotante. La solución que ha prevalecido hasta la fecha es la de optimizar el rendimiento de la ejecución a través de un masivo número de hilos. El hardware se aprovecha de un gran número de hilos de ejecución para encontrar trabajo que hacer cuando algunos de ellos están en espera por grandes latencias de acceso a la memoria, minimizando así la lógica de control necesaria para cada subproceso de ejecución. Las pequeñas memorias caché se proporcionan para ayudar a controlar los requisitos de ancho de banda de estas aplicaciones, de modo que varios subprocesos que tienen acceso a los mismos datos de la memoria no necesitan ir la DRAM. Como resultado de ello, una mayor área de chip está dedicado a los cálculos de punto flotante.

Dentro de un programa se cuenta con partes inherentemente secuenciales y del mismo modo otras paralelas por eso se vuelve necesaria la ejecución de un programa de manera conjunta usando la CPU para las partes secuenciales y la GPU para aquellos cálculos complejos que son altamente paralelizables. ésto se vuelve posible gracias al modelo de programación CUDA (Compute Unified Device Architecture) que se diseño para soportar

la ejecución conjunta entre CPU yGPU. El hecho de que cada vez se hayan popularizado mas el uso de los GPUs a derivado en una disminución de sus costos lo que ha permitido que la mayoría de las computadoras de escritorio y portátiles cuenten con una GPU sin ver en gran medida una elevación de su costo. Como resultado de las ventajas que la GPU ofrece sobre la CPU se puede esperar que cada vez mas aplicaciones complejas sean programadas o migradas para su ejecución sobre una GPU.

Además de todas las ventajas ya mencionadas NVIDIA para promover entre los científicos e investigadores el uso de las GPUs se dio la tarea de desarrollar CUDA que es una extensión de C y que elimina la tediosa tarea de usar APIs gráficas para llevar una aplicación a ejecutarse sobre una GPU. Sobre CUDA ahondaremos en secciones posteriores.

2.4.3. Arquitetura general de una GPU

La Figura 2.9 muestra la arquitectura típica de una GPU habilitada con CUDA. Se organiza en una serie de multiprocesadores de streaming multiprocessors (SMs). En la Figura 2.9, dos SMs forman un bloque; Sin embargo, el número de SMs en un bloque puede variar de una generación de GPUs CUDA a otra generación. Además, cada SM en la Figura 2.9 tiene una número de streaming procesors (SPs) que comparten la lógica de control y las instrucciones en caché. Cada GPU actualmente viene con hasta 4 gigabytes de Graphics Double Data Rate (GDDR) DRAM, referida como memoria global en la Figura 2.9. Estos GDDR DRAM difieren de las DRAM del sistema en la placa base de CPU en que está diseñada para tarjetas gráficas y, al igual que la memoria DRAM de la CPU, funcionaba según el estándar DDR, enviando dos bits por cada ciclo de reloj, pero en este caso los módulos de memoria GDDR se optimizan para lograr altas frecuencias de reloj acortando los tiempos de acceso de las células de memoria en comparación con la memoria DDR convencional. Esto es necesario dada la gran cantidad de datos que tienen que procesar las tarjetas gráficas. En aplicaciones gráficas esta memoria funciona para mantener imágenes de video y texturas de información para rendering tridimencional (3D) pero para computo general funciona como un muy alto ancho de banda, aunque con algo más de latencia de la memoria del sistema típico. Para aplicaciones masivamente paralelas, el ancho de banda superior compensa el aumento en la latencia.

El G80 que introdujo la arquitectura CUDA tenía 86.4 GB/s de ancho de banda de memoria, además de un ancho de banda de comunicación de 8 GB/s con la CPU. Una aplicación CUDA puede transferir datos desde la memoria del sistema a 4 GB/s, y al mismo tiempo cargar los datos de vuelta a la memoria del sistema a 4 GB/s. En conjunto, hay un total de 8 GB/s. El ancho de banda de comunicación es mucho menor que el ancho de banda de memoria y puede parecer una limitación; sin embargo, el ancho de banda PCI Express es comparable con el ancho de banda del bus frontal de la CPU a la memoria del sistema, así que no es realmente no es una limitación como podría parecer.

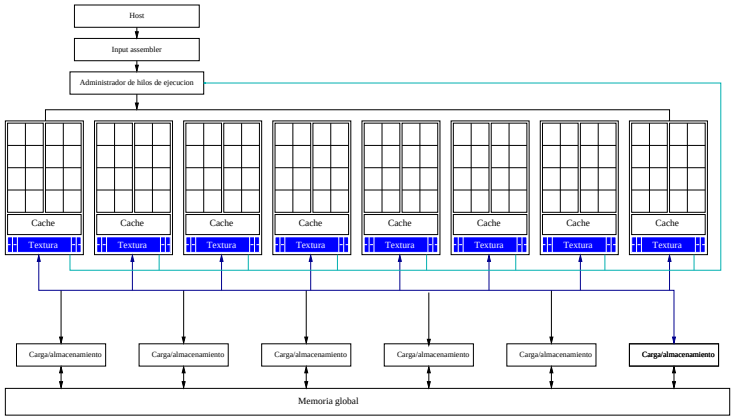

Figura 2.9: Estructura de memoria

Se espera que el ancho de banda de comunicación crezca como el ancho de banda del bus de CPU de la memoria del sistema crece en el futuro.

El chip G80 tiene 128 SP (16 SMS, cada uno con 8SP). Cada SP tiene una unidad multiplicador-sumador (MAD) y una unidad de multiplicación adicional. Con 128 SP, que es un total de más de 500 gigaflops. Adicionales a las unidades de funciones especiales que realizan funciones de punto flotante, como la raíz cuadrada (SQRT), así como las funciones trascendentales. Con 240 SPs, el GT200 excede un teraflops. Debido a que cada SP es masivamente divisible en hilos, puede ejecutar miles de hilos por aplicación. Una buena aplicación típicamente ejecuta entre 5000-12000 hilos simultáneamente en este chip. El chip G80 soporta hasta 768 hilos por SM, que suman alrededor de 12000 hilos de este chip. El más reciente GT200 soporta 1024 hilos por SM y hasta aproximadamente 30000 hilos por el chip. Por lo tanto, el nivel de paralelismo soportado por el hardware de la GPU está aumentando rápidamente.

2.5. CUDA

2.5.1. Paralelismo y Estructura de un programa en CUDA

Algunos investigadores se dieron cuenta de las ventajas que los procesadores gráficos podían ofrecer en la simulación de problemas complejos que requerían de un computo intensivo. Para tener acceso a los recursos computacionales de una GPU un programador tenia que encontrar la manera de convertir su problema en nativas operaciones gráficas para que el computo pudiera ser ejecutado a través de llamadas con OpenGL o la API DiretX. Para poder acceder al computo general a través de una GPU NVIDIA desarrolló el ambiente de programación paralelo CUDA (Compute Unified Device Architecture).

Muchas aplicaciones de software que procesan una gran cantidad de datos y que por lo tanto incurren en largos tiempos de ejecución en las computadoras de hoy en día están diseñadas para modelar el mundo real. Fuerzas de la naturaleza física de cuerpos rígidos y modelos de la dinámica de fluidos son movimientos que pueden ser evaluados de forma independiente dentro dentro de pequeños intervalos de tiempo. Tal independencia en la evaluación es la base de paralelismo de datos en estas aplicaciones.

El paralelismo de los datos se refiere a la propiedad del programa mediante el cual muchas operaciones aritméticas pueden realizarse con seguridad en las estructuras de datos de forma simultánea. Se puede ilustrar el concepto de paralelismo de datos con el ejemplo de una multiplicación matrices donde cada elemento de la matriz P se genera mediante la realización de un producto escalar entre una fila de entrada de la matriz M y una columna de entrada de matriz de N. En la Figura 2.10, el elemento resaltado de la matriz P se genera tomando el producto escalar de la fila resaltada de la matriz M y la columna resaltada de la matriz N. El cálculo de los diferentes elementos de la matriz P se pueden realizarse simultáneamente. Es decir, ninguno de estos productos punto afectará ningún otro resultado en la matriz P. Para matrices muy grandes como una de 1000 x 1000 se tendrían que realizar 1000000 de productos punto donde cada implica 1000 multiplicaciones y 1000 operaciones aritméticas acumulativas.

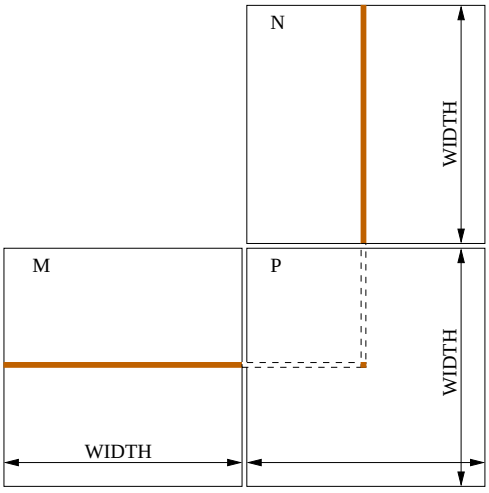

Figura 2.10: Estructura de memoria

Un programa en CUDA consta de una o mas fases que son ejecutadas en el host (CPU) o en el device como una GPU. las fases que muestran poco o ningún paralelismo serán ejecutadas en el host mientras que aquellas que muestren una gran cantidad de datos paralelos serán ejecutadas en el device. El código generado en CUDA debe ser un perfecto acoplamiento entre cogido que se ejecuta en el host y codigo que se ejecuta en el device.

el compilador C de NVIDIA (nvcc) separa los dos durante el proceso de compilación. El código del device usa una extensión de ANSI C con palabras clave para etiquetar funciones de datos paralelas llamadas "Kernelsz su estructura de datos asociada. Las funciones kernel generan una gran cantidad de hilos para explotar el paralelismo.

2.5.2. Tranferencia de datos

En CUDA, el host y los device tienen espacios de memoria diferentes. Esto hace evidente que los dispositivos son típicamente tarjetas de hardware que vienen con su propia memoria dinámica de acceso aleatorio (DRAM). Por ejemplo, el procesador NVIDIA T10 viene con hasta 4 GB (billón de bytes, o giga bytes) de memoria DRAM. para poder ejecutar un kernel en un dispositivo el programador necesita asignar memoria en el dispositivo y transferir los datos pertinentes de la memoria del host a la memoria del device. Después que la ejecución se ha llevado a cabo en el device es necesario regresar los datos resultantes de la memoria del device a la memoria del host y liberar la memoria del dispositivo que ya no es necesaria.

La figura 2.11 muestra una visión general del modelo de memoria, de la asignación, el movimiento y el uso de los distintos tipos de memoria de un dispositivo. En la parte inferior de la figura se ve la memoria global y la memoria constante. Estos son los tipos de datos en memoria que el código de host puede transferir hacia y desde el dispositivo, como se pude apreciar debido a la flecha bidireccionales entre el host y el devidce. La memoria constante permite acceso de sólo lectura para el código del dispositivo. Tenga en cuenta que la memoria principal no se muestra explícitamente en la Figura 2.11, pero se supone que deben figurar en el host.

El modelo de memoria cuenta con funciones de la API que ayudan al programador con la gestión de los datos entre las memorias. Entre las funciones que se tienen para la gestión de la memoria se tiene la función *cudaMalloc*() que se puede llamar desde el código de host para asignar a un objeto una parte de la memoria global. ésta función es muy parecida a la función de las bibliotecas de C *malloc*(). Esto es debido a que CUDA es C con extensiones mínimas. CUDA utiliza *malloc*() para gestionar la memoria del host y agrega *cudaMalloc*() como una extensión de la biblioteca de C. Con esto se minimiza el tiempo que le lleva al programador aprender CUDA.

El primer parámetro de la función *cudaMalloc*() es la dirección del apuntdor. A la dirección del apuntador se le debe hacer un cast a void (void **) debido a que la función no espera un valor de puntero específico; la función de asignación de memoria es una función general que no se limita a ningún tipo particular de objetos. El segundo parámetro de la función cudaMalloc() da el tamaño del objeto que se asignará, en términos de bytes.

