UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO

FACULTAD DE CIENCIAS FÍSICO MATEMÁTICAS

"Mat. Luis Manuel Rivera Gutiérrez"

ESTUDIO NUMÉRICO DE LA PROGRAMACIÓN EN PARALELO USANDO EL PROTOCOLO CUDA FORTRAN

TESIS

Para obtener el grado de

LICENCIADO EN CIENCIAS FÍSICO MATEMÁTICAS

PRESENTA:

SERGIO SÁNCHEZ LÓPEZ

ASESOR DE TESIS:

Doctor en Ciencias en Óptica
HÉCTOR I. PÉREZ AGUILAR

Morelia, Michoacán, Marzo de 2016

RESUMEN

El estudio de sistemas complejos en 3D o a gran escala demanda grandes recursos computacionales, que hasta hace poco tiempo sólo era posible realizar con grandes y costosas computadoras (clústers). Esta situación ha ido cambiando con la creación de las tarjetas de procesamiento gráfico (GPUs) que se pueden integrar en una computadora de escritorio e incluso en una portátil. Las tarjetas con el protocolo CUDA (Computer Unified Device Architecture) permiten la programación en paralelo, lo cual implica un gran ahorro de tiempo de cómputo para estudiar las diferentes propiedades y características de los sistemas complejos en diversas situaciones a gran escala. Esto es posible porque la programación en paralelo usa simultáneamente múltiples elementos de procesamiento para resolver un problema. Es decir, se consigue dividiendo el problema en partes independientes para que cada elemento de procesamiento pueda ejecutar su parte del algoritmo simultáneamente con los otros. El objetivo principal de este trabajo, es estudiar y explorar la enorme capacidad de cálculo de las GPUs de NVidia, mediante el modelo de programación en paralelo con CUDA FORTRAN. Hemos aplicado esta herramienta numérica en la implementación de operaciones básicas de álgebra lineal, como son: la suma, la multiplicación y la inversión de matrices para la solución de sistemas de ecuaciones lineales. Dentro de las diferentes formas de programación en paralelo que estudiamos, la forma más óptima para reducir el tiempo de cómputo considerablemente, es por medio de las librerías internas de CUBLAS (CUDA Basic Linear Algebra Subroutines) y CULA (CUDA Linear Algebra) que están optimizadas. Esto muestra que la arquitectura de cálculo paralelo CUDA es una gran herramienta numérica que permite modelar sistemas complejos en diferentes líneas de investigación.

Palabras clave: Programación en paralelo, GPU, CUDA FORTRAN, CUBLAS y CULA.

ABSTRACT

The study of complex systems in 3D or a large-scale demands substantial computational resources, that until recently was only possible with large and expensive computers (clusters). This situation has changed with the creation of graphics processing cards (GPUs), that can be integrated into a desktop computer or even into a laptop. Cards with the CUDA (Computer Unified Device Architecture) protocol allow parallel programming, which implies a great time-saving of the computation for studying the different properties and characteristics of the complex systems in several situations on a large scale. This is possible because parallel programming uses multiple processing elements simultaneously to solve a problem. That is to say, it is achieved by dividing the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The main objective of this work is to study and explore the massive computation capability of NVidia GPUs, using the model of parallel programming with CUDA FORTRAN. We have applied this numerical tool in the implementation of basic linear algebra operations, such as: addition, multiplication and inversion of matrices for solving systems of linear equations. Among the different ways of parallel programming that we studied, the most optimal way to reduce the computation time considerably, it is through internal libraries CUBLAS (CUDA Basic Linear Algebra Subroutines) and CULA (CUDA Linear Algebra) that are optimized. This demonstrates that the CUDA parallel computing platform is a substantial numerical tool that allows to model complex systems in different research areas.

Keywords: Parallel programing, GPU, CUDA FORTRAN, CUBLAS and CULA.

Dedicatorias

Al creador de todas las cosas, el que me ha acompañado y protegido siempre; por ello, con toda la humildad que de mi corazón puede emanar, dedico primeramente mi trabajo a Dios.

De igual forma, dedico este trabajo de tesis a mis padres Francisco Sánchez Gómez y Pascuala López Jimenez, por haberme dado el regalo de la vida, educarme y apoyarme en los momentos y circunstancias difíciles de mi vida.

También dedico esta tesis a mis hermanos Hugo y Sandra, a quienes quiero mucho y que siempre me han apoyado. Y por último, también quiero dedicarle este trabajo a mi tio Lorenzo Sánchez Gómez, por sus consejos recibidos en el transcurso de mi formación.

Fraternalmente:

Sergio Sánchez López

Agradecimientos

El presente trabajo fue posible llevarlo a cabo, gracias a su arduo apoyo, a su incansable paciencia del Dr. Héctor Pérez Aguilar, quien con sus conocimientos, experiencia, correcciones y aportaciones dedicados a la realización de esta tesis se llegó hasta la meta; es por ello, que les doy mi más amplia gratitud.

A la coordinación de Investigación Científica de la UMSNH que me apoyo con una beca del proyecto de investigación 9:33.

Agradezco en forma especial a los integrantes de mi comité, quienes me hicieron las correcciones y observaciones de mi trabajo de tesis, me refiero al Dr. Jorge Isidro Aranda Sánchez, Dra. Karina Mariela Figueroa, M.C. José Vega Cabrera y al M.C. Gabriel Arroyo, quienes cordialmente aceptaron colaborar para la revisión de dicho trabajo y llegar hasta su objetivo final.

A la institución de educación, la Facultad de Ciencias Físico-Matemáticas, donde en su espacio fue mi segunda casa y fue la fuente donde adquirí los conocimientos más elementales para poder desarrollarme como profesionista y servir a la sociedad con principios y valores, es por ello mi agradecimiento total a la máxima casa de estudios de nuestro estado, la U.M.S.N.H.

También deseo agradecer con mucho afecto y especial cariño a mi familia, padres, hermanos y demás familiares más cercanos quienes me dieron una palabra de aliento y me motivaron a seguir con mucho empeño en mis estudios en la Primaria, Secundaria, Preparatoria y Licenciatura; para ellos, de corazón mi eterno agradecimiento.

De igual forma agradezco a mis amigos y compañeros de la carrera: Luis Eduardo, José Eduardo, Nanci, Dante, Juan José y Uriel por todos esos momentos que hemos vivido a lo largo de esta época y los que quedan por vivir, entre otros.

Y también quiero agradecer a todos mis amigos que he conocido a lo largo de mi vida. En especial a Eligio, Jazmin, Gricelda y Paty, porque gracias a su apoyo, compañia y buenos consejos dia a dia motivaron mis deseos por avanzar hasta culminar este trabajo.

Sinceramente:

Sergio Sánchez López

Contenido

Página

Contenido (continuación)

Página

Lista de Figuras

Lista de Figuras (continuación)

Capítulo I

INTRODUCCIÓN

Con el surgimiento de las computadoras, se comenzó una era de progresos significativos en todas las áreas sociales. La computación ha sido una herramienta indispensable por su fiabilidad y rapidez operativa de datos, la cual está involucrada desde ambientes estudiantiles hasta la conquista del espacio. Cada año las mejoras en los componentes de las computadoras mejoran estas cualidades.

Una persona que tardaría horas en realizar cálculos, ahora son resueltos por una computadora en segundos. Sin embargo, actualmente existen problemas que demandan todavía mucho más tiempo de procesamiento en la computadora; por ejemplo la simulación del clima (usada para la predicción y el estudio del cambio climático) o la simulación de mecánica de fuidos (usada en el desarrollo de aviones), entre otros. Solucionar estos problemas pueden tardar desde minutos hasta días o meses.

Desde sus inicios, la solución para resolver problemas que involucran mucho recurso computacional, fue fabricar computadoras con procesadores cada vez más rápidos, pero con el paso de los años, esto parece alcanzar límites físicos (Sutter, 2005) que impedían continuar con esta tendencia. Por ejemplo, los fabricantes de microprocesadores, como Intel o AMD (Kirk y Hwu, 2012), incrementaron el rendimiento de las aplicaciones

durante más de dos décadas. Estos microprocesadores alcanzaban varios gigaflops (GFlops[1]) de cómputo en los equipos de escritorio y cientos de GFlops de cómputo en los servidores de un clúster.

Durante este periodo, los desarrolladores de software contaron con estas mejoras en el hardware para mejorar el rendimiento de sus aplicaciones de forma trasparente. La aplicación simplemente se ejecutaba más rápido en cada nueva generación de microprocesadores. Sin embargo, este modelo de mejora del hardware se vio limitado a principios de 2003 (Sutter, 2005), ya que debidos a problemas de disipación de calor y de altos niveles de consumos de energía ya no fue posible continuar aumentando la frecuencia de reloj a la que trabajaban sus productos. Por estas razones, se lanzaron a la búsqueda de nuevas formas de incrementar la potencia computacional de sus desarrollos, creando finalmente lo que hoy en día es algo habitual para todos: los procesadores multinúcleo (multicore).

Con estos cambios han producido un cambio de paradigma en la manera en que los programadores desarrollan el software. Así de esta forma se inicializa el concepto de paralelización y actualmente se han desarrollado computadoras paralelas y viables comercialmente con decenas, o cientos o hasta miles de núcleos o procesadores.

La paralelización consiste en la subdivisión de un problema, en subproblemas, cada uno de los cuales es resuelto de forma simultánea y por separado, ofreciendo reducciones importantes en cuanto al tiempo de ejecución (Aguilar y Leiss, 2004). Sin embargo, no todos los problemas son aptos para el empleo de la paralelización, y los que sí lo son, no tienen porqué responder de forma positiva a su resolución paralela. Hoy en día existen diferentes he-

rramientas para el desarrollo de la programación en paralelo que hace que esta actividad

[1]Por sus siglas en inglés, Giga Floating point operations per second.

se vuelva sencilla, obteniendo hardware y software que reduce el tiempo de ejecución considerablemente.

Hoy en dia se cuenta con muchas arquitecturas de cómputo de alto desempeño para la programación en paralelo (como las supercomputadoras, el clúster, etc.), que aunque reducen el tiempo de cómputo, éstas en muchas ocasiones son inviables por su alto costo. Esta situación ha motivado el estudio del uso de hardware secundario; en particular, las **Unidades de Procesamiento Gráficos** (GPUs[2]), que potencialmente ofrecen una capacidad importante de cómputo y tienen un costo asociado relativamente reducido. Actualmente en el mercado se cuenta con una tecnología para la implementación de la programación en paralelo que utiliza GPUs con el protocolo **CUDA**[3]. Esta tecnología, mejora considerablemente los tiempos de ejecución en paralelo.

