Universidad Nacional Autónoma de México
Facultad de Ingeniería

Análisis comparativo de plataformas de desarrollo en sistemas paralelos y distribuidos

Tesis
Que para obtener el título de
Ingeniero en Computación
Presenta:

Carlos Roberto Álvarez Barragán

Directora de Tesis: Ing. Laura Sandoval Montaño

Ciudad de México, Octubre 2012

Universidad Nacional Autónoma de México
Facultad de Ingeniería

Análisis comparativo de plataformas de desarrollo en sistemas paralelos y distribuidos

Tesis
Que para obtener el título de
Ingeniero en Computación
Presenta:
Carlos Roberto Álvarez Barragán

Directora de Tesis: Ing. Laura Sandoval Montaño

Ciudad de México, Octubre 2012

Este trabajo de tesis fue desarrollado parte del proyecto PAPIME PE104911 *Pertinencia de la enseñanza del cómputo paralelo en el currículo de las ingenierías.*

Trabajo de tesis distribuido bajo licencia Creative Commons 3.0 de reconocimiento y licenciamiento recíproco:

- **Reconocimiento** - Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).
- **Licenciamiento recíproco -** Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

Dedicatoria

A mis padres, por ser los fundamentos y estructura en mi vida.

A mis hermanos, por mantener mis pies en la tierra.

A mi familia, en especial a mi tía Jenny, por el impulso que me ha dado.

A mis amigos.

Agradecimientos

Agradezco a toda institución involucrada en mi desarrollo, específicamente a la UNAM por formarme como ingeniero. A todos mis profesores, quienes tienen una labor importante en sus manos.

Un agradecimiento especial para la Ing. Laura Sandoval por todo el apoyo recibido a lo largo del desarrollo de este trabajo y la carrera en general.

Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.

– Donald Knuth, *Computer Programming as an Art*, 1974

Tabla de contenido

Introducción

La tecnología es cambiante. Desde la rueda hasta la ingeniería espacial, la humanidad ha cambiado paradigmas una y otra vez a través de la ciencia. Específicamente la computadora ha sido uno de los inventos que ha roto modelos y rutinas. Su evolución acelerada ha dejado boquiabiertos a muchos. Uno de los cambios más recientes en la electrónica, es el desarrollo de múltiples unidades de procesamiento en un solo chip. Este avance, por si sólo, es un gran logro ingenieril, pero se estancó debido a un mal aprovechamiento. El problema podría compararse con tratar de martillar con un destornillador.

El panorama ha mejorado, pero el cómputo paralelo sigue sin ser un elemento central en el desarrollo de software moderno. Afortunadamente, se han desarrollado herramientas que facilitan la adaptación, pero en ocasiones se sabe poco o nada de ellas. Este trabajo busca presentarlas y analizarlas para que sean mejor aprovechadas.

Uno de los objetivos centrales de la tesis es encontrar las ventajas y desventajas de distintas herramientas que facilitan el desarrollo sobre arquitecturas paralelas y distribuidas. Se busca entender cómo estas aplicaciones afectan la eficiencia de los distintos programas implementados, y cómo un algoritmo diseñado para trabajar en paralelo puede aprovechar las ventajas de las plataformas al máximo.

Es vital aclarar que por plataformas de desarrollo se hace referencia a *Application Programming Interfaces* (APIs por sus siglas en inglés) que facilitan la implementación de técnicas para distribuir tareas. Son definidas como plataformas porque forman parte de la estructura de desarrollo. Proveen una buena base que busca simplificar el trabajo.

Aunque el trabajo se centra en las plataformas de desarrollo; también toca otros temas relacionados sumamente importantes.

El capítulo 1 habla sobre los fundamentos teóricos relevantes para el trabajo. Éste resume mucha información con la intención de fundamentar las ideas desarrolladas a lo largo de la tesis. Habla brevemente dela historia del cómputo paralelo, de las arquitecturas computacionales y el análisis de algoritmos. El objetivo del capítulo es que los lectores del trabajo se familiaricen con el contexto actual de la tecnología en general y del tema en particular. Sin un contexto adecuado es difícil entender el alcance del tópico que se está tratando.

El capítulo 2 abarca la teoría relacionada con las dos plataformas que se utilizarán: Cilk y CUDA. Se detalla el funcionamiento de cada una, los problemas que pretenden resolver y se ejemplifica su uso. Técnicamente explica algo del código que es necesario para implementar algunos algoritmos clásicos.

En el 3er capítulo se encuentra la metodología que se utilizará durante la parte práctica de la tesis y plantea los ejercicios que se realizarán. El planteamiento consolida la mayoría de los

conceptos abordados en la primera parte del trabajo. Cada ejercicio cuenta con la explicación del algoritmo, el análisis secuencial y paralelo; también fragmentos trascendentales de código que será utilizado para obtener los resultados. Los algoritmos seleccionados están relacionados con tareas simples pero recurrentes en la ciencia de la computación: búsquedas, ordenamientos y operaciones con matrices. Así como forman parte elemental del cómputo tradicional, se considera importante implementar las variantes correspondientes al cómputo paralelo con ayuda de las plataformas de desarrollo.

Por supuesto se presentaron obstáculos que son explicados a detalle durante la implementación, pero los conocimientos del tema y una investigación adecuada permitieron resolver las trabas que surgieron.

El 4to capítulo se enfoca en el análisis de los datos partir de las prácticas. La información obtenida fue utilizada en representaciones gráficas, y fue interpretada con la intención de dar pie a las conclusiones de la tesis. El conjunto de datos busca de refrendar la teoría planteada en los capítulos previos.

No hay tesis sin conclusiones que cierren adecuadamente el trabajo. En esta sección se culmina con los puntos relevantes del trabajo, los resultados y el aprendizaje obtenido gracias al tiempo invertido. Se habla sobre la relevancia de las plataformas y su vigencia.

Cabe mencionar que esta tesis forma parte del proyecto PAPIME PE104911 *Pertinencia de la enseñanza del cómputo paralelo en el currículo de las ingenierías.* Este proyecto como su nombre lo indica busca mejorar el temario de algunas materias de la carrera de ingeniería en computación para incluir temas relacionados con el cómputo paralelo. La tesis respalda esto analizando nuevas herramientas y que de esta forma sean consideradas como instrumentos didácticos en las distintas materias de la carrera. Por esto no está de más hacer énfasis sobre la repercusión de una actualización educativa en términos de cómputo paralelo, para que los futuros ingenieros estén preparados para enfrentar lo que ya es una realidad en el mundo de la informática.

I Marco teórico y estado del arte

Antecedentes históricos y actualidad

Muy breve resumen de la historia de la computación

La tecnología en general y la computación en particular son áreas de la ciencia que han evolucionado rápidamente. La computación ha dado pasos agigantados desde mediados del siglo XX. Sin embargo, los conceptos fundamentales de esta ciencia surgieron desde que el humano empezó a trabajar con números. Tras siglos de desarrollo de las ciencias exactas, la máquina calculadora de Pascal fue una de las precursoras primitivas a las tecnologías actuales. Eventualmente, surgieron ideas y desarrollos científicos como: la máquina analítica, tarjetas perforadas, bulbos, transistores y chips de silicio. Cada avance compactó y volvió más eficiente a las computadoras hasta llegar a los electrónicos actuales.

Más rápido y compacto

Gordon E. Moore, uno de los fundadores de Intel, predijo que el número de componentes en los circuitos integrados se duplicaría en periodos de 18 a 24 meses. Él también mencionó en su artículo de 1965 que esta tendencia continuaría por lo menos por 10 años [1]. Casi 50 años después la ley sigue de alguna forma vigente; sigue existiendo un crecimiento constante, sin embargo, es menor respecto al fuerte crecimiento inicial de las décadas de los 70 y 80 [2]. Las observaciones de Moore han marcado el paso del desarrollo en la industria del hardware [3].

No ha sido posible mantener la tendencia inicial indefinidamente. De 1986 al 2002 el desempeño de los microprocesadores se incrementó, en promedio, 50% por año. Pero en los últimos años la tendencia ha cambiado; desde el año 2002 el desempeño de procesadores de un solo núcleo se ha incrementado únicamente en un 20%. A pesar de que sigue siendo una tendencia positiva, la caída es considerable. Esta baja se dio en parte porque en el 2005 la industria cambió de rumbo y decidió que para no llegar a un límite en desempeño debían cambiar su forma de atacar el problema. Esto dio mucha fuerza a las arquitecturas con más de una unidad de procesamiento en un solo circuito integrado. Así inicio la era de los comercialmente llamados “multicore”. [2]

¿Por qué es relevante el tema?

A pesar de que se dio un boom en las ventas comerciales de computadoras con más de un núcleo, los desarrolladores con poca información sobre el tema no le prestaban mayor atención a la nueva tendencia. Ellos siempre habían estado acostumbrados a que el hardware se adaptaba al software y no al revés. Por primera vez en la historia el desarrollo de software debió cambiar sus técnicas para implementar de la mejor manera posible soluciones en arquitecturas paralelas. [4], [5]

Por desgracia un pensamiento distribuido no es algo que se enseñaba o se enseñe comúnmente en las carreras de ingeniería en computación o desarrollo de software [2]. Por lo tanto, para muchos programadores el nuevo enfoque no parecía tan natural. Esta serie de eventos ha creado la necesidad de explicar, argumentar y convencer a la industria y a las instituciones educativas el valor de enseñar e implementar los nuevos paradigmas.

¿Por qué son útiles y por qué es importante desarrollar buenas prácticas y extender nuestros conocimientos más allá de la programación secuencial?

Hay varias razones que hacen de los sistemas paralelos opciones viables y necesarias. Entre otras los límites físicos que se están alcanzando con los procesadores de un solo núcleo.

El hardware eventualmente alcanzará varios límites. Uno de los que se vislumbra a mediano plazo es la miniaturización. Surgen varios problemas de hacer los transistores –y por ende microprocesadores– cada vez más pequeños. Primero se tiene el límite físico, los transistores atómicos ya han sido creados [6] y hacerlos más pequeños es una tarea que actualmente parece imposible. Gracias a la miniaturización, se pueden incrementar los transistores en un circuito integrado, pero también se necesita más energía para su funcionamiento. Mientras más energía se utiliza, más es disipada como calor. Se acumula tanto calor en los procesadores de alto desempeño que los sistemas de enfriamiento simples, que se encuentran en computadoras comerciales, no son capaces de disiparlo. De ahí surge el segundo problema: la necesidad insaciable de energía de los procesadores y sistemas de cómputo. La energía va relacionada con el costo de un sistema. No forma parte del costo directo que se cubre con la compra del equipo, pero sí es un costo indirecto. Sin embargo, el costo de los electrónicos también se ve afectado. Un procesador de un solo núcleo con un gran poder es considerablemente más caro que uno de dos núcleos, ya que el proceso de miniaturización del procesador monolítico es considerablemente más sofisticado. El gasto necesario para tener equipos de alto poder con un procesador monolítico vuelve inviable la inversión, porque la ganancia que se obtiene por unidad monetaria es menor mientras más poderoso es el microprocesador [2].

Las arquitecturas con un único núcleo también cuentan con ventajas. A pesar de ello, beneficio no es suficientemente alto en relación al costo, por lo que están desapareciendo, haciendo del cambio en el desarrollo aún más relevante.

Se podría pensar que más poder de cómputo es innecesario, y quizás en términos de cómputo doméstico sí se llegue eventualmente a un límite. Sin embargo, los consumidores siguen exigiendo aplicaciones cada vez más sofisticadas y detalladas. Adicionalmente, en el ámbito de la investigación y desarrollo científico es indispensable el poder de cómputo para poder procesar enormes cantidades de información rápidamente. Con mayor poder de cómputo se podría predecir con mayor anticipación, precisión y rapidez el tiempo meteorológico, también desarrollar métodos de producción más eficientes y predecir cómo los productos pueden afectar nuestro entorno antes de que puedan causar algún daño. La cantidad de información que se maneja aumenta exponencialmente [7], pero de nada sirve si no es posible analizarla e

interpretarla. Estas son sólo algunas razones por las que no es posible conformase con el poder de las computadoras actuales.

Tendencias en la industria del software

La industria del software guio a las nuevas tecnologías por muchos años. El desarrollo de software presionaba a los fabricantes de hardware para cubrir los crecientes requerimientos de sus aplicaciones y sistemas operativos. El hardware, al crecer por muchos años con la misma arquitectura, le dio muchas facilidades al desarrollo de software [4]. Un programa que tenía problemas de desempeño solucionaba él mismo en la siguiente generación de procesadores. Los desarrolladores no se tenían que preocupar, ya que sus aplicaciones funcionarían mejor con cada mejora en el desempeño del hardware. Por algunos años se ignoró el potencial de las nuevas arquitecturas; pero los papeles se han invertido. Hoy en día el hardware es cada vez más sofisticado y en muchos casos no es aprovechado al máximo por desarrolladores.

Entonces surge la pregunta: ¿Por qué es importante aprender a programar en paralelo? Peter Pacheco, autor del libro "Introduction to Parallel Programming", lo responde de manera sencilla: "El simple hecho de añadir más procesadores no mejora mágicamente el rendimiento de la gran mayoría de programas secuenciales." [1] Es necesario que las nuevas aplicaciones sean pensadas para trabajar sobre múltiples núcleos, porque la industria parece haberse decidido y no hay vuelta atrás [4]. En otras palabras, desarrollar aplicaciones que se adapten favorablemente al cambio, donde cambio significa incremento constante de procesadores.

Hay suficientes razones para mejorar las prácticas actuales del desarrollo de software. Este trabajo busca, en parte, fomentar nuevos conceptos y herramientas.

Arquitecturas

Para escribir programas paralelos eficientemente se necesita conocimiento de la plataforma física (hardware). También es necesario entender los conceptos centrales del paralelismo. En esta sección se describen a grandes rasgos las arquitecturas de hardware.

Arquitectura Secuencial

Los sistemas secuenciales son los que han dominado el mercado por más tiempo y se encargan de realizar una tarea a la vez. La arquitectura que ha dominado la mayor parte del mercado de las computadoras a lo largo del tiempo es la **Arquitectura von Neumann**. Esta arquitectura consiste de una **unidad de procesamiento central, una memoria principal y la conexión** que existe entre las partes.

[1] Capítulo 1 P. Pacheco, *An Introduction to Parallel Programming*, 1st ed. Morgan Kauffman, 2011, p. 382.

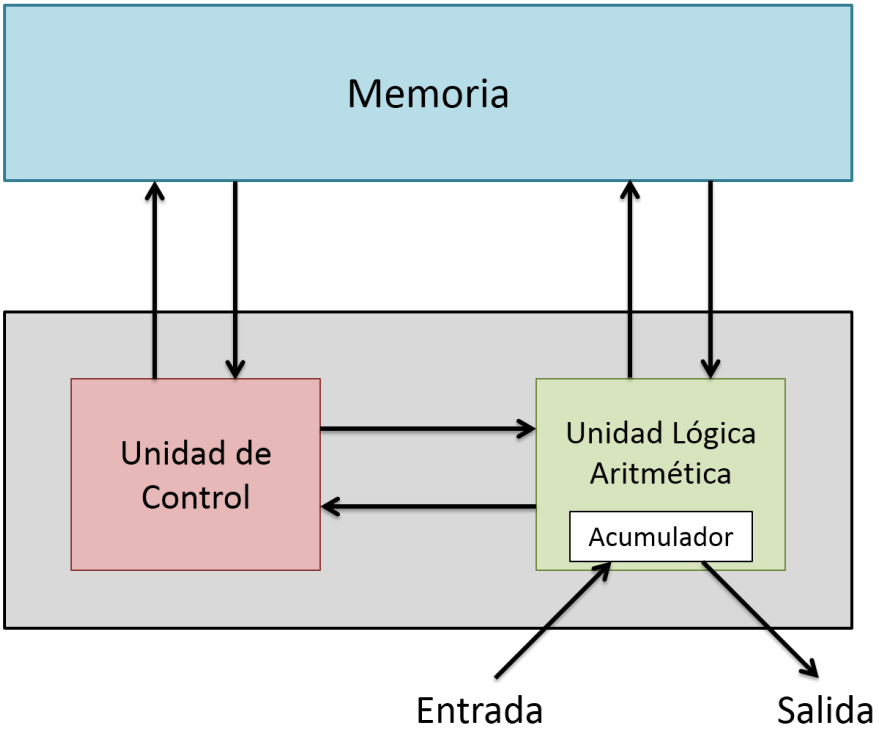


Figura 1 Diagrama de arquitectura Von Neumann

La **memoria** está compuesta de un conjunto de bloques de almacenamiento, cada uno capaz de guardar instrucciones e información. Cada bloque, a su vez, está formado por una dirección de memoria y la información que pertenece al bloque.

La **unidad de procesamiento central** (CPU por sus siglas en inglés) cuenta con una unidad de control y una unidad lógica aritmética.

- La **unidad de control** es responsable de coordinar las instrucciones que deben ser ejecutadas dentro de un programa y el orden que deben llevar. Adicionalmente cuenta con un registro llamado el **contador de programa** que almacena la siguiente instrucción a ejecutar.
- **La unidad lógica aritmética** se encarga de procesar comandos a través de operaciones simples (suma, resta, and, or, etc.).
- La información que será procesada inmediatamente es almacenada en una memoria de rápido acceso; los **registros**.
- La comunicación entre las distintas partes se da con distintos **bus** que sirven para transferir instrucciones e información entre memoria y CPU. Esta conexión consiste de una colección de cables paralelos y un microcontrolador para regular el flujo.

Esta es una breve descripción de los conceptos básicos de la arquitectura que forma parte de los cimientos de la computación moderna. Por supuesto hay variantes que buscan optimizar la Arquitectura von Neumann, pero la idea central permanece.

A pesar de que esta arquitectura fue crucial para el desarrollo de sistemas modernos, es posible encontrar un problema con ella. La separación entre CPU y memoria principal es conocida como el "cuello de botella de von Neumann" [2] ya que la conexión entre los dos elementos principales de la computadora es la que define la velocidad a la que se puede acceder a instrucciones o información. El problema se agrava porque la mayor cantidad de instrucciones e información se encuentra fuera del CPU.

Con adiciones a la arquitectura como lo es el uso de memoria caché y paralelismo a nivel de instrucciones ("Pipeline" de instrucciones) se ha logrado mejorar significativamente en su rendimiento [8]. Hoy en día prácticamente cualquier procesador moderno cuenta con dichas mejoras. A pesar de esto, fue necesario seguir adelante y se empezó a invertir en un paralelismo real, es decir, en la creación de múltiples unidades de procesamiento que auténticamente puedan llevar acabo tareas simultáneas.

Arquitecturas Paralelas

El hardware paralelo no es nuevo. Ya ha sido utilizado por varios años [9], y su uso se ha vuelto intensivo con la necesidad de sistemas súper-escalable a bajo costo [10]. El uso de clústeres de procesamiento ha abierto el camino para aplicaciones que requieren más poder de procesamiento de lo normal. Dividir la carga de trabajo es el lema de las "nuevas" arquitecturas. En la actualidad, es posible decir que el desarrollo paralelo se ha simplificado considerablemente gracias a las interfaces de programación de aplicaciones, o API por sus siglas en inglés (*Application Programming Interface*), que facilitan la división de tareas.

Categorización y conceptos

En el cómputo paralelo la **taxonomía de Flynn** es usada para clasificar arquitecturas. Ésta clasifica un sistema de acuerdo al **número de flujos de instrucciones y el número de flujos de datos** que puede manejar simultáneamente [11]. La arquitectura secuencial clásica es, por lo tanto, un sistema de **flujo de instrucciones sencillo y flujo de datos sencillo, SISD** (*single instruction stream, single data stream*).

SIMD – Single Instruction, Multiple Data

Estos son sistemas que manejan **un solo flujo de instrucciones y múltiples flujos de datos**. Estos sistemas ya son considerados paralelos. Operan en múltiples flujos de datos pero normalmente se realiza la misma operación sobre los distintos flujos. Estos sistemas se caracterizan por tener una única unidad de control y múltiples unidades lógicas aritméticas (ALU, en inglés: *Arithmetic Logic Units*). Una instrucción es difundida, desde la unidad de control, a todas las unidades lógicas y cada una lleva a cabo la operación en el bloque de información que tiene asignado.

Un ejemplo de este tipo de sistemas son las **unidades de procesamiento gráfico, GPU** por sus siglas en inglés (*Graphic Processing Unit*). Estas unidades cuentan con una gran cantidad de unidades lógico aritméticas que permiten que las enormes cantidades de información, necesarias para renderizar gráficos complejos, puedan ser procesadas eficientemente. Sin embargo, es necesaria una gran cantidad de información para realmente aprovechar a los GPUs, y en problemas pequeños, esta arquitectura puede ser ineficiente. A pesar de que la base de la estructura de una tarjeta gráfica es un sistema SIMD, algunas tarjetas gráficas más sofisticadas cuentan con varios núcleos que también son capaces de ejecutar instrucciones independientemente por lo que son dispositivos híbridos [2].

MIMD – Multiple Instruction, Multiple Data

Los sistemas MIMD son aquellos que llevan a cabo **múltiples instrucciones sobre múltiples flujos de datos**. Estos sistemas constan de colecciones de varias unidades de procesamiento totalmente independientes una de otra. A diferencia de los sistemas SIMD, los MIMD son normalmente asíncronos. De hecho la independencia es tal que en algunos sistemas el desarrollador debe implementar algunos elementos que permitan la comunicación entre los distintos procesos. Aquí es donde las nuevas herramientas se vuelven relevantes ya que permiten que el programador se olvide de implementar los detalles de bajo nivel.

Hay dos tipos principales de arquitecturas MIMD: una que utiliza memoria compartida y otra que utiliza memoria distribuida.

Están, por un lado, los sistemas paralelos que utilizan **memoria compartida** para intercambiar información. Ésta es utilizada por todas las unidades de procesamiento y se debe tener cuidado con los accesos simultáneos. Al compartir la información esta puede verse corrompida si es utilizada por más de un procesador simultáneamente.

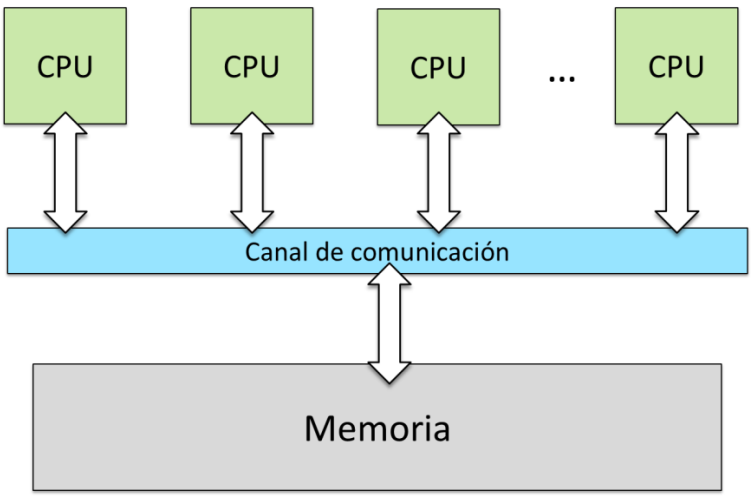


Figura 2 Diagrama de arquitectura con memoria compartida.

Por otro lado, un sistema con **memoria distribuida** está compuesto de unidades de procesamiento y memoria independientes que realizan un trabajo en equipo. Al no tener

memoria compartida no se puede modificar la memoria de una unidad a través de otra, pero sí existen problemas como el de la sincronización de la información y el uso adecuado de mensajes.

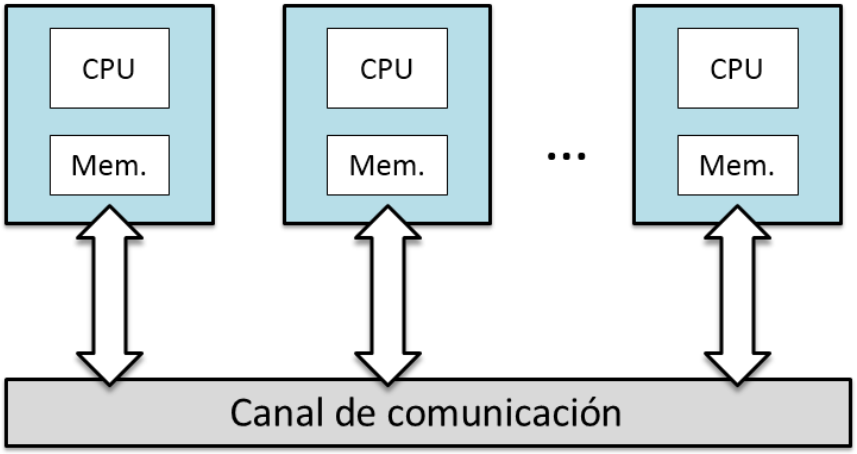


Figura 3 Diagrama de arquitectura con memoria distribuida.

Memoria compartida vs memoria distribuida

En general los programadores han adoptado el desarrollo para sistemas de memoria compartida, ya que el hecho de poder coordinar el trabajo a través de estructuras de datos es considerablemente más simple que el envío explícito de mensajes. Adicionalmente, la infraestructura es más barata para aplicaciones que no tienen requerimientos muy complejos.

Sin embargo, el principal problema con los sistemas de memoria compartida es la escalabilidad. Al crecer el número de procesadores, la posibilidad de que haya conflictos se incrementa considerablemente. Por ello los buses para la interconexión son adecuados únicamente para una cantidad reducida de microprocesadores, e implementar una red de interconexión sofisticada es sumamente costoso. Por otro lado, los sistemas de memoria distribuida son escalables a muy bajo costo y extremadamente útiles para proyectos en los que el poder de cómputo es prioridad; su desventaja se encuentra en que al programar hay que implementar la comunicación entre procesos también.

Más categorías

Además de la taxonomía de Flynn, que sirve para categorizar las arquitecturas. Es posible catalogar los algoritmos dependiendo de cómo es abordado el problema:

Paralelismo en datos – Existe con problemas que se resuelven dividiendo la información. Es utilizado principalmente en sistemas SIMD, en donde los flujos de información son divididos, pero en términos generales modificados por la misma operación. Esta categoría es particularmente eficiente cuando se debe trabajar con grandes cantidades de datos.