En la Lista 2.1 muestra un código sencillo que ilustra el uso de *cudaMalloc*(). Se pasa la

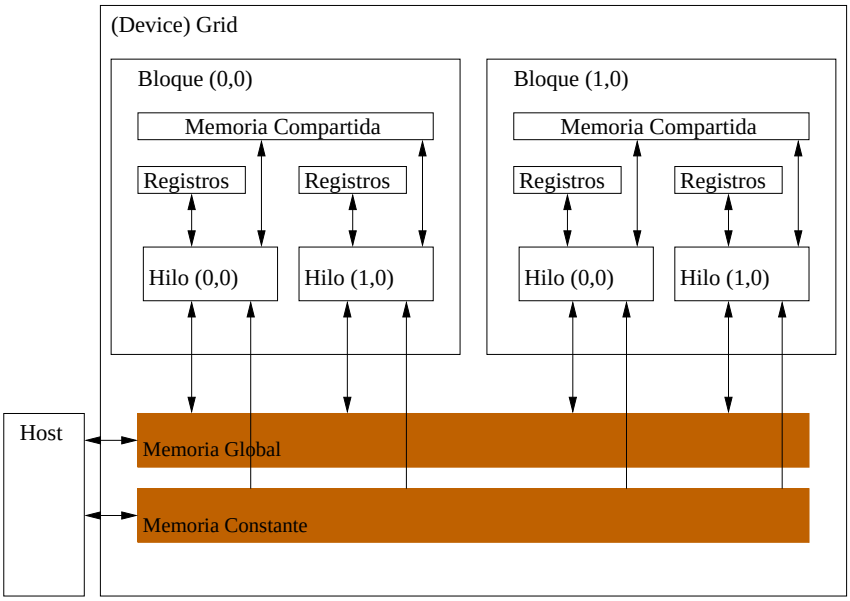

Figura 2.11: Estructura de memoria

dirección de Md como primer parámetro después de hacer un cast a void; es decir, Md es el puntero que señala a la región de memoria global del dispositivo asignada para la matriz M. El tamaño de la matriz asignada será N*N*4 puesto que la matriz es de tamaño NxN. Después del computo se llama a *cudaFree*() con un puntero Md como entrada para liberar el espacio de almacenamiento para la matriz M de la memoria global del dispositivo.

```
1
2           float *Md
3           int size = Width * Width * sizeof(float);
4           cudaMalloc((void**)&Md, size);
5           ...
6           cudaFree(Md);
7           \caption{Ejemplo de uso cudaMalloc}
8           \label{fig: ejemCmalloc}
```

Listing 2.1: Ejemplo de uso de cudaMalloc

Una vez que el programa ha asignado la memoria global del dispositivo, es posible tranferir los datos del host al device. Esto se consigue llamando a una de las funciones de la API de CUDA, *cudaMemcpy*(), para la transferencia del host al device y a la inversa. La función *cudaMemcpy*() toma cuatro parámetros. El primer parámetro es un puntero a la ubicación de destino para la operación de copia. El segundo parámetros son los datos de origen del objeto que desea copiar. El tercer parámetro especifica el número de bytes a copiar. El cuarto parámetro indica el el tipo de transferencia que se hará: desde la memoria principal a la memoria principal, desde la memoria principal a

la memoria del dispositivo, desde la memoria del dispositivo a la memoria principal o desde la memoria del dispositivo a la memoria del dispositivo. cudaMemcpy() no se puede utilizar para hacer copias de datos entre diferentes dispositivos en los sistemas multi-GPU.

Supongamos que M, P, Md, Pd, y el tamaño ya se han establecido; las dos llamadas a funciones se muestran a continuación. $cudaMemcpyHostToDevice$ y $cudaMemcpyDeviceToHost$, son constantes predefinidas del entorno de programación CUDA. La misma función se puede utilizar para transferir datos en ambas direcciones pero se debe ser muy cuidadoso con los punteros de origen y de destino y el uso de la constante apropiada para el tipo de transferencia.

$$cudaMemcpy(Md, M, \text{ tamaño }, cudaMemcpyHostToDevice);$$

$$cudaMemcpy(P, Pd, \text{ tamaño }, cudaMemcpyDeviceToHost);$$

La asignación de la memoria y los diferentes tipos de transferencia de datos son fundamentales para obtener una exitosa ejecución.

2.5.3. Funciones kernel y manejo de hilos

En CUDA, una función kernel especifica el código a ser ejecutado por todos los hilos durante una fase paralela. Debido a que todos estos hilos ejecutan el mismo código, la programación CUDA es una instancia del programa único conocido, single-program multiple-data (SPMD), un estilo de programación popular para sistemas masivamente paralelos. Enalgunas de las extensiones que se hicieron a C fueron añadir palabras clave como __global__ que indica que la función es un kernel y que se puede llamar desde una función de acogida para generar una grid de hilos en un dispositivo.

CUDA extiende la declaración de funciones C con tres palabras. Los significados de estas palabras clave se resumen en la Figura 2.13. La palabra clave __global__ indica que la función que está declarada es una función del kernel CUDA. La función se ejecutará en el dispositivo y sólo se puede llamar desde el host para generar una grid de hilos en un dispositivo. Además __global__, hay otras dos palabras clave que se pueden utilizar en frente de una declaración de la función. La palabra clave __device__ indica que la función está declarada es una función del dispositivo CUDA. Una función del dispositivo se ejecuta en un dispositivo de CUDA y sólo se puede llamar desde un kernel de la función u otra función del dispositivo. Funciones del device no pueden tener llamadas a funciones recursivas ni llamadas a funcines indirectas a través de punteros en ellos. La palabra clave __host__ indica que la función está siendo declarada es una función del host. Una función de host es simplemente una función de C tradicional que se ejecuta en el host y sólo puede ser llamado de otra función de host. Por defecto, todas las funciones en un

	Ejecutada en:	Llamada desde:
__device__ float DeviceFunc()	Device	Device
__global__ void kernelFunc()	Device	host
__host__ float HostFunc()	host	host

Figura 2.12: Estructura de memoria

programa CUDA son funciones host si no tienen alguna de las palabras clave de CUDA en su declaración.

También es posible utilizar tanto __host__ como __device__ en una declaración de la función. Esta combinación activa el sistema de compilación para generar dos versiones de la misma función. Una se ejecuta en el host y sólo se puede llamar desde una función host. El otro se ejecuta en el dispositivo y sólo se puede llamar desde un dispositivo o una función del kernel. Esto apoya un uso común cuando el mismo código fuente se puede simplemente volver a compilar para generar una versión del dispositivo.

Otras extensiones de C que se deben mencionar son las palabras clave threadIdx.x y threadIdx.y, que se refieren a los índices de hilo. Todos los hilos ejecutan el mismo código del kernel. Es necesario un mecanismo que les permita distinguirse y dirigirse a sí mismos hacia las partes particulares de la estructura de datos que han sido designados para trabajar. Estas palabras clave identifican las variables predefinidas que permiten que un hilo acceder a los registros de hardware en tiempo de ejecución. Diferentes hilos verán diferentes valores en sus variables threadIdx.x y threadIdx.y.

Cuando se invoca un kernel se ejecuta una grod de hilos paralelos. Cada grid típicamente se compone de miles de millones de hilos de GPU ligeros, creando suficientes subprocesos para utilizar plenamente el hardware que a menudo requiere una gran cantidad de paralelismo de datos; Por ejemplo, cada elemento de una matriz grande podría ser calculada en un hilo separado.

Los hilos en una grid se organizan en una jerarquía de dos niveles, como se ilustra en la figura 3.13. en un pequeño número de t se muestran en la Figura 2.13. En realidad, una grid consistirá típicamente de muchos más hilos. En el nivel superior, cada grid se compone de uno o más bloques. Todos los bloques en una grid tienen el mismo número de hilos. En la figura 2.13, la grid 1 está organizada como una matriz 2x2 de 4 bloques. Cada bloque tiene unicas coordenadas bidimensionales dado por las palabras clave específicas CUDA blockIdx.x y blockIdx.y. Todos los bloques deben tener el mismo número de hilos organizados de la misma manera.

En cambio cada bloque a su vez está organizado como una matriz tridimensional de hilos con un tamaño total de hasta 512 hilos. Las coordenadas de los hilos en un bloque

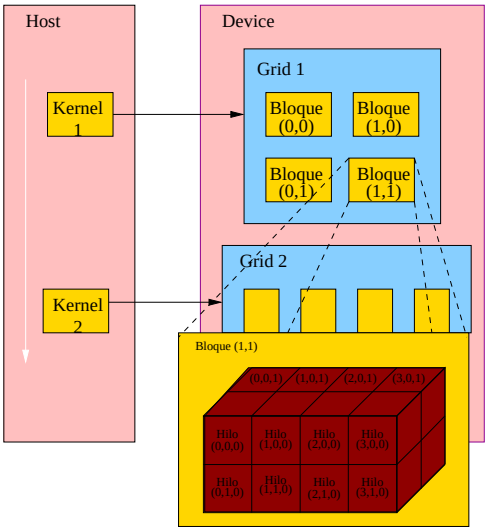

Figura 2.13: Estructura de memoria

se definen únicamente por tres índices: threadIdx.x, threadIdx.y y threadIdx.z. No todas las aplicaciones utilizan las tres dimensiones de un bloque de hilos. En la figura 3.13, cada bloque está organizado en una matriz 4x2x2 tridimensional de hilos. Esto le da a la grid 1 un total de 4x16 = 64 hilos.

2.6. Resumen

En este capítulo se presentó una vista a lo que es el computo paralelo, la importancia que tiene hoy en día en el ámbito científico y algunos de los enfoques en los que se aplica la división de un problema en pequeñas partes que mas tarde son resueltas de manera simultanea. Debido a las características que poseen las GPUs no todo tipo de problema es adecuado para resolverse sobre este tipo de arquitectura. El método de Monte Carlo es intrínsecamente paralelo por lo que es un candidato idóneo para ser programado sobre GPUs. CUDA es un conjunto de herramientas de desarrollo creadas por NVIDIA que ademas ha incluido un gran conjunto de bibliotecas útiles al programador. Tal es el caso de cuRAND que se uso para la generación de muestras aleatorias con diferentes distribuciones.

Capítulo 3

Integral de Wiener

3.1. Introducción

En 1827 el botánico francés Robert Brown observo pequeñas partículas de polen en un liquido danzando por doquier sin una razón aparente. Este movimiento fue llamado movimiento browniano. Más tarde la definición formal de este movimiento la hizo el matemático Robert Wiener y a éste se le denominó proceso de Wiener. Las trayectorias resultantes de un proceso de Wiener son continuas pero no derivables por lo tanto se requiere de la construcción de una integral estocástica. La integral de Wiener es una integral estocástica sobre un espacio de funciones medibles o sobre un funcional.

3.2. Proceso de Wiener

Para introducir a los procesos de Wiener se describirá un "experimento" sencillo. Pensemos en un individuo que posee una riqueza inicial igual a $W(0)$ unidades monetarias; un periodo más tarde el individuo extrae una muestra de una variable aleatoria distribuida normalmente con media 0 y varianza 1 y el resultado de esta extracción será añadido (o sustraído) de la riqueza inicial. La riqueza del individuo tras el experimento será denotada como $W(1)$. Un periodo más tarde el experimento vuelve a repetirse extrayendo una nueva muestra a partir de una variable aleatoria normal, independientemente de la anterior, pero con la misma media y varianza. El resultado de la muestra vuelve a ser añadido (o sustraído) y ahora su riqueza es $W(2)$. Este experimento e ira repitiendo sucesivamente, de tal forma que al cabo de t periodos la riqueza del inversor será $W(t)$. La cantidad acumulada al cabo de t periodos puede describirse mediante expresión (3.1).

$$W(t+1) = W(t) + \xi_{t+1} \text{ donde } W(0) = W_0; \text{ y } \xi \sim N(0,1) \tag{3.1}$$

Una función $W(t)$ como la que se acaba de describir recibe la denominación de función aleatoria o bien, al ser el argumento de la función el tiempo, un proceso estocástico [13]. Observe que $W(t)$ verifica las siguientes propiedades:

$$E_t[W(t+1)] - W(t) = E_t[\xi t + 1] = 0 \tag{3.2}$$

$$Var_t[W(t+1)] - W(t) = Var_t[\xi t + 1] = 1 \tag{3.3}$$

$$E_t[(W(t+1)] - W(t))(W(t+j)] - W(t+j-1))] = 0 \text{ para todo } j > 1 \tag{3.4}$$

La propiedad (3.4) implica que las variaciones de $W(t)$ correspondientes a dos intervalos del tiempo distintos son independientes y se deriva del supuesto de que las variables ξ_t son variables aleatorias independientes.Una posible realización de este proceso ha sido representa en la figura 7.1.