I.1. Aplicaciones del protocolo CUDA

Desde su aparición, a principios de 2007, una diversidad de empresas y aplicaciones han tenido gran éxito por haber utilizado el protocolo CUDA. Estos beneficios a menudo incluyen órdenes de aumento de rendimiento en los últimos estados de la implementación de la técnica. Algunos ejemplos de las aplicaciones (Sanders y Kandrot, 2010) se mencionan enseguida.

Imágenes médicas

El número de personas que han sido afectados por el cáncer de mama, ha ido en aumento en los últimos 20 años. Gracias en gran parte a los esfuerzos de muchos, la concientización e investigación de la prevención y cura de esta enfermedad, también ha aumentado en los últimos años. Finalmente, todos los casos de cáncer de mama que se

[2]Por sus siglas en inglés, Graphics Processor Unit.

[3]Por sus siglas en inglés, Compute Unified Device Architecture.

detectan a tiempo, son suficientes para prevenir los efectos destructores de la radiación y quimioterapia, las cicatrices que dejan las operaciones y las consecuencias severas de los casos que no responden a los tratamientos. Como resultado, investigadores como Sanders (Sanders y Kandrot, 2010) comparten un interés fuerte de encontrar rápida y precisa los primeros signos de cáncer de mama.

La mamografía, uno de los mejores métodos para la detección temprana del cáncer de mama, tiene significativas limitaciones. Dos o más imágenes necesitan tomarse y la película debe ser desarrollada y leída por un médico calificado para identificar los potenciales tumores. Adicionalmente, el proceso de rayos X, implica los riesgos de que las repetidas radiaciones queden en el pecho del paciente. Después de un estudio minucioso, los médicos suelen necesitar más información, imágenes más precisas y en ocasiones, hasta una biopsia en un intento de eliminar la posibilidad del cáncer. Estos resultados errados positivos incurren en costosos trabajos de seguimiento que conllevan a crear un estrés innecesario en el paciente, hasta que se puedan sacar las conclusiones finales.

Las imágenes de ultrasonido son más seguras que las imágenes por rayos X, por lo que los médicos optan por usar esto en conjunto con mamografías para ayudarse en el cuidado y diagnóstico del cáncer. Pero los convencionales ultrasonidos, también tienen sus limitaciones. Como resultado de esto nacen los "Sistemas médicos TechniScan" (ver Fig. 1). TechniScan ha desarrollado un prometedor método de ultra sonido tridimensional; sin embargo, esta solución no puede ser puesta en práctica, por una simple razón: limitaciones computacionales (cálculo). En pocas palabras, la conversión de los datos del ultrasonido, se concentran en el cálculo de imágenes en tres dimensiones y el mismo requiere tiempo; además que resulta costoso para el uso práctico.

La introducción de la primera GPU, basada en arquitectura CUDA de NVidia provee

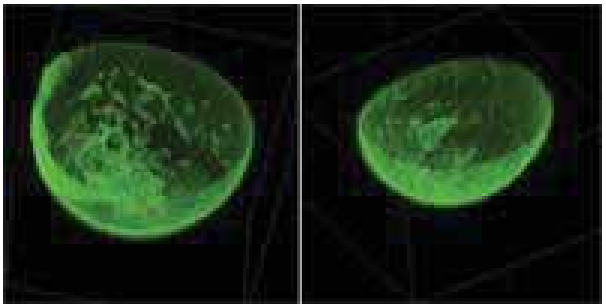

Figura 1. Rendizado volumétrico del sistema del ultrasonido para la mama completa de TechniScan.

una plataforma en la que TechniScan podría transformar los sueños de sus fundadores en la realidad. Como su nombre lo indica, su sistema de imagen ultrasonido Svara, usa ondas ultrasónicas para la imagen del pecho del paciente. El sistema Svara de TechniScan, se basa en dos procesadores Tesla C1060 de NVidia, con el fin de procesar 35 gigabytes (GB) de datos generados por un reconocimiento de 15 minutos. Gracias a la potencia computacional del modelo Tesla C1060, con sólo 20 minutos basta para que el doctor pueda manipular a gran nivel, la imagen tridimensional del cáncer de mama en la mujer.

Sector de energía

El costo de la perforación en la exploración profunda de pozos de petróleo puede llegar a cientos de millones de dólares. En muchos casos, existe apenas una posibilidad de perforar un pozo con éxito. La tecnología de imágenes en profundidad basada en CUDA de SeismicCity (ver Fig. 2) interpreta datos sísmicos que llevan a la selección de nuevas ubicaciones para la perforación con mucho más rapidez de lo que podrían hacerlo los sistemas anteriores.

Para mejorar la calidad y eficiencia de sus imágenes SeismicCity se inclinó por CUDA y la GPU de NVidia Tesla de la serie 8. Con ello se logró un aumento de hasta 14 veces en el rendimiento con relación a la configuración anterior basada en la CPU.

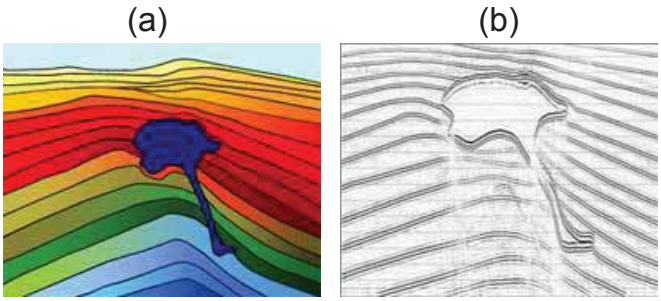

Figura 2. (a) Modelo geológico de la subsuperficie que muestra objetivos de la exploración del cuerpo salino. (b) Imagen sismica mediante la aplicación de tecnología de imágenes en profundidad.

Dinámica de fluido computacional

Por algunos años, el diseño de rotores y cuchillas eficientemente altos, sigue siendo un arte oscuro. El movimiento asombroso del aire y los fluidos alrededor de estos dispositivos o maquinarias, no pueden ser eficientemente modelados por formulaciones simples o tradicionales; así, las simulaciones precisas resultan demasiado costosas computacionalmente hablando. Solamente las grandes supercomputadoras en el mundo, podían usar recursos de cómputo a la par de sistemas modernos numéricos requeridos para desarrollar y validar los diseños. Como son pocos los que tienen acceso a este tipo de máquinas, la innovación en el diseño de estas máquinas seguía paralizado.

El Dr. Graham Pullan y el estudiante de Doctorado Tobias Brandvik (Brandvik y Pullan, 2010) del grupo "many-core", han identificado correctamente el potencial de la arquitectura CUDA para acelerar la dinámica de fluidos sin precedentes. Sus primeras investigaciones indicaron que los niveles aceptables de rendimiento sobre el potencial de la GPU, se podían realizar en estaciones de trabajo personales. Después, el uso de un pequeño clúster de GPU, superó fácilmente a las supercomputadoras costosas y además, ratificó sus sospechas de que las capacidades de las GPU de NVidia, coincidían con los problemas que pretendían resolver.

Para los investigadores de la Universidad de Cambridge, las mejoras masivas de

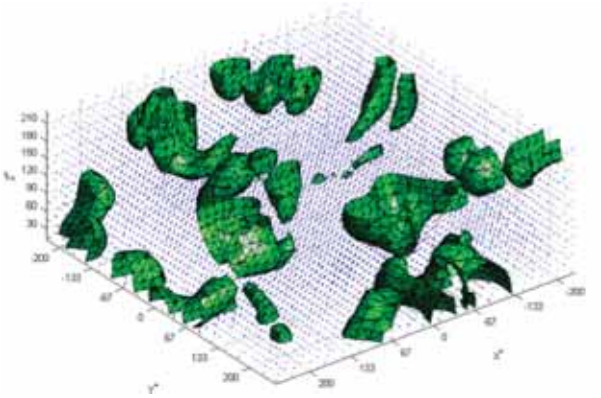

Figura 3. Estructuras de un fluido en una región 3D (medido con PIV tomográfico) implementado con CUDA.

rendimiento que ofrecía C para CUDA representan más que simple aumento progresivo en sus recursos de supercomputación. La disponibilidad del cómputo GPU por su bajo costo, facilita las investigaciones en Cambridge para llevar a cabo una rápida experimentación. Recibiendo resultados experimentales con segundos de diferencia entre cada uno, el proceso de retroalimentación en que los investigadores se basan, ayudan a llegar a grandes avances. Como resultado, el uso de clústers de GPUs ha transformado fundamentalmente el enfoque de sus investigaciones. Casi la simulación interactiva ha desencadenado nuevas oportunidades para innovar y crear un nuevo campo de investigación.

El interés de este presente trabajo de tesis, es estudiar y mostrar las capacidades de cómputo que posee la GPU de NVidia para solucionar problemas que pueden ser tratados por métodos en forma matricial o de cualquier sistema complejo; donde requieren procesamiento de una gran cantidad de datos. En particular, implementaremos los problemas básicos de álgebra lineal, como son: la suma, multiplicación e inversión de matrices; asi como también la solución de sistemas de ecuaciones lineales.

I.2. Estructura de la tesis

Este trabajo de tesis está estructurado en seis capítulos que describiremos a continuación.

En el capítulo II nos centraremos en la programación en paralelo y los diferentes aspectos a tener en cuenta para desarrollar programas con arquitecturas en paralelo. De esta forma se pretende explicar conceptos importantes que servirán para el presente trabajo.

En el capítulo III se describen las características de la arquitectura de la GPU de NVidia y también se da una breve historia de las GPUs. Y finalmente, se describen algunos aspectos importantes a tener en cuenta a la hora de solucionar problemas cómputacionalmente; donde requieran uso ya sea de CPU o GPU.

En el capítulo IV se describen los lineamientos básicos del modelo de programación con CUDA FORTRAN. También hablaremos de la organización de los hilos y su ejecución en el kernel. Por último iniciaremos con la programación paralela, describiendo la forma de compilar y ejecutar un programa con CUDA FORTRAN. Posteriormente mostraremos un ejemplo sencillo, explicando los pasos llevados a cabo.