Paralelismo en tareas - Se trata de repartir las tareas que deben llevarse a cabo y es más fácil implementarlo en sistemas tipo MIMD. En este caso, distintas funciones son ejecutadas por diferentes unidades de procesamiento. Este tipo de paralelismo puede ser muy útil para paralelizar algoritmos con más ramificaciones.

Para ilustrar ambos casos se usará la siguiente analogía: Un grupo de diez inspectores debe realizar 10 pruebas de seguridad distintas a 1000 coches. Los trabajadores representan procesadores, los autos son datos y las pruebas equivalen a tareas. Hay dos formas de resolver el problema:

Con "paralelismo en datos" se divide el número de coches que se deben revisar entre los 10 trabajadores. Es decir, cada uno debe probar 100 coches.

Con "paralelismo en tareas" cada inspector debe realizar 1 sola prueba a cada uno de los 1000 autos. El problema que podría surgir es que no todos los inspectores pueden revisar el mismo coche simultáneamente. Por lo tanto, hay que prestar atención en coordinación de la carga de trabajo, la comunicación y la sincronización.

Conceptos básicos de la división de tareas

Un programa secuencial en términos generales es más simple que uno paralelo. Por ejemplo, suponiendo que se tiene un programa secuencial y otro paralelo, ambos buscan alcanzar el mismo objetivo, aunque sea por diferentes caminos. Si sólo se considera el objetivo final se puede pensar que ambos deben tener una complejidad similar. Sin embargo, el programa paralelo debe (en la mayoría de los casos) hacer una o varias tareas adicionales para llegar a la solución final del problema. Con el paradigma paralelo es necesario que los distintos núcleos se comuniquen, balancear la carga de trabajo y sincronizar la información que trabajan.

Para abordar estos conceptos se hará una analogía con un trabajo en equipo. Normalmente, trabajar con un grupo de personas significa dividir las tareas, para posteriormente reunir los resultados parciales y obtener uno final. No obstante, una división de trabajo no siempre es tan sencilla, por lo que es pertinente tomar en cuenta estos elementos en los programas paralelos.

- **Comunicación:** Cualquier equipo necesita una buena comunicación, especialmente cuando dos o más tareas están relacionadas entre si. Si un elemento del equipo encuentra información que puede afectar el trabajo de uno de los compañeros, éste debe informarlo, pues establecer un canal de comunicación es fundamental. En el caso de los procesos hay distintas formas de lograr esto (mensajes, memoria compartida, archivos, etcétera) dependiendo de qué se quiera comunicar y la arquitectura sobre la que se esté trabajando.
    - **Mensajes -** son bloques de información que se transmiten de un emisor a un receptor. Normalmente utilizado en arquitecturas de memoria distribuida.

- Las **estructuras de datos compartidas** son usadas principalmente sobre plataformas de memoria compartida. Regularmente, un proceso crea o modifica una estructura y posteriormente permite que otros procesos tengan acceso a la información modificada.

- Los **archivos** pueden funcionar de forma similar a como lo hace estructura de datos compartida. Sin embargo, se accede a estos a través de memoria secundaria (disco duro, memoria externa). Por lo que, mientras todos los procesos tengan acceso común a la memoria secundaria, puede utilizarse en cualquiera de los paradigmas.

- **Balance de carga:** En un equipo nunca falta algún integrante que se encarga de trabajar más que los demás. Ya sea juntando información o cubriendo a otros cuando éstos no aportan al trabajo. Así como no es una situación deseable en los trabajos de equipo, tampoco es algo que se busca en los desarrollos paralelos. De hecho, si existiera un desbalance de carga considerable, se pierde el objetivo inicial y el programa se empieza a parecer a uno secuencial. Debido a esto, se pueden entender la relevancia de una buena distribución en un sistema paralelo. Para evitar un desbalance de carga se plantea bien la solución, ya que una distribución adecuada de trabajo se define desde el diseño del algoritmo.

- **Sincronización:** Extendiendo la analogía, en ocasiones es posible que más de una de las tareas del trabajo abarquen puntos similares, o que más de un integrante del grupo modifique parte del trabajo final simultáneamente. Esto puede causar algunos problemas ya que, de no "sincronizarse" los cambios, puede haber pérdida de información. Algo similar pasa en la programación paralela, cuando un proceso hace uso de recursos compartidos, que están siendo utilizados por otro hilo de ejecución. Todos los procesos deben estar al tanto de los cambios. A este concepto se le llama sincronización. Sin embargo, puede estar ya implementado en algunas bibliotecas de desarrollo.

Problemas recurrentes del paralelismo

El mal manejo de la comunicación, balance de carga o sincronización deriva en problemas que surgen cuando se trabaja en paralelo. Estos están principalmente relacionados con la integridad de la información durante la ejecución del programa.

Las **secciones críticas** son segmentos de código que son utilizados por más de un proceso. Esto quiere decir que más de un hilo de ejecución completa el mismo conjunto de instrucciones. Al encontrarse en el mismo ámbito, los procesos tienen acceso a las mismas estructuras de datos y de no tener cuidado se puede perder o corromper la información.

Cuando el flujo de una sección crítica no es controlado adecuadamente es posible que surjan **condiciones de carrera** que pueden causar errores debido a que el resultado depende del orden en el que los distintos procesos atraviesan la sección crítica. Hay dos tipos de carreras:

- **Carreras de datos -** ocurren cuando más de un proceso escribe sobre el mismo espacio de memoria, sin haber sincronizado las operaciones de alguna forma.

- **Carreras deterministas -** éstas suceden en caso de que se bloqueé el acceso a memoria, para que sólo un proceso pueda modificarla. Sin embargo, el orden en el que los procesos modifican la variable influye, especialmente si la operación que se está realizando no es conmutativa.

Los problemas mencionados previamente surgen principalmente cuando se trabaja con memoria compartida. Sin embargo, el uso de memoria distribuida puede generar problemas similares si se manejan mal los mensajes.

Soluciones

La solución más simple, y no por ello menos eficiente, es la **exclusión mutua.** Ésta busca evitar que varios procesos trabajen dentro de una sección crítica si esta se encuentra previamente ocupada. Mientras ésta continúe ocupada, los demás procesos seguirán esperando. Una vez que termina, la sección se libera y otro proceso puede utilizarla. Una forma de implementar esta solución es a través de una bandera que informa a los demás procesos si la sección crítica está ocupada o no.

Un **semáforo** es una solución similar a la exclusión mutua. Ligeramente más versátil, es una variable que le asigna prioridad de entrada a los procesos durante la espera. El criterio puede ser que el proceso con mayor tiempo de espera sea el próximo en acceder, o por ser un proceso prioritario debido la tarea que desempeña.

Estas soluciones son la base para otras más complejas. El principal problema con resolver el problema de las secciones críticas es que se pierde el paralelismo de un programa.

Aplicaciones del paralelismo

Procesamiento de imágenes médicas

Una de las técnicas para diagnosticar cáncer de mama es a través de mamografías. Este es uno de los métodos que permite detección temprana del problema. Sin embargo, la técnica es cara y puede requerir varias imágenes de rayos X que generan riesgos secundarios para la paciente.

Por otro lado, el ultrasonido es más seguro, pero un ultrasonido convencional tiene sus limitantes. Sin embargo, un desarrollo prometedor sobre un ultrasonido de tres dimensiones pero la solución no se había implementado por falta de poder de cómputo. Con el uso de cómputo paralelo sobre GPUs se pudo implementar una solución.

Ciencias ambientales

Los detergentes son unos de los contaminantes más utilizados, el uso masivo los vuelve unos de los contaminantes más dañinos para el medio ambiente. Por ello, algunos científicos buscan que éstos mantengan su eficiencia, y al mismo tiempo reducir el impacto negativo que tienen.

Los surfactantes son el ingrediente más dañino de los detergentes, pero este compuesto define muchas de las características imprescindibles. Son los agentes que se adhieren a la suciedad y de esta forma en el momento del enjuague se llevan la suciedad que acumularon. Sin embargo, probar nuevos surfactantes es un proceso largo y costoso.

La Universidad de Temple que junto con Procter & Gamble ha estado trabajando en un simulador molecular de las interacciones de los surfactantes con el agua, tierra y otros materiales. Esto no podría ser llevado a cabo sin el poder masivo del cómputo gráfico. Ahora su proyecto cuenta con avances importantes.

Las aplicaciones van mucho más allá pero dos ejemplos concretos proveen una buena idea de lo que se puede hacer con esta tecnología. No sólo la industria de los videojuegos se ve beneficiada por los avances tan grandes que se han logrado. La astronomía con simulaciones sumamente complejas, química con el cálculo de proteínas, meteorología con predicción más sofisticada del clima, son algunas de las materias que requieren de cómputo de muy alto desempeño para alcanzar sus objetivos.

Algoritmos

En el planteamiento de los problemas que el presente trabajo cubre, se recurre al análisis de los algoritmos. Éste es un tema cuyo dominio es obligado para cualquiera que busca desarrollar software de calidad, ya que el análisis sirve para respaldar decisiones del diseño de una aplicación.

Cimientos del desarrollo

El diseño y análisis de algoritmos es un tema vital de la Ingeniería y Ciencias de la computación. Un algoritmo es el conjunto de pasos requeridos para obtener un resultado. Es posible decir que los algoritmos son una "receta" con la que se encuentra la solución a un problema determinado. Son soluciones genéricas que pueden ser implementadas en todo tipo de plataformas y con todo tipo de tecnologías.

En el libro "Introduction to Algorithms"[12] por los autores Cormen, Leiserson, Rivest y Stein se encuentra la siguiente definición:

"Algoritmo es cualquier procedimiento computacional, bien definido, que toma un valor o grupo de valores (parámetros) como entrada y a su vez genera uno o más valores que son conocidos como salida o resultado. Un algoritmo es por lo tanto una serie de pasos que transforman la entrada en la salida."[2]

Algunos de los tipos de problemas para los cuales existen algoritmos son:

Ordenamientos: Uno de los problemas más sencillos de comprender, porque es simple e intuitivo. Como entrada se tiene una serie de elementos desordenados, y el objetivo es ordenarlos con base en un criterio determinado (tamaño, valor, orden alfabético). Es una

[2] Capítulo 1, p. 5, *Introduction to Algorithms*, 3rd ed. Boston, MA: The MIT Press, 2009, p. 1292.

tarea que a su vez es utilizada para resolver problemas más complicados, o para proveer funcionalidades básicas a una aplicación.

Búsquedas: Necesidad diaria del ser humano moderno. La búsqueda de información en la actualidad se ha vuelto sumamente eficiente. Google creó una buena solución a este problema, probando que la solución de problemas "simples" puede trae sus recompensas. Los buscadores solucionan el problema con el uso de algoritmos sumamente eficientes.

Seguridad: En esta rama de la computación se encuentran algoritmos criptográficos utilizados para proteger la integridad y confidencialidad de los datos. Se han implementando distintos estándares, buscando mejorar las condiciones de la información en la red y nuestros equipos. Los algoritmos de cifrado y digestivos como RC4, AES, SHA1 son todos estándares de la industria.

Computación gráfica: La mayoría de las aplicaciones de diseño gráfico trabajan con algoritmos sumamente sofisticados para proveer herramientas como filtros, iluminación y sombras, manejo de colisiones, entre otros.

Son algunas de las disciplinas que utilizan algoritmos eficientes para solucionar problemas grandes. Sin embargo, los algoritmos están en todos lados: economía, manufactura y todo tipo de aplicaciones científicas (predicción climática, procesamiento del genoma humano y descubrimiento de proteínas).

Análisis de algoritmos

Ya que los algoritmos abarcan muchos problemas y proveen muchas soluciones surgen nuevos inconvenientes. Esto se debe a que es posible encontrar más de una respuesta a un problema y cada una puede ser satisfactoria. Pero, ¿qué pasa si las soluciones existentes no convencen?, ¿es posible diseñar nuevas soluciones? Entonces, ¿cuál algoritmo se debe de utilizar? ¿Cuál es la mejor solución a un problema? Estas preguntas dan pie al análisis de algoritmos; tema central de las ciencias de la computación y el desarrollo de software. En la mayoría de los casos hay más de una forma de resolver un problema. El trabajo de cualquier ingeniero es saber identificar y justificar las soluciones que mejor resuelven un problema, pero no es tan simple. Hay que tomar en cuenta que ciertos algoritmos pueden funcionar mejor o peor dependiendo del contexto. Por ello, un análisis adecuado puede hacer la diferencia entre una buena y una óptima implementación.

El análisis es una herramienta teórica. Similar a lo que sucede cuando se modela matemáticamente, no se toman en cuenta todas las variables que forman parte del problema real. Y aunque el modelo teórico no considera toda la información, el resultado suele ser aceptable; y, en la mayoría de los casos muy similar a la realidad. En el caso del análisis de algoritmos se eliminan variables como la marca o la calidad de la computadora en la que se ejecutará la aplicación.

Esto no significa que la solución final (real) no deba ser probada minuciosamente. Una buena solución teórica sienta bases fuertes para proceder al desarrollo final. Que, de ser

suficientemente robusta, en teoría sólo requiere una buena implementación (con todo lo que "una buena implementación" implica).

"El análisis de algoritmos es el estudio del desempeño y manejo de recursos de un programa computacional."[3]

Uno de los objetivos del desarrollo es el desempeño óptimo de un programa computacional. Claramente no es la única característica deseable en una aplicación, pero un desempeño óptimo permite obtener otros beneficios, como mejores gráficos, aplicaciones más seguras, escalables y robustas. Ya que si la aplicación ni siquiera puede resolver el problema central satisfactoriamente no es posible añadir "lujos". Su importancia yace en crear cimientos sólidos para el programa.

El análisis de algoritmos mide dos recursos computacionales: el tiempo de ejecución y la memoria que un programa ocupará. Éstos se obtienen en función del tamaño de la entrada. Hacer que un algoritmo sea eficaz (rápido) y además eficiente (mínimo gasto de memoria) es una tarea sumamente difícil, muchas veces se tiene que establecer un punto medio, ya que un programa no puede llegar a ser demasiado rápido sin un gran costo en memoria y viceversa.

Este trabajo se enfoca principalmente en la ganancia temporal de un programa con el uso de más procesadores, pero si se considera pertinente, se mencionará la complejidad de memoria.

Antes de ahondar más en lo que es el análisis de algoritmos es necesario definir la notación asintótica.

Notación asintótica

La notación es el lenguaje con el cual se expresa el análisis. Ésta sirve para representar de manera simple el comportamiento de un algoritmo. Así es fácil comparar la eficiencia de dos o más algoritmos.

La notación asintótica se asemeja a las funciones algebraicas, pero en ella se toma en cuenta únicamente el término de mayor orden dentro de la expresión. Para obtener la complejidad en tiempo de forma "manual" se cuentan el número de instrucciones que ejecuta un programa. El objetivo es obtener una función que dependa del tamaño de la entrada (se representa con una 'n').

[3] Capítulo 2, p. 23, *Introduction to Algorithms*, 3rd ed. Boston, MA: The MIT Press, 2009, p. 1292.

```
variables suma, n, i=1
leer(n)
repetir(desde 1 hasta n)
      suma=suma+i
      i=i+1
imprime("promedio 0 a %d es %.2f",n,suma/n)
```

Código 1 Pseudocódigo para la obtención de un promedio dada una entrada n.

En el pseudocódigo anterior se recibe una variable entera llamada 'n'; a partir de ésta se repiten las operaciones dentro del ciclo 'n' veces. Si se cuentan las instrucciones es posible darse cuenta que todo lo que se encuentra fuera del ciclo siempre tomará la misma cantidad de tiempo (es constante). Sin embargo el ciclo es el corazón del algoritmo. Dentro de éste, las operaciones se ejecutan dependiendo del tamaño de la entrada (n). Si la 'n' es igual a 5, las operaciones que contiene la estructura de control se repetirán 5 veces. Cuando se habla de una entrada cualquiera, se dice que el ciclo se completa 'n' veces. Por lo tanto, es posible representarlo con una función lineal:

$$T(n) = n + c$$

Donde:

- $T(n)$, es igual al tiempo total que tomará la ejecución del programa.
- n, es la variable que se toma como entrada.
- c, es el tiempo que requieren las operaciones constantes del pseudocódigo.

Sin embargo, en notación asintótica sólo es relevante el término de mayor orden el resultado será algo similar a:

$$T(n) = n$$

Este análisis de complejidad es sencillo. Es posible concluir que el crecimiento en tiempo respecto a la entrada de este programa es lineal. Por lo tanto, cuando n tiene un valor de 1,000,000 el programa tomará 1,000,000α unidades de tiempo (donde α es una constante de tiempo). El análisis no siempre es tan sencillo; las funciones recursivas pueden ser ligeramente más difíciles de representar.

En el ejemplo anterior se usó la notación 'O' mayúscula. A continuación se explica a detalle el significado de esa y otras notaciones necesarias para el análisis.

- **Notación O**

$$O\big(g(n)\big) = \{f(n) \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)\}$$

$f(n)$ forma parte de un conjunto de funciones $O(g(n))$ si y sólo si existen las constantes positivas c y n_0, tal que para toda $n \geq n_0$, la siguiente expresión $0 \leq f(n) \leq cg(n)$ se cumple.

- **Notación Ω**

$$\Omega\big(g(n)\big) = \{f(n) \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq cg(n) \leq f(n)\}$$

$f(n)$ forma parte de un conjunto de funciones $\Omega(g(n))$ si y sólo si existen las constantes positivas c y n_0, tal que para toda $n \geq n_0$, la siguiente expresión $0 \leq cg(n) \leq f(n)$ se cumple.

- **Notación Θ**

$$\theta(g(n)) = \{f(n) \mid \exists c_1 > 0, \exists c_2 > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

$f(n)$ forma parte de un conjunto de funciones, $\theta(g(n))$ si y sólo si existen las constantes positivas c_1, c_2 y n_0, tal que para toda $n \geq n_0$, la siguiente expresión $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ se cumple.

Las definiciones matemáticas pueden parecer complicadas, pero representan conceptos simples. A continuación, se desglosan los distintos elementos:

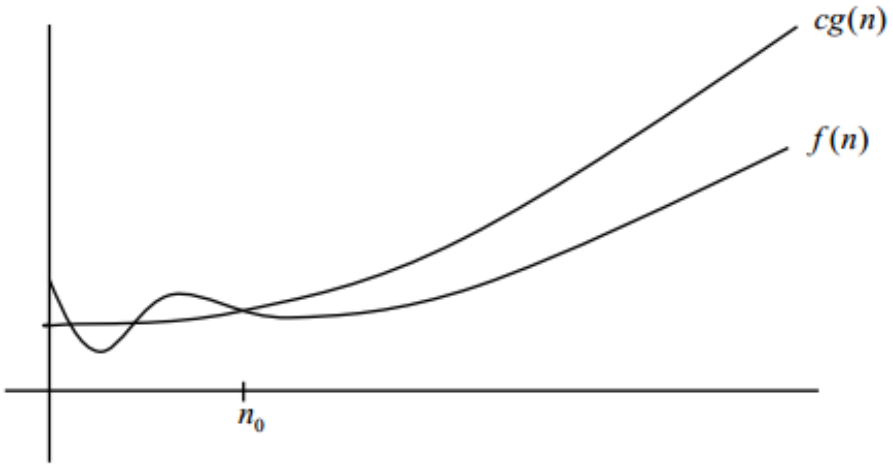
- $\exists \boldsymbol{c > 0}$ Debe **existir** una constante **c mayor a cero**. Esta constante multiplica a la función $g(n)$, que representan el límite asintótico (sea mayor o menor) de $f(n)$. En el caso de la notación θ se tienen dos fronteras, por lo que son necesarias dos constantes diferentes.

- $\exists \boldsymbol{n_0 > 0}, \forall \boldsymbol{n \geq n_0}$ Estas son dos expresiones. La primera dice que es necesario que **exista n_0 mayor a 0**. La segunda expresión **para toda n es mayor a n_0**. Juntando los dos significados es posible concluir que, **a partir de que n es mayor a n_0,** $f(n)$ debe encontrarse dentro de los límites correspondientes a la notación con la que se esté trabajando (O, Ω o Θ).

- Hay 3 distintos límites dependiendo de la notación.

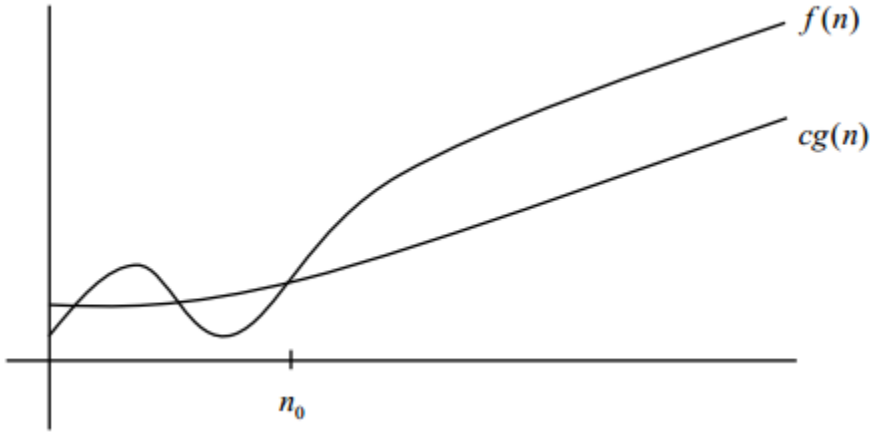
    - $0 \leq f(n) \leq cg(n)$ Es el caso de la **notación O** $f(n)$ para toda ($\forall n \geq n_0$) debe ser **mayor a 0 y menor o igual a $\boldsymbol{cg(n)}$** . De esta forma las condiciones se

cumplen y $f(n)$ forma parte del conjunto de funciones menores e iguales a $\boldsymbol{cg(n)}$.



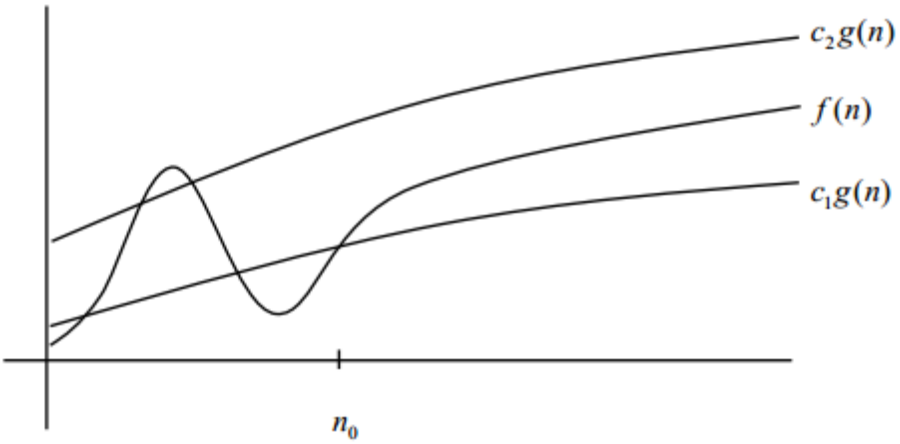
Gráfica 1 Representación genérica de la notación O [13].

- $0 \leq cg(n) \leq f(n)$ para la **notación Ω,** $f(n)$ debe de ser mayor a $cg(n)$ a partir de n_0.



Gráfica 2Representación genérica de la notación Omega [13].

- $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para la **notación Θ,** $f(n)$debe permanecer dentro del rango de valores entre $c_1 g(n)$ y $c_2 g(n)$ a partir de n_0.



Gráfica 3 Representación genérica de la notación Theta [13].

Las tres notaciones representan tres situaciones distintas. 'O' representa un límite superior, por lo tanto el algoritmo no puede utilizar más recursos de lo que resulte del análisis. Θ representa un intervalo y Ω representa un mínimo, que a su vez representa la menor cantidad de recursos requeridos para completar el programa. El objetivo detrás de estos conceptos, es **entender el desempeño en las condiciones menos favorables**. De esta forma es posible dar una **garantía** sobre la eficiencia del programa y esta se obtiene con la notación 'O'.

Análisis de Algoritmos Paralelos

El análisis para algoritmos paralelos cuenta con las mismas bases, pero es necesario revisar conceptos adicionales. Hasta ahora se ha justificado el uso de algoritmos paralelos y se han cubierto los cimientos para entenderlos. A fin de cuentas, el objetivo del cómputo paralelo es simple: un mejor desempeño y escalabilidad.

Actualmente, existen aplicaciones que aprovechan el paralelismo, éstas todavía contienen rutinas secuenciales debidas, principalmente, a secciones críticas o cuellos de botella.

El **incremento en velocidad a través de la escalabilidad,** es el objetivo natural del cómputo paralelo. Para obtener el incremento es necesario obtener el **tiempo de ejecución en paralelo** del algoritmo (T_p) que, a su vez se calcula dividiendo el **tiempo** que toma al algoritmo ejecutarse **en un solo procesador** (tiempo secuencial T_s) entre el **número de procesadores** (p).

$$T_{paralelo} = \frac{T_{secuencial}}{p}$$

Una vez que se tiene T_p, el incremento en velocidad está definido por la ganancia de T_p sobre T_s.

$$S = \frac{T_{secuencial}}{T_{paralelo}}$$

Además, la **eficiencia** es el incremento en velocidad (S) obtenido entre el número de procesadores (p).

$$E = \frac{S}{p}$$

Estos datos teóricos sirven para respaldar el desarrollo paralelo de un algoritmo, porque si no se tiene una ganancia suficiente o la eficiencia es baja, hay que valorar el esfuerzo y costo ligados a modificar el programa.

Límites del paralelismo

Aunque un algoritmo paralelo signifique un mejor desempeño, las viejas prácticas de desarrollo secuencial derivan en cuellos de botella, y estos limitan el potencial.

En los años 60, Gene Amdahl planteó el límite teórico de la paralelización. Él dice que, a menos que virtualmente todo programa serial sea paralelizado, el potencial incremento en velocidad será limitado, independientemente del número de núcleos de procesamiento disponibles [14]. Por lo tanto, sólo se puede paralelizar parcialmente un programa, ya que aún no existe un programa cien por ciento divisible.