Una propiedad importante del proceso $W(t)$ es la esperanza condicionada de $W(t+1)$, $E[W(t+1)|W(t), W(t-1), W(t-2), \ldots]$, depende exclusivamente del valor de $w(t)$, siendo totalmente independiente de la senda que haya seguido la variable W hasta t, es decir, la expectativa en t obre el valor futuro de W no depende de cómo se haya llegado hasta el valor $W(t)$, de la historia pasada de W, sino sólo de su valor en t. En cierto modo se podría afirmar que el proceso estocástico $W(t)$ no tiene memoria, los valores previos de $W(t)$ no influyen en los posibles valores que pueda adoptar dicha variable en el futuro. Esta propiedad es muy importante en la medida en que se vaya a utilizar este tipo de proceso estocástico para modelar el comportamiento a lo largo del tiempo de una variable financiera (precio de una acción, un tipo de interés, un tipo de cambio, etc.) ya que supone admitir de forma implícita la hipótesis de eficiencia de los mercados financieros.

En el ejemplo anterior no se especificó la amplitud del intervalo de tiempo entre "experimentos". Si los periodos de tiempo considerados fuesen anuales, sería deseable escribir un proceso de similares características, pero en el que las variaciones de $W(t)$ puedan darse con mayor frecuencia, por ejemplo, cada $1/n$ años, siendo n un entero mayor que 1. Este nuevo proceso se describe en (3.2)

$$W(t + \Delta t) = W(t) + \xi_{t+\Delta t} \tag{3.5}$$

$$\text{Siendo } \delta t = 1/n;\ W(0) = W_0;\ \xi_{t+\delta t} \sim N(0, \delta t)$$

Este nuevo proceso estocástico verifica, entre otras, las siguientes propiedades:

$$E_t[W(t + \Delta t)] - W(t) = 0 \tag{3.6}$$

$$Var_t[W(t + \Delta t)] - W(t) = \Delta t \tag{3.7}$$

$$E_t[(W(t+\Delta t)] - W(t))(W(t+j\Delta t)] - W(t+(j-1)\Delta t))] = 0 \text{ para todo } j > 1 \quad (3.8)$$

Para un periodo de amplitud $T = \delta t$, de una variable $W(t)$ que sigue un proceso como el que acabamos de describir. La variación que experimentara esta variable al cabo de N periodos de amplitud deltat puede expresarse como:

$$W(t+T) - W(t) = \sum_{j=1}^{n} \xi_{t+j\Delta t} \quad (3.9)$$

Donde

$$\xi t + j\Delta t \sim N(0, \Delta t)$$

De la expresión (3.9) se pueden deducir las siguientes propiedades de $W(t+T) - W(t)$:

$$E_t[W(t+T) - W(t)] = E_t[W(t) + \sum_{j=1}^{n} \xi_{t+j\Delta t} - W(t)] = 0; \quad (3.10)$$

$$Var_t[W(t+T) - W(t)] = Var_t[W(t) + \sum_{j=1}^{n} \xi_{t+j\Delta t} - W(t)] = N\Delta t = T \quad (3.11)$$

De estos resultados se deduce que la variación esperada en el valor de la variable W(t) durante el periodo $[t, t+T]$ es nula, mientras que la incertidumbre respecto al posible valor que pueda adoptar dicha variable en $t+T$ se incrementa en función de la amplitud del intervalo de tiempo considerado T, es decir cuanto más no adentramos en fechas futuras. Obsérvese que, en la medida que ξ_t se distribuye normalmente $W(t+T) - W(t)$ también se distribuye como una normal con media cero y varianza T, es decir $W(t+T) - W(t) \sim N(0, T)$

Podemos entonces interpretar un proceso de Wiener como el limite cuando $\Delta t \xrightarrow{0}$ de un proceso como el que acabamos de describir es decir:

$$W(t+dt) = W(t) + \xi_{t+dt} \quad (3.12)$$

Siendo $W(0) = W_0$ y $\xi_{t+dt} \sim N(0, dt)$

Si se define dW como $W(t+dt) - W(t)$ se verifican las siguientes propiedades:

$$E_t[dW(t)] = 0; \quad (3.13)$$

$$E_t[dW(t) \cdot dt] = E_t[dW(t)]dt = 0; \quad (3.14)$$

$$Var_t[dW(t)] = E_t[dW(t)^2] = dt \quad (3.15)$$

Así mismo, se verifica que $W(t)$ es un proceso continuo respecto a t, pero $W(t)$ no es diferenciable para ningún valor de t.
Llegado a este punto, cabria preguntarse en qué medida un proceso de Wiener es apropiado para representar el comportamiento de variables financieras. Por ejemplo, supongamos que estamos tratando de describir el comportamiento del precio de una acción en t, que denotaremos por $S(t)$ mediante un proceso de Wiener. Según las propiedades que hemos analizado, la variación esperada $S(t)$ sería cero y por lo tanto el valor esperado del precio de la acción en cualquier fecha futura $t+T$ seria el precio de la acción en t, es decir $S(t)$ que no parece muy adecuado.
Sin embargo es posible definir procesos estocásticos más generales a partir de los procesos de Wiener. Por ejemplo, consideremos un proceso estocástico discreto $X(t)$ definido como:

$$X(t+1) = X(t)a + b\xi_{t+1}$$

Siendo $X(0) = X_0$, $\xi_t \sim N(0,1)$ y $a, b \in R$

Observe que ahora $X(t)$ verifica las siguientes propiedades:

$$E_t[X(t+1) - X(t)] = E_t[a + b\xi_{t+1}] = a + bE_t[\xi_{t+1}] = a \tag{3.16}$$

$$Var_t[X(t+1) - X(t)] = Var_t[a + b\xi_{t+1}] = b^2 Var_t[\xi_{t+1}] = b^2; \tag{3.17}$$

Es decir la variación esperada de la variable $X(t)$ por unidad de tiempo es igual al parámetro a, y la varianza condicionada de la variación de la variable $X(t)$ por unidad de tiempo es igual a b^2. El nuevo proceso ha sido dotado de una "tendencia" (que depende del parámetro a) de igual forma es posible controlar la variabilidad o volatilidad del proceso mediante el parámetro, b. Un proceso de similares características, pero observado cada $1/n$ años ($n > 1$), puede describirse mediante la expresión (3.18).

$$X(t + \Delta t) = X(t) + a\Delta t + bE_t[\xi_{t+\Delta t} \tag{3.18}$$

$X(0) = X_0$, siendo ahora $\xi_t + \Delta t \sim N(0, \Delta t)$ y $\Delta t = 1/n$

En un proceso estocástico como el descritó en la expresión (3.18), la variación esperada de la variable $X(t)$ para un periodo de amplitud $T = \Delta t$, viene dada por

$$E_t[X(t+T) - X(t)] = E_t[\sum_{j=1}^{n} X(t + j\Delta t)] = E_t[\sum_{j=1}^{n} a\Delta t + b\xi_{t+j\Delta t}] = N \cdot a \cdot \Delta t = aT \tag{3.19}$$

Y la varianza condicionada de a variación de la variable $X(t)$ para dicho periodo es

$$Var_t[X(t+T) - X(t)] = Var_t[\sum_{j=1}^{n} X(t + j\Delta t)] = Var_t[\sum_{j=1}^{n} a\Delta t + b\xi_{t+j\Delta t}] = \sum_{j=1}^{n} b^2 Var_t[\xi_{t+j\Delta t}] = b^2 \cdot N \tag{3.20}$$

Hay que señalar además que $X(t)$ se obtiene como la suma de variables normales, por lo que $X(t+T)$ también se distribuye según una variable aleatoria normal, es decir $X(t+T) \sim N[X(t) + aT, b^2 T]$

3.2.1. Movimiento browniano aritmético

De los resultados que hemos obtenido se deduce que el parámetro a puede interpretarse como la variación esperada de de la variable X por unidad de tiempo y b^2 como la varianza condicionada de la variación de la variable X por unidad de tiempo. En el caso de que $\Delta t \to 0$, podemos escribir

$$dX(t) = adt + bdW(t); X(0) = x_0 \tag{3.21}$$

Que es la notación que usaremos para generalizar los procesos de Wiener dotándolos de tendencia (a) y permitir (mediante el parámetro b) diferenciar distintos comportamientos en cuanto a su volatilidad, es decir en cuanto a la incertidumbre sobre las variaciones de las variables objeto de análisis. Los procesos estocásticos como el que se acaba de describir reciben el nombre de Movimiento Browniano Aritmético (MBA)[13] y ha sido utilizado en algunos modelos para describir el comportamiento de los tipos de interés. Existe un caso particular de MBA para describir el comportamiento a lo largo del tiempo del interés capitalizable instantáneamente r(t). El modelo es el siguiente:

$$dr(t) = \sigma dW(t); r(0) = r_0 \tag{3.22}$$

según este modelo, el tipo de interés instantáneo sigue un MBA con tendencia nula ($a = 0$) y varianza condicionada por unidad de tiempo igual σ^2. El tipo de interés instantáneo en una fecha futura $t > 0$ se distribuye como una normal con media r_0 y varianza $\sigma^2 t$. se debe notar que entre mayor es el valor de t, es decir cuando más nos adentramos en el futuro, la incertidumbre con respecto a los valores que puede adoptar $r(t)$ son cada vez mayores al ser mayor la varianza de $r(t)$.

3.2.2. Movimiento browniano geométrico

Este tipo de proceso estocástico es comúnmente para describir variables financieras, denotaremos como MBG y se puede describir como:

$$dX(t) = \mu X(t)dt + \sigma X(t)dW(t) \tag{3.23}$$

o alternativamente

$$dX(t)/X(t) = \mu dt + \sigma dW(t) \tag{3.24}$$

En este caso la función de tendencia es $\alpha(X(t), t) = \mu \cdot X(t)$ y la función de volatilidad, $\sigma(X(t), t) = \sigma X(t)$. Este proceso generalmente es usado para describir el comportamiento del precio de las acciones que no pagan dividendos o en general variables que crezcan exponencialmente.

Se debe tener en cuenta que si se utiliza un MBA para describir el comportamiento del precio de una acción, entonces su variación esperada para un periodo de tiempo dado

seria la misma con independencia del nivel del precio de la acción. Por lo tanto, entre mayor sea el precio de la acción menor será su rendimiento esperado por el contrario si se supone que el precio de la acción sigue un MBG como el que se acaba de describir, la variación esperada del precio de la acción para un periodo de tiempo dado Δt vendrá dada por $E_t[X(t+\Delta t) - X(t)] = \mu X(t)\Delta t$ siendo su rendimiento esperado, aproximadamente $E_t[\Delta X(t)/X(t)] \approx \mu \cdot \delta t$ por lo que μ se puede interpretar como el rendimiento esperado de la acción [13].

3.3. Medida de Wiener

Ahora veremos el proceso de Wiener como la construcción de una medida de probalilidad en un espacio de caminos continuos. En apartados anteriores hemos definido el proceso de Wiener como el límite de una suma de variables aleatorias de un modo parecido al que utilizamos para integrar una función continua: Sumamos una gran cantidad de variables de varianza pequeña. Conocemos la distribución límite de estas variables, pero no sabemos si les corresponde algún espacio de probabilidad, con una $\sigma-$álgebra de conjuntos medibles bien definida. Veamos en primer lugar que el mismo proceso de construcción, y la escala utilizada para a nadir las pruebas independientes, hacen que una muestra del proceso de Wiener sea una función continua (en un sentido probabilístico). Calculemos la siguiente probabilidad:

$$P(|W_{t+\Delta t} - W_t| > \epsilon) = P(|\sqrt{\Delta t}N(0,1)| > \epsilon) = \frac{1}{\sqrt{2\pi\Delta t}} \int_{|x|>\epsilon} \mathrm{e}^{-\frac{x^2}{2\Delta t}} dx$$

Por otra parte, para $\epsilon/\sqrt{\Delta t} > 1$

$$\int_{|x|>\epsilon} \mathrm{e}^{-\frac{x^2}{2\Delta t}} dx = 2\int_{\epsilon}^{\infty} \mathrm{e}^{-\frac{x^2}{2\Delta t}} dx \leq 2\int_{\epsilon}^{\infty} \mathrm{e}^{-\frac{x}{2\sqrt{\Delta t}}} dx = \frac{\mathrm{e}^{-\frac{x^2}{2\Delta t}}}{\sqrt{\Delta t}}$$

De modo que:

$$P(|W_{t+\Delta t} - W_t| > \epsilon) = o((\Delta t)^n) \forall n > 0,\ \Delta t \to 0 \tag{3.25}$$

donde la igualdad (3.25) se obtiene teniendo en cuenta que $\lim_{u\to\infty} u^n(\mathrm{e})^{-}\mathrm{u} = 0$ para todo $n > 0$ Si bien esta relación nos dice que la probabilidad de dar un salto es de un orden arbitrariamente pequeño respecto de Δt, esto no implica regularidad alguna. Es decir, puede haber una muestra límite (que tendríamos que definir con precisión) que sea discontinua, ya que en principio cualquier gráfica se podría obtener al sumar la infinidad de variables independientes. La noción de la regularidad de estas muestras no tiene sentido hasta que no definamos un ambiente natural en el que deben vivir los procesos límite y le asignemos una medida de probabilidad. En ese caso podríamos intentar demostrar una afirmación del tipo: "el conjunto de funciones discontinuas tiene medida cero", o "el conjunto de funciones diferenciables tiene medida cero".