En el capítulo V se muestran los resultados obtenidos bajo la implementación de los programas en paralelo y secuencial en las aplicaciones básicas de álgebra lineal.

En el capítulo VI se dan las conclusiones principales de esta tesis.

Capítulo II

PROGRAMACIÓN EN PARALELO

En este capítulo nos centraremos en la definición y términos usados en la programación en paralelo, con el fin de ser capaces de entender correctamente el contexto con el protocolo CUDA. En particular se presentan los tipos de arquitectura de MPI y OpenMP.

II.1. Definición de programación en paralelo

La programación en paralelo es en la actualidad un área de investigación motivada por diversos factores. El objetivo primario de la programación en paralelo es mejorar la velocidad de cálculo numérico.

Desde una perspectiva de cálculo, la computación en paralelo es una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente. Se basa en el principio de que los problemas grandes se pueden dividir en partes más pequeñas que pueden resolverse de forma concurrente (en paralelo).

Por otro lado, desde la perspectiva del programador, la computación en paralelo se puede definir como el uso simultáneo de múltiples recursos de computación (núcleos o procesadores) para realizar los cálculos simultáneos. Los aspectos software y hardware de la computación paralela están estrechamente entrelazadas. De hecho, la computación

en paralelo por lo general implica dos áreas distintas de la tecnología computacional:

- Arquitectura de la computadora (aspecto hardware).
- Programación en paralelo (aspecto software).

La **arquitectura de la computadora** se centra en apoyar el paralelismo a nivel de arquitectura, mientras que la **programación en paralelo** se centra en resolver un problema simultáneamente por completo, usando la potencia de cálculo de la arquitectura de las computadoras. A fin de lograr la ejecución paralela en el software, el hardware debe proporcionar una plataforma que admite la ejecución simultánea de varios procesos o múltiples subprocesos.

II.2. Breve historia de la programación en paralelo

La historia de la computación en paralelo está empalmada con la propia historia de la computación en general. Si cierto es que los orígenes de la computación ya se remontan a tiempos remotos; por ejemplo, con el invento del ábaco. En el caso de la computación en paralelo sus inicios se podría establecer en torno a mediados del siglo XX con el desarrollo de la primera computadora de segunda generación transitoria.

En los años 50 ocurrieron los primeros intentos de construcción de máquinas paralelas, pero es hasta los años 60-70 se construyeron las primeras (Aguilar y Leiss, 2004). La mayoría de esas máquinas eran máquinas vectoriales mono-procesadoras. A partir de los 70s surgieron las máquinas vectoriales multiprocesadoras. Todas disponían de una memoria compartida, pero eran muy caras.

En 1950 von Neumann considera analogías entre computadora-cerebro y propone un modelo de cálculo en paralelo. En 1958, Gill escribió sobre las bases de la programación en paralelo y un año después Holland planteó la posibilidad de ejecutar un

número determinado de programas simultáneamente. En 1963 Conway, describe el diseño de una computadora en paralelo para su programación. Pero no fue hasta 1981 cuando se presentó el primer sistema paralelo comercial, "el Butterfly", distribuido por BBN Computers Advanced. Este sistema era capaz de dividir su trabajo entre 256 procesadores, en especial microprocesadores Motorola 68000, que se conectaban a través de una red multi-etapa con memoria de 500 Kbytes por procesador. Se llegaron a vender 35 máquinas, la mayoría a universidades y centros de investigación.

En los años ochenta y principios de los noventa comenzó la denominada era de la época dorada de la programación en paralelo; particularmente, en el paralelismo a nivel de datos. Entre las arquitecturas más destacadas cabe citar la Connection Machine, MasPar y Cray; así como verdaderas supercomputadoras increíblemente poderosas y muy costosas. Esto derivó en lo que se denominó época oscura de la computación en paralelo, puesto que fue difícil vender estos equipos tan potentes.

En los últimos años el rendimiento de los computadoras en paralelo ha aumentado significativamente. El último fenómeno es la democratización de la programación en paralelo, con la llegada de las GPUs multinúcleo a los hogares (Jiménez de Parga, 2011).

II.3. Tipos de Paralelismo

Hoy en día, el paralelismo está en todas partes, y la programación en paralelo se está convirtiendo en la corriente principal en el mundo de la programación. Paralelismo en múltiples niveles es la fuerza motriz de diseño de la arquitectura. Hay dos tipos fundamentales de paralelismo en aplicaciones:

- Paralelismo de tareas.
- Paralelismo de datos.

El Paralelismo de tarea consiste en aplicar simultáneamente diferentes operaciones a distintos elementos de datos. El paralelismo de datos se caracteriza por la ejecución paralela de la misma operación sobre distintos datos. En otras palabras, múltiples unidades funcionales aplican simultáneamente la misma operación a un subconjunto de elementos.

En la programación bajo el protocolo CUDA es especialmente adecuado para abordar los problemas que se pueden expresar como cómputo de datos en paralelo (Cheng *et al.*, 2014). El primer paso en el diseño de un programa en paralelo de datos es dividir los datos a través de los subprocesos, con cada subproceso trabajando en una porción de los datos. Este caso de describirá con mayor detalle en el capítulo III.

II.4. Conceptos y terminologías de la programación en paralelo

La programación en paralelo tiene su propio lenguaje. Algunos de los términos más utilizados se describen a continuación.

Tarea:

Sección lógica discreta de trabajo computacional. Típicamente es un programa o conjunto de instrucciones que se ejecutan por un procesador.

Tarea paralela:

Tarea que puede ser ejecutada de forma segura por múltiples procesadores simultáneamente.

Ejecución en serie:

Ejecución de un programa de forma secuencial; es decir, una expresión en cada instante de tiempo. No obstante, todas las tareas paralelas tendrán secciones de un

programa paralelo que deben ser ejecutadas en serie.

Algoritmos paralelos:

Son métodos para resolver problemas computacionales en computadoras paralelizables.

Ejecución en paralelo:

Ejecución de un programa por más de una tarea, siendo cada tarea capaz de ejecutar la misma expresión o una diferente, y todas en un mismo instante de tiempo.

Memoria compartida:

Desde un punto de vista estrictamente de hardware, describe una arquitectura de equipo donde todos los procesadores tienen acceso directo a la memoria física común. En el sentido del software, describe un modelo donde todas las tareas paralelas tienen la misma representación de la memoria que pueden direccionar y acceder directamente a las mismas localizaciones de una memoria lógica, sin preocuparse donde se encuentra la memoria física.

Comunicaciones:

Tradicionalmente, las tareas paralelas necesitan intercambiar datos. Hay varias formas en que esto se puede lograr, tales como tener una memoria compartida en bus o sobre una red. Sin embargo, en la actualidad, al hecho de intercambiar datos se le denomina comunicaciones, independientemente del método empleado.

Sincronización:

Es la coordinación de tareas paralelas en tiempo real, a menudo asociado a las comunicaciones. La forma en que se implementa la sincronización entre tareas es poniendo un punto de sincronismo dentro del código de una aplicación, de modo que las tareas no continuarán con su trabajo hasta que todas las tareas hayan llegado al mismo punto de sincronismo. La sincronización implica la espera de al menos una tarea y, por lo

tanto, puede causar el incremento del tiempo de ejecución de la aplicación paralela.

Granularidad:

La Granularidad puede ser definida como la relación entre el tiempo requerido por una operación de comunicación básica y el tiempo requerido por un cómputo básico.

Una granularidad pequeña (grano fino) implica más comunicación y cambios de contextos entre las tareas (es una aplicación con comunicación intensiva). Una granularidad grande (grano fuerte) implica menos comunicación, pero puede que potenciales tareas concurrentes queden agrupadas, y por consiguiente, ejecutadas secuencialmente.

Concurrencia:

Define la ejecución simultánea de dos o más procesos en uno o más procesadores.

Escalabilidad:

Capacidad de un sistema paralelo (hardware o software) para demostrar un incremento proporcional en la velocidad de ejecución en paralelo con el aumento del número de procesadores. Un factor importante que influye en la escalabilidad es la forma en que se ha diseñado el código de la aplicación paralela a ejecutar.

II.5. Arquitecturas paralelas

En esta sección dedicaremos un breve estudio a las arquitecturas paralelas. En primer lugar, en la subsección II.5.1 revisaremos la clasificación de las computadoras llevadas a cabo por Flynn. En la subsección II.5.2 dividiremos las máquinas paralelas en dos tipos básicos atendiendo a la configuración de la memoria: las computadoras paralelas de memoria compartida y las computadoras paralelas de memoria distribuida.

II.5.1. Tipos de arquitectura

Para clasificar las arquitecturas paralelas existen diferentes clasificaciones, una de las más utilizadas es la taxonomía de Michael J. Flynn quien lo propuso en 1966 (Hennessy y Patterson, 2012), y se basa en el flujo que siguen los datos dentro de la máquina y de las instrucciones sobre esos datos.

Esta clasificación distingue arquitecturas multiprocesadores de acuerdo con dos dimensiones independientes: datos e instrucciones, donde cada una de estas dimensiones puede tener sólo dos posibles estados: simple (single) y múltiple (multiple). Un estudio más extenso sobre la clasificación de las computadoras y diversos aspectos referentes a la arquitectura de computadoras puede encontrarse en Quinn (2004).

Arquitectura tipo SISD (Single Instruction Single Data)

Este tipo de arquitectura corresponde a la idea de una única instrucción ejecutándose sobre un único conjunto de datos. Es una arquitectura secuencial; es decir, las instrucciones son ejecutadas unas detrás de otras.

Arquitectura tipo SIMD (Single Instruction Multiple Data)

A diferencia del SISD, aquí existen múltiples procesadores que sincronizadamente ejecutan la misma secuencia de instrucciones, pero en diferentes datos.

Arquitectura tipo MIMD (Multiple Instruction Multiple Data)

En este tipo de computadora, distintas instrucciones se ejecutan sobre los distintos conjuntos de datos. En este tipo de máquinas, cada elemento de proceso ejecuta su propio juego de instrucciones sobre los datos que contienen. Existen, por tanto, vari—os elementos de proceso y cada uno con su propia unidad de control, con lo que el funcionamiento del sistema es asíncrono.

II.5.2. Modelo de acceso a la memoria

Atendiendo a la configuración de la memoria, se puede establecer una primera clasificación básica de las computadoras paralelas en dos clases: compartida y distribuida. Es importante mencionar que en este trabajo estamos interesados en la memoria compartida.