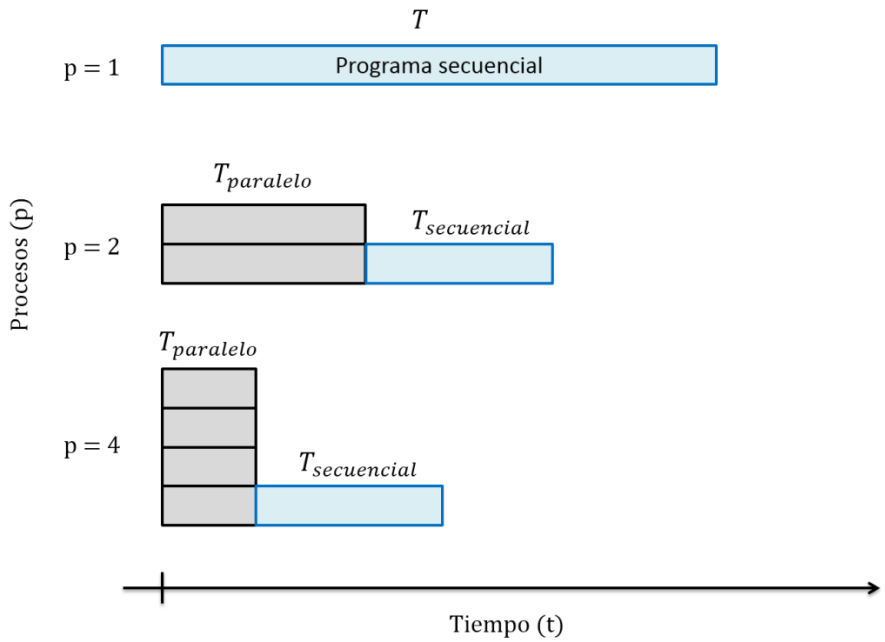


Figura 4 Visualización de los límites del paralelismo.

En teoría, aunque el número de procesadores y poder de cómputo tienda a infinito, existe un límite en velocidad debido a las partes secuenciales del programa. Bajo esta premisa, el tiempo de ejecución de la sección 'paralelizable' tendería a un tiempo constante, pero el tiempo secuencial se mantendrá independientemente de la cantidad de procesadores.

Trabajo, profundidad y paralelismo

El análisis de algoritmos paralelos cambia un poco respecto al secuencial. Principalmente porque se miden dos elementos distintos.

El **trabajo** es equivalente a la complejidad temporal para el análisis de algoritmos secuenciales, que es la cantidad total de operaciones realizadas por un algoritmo. Es decir, **el tiempo que tomaría al programa ejecutarse, si sólo contara con un procesador**. Este atributo se representa con una W.

Por otro lado, la **profundidad** es la cadena más larga de operaciones dependientes entre sí. **Es equivalente al tiempo que tomaría el programa en ejecutarse si se contara con procesadores infinitos**. Este atributo se representa con una D.

Por último, con el trabajo y la profundidad es posible calcular qué tan 'paralelizable' es un programa. A este concepto se le conoce como **paralelismo,** y se calcula de la siguiente manera:

$$P(n) = \frac{W(n)}{D(n)}$$

Este valor dice qué tan 'paralelizable' es un algoritmo, ya que, mientras mayor sea el resultado de la división, el algoritmo es un mejor candidato a ser adaptado.

II Plataformas

Cilk

Cilk es una potente herramienta que ha cambiado mucho en casi veinte años. Los creadores definen a **Cilk++ como una extensión del lenguaje C++** para simplificar la creación de aplicaciones que eficientemente exploten el potencial de múltiples procesadores [15]. También es un sistema en tiempo de ejecución para la implementación de algoritmos paralelos.

La filosofía detrás de Cilk busca que un programador pueda concentrarse en estructurar el desarrollo para exponer y explotar el paralelismo, y dejar que la herramienta se encargue de las responsabilidades de bajo nivel, como balance de carga, comunicación entre procesos y sincronización. Es particularmente útil cuando se trabaja con algoritmos del tipo **divide y vencerás**[16]**,** ya que esta técnica es usada para resolver problemas al dividir el problema principal en sub-tareas que pueden ser resueltas independientemente.

Las tareas pueden ser implementadas en funciones independientes o en estructuras iterativas. El **sistema en tiempo de ejecución es el encargado de planificar para que la ejecución se complete eficientemente** [16]. El programador puede especificar el número de procesadores que se usarán durante la ejecución.

Un poco de historia

Cilk surgió en 1994 como parte del trabajo de Charles E. Leiserson en el Laboratorio de Ciencias de la Computación del MIT. El éxito y el prospecto de procesadores comerciales con varios núcleos de procesamiento dispararon a Cilk. Leiserson inició Cilk Arts en Massachusetts donde desarrollaró una versión mejorada: Cilk++. Esta versión ya incluía la posibilidad de paralelizar estructuras iterativas, entre otras cosas. En Julio del 2009 Intel adquirió Cilk Art y liberó su compilador ICC con Intel Cilk Plus que provee una manera sencilla de habilitar la extensión en aplicaciones de C y C++. [15], [17], [18]

Con Cilk se desarrollaron proyectos para concursos de ajedrez. Los resultados cada año mejoraban y siempre se posicionaron dentro de los tres primeros lugares [17], demostrando que se pueden desarrollar programas bajo el paradigma de programación paralela sumamente sofisticados y escalables. Dentro de las características destacables se encuentra el potencial de crecimiento en desempeño si se incrementan las unidades de procesamiento, lo que vuelve el software escalable automáticamente. Transmitir el alcance de la escalabilidad ha sido parte central de los esfuerzos de intelectuales, pero principalmente de la industria del hardware (Intel y AMD), para educar a las nuevas generaciones para que piensen “en paralelo”.

La herramienta ha cambiado desde que fue comprada por Intel, pero no demasiado. Durante el trabajo se utilizará una versión libre lanzada en el 2009 (Intel Cilk++) [17], [18].

Características

- **Sencillo**, porque únicamente requiere un puñado de palabras reservadas para funcionar adecuadamente.
- Cuenta con un **traductor** que se encarga de convertir las directivas de Cilk en instrucciones de C/C++ para que la compilación sea óptima.
- **Un sistema en tiempo de ejecución**, probablemente la pieza más importante. Se encarga de hacer el desempeño del programa sumamente eficiente. Se ocupa de los detalles que habían atormentado a desarrolladores de aplicaciones para sistemas multicore por algún tiempo (balance de carga, comunicación entre procesos y sincronización).

Requerimientos

- Computadora con arquitectura Intel Pentium 3 en adelante, preferiblemente debe ser una unidad con múltiples procesadores.
- Puede ser plataforma Linux o Windows (a partir de XP); no se cuenta con soporte para OSX.
- Opcionalmente se puede contar con Visual Studio 2005 o 2008 para trabajar con Cilk.

Limitantes

La principal limitante de Cilk es el fuerte enfoque en problemas del tipo **divide y vencerás.** La ventaja de este problema es que el paradigma es suficientemente general para que muchos de los problemas con programación paralela puedan ser resueltos con este enfoque.

Otra limitante es que, a pesar de ser altamente portable, Cilk requiere arquitectura Intel para funcionar. Este problema tampoco es demasiado grande si se considera que el mercado sigue dominado por Intel. A pesar de ello, un proyecto de desarrollo serio no puede limitar a sus clientes por el procesador que utilicen, pero aplicaciones específicas se pueden ver beneficiadas por la plataforma.

Fuera de eso Cilk es robusta y le da muchos beneficios a los desarrolladores.

Código

Cilk es interesante debido a su simpleza y flexibilidad. Un ejemplo, con **una de las primeras versiones de la extensión,** lo demuestra.

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
        int x, y;

        x = spawn fib (n-1);
        y = spawn fib (n-2);

        sync;

        return (x+y);
    }
}
```

Código 2 Ejemplo de una de las primeras versiones de Cilk

El Código 2 ilustra un programa en C que contiene algunas palabras adicionales que son la extensión que Cilk provee al lenguaje de programación. Con esas adiciones es suficiente para crear una versión paralela del algoritmo recursivo para obtener un número de Fibonacci. Afortunadamente, el concepto minimalista permanece en la versión actual.

A partir de ahora se escribirá código perteneciente a la versión que se utilizará a lo largo del trabajo. A continuación se abarcan las palabras reservadas de Cilk.

- **cilk_main** Es similar a cualquier función main por lo que puede recibir argumentos. Esta función indica al compilador que Cilk será utilizado en el programa y que el código contendrá directivas que serán traducidas antes de ser compiladas. Con esta función se sientan las bases para trabajar en paralelo; inicializa un conjunto de hilos de trabajo.

```
int cilk_main ();
int cilk_main (int argc, char *argv[]);
int cilk_main (int argc, char *argv[], char *env[]);
```

- **cilk_spawn** es una instrucción que le dice a Cilk que la función que esta precede **puede ser ejecutada en paralelo**, ya que al final el sistema en tiempo de ejecución es el que decide el flujo del programa.

```
var = cilk_spawn func(args);
cilk_spawn func(args);
```

Como cualquier función de C/C++ es posible regresar un valor y el uso de parámetros dentro de la función es válido.

- **cilk_for** es un remplazo a la estructura de control clásica, cilk_for permite que las iteraciones del ciclo se ejecuten en paralelo. El compilador de Cilk++ convierte la instrucción en su equivalente recursivo, generando de esta forma un algoritmo del tipo divide y vencerás**:**

```
cilk_for (int i = begin; i < end; i += 2)
      f(i);
```

Esta instrucción es sumamente útil ya que es muy común el uso de ciclos dentro de un algoritmo.

- **cilk_sync** directiva que no permite que la función en la que se encuentra se ejecute en paralelo respecto a sus procesos hijos, por lo que debe esperar hasta que terminen. De esta forma se pueden evitar condiciones de carrera. Una vez que los hijos completan sus tareas, la función actual puede continuar.

```
cilk_sync;
```

cilk_sync sólo sincroniza la función padre con los procesos que derivaron de ella. Otras funciones y sus ramificaciones no se ven afectados.

Modelo de ejecución

Grafos dirigidos acíclicos

Para explicar el modelo de ejecución de Cilk es necesario recurrir a los grafos acíclicos dirigidos DAG (por sus siglas en inglés *directed acyclic*). Un DAG, como cualquier grafo, está compuesto por aristas y nodos. Las aristas representan instrucciones en ejecución y los nodos puntos a partir de los cuales se generan nuevos flujos de ejecución o donde se sincronizan y unen.

Un nodo a partir del cual se inician procesos en paralelo cuenta siempre con un flujo de entrada y dos de salida. Por otro lado, están los nodos de sincronización. Éstos tienen por lo menos dos entradas y no más de una salida.

El diagrama que se muestra a continuación representa un DAG para un programa sencillo.

```
funcion1(); // ejecuta arista 1
cilk_spawn funcion3();//genera arista 3
funcion2(); //ejecuta arista 2
cilk_sync; //unión en nodo B
funcion4(); //ejecuta arista 4
```

Código 3 Funciones ejecutadas en paralelo con Cilk.

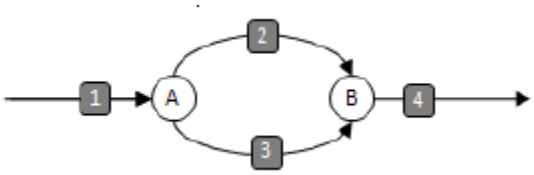


Figura 5 Grafo acíclico dirigido (DAG), representa al Código 3.

Las instrucciones definen el flujo de la gráfica. Cabe aclarar que un diagrama DAG no depende del número de procesadores de la máquina, más bien de las instrucciones dentro del código y del sistema en tiempo de ejecución. A partir de un DAG se pueden determinar algunos datos, como lo son el **Trabajo (W) y Profundidad (D)** y por lo tanto, **Paralelismo (P)**.

En teoría, al hablar de paralelismo se espera que las distintas instrucciones se ejecuten simultáneamente. En cambio, con Cilk las instrucciones a llevarse a cabo en paralelo (indicadas con cilk_spawn o cilk_for) **pueden** ejecutarse en paralelo pero **no es un requisito**. El planificador de tareas es el encargado de definir quién ejecutará las tareas. El resultado es el más eficiente ya que el sistema en tiempo de ejecución busca que el programa se vea totalmente beneficiado por Cilk.

Cilk utiliza el concepto de trabajadores. Un trabajador es un hilo de ejecución. Cuando sólo hay un trabajador, este se encarga de llevar a cabo todas las instrucciones. Si hay más de uno, se puede completar la ejecución de dos formas distintas.

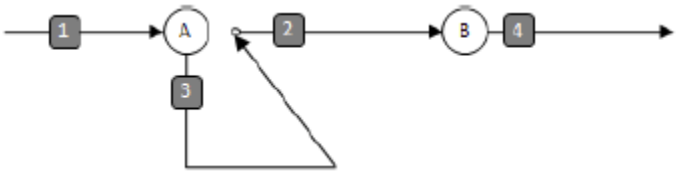


Figura 6 Flujo de ejecución para un solo procesador.

El planificador decide si las instrucciones 2 y 3 son completadas por distintos trabajadores. Cuando dentro de una función un trabajador encuentra la instrucción cilk_spawn, él mismo se vuelve el encargado de ejecutar la función "hija". En este caso las instrucciones 1 y 3 son ejecutadas por el mismo trabajador. Si hay un trabajador adicional disponible la instrucción 2 puede ejecutarse por el segundo trabajador. A esta acción se le llama "robar" tareas: la continuación del trabajo del primer hilo (si este hubiera sido el único trabajador) es "robado" por el segundo.

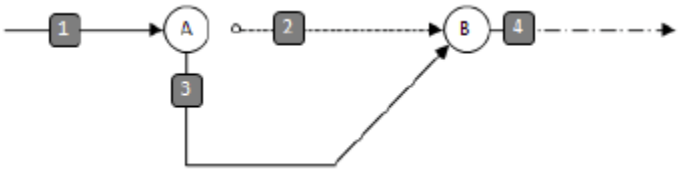


Figura 7 Posible flujo de ejecución para 2 o más procesadores ("robo de trabajo").

Se puede apreciar que el modelo de ejecución es bastante simple, especialmente porque de los detalles complejos se encarga el mismo Cilk.

Lo anterior se puede resumir con los siguientes puntos:

1. Después de cilk_spawn, la instrucción es ejecutada por el mismo trabajador que está ejecutando la función padre.
2. El resto de la función padre puede o no ser ejecutada (robo de tareas) por un trabajador distinto.
3. Después de cilk_sync, la ejecución puede proceder con cualquiera de los trabajadores que llevaron a cabo las instrucciones previas dentro de la función.

Manejo de problemas ligados al paralelismo con Cilk

En general, la mejor forma de evitar las condiciones de carrera es a través de un algoritmo bien estructurado. Sin embargo, en caso de no poder hacer mayores cambios, Cilk provee ayuda a través de **mutex y reductores**.

Mutex es una palabra construida a partir del concepto exclusión mutua (**mut**ual **ex**clusión).

```
cilk::mutex mut;
int sum = 0;
cilk_for (int i=0; i<10; ++i)
{
      mut.lock();
      sum = sum + i;
      mut.unlock();
}
```

Código 4 Ejemplo del uso de bloqueos para evitar carreras de datos.

La sección que se encuentra entre las instrucciones mut.lock y mut.unlock es la sección crítica. Pero gracias a que cuenta con la "protección" del mecanismo de exclusión mutua no se genera una **carrera de datos.** Sin embargo, los bloqueos no eliminan la posibilidad de una carrera determinista (es decir, el orden en el que los procesos alteren el resultado sí importa si la operación u operaciones no son conmutativas). En este caso en particular, como la suma es una operación conmutativa, el orden es irrelevante. Debido a este problema Cilk presenta una solución considerablemente más eficiente: los **reductores.**

Un **reductor** es una variable o estructura de datos compartida por varios procesos que es administrada a través del sistema en tiempo de ejecución. Lo que éste hace es generar copias de la variable para que cada uno de los hilos pueda hacer las modificaciones dentro de su propio ámbito, y el sistema en tiempo de ejecución se encarga de actualizar todas las copias. De esta forma no es necesario realizar bloqueos. Así se evita reducir la eficiencia del paralelismo, y simultáneamente resuelve el problema de las carreras de datos [15].

```
cilk::reducer_opadd<unsigned int> total;
cilk_for(unsigned int i = 1; i <= n; ++i)
{
      total += compute(i);
}
printf("Total %d",total.get_value());
```

Código 5 Ejemplo del uso de reductores.

Como se puede ver, su uso es bastante simple. Sencillamente se debe declarar la variable como reductor. Se puede escribir sobre el reductor normalmente, pero cuando es necesario leer la información que contiene se debe hacer a través del método "get_value()". Esta solución es sumamente elegante y útil para el desarrollo en paralelo.

Construir y ejecutar un programa con Cilk

Al instalar el paquete para desarrolladores, Cilk agrega algunos comandos para poder compilar, ejecutar y probar los programas creados. A continuación se revisan las instrucciones más comunes.

cilkpp sirve para compilar el código fuente y crear un ejecutable. Este código debe estar escrito con C++ con las palabras necesarias de Cilk. El archivo debe tener una terminación **.cilk**.

Una vez que se generó el archivo **.exe** se puede ejecutar normalmente:

```
>holamundo.exe
```

Con Cilk es posible asignar el número de trabajadores que se desea asignarle al programa a través del parámetro **-cilk_set_worker_count**.

```
>holamundo.exe -cilk_set_worker_count=N
```

Donde N representa un número entre 1 y M (los procesadores que posee la máquina en la que se está trabajando).

Cilk también puede buscar condiciones de carrera (ver conceptos básicos del paralelismo) con la instrucción **cilkscreen**:

```
>cilkscreen holamundo.exe
0.078 seconds
No errors found by Cilkscreen
```

Un análisis extremadamente útil que puede realizarse con Cilk es un análisis de escalabilidad. Este análisis se puede hacer desde línea de comandos con la instrucción **cilkview**.