3.3.1. Construcción de la medida de Wiener

El proceso de construcción de la medida se basa en el hecho de que la familia de variables W_t está incluida de forma natural dentro del conjunto de funciones reales definidas en la semirrecta real positiva. Es decir, una muestra del proceso límite es como asignar un valor real a cada $t > 0$ y en ese sentido nuestro espacio muestral debería ser el conjunto enorme de las funciones reales definidas en $[0, +\infty)$. Sin embargo, teniendo en cuenta (3.25) es posible restringirse al conjunto Ω de funciones continuas $\omega : [0, +\infty] \to \mathbb{R}$ nos devuelve el valor de la función continua ω en t:

$$W_t(\omega) = \omega(t)$$

La distribución de probabilidad de W_t nos permite calcular la probabilidad de encontrar a la variable en un intervalo real determinado. Esta distribución marginal nos permitiría definor la medida para unos subconjuntos de Ω muy particulares:

$$P(a \leq W_t \leq b) = \text{ medida de los } \omega \text{ que pasan por el intervalo } [a, b] \text{ en } t$$

El conjunto

$$C_{[a,b];t} := \omega \in \Omega : a \leq \omega(t) \leq b$$

Se llama conjunto cilíndrico o ventana y su medida de probabilidad la definimos entonces como:

$$P(C_{[a,b];t}) := \int_a^b \phi(x,t)dx, \ \phi(x,t) := \frac{\mathrm{e}^{\frac{x^2}{2t}}}{\sqrt{2\pi t}} \tag{3.26}$$

Esto es, usando la distribución de la wariable W_t, que es normal de varianza t y media cero. Para definir la probabilidad de la intersección de dos ventanas $C_{[a_1,b_1];t1} y C_{[a_2,b_2];t2}$ debemos tener en cuenta la distribución conjunta de las variables W_{t1} y W_{t2} $(t_1 < t_2)$ caracterizada por la independencia del incremento $\Delta W_{t1} := W_{t2} - W_{t1}$ respecto de W_{t1}. La probabilidad que queremos medir es la de las funciones continuas que pasan en t_1 por un intervalo $[a1, b1]$ y en $t2$ por un intervalo $[a2, b2]$. Para ello notemos que

$$P(a_1 \leq W_{t_1} \leq b_1, a_2 \leq W_{t_2} \leq b_2) = \int_{a1}^{b1} \int_{a2}^{b2} p(x_1, t_1; x_2, t_2)dx_1dx_2 \tag{3.27}$$

$$= \int_{a1}^{b1} \int_{a2}^{b2} p(x_2, t_2|x_1, t_1)p(x_1, t_1)dx_1dx_2 = \int_{a1}^{b1} p(x_1, t_1)(\int_{a2}^{b2} p(x_2, t_2|x_1, t_1)dx_2)dx_1 \tag{3.28}$$

Donde en igualdad (3.27)se usa la densidad de la distribución conjunta

$$p(x_1, t_1; x_2, t_2)dx_1dx_2 := P(x_1 \leq W_{t_1} \leq x_1 + dx_1, a_2 \leq W_{t_2} \leq x_2 + dx_2)$$

La independencia de los incrementos nos dice que

$$p(x_2,t_2|x_1,t_1)dx_2 = P(x_2 \leq W_{t2} \leq x_2+dx_2|W_{t_1}=x_1) = P(x_2-x_1 \leq \Delta W_{t_2-t_1} \leq x_2-x_1+dx_2) = \phi(x_2 \tag{3.29}$$

donde $\Delta W_{t_2-t_1} := W_{t_2} - W_{t_1}$. Finalmente:

$$P(C_{[a_1,b_1];t1} \cap C_{[a_2,b_2];t2}) := \int_{a1}^{b1} \phi(x_1,t_1)(\int_{a_1}^{b_1} \phi(x_2-x_1,t_2-t_1)dx_2)dx_1$$

Habiendo definido la probabilidad de la intersección de eventos elementales, tenemos entonces definida la unión:

$$P(C_{[a_1,b_1];t1} \cap C_{[a_2,b_2];t2}) = P(C_{[a_1,b_1];t_1}) + P(C_{[a_2,b_2];t2}) - P(P(C_{[a_1,b_1];t1} \cap C_{[a_2,b_2];t2}))$$

Esto nos permite definir una medida en el álgebra generada por los conjuntos cilíndricos f_0 (uniones e intersecciones finitas de estos conjuntos) [14]. A partir de aquí podemos definir la sigma álgebra f como la mínima que contiene a f_0 y la medida se extiende en forma continua a todos ellos. La definición de esta medida, debida a Norbert Wiener, nos permite integrar funcionales definidos en Ω (si probamos previamente que el funcional es medible). Podemos interpretar, dado un funcional $F(\omega)$, a su integral
$\int_\Omega \mathsf{F}(\omega)dP_\omega = \int_\Omega \mathsf{F}(\omega)dW$
como el valor esperado del funcional F cuando ω recorre las funciones continuas en un intervalo dado.

3.4. Construcción de la medida de Wiener en finanzas

Si queremos, por otra parte, analizar la dinámica de la variable X, donde ésta presenta movimientos aleatorios continuos y está determinada tanto por situaciones deterministas como por la incertidumbre, de manera que contiene la información generada por un proceso estocástico y tenemos entonces que representar la dinámica estocástica de a través de la denominada ecuación diferencial estocástica:

$$dX(t) = \mu X(t)dt + \sigma X(t)dW(t) \tag{3.30}$$

Donde $\mu X(t)dt$ es un término de tendencia y $\sigma X(t)dW(t)$ es el diferencial de un proceso de Wiener (estocástico). D Donde $\mu X(t)dt$ mide la variabilidad de la variable a lo largo de su trayectoria en el tiempo y $\sigma X(t)dW(t)$ es una constante determinista. Por consecuencia, representa una familia indexada de variables aleatorias en el intervalo de tiempo [0, T], cuyas trayectorias que se obtengan son una aproximación de una simulación discreta de un proceso continuo, las cuales son curvas continuas que no son diferenciables en ningún punto.

La derivación de la ecuación (3.30) proviene de lo siguiente: Dado un proceso de Wiener W, se dice que cualquier proceso g pertenece a la clase $L^2[a,b]$ si es adaptado a la filatación

$$\{\mathsf{F}_t^W\}_{t\geq 0} \text{ tal que } \int_a^b |g(s)|^2 ds < \infty$$

.

Es decir, el proceso debe ser de cuadrado integrable, donde el conjunto de puntos donde no se cumple es despreciable. Además, el proceso es simple en $[a,b]$ si existe un conjunto de puntos determinísticos $a = t_0 < t_1 < \cdots < t_n = b$ y un conjunto de constantes $c_0, c_1, \cdots, c_{n-1}$ tales que $g(t) = ck$ si $t_k < t < \cdots < t_{k+1}$ para $k = 0, 1, \cdots, n-1$, lo que implica la constancia de g.

Por lo anterior, la integral estocástica en el intervalo $[a,b]$ del proceso simple g será

$$\int_a^b g(s)dW(s) = \sum_{k=0}^{n-1} g(k)[W(t_{k+1}) - W(t_k)]$$

Si existe una sucesión de procesos simples cuando $n \to \infty$ entonces podemos obtener la sucesión $X_n = \int_a^b g_n(s)dW(s)$, para cada n, de forma que la sucesión X_n tiene el límite, X cuando $n \to \infty$ entonces:

$$X_n = \int_a^b g(s)dW(s) = \lim_{n\to+\infty} \int_a^b g(s)dW(s) \tag{3.31}$$

Donde X es un proceso estocástico de ruido blanco y dadas las propiedades de esperanza nula y varianza, entonces X es F_t^W-medible p orque está dentro de las trayectorias del proceso de Winer en el intervalo $[a,b]$

Lo anterior implica que para cualquier proceso $g \in L^2$, el proceso x definido en (3.31) para el intervalo $[0,t]$ es una F_t-martingala. En consecuencia, si consideramos adicionalmente al proceso estocástico X un número real x_0 y μ y σ como dos procesos F_t^W adaptados, tenemos que:

$$X(t) = x_0 + \int_0^t \mu(s)ds + \int_0^t \sigma(s)dW(t) \text{ para toda } t \geq 0 \tag{3.32}$$

Si consideramos la condición inicial $X(0) = x_0$ y diferenciamos (3.32) obtenemos el diferencial estoccastico de X dado por:

$$dX(t) = \mu(t)dt + \sigma(t)dW(t) \tag{3.33}$$

Esta ecuación representa la dinámica estocástica de X, donde el primer término corresponde a la tendencia y el segundo término es el componente de ruido gaussiano (aleatorio).

Ambos términos pueden ser representados por funciones del tipo $f = f(t, X)$ siendo éstas diferenciables una vez en t y doblemente diferenciables en X. Por tanto, podemos transformar (3.32) en lo que se denomina proceso de difusión, el cual es un proceso estocástico dado por la ecuación:

$$dZ = f(t, X(t))(t) = \mu(t, X(t))dt + \sigma(t, x(t))dW(t) \tag{3.34}$$

La cual puede relacionarse con modelos determinísticos con incertidumbre y es plenamente coincidente con (3.30) en términos del tiempo y el problema entonces se reduce a encontrar un proceso estocástico que satisfaga la ecuación diferencial estocástica y la condición inicial. Sin embargo, para resolver la ecuación diferencial estocástica, es necesario que se encuentre primero el diferencial de una función de un proceso estocástico. Para hacerlo se utiliza el denominado Lema de Ito que proporciona la solución simple a una ecuación diferencial estocástica como (3.33). Donde, , entonces la solución es el proceso estocástico denominado además, si μ y σ son constantes y $\sigma > 0$, entonces la solución es un proceso estocástico denominado Movimiento browniano geométrico, la cual cumple con las siguientes propiedades:

- X es un proceso estocástico F_t^W-adaptado
- X tiene trayectorias continuas,
- X es un proceso de Markov
- Existe una constante c tal que $E[|x(t)|] \leq ce^{ct}(1 + |x_0|^2)$

así la solución está dada por:

$$X(t) = x_0(\mathrm{e})^{(\mu - \frac{1}{2}\sigma^2)\mathrm{t} + \sigma\mathrm{W(t)}} \tag{3.35}$$

cuyo valor esperado es:

$$E[X(t)] = x_0(\mathrm{e})^{\mu\mathrm{t}} \tag{3.36}$$

(3.35) es una solución válida cuando el proceso estocástico tiene un sola fuente de aleatoriedad. Si hay más de una fuente, entonces se tendrán soluciones que muestran procesos brownianos n-dimensionales [15].

3.5. Resumen

En este capítulo se presento una descripción de lo que es la integral de Wiener comenzando desde su origen como un proceso estocástico hasta llegar a la construcción de una integral apta para este tipo de proceso. La integral de Wiener es una integral funcional

debido a que se integra sobre el espacio de funciones. Al tratarse de una integral estocástica su resolución no es inmediata por ellos se requiere del uso del método de Monte Carlo para generar las trayectorias según las propiedades de la integral. Resolver esta integral por métodos analíticos resulta muy complicado por lo que debe ser calculada por medio de un método numérico. Las variables aleatorias generadas para la construcción de las trayectorias son obtenidas por medio de un algoritmo que arroja muestras con comportamiento normal debido a que los problemas que se tratan requieren de este tipo de comportamiento.

Capítulo 4

Diseño del sistema

Este capítulo tiene como objetivo presentar el análisis y el diseño de la arquitectura general del sistema. Se presenta la descripción y la implementación de los algoritmos de manera secuencial así como su implementación paralela en CUDA. Se muestra la generación de trayectorias, la forma en la que deben ser introducidos los datos al igual que la forma en la que se muestran las salidas.