Memoria compartida

En este modelo, una máquina paralela con memoria compartida (una computadora con muchos CPUs o núcleos que acceden al mismo espacio de memoria), los mensajes pueden ser enviados depositando sus contenidos sobre un área de memoria compartida, ver Fig. 24.

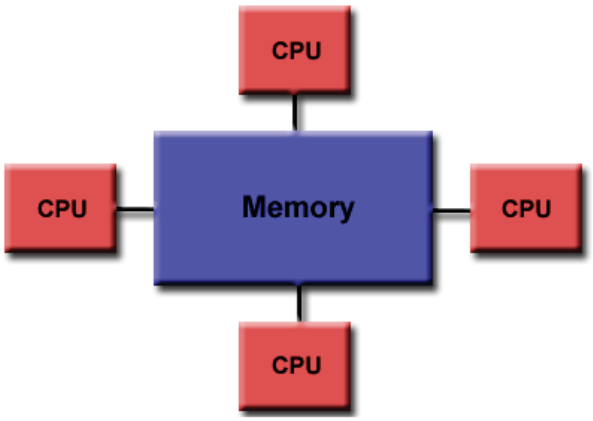

Figura 4. Memoria compartida por un conjunto de procesadores.

Una de las ventajas del método de memoria compartida es que el espacio de direccionamiento global presenta un fácil manejo desde el punto de vista del programador a la hora de acceder a memoria. Además, la compartición de datos entre tareas es rápido y uniforme, debido a la proximidad de la memoria a las CPUs. Por el contrario, una gran desventaja que presenta este sistema es que al añadir más CPUs incrementa de forma geométrica el tráfico asociado con la gestión de la memoria.

Memoria distribuida

En este tipo de máquina, la memoria está físicamente distribuida entre los procesadores, cada memoria local es accesible directamente sólo por su procesador, no existiendo una memoria global común. La comunicación entre los distintos procesadores debe realizarse mediante paso de mensajes.

Una computadora con memoria distribuida como se muestra en la Fig. 5, consta de una colección de ordenadores independientes llamados nodos. Cada nodo inicia su propio programa y se comunica con los otros nodos, enviando y recibiendo mensajes, utilizando las rutinas enviar/recibir destinadas a este propósito.

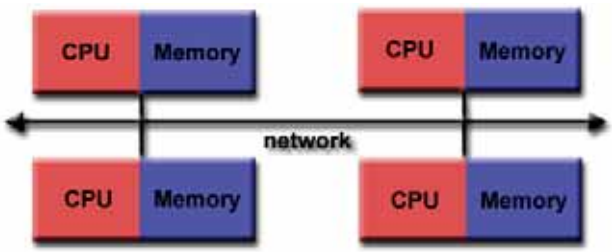

Figura 5. Sistema de memoria distribuida.

Este sistema, a diferencia del anterior (sistema de memoria compartida), sí es escalable en cuanto a lo que a memoria se refiere a la hora de aumentar el número de procesadores. Cada procesador puede acceder rápidamente a su propia memoria sin ninguna interfaz y sin ningún coste operativo incurrido, siempre manteniendo la coherencia de la caché.

II.6. Plataformas de programación en paralelo

Los lenguajes de programación en paralelo, las bibliotecas, las APIs[1] y los modelos de programación en paralelo han sido creados para la programación de computadoras con

[1]Por sus siglas en inglés, Application Programming Interfaces.

arquitectura en paralelo. Estos generalmente se pueden dividir en clases basadas en las suposiciones que se hacen sobre la arquitectura de memoria subyacente: compartida, distribuida, o compartida-distribuida. Los lenguajes de programación de memoria compartida se comunican mediante la manipulación de variables en la memoria compartida. En cambio, en la arquitectura con memoria distribuida se utiliza el paso de mensajes.

Hay muchos lenguajes y modelos de programación en paralelo propuestos en las últimas décadas. Los que son más utilizados son **Message Passing Interface** (MPI) para sistemas con memoria distribuida y **OpenMP** para sistemas con memoria compartida. Ambos se han convertido en las interfaces de programación estandarizadas y apoyadas por los principales vendedores de ordenadores.

Sin embargo, hoy en día el protocolo **CUDA**, es una nueva plataforma de programación en paralelo que ofrece la memoria compartida para la ejecución en paralelo en la GPU. Ha demostrado ser muy exitoso en la programación de multihilo con cientos de núcleos. Por ejemplo, los científicos en toda la industria y el mundo académico ya están usando CUDA para alcanzar aceleraciones espectaculares sobre los códigos de producción y de investigación. En este trabajo, nos enfocaremos en la programación en paralelo utilizando CUDA. En el capítulo III daremos una descripción detallada de la arquitectura de CUDA.

II.6.1. Breve descripción de MPI y OpenMP

Programación MPI

MPI es una especificación para las operaciones de paso de mensajes diseñadas para ser usada en programas que exploten la existencia de múltiples procesadores (para más detalles ver (Pacheco, 2011; Puente, 2015)). Define cada trabajador como un proceso y proporciona enlaces de lenguaje para C, C ++ y FORTRAN. MPI ofrece un importante

conjunto de bibliotecas para escribir, depurar y probar el rendimiento en programas distribuidos.

La ventaja para el usuario es que, MPI ha sido estandarizada en muchos niveles. Por ejemplo, ya que la sintaxis es estándar, se puede estar seguro de que el código MPI se ejecutará bajo cualquier aplicación de MPI. Dado el comportamiento funcional de MPI, "las llamadas" también están estandarizadas. Además las llamadas de MPI deben comportarse de la misma forma, lo que garantiza la portabilidad de sus programas paralelos. Sin embargo, los resultados pueden variar de una implementación a otra.

Programación OpenMP

OpenMP es una API, cuyas características pretenden facilitar el desarrollo de programas paralelos (Quinn, 2004). Consta de directivas de compilador, librerías en tiempo de ejecución y variables de entorno que facilitan la descripción de una o más porciones de un programa que deben ser ejecutadas por varias unidades de procesamiento de manera simultánea. Asimismo permiten especificar los recursos a utilizar (tamaño de memoria compartida, cantidad de hilos o threads de ejecución) en cada una de estas unidades y determinar de qué manera se dividirá la carga de trabajo entre las mismas. Lejos de ser un lenguaje de programación provee a los desarrolladores de una notación especial que puede ser utilizada en programas escritos en C, C++ o FORTRAN.

Entre sus principales ventajas se encuentra el hecho de ser multiplataforma, ya que soporta una variedad de sistemas operativos (UNIX, Linux, Solaris, Windows), así como un alto número de compiladores, entre los que cabe hacer referencia a GNU gcc, xl de IBM y los compiladores de las herramientas de desarrollo Solaris Studio de Oracle y Visual Studio C++ de Microsoft.

Por otro lado, en su nivel más simple, puede utilizarse OpenMP para paralelizar programas secuenciales sin grandes cambios al código fuente del mismo, más allá de

un pragma o directiva de compilador para señalar el inicio y/o fin de una sección de código a paralelizar. Para esto basta con agregar la directiva **#pragma** omp antes del bucle a paralelizar y el compilador se encargará de paralelizarlo. Por supuesto estas instrucciones cuentan con un rango de parámetros y opciones que escapan de los alcances de este documento. Una desventaja de OpenMP es el alto costo computacional (overhead) "inaceptable" presente al paralelizar aplicaciones con cantidades "relativamente grandes" de threads.

II.7. Paradigmas de programación en paralelo

¿Qué hacer ante un problema a paralelizar?

Es una pregunta abierta, para la cual no existe una única respuesta, ya que depende de la naturaleza del problema. Sólo se podrán paralelizar aquellos problemas cuya estructura lógica subyacente sea de naturaleza paralela; es decir, que nuestro problema pueda ser analizado a partir de la descomposición de sus partes, y estas partes tengan cierto grado de independencia.

Se pueden desarrollar aplicaciones en paralelo mediante el uso de los siguientes paradigmas:

- Lenguaje C, FORTRAN y C++ usando procesos UNIX (Nivel Proceso).
- Lenguaje C con hilos POSIX (Nivel hilo).
- Código fuente en FORTRAN, C o C++ usando directivas.
- Lenguaje C o FORTRAN usando envío de mensajes (Nivel Proceso).

Lenguaje C, FORTRAN y C++ usando procesos UNIX (Nivel Proceso)

Un proceso UNIX es una instancia de un programa en ejecución. Consta de un espacio de direcciones, atributos del proceso y un hilo de ejecución. El kernel o núcleo es el encargado de crear procesos y distribuirlos a diferentes CPUs, así como de maximizar la utilización del sistema.

Lenguaje C con hilos POSIX (Nivel hilo)

En este método es necesario que el sistema operativo de la máquina paralela soporte hilos POSIX. Por ejemplo IRIX, que es el sistema operativo de Origin 2000 que soporta hilos POSIX.

Código fuente en FORTRAN, C o C++ usando directivas (Nivel Sentencia)

El *paralelismo a nivel sentencia* da la facilidad de que un programa escrito en forma serial pueda reescribirse manual o automáticamente en forma paralela, mediante el uso de algunas herramientas (PCA, PFA, OpenMP) de los sistemas de memoria compartida.

El paralelismo a nivel sentencia consiste en paralelizar declaraciones de ciclos DO o FOR en el código fuente de un programa, mediante la implementación de directivas; además de rutinas o procedimientos más complejos.

Una vez que al código fuente se le han insertado las directivas o pragmas, el programa puede compilarse y ejecutarse en un ambiente serial o paralelo. Serial en el sentido de que si no se cuenta con un compilador paralelizador que soporte el uso de directivas o que la plataforma no sea de multiprocesamiento, las directivas se ignoran totalmente y se ejecuta en forma secuencial.

Lenguaje C o FORTRAN usando envío de Mensajes (Nivel Proceso)

En sistemas de memoria distribuida donde cada procesador tiene su propia memoria, el programa a ejecutarse en paralelo debe hacer uso del modelo de "Envío de Mensajes". Este modelo es apropiado para la comunicación y sincronización entre pro-

cesos que se ejecutan de manera independiente (con su propio espacio de direcciones) en diferentes computadoras. La programación consiste en enlazar y hacer llamadas dentro del programa por medio de bibliotecas que manejarán el intercambio de datos entre los procesadores. Algunas de las bibliotecas son:

- Message Passing Interface (MPI).
- Parallel Virtual Machine (PVM).
- Shared Memory (SHMEM).