```
>cilkview -trials all N holamundo.exe
```

Donde N, de nuevo, representa un número entre 1 y M. Cilk realiza la ejecución del programa con la cantidad de núcleos deseados por el usuario (N) y a partir de la información realiza estimados acerca de las posibilidades de paralelización del programa.

Todos los comandos que se han mencionado hasta ahora se integran automáticamente a Visual Studio 2008 o 2005. El uso a través de Visual Studio no cambia demasiado, simplemente se cuenta con una interfaz gráfica.

En teoría el manejo de esta plataforma es bastante sencillo y el análisis que se realiza en el trabajo busca extender la teoría y corroborarla. Además cobra importancia si se considera lo complicado que puede ser llegar a programar una aplicación que aprovecha las ventajas de muchos núcleos. Por ello, vale la pena entender Cilk y tomarla en cuenta a la hora de solucionar algún problema.

CUDA (Compute Unified Device Architecture)

Hace poco tiempo el cómputo paralelo era visto como una solución exótica; una especialización dentro de la ciencia de la computación con aplicaciones limitadas. Esta percepción ha cambiado profundamente en los últimos años con el surgimiento de arquitecturas híbridas [19]. La necesidad creciente de aplicaciones paralelas eficientes hace que los desarrolladores busquen herramientas que les faciliten la vida. Tal vez, CUDA pueda remediar o, por lo menos, simplificar este problema.

Es una plataforma para operar y programar en paralelo creada por NVIDIA Corporation. Enfocada en paralelismo en datos, permite el incremento en desempeño computacional a través del potencial de los procesadores gráficos (GPU). La plataforma fue desarrollada en C para ser utilizada sobre este lenguaje, pero cuenta con una variedad de bibliotecas no oficiales para varios lenguajes de programación (Ruby, Java, Python, entre otros).

Historia del procesamiento gráfico y CUDA

A comparación del procesamiento con los CPU tradicionales, realizar cálculos en un procesador gráfico es un concepto novedoso. Principalmente, el concepto de un GPU trabajando por si solo es innovador.

La situación del procesamiento de gráficos evolucionó radicalmente en los años 80 y 90. El crecimiento en popularidad de sistemas operativos gráficos ayudó a crear un mercado para un nuevo tipo de procesador. A principios de los 90, los usuarios empezaron a comprar aceleradores de pantallas de dos dimensiones (2D) para computadoras personales. Estos aceleradores ofrecían operaciones sobre mapas de bits frecuentemente asistidos por hardware para ayudar en el despliegue y usabilidad de sistemas operativos gráficos.

Para mediados de la década de los 90 la demanda de aplicaciones que implementaban el uso de gráficos en 3 dimensiones había escalado rápidamente, preparando el escenario para dos desarrollos novedosos. Sumado a esto, el lanzamiento de juegos detallados en 3D creó el objetivo de mejorar las gráficas lo más posible para hacerlos más y más realistas.

Desde el punto de vista del cómputo paralelo, el lanzamiento de la línea de tarjetas GeForce 3 en el 2001 representó un cambio significativo en tecnología GPU. Fue el primer chip en implementar el estándar DirectX 8.0 de Microsoft, el cual requería que se pudieran cumplir con tareas lógicas de programación y el manejo de sombreado gráfico simultáneamente. Por primera vez, los desarrolladores tuvieron algo de control sobre las operaciones realizadas en el GPU.

Esta inflexión en la tecnología atrajo a muchos científicos e investigadores por la posibilidad de usar hardware gráfico para fines distintos. El enfoque general en un principio era poco claro, ya que las interfaces de programación estándar (OpenGL y DirectX) eran la única forma de interactuar con el GPU. Cualquier intento de llevar a cabo operaciones arbitrarias eran frenadas por el sistema. Debido a esto los investigadores buscaron soluciones que

funcionaran a través de los gráficos; enmascaraban los problemas para que parecieran renderizados tradicionales.

Este modelo era bastante limitado, ya que no había libertad de instrucciones y memoria. Adicionalmente no había un soporte adecuado de operaciones con punto flotante. Las herramientas para desarrollo eran limitadas y el rastreo y corrección de errores sumamente difícil. Por último, para utilizar el poder de los procesadores gráficos era necesario aprender alguna biblioteca estándar. Pero estos problemas no duraron demasiado, ya que surgieron soluciones para estos problemas; las interfaces de programación (API) de propósito general que aprovechan las capacidades de los procesadores gráficos. Una de las interfaces es CUDA.

La información obtenida sobre la historia de los procesadores gráficos se obtuvo principalmente de [20].

Características

- Extensiones mínimas del lenguaje C.
    - Escribir programa secuencial.
    - Instanciar para que se ejecute en paralelo.
    - Modelo y lenguaje de programación familiar.
- CUDA es un modelo de programación paralela escalable.
    - Puede ejecutarse en cualquier número de procesadores sin recompilarse.
    - Cuenta con un sistema en tiempo de ejecución.
- El paralelismo de CUDA requiere de ambas CPU (Anfitrión) y GPU (Dispositivo).

Requerimientos

La infraestructura necesaria para hacer uso de CUDA no es demasiado cara o sofisticada. La pieza primordial: una tarjeta gráfica. Sin embargo, es posible emular y codificar con ayuda del kit de desarrollo.

Hardware

- En cualquier PC con NVIDIA GeForce 8,9 o 200 o compilar en modo Emulador.

Software

- Windows, Mac o Linux
- Compilador C
- Drivers de NVIDIA
- CUDA SDK

El conjunto de herramientas de CUDA incluye un compilador para C y soporte para bibliotecas gráficas como OpenGL y DirectX 11.

Utiliza la arquitectura GPU de NVIDIA para acelerar las aplicaciones. El hardware y el software fueron diseñados para trabajar juntos, y de esta manera exponer el poder computacional de

NVIDIA y facilitar el cómputo con procesadores gráficos en general. Vale la pena mencionar que CUDA trabaja con paralelismo en datos.

Limitantes

Estas limitantes no son subjetivas; de hecho son elementos documentados. Por eso son mencionados en esta sección antes de cualquier análisis.

CUDA es una plataforma sólida, pero cuenta con algunas restricciones que hay que tomar en cuenta al decidir si es la herramienta más adecuada para el proyecto que se esté trabajando.

NVIDIA creó la herramienta y busca que se vuelva un estándar en computación a través de GPU para sacar provecho de una necesidad creada. A pesar de que las tarjetas gráficas son comunes hoy en día, no todos cuentan con una tarjeta NVIDIA o de otras marcas que puedan ser aprovechadas para este tipo de desarrollo.

CUDA no utiliza recursión hasta la versión 3 del kit de desarrollo, debido a la forma en la que ha sido estructurado hasta ahora los apuntadores a funciones no son compatibles con las operaciones en tarjeta gráfica [21].

El flujo de ejecución de la plataforma genera algunos cuellos de botella de los cuales se habla en la siguiente sección. Adicionalmente hay que tener en cuenta que el tamaño de la memoria de gráficos (que es imprescindible para CUDA) no es tan grande como la memoria principal, por lo que hay que tener un muy buen manejo de la información.

Ninguna herramienta puede cubrir todas las necesidades para resolver un problema. Por ello es necesario analizar cuál es la más óptima en un contexto específico.

Conceptos básicos

La estructura de trabajo de CUDA se basa en la combinación adecuada de tareas entre GPU y CPU. El GPU se encargará de realizar operaciones pesadas aritméticamente hablando, ya que cuenta con muchos núcleos capaces de realizar este tipo de operaciones rápida y eficientemente. Por otro lado, el CPU mantiene la dirección y flujo del programa, indicándole a la tarjeta gráfica qué operaciones son necesarias. El intercambio de información entre el dispositivo y la máquina anfitrión ocurre a través de la memoria principal y la memoria gráfica.

Se explicó previamente qué parte del código se ejecuta en la tarjeta gráfica. Este “código paralelo” corre sobre muchos hilos del GPU; y los hilos a su vez se agrupan en bloques.

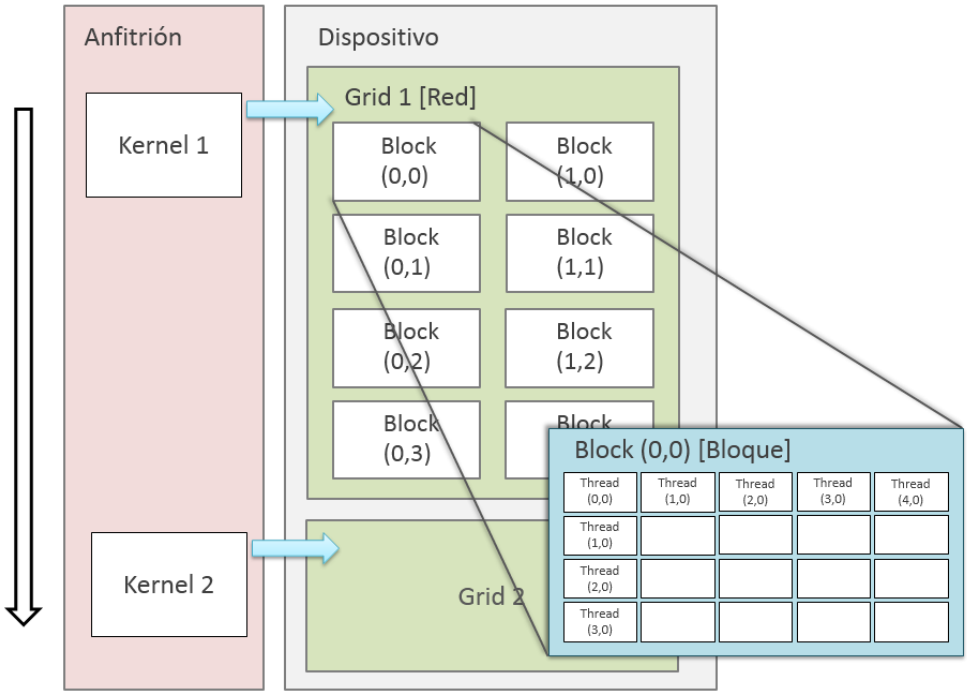


Figura 8 Diagrama de la arquitectura lógica de un procesador gráfico.

Red: Es un conjunto de bloques asignados para una ejecución en particular.

Bloque: Conjunto de hilos

- Pueden sincronizar su ejecución.
- Comparten memoria que pueden usar para comunicarse.

Hilos

- Cada hilo es libre de seguir flujos independientes.
- Cada hilo cuenta con variables identificadoras de hilo y bloque.

Kernel

El kernel en este caso es una función que es implementada por la tarjeta gráfica. Únicamente se puede ejecutar una función kernel a la vez (a menos que se cuente con más de una tarjeta gráfica). No todas las funciones que hacen uso del GPU deben llamarse kernel, pero sirven para ilustrar los ejemplos.

Sistema en tiempo de ejecución

Parecido a lo visto con Cilk, CUDA cuenta con un sistema en tiempo de ejecución. Esta herramienta se encarga de algunas tareas de bajo nivel que le ahorran algo de trabajo al usuario:

- Ejecuta código en la tarjeta gráfica.
- Manipula los recursos del programa en ejecución.
- Divide la información para que no exceda los límites de un núcleo.
- Planifica la ejecución de los bloques del dispositivo.

Memoria

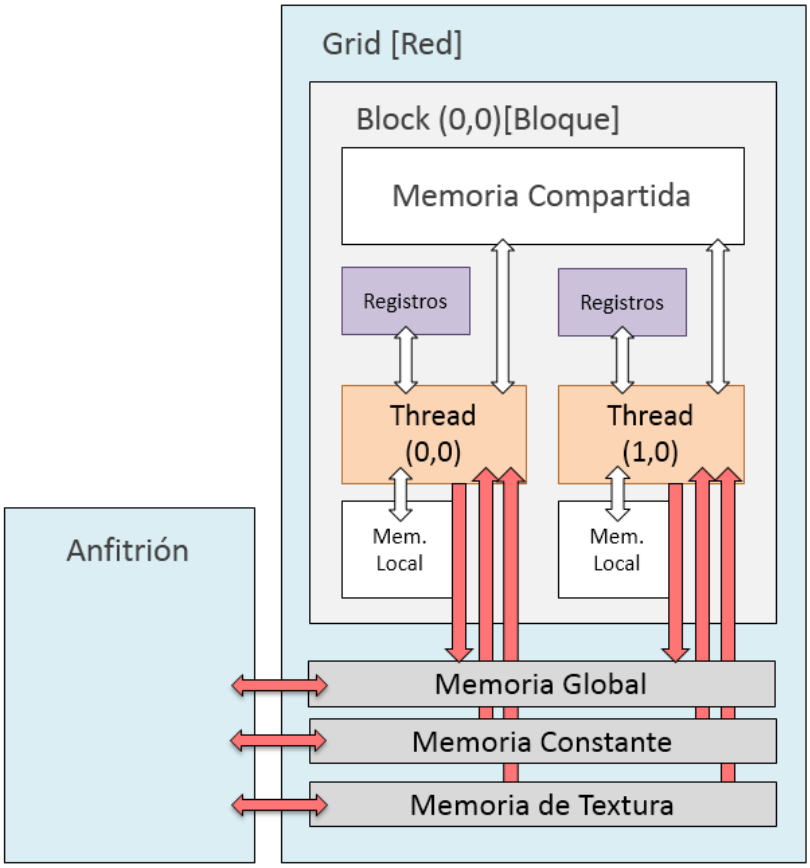


Figura 9 Distribución de la memoria en un dispositivo de procesamiento gráfico.

Hay varios tipos de memoria a la que tienen acceso los hilos de ejecución.

Registros, son la memoria que responde más rápido a las peticiones. Es pequeña y simple, principalmente utilizada para los datos que están siendo procesados inmediatamente.

Memoria compartida, un peldaño debajo de los registros en términos de velocidad. A través de ésta se puede compartir información entre los flujos de ejecución.

Memoria de dispositivo, esta memoria incluye a la memoria local, global, constante y de textura. Estas memorias no pueden ser operadas tan rápidamente, pero cuentan con una mayor capacidad y se pueden comunicar con la máquina anfitrión.

Modelo de ejecución de CUDA

El modelo que utiliza CUDA es bastante simple. Como ya se ha mencionado, la tarjeta gráfica se encarga de operar en paralelo grandes cantidades de información. Para ello hay que hacer los datos disponibles en memoria gráfica. Una vez que se tiene acceso a la información la tarjeta opera con base en instrucciones enviadas por el CPU.

Para programar en CUDA hay que seguir 4 pasos elementales:

1. **Crear datos en memoria principal:** Toda la información ya sea trabajada por CPU o GPU primero es creada por la función principal.
2. **Copiar los datos a memoria gráfica:** La información que vaya a ser trabajada por CUDA debe moverse a la memoria de gráficos. Es necesario considerar los límites de esta memoria ya que difícilmente tendrá la capacidad de la memoria principal.
3. **Operar los datos en GPU**: Todas las operaciones necesarias se llevan a cabo en GPU; y ya que este dispositivo está optimizado para realizar operaciones matemáticas sencillas en grandes cantidades, debe favorecer el programa si es que está bien estructurado.
4. **Copiar los datos a memoria principal**: Una vez que se terminó de procesar la información es necesario regresarla para realizar, concluir o alistar los datos para más procesamiento.

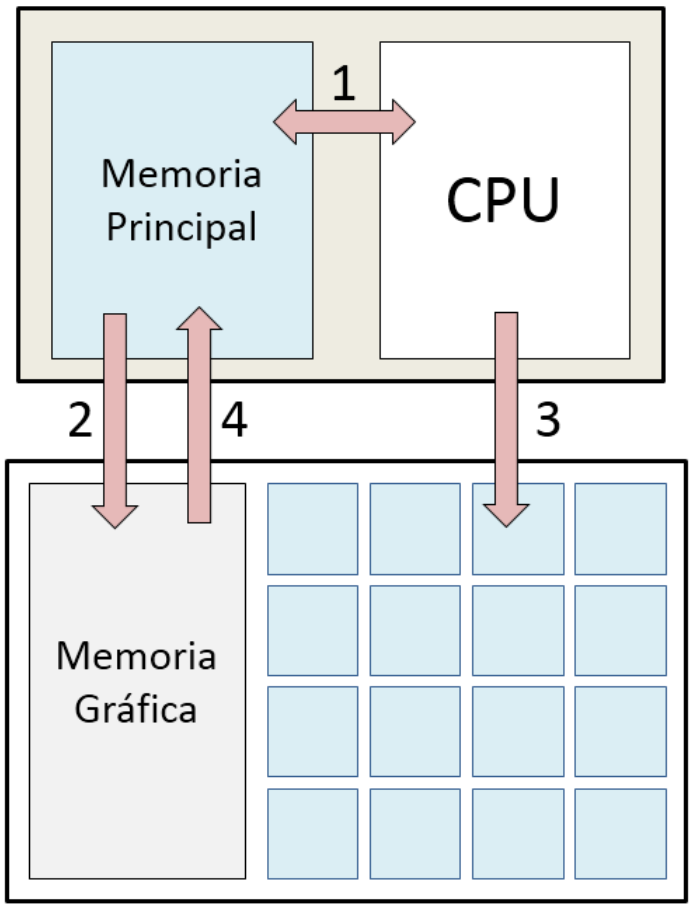


Figura 10 Flujo de ejecución CUDA.

En la actualidad las computadoras y tarjetas gráficas no tienen problemas al realizar una gran cantidad de operaciones. La realidad es que el hardware actual es extremadamente potente. Sin embargo, al observar el diagrama se puede apreciar que hay un punto débil con este paradigma en particular. Hay un cuello de botella a la hora de mover datos hacia y desde memoria gráfica, por lo que se está hablando de dos puntos críticos dentro del flujo de ejecución. Es un problema similar a lo que se encuentra en la arquitectura Von Neumann. El modelo no ha cambiado en los últimos años, aunque sí cuenta con mejoras en el desempeño debido al incremento en velocidad de procesamiento.

Flujo de compilación

CUDA trabaja principalmente como extensión de C. Si se utiliza con otros lenguajes de programación funciona a través de bibliotecas de código basadas en CUDA C. A continuación se presenta el flujo:

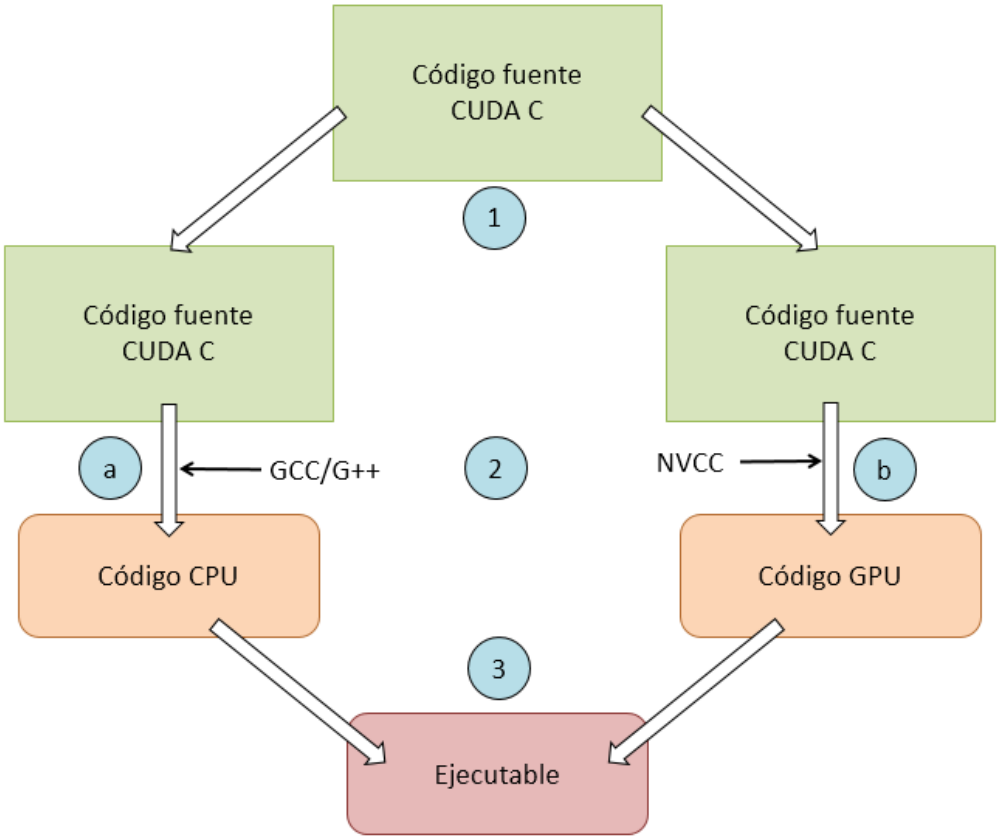


Figura 11 Flujo de compilación de CUDA.

1. El código fuente se divide en código para la máquina anfitrión y código para el dispositivo gráfico.
2. Compilación:
    a. El código del anfitrión es código C común y corriente; éste es compilado por gcc o vscc (compilador de Visual Studio para C).
    b. El código del dispositivo es compilado por nvcc, que es un compilador incluido con el kit de desarrollo.
3. Después los resultados de cada una de las compilaciones se vuelven a compilar en un solo ejecutable con gcc o vscc.

El flujo es interesante y bastante innovador. Además, no interfiere fuertemente con las versiones estándar de C/C++.

Código

Para utilizar CUDA es necesario aprender algunas palabras reservadas adicionales, y como se ha mencionado previamente, CUDA funciona como extensión de C.

Se empezará con un sencillo “hola mundo” que introduce algunas nuevas funciones.

```
#include <stdio.h>

__global__ void kernel (void){
}

int main (void){
      kernel <<<1,1>>>();
      printf("Hola mundo!\n");
      return 0;
}
```

Código 6 Hola mundo con CUDA.

Las anotaciones resaltadas con color son las extensiones de CUDA para este programa.

El calificador **__global__** alerta al compilador que la función puede ser ejecutada en el dispositivo gráfico. El compilador de CUDA **nvcc** le da la función kernel al dispositivo gráfico mientras que al compilador regular le da la función principal (main).

De manera similar, la función **kernel** simplemente se refiere al código que será ejecutado por la tarjeta gráfica. Ese es el secreto detrás de CUDA: la capacidad de hacer llamadas al dispositivo desde la máquina anfitrión. Los números dentro de los paréntesis triangulares no son argumentos recibidos por la función kernel. Más bien son utilizados por el sistema en tiempo de ejecución.

El siguiente ejemplo realiza una suma con la ayuda del GPU:

```
#include <stdio.h>

__global__ void add (int a, int b, int *c){
      *c = a + b;
}

int main (void){
      int c, *dev_c;
      cudaMalloc((void**)&dev_c, sizeof(int);
      add <<<1,1>>>(2,7,dev_c);
      cudaMemcpy(&c,dev_c,sizeof(int),cudaMemcpyDeviceToHost);
      printf("2+7=%d\n",c);
      cudaFree(dev_c);
      return 0;
}
```

Código 7 Suma de vectores.

En este segundo ejemplo hay tres nuevas funciones que son necesarias para trabajar con la tarjeta gráfica. Éstas se encargan de reservar espacio en memoria que será utilizado por el dispositivo y, una vez que este termina, otra función libera la memoria.

cudaMalloc, es similar a la función de C malloc (memory allocation) que se encarga de reservar el espacio en memoria. En el caso de CUDA es en la memoria de gráficos. El primer parámetro es un apuntador a la dirección donde se almacenarán los datos. El segundo es el tamaño de memoria que será reservado.

Un recordatorio importante es que con el apuntador que se creó para memoria gráfica no es posible leer o escribir dentro de la función principal. Fuera de esa restricción se puede hacer con el apuntador cualquier otra operación (aritmética de apuntadores).

cudaFree, es una función simple; libera la memoria gráfica. Vale la pena mencionar que la función de C "free" no funciona para liberar espacio de memoria gráfica. Por ello es necesario usar cudaFree, pero el funcionamiento es totalmente equivalente.

cudaMemcpy, debido a la restricción mencionada previamente respecto a los apuntadores en distintos ámbitos, es necesario copiar la información de la memoria gráfica a la memoria principal. De esta manera se podrá trabajar con el resultado final. Los parámetros de la función son:

1. Los apuntadores con los que se intercambiará la información.
2. El tamaño de los datos que están siendo copiados.
3. El sentido en el que se están moviendo los datos, en este caso del dispositivo a la máquina anfitrión (**cudaMemcpyDeviceToHost**).

En caso de querer copiar información del anfitrión al dispositivo gráfico se utiliza **cudaMemcpyHostToDevice**. De la misma manera, para copiar memoria dentro del dispositivo, se hace uso de **cudaMemcpyDeviceToDevice.** Por último, si se quisiera hacer una copia entre apuntadores dentro de la memoria principal, simplemente se utiliza la función estándar de C memcpy.

Con este ejemplo se cubren las funciones básicas para el manejo de CUDA. Pero no se ha hablado de lo que es necesario para trabajar en paralelo. A continuación se presenta un ejemplo para demostrar el funcionamiento:

```
#define N 10

void add( int *a, int *b, int *c ) {
    int tid = 0;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;
    }
}

int main( void ) {
    int a[N], b[N], c[N];

    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
    add( a, b, c );

    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
    return 0;
}
```

Código 8 Algoritmo secuencial.

Este programa se encarga de sumar todos los elementos de un arreglo con sus equivalentes de un segundo arreglo, es decir, A[1]+B[1], A[2]+B[2],…,A[N]+B[N]. A esta operación se le conoce como suma de vectores. En C el algoritmo es sencillo, pero la intención es realizarlo en paralelo.

```
#define N   10

__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;
     if (tid < N)
          c[tid] = a[tid] + b[tid];
}
```

Código 9 Algoritmo paralelo.

```
int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    cudaMalloc((void**)&dev_a,N*sizeof(int));
    cudaMalloc((void**)&dev_b,N*sizeof(int));
    cudaMalloc((void**)&dev_c,N*sizeof(int));

    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    cudaMemcpy(dev_a,a,N*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b,b,N*sizeof(int),cudaMemcpyHostToDevice);

    add<<<N,1>>>( dev_a, dev_b, dev_c );

    cudaMemcpy(c,dev_c,N*sizeof(int),cudaMemcpyDeviceToHost);

    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Código 10 Suma vectores en paralelo.

Es posible observar funciones básicas que ya fueron mencionadas. Aun así, destacan algunas líneas diferentes.

Resalta el tamaño del arreglo, que es definido por una constante (N). Es posible acceder a la constante desde el dispositivo o la memoria principal.

La función __global__ add recibe tres parámetros: los dos vectores de entrada y el vector de salida. De hecho, si se simplifica un poco el problema, este código hace casi lo mismo que el ejemplo anterior: sumar dos números.

La gran diferencia es que ahora, entre los paréntesis triangulares, en lugar de escribir un valor (1) se envía una constante (N) que es fácilmente modificable. Esta **N le dice al sistema en tiempo de ejecución el número de bloques en paralelo que se ejecutarán.** En este caso particular, 10.

La función add cuenta con una línea: **tid = blockIdx.x.**

Ésta representa el identificador del bloque que está ejecutando la función. Es posible decir que la función se copia N veces y que cada copia es ejecutada por un bloque específico. De esta forma es posible realizar N operaciones en paralelo que aceleran el proceso considerablemente.

Con estos ejemplos se cubren las operaciones básicas de CUDA. Existen más instrucciones que, en caso de ser utilizadas a lo largo del trabajo, serán explicadas a detalle.

III Metodología y planteamiento

Es prácticamente imposible encontrar una herramienta para resolver cualquier problema, más bien se debe entender el contexto antes de decidir qué se utilizará para solucionar un problema.

El objetivo del trabajo, es identificar cuándo es mejor usar una u otra plataforma para solucionar un problema. No es realista pensar que una sea absolutamente mejor que la otra. Desde la descripción de sus características es posible darse cuenta que son muy diferentes. La parte práctica de la tesis busca exponer las cualidades y defectos de cada una.

Metodología

Primero se plantearán distintos problemas comunes dentro de las ciencias de la computación. Cada algoritmo será analizado y paralelizado. El análisis teórico simplemente representa lo que se espera de los resultados prácticos. En este capítulo se encuentran los distintos planteamientos.

Posteriormente, se obtendrán 10 muestras de cada implementación sobre cada uno de los 4 de procesadores. En el caso de Cilk, la cantidad de procesadores irá aumentando y los datos obtenidos buscarán afirmar la teoría. En el caso de CUDA no se puede llegar realizar un análisis con distintas unidades de procesamiento pero se evaluará la mejora en desempeño. Los programas recibirán una entrada promedio grande (entre 10^6 y 30^8). En caso de ser necesario se probará una entrada que sea particularmente perjudicial para el algoritmo (el peor caso). A partir de los datos obtenidos se podrá llevar a cabo el análisis cuantitativo.

El capítulo 4 muestra los resultados de las prácticas. Este análisis de datos contará con dos secciones, una sobre el análisis cuantitativo y otra sobre el cualitativo.

Cuantitativamente, se medirá el incremento en eficiencia del programa. Las distintas plataformas no pueden resolver los mismos problemas debido al ámbito de trabajo de cada una (paralelismo en tareas para Cilk y paralelismo en datos para CUDA). Por lo que el análisis de cada una es independiente. El resultado de este análisis será una tabla de datos que ilustrará el desempeño del programa en distintas circunstancias. El parámetro primordial de la práctica será el número de procesadores. El tamaño de la entrada será fijo para cada problema.

Cualitativamente, el análisis se centrará en las facilidades que presenta la herramienta para paralelizar un programa y qué tan sencillo es implementar la solución desde el punto de vista del desarrollador. Se consideran las complicaciones que hayan surgido durante el desarrollo o las facilidades que pueden ser aprovechadas con cada plataforma. No es tan simple asignar números al resultado de este análisis, pero se compartirá la experiencia de haber paralelizado el programa con una u otra herramienta.

Finalmente, se concluirán las condiciones en las que cada plataforma funciona mejor y en cuáles es inviable su uso.

Planteamiento teórico

A partir de ahora se describe el proceso de análisis y codificación de algoritmos. Estos serán escritos con pseudocódigo para que su representación sea la más general posible. Por otro lado, el código estará escrito en C/C++ estándar junto con las palabras reservadas pertenecientes a las plataformas de desarrollo. Cada práctica cuenta con una descripción del algoritmo, su análisis y la codificación secuencial y paralela.

Práctica 1: Búsquedas con Cilk

Buscar dentro de un conjunto de datos es una tarea recurrente en programación. De hecho, no es una labor demasiado complicada. Hay varios tipos de búsqueda, pero las más simples son la búsqueda lineal y la búsqueda binaria.

La **búsqueda lineal** revisa ordenadamente cada una de las casillas empezando por la primera y repite la operación hasta encontrar el valor deseado. En caso de llegar al final del arreglo y no haber encontrado el valor buscado, el resultado es un código de error.

```
busquedaLineal(A[n], d, l)
        for(i=0 ; i<l ; i++)
                if(A[i]==d)
                        return i;
        return -1;
```

Código 11 Algoritmo de la búsqueda lineal.

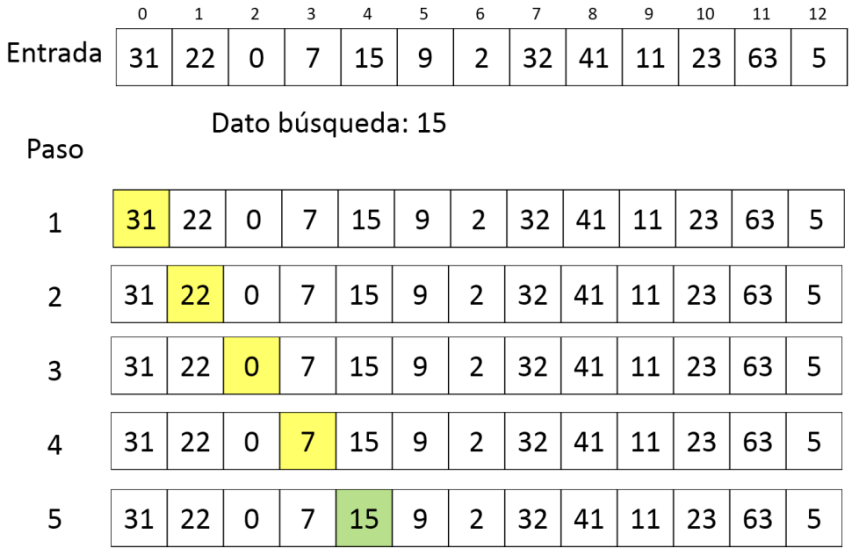


Figura 12 Visualización de una búsqueda lineal.

Es fácil inferir que en el **peor de los casos** la búsqueda lineal tiene **complejidad lineal $O(n)$,** ya que **debe recorrer todo el arreglo**. Por el contrario, en el mejor caso, únicamente necesita **una comparación**. Todavía es necesario calcular el caso promedio. Para empezar, se supone que la probabilidad de encontrar el dato en cualquier casilla es:

$$p_i = \frac{1}{n}$$

Donde i se refiere al índice del arreglo. Entonces se tiene una probabilidad equitativamente distribuida. Por lo tanto, la suma de las probabilidades debe dar uno.

$$p_1 + p_2 + p_3 + \cdots + p_n = 1$$

Con estos datos, se puede definir el desempeño del algoritmo de la siguiente forma:

$$T(n) = 1p_1 + 2\,p_2 + 3p_3 + \cdots + np_n$$

Donde los coeficientes representan el número de operaciones necesarias para llegar a la 'iesima' casilla; aumentan ya que cada iteración realiza una comparación más. Es posible factorizar las probabilidades porque todas tienen el mismo valor.

$$T(n) = (1 + 2 + 3 + \cdots + n)\frac{1}{n}$$

El primer término del producto es una sucesión aritmética[4], por lo que se puede simplificar de la siguiente forma.

$$T(n) = n\frac{(n+1)}{2}\frac{1}{n}$$

$$T(n) = \frac{(n+1)}{2}$$

Por lo tanto, **el caso promedio** de la búsqueda secuencial es:

$$\Theta\left(\frac{n+1}{2}\right)$$

Una vez analizada la búsqueda secuencial, es necesario escribir sobre **búsqueda binaria;** un algoritmo poderoso y sencillo de comprender. El requisito para que este algoritmo sea realmente eficiente es que el arreglo de datos se encuentre ordenado. Se inicia comparando el dato de la casilla que se encuentra a la mitad del arreglo (pivote) con el dato de búsqueda. En caso de que el dato no se encuentre ahí, se repetirá la acción, pero está vez con la mitad a la izquierda o a la derecha del pivote. Sí el dato es **menor se usa el lado izquierdo, en caso contrario, se busca en el lado derecho**. Se repetirá el proceso hasta que se encuentre el dato o se acaben las posibilidades, en cuyo caso se regresa un dato (normalmente negativo, o nulo)

[4] Ver apéndice, *sucesión aritmética*.

que representa una búsqueda no exitosa. **En caso de encontrarse el dato se regresa el índice de la casilla donde apareció.**

```
busquedaBinaria(A[n], d, ini, fin)
        m = ini+(fin-ini)/2
        if(m>0)
                if(A[m]==d)
                        return m;
                if(A[m]<d)
                        busquedaBinaria(A[n],d,ini,m-1);
                if(A[m]>d)
                        busquedaBinaria(A[n],d,m+1,fin);
        else
                return -1;
```

Código 12 Algoritmo de la búsqueda binaria recursiva.

```
int b_bin_recursiva(int A[], int inf,int sup,int d)
{
      int m = inf+(sup-inf)/2;
      if(inf<=sup)
      {
            if(A[m]==d)
                  return m;
            if(A[m]>d)
                  return b_bin_recursiva(A,inf,sup-1,d);
            if(A[m]<d)
                  return b_bin_recursiva(A,m+1,sup,d);
      }
      return -1;
}
```

Código 13 Implementación de la búsqueda binaria en C.