4.1. Diseño secuencial

El diseño secuencial nos ayuda a darnos cuanto Cuanto es que el tiempo disminuye o aumenta en el peor de los casos. En el presente trabajo el diseño secuencial se llevo a cabo en el lenguaje de programacion C debido a su gran similitud con CUDA. En el diseño secuencial Se utilizo una imlementación de Mersenne Twister en lenguaje C.

4.2. Números aleatorios

Para la generacion de los números aleatorios se utilizo una implementacion del Mersenne Twister en C[1]. Por medio de este algoritmo se obtuvo un arreglo con numeros entre cero y uno con distribución uniforme. Este arreglo es el que se pasa como argumento a una fioncion llamada boxmuller. En la función boxmuller se implementa el algoritmo Box-Muler para generar pares de numeros con media cero y varianza uno. El proceso para ejecutar ambas funciones y las acciones llevadas a cabo en forma similar a lo que se hace en la implementación paralela. En la figura **??** se muestra el proceso que sigue el diseño puramente secuencial.

En la función encargada de generar los caminos aleatorios se obtiene sólo un promedio det total de caminos generados y es lo que la función regresa al proceso principal. Una vez

[1]http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/VERSIONS/C-LANG/ver991029.html

que el proceso principal tiene este promedio es usado para obtener el precio aproximado del de la opcion del bien subyacente.

4.3. Diseño paralelo

La iplementación paralela[2] hecha en CUDA se asume que muy pocas veces los caminos simulados van más alla de los cientos de millones, entonces para mantener a la GPU eficientemente ocupada lo que se hace es valorar mas de una opción a la vez. SÃ se quiere valorar una opción entonces podemos optar por usar un sólo bloque de hilos o mas de un bloque de hilos en una grid en una sola dimensión. El problema de valorar múltiples opciones a la vez es un buen candidato para pensar en una grid bidimensional donde los bloques en X representa los bloques por opción y Y en número de opciones a las cuales están siendo valoradas. La figura 4.2 representa la forma en la que luciria la grid en caso de estar valorando 8 opciones y que para su calculo cada opción cuente con 16 bloques de 256 hilos cada uno.

4.3.1. Generación de números pseudoaleatorios

Para la generación de números pseudoaleatorios primero se generaron números con una ditribución uniforme con el generador Merssenne Twister. Despues de obtener los números con distribución uniformes se aplica una tranformacion para obtener la distribución deseada, en este caso normal.

4.3.1.1. Mersenne Twister

El algoritmo es apropiado para el modelo de programación CUDA ya que puede usar la aritmética bit a bit y una cantidad arbitraria de escrituras a memoria. Por un lado, el t Mersenne twister, como la mayoría de los generadores pseudoaleatorios, es iterativo, por lo que es difícil de paralelizar en un solo paso de actualización de estado tornado entre varios hilos de ejecución. Por otro lado la GPU tiene que tener miles de hilos en la grid de lanzamiento con el fin de ser utilizado plenamente. La solución corta y simple es tener muchos Mersenne twister simultáneamente procesados en paralelo. Pero incluso "muy diferentes"(por definición) valores iniciales de estado no impiden la emisión de secuencias correlacionadoas por cada generador compartiendo idénticos parámetros. Para resolver este problema y permitir la ejecución eficiente de Mersenne Twister en arquitecturas paralelas, dcmt, una biblioteca sin conexión especial para la creación dinámica de los parámetros de Mersenne Twisters, fue desarrollado por Makoto Matsumoto y Takuji Nishimura.

[2]http://developer.download.nvidia.com/compute/cuda/1.1-Beta

La biblioteca acepta los 16 bits del id del hilo como una de las entradas, y codifica este valor en los parámetros de Mersenne Twister sobre una base por "hilo", de modo que cada hilo puede actualizar el twister de forma independiente, al tiempo que conserva buena aleatoriedad de la salida final.

SpawnTwisters.c utilizando la biblioteca dcmt0.3, se ejecuta primero, para precomputar configuraciones para cada índice de hilo de la grid CUDA, la cual es cargada por la aplicación paralela de Mersenne Twisters en tiempo de ejecución. A pesar de que un Mersenne twister está completamente definido por 11 parámetros, sólo los parámetros del vectores de bits varían en función de cada hilo por el mismo período y semillas dcmt (que es el caso): a - la fila inferior de la matriz A; b, c - templado máscaras de x.T transformación. El resto de los parámetros (enteros) se comparten entre todos los hilos, y puede ser simplemente inline en la fuente. En tiempo de ejecución de cada uno de CUDA hilos de almacena el estado en la memoria local del arreglo, y ya que n y m son los mismos para todos los hilos, cada hilo dentro de un warp accesa al mismo índice de estado, y conjuntos de estado lee / escribe siempre se fundieron. La figura 4.3 ilustra el proceso paralelo que se siguió en la creación de los números paralelos con distribución uniforme.

4.3.1.2. Variables aleatorias normales

Una variable aleatoria normal X está distribuída normalmente con media μ y varianza σ, si la función de densidad de probabilidad está dada por:

$$f(x) = \frac{1}{2\pi\sigma} e^{\frac{(x-\mu)^2}{2\sigma^2}}$$

4.3.1.3. Aplicando el método

Tenemos X y Y que son variables aleatorias unitarias independientes y R y θ las coordenadas polares del vector (X, Y) entonces:

$$R^2 = X^2 + Y^2 \ \tan\theta = \frac{Y}{X}$$

Como X y Y son independientes, entonces su densidad conjunta es el producto de sus densidades individuales que está dada por:

$$f(x,y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}}$$

$$f(x,y) = \frac{1}{2\pi} e^{-\frac{x^2+y^2}{2}} \tag{4.1}$$

Para determinar la densidad conjunta de R^2 y θ la llamaremos $f(d, \theta)$, haciendo el cambio de variables:

$$d = x^2 + y^2,\ \theta = tg^{-1}(\frac{y}{x})$$

El cambio de variable se expresó en las dos funciones en terminos de x y y para luego poder alcular el Jacobiano. Para ello recordemos su definición:

Sean $x = g(u, v)$ y $y = h(u, v)$ entonces el jacobiano de x y y con respecto a u y v es como se muestra a continuación denotado por:

$$\frac{\partial(x,y)}{\partial(u,v)} = \begin{vmatrix} \frac{\partial x}{\partial u} & \frac{\partial y}{\partial u} \\ \frac{\partial x}{\partial v} & \frac{\partial x}{\partial v} \end{vmatrix} = |\frac{\partial x}{\partial u}\frac{\partial y}{\partial u} - \frac{\partial x}{\partial v}\frac{\partial y}{\partial v}|$$

Antes de encontrar el jacobiano se calculan las derivadas parciales de x y y.

$$d = x^2 + y^2$$

$$\frac{\partial d}{\partial x} = 2x\ \frac{\partial d}{\partial y} = 2y$$

$$\theta = tg^{-1}(\frac{y}{x})$$

$$\frac{\partial \theta}{\partial x} = \frac{1}{1 + (\frac{y}{x})^2}\frac{-y}{x^2}$$

$$\frac{\partial \theta}{\partial y} = \frac{1}{1 + (\frac{y}{x})^2}\frac{1}{x}$$

Ya que e han calculado las derivadas parciales se calcula el determinante de ellas, aplicando la difinicion anterior del jacobiano, pero on respecto a x y y. Entonces:

$$J^f_{xy}\frac{\partial(d,\theta)}{\partial(x,y)} = \begin{bmatrix} 2x & \frac{1}{1+(\frac{y}{x})^2}\frac{-y}{x^2} \\ 2y & \frac{1}{1+(\frac{y}{x})^2}\frac{1}{x} \end{bmatrix}$$

$$= 2x\left(\frac{1}{1+(\frac{y}{x})^2}\right)\left(\frac{1}{x}\right) - 2y\left(\frac{1}{1+(\frac{y}{x})^2}\right)\frac{-y}{x^2} = J^f_{xy} = 2$$

El jacobiano de esta transformacion es decir, el determinante de las derivadas parciales de d y $theta$ con respecto a x y y es igual a 2.La ecuación 4.1 implica que la función de densidad conjunta de R^2 y θ está dada por:

$$f(d,\theta) = \frac{1}{2}\frac{1}{2\pi}e^{-\frac{d}{2}},\ 0 < d < \infty,\ 0 < \theta < 2\pi$$

Como las variables R y θ son independientes, sus funciones de densidad vienen dadas por:

$$f(r) = re^{-\frac{r^2}{2}} \Rightarrow F(r) = 1 - e^{-\frac{r^2}{2}} \Rightarrow$$

$$f(\theta) = \frac{1}{2\pi} \Rightarrow F(\theta) = \frac{\theta}{2\pi} \Rightarrow \theta = 2\pi U$$

Ahora es posible generar un par de variables aleatorias normales estándar independentes X y Y utilizando R^2 y θ para generar primero sus coordenadas polares ($X = R\cos\theta$ $Y = R\,\mathrm{sen}\,\theta$) y luego transformarlas de nuevo en coordenada retangulares.

Para generar variables aleatorias normales se hace uso del metodo de Box-Muller en el siguiente orden:

1. Generar números aleatorios U_1 y U_2
2. $R^2 = -2\log U_1$ entonces R^2 e exponencial con media 2, $\theta = 2\pi U_2$ θ es uniforme entre 0 y 2π
3. Ahora sean

$$X = R\cos\theta = \sqrt{-2\log U_1}\cos 2\pi U_2$$

$$Y = R\,\mathrm{sen}\,\theta = \sqrt{-2\log U_1}\,\mathrm{sen}\,2\pi U_2$$

El kernel encargado de transformar las variables uniformes a variables normale recibe un arreglo que antes ya fue generado en el kernel Mersenne Twister. éste arreglo es enviado a el kernel encargado de ejecutar Box-Muller con 32 bloques de 128 hilos cada uno. Se tiene un total de 4096 hilos. Cada hilo está encargado de generar tantas muestras normales como el total de muestras uniformes entre 4096. El orden que se sigue para la creación de números con distribucion normal a partir de los obtenidos con una distribución uniforme es ilustrado en la figura 4.4.

4.3.2. Generación de trayectorias

Para la genaración de los posibles caminos que un precio puede recorrer se siguieron dos prespectivas. En la primera se usa mas un bloque para la generación de trayectorias y en la segunda se utiliza un sólo bloque para la generación. La elección de una u otra solución se hace con base en la cantidad de caminos que se quieren simular. En la figufa se muestra un diagrama de flujo que ilustra como se toma la decición de usar uno o mas bloques y que es lo que cada elección conlleva.

4.3.3. Uso de un bloque por opción

En esta perspectiva lo que se hace es tomar ventaja de la memoria conpartida entre los hilos de un bloque y así obtener un vector del tamaño del bloque con las sumas parciales. Se declara un arreglo A en memoria compartida del tamaño del bloque. Con el id de cada hilo se accesara a cada posición del arreglo. Cada hilo es responsable del cálculo de

varios aminos de simulación y la acumulación de las sumas obtenidas en la pocisión que le corresponde en el arreglo según se id.

En la figura 4.6 se muestra una representacion de lo que se hace.

Una vez obtenido el arreglo con todas las sumas parciales éste se somete a una reducción en la que se hace la suma total y es enviada a la pocición cero del arreglo A. Esta posición cero es la que se regresara al host para realizar el cálculo final de la opción y los intervalos de confianza. En la reducción lo que se hace es dividir en dos partes el arreglo A y sumar así el elemento A_0 con el elemento A_{n-1}, el elemento A_1 con el elemento A_{n-2} y asi sucesivamente hasta que las sumas se colocan en los primeros $n/2$ elementos. Para ilstrar esto en la figura 4.8 se muestra un ejemplo de lo que sucedería con un arreglo de tamaño 8. En nuestra implementación se tiene un arreglo de tamaño 128.

En la implementación se tienen dos arreglos de tamaño 128. En el primer arreglo se acumulan las sumas parciales obtenidas y en el segundo la sumas parciales de los cuadrados obtenidos de cada caálculo, esto para mas tarde calcular la desviación estandar de los caminos de simulación y los intervalos de confianza.

4.3.4. Más de un bloque por opción

En este enfoque lo que se pretende es mantener eficientemente ocupada a la GPU, pues al ser muchos los calculos que se tienen que hacer es conveniente que el trabajo sea dividido entre los hilos de más de un bloque. En este caso al estar involucrado más de un bloque en los calculos para determinar el precio de una misma opció no es posible usar memoria compartida pues ésta se comparte sólo entre los hilos de un mismo bloque. La estrategia que se sigue es usar la memoria del device. En la figura **??** se muestra lo que se hace al usar más de un bloque por opción.