En MPI, PVM y SHMEM, uno de los principales esquemas que se emplean en la comunicación y generación de los procesos es el modelo de **Maestro-Esclavo**. Un proceso maestro divide y distribuye el trabajo en subtareas, que las asigna a cada nodo conocido como **trabajador** o esclavo. Terminando cada trabajador su parte, envían los resultados al maestro para que este recopile la información y la presente.

Capítulo III

ARQUITECTURA DEL PROTOCOLO CUDA

En este capítulo se presentan los conceptos básicos relacionados con la tecnología CUDA NVidia. Posteriormente se explica en detalle el hadware de GPU NVidia con cada uno de sus componentes fundamentales.

III.1. Definición de CUDA

En noviembre de 2006, NVidia[1] introduce el protocolo CUDA que hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por NVidia (NVIDIA, 2015).

CUDA (Arquitectura Unificada de Dispositivos de Cómputo). Este protocolo es una plataforma de computación en paralelo que presenta un nuevo modelo de programación y un conjunto de instrucciones, que combinados permiten utilizar la **unidad de procesamiento gráfico** (GPU) de NVidia; para resolver grandes problemas complejos

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

de forma más eficiente que si se utilizase una CPU. De esta forma un nuevo nombre ha surgido para referirse a las GPUs, *General-Purpose computing Graphic Processing Units* también llamadas GPGPU, y la idea consiste en utilizar las altas concurrencias y el paralelismo de las tarjetas gráficas para aplicaciones de cómputo general.

Para programar en la arquitectura de CUDA, actualmente los desarrolladores pueden usar C para CUDA, el cual es el lenguaje de programación con extensiones provistas por NVidia. También es compatible con múltiples lenguajes de programación de uso - generalizado como C++, Python, FORTRAN o Java (NVIDIA, 2015). En este trabajo nos centraremos en el lenguaje de programación FORTRAN, es decir CUDA FORTRAN.

Entre las ventajas de CUDA encontramos principalmente la capacidad de acceder a la memoria compartida de las tarjetas gráficas, la cual alcanza velocidades mucho mayores que la memoria (global) del dispositivo. Sin embargo, la desventaja más notoria es que sólo se puede utilizar en GPUs de la marca NVidia, por lo que la portabilidad de las aplicaciones desarrolladas para esta arquitectura está claramente limitada a las tarjetas gráficas de este vendedor.

Por último, a manera general, se entiende que la arquitectura CUDA es la arquitectura en la GPU de NVidia hablando a nivel hadware, ya que en el capítulo IV también hablaremos de la arquitectura de GPU a nivel de programación.

III.2. GPU

Una **GPU** (conocido comúnmente como tarjeta gráfica), es un dispositivo de hardware que por sus características, se encarga del procesamiento de gráficos en una computadora, logrando aligerar en este aspecto la carga de trabajo en la CPU (Unidad de Procesamiento Central) en aplicaciones como los videojuegos o aplicaciones 3D interac-

tivas. Hoy en día se pueden encontrar GPUs en una gran cantidad de dispositivos como lo son: computadoras de escritorio, computadoras portátiles, consolas de videojuegos y video proyectores, entre otros.

Una tarjeta GPU es altamente especializada, pues está pensada para realizar una sola tarea. Por ejemplo, el procesamiento gráfico, que realiza una gran cantidad de cálculos aritméticos sobre números reales (representados en una computadora mediante la representación de coma flotante). Esto permite aprovechar la gran potencia de cálculo de las GPUs para aplicaciones no relacionadas con los gráficos, tal como se mencionó previamente con la GPGPU.

III.2.1. Breve historia de las GPUs

Los procesadores han evolucionado, tanto en velocidad de su reloj interno, como en la cantidad de núcleos. Entre tanto, el estado del procesamiento gráfico pasó por un proceso de revolución dramático. A finales de los años 80 y principios de los 90, el aumento de la popularidad de los sistemas operativos gráficos como Microsoft Windows, ayudó a crear un mercado para un nuevo tipo de procesador. Al comienzo de 1990, los usuarios empezaban a adquirir tarjetas de aceleramiento gráfico en dos dimensiones (2D) (Harrigan, 2004), para sus computadoras personales. Esas tarjetas gráficas jugaban un rol como asistentes que ayudaban en operaciones de mapas de bits en sistemas operativos gráficos.

En la misma época, en el mundo de la computación, a un nivel profesional, una compañía de nombre Silicon Graphics, popularizó el uso de gráficas en tres dimensiones (3D) en una variedad de mercados, incluyendo aplicaciones de gobierno, defensa y visualización técnica y científica; así como una forma de proveer herramientas para estudiar efectos cinematográficos .

En 1992, Silicon Graphics, abrió la interfaz de programación para este hardware, realizando la librería OpenGL. Silicon Graphics intentó con esta librería, para ser el estándar usado como plataforma independiente y poder escribir aplicaciones gráficas en 3D. Al igual que el procesamiento en paralelo y la CPU, sólo sería cuestión de tiempo para que todas estas aplicaciones y tecnologías encontraran un camino dirigido hacia las aplicaciones de consumo.

A mediados de 1990, la demanda de aplicaciones empleando gráficos 3D había escalado velozmente, preparando el escenario para desarrollos significativos. El cual fue alentado primero, por el desarrollo de juegos en primera persona (first-person), como Doom, Dukem 3D y Quake, ayudó en el inicio de una búsqueda para crear entornos 3D más realistas, para juegos de computadora. Aunque los gráficos en 3D trabajarían eventualmente en el camino de juegos de computadoras, la popularidad del género de los juegos de disparos en primera persona, generó la adopción de los gráficos 3D en la informática de consumo. Al mismo tiempo, empresas como NVidia, ATI Technologies y 3dfx Interactive**,** comenzaron a promocionar aceleradores gráficos, lo suficientemente factible para atraer la atención general. Estos desarrolladores asientan los gráficos en 3D, como una tecnología que figura y promete ocupar un lugar destacado en los próximos años.

Desde el punto de vista del procesamiento en paralelo, el lanzamiento de la serie GeForce 3 de NVidia en 2001, representa el más importante avance en la tecnología GPU. La serie GeForce 3 fue la primera en la industria computacional en implementar, el entonces nuevo estándar DirectX 8.0 de Microsoft. Dicho estándar requería que el hardware compatible contuviera vértices programables arrancables y etapas programables de sombreado de pixeles. Por primera vez, los desarrolladores tenían algún control sobre los cálculos exactos que se pueden realizar en las GPUs.

III.2.2. Principios de programación en la GPU

El lanzamiento de las GPUs que tienen múltiples líneas de programación, atrajo a investigadores a la posibilidad de usar el hardware, para algo más que representar OpenGL o DirectX. Durante el enfoque general que se dió en los inicios de la computación GPU fue bastante complicado, debido a que los estándares gráficos API, como OpenGL y DirectX, siguen siendo la única forma de interactuar con la GPU. Por eso, los investigadores exploraron la computación de propósitos generales, a través de gráficos API, tratando de resolver problemas por medio de las GPUs.

Fundamentalmente, las GPUs desde hace dos décadas fueron diseñadas para producir un color por cada pixel en la pantalla, utilizando unidades aritméticas programables, conocidas como sombreado de pixeles (pixel shaders). En general, un sombreado de pixel usa coordenadas (x, y) para posicionar en la pantalla, como alguna información adicional, y combinar varias entradas para determinar el color final. Esta información extra, podría ser: colores de entradas, coordenadas de texturas o algún otro atributo que se pasa al sombreado mientras se ejecuta. Pero debido a que la aritmética que se realiza sobre las entradas de colores y texturas, estaba completamente controlada por el programador, y los investigadores observaron que esas entradas de "colores", podrían ser cualquier dato.

Por otra parte, si las entradas eran datos numéricos con algún valor, más que un color, los desarrolladores podrían programar los sombreados de pixeles para realizar los cálculos computacionales sobre estos datos. Los resultados retomaron a la GPU como un pixel final de "color", a pesar de que los colores serían simplemente los resultados de los cálculos, que el programador había dado a las entradas de la GPU. Esos datos podrían ser leídos por los investigadores y la GPU nunca volvería a ser la más sabia. En esencia, la GPU estaba siendo engañada en la realización de tareas, haciendo que

esas tareas aparecieran como si se tratara de representación estándar. Este engaño o trampa, fue muy ingenioso pero además, muy engorroso.

Debido a la aritmética de rendimiento superior de las GPUs, los resultados iniciales desde esos experimentos, prometían un futuro vislumbrante para la programación en GPU. Sin embargo, el modelo de programación todavía era demasiado restrictivo para la masa de desarrolladores para formarse. Hubo restricciones de recursos, desde que los programas podrían recibir datos de entrada de sólo un conjunto de colores de entrada y texturas. También había limitaciones sobre cómo y dónde, el programador podría escribir resultados en la memoria, ya que los algoritmos requieren de la habilidad para escribir en localidades arbitrarias en memoria y no se podría ejecutar en la GPU. Por otra parte, era casi imposible de predecir, como la GPU se ocupará de datos de coma flotante, si el manejo de datos de coma flotante era total, por lo que la mayoría de los cálculos científicos serían incapaces de usar la GPU.

III.2.3. Filosofía del diseño CUDA

No sería hasta cinco años después del lanzamiento de la serie GeForce 3, que la computación GPU estaría lista para el momento estelar. En noviembre del año 2006 NVidia dio a conocer el primer GPU DirectX 10 a la industria: el modelo **GeForce 8800 GTX.** Este modelo fue además, el primer GPU en ser construido con la arquitectura CUDA de NVIDIA. Esta arquitectura incluye nuevos componentes, diseñados estrictamente para la computación GPU, cuya finalidad era aliviar muchas de las limitaciones que impedian las anteriores GPUs.

Actualmente las GPUs de NVidia se han desarrollado tomando en cuenta la creación de aplicaciones de propósito general y se caracteriza por el alto grado de paralelismo, ya que soporta una gran cantidad de hilos (threads) en ejecución al mismo tiempo, esto

comparado con los procesadores más actuales. Por lo tanto, la filosofía general para el diseño de GPUs de CUDA es optimizar la ejecución masiva de hilos.