Este código únicamente se apega a los pasos del algoritmo recursivo.

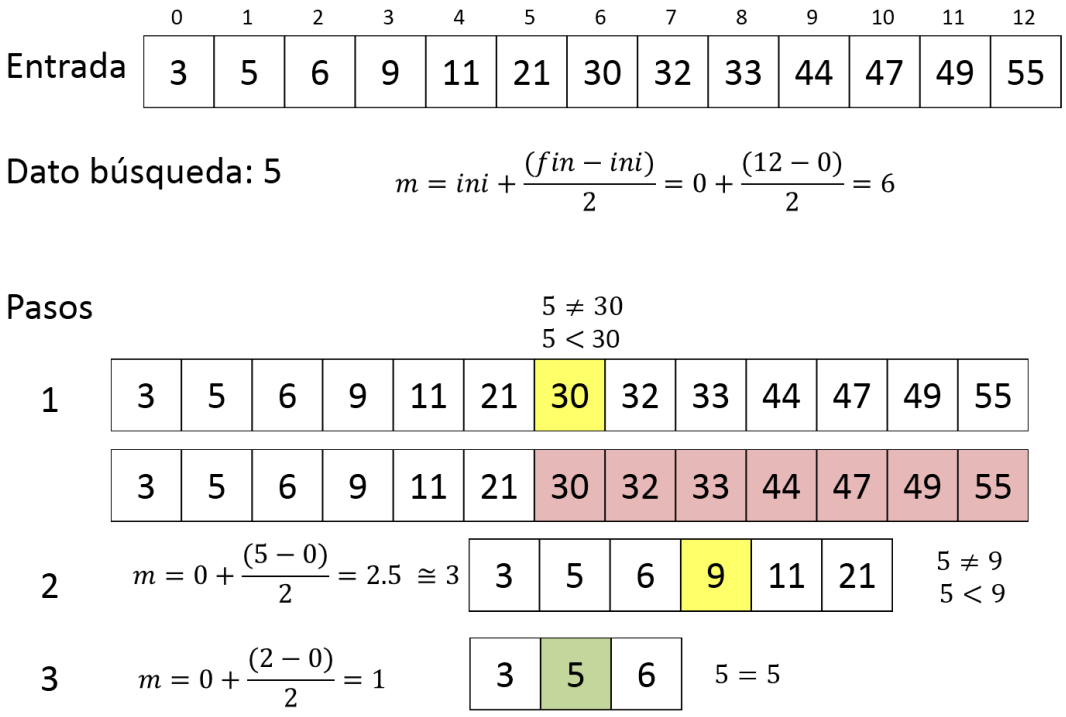


Figura 13 Visualización de un caso concreto de la búsqueda binara.

El ejemplo mostrado en la figura anterior es un caso particular de la búsqueda binaria para un arreglo con 13 elementos. El arreglo está ordenado, que es el único requisito para la búsqueda. El dato que se buscará dentro del arreglo es el número 5. Cada vez que se realiza la búsqueda es necesario calcular la casilla a la mitad del arreglo[5], ya que el dato que contenga es el que servirá como pivote. Si el dato se encuentra en la casilla 'm' la búsqueda queda resuelta. En el ejemplo, para el primer paso el pivote es 30; no es el dato de búsqueda. Dado que 5 es menor a 30 se repite la búsqueda; esta vez sólo en la primera mitad del arreglo. Repitiendo este procedimiento base, con únicamente tres pasos se llega al resultado. El número 5 se encuentra en el arreglo en la casilla 1. Aunque la entrada contaba con 13 números 3 comparaciones fueron suficientes.

Se observa que cada vez que se realiza una comparación, disminuye el número de elementos de búsqueda a la mitad, independientemente de la entrada del algoritmo. Esta acción reduce rápidamente el conjunto de búsqueda. Hacia el final, se puede decir que la búsqueda se vuelve más refinada, ya que no elimina tantos elementos.

Esta es la recurrencia con la que se representa la búsqueda binaria.

$$T(n) = T(n/2) + O(1)$$

[5] El cálculo de m se realiza de esa forma para evitar desbordamiento de la memoria Ver apéndice, *desbordamiento de memoria*.

- $T(n)$ - Representa el tiempo total que tomará a la búsqueda dado un arreglo de tamaño n.
- $T(n/2)$ - Significa que se vuelve a realizar exactamente la misma función con la mitad de los elementos.
- $O(1)$ - Representa las operaciones constantes que forman parte del algoritmo.

Esta recurrencia cumple los requisitos para resolverla a través del primer caso del método maestro del análisis de algoritmos. El resultado que se obtiene es justamente:

$$O(\lg n)$$ [6]

Para poner la eficiencia de la búsqueda binaria en perspectiva, si se busca dentro de un **arreglo de un millón de elementos** el algoritmo dará un **resultado en máximo 20 pasos**. Esto, es sumamente eficiente.

La única condición con la que cuenta este algoritmo de búsqueda es también su mayor debilidad: el arreglo debe estar ordenado. En cambio, la búsqueda lineal no cuenta con esa restricción.

A partir de la búsqueda binaria puede derivarse un algoritmo similar; diseñado para manejar cualquier entrada, ya sea ordenada o desordenada. La desventaja es que el cambio afecta considerablemente el desempeño del algoritmo.

```
busqueda(A[n], d, ini, fin)
        m = ini+(fin-ini)/2
        if(m>0)
                if(A[m]==d)
                        return m;
                busqueda(A[n],d,ini,m-1);
                busqueda(A[n],d,m+1,fin);
        else
                return -1;
```

Código 14 Algoritmo de búsqueda recursiva que tolera entradas en desorden.

[6] $\lg(n)$ equivale a $\log_2(n)$.

```
int busqueda_recursiva(int A[], int inf,int sup,int d)
{
	int m = inf+(sup-inf)/2;
	int a=-1,b=-1;
	if(inf<=sup)
	{
		if(A[m]==d)
			return m;
		else
		{
			a = busqueda_recursiva (A,inf,m-1,d);
			b = busqueda_recursiva (A,m+1,sup,d);
			if(a>=0)
				return a;
			else if(b>=0)
				return b;
		}
	}
	return -1;
}
```

Código 15 Implementación del algoritmo recursivo.

Incluso antes de resolver la recurrencia es posible apreciar que la entrada no se reduce con cada iteración. Es posible anticipar que se contará con complejidad lineal. Una vez más se utiliza el primer caso del método maestro para resolver la recurrencia.

$$T(n) = 2T(n/2) + O(1)$$

$$T(n) = O(n)$$

Pareciera que este algoritmo no tiene ningún tipo de ventaja. De hecho es similar a la búsqueda lineal, porque revisa todos los elementos en el peor caso, pero que realiza comparaciones en otro orden.

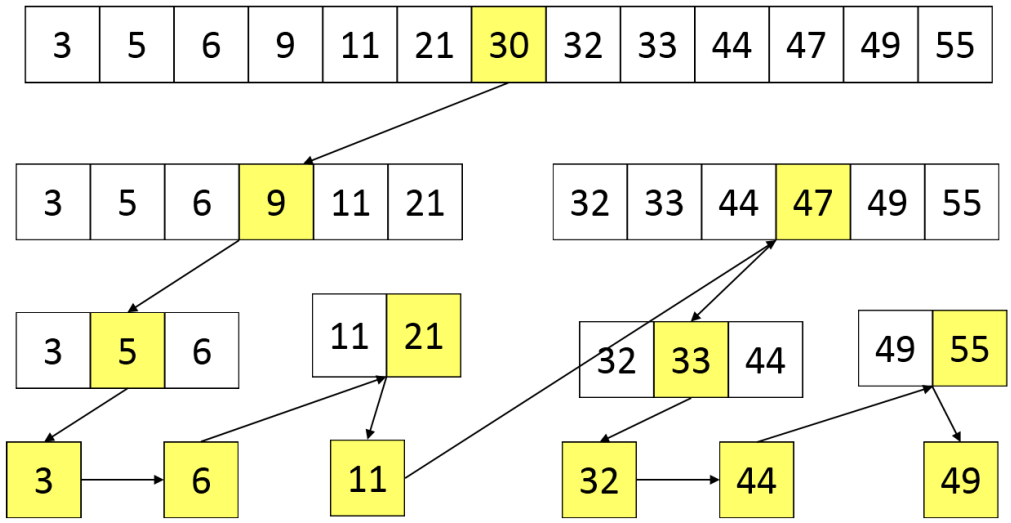


Figura 14 Representación visual del flujo de trabajo del algoritmo de búsqueda.

Sin embargo, esta variante del algoritmo es del tipo divide y vencerás. Debido a esto puede fácilmente adaptarse para aprovechar las capacidades de Cilk. Aquí es donde el procesamiento paralelo debe marcar la pauta, ya que más procesadores significan un mejor rendimiento.

Para implementar el algoritmo en paralelo simplemente es necesario generar un nuevo hilo de ejecución con cada llamada a la función "búsqueda". El análisis teórico para crecimiento en paralelo es importante definirlo. Para ello, se debe encontrar el trabajo, W(n), que equivale al desempeño, T(n), para cuando se cuenta con un solo procesador. Este caso es el mismo resultado del análisis secuencial.

$$W(n) = T(n) = O(n)$$

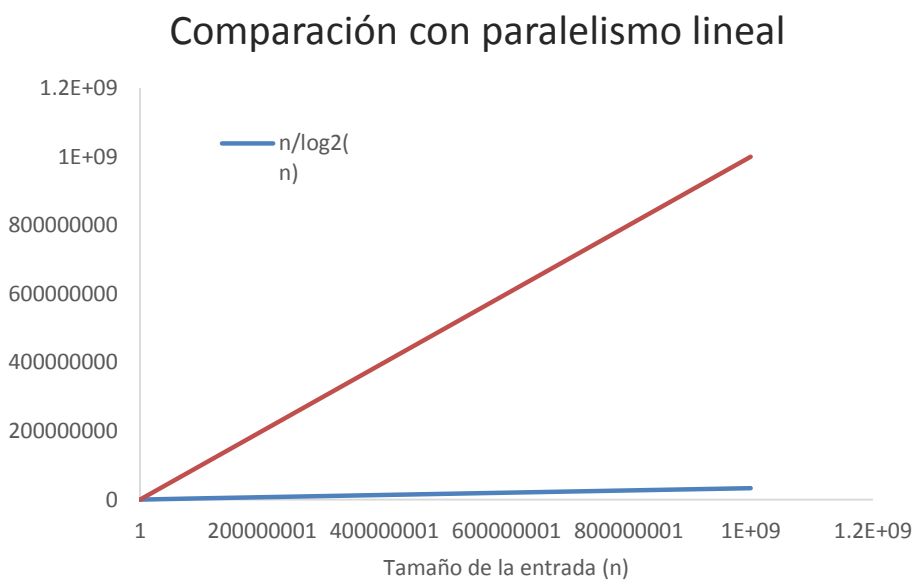
Por otro lado, es necesario calcular la profundidad D(n). La profundidad, es el desempeño teórico cuando se cuenta con una cantidad infinita de procesadores, que a su vez equivale a la cadena más grande de operaciones dependientes. Esto es posible observarlo en la Figura 14. Sin contar el primer nivel, donde el arreglo está completo cada vez que se divide el arreglo en 2 se agrega un nivel. Para el ejemplo, se tienen 3 niveles, este número está muy relacionado con el hecho de que el algoritmo se divide en 2 con cada iteración. Por lo tanto, se puede deducir que la profundidad es el logaritmo base dos de 'n'.

$$D(n) = \log_2 n$$

Con estos parámetros se define que el paralelismo para esta variante de la búsqueda binaria es:

$$P(n) = O\left(\frac{n}{\log_2 n}\right)$$

Al compararlo con un paralelismo lineal se observa lo siguiente.



Gráfica 4 Compara un crecimiento lineal con el paralelismo teórico.

En este caso el crecimiento no es el ideal. La desventaja en la función se debe a que la división reduce el crecimiento. A pesar de ello el paralelismo es creciente.

```
int b_bin_paralela(int A[], int inf,int sup,int d)
{
	int m = inf+(sup-inf)/2;
	int a=-1,b=-1;
	if(inf<=sup)
	{
		if(A[m]==d)
			return m;
		else
		{
			a = cilk_spawn b_bin_paralela(A,inf,m-1,d);
			b = cilk_spawn b_bin_paralela(A,m+1,sup,d);

			cilk_sync;
			if(a>=0)
				return a;
			else if(b>=0)
				return b;
		}
	}
	return -1;
}
```

Código 16 Implementación del algoritmo paralelo con C y Cilk.

El programa finalmente representa una búsqueda puede trabajar en paralelo. Cada búsqueda con una entrada menor es ejecutada por un nuevo hilo de ejecución. Una vez que concluye la

búsqueda parcial se revisa si se encontró el dato. El análisis mostrará los resultados empíricos.

Práctica 2: Quicksort con Cilk

***Quicksort* u ordenamiento rápido** es un algoritmo de ordenamiento sumamente eficiente por ello y otras ventajas que provee, es de los más populares en el desarrollo de software. Es una rutina del tipo divide y vencerás, por ende, es una buena prueba para Cilk. Consiste en utilizar un elemento del arreglo como pivote. Éste sirve para dividir o "partir" el arreglo en dos partes, no necesariamente iguales. Los valores en las primeras casillas (a la izquierda del pivote) son menores a p y los valores del lado derecho son mayores.

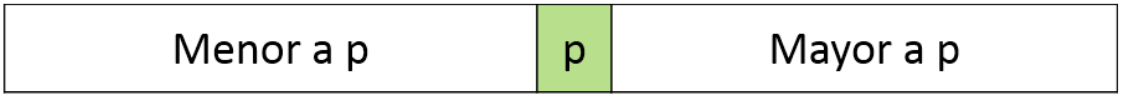


Figura 15 Mapa del arreglo de datos después de una partición.

Una vez que se terminó de dividir el arreglo se repetirá la operación, pero con cada una de las mitades. De esta forma se puede definir el siguiente algoritmo:

```
quicksort(A, ini, fin)
        p = particion(A,ini,fin);
        quicksort (A,ini,p-1);
        quicksort (A,p+1,fin);
```

Código 17 Algoritmo clásico de quicksort.

La pieza central del algoritmo es la partición.

```
particion(A, ini, fin)
        p = A[ini], índice = ini, j;
        for(j desde p+1 -> fin - 1)
                if(A[j] < pivote)
                        indice++;
                        intercambia(A[indice],A[j]);
        intercambia(A[p],A[indice]);
        return indice;
```

Código 18 Algoritmo para realizar la partición de un arreglo.

La partición funciona de la siguiente forma:

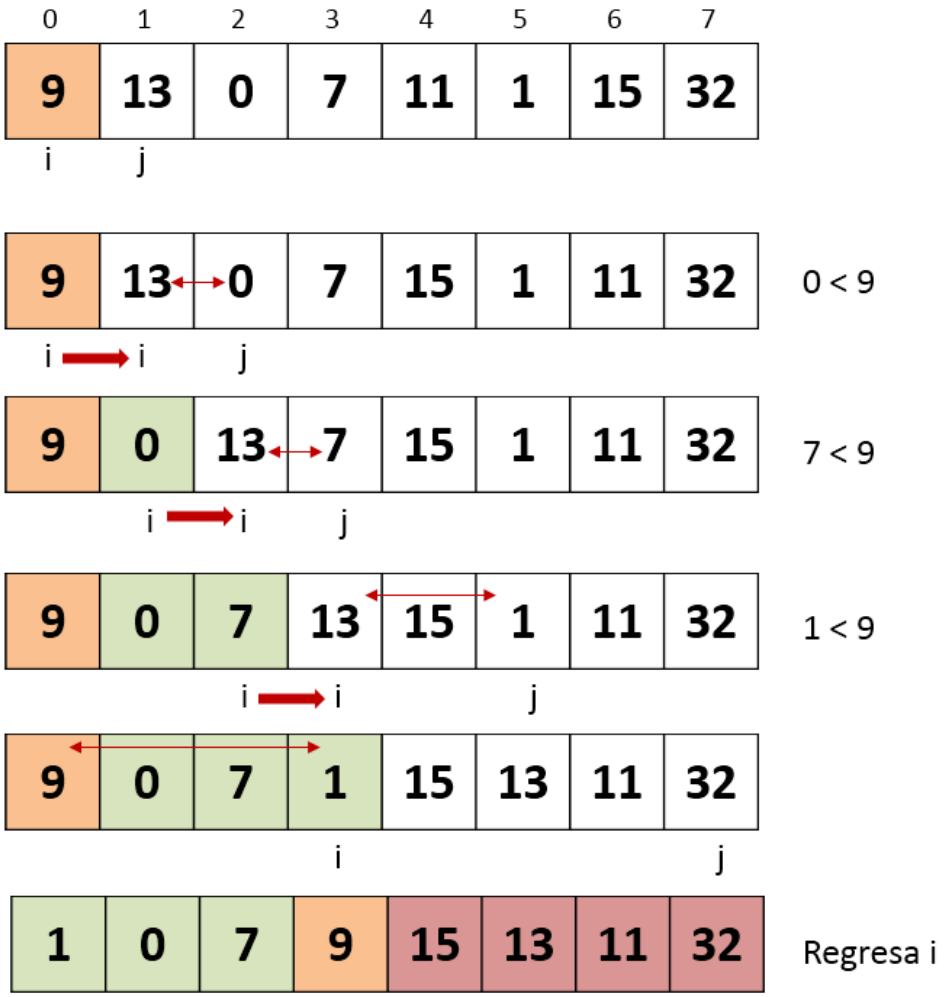


Figura 16 Representación gráfica de un ejemplo de partición.

Para el ejemplo se toma el primer número del arreglo como pivote y se compara con el resto de los elementos. La variable j es la encargada de recorrer cada una de las casillas. El índice i define dónde se encuentra la partición. Debido a que en un principio nada se ha categorizado, i empieza en la primera posición. Cuando se **encuentra un número menor al pivote** (9) ocurren dos cosas.

1. El índice (i) suma una posición.
2. Se intercambia el valor de la casilla A[i] (13) con el valor menor al pivote A[j] (0).

De esta forma se obtiene el segundo estado de la Figura 16. Siempre que se encuentre un elemento menor al pivote se repite la operación antes mencionada.

En caso contrario, si el número que se está comparando es mayor al pivote, no se modifica nada dentro del arreglo y el recorrido continúa.

Una vez que se termina de recorrer el arreglo, el valor de la primera casilla (el pivote) se intercambia con el valor que se encuentra en la frontera (A[0] se intercambia con A[i]). Así se obtienen un arreglo que tiene la forma mostrada en la Figura. Finalmente, la función devuelve el valor de la variable 'i', y con ella se puede repetir la partición, pero con dos subconjuntos de la entrada inicial.

Quicksort es un buen método de ordenamiento. A diferencia de otros algoritmos como *mergesort* (búsqueda por mezcla), no requiere almacenamiento auxiliar para funcionar y el desempeño es equivalente en la mayoría de los casos $O(n \log n)$ [12].

El arreglo se divide en dos partes una y otra vez, por lo tanto es posible representar la recurrencia de la siguiente forma:

$$T(n) = T(n-k) + T(k) + \theta(n)$$

Donde:

- $T(n-k)$ y $T(k)$ representan a quicksort con subconjuntos del inicial.
- $\theta(n)$ es el tiempo que toma la partición para una entrada de tamaño n.

Esta recurrencia es algo diferente a la que se había utilizado para representar la búsqueda binaria.

Peor caso

Lo peor que podría pasar para este ordenamiento es que la partición sólo descartara un número. Esta situación se da cuando el pivote es el número más pequeño del arreglo o el más grande. Esto significa que el trabajo es casi el mismo para el siguiente paso.

$$T(n) = T(n-1) + T(0) + \theta(n)$$
$$T(n) = T(n-1) + \theta(n)$$

Si esto pasara sólo una vez a lo largo de toda la ejecución no habría mayor problema. Pero cuando se repite recurrentemente, resulta en un desbalance de carga. Por lo tanto, el peor caso es cuando el arreglo está ordenado ascendente o descendentemente. Por lo tanto el trabajo total se puede representar así.

$$\begin{array}{rc} & T(n) + \theta(n) \\ & T(n-1) + \theta(n-1) \\ + & T(n-2) + \theta(n-2) \\ & \vdots \\ & T(n) = T(2) + \theta(2) \\ & T(n) = T(1) + \theta(1) \\ & - - - - - - - - - - \\ & T(n) \end{array}$$

Que equivale a:

$$T(n) = \sum_{k=0}^{n} T(n-k) + \theta(n-k)$$

Es posible ver que la recursión tiene forma de una sucesión aritmética que resulta en $O(n^2)$.

$$T(n) = O(n^2)$$

Con el análisis es posible concluir dos cosas, que son características destacables del ordenamiento.

- El desempeño en el peor de los casos para Quicksort es $O(n^2)$
- El peor caso es una entrada ordenada o casi ordenada; ya sea en orden ascendente o descendente.

Este desempeño no es totalmente indeseable, pero algoritmos como mergesort y heapsort tienen un desempeño en el peor caso de $\theta(n \log n)$ [12]. Sin embargo, es un problema no demasiado grande y tiene una solución: desordenar la entrada. La variante que implementa la solución se conoce como *Randomized Quicksort*. Esta variante tiene un desempeño para todos los casos de $\theta(n \log n)$.

El mejor caso

Así como el peor caso prácticamente no ahorra trabajo, el mejor caso divide la entrada de la mejor forma posible: a la mitad.

$$T(n) = 2T(n/2) + \theta(n)$$

Esta recursión es posible solucionarla con el segundo caso del método maestro. De esta forma se obtiene que para el mejor caso el desempeño del algoritmo.

$$T(n) = \theta(n \log n)$$

Este valor es el límite para algoritmos de ordenamiento que utilizan comparaciones y es el mismo para heapsort y mergesort. Estos tres métodos son óptimos.

El caso promedio involucra un análisis probabilístico más exhaustivo y se quiere evitar una mayor digresión respecto al tema central. La complejidad promedio es igual a la obtenida en el mejor caso [12]:

$$T(n) = \theta(n \log n)$$

La paralelización del algoritmo está claramente marcada por las llamadas recursivas. Cada una debe generar bifurcaciones en el flujo de ejecución. Por su parte, la función encargada de la partición es sumamente difícil de paralelizar bajo este modelo. Principalmente porque se hacen muchas escrituras a memoria compartida, lo que desata un sin número de carreras de datos.

El trabajo (W) ya fue calculado cuando se obtuvo la complejidad promedio del algoritmo.

$$W(n) = O(n \log n).$$

Para obtener la profundidad se puede hacer uso de la siguiente recurrencia.

$$D(n) = T_{\infty}(n) = T(n/2) + O(n)$$

Por el tercer caso del método maestro, resulta en una función lineal.

$$D(n) = O(n)$$

Por lo tanto, el paralelismo es logarítmico.

$$P(n) = \frac{W(n)}{D(n)} = \frac{O(n \log n)}{O(n)} = \mathrm{O}(\log n)$$

Este crecimiento teórico, no es malo; tampoco es ideal, ya que un crecimiento ideal es lineal. El principal problema yace en la partición del arreglo ya que es una rutina meramente secuencial. No obstante el paralelizar el algoritmo, en teoría, debe resultar en un mejor desempeño.

Dada toda esta información necesaria para entender mejor al algoritmo es posible continuar. La versión serial de Quicksort se presenta a continuación.

```
int particion(int *A, int ini, int fin)
{
    int pivote = A[ini], indice = ini, temp, j;
    for (j = (ini+1); j <= fin; j++)
    {
        if (A[j] <= pivote)
        {
            indice++;
            temp = A[indice];
            A[indice] = A[j];
            A[j] = temp;
        }
    }
    temp = A[ini];
    A[ini] = A[indice];
    A[indice] = temp;
    return indice;
}

void quicksort(int *A, int ini, int fin)
{
    int p, tam = fin - ini;
    if(tam>1)
    {
        p = particion(A, ini, fin);
        quicksort(A, ini, p);
        quicksort(A, p + 1, fin);
    }
}
```

Código 19 Implementación secuencial de quicksort en C.

La versión paralela del algoritmo no cambia demasiado. De hecho únicamente se agregan las palabras reservadas de Cilk para tener el programa listo.

```
int particion(int *A, int ini, int fin)
{
    int pivote = A[ini], indice = ini, temp, j;
    for (j = (ini+1); j <= fin; j++)
    {
        if (A[j] < pivote)
        {
            indice++;
            temp = A[indice];
            A[indice] = A[j];
            A[j] = temp;
        }
    }
    temp = A[ini];
    A[ini] = A[indice];
    A[indice] = temp;
    return indice;
}

void quicksort(int *A, int ini, int fin)
{
    int p, tam = fin - ini;
    if(tam>1)
    {
        p = particion(A, ini, fin);
        cilk_spawn quicksort(A, ini, p);
        cilk_spawn quicksort(A, p + 1, fin);
        cilk_sync;
    }
}
```

Código 20 Implementación de quicksort paralelo con C y Cilk.

El siguiente capítulo describirá los resultados obtenidos con los programas descritos. Por el momento, se continuará con el planteamiento para problemas que fueron implementados con CUDA.

Ejercicio 3: Multiplicación de matrices con CUDA

CUDA es una plataforma que tiene ventajas muy distintas a las que ofrece Cilk. El paralelismo de tareas no es realmente factible en CUDA, no fue diseñado para eso. CUDA cobra relevancia, cuando es necesario realizar operaciones con grandes cantidades de datos. En esos casos es una herramienta potente.

La multiplicación de matrices es una operación básica que sirve a su vez para resolver otros problemas. Para llevar a cabo la multiplicación hay que realizar el producto punto de las filas de una matriz con las columnas de la otra.