Ahora para reducir la suma es necesario mandar a llamar a un nuevo kernel para hacer una reducción de los elementos en la memoria del device a un arreglo de tamaño 128 localizado en memoria compartida. Se dividen los 4096 elementos por el número de elementos que tiene el nuevo arreglo en memoria compartida. En este caso son 32, cada elelmento en memoria compartida va a tener la suma de 32 elementos del arreglo en la memoria del device. Una vez que ya se tiene la suma de todos los elementos en la memoria comparida es posible mandar a llamar la anterior reducción donde el arreglo se va partiendo en dos hasta tener el total de los elementos en el elemento cero.

4.4. Entrada y salida de datos

En cuanto a la entrada y salida de los datos es necesario contar con un precio inicial S_0, el precio del bien subyacente X, la volatilidad v, la tasa libre de riesgo r, el tiempo de vencimiento de la opción T, el incremento de tiempo Δt y la barrera que puede tocar esa opción. para efectos de pruba todos los datos son generado de forma aleatoria con la

función RandFloat. Estos datos que se generaron de manera aleatoria son almacenados una vez en arreglos de tamaño máximo 256 debido a que éste es el número máximo de valoración de opciones con los cuales se hicieron pruebas.

La salida de las pruebas es un archivo con el valor de la opción que arrojo Monte Carlo y los intervalos de confianza.

4.5. Resumen

Este capítulo presento el diseño de la arquitectura desarrollada. El diseño presenta implementaciones con diferentes cantidades de trayectorias pues esto es necesario para hacer comparaciones respecto a los resultados arrojados. La entrada y salida de los datos es una parte importante. ésta es la parte que mas interesa al usuario final es por eso que se presentas los formatos en los cuales los datos requeridos deben ser introducidos y la manera en que se muestra el resultado.

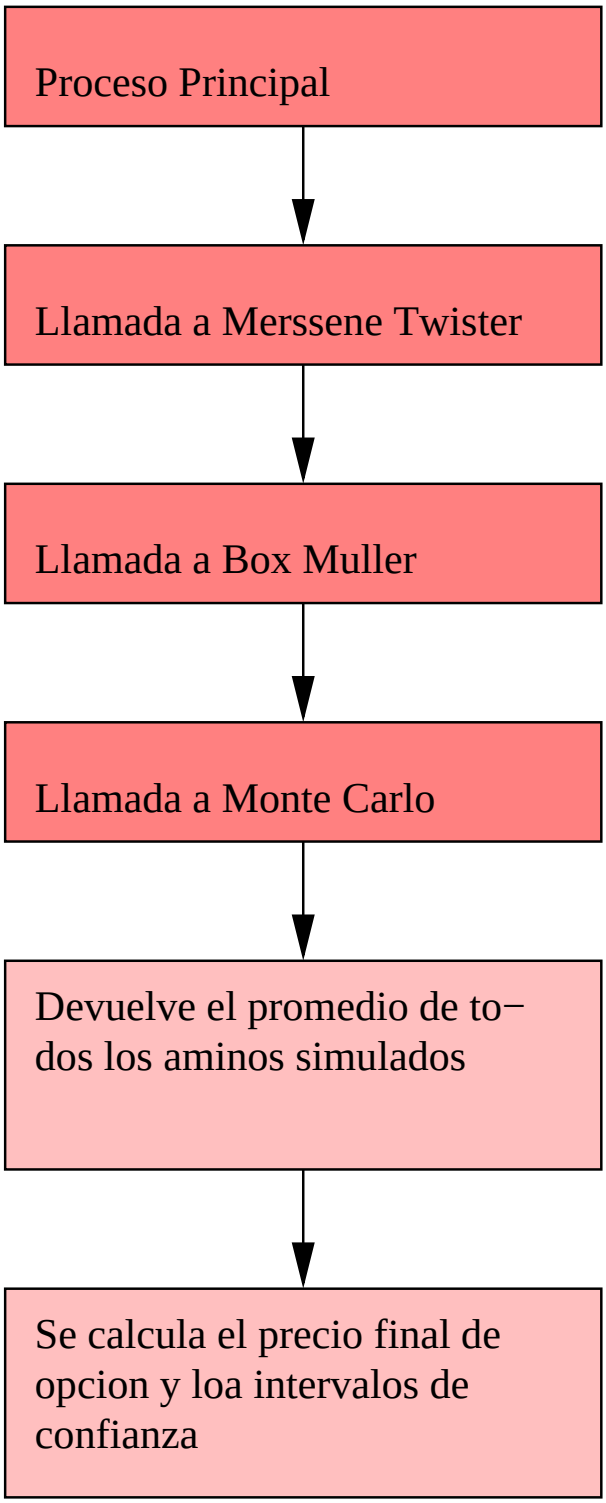

Figura 4.1: Diseño secuencial.

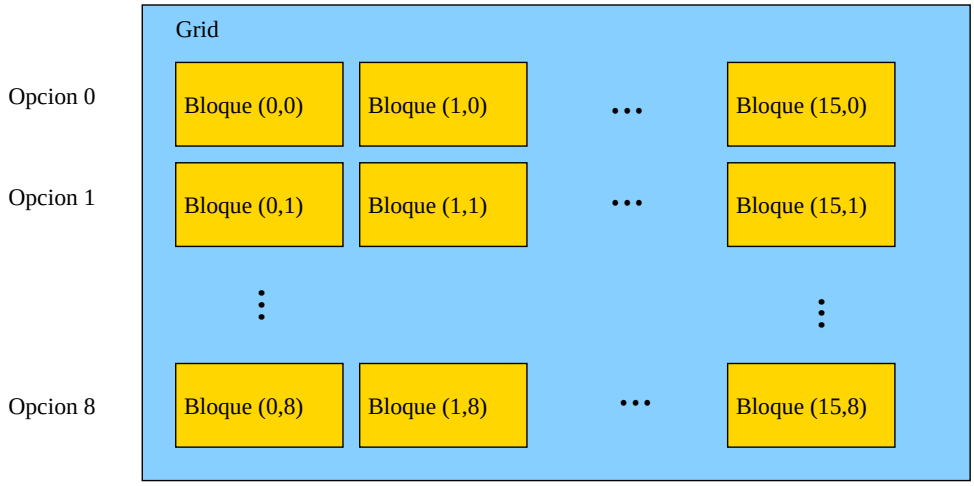

Figura 4.2: Grid para la valoración de 8 opciones con 16 bloques cada una.

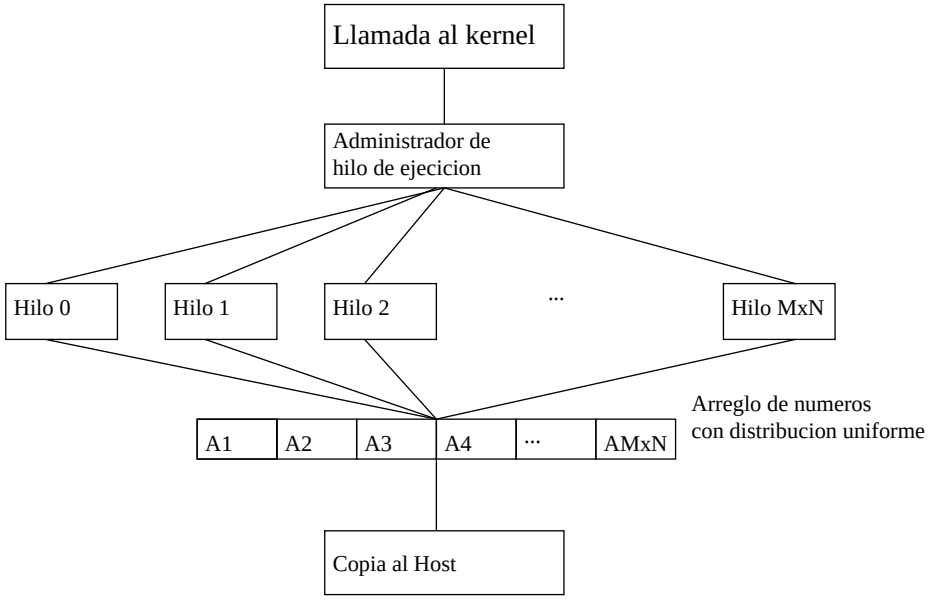

Figura 4.3: Proceso de ejecución del kernel para generar números con distribución uniforme a través de Mersenne Twister.

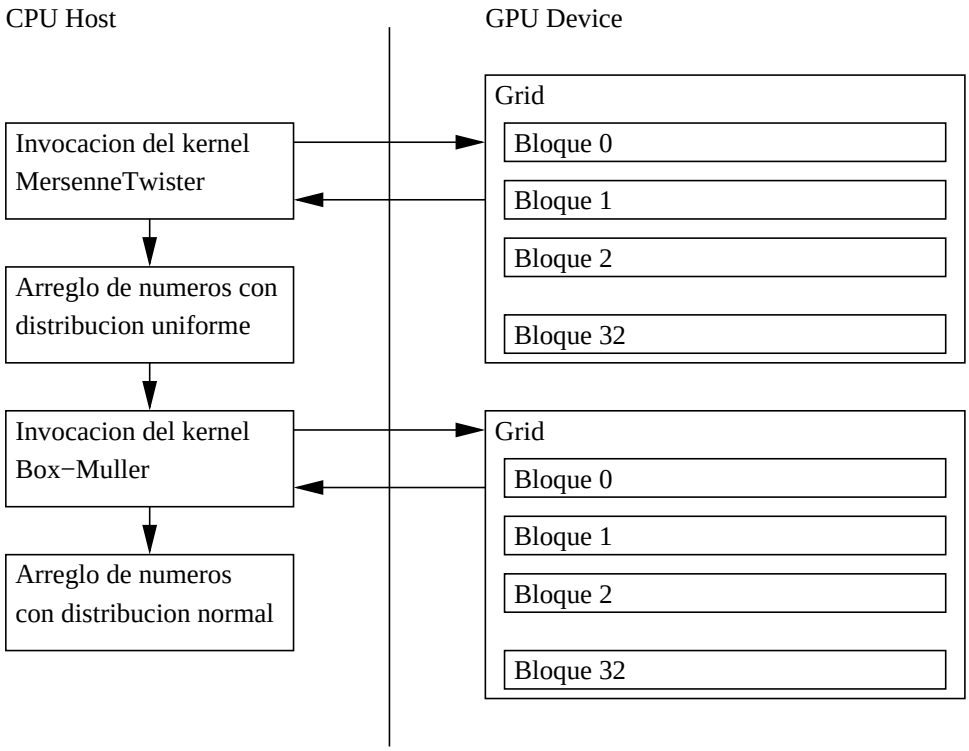

Figura 4.4: Proceso de ejecución de los kernel para generar números con distribución normal.

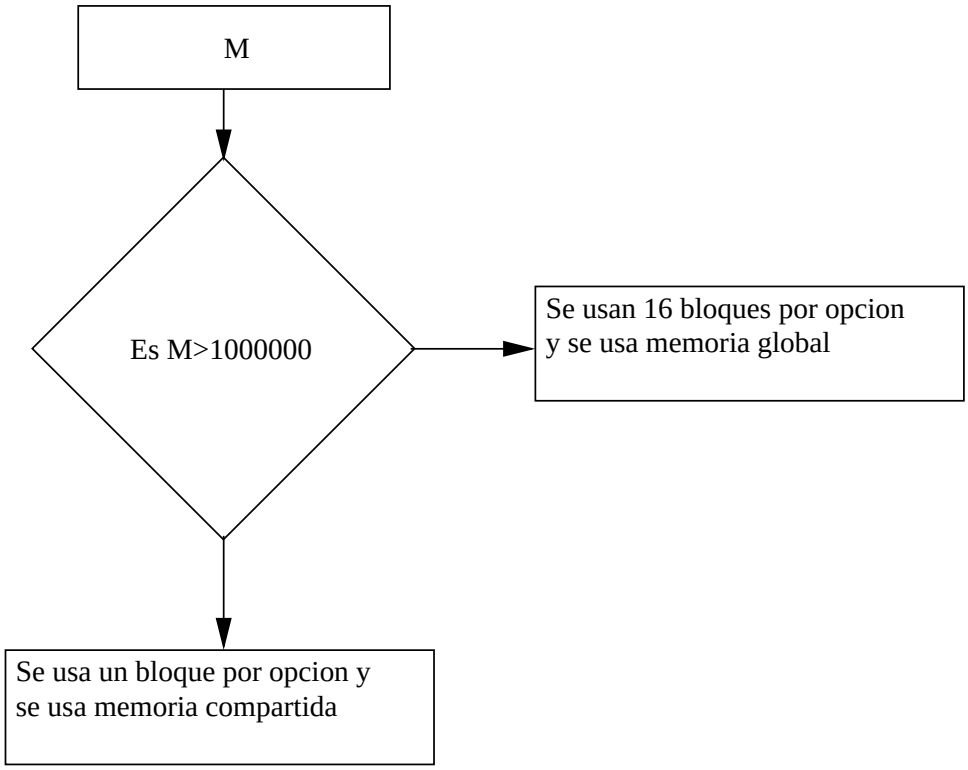

Figura 4.5: Proceso de decisión respecto a los bloques y el tipo de memoria usada para los cálculos.