La compañia NVidia demostró la evolución de sus productos en comparación con la evolución de procesadores de Intel, como se muestra en la Fig. 6. Esta comparación muestra como la capacidad de cómputo de las GPUs crece de forma exponencial, mientras que la de las CPUs crece de forma lineal (NVIDIA, 2015).

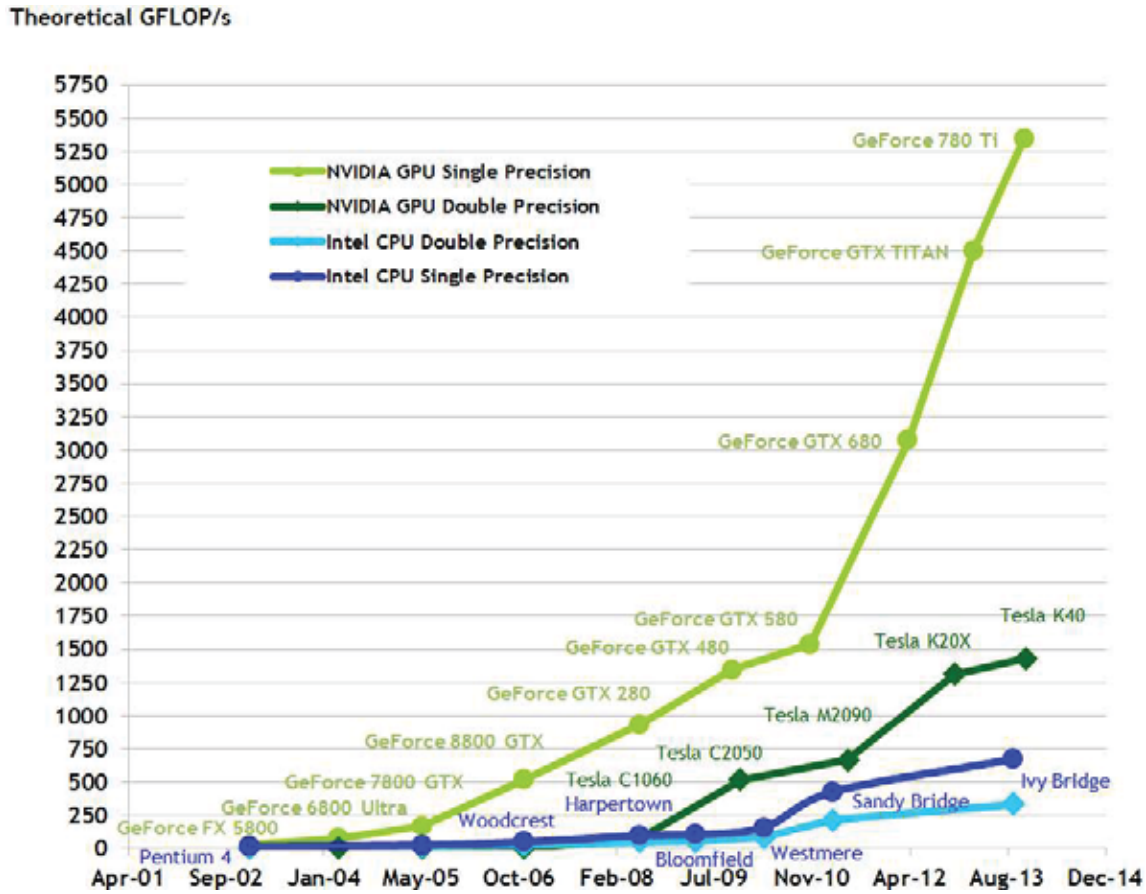

Figura 6. Comparación de rendimiento entre CPUs de Intel y GPUs de NVIDIA

La razón de esta diferencia de rendimiento es que la GPU está especializada en cómputo intensivo y en computación altamente paralela. Por lo tanto, la GPU está diseñada de manera que más transistores se dedican al procesamiento de datos.

Una forma sencilla de comprobar rápidamente esta discrepancia es viendo la dife—rencia entre la arquitectura de una CPU multicore y una GPU moderna como se ilustra en la Fig. 7.

Figura 7. CPU y GPU tienen básicamente diferente filosofia de diseño.

En la Fig. 7, se muestra el esquema de un procesador multicore que cuenta con 4 núcleos (cores). En cambio la GPU cuenta con 8 multiprocesadores con 16 núcleos cada uno, con un total de 128 núcleos. Tanto la CPU como la GPU tienen una memoria global DRAM que es de mayor tamaño con una latencia mayor que cualquiera de las cachés y además es compartida entre todos los cores.

III.3. Arquitectura GPU con el protocolo CUDA

Las arquitecturas a nivel hadware de GPU NVidia aptas para CUDA están compuestas por dos componentes principales:

- Memoria Global.
- Multiprocesadores Paralelos o **Streaming Multiprocessors** (SMs).

La Memoria Global es la memoria de la GPU y es accesible tanto por la GPU y la CPU. Los multiprocesadores se agrupan de tres en tres o de dos en dos como se ilustra

en la Fig. 8. En esta figura puede apreciarse la arquitectura de una tarjeta NVidia con 14 multiprocesadores (SMs), con 8 núcleos o **streaming processors** (SPs) cada uno y sus respectivas unidades de memoria. El número de SMs varían desde las arquitecturas más antiguas, hasta las más modernas y de mayor gama.

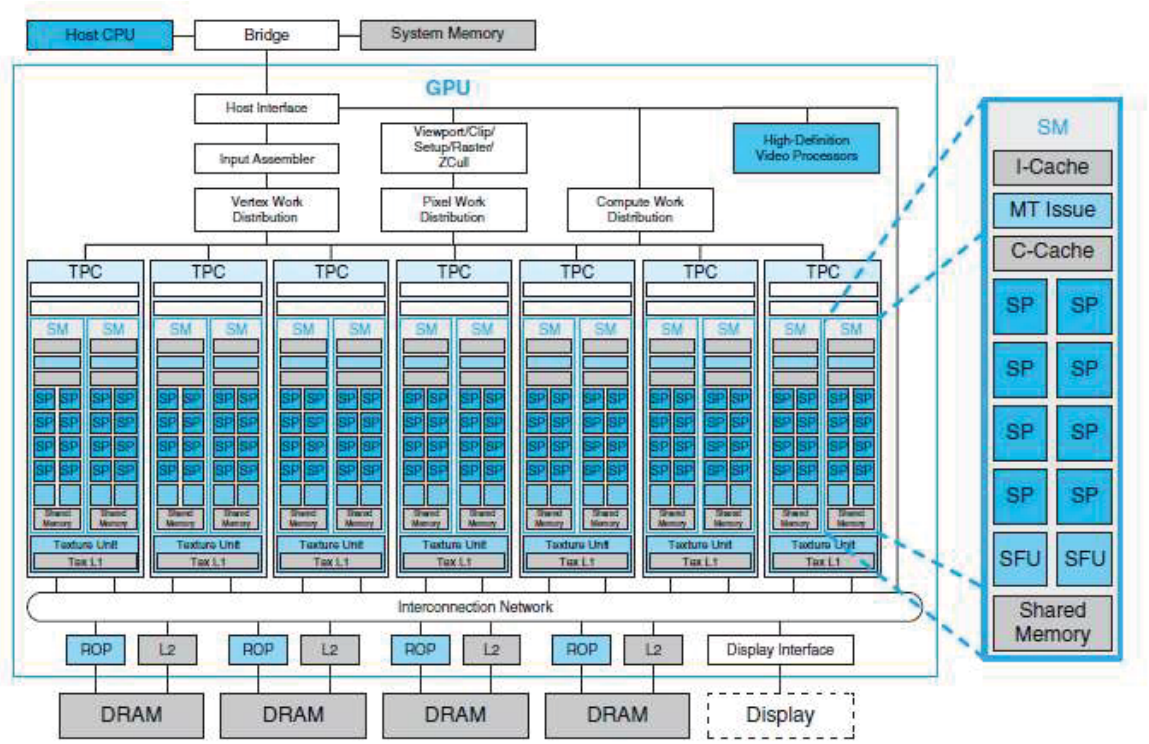

Figura 8. Arquitectura de una tarjeta NVidia con 112 núcleos.

Ahora, describiremos la arquitectura básica de un **multiprocesador SM** compuesto por M núcleos, como se muestra en la Fig. 9. Todos los núcleos comparten una unidad de control, de modo que todos ejecutan la misma instrucción, pero cada núcleo ejecuta un hilo diferente; es decir, un SM ejecuta M hilos que comparten la misma instrucción. Esto es a lo que se denomina arquitectura **SIMT**[2]. Cada núcleo tiene un banco privado de registros, lo que permite cierta independencia entre los distintos hilos ejecutados en un mismo multiprocesador SM. En el capitulo IV se mostrará en detalle

[2]Por sus siglas en inglés, Single Instruction, Multiple-Thread.

lo referido a la ejecución del software de los hilos.

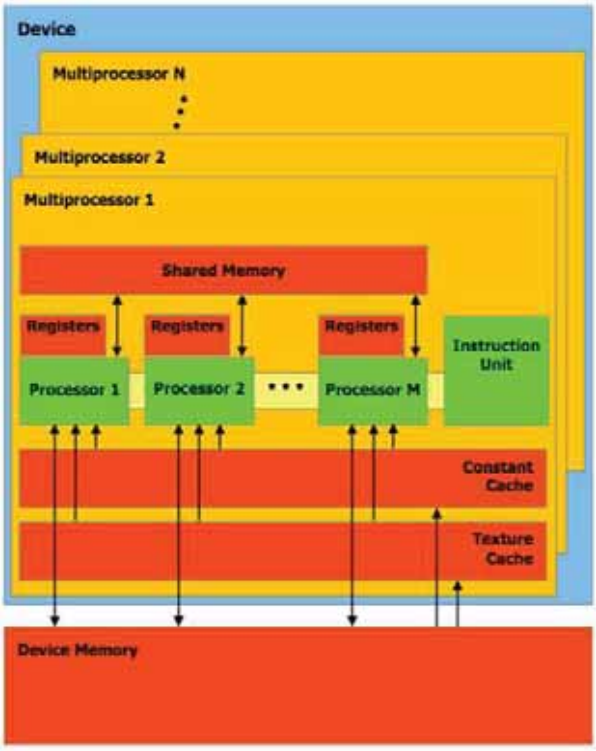

Figura 9. Arquitectura interna de un SM (Streaming Multiprocessor).