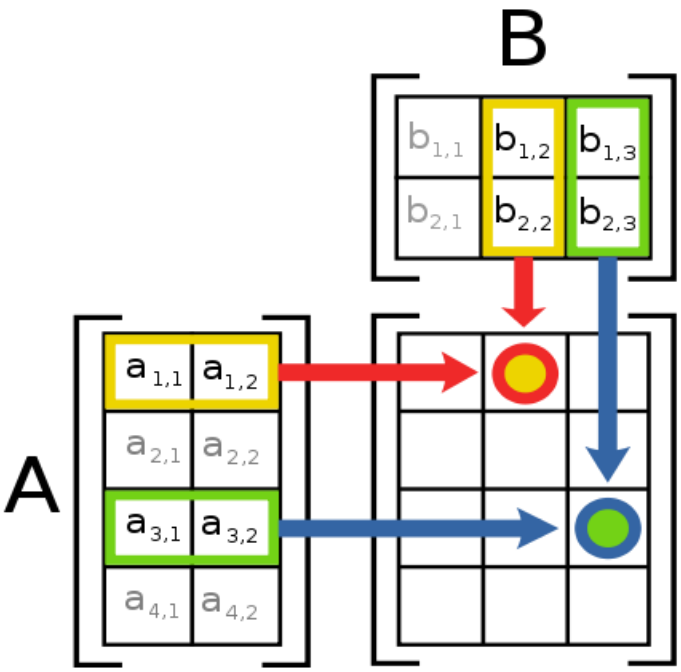


Figura 17 Visualización de la multiplicación de dos matrices.

De esta forma se obtienen los elementos que generan una nueva matriz. Un algoritmo secuencial simple es el siguiente:

```
multiplicacionM(A[AY][AX], B[BY][BX])
        C[AY][BX]
        for (i = 0 -> AY)
                for(j = 0 -> BX)
                        r1 = 0
                        for(k = 0 -> AX)
                                r1 += A[i][j+k] * B[j+k][i]
                        C[i][j] = r1
        return C
```

Código 21 Algoritmo iterativo para la multiplicación de matrices.

El análisis de este algoritmo es simple porque no es una recurrencia. Por lo tanto, es posible "contar" las instrucciones para definir su complejidad. Cuando se tiene un ciclo anidado lo más sencillo es multiplicar el número de instrucciones de cada uno. Dado que en este caso hay tres ciclos anidados se realizan dos multiplicaciones. En este caso, los ciclos dependen del tamaño de las matrices de entrada: AY, BX y AX o BY (que para que se pueda hacer la multiplicación deben ser iguales). Como los tres datos no son necesariamente iguales, el desempeño puede tomar la siguiente forma.

$$\Theta(mnk) = mnk$$

Donde:

- AY es igual a m
- BX es igual a n
- AX es igual a k
- Θ(mnk) representa la el desempeño promedio, ya que siempre se tiene que cubrir esa cantidad de operaciones.

Es posible ver tres variables lo que vuelve la función con crecimiento geométrico. Definitivamente no es el algoritmo más eficiente, pero sigue siendo una complejidad aceptable. Especialmente, frente a algoritmos con complejidad exponencial, que no son difíciles de encontrar.

```
int** mult(int A[][], int B[][])
{
	int i, j, k;
	for(i = 0; i < AY; i++)
		for (j = 0; j < BX; ++j)
		{
			for (k = 0; k < AX; ++k)
			{
				C[i][j] += A[i][k] * B[k][j];
			}
		}
}
```

Código 22 Implementación de la multiplicación de matrices secuencial con C.

Paralelizar este algoritmo no es demasiado complicado. Lo que se debe hacer es **la misma operación muchas veces simultáneamente**, por lo que se puede dividir la información y utilizar el GPU para llevar a cabo la tarea.

El análisis del algoritmo paralelo se obtiene una vez más con el trabajo y la profundidad. Por el lado del trabajo (W).

$$W(mnk) = nmk$$

Adicionalmente si se habla de matrices cuadradas que se multiplican las tres variables son iguales y el trabajo como una función cúbica.

$$W(n) = n^3$$

Para la profundidad, la cadena de operaciones dependientes son únicamente las multiplicaciones necesarias para calcular un elemento de la matriz resultante. Es decir, cada hilo realizará k multiplicaciones. Por lo tanto, cuando se cuenta con poder de procesamiento tan grande la profundidad es lineal, mientras que el trabajo es polinomial.

$$D(n) = O(n)$$

$$P(n) = O\left(\frac{n^3}{n}\right) = O(n^2)$$

El paralelismo no refleja la eficiencia del algoritmo, más bien representa que tan escalable es para una entrada en particular. Esto significa que con el uso de un GPU el algoritmo es, en teoría, sumamente escalable. La realidad debe acercarse bastante, aunque es necesario contemplar el cuello de botella que se debe a la transferencia de información entre máquina anfitrión y el dispositivo gráfico. El costo de éste no es demasiado grande, pero significará una reducción en desempeño real que no debe de causar mayor preocupación. Se ha dicho que mientras más procesadores se tengan más rápido irá el programa. En este caso los procesadores son parte de la tarjeta gráfica. Por lo que la mejora en desempeño dependerá de la capacidad de ésta.

```
//Función principal.
int main(int argc, char** argv)
{
	int *A, *B, *C, memoria_M, n, bloque, bloque2;
	double t, t2;

	srand(time(0));
	cout << "Tamanio matrices (mayor a 63) n: ";
	cin >> n;
	bloque = n/64;
	bloque2 = bloque;

	memoria_M = sizeof(int)*n*n;
	// Reservar memoria en el dispositivo grafico.
	int* A_kernel;
	int* B_kernel;
	int* C_kernel;

	// Reservar memoria para las matrices e inicializar.
	A = poblar(memoria_M, n*n,false);
	B = poblar(memoria_M, n*n,false);
	C = (int*) malloc(memoria_M);

	//Reservar memoria en el dispositivo de procesemiento gráfico
	cudaMalloc((void**) &A_kernel, memoria_M);
	cudaMalloc((void**) &B_kernel, memoria_M);
	cudaMalloc((void**) &C_kernel, memoria_M);

	//Tiempo1 incluye transferencia de datos
	t=tiempo();
	// Copiar datos al dispositivo
	cudaMemcpy(A_kernel, A, memoria_M, cudaMemcpyHostToDevice);
	cudaMemcpy(B_kernel, B, memoria_M, cudaMemcpyHostToDevice);


	// Parametros del dispositivo grafico
	dim3 threads(bloque, bloque);
	dim3 grid(n / threads.x, n / threads.y);

	//Tiempo2 solo registra la multiplicación
	t2 = tiempo();
	multiplicacion<<< grid, threads >>>(C_kernel, A_kernel, B_kernel,
	                                    n, n, bloque2);
	t2 = tiempo() - t2;

	cudaMemcpy(C, C_kernel, memoria_M, cudaMemcpyDeviceToHost);
	t=tiempo() - t;

	printf("%.3f\t%.3f\n",t,t2);
	cudaFree(A);
	cudaFree(B);
	cudaFree(C);
}
```

Código 23 Función principal de la implementación de la multiplicación en paralelo con C y CUDA.

Esta es la función principal, cuyas tareas incluyen, reservar e inicializar las matrices A, B y C. Vale la pena recalcar que **las matrices no son bidimensionales**, pero se reserva todo el espacio necesario en un solo arreglo lineal. El resultado final es el mismo. Al saberse el tamaño de las filas es posible expresar y multiplicar las matrices correctamente.

Adicionalmente, la función principal inicializa el kernel que será ejecutado por la tarjeta gráfica. A éste le envía los parámetros para una distribución adecuada del trabajo, como lo es el tamaño de la red de procesamiento y de los bloques que serán usados.

```
__global__ void multiplicacion(int* C, int* A, int* B, int xA,
                               int xB, int bloque)
{
      int tx = blockIdx.x * bloque + threadIdx.x;
      int ty = blockIdx.y * bloque + threadIdx.y;
      int valorA;
      int valorB;
      int valorC = 0;

      for (int i = 0; i < xA; ++i)
      {
            //la multiplicacion define la fila
            //y la suma define la columna
            valorA = A[ty * xA + i];//equivale a [xA][i]
            valorB = B[i * xB + tx];//equivale a [xB][tx]
            valorC += valorA * valorB;
      }
      C[ty * xB + tx] = valorC;//equivale a [xB][tx]
}
```

Código 24 Kernel que será ejecutado por el procesador gráfico.

Este es el código que ejecutará la tarjeta gráfica. Debido a que se quiere probar el algoritmo con matrices grandes es necesario utilizar más de un bloque de hilos de procesamiento. Cada bloque individualmente contiene hasta 512 hilos pero se descompondrá el problema de forma distinta para tener la mejor distribución de trabajo posible.

Una red de procesamiento contiene 64 x 64 bloques y en cada bloque se usarán únicamente n/64 hilos, así es como se logra obtener los elementos de procesamiento necesario para distintas matrices. En este programa en particular las matrices deben ser divisibles entre 64. Cada hilo será responsable de un producto punto.

Dentro del código las variables tx y ty son las encargadas de definir los elementos que serán multiplicados. También son usados para definir la casilla donde el resultado se almacenará en el arreglo resultante. Todo esto sucede dentro del ciclo for, que realiza el producto punto de dos vectores y guarda el resultado en C.

Las prácticas serán realizadas en el laboratorio de Intel. Todos los datos resultantes están incluidos en el capítulo 4 y en el apéndice.

IV Análisis de resultados

Como se mencionó en la metodología, el análisis está dividido en dos, el análisis cuantitativo y el análisis cualitativo. El primero, se enfoca en los números obtenidos de las ejecuciones de los programas. El segundo, describirá los requisitos y problemas que surgen de programar con una u otra plataforma.

Entre las herramientas utilizadas para las prácticas está **Visual Studio 2008;** versión de la suite de desarrollo de Microsoft que es compatible con Cilk y CUDA. Las piezas centrales son provistas por Intel con Cilk y NVIDIA con el kit de desarrollo de CUDA. Todo el código analizado se encuentra en el apéndice. La mayor parte del código es original; escrito a partir de los algoritmos analizados. Cualquier biblioteca o función adicional está citada en las fuentes. Se codificó con C/C++ estándar, sin considerar las extensiones utilizadas por las plataformas de desarrollo paralelo.

Cilk cuenta con un analizador de escalabilidad que permite obtener un valor práctico para el paralelismo de un programa ejecutado dada una entrada particular. También con ayuda de algunas instrucciones de Cilk se mide el tiempo para las funciones ejecutadas en paralelo de esta forma es como se obtienen los datos en las tablas.

CUDA requiere de un kit de desarrollo que incluye el compilador y el sistema en tiempo de ejecución, sin embargo, también se puede integrar con Visual Studio para facilitar la compilación.

A través de los resultados se busca demostrar que el paralelismo es necesario para un desarrollo de software óptimo. La gran cantidad de instrumentos de desarrollo que han aparecido en la última década es una ventaja, pero es importante elegir la plataforma adecuada para un problema determinado.

Análisis cuantitativo

Resultados práctica 1: Búsquedas

La idea de un algoritmo paralelo de búsqueda surgió de la búsqueda binaria, que es un excelente algoritmo. De hecho si se cuenta **con una entrada ordenada no existe motivo para usar otro algoritmo**. Los tiempos que arrojó la búsqueda para una entrada de 300 millones son interesantes.

Búsqueda Binaria		
	Entrada (n)	
Ejecución	100,000,000	300,000,000
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0
6	0	0
7	0	0
8	0	0
9	0	0
10	0	0

Tabla 1 Resultados de la búsqueda binaria recursiva.

Las unidades de tiempo de los resultados son segundos. El tiempo de ejecución para la búsqueda fue tan pequeño que la computadora no fue capaz de registrar una cantidad. **El logaritmo base dos de 300 millones es 28.16**, esto significa que en el peor de los casos la búsqueda binaria realiza 28 operaciones para encontrar un número. Por lo tanto, un algoritmo más eficiente es difícil de encontrar.

La situación cambia considerablemente para la búsqueda binaria cuando se tiene una entrada grande desordenada. En este caso, es necesario ordenar la entrada primero y posteriormente hacer la búsqueda. De hecho encontrar el elemento no tiene relevancia asintóticamente hablando; el término relevante es el ordenamiento. Para este ejercicio se utilizó una entrada de 300 millones, esta vez desordenada. El ordenamiento utilizado fue quicksort.

Quicksort Secuencial				Quicksort Paralelo (4 núcleos)	
	Entrada (n)			Entrada (n)	
Ejecución	100,000,000	300,000,000		100,000,000	300,000,000
1	26.5065	85.3929		13.822	39
2	27.2166	85.0249		13.838	38.985
3	26.5445	86.2449		13.401	38.984
4	25.7125	86.9125		13.4	38.61
5	25.5585	85.7531		13.775	39.204
6	25.3212	87.0368		13.869	39.11
7	26.1589	85.3929		13.604	38.688
8	25.321	84.9824		13.696	38.938
9	27.0127	85.0257		13.65	38.829
10	27.0021	85.2684		13.479	38.953
Promedio	**26.23545**	**85.70345**		**13.6534**	**38.9301**

Tabla 2 Resultados del ordenamiento previo a la búsqueda binaria.

Aunque se utilice una variante paralela del ordenamiento (4 núcleos), la búsqueda resulta lenta.

A pesar de la desventaja primordial de la búsqueda binaria, se cuenta con una búsqueda secuencial, que es sumamente simple y tiene una complejidad lineal. Por lo tanto, a pesar de no ser tan eficiente, no se queda atrás. Además, cuenta con la ventaja de que el orden de la entrada es irrelevante.

Búsqueda Lineal		
	Entrada (n)	
Ejecución	**100000000**	**300000000**
1	0.106006	0.633037
2	0.0109999	0.812046
3	0.130008	0.0610039
4	0.248014	0.735042
5	0.251014	0.505029
6	0.241014	0.744042
7	0.0690041	0.232014
8	0.239014	0.501028
9	0.104006	0.399022
10	0.025001	0.656038
Promedio	**0.1424081**	**0.52783019**

Tabla 3 Resultados de la búsqueda lineal.

¿Es posible superar a la búsqueda secuencial en términos de eficiencia y escalabilidad? La realidad es que es una tarea es difícil. Se pensó en utilizar una variante de la búsqueda binaria para lograr este objetivo. Sin embargo, la complejidad en el peor de los casos para esta búsqueda, no es mejor que su equivalente para la búsqueda lineal. La ventaja con la que cuenta es, que al ser un algoritmo recursivo, es fácilmente paralelizable. Como se mencionó en el planteamiento, el paralelismo teórico es:

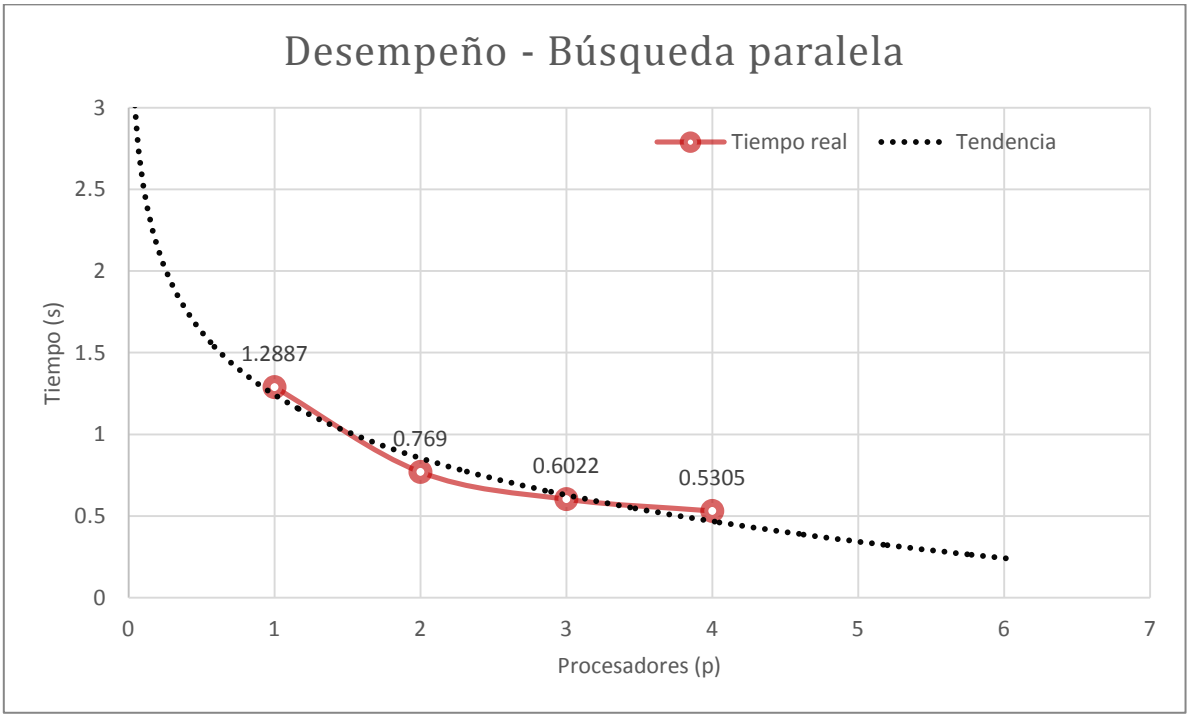
$$P(n) = O\left(\frac{n}{\log_2 n}\right)$$

Que a pesar de no ser demasiado escalable, es creciente. Empíricamente se obtuvieron los siguientes resultados para una entrada de 300 millones.

Búsqueda Binaria				
	Procesadores (p)			
Ejecución	**1**	**2**	**3**	**4**
1	1.295	0.78	0.624	0.53
2	1.279	0.764	0.624	0.531
3	1.295	0.78	0.593	0.546
4	1.295	0.749	0.608	0.531
5	1.279	0.764	0.593	0.53
6	1.28	0.765	0.593	0.53
7	1.295	0.78	0.593	0.515
8	1.279	0.764	0.608	0.531
9	1.311	0.764	0.593	0.546
10	1.279	0.78	0.593	0.515
Promedio	**1.2887**	**0.769**	**0.6022**	**0.5305**
Incremento	**1**	**1.67581274**	**2.13998672**	**2.42921772**
Paralelismo práctico		**2.37**		

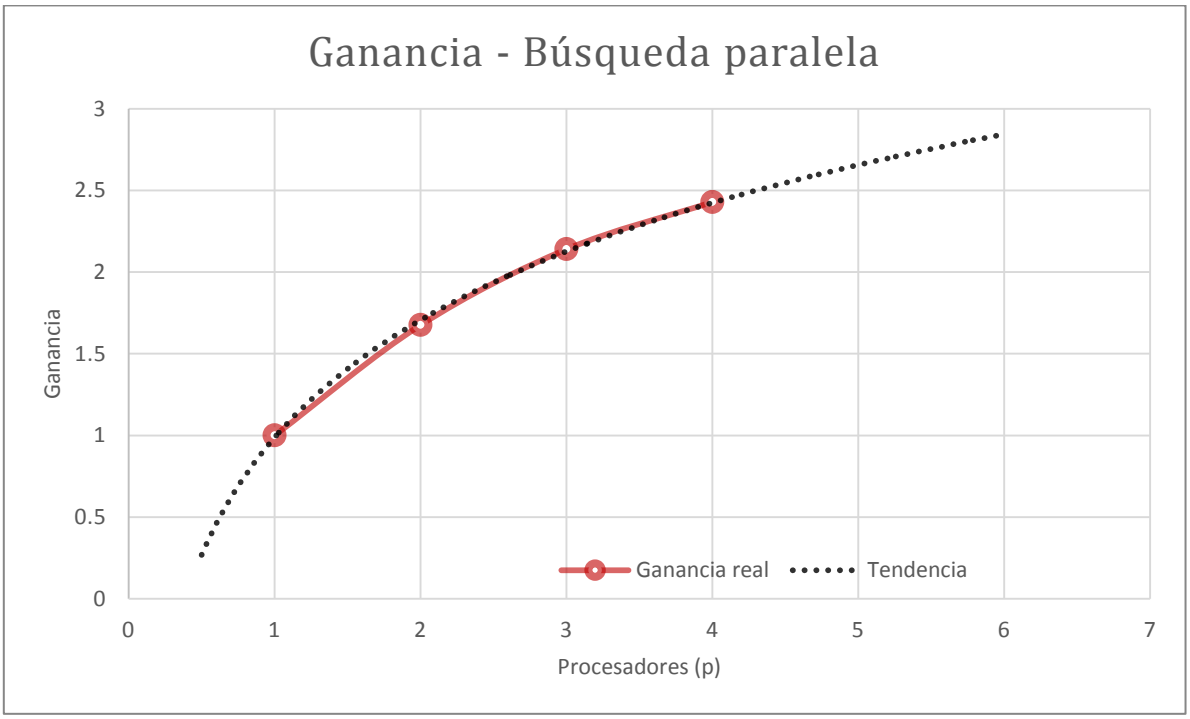
Tabla 4 Resultados de la búsqueda recursiva paralela.

Estos resultados no son demasiado alentadores. No se supera a la búsqueda lineal y el límite de escalabilidad resulto ser, en la práctica, bajo. La reducción temporal se acopla bien a un comportamiento logarítmico.



Gráfica 5 Representación de los valores temporales promedio de la búsqueda paralela.

Independientemente de no superar a los otros algoritmos de búsqueda, es interesante ver cómo el tiempo de ejecución se reduce al incrementarse el número de procesadores. Es notorio que las mejoras en desempeño perdieron velocidad rápidamente. La razón detrás de esto se encuentra en el **paralelismo práctico, que para una entrada de 300 millones fue de 2.37**. Esto significa que después de 2 procesadores las mejoras serán mucho más limitadas o nulas. El algoritmo no es altamente escalable, y hay que contemplar el costo beneficio de usar más procesadores para esta tarea.



Gráfica 6 Representación de la ganancia con cada incremento de procesadores.

Por desgracia, no se obtuvo un algoritmo más eficiente que la búsqueda binaria o lineal, pero eso no le quita potencial del paralelismo que puede hacer que hasta un algoritmo poco escalable tenga mejoras notorias (casi 150%). El incremento también se acopla a una tendencia logarítmica, principalmente porque después de la barrera de los 2 procesadores se pierde escalabilidad y eventualmente se podría llegar a una pared de desempeño. La tendencia se obtiene a partir de los datos experimentales y más datos significan más precisión, para ello habría sido necesario contar con más procesadores para pruebas.

Resultados práctica 2: Ordenamiento

Quicksort que es uno de los más eficientes algoritmos de ordenamiento. Es particularmente paralelizable debido a que es un algoritmo recursivo.

Teóricamente tiene un paralelismo logarítmico, lo que significa que es escalable, pero no demasiado. Esto se debe a la rutina que divide el arreglo en dos, no es fácilmente paralelizable.

$$P(n) = \mathrm{O}(\log n)$$

Se realizaron pruebas con el algoritmo secuencial y el paralelo. El secuencial trabajó con varias entradas hasta 300 millones, pero para comparar se analizó el desempeño para 10 millones de números. La entrada no contiene números repetidos y el conjunto va desde 0 hasta 9,999,999 sin saltarse un solo número. La entrada se encuentra desordenada.

Quicksort Secuencial					
	Entrada (n)				
Ejecución	**100,000**	**1000,000**	**10,000,000**	**100,000,000**	**300,000,000**
1	0.0170009	0.205011	2.25613	26.5065	85.3929
2	0.0170009	0.196011	2.26013	27.2166	85.0249
3	0.016001	0.193011	2.30313	26.5445	86.2449
4	0.016001	0.198012	2.30013	25.7125	86.9125
5	0.016001	0.194011	2.39414	25.5585	85.7531
6	0.0170009	0.190011	2.24313	25.3212	87.0368
7	0.0170012	0.193011	2.29613	26.1589	85.3929
8	0.016	0.196011	2.42514	25.321	84.9824
9	0.0170012	0.195011	2.27113	27.0127	85.0257
10	0.0170009	0.196012	2.29313	27.0021	85.2684
Promedio	**0.0166009**	**0.1956112**	**2.304232**	**26.23545**	**85.70345**

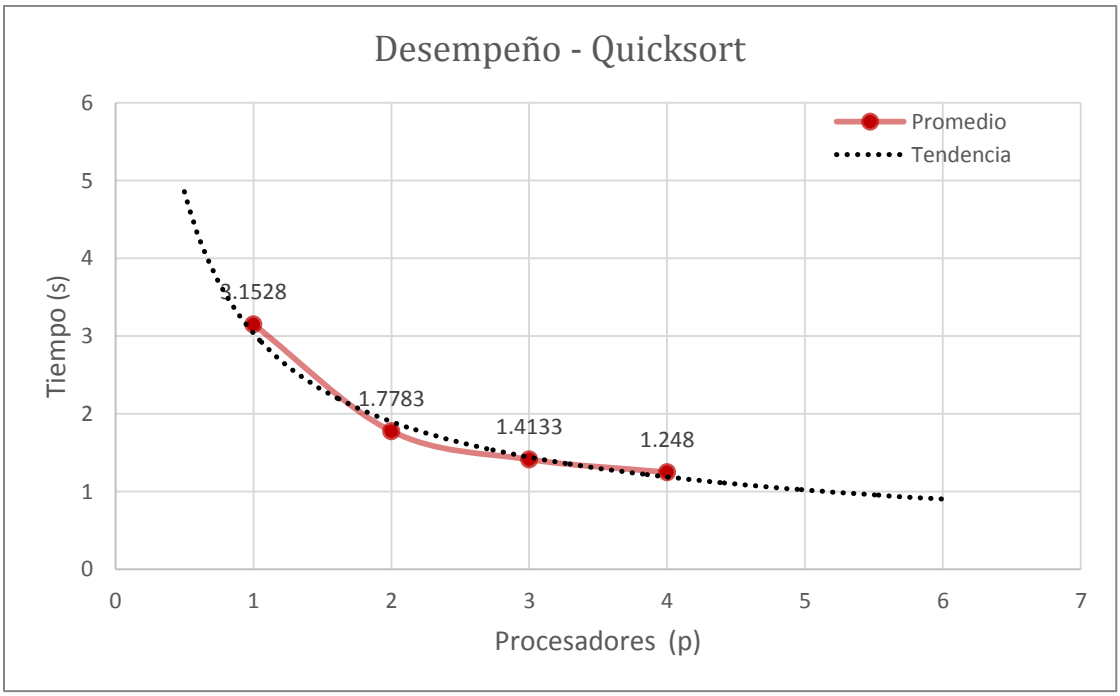
Tabla 5 Resultados de Quicksort secuencial.