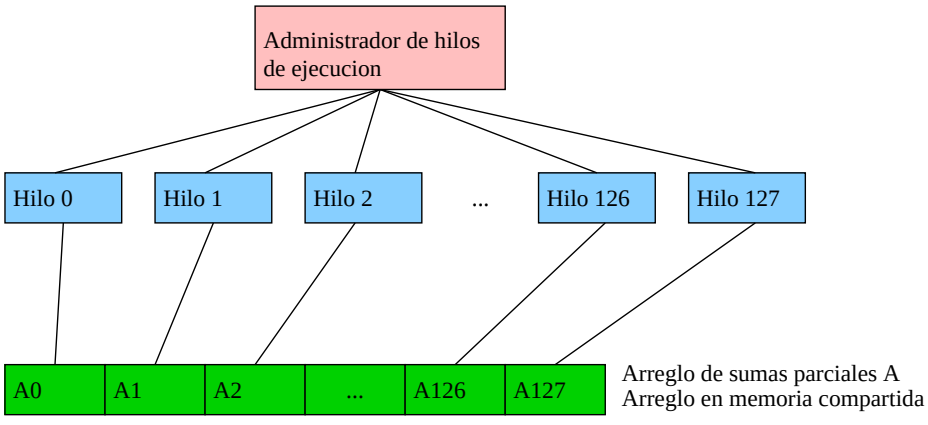

Figura 4.6: Acumulación de sumas parciales en la memoria compartida del bloque.

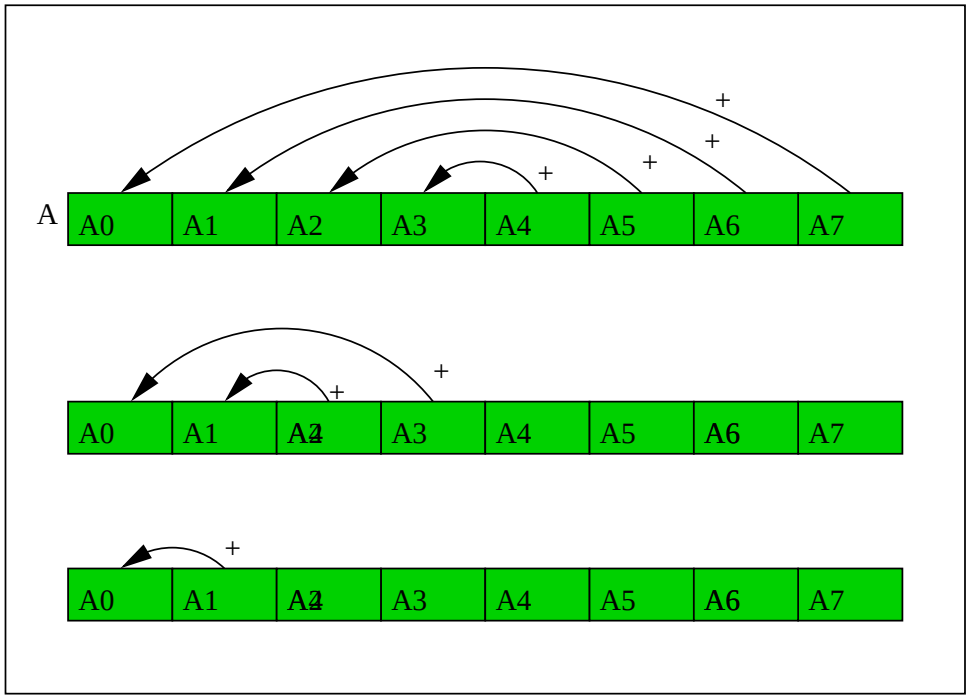

Figura 4.7: Reducción de las suma de un arreglo con sumas parciales en memoria compartida.

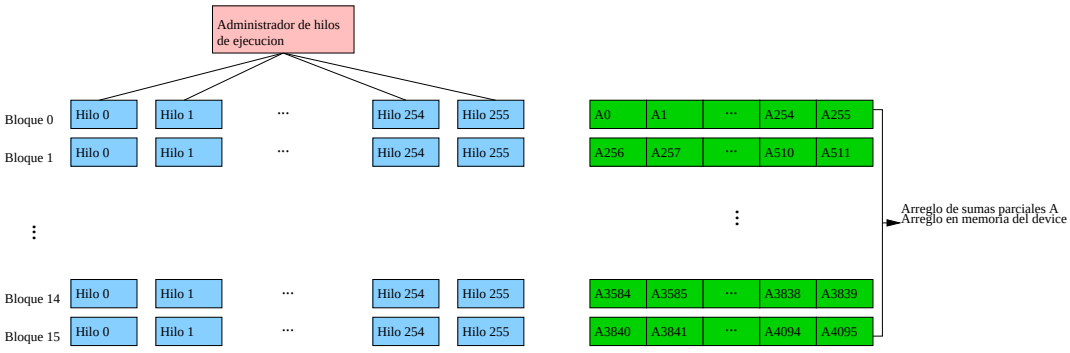

Figura 4.8: Ejecución de un kernel con múltiples bloques por opción

Capítulo 5

Resultados

En este capítiulo se presentan los resultados obtenidos en el presente trabajo de tesis. Se presentan las tablas de tanto de la implementación secuencial como de la paralela y los tiempos de ejecución.

5.1. Resultados de la implementación paralela

Los productos obtenidos con la implementación paralela hecha en cuda son un programa que ejecuta el método de Monte Carlo para opciones de barrera, pero que facilmente se extiende a cualquier opción que tenga dependencía de caminos. Se muestran los precios que obtenidos con diferentes combinaciones de parametros que van de 100 a 10000 con 100 o 1000 pasos cada una de ellas.

en la tabla 5.1 Se muestran las aproximaciones obtenidas con las diferentes combinaciones entre caminos y pasos por camino. No se muestran mas tablas con el contenido de 8, 16, 32 o mas opciones debido a que para una misma opcion al producirse la misma secuencia de numeros aleatorios se genera el mismo resultado. Por lo tanto en las aproximaciones obtenidas los unicos parametros que influyen son los caminos y los pasos.

La tabla 5.2 se muestran las aproximaciones obtenidas con la emplementación secuencial en C.

caminos/pasos	100	1000
100	0.04098	0.28714
1000	0.411352	0.293581
10000	0.391853	0.10345

Tabla 5.1: Aproximaciones implementación paralela

caminos/pasos	100	1000
100	0.164529	0.089883
1000	0.134951	0.0.089657
10000	0.104002	0.105598

Tabla 5.2: Aproximaciones implementación Secuencial.

5.1.1. Tiempos de ejecución para las pruebas realizadas con opciones de Barrera

En la presente secció se presentan las gráficas que muestran en verde el tiempo de ejecución de la implementación paralela con CUDA. cada gráfica muestra una diferente combinacion de caminatas generadas y el número de pasos que hay en cada una de ellas. las caminatas van de 100 a 10000 y el número de pasos de 100 a 1000. cada ejecución se realizó para valorar 2, 16, 32, 64, 128 y 256 opciones.

La gráfica 5.1 muestra la comparación de los tiempos de ejecución entre la implementación paralela y la implementación secuencial.

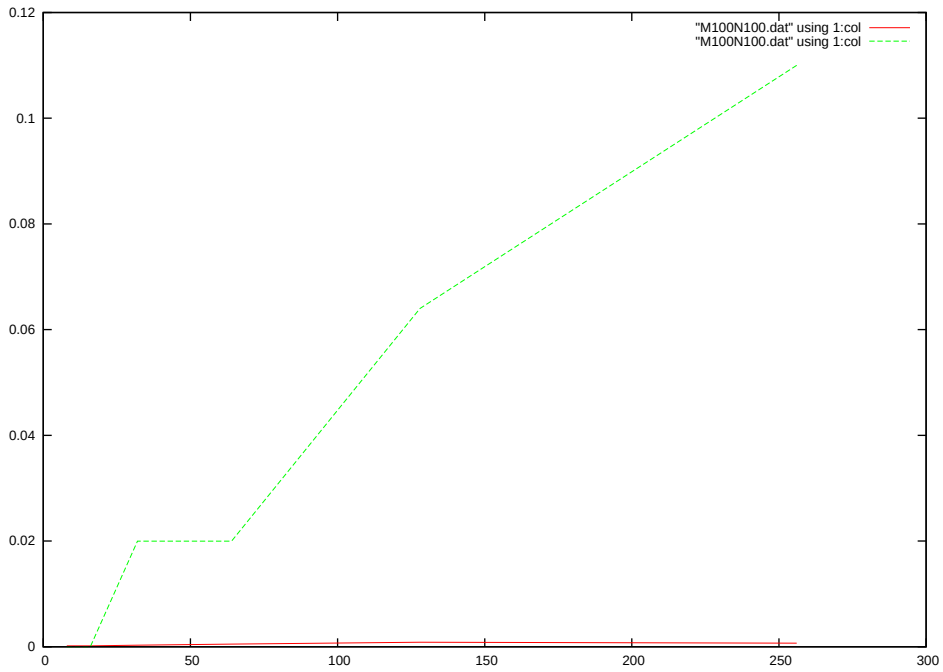

Figura 5.1: Tiempos de ejecución con 100 caminatas aleatorias y 100 pasos por caminata.

la gráfica 5.2 se muestran los tiempos de ejecución entre las implementaciones realizadas con un total de 1000 caminos y 100 pasos por camino.

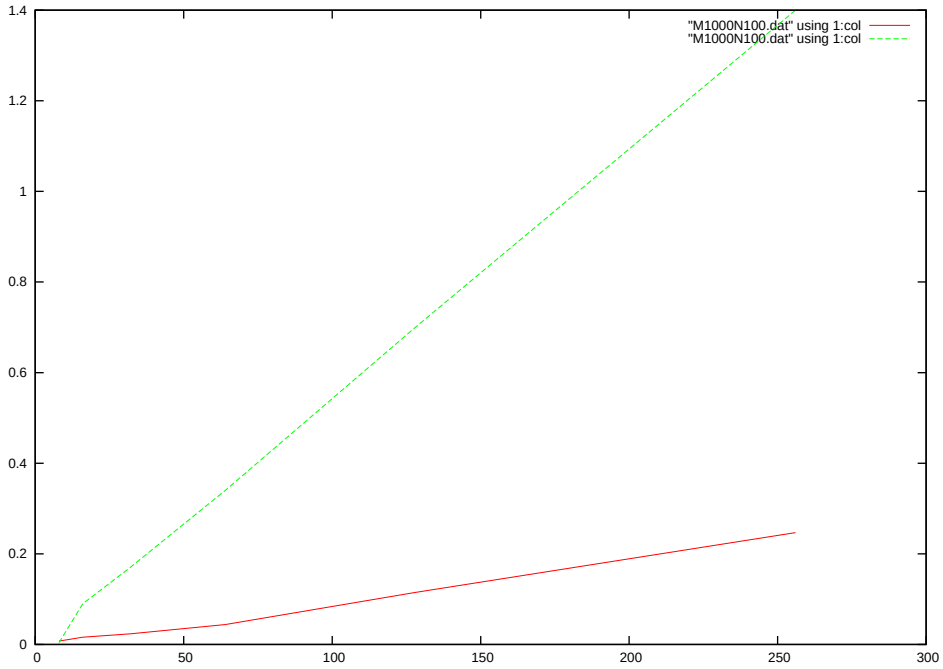

Figura 5.2: Tiempos de ejecución con 1000 caminatas aleatorias y 100 pasos por caminata.

La gráfica 5.3 muestra los tiempos de ejecución de las implementaciones realizadas con un total de 100 caminatas con 1000 pasos en cada una de ellas.