Dentro de un SM encontramos también 3 tipos de memoria compartidas entre los distintos núcleos. La primera, una memoria compartida de lectura/escritura de 16KB, que ofrece un tiempo de acceso similar al de un registro. La segunda y tercera corresponden con dos cachés de sólo lectura: una de constantes y otra de texturas.

Estos multiprocesadores SM han sido diseñados para ejecutar cientos de hilos concurrentemente. Para poder crear, manejar, planificar y ejecutar cantidades tan altas de hilos, el conjunto de hilos que se ejecuta en un multiprocesador SM es dividido en grupos de 32 threads denominados warps. Un warp es por tanto, un grupo de 32 hilos, el cual es la unidad mínima de hilos ejecutados por un multiprocesador SM. Los hilos que pertenecen a un mismo warp empiezan juntos en la misma dirección del programa,

a pesar de que cada uno de ellos tiene su propio contador de instrucciones y tiene su propio registro de estado que por tanto son libres de separarse del hilo de ejecución del warp y de ejecutarse independientemente. El contexto de ejecución de cada uno de los warps es mantenido en el multiprocesador SM durante toda la vida útil del warp.

III.3.1. Descripción de las memorias en la GPU

En esta parte hacemos una descripción de los dispositivos de memoria previamente mencionadas. Es una descripción física donde se describe su situación y sus funciones. Más adelante en el capítulo IV veremos la visión lógica que proporciona CUDA y que utiliza los distintos tipos de memoria descritos a continuación.

- **Registros:**

Todo multiprocesador SM tiene un banco de registros y son los dispositivos de memoria más rápidos con los que cuenta una GPU.

- **Memoria compartida:**

Al igual que los registros esta memoria está físicamente localizada en el multiprocesador SM. Es una memoria rápida que permite realizar accesos con poca latencia. Como en toda jerarquía de memoria, cuanto menor es la latencia menor es la capacidad.

- **Memoria global:**

Esta memoria está situada físicamente en la DRAM de la GPU y es accesible desde todos los hilos residentes en la GPU en modo lectura/escritura. Tiene una alta latencia por lo que requiere un trato especial para realizar los accesos (Farber, 2011).

Esta memoria puede ser comparada con la memoria principal de una CPU. Es la memoria con más capacidad de la jerarquía de memoria pero la más lenta. Tal y como

sucede en la CPU, la jerarquía de una GPU moderna cuenta con dos niveles de caché que intenta ocultar en cierta medida la alta latencia de acceso a esta memoria.

- **Memoria de constantes y memoria de texturas:**

Son memorias situadas físicamente en la memoria DRAM de la GPU y son accesibles en modo lectura. Además, son memorias específicas para el tratamiento de gráficos, aunque pueden utilizarse en el ámbito general. Ambas cuentan con una caché específica dentro de cada multiprocesador SM.

III.4. CUDA: como una plataforma para la programación heterogénea

La computación heterogénea se refiere a los sistemas que utilizan más de un tipo de procesador. Se trata de sistemas multi-core que obtienen un rendimiento no sólo mediante la adición de núcleos (cores), sino también mediante la incorporación de capacidades de procesamiento especializados para manejar tareas particulares.

La arquitectura de sistemas heterogéneos utiliza varios tipos de procesadores (normalmente CPU y GPU), por lo general en el mismo chip. El procesamiento en la GPU, aparte de sus conocidas capacidades de procesamiento gráficos en 3D, también puede realizar cálculos matemáticos intensivos en conjuntos de datos muy grandes, mientras que las CPU ejecutan el sistema operativo y realizan tareas de serie tradicionales.

Una GPU actualmente no es una plataforma independiente sino un coprocesador de la CPU, que posee su propia memoria. La comunicación de la GPU con la CPU se realiza a través de un bus llamado PCI-Express. Dicho bus consta de dos vías (una de envío y otra de recepción), como se muestra en la Fig. 10. Es por ello que, en términos

de programación en CUDA, la CPU se llama “Host” (anfitrión) y la GPU de NVIDIA se llama “Device” (dispositivo).

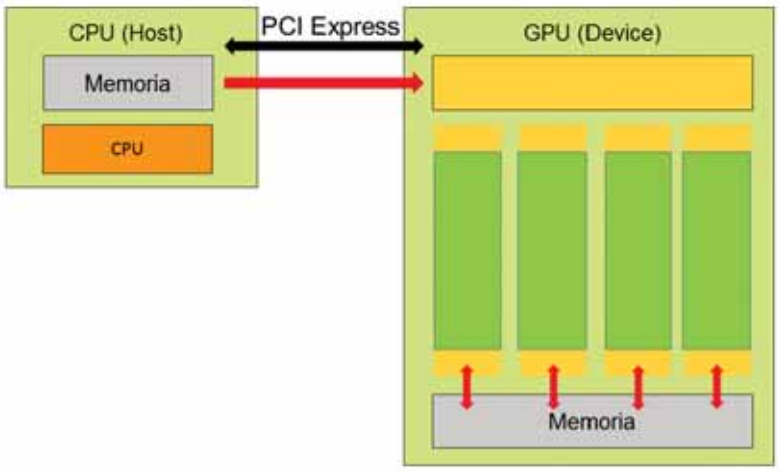

Figura 10. GPU como coprocesador.

Una aplicación heterogénea (CUDA) consiste en dos partes:

- Código Host.
- Código Device.

El código Host se ejecuta en la CPU y el código Device se ejecuta en la GPU. Una aplicación que se ejecuta en una plataforma heterogénea normalmente es inicializado por la CPU. El código de CPU es responsable de manejar el entorno, el código y los datos para el dispositivo antes de que la carga calcule las tareas intensivas sobre el dispositivo.

Con las aplicaciones intensivas de cómputo, las secciones del programa a menudo presentan una rica cantidad de paralelismo de datos. Las GPUs son usadas para acelerar la ejecución de esta parte del paralelismo de datos. Es por ello que las GPUs suelen ser llamados como acelerador de hardware.

La plataforma de cómputo GPU de NVIDIA está disponible en las siguientes fami–lias.

- *Tegra*
- *GeForce*
- *Quadro*
- *Tesla*

La familia de productos Tegra está diseñado para dispositivos móviles e integrados, como tabletas y teléfonos, la de GeForce para gráficos, la de Quadro para la visualización profesional y la de Tesla para la computación paralela de datos.

Hay dos características importantes que describen la capacidad de la GPU:

- Número de núcleos CUDA.
- Tamaño de memoria.

NVidia utiliza un término especial, la capacidad de cómputo, para describir las versiones de hardware de aceleradores GPU que pertenecen a toda la familia de productos Tesla. La versión de los productos Tesla se da en la Tabla 1.

Tabla 1. Productos de la tarjeta Tesla.

GPU	Capacidad de cómputo	Núm. núcleos
Tesla K40	3.5	2880
Tesla K20	3.5	2496
Tesla K10	3.0	1536
Tesla C2070	2.0	448
Tesla C1060	1.3	240

En este trabajo de tesis se utilizó una tarjeta NVidia Tesla K40 (NVIDIA, 2013). Esta arquitectura consta de 2280 núcleos de procesamiento en paralelo que puede ejecutar colectivamente miles de hilos y con una memoria de 12 GB que permite procesar un conjunto de datos 2 veces más grandes.

Figura 11. Aspecto físico de una targeta gráfica NVidia con GPU Tesla K40.

III.4.1. Paradigmas de la programación heterogénea

La computación en GPU no pretende sustituir la computación en la CPU. Cada enfoque tiene ventajas para ciertos tipos de programas. La programación en CPU es bueno para el control de tareas intensivas y la programación en GPU es buena para el cálculo de datos paralelos en tareas intensivas. Cuando las CPUs se complementan con las GPUs, lo convierte en una poderosa combinación para el cálculo de aplicaciones. La CPU es optimizada para cargas de trabajo dinámicas marcadas por las secuencias cortas de operaciones de cálculo y flujo de control impredecible y las GPUs tienen como objetivo las cargas de trabajo que están dominadas por tareas de cálculo con flujo de control simple. Hay dos dimensiones que diferencian el alcance de las aplicaciones para CPU y

GPU:

- Paralelismo de datos.
- Tamaño de datos.

Si un problema tiene tamaño de datos pequeño, la lógica de control es sofisticada y el paralelismo es de bajo nivel, entonces la CPU es buena opción debido a su capacidad de manejar la lógica compleja y el paralelismo a nivel de instrucción. En cambio, si el problema se trata del procesamiento de una gran cantidad de datos que expone el paralelismo de datos masivos, entonces la GPU es la mejor opción, ya que tiene un gran número de núcleos programables.

La combinación de la CPU+GPU son arquitecturas heterogéneas para la programación paralela que han evolucionado ya que la CPU y la GPU tienen cualidades complementarias que permiten a las aplicaciones dar un mejor uso a estos procesadores. Es por esta razón, y con la idea de apoyar la CPU conjunta con la GPU para la ejecución de una aplicación, NVidia diseñó el modelo de programación con CUDA. En el capítulo IV se describirá este modelo de programación.

Capítulo IV

MODELO DE PROGRAMACIÓN EN CUDA FORTRAN

En este capítulo se introduce la programación de la GPU mediante CUDA FORTRAN, detallando sus características básicas de: funciones kernel, hilos, bloques y mallas, organización y asignación de recursos, ejecución de un kernel, entre otras. Además, se explicará un ejemplo en CUDA FORTRAN a través de un ejemplo simple.

IV.1. Requerimiento

Los requisitos para desarrollar un programa en CUDA FORTRAN son los siguientes:

- Un procesador (GPU) habilitado para CUDA en una tarjeta gráfica Nvidia.
- Controlador (driver) de la tarjeta gráfica.
- El kit de desarrollo de CUDA.
- Sistema operativo LINUX (en este caso).
- Un compilador para CUDA FORTRAN.

Se puede adquirir el driver y el kit de desarrollo en la página web oficial de Nvidia cuya dirección es *http://www.nvidia.com/cuda*. Allí se podrá descargar el driver genérico y el kit de desarrollo, donde se encuentran las librerías y conjunto de recursos para la programación con CUDA. Para este trabajo hemos usado el compilador de PGI FORTRAN. Éste incluye soporte para CUDA FORTRAN en Linux, Apple OSX y Windows.