Es posible observar que el tiempo de ejecución se dispara a partir de los 100 millones de números. La intención es mejorar el desempeño con la versión paralela del algoritmo. Para la cual se obtuvieron los siguientes resultados.

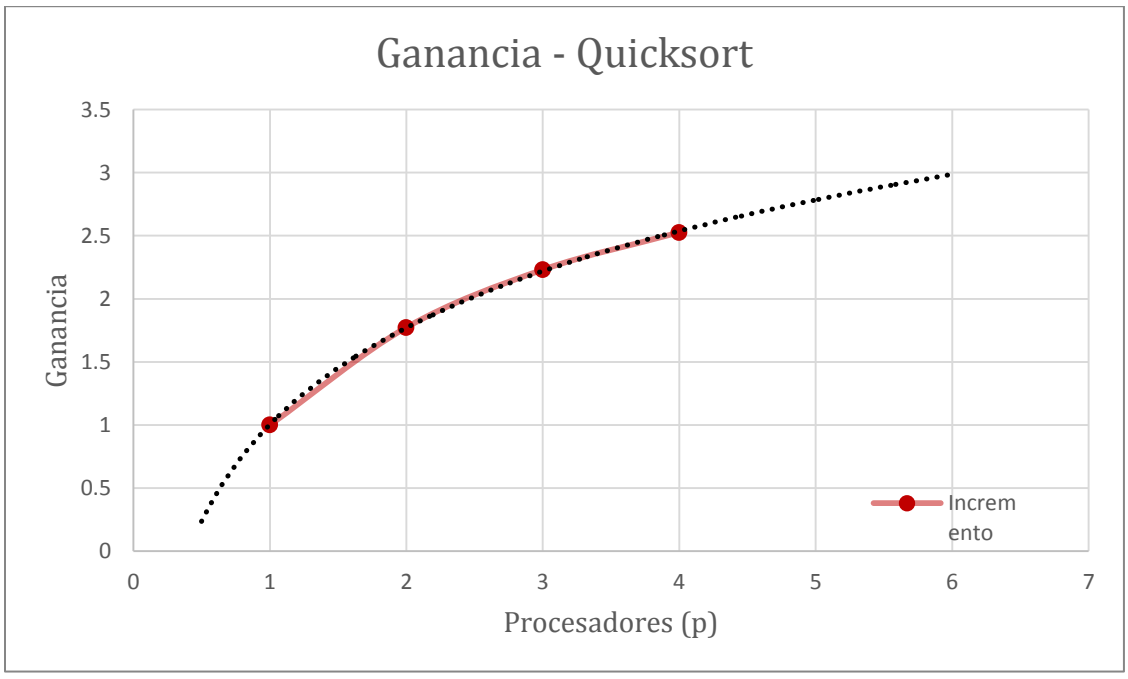
Quicksort Paralelo				
	Procesadores (p)			
Ejecución	**1**	**2**	**3**	**4**
1	3.058	1.716	1.388	1.216
2	3.058	1.716	1.42	1.248
3	3.089	1.794	1.404	1.264
4	3.557	1.763	1.419	1.248
5	3.104	1.763	1.419	1.248
6	3.198	1.778	1.42	1.217
7	3.089	1.887	1.42	1.232
8	3.151	1.809	1.435	1.295
9	3.104	1.81	1.404	1.248
10	3.12	1.747	1.404	1.264
Promedio	**3.1528**	**1.7783**	**1.4133**	**1.248**
Incremento	**1**	**1.7729292**	**2.23080733**	**2.52628205**
Paralelismo	**2.9**			

Tabla 6 Resultados de Quicksort paralelo.

Es necesario recalcar, que de los datos obtenidos que para un solo procesador, la versión paralela ocupa más tiempo completar el ordenamiento. La razón detrás de la diferencia en tiempos, se debe a que Cilk lleva consigo un costo por el simple hecho de ser utilizado. Las llamadas a funciones tipo Cilk y el sistema en tiempo de ejecución consumen recursos. No es un costo demasiado grande en este caso y como el desempeño mejora para múltiples procesadores es posible absorber el costo.



Gráfica 7 Representación de los valores temporales promedio del ordenamiento en paralelo.



Gráfica 8 Representación de la ganancia con cada incremento de procesadores para el ordenamiento en paralelo.

La mejora de 1 a 2 procesadores fue de 77% y 123% con 3 (respecto al tiempo para un procesador). La diferencia relativa entre el segundo grupo de datos y el tercero es de 45.79% lo que representa un crecimiento 40% menor respecto al primer incremento. Algo similar se observa entre los datos de 3 y 4 procesadores. Aunque el crecimiento respecto a un

procesador es del 152% relativo al tiempo con 3 procesadores es únicamente del 29.54%. Esto significa que el crecimiento una vez más se desacelera.

Porcentaje (%)			
Tiempo	**Respecto a 1**	**Respecto a 2**	**Respecto a 3**
1			
1.7729	77.29		
2.2308	123.08	45.79	
2.5262	152.62	75.33	29.54

Tabla 7 Datos sobre la mejora porcentual en con cada incremento.

Todos estos datos hablan de una escalabilidad reducida. En un principio se tiene un salto grande en desempeño pero este incremento no se mantiene. El paralelismo práctico, para el ordenamiento es de 2.9 lo que limita la escalabilidad principalmente a partir del tercer procesador. Es posible obtener mejores resultados si se siguen incorporando núcleos, pero el costo se incrementa y el beneficio es reducido.

Esta variante logró superar a su equivalente secuencial y se puede considerar como un resultado favorable. Sin embargo, la escalabilidad reducida hace que el logro se vea mermado. El problema central se encuentra en la partición.

Resultados práctica 3: Multiplicación de matrices

Se escogió este problema por varias razones.

1. Este es un algoritmo que opera una gran cantidad de información. Lo que lo presta para la implementación de paralelismo en datos.
2. Realiza una sola operación y esta simpleza sirve para apreciar mejor este paradigma.
3. Por otro lado, el algebra matricial y vectorial es muy utilizada en el ámbito de la computación gráfica, y aunque no es el único problema que se puede resolver con procesadores gráficos, sienta una base sólida para otros problemas.

En resumidas cuentas es simple, paralilzable y un buen modelo a seguir para problemas similares.

El tamaño de las matrices que se usaron en los programas fue de 1024x1024 (porque son divisibles entre 64). Por lo tanto, se operan 1 millón de números. Se usó esta entrada ya que representa suficiente trabajo para el algoritmo secuencial y con ella es posible observar el potencial del algoritmo paralelo.

El algoritmo serial, tiene una complejidad cúbica para matrices cuadradas, por lo que el crecimiento varía fuertemente entre 512 y 1024 elementos por lado.

Multiplicación Secuencial				
	Entrada (lado = n)			
Ejecución	**128**	**256**	**512**	**1024**
1	0.022	0.086	0.657	13.286
2	0.009	0.067	0.625	13.37
3	0.008	0.068	0.638	13.213
4	0.009	0.068	0.641	13.158
5	0.008	0.067	0.63	13.199
6	0.009	0.069	0.641	13.217
7	0.008	0.067	0.639	13.201
8	0.009	0.068	0.635	13.296
9	0.008	0.067	0.641	13.273
10	0.009	0.067	0.648	13.165
Promedio	**0.0099**	**0.0694**	**0.6395**	**13.2378**

Tabla 8 Resultados de la multiplicación de matrices secuencial.

Por otro lado, el algoritmo paralelo prácticamente no toma tiempo para multiplicar 1,048,576 de números. De hecho la computadora registra tiempos de ejecución sumamente bajos. Lo que sí toma tiempo es el intercambio de datos entre el CPU y el dispositivo gráfico. Este es un cuello de botella que hay que tomar en cuenta al trabajar con arquitecturas híbridas.

Multiplicación Paralela							
Transferencia de datos y multiplicación						Sólo multiplicación	
	Entrada (n)					Entrada (n)	
Ejecución	**128**	**256**	**512**	**1024**		**512**	**1024**
1	0.005	0.31	2.849	4.23		0	0
2	0.005	0.31	2.793	4.26		0	0.001
3	0.005	0.307	2.792	4.254		0	0.001
4	0.005	0.311	2.793	4.23		0	0
5	0.005	0.308	2.793	4.238		0	0
6	0.005	0.312	2.795	4.231		0	0
7	0.005	0.308	2.795	4.234		0	0
8	0	0.307	2.793	4.242		0	0.001
9	0.005	0.307	2.793	4.231		0	0
10	0	0.31	2.794	4.241		0	0
Promedio	**0.004**	**0.309**	**2.799**	**4.2391**		**0**	**0.0003**

Tabla 9 Resultados de la multiplicación de matrices paralela.

La tabla de la izquierda representa la multiplicación e incluye el tiempo que toma copiar los datos a la memoria de gráficos y de regreso. Cabe recalcar que la multiplicación por si sola es ejecutada muy rápidamente por el dispositivo gráfico.

Con CUDA no fue posible calcular la velocidad de procesamiento para diferentes cantidades de unidades de procesamiento ya que el control de la tarjeta gráfica no se le da al desarrollador en su totalidad. De hecho, el programa debe dividirse de la mejor forma posible para que el dispositivo pueda completar su objetivo. De lo contrario, puede que no obtenga un resultado coherente. Las observaciones para entradas de 512 y 1024 elementos son alentadoras y demuestran el potencial de la arquitectura.

Análisis cualitativo

Desarrollo con Cilk

Cilk es muy simple y la curva de aprendizaje es baja. Cuenta con cuatro palabras reservadas centrales (cilk_main, cilk_spawn, cilk_for y cilk_sync), hay más de cuatro pero esas son necesarias para paralelizar un algoritmo. Otras características valiosas son la capacidad de manejar carreras de datos automáticamente con reductores y también implementa exclusión mutua con bloqueos. La instalación no fue problemática, el requerimiento central es contar con un procesador Intel. Trabajar con Visual Studio facilita el desarrollo, pero no es indispensable. Cilk es muy flexible.

Con Cilk es más importante concentrarse en el algoritmo a grandes rasgos, que en los detalles de bajo nivel relacionados con el cómputo paralelo. Esto no significa que Cilk resuelva absolutamente los problemas de desarrollo de bajo nivel. Así que un entendimiento

adecuado de los conceptos fundamentales (balance de carga, carreras de datos, etc.) puede hacer toda la diferencia.

Desde el punto de vista de un desarrollador, la tarea de transformar un programa secuencial con Cilk es sumamente sencilla. Únicamente es necesario agregar las palabras adecuadas antes de funciones o ciclos y sincronizar cuando sea necesario. Para que la tarea sea así de simple, el algoritmo debe cumplir con ciertos requisitos; ser recursivo o contar con paralelismo en tareas. Además es deseable que la profundidad del algoritmo no sea pequeña respecto al trabajo total y de esta manera potenciar la escalabilidad. No todos los algoritmos cumplen estas características, y aun cuando no las cumplen hay formas de adaptarlos para Cilk. En estos casos, Cilk se vuelve una herramienta limitada. Afortunadamente, muchos algoritmos pueden resolverse recursivamente y pueden optimizarse para trabajar con Cilk.

Convertir la búsqueda binaria y quicksort a algoritmos paralelos no implicó un esfuerzo mayor a la hora de codificar, simplemente se añadieron los comandos necesarios. El reto fue diseñar las variantes de los algoritmos que realmente aprovechen el paralelismo, especialmente con la búsqueda. Ambos casos aprovecharon el poder de procesamiento adicional, pero no resultaron paralelizables a gran escala. Parte se debe a la naturaleza de los algoritmos, y es probable que se deba recurrir a búsquedas y ordenamiento totalmente diferentes a los tradicionales (diseñados secuencialmente).

Cilk resultó ser una herramienta simple, esta característica le da un gran potencial. El desarrollo significó una buena experiencia y el aprendizaje obtenido va más allá de la plataforma. Los conocimientos sobre algoritmos y paralelismo, reforzados para poder utilizar Cilk, son de gran utilidad en el desarrollo en general.

Desarrollo con CUDA

CUDA es una herramienta mucho más compleja que Cilk. Empezando por los requerimientos y la instalación. Es necesario contar con una tarjeta gráfica NVIDIA y hay una lista de tarjetas gráficas compatibles, porque no todas soportan la plataforma. A pesar de que hoy en día no es raro contar con una tarjeta gráfica de este tipo, el requerimiento no es tan fácil de satisfacer como lo es contar con un procesador Intel.

A diferencia de Cilk, CUDA requiere de la instalación de dos paquetes de software para funcionar:

- Driver
- Kit de desarrollo
- Visual Studio (opcional)

La instalación no es muy complicada. Sólo hay que tener en cuenta el orden en el que se instalan los productos. Visual Studio debe de estar instalado antes del software de NVIDIA para que se configure adecuadamente.

Para este trabajo no se ocuparon ni la mitad de las palabras reservadas de CUDA, pero las bases que se sentaron sirven para desarrollar proyectos más completos.

El desarrollo se divide en dos una vez más.

- Encontrar el algoritmo adecuado.
- Balancearlo para que aproveche al dispositivo gráfico.

Los **algoritmos que aprovechan las capacidades de los procesadores gráficos** adecuadamente normalmente involucran una **gran cantidad de operaciones sobre datos independientes entre sí**. Como lo son los algoritmos de detección de colisiones e iluminación y sombreado.

Por otro lado, el balance de trabajo no lo lleva a cabo CUDA automáticamente. También es necesario diferenciar claramente lo que se ejecuta en el CPU y en el GPU. También es necesario copiar explícitamente de memoria principal al dispositivo gráfico y regresarla para que el CPU pueda interpretarla. Esto hace menos flexibles a los programas en CUDA, y puede complicar el desarrollo.

Para la implementación de la multiplicación paralela fue necesario tener en cuenta que los bloques de procesamiento sólo cuentan con 512 hilos, y si se busca hacer una multiplicación con más de 512 elementos por lado es necesario utilizar más bloques. Para la multiplicación se dividía el número de lados de la entrada (n) entre 64 bloques y eso resultaba en el número de hilos que ejecutaría cada bloque. A través de la identificación con la que cuenta cada bloque y cada hilo es posible asignar los valores a multiplicar. Aunque no es tan flexible como Cilk es posible adaptar el programa a las necesidades del desarrollador, simplemente hay que invertir una mayor cantidad de tiempo en los detalles del flujo de ejecución.

El resultado del desarrollo, fue un programa sumamente eficiente, la multiplicación se llevó a cabo para un millón de números en menos de un segundo. La desventaja de la arquitectura aparece con la dependencia de transferir información desde la memoria principal a la memoria gráfica. Estas operaciones sí ocuparon tiempo y de no tomarse en cuenta este problema, se puede perder el propósito del procesamiento sobre tarjetas gráficas.

A fin de cuentas, queda mucho que aprender sobre este tipo de arquitectura y plataforma. Pero fue sumamente interesante desarrollar y observar las mejoras obtenidas, a pesar de los constantes obstáculos.

Conclusiones

Con este trabajo se tuvo la oportunidad de reafirmar los conceptos que son base del análisis de algoritmos. Aunque esté lejos de ser el tema central del trabajo es un tema necesario. Ya que los ingenieros tienen regularmente un trabajo de carácter metódico este tipo de tareas respaldan argumentos y decisiones. En este ámbito la conclusión más importante, se encuentra en que **el trabajo teórico que involucra tareas como el diseño y análisis de algoritmos no pierde vigencia**. Aún en el desarrollo de nuevos paradigmas el análisis sustenta la práctica.

Dentro el trabajo fue utilizado para sentar las bases de las variantes paralelas de algoritmos clásicos. Los resultados prácticos tal vez no fueron sorprendentes, pero en realidad cumplieron con las expectativas presentadas en el planteamiento.

Después de meses de investigación, es posible concluir que **el paralelismo es la evolución natural del cómputo**. Es un paradigma que rompe con las barreras físicas más próximas con las que se enfrenta el hardware. Además, extiende el tiempo de crecimiento, en términos de poder de procesamiento algunos años.

El desarrollo en paralelo es más complicado que el secuencial. Esto es, en parte, porque se acostumbra enseñar a resolver problemas de la segunda forma. La investigación, programación e implementación rescataron conceptos que se aprendieron en el salón de clases, pero que no habían sido utilizados en la práctica.

El paralelismo trae consigo complicaciones, pero ya hay gente sumamente capaz que ha trabajado para simplificar el desarrollo en este rubro. Cilk y CUDA son las bases de un desarrollo paralelo simple y poderoso. Cilk, con el manejo óptimo de múltiples tareas, y CUDA, una máquina de procesamiento lógico matemático. Ambos facilitan ciertas tareas a los programadores.

Cilk resultó ser una de las herramientas más flexibles y útiles para el desarrollo paralelo. Las ventajas mencionadas están ligadas a su simpleza. Es poco probable que llegue a ser utilizada en ambientes de producción, pero para cómputo académico y de investigación no puede quedar descartada. Tiene potencial como instrumento didáctico. Cilk vuelve cualquier programa secuencial en uno paralelo, y si está bien diseñado este se transforma en una herramienta ideal y escalable.

Por otro lado, CUDA resultó ser un mayor reto. No se encarga de los detalles de bajo nivel como Cilk, y esto hace del aprendizaje un reto. Un algoritmo que aprovecha el poder de los procesadores gráficos, se vuelve irreconocible una vez que se implementa correctamente. La fuerza bruta de un GPU debe considerarse para la solución de problemas que manejan una gran cantidad de datos.

CUDA cuenta con todo el apoyo de una empresa como NVIDIA, que ha invertido para que la plataforma crezca. No sólo eso, las plataformas híbridas (combinación CPU /GPU) están siendo aprovechadas incluso por aplicaciones comerciales [19]. **CUDA, de las dos plataformas analizadas, es la que cuenta con mayor poder y potencial.** Así que hay más de una buena razón para continuar trabajando sobre esta línea.

Ambas plataformas tienen ventajas y desventajas, pero comparten una característica: de no buscarse la solución adecuada, por más poderosas que sean, no podrán solucionar el problema.

Al final, los resultados no rompieron records, pero fundamentan el potencial del paralelismo y los datos confirman que los algoritmos son escalables. Es claro que se pueden realizar afinaciones en cada uno de los programas (la mejora es un proceso continuo). Sin embargo, los verdaderos resultados se encuentran a lo largo de todo el proceso, desde el análisis del algoritmo hasta la implementación y obtención de datos. El proceso como tal, es un resultado favorable, que está sustentado por conocimientos obtenidos durante la carrera y gracias a la investigación necesaria para la tesis.

Este trabajo no es una predicción de lo que ocurrirá en la industria, es un reflejo del presente. Tal vez no sea el presente más sonado en círculos de discusión, pero son cambios tangibles [22].

En la actualidad la mayoría de los desarrolladores profesionales y en formación tienen bases para solucionar problemas especialmente de manera secuencial. El ambiente educativo y profesional está cambiando, especialmente ahora que el paralelismo está más presente que nunca[4], [5].

Este trabajo, como parte de un proyecto educativo, busca presentar un panorama de las necesidades educativas de las próximas generaciones de ingenieros en computación. Busca servir como referencia de las bases teóricas y prácticas necesarias para empezar a introducir temas de suma relevancia al currículo de la carrera.

Hay muchos otros cambios en el horizonte, pero el paralelismo es vigente. Su presencia es tan fuerte que, pase lo que pase, cambiará la mentalidad de la industria.

Reflexiones finales

La cantidad de conocimientos necesarios para culminar mi tesis, fue mayor de lo esperado en un principio. Entiendo que tal vez es necesario acotar más este tipo de trabajos, pero mi renuencia a especializarme en demasía al final de la licenciatura me presentó un gran obstáculo, y de igual manera una gran satisfacción

El trabajo representó un enorme aprendizaje personal; fue un verdadero reto. Los temas que abarca el trabajo, me interesan mucho. Estoy seguro que se puede llegar a detallar cada tópico al punto de escribir una tesis sobre cada algoritmo, arquitectura y plataforma aludido a lo

largo de la tesis, pero de hacerlo no existiría un enfoque necesario para sacar el proyecto adelante. Sin embargo, da pie a mucho más trabajo y aprendizaje.

Apéndice

Apéndice A - Conceptos

Progresión aritmética

En matemáticas una progresión aritmética es de la siguiente forma:

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n$$

$$\sum_{k=1}^{n} k = \frac{1}{2} n(n+1)$$

Una forma de demostrarlo es por inducción matemática.

Se prueba para el primer elemento.

$$\sum_{k=1}^{1} k = 1$$

Usando la fórmula.

$$\sum_{k=1}^{n} k = \frac{1}{2} 1(1+1) = \frac{2}{2} = 1$$

Se asume que funciona para n pero hay que probarlo para n+1.

$$\sum_{k=1}^{n+1} k = \frac{1}{2}(n+1)(n+2)$$

Ya que las progresiones aritméticas son lineales se puede dividir la sumatoria.

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1)$$

$$= \frac{1}{2} n(n+1) + (n+1)$$

$$= \left(\frac{1}{2} n + 1\right)(n+1)$$

$$= \frac{1}{2}\left(\frac{2}{2} n + 2\right) + (n+1)$$

$$= \frac{1}{2}(n+2)(n+1) \quad \blacksquare$$

De esta forma queda demostrada la fórmula para progresiones aritméticas.

Desbordamiento de memoria

El desbordamiento de memoria sucede cuando se excede el espacio de almacenamiento designado. Por ejemplo: cuando se maneja una variable de tipo entero esta está conformada por 32 bits que equivalen a 4 bytes. La cantidad máxima que puede almacenar una variable de esta índole es $2^{32} - 1$. Por lo tanto, si se tiene dos números suficientemente grandes y se suman es posible exceder la cantidad de memoria reservada. Es un caso poco común, pero cuando se diseñan productos robustos, se espera que puedan soportar este tipo de eventos.

'm' se calcula: m= $ini + \frac{(fin-ini)}{2}$, que es equivalente a m= $\frac{(fin+ini)}{2}$ el problema es que de esta forma se puede incurrir en un **desbordamiento de memoria**. Por lo que se recomienda evitar problemas utilizando la primera operación, que realiza una resta y división antes de la suma [23].

Método maestro

El método maestro es una manera sencilla de resolver recurrencia, es un complemento a la notación asintótica para el análisis de algoritmos. No se pueden resolver todas las recurrencias con este método pero suficientes como para tener relevancia.

Para poder ser resulta por este método la recurrencia debe tener la siguiente forma.

$$T(n) = aT(n/b) + f(n)$$

Donde a y b son constantes y cumplen, $a \geq 1$ y $b > 1$. $f(n)$ es una función asintóticamente positiva, es decir, existe un valor a partir de la cual el resultado de la función siempre será positivo.

El método maestro consta de tres casos.

- Si $f(n) = O(n^{\log_b a - \epsilon})$ para una constante $\epsilon > 0$, entonces el resultado de la recurrencia es $T(n) = \Theta(n^{\log_b a})$.
- Si $f(n) = \Theta(n^{\log_b a} \lg^k n)$, para una constante $k \geq 0$, entonces el resultado de la recurrencia es $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
- Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ para una constante $\epsilon > 0$, entonces el resultado de la recurrencia es $T(n) = f(n)$.

La clave del método maestro es comparar $f(n)$ con $n^{\log_b a}$.

- Si $f(n)$ es menor se cumple el primer caso.
- Si $f(n)$ es igual se cumple el segundo caso.
- Si $f(n)$ es mayor se cumple el tercer caso.