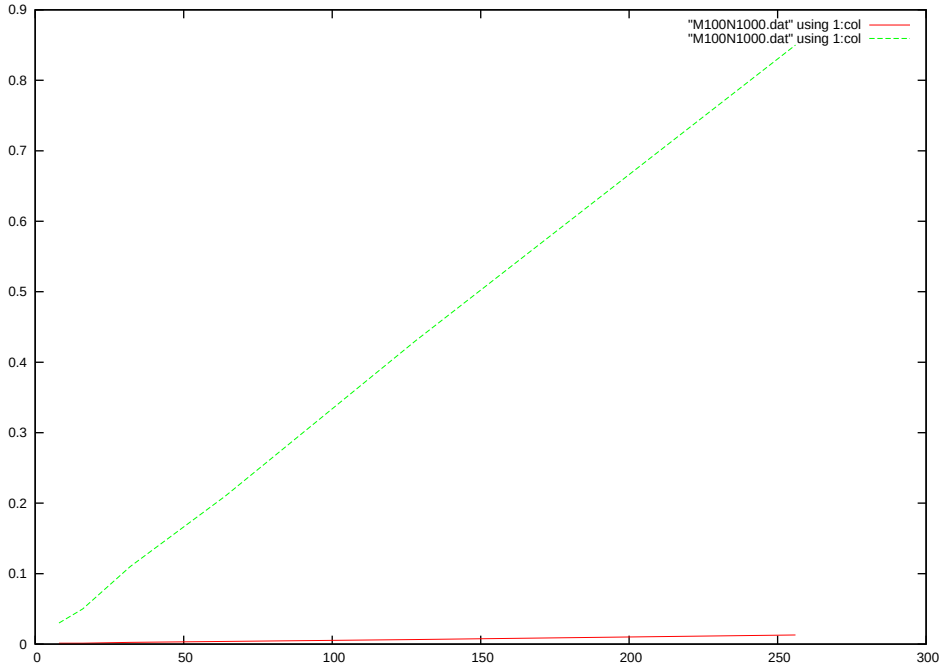

Figura 5.3: Tiempos de ejecución con 100 caminatas aleatorias y 1000 pasos por caminata.

La gráfica 5.4 Muestra los tiempos de jecucion obtenidos para 1000 caminatas aleatorias con 1000 pasos en cada una de ellas.

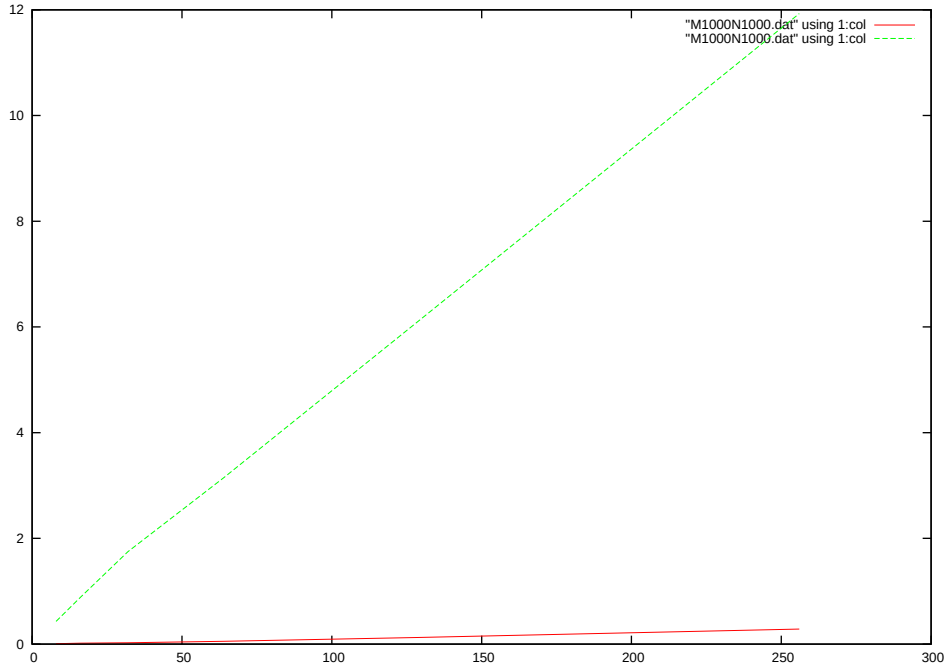

Figura 5.4: Tiempos de ejecución con 1000 caminatas aleatorias y 1000 pasos por caminata.

La gráfica 5.5 Muestra los tiempos de ejecución obtenidos generando 10000 caminatas aleatorias con 100 pasos por caminata.

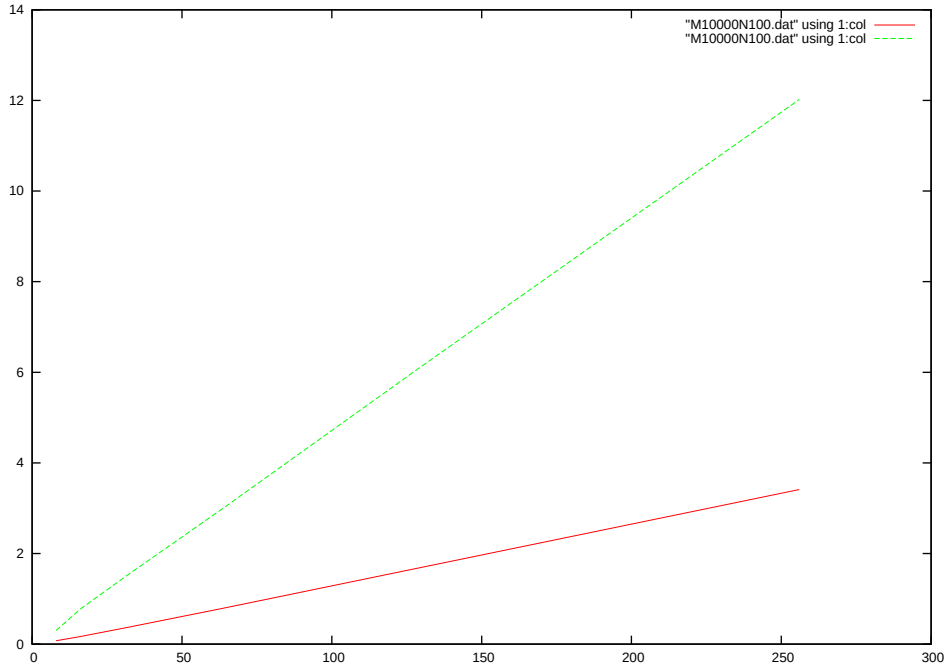

Figura 5.5: Tiempos de ejecución con 10000 caminatas aleatorias y 100 pasos por caminata.

La gráfica 5.6 muestras los tiempos de jecución obtenidos simulando 10000 caminatas con 1000 pasos en cada una de ellas.

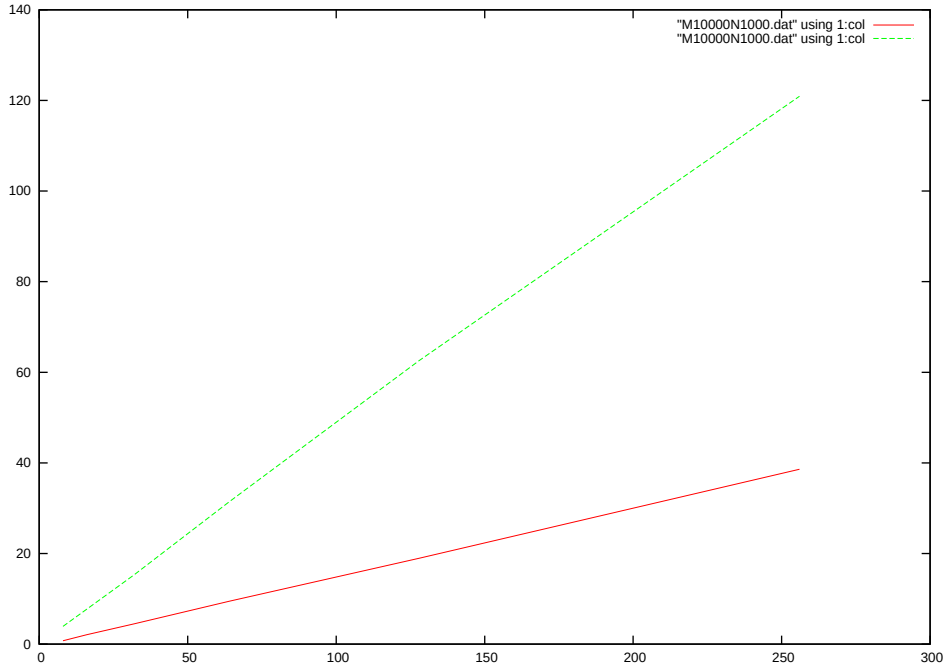

Figura 5.6: Tiempos de ejecución con 10000 caminatas aleatorias y 1000 pasos por caminata.

Capítulo 6

Concluciones y trabajo futuro

6.1. Conclusiones

Monte Carlo es un método muy costoso computacionalmete, pero tiene la ventaja de ser flexible y facilmente adaptable al cálculo de cualquier opción que tenga dependencia de caminos.

El método de Monte Carlo es un candidato ideal para la programación paralela de bido a su paralelismo intrinseco. A medida que la precisión de los resultdos quiere ser aumentada es necesario incrementar el número de ensayos que se realizan. En la presente implementación se tienen no sólo un determinado número de simulación de caminos, ademas cada camino se divide en cierto número de pasos lo que complica aún más el cálculo pues se requieren tantos números aleatorios diferentes como simulaciones de caminos se quieran hacer por el total de los pasos y cada camino podía tener una cantidad de pasos muy grande, aún cuando los caminos sean pocos.

El modelo de programación CUDA fue adecuado para la resolución del método de Monte Carlo debido a que quita toda la responsabilidad de pensar en como realmente se está ejecutando en hardware el programa. Es necesario conocer como es que la paralelización, la división del trabajo y la ejecución se está llevando a cabo, pero sin duda alguna es un lenguaje apropiado para ser usado en computo cientifico debido a su facilidad de aprender y lo parecido que tiene con C.

6.2. Trabajo futuro

Como trabajo futuro se tiene:

- Ampliar el programa para permitir la resolución de las integrales de Ito y Feyman.

- Ampliar el programa para permitir la generación de trayectorias utiles en problemas relacionados con las fisica, la quimica, la biologia etc.
- La realización de una interfaz grafica que pueda cargar datos reales de forma automatica o poder hacer configuraciones de entrada dentro de la misma interfaz además de una obtencion de resultados en archivos o en la interfaz.
- Realizar los ajustes pertienetes para que el programa pueda des ejecutado en cualquier tarjeta de la familia Nvidia.

Bibliografía

[1] Michael B. Giles. Multilevel monte carlo path simulation. *Operations Research*, 56(3):607–617, 2008.

[2] Michael B. Giles. *Monte Carlo and Quasi-Monte Carlo Methods 2006*, chapter Improved multilevel Monte Carlo convergence using the Milstein scheme, pages 343–358. Springuer, Berlin, Alemania, 2006.

[3] J. Andersen. Simple and efficient simulation of the heston stochastic volatility model. *Journal of Computational Finance*, 11(3):1–42, 2008.

[4] Elke Korn Ralf Korn and Gerald Kroisandt. *Monte Carlo Methods and Models in Finance and Insurance*. CRC Press, 1st edition edition, 2010.

[5] David P. Landau and Kurt Binder. *A Guide to Monte Carlo Simulations in Statistical Physics*. Cambridge University Press, 2nd edition, 2005.

[6] J. M. Thijssen. *Computational Physics*. Cambridge University Press, 2nd edition, 1999.

[7] John C. Hull. *Introducción a los mercados de futuros y opciones*. Prentice Hall, 6th edition, 2009.

[8] Andrew S. Tanenbaum; Todd Austin. *Structured Computer Organization*. Prentice Hall, 6th edition, 2013.

[9] et al S. Egger. Simultaneous multithreading: A platform for next-generation processors. *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1147 – 1153, Noviembre 2012.

[10] IBM. Blue gene, 2014.

[11] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.

[12] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions*, C-21(9):948–960, 1972.

[13] Nave J. Navarro E. *Fundamentos de matemáticas financieras*. Antoni Bosch editor, 1st edition, 2001.

[14] J. L. Doob. *Stochastic Processes*. Wiley, 1st edition, 1953.

[15] J. L. Hernández. Usos y limitaciones de la dinámica estocástica en el análisis macroeconómico convencional. In F. Miranda M. Ramos, editor, *Tópicos Selectos de Optimización*, volume 1, 2012.