Ejemplo 1:

$$T(n) = T(n/2) + n$$

$$n^{\log_b a} = n^{\log_2 1} = 1 < n$$

$$\text{Caso 3} \therefore T(n) = n$$

Ejemplo 2:

$$T(n) = 4T(n/2) + n^2$$

$$n^{\log_b a} = n^{\log_2 4} = n^2 = n^2$$

$$\text{Caso 2} \therefore T(n) = n^2 \lg n$$

Ejemplo 3:

$$T(n) = 27T(n/3) + n^2$$

$$n^{\log_b a} = n^{\log_3 27} = n^3 > n^2$$

$$\text{Caso 1} \therefore T(n) = n^3$$

Apéndice B – Código

Funciones para el manejo de arreglos

```
//arreglo.h
#include <iostream>

using namespace std;

 int* poblar(int t, int o)
{
       int *A;
      A = ( int *)malloc(sizeof( int)*t);
       int i, randNum, num=0;
      for(i=0;i<t;i++)
    {
            if(o==1)
            {
                  randNum = rand()%3;
                  if(randNum==2)
                        num+=2;
                  else if(randNum==1)
                        num++;
            }
        A[i]=num;
        num++;
    }
      return A;
}

bool verificar_orden(int n, int A[])
{
      //Solo funciona para arreglos ordenados cuyos indices coinciden con
el valor.
      int i;
      for(i = 0; i < n; i++)
            if(i!=A[i])
                  return false;
      return true;
}

void imprimir(int t, int A[])
{
      int i = 0;
      for(int i=0; i<t; i++)
            cout << "A[" << i << "] " << A[i] << "\n";
}
```

Búsquedas secuenciales

```
//Función principal

#include <algorithm>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "arreglo.h"
#include "busqueda.h"
#include "tiempo.h"


using namespace std;

int main()
{
	int *A;
	int dato, n=0;
	double t;
	//Inicialización
	srand(time(0));
	cout << "n: ";
	cin >> n;
	A=poblar(n,0);
	std::random_shuffle(A, A + n);

	//Búsqueda binaria
	dato = rand()%(n);
	cout << "Buscando " << dato << " dentro del arreglo\n";
	t = tiempo();
	dato = b_binaria(A,0,n-1,dato);
	t = tiempo() - t;
	cout << "Tiempo busqueda binaria: "<< t << " segundos.\n";
	if(dato>=0)
		cout << "Se encontro: " << A[dato] << " en "<< dato << "\n";
	else
		cout << "No se encuentra en el arreglo.\n";

	//Búsqueda lineal
	dato = rand()%(n);
	cout << "Buscando " << dato << " dentro del arreglo\n";
	t = tiempo();
	dato = b_secuencial(A,n,dato);
	t = tiempo() - t;
	cout << "Tiempo busqueda lineal: "<< t << " segundos.\n";
	if(dato>=0)
		cout << "Se encontro: " << A[dato] << " en "<< dato << "\n";
	else
		cout << "No se encuentra en el arreglo.\n";
	return 0;
}
```

```
//Funciones de búsqueda. busqueda.h

int b_secuencial(int A[], int n, int d)
{
	int i;
	for(i = 0;i<n;i++)
		if(A[i]==d)
			return i;
	return -1;
}

int b_binaria(int A[], int inf, int sup, int d)
{
	int m = inf +(sup-inf)/2;
	while(inf<=sup)
	{
		if(A[m]==d)
			return m;
		if(A[m]>d)
			return b_binaria(A,inf,m-1,d);
		if(A[m]<d)
			return b_binaria(A,m+1,sup,d);
	}
	return -1;
}

int b_recursiva_desorden(int A[], int inf,int sup,int d)
{
	int m = inf+(sup-inf)/2;
	int a=-1,b=-1;
	if(inf<=sup)
	{
		if(A[m]==d)
			return m;
		else
		{
			a =  b_recursiva_desorden(A,inf,m-1,d);
			b =  b_recursiva_desorden(A,m+1,sup,d);
			if(a>=0)
				return a;
			else if(b>=0)
				return b;
		}
	}
	return -1;
}
```

Búsqueda en paralelo

```
//Función principal

#include <algorithm>
#include <cilkview.h>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "arreglo.h"
#include "busqueda_paralela.h"

using namespace std;
cilk::cilkview cv;

int cilk_main()
{
	int *A;
	int dato, n=0;
	double t;
	//Inicialización
	srand(time(0));
	cout << "Tamanio arreglo n: ";
	cin >> n;
	A=poblar(n,0);
	std::random_shuffle(A, A + n);
	dato = rand()%(n);
	cout << "Buscando " << dato << " dentro del arreglo\n";

	//Búsqueda
	cv.reset();
	cv.start();
	dato = b_paralela(A,0,n-1,dato);
	cv.stop();
	t = (long)cv.accumulated_milliseconds();
	cout << "Tiempo: "<< t/1000 << "segundos\n ";

	//Resultados
	if(dato>=0)
		cout <<"Se encontro: " << A[dato] <<" en "<<dato<< "\n";
	else
		cout << "No se encuentra en el arreglo.\n";
	return 0;
}
```

```
//Función de búsqueda

int b_paralela(int A[], int inf,int sup,int d)
{
	int m = inf+(sup-inf)/2;
	int a=-1,b=-1;
	if(inf<=sup)
	{
		if(A[m]==d)
			return m;
		else
		{
			a = cilk_spawn b_paralela(A,inf,m-1,d);
			b = b_paralela(A,m+1,sup,d);
			cilk_sync;
			if(a>=0){
				return a;
			}
			else (b>=0)
			{
				return b;
			}
		}
	}
	return -1;
}
```

Quicksort secuencial

```
//Función principal

#include <algorithm>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

#include "arreglo.h"
#include "quicksort.h"
#include "tiempo.h"

int main(int argc, char* argv[])
{
    int *A;
      int i, n;
      double t;
      printf("Tamanio arreglo n: ");
      cin>>n;
      A=poblar(n,0);
      std::random_shuffle(A, A + n);

      t = tiempo();
      quicksort(A,0,n-1);
      t = tiempo() - t;

      cout << "T = "<< t/1000 << " segundos.\n";

      if(verificar_orden(n,A))
            cout << "Arreglo ordenado.\n";
      else
            cout << "Error.\n";
      return 0;
}
```

```
//Función quicksort. quicksort.h
int particion(int *A, int ini, int fin)
{
    int pivote = A[ini], indice = ini, temp, j;
    for (j = (ini+1); j <= fin; j++)
    {
        if (A[j] < pivote)
        {
            indice++;
            temp = A[indice];
            A[indice] = A[j];
            A[j] = temp;
        }
    }
    temp = A[ini];
    A[ini] = A[indice];
    A[indice] = temp;
    return indice;
}

void quicksort(int *A, int ini, int fin)
{
    int p, tam = fin - ini;
    if(tam>1)
    {
        p = particion(A, ini, fin);
        quicksort(A, ini, p);
        quicksort(A, p + 1, fin);
    }
}
```

Quicksort paralelo

```
//Función principal
#include <algorithm>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

#include "arreglo.h"
#include "cilkview.h"
#include "quicksort_paralelo.h"

cilk::cilkview cv;

int cilk_main(int argc, char* argv[])
{
    int *A;
      int i, n;
      double t;
      printf("Tamanio arreglo n: ");
      cin>>n;
      A=poblar(n,0);

      std::random_shuffle(A, A + n);

      cv.reset();
      cv.start();
      quicksort(A,0,n-1);
      cv.stop();
      t = (long)cv.accumulated_milliseconds();
      cout << "T = "<< t/1000 << " segundos.\n";

      if(verificar_orden(n,A))
            cout << "Arreglo ordenado.\n";
      else
            cout << "Error.\n";
      return 0;
}
```

```
//Función quicksort. quicksort_paralelo.h
int particion(int *A, int ini, int fin)
{
    int pivote = A[ini], indice = ini, temp, j;
    for (j = (ini+1); j <= fin; j++)
    {
        if (A[j] < pivote)
        {
            indice++;
            temp = A[indice];
            A[indice] = A[j];
            A[j] = temp;
        }
    }
    temp = A[ini];
    A[ini] = A[indice];
    A[indice] = temp;
    return indice;
}


void quicksort(int *A, int ini, int fin)
{
    int p, tam = fin - ini;
    if(tam>1)
    {
        p = particion(A, ini, fin);
        cilk_spawn quicksort(A, ini, p);
        quicksort(A, p + 1, fin);
        cilk_sync;
    }
}
```

Multiplicación de matrices secuencial

```
//Función principal.
int xA,yA,xB;
int main(char argv, char* argc[])
{
	int *A, *B, *C;
	int i;
	double t;
	//Datos iniciales para definir tamaño de las matrices.
	printf("xA/yB: ");
	cin>>xA;
	printf("yA: ");
	cin>>yA;
	printf("xB: ");
	cin>>xB;

	srand(time(0));
	//Inicialización de valores.
	A=poblar(xA*yA,false);
	B=poblar(xB*xA,false);
	C=poblar(yA*xB,true);

	//Multiplicación
	t = tiempo();
	multiplicar(A,B,C);
	t = tiempo() - t;
	printf("%.3f\n",t);
}
```

```
//Funciones auxiliares dentro del mismo archivo. Main.cpp
void multiplicar(int *A, int *B, int *C)
{
      int i, j, k;
      for(i = 0; i < yA; i++)
            for (j = 0; j < xB; ++j)
            {
                  for (k = 0; k < xA; ++k)
                  {
                        C[i*xB+j] += A[i*xA+k] * B[k*xB+j];
                  }
            }
}

int * poblar(int n, bool ceros)
{
   int *A = (int*) malloc(sizeof(int)*n);
   for (int i = 0; i < n; ++i)
   {
      if(ceros)
              A[i] = 0;
        else
              A[i] = rand() % MAX;
   }
   return A;
}
```

Multiplicación de matrices en paralelo

```
//Función principal.
int main(int argc, char** argv)
{
	int *A, *B, *C, memoria_M, n, bloque, bloque2;
	double t, t2;

	srand(time(0));
	cout << "Tamanio matrices (mayor a 63) n: ";
	cin >> n;
	bloque = n/64;
	bloque2 = bloque;

	memoria_M = sizeof(int)*n*n;
	// Reservar memoria en el dispositivo grafico.
	int* A_kernel;
	int* B_kernel;
	int* C_kernel;

	// Reservar memoria para las matrices e inicializar.
	A = poblar(memoria_M, n*n,false);
	B = poblar(memoria_M, n*n,false);
	C = (int*) malloc(memoria_M);

	//Reservar memoria en el dispositivo de procesemiento gráfico
	cudaMalloc((void**) &A_kernel, memoria_M);
	cudaMalloc((void**) &B_kernel, memoria_M);
	cudaMalloc((void**) &C_kernel, memoria_M);

	//Tiempo1 incluye transferencia de datos
	t=tiempo();
	// Copiar datos al dispositivo
	cudaMemcpy(A_kernel, A, memoria_M, cudaMemcpyHostToDevice);
	cudaMemcpy(B_kernel, B, memoria_M, cudaMemcpyHostToDevice);


	// Parametros del dispositivo grafico
	dim3 threads(bloque, bloque);
	dim3 grid(n / threads.x, n / threads.y);

	//Tiempo2 solo registra la multiplicación
	t2 = tiempo();
	multiplicacion<<< grid, threads >>>(C_kernel, A_kernel, B_kernel,
	                                    n, n, bloque2);
	t2 = tiempo() - t2;

	cudaMemcpy(C, C_kernel, memoria_M, cudaMemcpyDeviceToHost);
	t=tiempo() - t;

	printf("%.3f\t%.3f\n",t,t2);
	cudaFree(A);
	cudaFree(B);
	cudaFree(C);
}
```

```
//Bibliotecas y funciones auxiliares, en el mismo archivo que la función
//principal. main.cu}

#define MAX 10

__global__ void multiplicacion(int* C, int* A, int* B, int xA, int xB,
int bloque)
{
      int tx = blockIdx.x * bloque + threadIdx.x;
      int ty = blockIdx.y * bloque + threadIdx.y;
      int valorA;
      int valorB;
      int valorC = 0;

      for (int i = 0; i < xA; ++i)
      {
            //la multiplicacion define la fila y la suma define la
columna
            valorA = A[ty * xA + i];//equivale a [xA][i]
            valorB = B[i * xB + tx];//equivale a [xB][tx]
            valorC += valorA * valorB;
      }
      C[ty * xB + tx] = valorC;//equivale a [xB][tx]
}


int * poblar(int n, int m, bool ceros)
{
   int *A = (int*) malloc(n);
   for (int i = 0; i < m; ++i)
   {
      if(ceros)
              A[i] = 0;
        else
              A[i] = rand() % MAX;
   }
   return A;
}
```

Apendice C – Tablas de resultados

Búsqueda Binaria - Entrada ordenada				
	Entrada (n)			
Ejecución	1,000,000	10,000,000	100,000,000	300,000,000
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0
10	0	0	0	0
11	0	0	0	0
12	0	0	0	0
13	0	0	0	0
14	0	0	0	0
15	0	0	0	0
16	0	0	0	0
17	0	0	0	0
18	0	0	0	0
19	0	0	0	0
20	0	0	0	0
Promedio	0	0	0	0
Varianza	0	0	0	0
Moda	0	0	0	0
Mediana	0	0	0	0

Búsqueda Lineal - Entrada ordenada				
	Entrada (n)			
Ejecución	1,000,000	10,000,000	100,000,000	300,000,000
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0.00099993
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0.00099993
9	0	0	0	0
10	0	0	0	0
11	0	0	0	0
12	0	0	0	0
13	0	0	0	0
14	0	0	0	0.00098999
15	0	0	0	0
16	0	0	0	0
17	0	0	0	0
18	0	0	0	0
19	0	0	0	0
20	0	0	0	0
Promedio	0	0	0	0.00014949
Varianza	0	0	0	1.2664E-07
Moda	0	0	0	0
Mediana	0	0	0	0

Búsqueda Lineal - Entrada desordenada				
	Entrada (n)			
Ejecución	**1,000,000**	**10,000,000**	**100,000,000**	**300,000,000**
1	0.0010	0.0210	0.1060	0.6330
2	0.0030	0.0240	0.0110	0.8120
3	0.0030	0.0130	0.1300	0.0610
4	0.0020	0.0050	0.2480	0.7350
5	0.0010	0.0250	0.2510	0.5050
6	0.0030	0.0150	0.2410	0.7440
7	0.0020	0.0120	0.0690	0.2320
8	0.0000	0.0120	0.2390	0.5010
9	0.0020	0.0090	0.1040	0.3990
10	0.0020	0.0070	0.0250	0.6560
11	0.0020	0.0270	0.2170	0.2800
12	0.0020	0.0110	0.0330	0.1890
13	0.0010	0.0150	0.1450	0.5080
14	0.0020	0.0180	0.0710	0.4130
15	0.0000	0.0200	0.1760	0.5490
16	0.0070	0.0160	0.0940	0.7870
17	0.0030	0.0040	0.1140	0.7900
18	0.0080	0.0100	0.0630	0.5290
19	0.0010	0.0010	0.2190	0.2730
20	0.0010	0.0020	0.0680	0.4830
Promedio	0.0023	0.0134	0.1312	0.5040
Varianza	3.810E-06	5.453E-05	0.0062	0.0451
Moda	0.00200	0.01200	#N/A	#N/A
Mediana	0.0020	0.0125	0.1100	0.5065

Búsqueda Recursiva - Entrada ordenada				
	Entrada (n)			
Ejecución	**1,000,000**	**10,000,000**	**100,000,000**	**300,000,000**
1	0.028002	0.165009	1.63009	4.91628
2	0.0170009	0.16701	1.64509	4.91728
3	0.0170012	0.16801	1.6611	4.93228
4	0.0170009	0.163009	1.64209	4.90728
5	0.0170009	0.164009	1.64109	4.94928
6	0.0180011	0.16901	1.62909	4.95728
7	0.0170012	0.16601	1.63409	4.95028
8	0.0180011	0.166009	1.62709	4.94028
9	0.0180008	0.16601	1.63109	4.94128
10	0.0170009	0.164009	1.63109	4.97528
11	0.0180011	0.16401	1.66309	4.96128
12	0.0170012	0.165009	1.65409	4.92528
13	0.0170009	0.16801	1.64509	4.92628
14	0.018002	0.165009	1.6481	4.90628
15	0.0170012	0.170009	1.64009	4.90128
16	0.0170009	0.16601	1.64509	4.88928
17	0.0170009	0.166009	1.65909	4.89428
18	0.0170012	0.17101	1.6521	4.89228
19	0.0170009	0.17001	1.6531	4.89128
20	0.0170009	0.16801	1.65409	4.92828
Promedio	0.01780111	0.16655955	1.644292	4.92513
Varianza	5.66103E-06	4.8486E-06	0.0001208	0.00062343
Moda	0.0170009	0.165009	1.64509	#N/A
Mediana	0.0170012	0.16601	1.64509	4.92578

Búsqueda Recursiva - Entrada desordenada				
	Entrada (n)			
Ejecución	**1,000,000**	**10,000,000**	**100,000,000**	**300,000,000**
1	0.0170	0.1660	1.6291	4.9433
2	0.0170	0.1640	1.6421	4.9473
3	0.0170	0.1660	1.6501	4.9513
4	0.0170	0.1660	1.6581	5.0473
5	0.0170	0.1660	1.6521	4.9753
6	0.0170	0.1650	1.6511	4.9443
7	0.0170	0.1650	1.6491	4.9593
8	0.0170	0.1650	1.6481	4.9713
9	0.0170	0.1660	1.6491	4.9273
10	0.0170	0.1650	1.6461	4.9433
11	0.0170	0.1640	1.6301	5.0213
12	0.0170	0.1650	1.6321	4.9603
13	0.0170	0.1640	1.6491	5.0513
14	0.0170	0.1630	1.6361	5.1543
15	0.0160	0.1640	1.6351	5.2933
16	0.0160	0.1640	1.6361	5.1363
17	0.0160	0.1640	1.6321	4.9963
18	0.0170	0.1630	1.6321	5.0533
19	0.0170	0.1650	1.6321	5.1343
20	0.0170	0.1640	1.6551	5.0913
Promedio	0.0169	0.1647	1.6422	5.0251
Varianza	1.2749E-07	9.1018E-07	8.5151E-05	0.00875066
Moda	0.0170	0.1640	1.6321	4.9433
Mediana	0.0170	0.1650	1.6441	4.9858

Búsqueda Paralela -- n = 10,000,000				
	Procesadores (n)			
Ejecución	**1**	**2**	**3**	**4**
1	1.295	0.780	0.624	0.530
2	1.279	0.764	0.624	0.531
3	1.295	0.780	0.593	0.546
4	1.295	0.749	0.608	0.531
5	1.279	0.764	0.593	0.530
6	1.280	0.765	0.593	0.530
7	1.295	0.780	0.593	0.515
8	1.279	0.764	0.608	0.531
9	1.311	0.764	0.593	0.546
10	1.279	0.780	0.593	0.515
11	1.217	0.702	0.624	0.530
12	1.248	0.718	0.687	0.515
13	1.279	0.774	0.608	0.515
14	1.248	0.718	0.624	0.561
15	1.248	0.786	0.593	0.578
16	1.263	0.733	0.593	0.530
17	1.295	0.733	0.609	0.515
18	1.232	0.718	0.608	0.546
19	1.279	0.733	0.562	0.530
20	1.217	0.749	0.593	0.515
Promedio	1.2707	0.7527	0.6062	0.5320
Incremento	1	1.68812276	2.0962633	2.38843985
Eficiencia	1	0.84406138	0.69875443	0.59710996
Paralelismo	2.37			
Varianza	0.000697	0.000629	0.000567	0.00027
Moda	1.279	0.780	0.593	0.530
Mediana	1.279	0.764	0.601	0.530

Quicksort Secuencial				
	Entrada (n)			
Ejecución	**1,000,000**	**10,000,000**	**100,000,000**	**300,000,000**
1	0.205	2.256	26.507	85.3929
2	0.196	2.260	27.217	85.0249
3	0.193	2.303	26.545	86.2449
4	0.198	2.300	25.713	86.9125
5	0.194	2.394	25.559	85.7531
6	0.190	2.243	25.321	87.0368
7	0.193	2.296	26.159	85.3929
8	0.196	2.425	25.321	84.9824
9	0.195	2.271	27.013	85.0257
10	0.196	2.293	27.002	85.2684
11	0.198	2.288	26.277	84.9907
12	0.194	2.294	25.931	86.0397
13	0.197	2.290	26.593	84.3467
14	0.200	2.340	25.894	86.7997
15	0.201	2.275	26.814	85.8197
16	0.202	2.243	25.934	86.2496
17	0.206	2.246	26.162	86.0877
18	0.198	2.233	26.582	85.2667
19	0.199	2.277	26.347	85.9307
20	0.207	2.274	25.863	85.3147
Promedio	0.1979	2.2902	26.2373	85.6940
Varianza	1.959E-05	2.227E-03	2.887E-01	4.936E-01
Moda	0.196	2.2431	#N/A	85.3929
Mediana	0.198	2.2826	26.2190	85.5730

Quicksort Paralelo -- n = 10,000,000				
	Procesadores (n)			
Ejecución	**1**	**2**	**3**	**4**
1	3.058	1.716	1.388	1.216
2	3.058	1.716	1.42	1.248
3	3.089	1.794	1.404	1.264
4	3.557	1.763	1.419	1.248
5	3.104	1.763	1.419	1.248
6	3.198	1.778	1.42	1.217
7	3.089	1.887	1.42	1.232
8	3.151	1.809	1.435	1.295
9	3.104	1.81	1.404	1.248
10	3.12	1.747	1.404	1.264
11	3.192	1.685	1.373	1.264
12	3.198	1.716	1.373	1.233
13	3.245	1.747	1.404	1.232
14	3.220	1.816	1.404	1.179
15	3.229	1.747	1.389	1.248
16	3.183	1.778	1.389	1.232
17	3.008	1.700	1.514	1.233
18	3.029	1.800	1.404	1.232
19	3.160	1.747	1.389	1.217
20	3.129	1.716	1.498	1.264
Promedio	3.1561	1.7618	1.4135	1.2407
Incremento	1	1.79142898	2.23279094	2.54376562
Eficiencia	1	0.89571449	0.74426365	0.6359414
Paralelismo	2.9			
Varianza	0.012870	0.002259	0.001208	0.00057
Moda	3.058	1.716	1.404	1.248
Mediana	3.140	1.755	1.404	1.241

Multiplicación de Matrices Secuencial				
	Entrada (n)			
Ejecución	**128**	**256**	**512**	**1,024**
1	0.022	0.086	0.657	13.286
2	0.009	0.067	0.625	13.37
3	0.008	0.068	0.638	13.213
4	0.009	0.068	0.641	13.158
5	0.008	0.067	0.63	13.199
6	0.009	0.069	0.641	13.217
7	0.008	0.067	0.639	13.201
8	0.009	0.068	0.635	13.296
9	0.008	0.067	0.641	13.273
10	0.009	0.067	0.648	13.165
11	0.010	0.081	0.672	13.3130
12	0.009	0.070	0.661	13.2540
13	0.009	0.070	0.689	13.2230
14	0.009	0.070	0.835	13.2960
15	0.009	0.071	0.867	13.2160
16	0.010	0.070	0.663	13.4310
17	0.010	0.071	0.679	13.3310
18	0.009	0.070	0.684	13.2840
19	0.008	0.071	0.702	13.4910
20	0.009	0.071	0.644	13.7290
Promedio	0.0096	0.0705	0.6746	13.2973
Varianza	8.55E-06	2.17E-05	3.91E-03	1.66E-02
Moda	0.009	0.0670	0.6410	13.2960
Mediana	0.009	0.0700	0.6525	13.2785

Transferencia de datos y multiplicación						Multiplicación en GPU	
	Entrada (n)					Entrada (n)	
Ejecución	128	256	512	1024		512	1024
1	0.005	0.31	2.849	4.23		0	0
2	0.005	0.31	2.793	4.26		0	0.001
3	0.005	0.307	2.792	4.254		0	0.001
4	0.005	0.311	2.793	4.23		0	0
5	0.005	0.308	2.605	4.238		0	0
6	0.005	0.312	2.795	4.231		0	0
7	0.005	0.308	2.795	4.234		0	0
8	0	0.307	2.793	4.242		0	0.001
9	0.005	0.307	2.793	4.231		0	0
10	0	0.31	2.794	4.241		0	0
11	0	0.31	2.79	4.21		0	0
12	0	0.312	2.83	4.2		0	0
13	0.004	0.31	2.795	4.26		0	0
14	0.005	0.3	2.795	4.25		0	0
15	0	0.309	2.794	4.254		0	0.001
16	0	0.308	2.79	4.241		0	0
17	0.003	0.311	2.823	4.26		0	0
18	0.005	0.31	2.792	4.241		0	0
19	0.005	0.312	2.792	4.231		0	0.001
20	0.005	0.307	2.794	4.23		0	0
Promedio	0.004	0.309	2.7802	4.2391		0	0.0003
Varianza	5.03E-06	7.05E-06	2.03E-03	2.37E-04		0E+00	1.88E-07
Moda	0.005	0.3100	2.7930	4.2300		0.0000	0.0000
Mediana	0.005	0.3100	2.7935	4.2395		0.0000	0.0000

Referencias

[1] G. Moore, “Cramming more components onto integrated circuits,” *Proceedings of the IEEE*, vol. 86, no. 1, pags. 82–85, 1998.

[2] P. Pacheco, *An Introduction to Parallel Programming*, 1st ed. Morgan Kauffman, 2011

[3] D. Cornelis and B. V. D. Meulen, *Getting New Technologies Together: Studies in Making Sociotechnical Order*. Walter De Gruyter Inc, 1998, pags. 206–208.

[4] B. Burgstaller and B. Scholz, “The Shift to Multicore Architectures,” *University of Sydney*. pags. 1–24, 2011.

[5] H. Sutter, “The Free Lunch Is Over,” 2005. Disponible en: http://www.gotw.ca/publications/concurrency-ddj.htm. [Junio-2012].

[6] S. R. DAS, “A Single-Atom Transistor,” 2012. Disponible en: http://spectrum.ieee.org/semiconductors/nanotechnology/a-singleatom-transistor/. [Agosto-2012].

[7] J. F. Gantz, D. Reinsel, C. Chute, W. Schlichting, J. McArthur, S. Minton, I. Xheneti, A. Toncheva, and A. Manfrediz, “The Expanding Digital Universe: A Forecast of Worldwide Information Growth Through 2010,” 2007.

[8] T. Rauber and G. Rünger, *Parallel Programming: for Multicore and Clyster Systems*, 2da ed. Springer-Lehrbuch, 2010, pags. 7–91.

[9] G. Pfister, *In Search of Clusters*, 2da ed. Prentice Hall, 1997, pags. 32–37.

[10] J. Attwood, “Building a Computer the Google Way,” *Coding Horror*, 2007. Disponible en: http://www.codinghorror.com/blog/2007/03/building-a-computer-the-google-way.html [Septiembre-2012].

[11] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*, 1ra ed. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1994, pags. 16–24.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Boston, MA: The MIT Press, 2009.

[13] D. Helmbold, “Asymptotic Growth of Functions,” *University of California, Santa Cruz*, 2004. Disponible en: http://classes.soe.ucsc.edu/cmps102/Spring04/TantaloAsymp.pdf [Junio-2012].

[14] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *Proceedings of the April 1820 1967 spring joint computer conference*, vol. 30, no. 3, pags. 483–485, 1967.

[15] MIT, "Cilk 5.4.6 Reference Manual." Computer Science Lab, Supercomputer Technologies Group, Boston, MA, pags. 56, 1998.

[16] C. E. Leiserson, "Multithreaded Programming in Cilk 1,"Computer Science Lab, Supercomputer Technologies Group, 2006. Disponible en: http://supertech.csail.mit.edu/cilk/lecture-1.pdf [Agosto 2012].

[17] MIT, "The Cilk Project," Computer Science Lab, Supercomputer Technologies Group, 2000. Disponible en: http://supertech.csail.mit.edu/cilk/ [Agosto 2012].

[18] Intel, "Intel Cilk ++ SDK Programmer ' s Guide," 2009. Disponible en: http://software.intel.com/sites/default/files/m/7/e/2/8/6/23634-Cilk-Programmers-Guide.pdf [Agosto 2012].

[19] A. Tumeo, "Computing with CUDA." NVIDIA, Milan, pags. 1–32, 2008.

[20] J. Sanders and E. Kandrot, *Cuda by example*, 1st ed. Ann Arbor, MI: Addison-Wesley, 2011, pags. 290.

[21] R. Membarth, "CUDA Parallel Programming Introduction." University of Erlangen-Nuremberg, pags. 1–26, 2009.

[22] Intel, "Parallelism for the Masses : Opportunities and Challenges," in *Parallel Thinking Seminar*, 2008, pags. 1–41.

[23] J. Bentley, *Programming Pearls*, 2da ed. Addison-Wesley, 2000, pags. 5–15.