Verificación de requisitos

Para comprobar que se cuenta con una tarjeta apta para CUDA, desde una terminal o consola de Linux se ejecuta el siguiente comando (ver Fig. 12).

```
$ lspci | grep -i NVIDIA
```

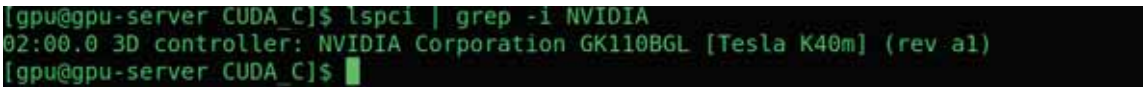

Figura 12. Targeta gráfica instalada.

Es necesario conocer el modelo de la tarjeta gráfica para descargar el driver o controlador correcto.

IV.2. Conceptos básicos

Para empezar a programar necesitamos definir algunos términos generales. CUDA FORTRAN (PGI, 2015) es un conjunto pequeño de extensiones de FORTRAN que se apoya y se basa en la arquitectura de cómputo con CUDA y es un modelo de programación híbrido, lo que significa que parte del código se ejecuta en la CPU (host) y la otra parte en la GPU (device). O bien, de manera concisa:

- **Host**: la CPU y la memoria (memoria principal).
- **Device**: la GPU y la memoria (memoria del dispositivo).

Otro concepto importante que vamos a utilizar son los hilos (threads). Un hilo en CUDA es una unidad de ejecución en la que se procesa parte de los datos y que puede sincronizarse y compartir información con otros hilos.

IV.2.1. Los hilos y su ejecución dentro de la GPU

Un hilo dentro de la GPU es ejecutado en uno de los núcleos SP. A nivel de código fuente, todas las instrucciones que componen el plan de ejecución de un hilo son programadas en funciones llamadas "**Kernels**".

Los hilos son agrupados en "**blocks**" (bloques), de manera tal que todos los hilos de un mismo bloque comparten un mismo espacio de memoria. Físicamente, cada bloque es ejecutado dentro de un Multiprocessor (SM). Cada SM puede ejecutar hasta ocho bloques de forma simultánea, siempre y cuando se cumplan todas las restricciones sobre los recursos del multiprocesador (Hong y Kim, 2009). El multiprocesador crea, gestiona y ejecuta hilos concurrentes en el hardware. Y cada hilo se ejecuta de forma independiente con su propia dirección de instruccion y registros de estado. El conjunto de todos los hilos y bloques de una aplicación se denomina "**grid**" (malla), y se define una sola malla por cada tarjeta de video disponible en el sistema. En la Fig. 13 se ilustra esta explicación.

Es interesante notar que al momento de programar un bloque, sus hilos pueden estar organizados de acuerdo a la aplicación, en bloques de una dimensión (para manejar vectores), de dos dimensiones (para manejar matrices) y de tres dimensiones (para manejar arreglos tridimensionales). Una malla también puede tener a sus bloques organizados en la forma de un vector o de una matriz, o hasta de tres dimensiones para las GPUs modernas. Por lo tanto, CUDA consigue proporcionar el paralelismo a través de la malla, bloque e hilo.

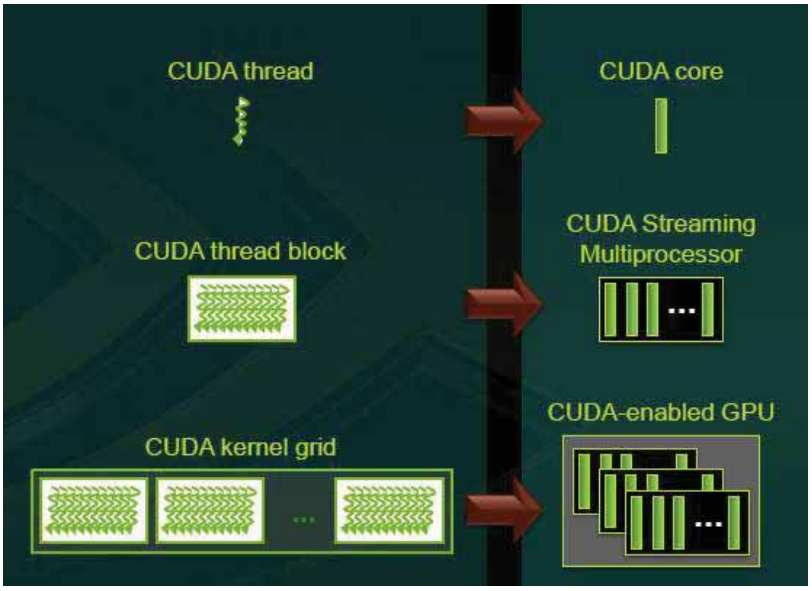

Figura 13. Correspondencia entre hilos y hardware.

También es importante señalar que cada tarjeta tiene definido una cantidad máxima de hilos a ejecutar por cada bloque, como se describirá en la siguiente sección explicando el concepto de "Compute Capability" que corresponde a una forma de clasificar estas tarjetas en función de sus características principales.

IV.2.2. Capacidad de cómputo

El concepto de "Compute Capability" es usado por NVidia para clasificar sus tarjetas gráficas (compatibles con CUDA) en función de sus características de hardware, de sus capacidades a nivel de cálculo y de su configuración de memoria. Es un indicador que permite saber de inmediato el nivel de exigencia máximo al que puede ser sometida una tarjeta gráfica.

Como ya habíamos mencionado en el capítulo anterior, para este trabajo hemos usado una tarjeta gráfica NVidia con GPU Tesla K40M, cuyas principales características son expuestas en la Fig. 14.

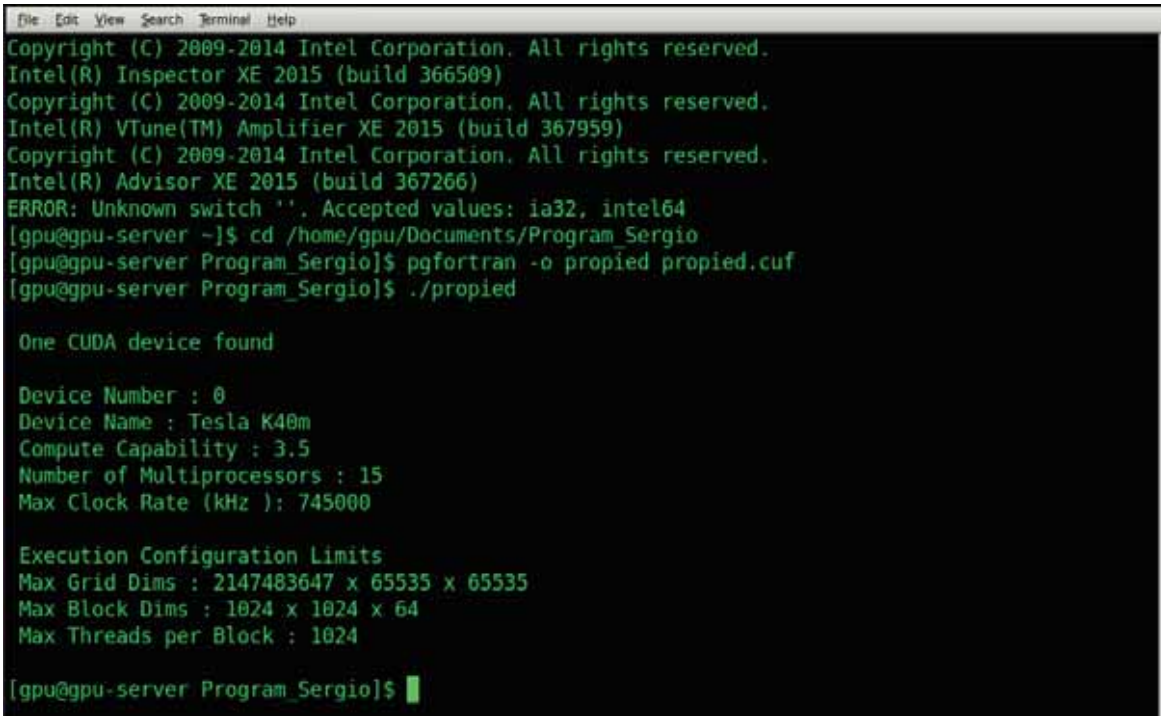

Figura 14. Capacidad de Cómputo.

En la Fig. 14, se observan algunas características del dispositivo, como son: nombre del dispositivo, capacidad de cómputo, número de multiprocesadores y al final se muestra el máximo hilo por bloque que es una característica importante para programar.

IV.3. Generalidades de la programación con CUDA FORTRAN

No todos los problemas pueden ser resueltos en la GPU, los más adecuados son aquellos que pueden resolverse mediante la aplicación del paradigma paralelo de datos, es decir aplican la misma sentencia o secuencia de código a todos los datos de entrada. Se puede decir que una solución de un problema en GPU será más ventajosa respecto a la solución en la CPU si la aplicación tiene las siguientes propiedades:

- El algoritmo tiene un orden de ejecución cuadrático o superior: el tiempo necesario para realizar la transferencia de datos entre la CPU y la GPU tiene un gran costo, el cual no suele verse compensado por el bajo costo computacional de un método lineal.

- Es mayor la carga de cálculo computacional en cada hilo: de nuevo para compensar el tiempo de transferencia de información es conveniente que cada hilo posea una carga computacional considerable.

- Es menor la dependencia entre los datos para realizar los cálculos, esto es posible si cada SM sólo necesita de los datos de su memoria local o compartida y no necesita acceder a memoria global, la cual tiene un acceso más lento.

- Es menor la transferencia de información entre CPU y GPU. La situación óptima es cuando la transferencia sólo se realiza una vez, al comienzo y al final del proceso. Esto significa una transferencia de los datos de entrada, desde la CPU a la GPU, y una al final, desde la GPU a la CPU, para obtener los resultados. Es bueno no tener transferencias intermedias, ya sea de resultados parciales o datos de entradas intermedios.

- No existen secciones críticas; es decir, varios procesos no necesitan escribir en las mismas posiciones de memoria. Las lecturas de memoria global y compartida puede ser simultánea, pero las escrituras en la misma posición de memoria plantea un acceso a un recurso compartido, lo cual implica contar con un mecanismo de acceso seguro. Este proceso hace más lenta la solución del proceso global.

Además, es necesario que las estructuras de datos en la aplicación que ejecuta en la CPU se adapten o puedan transformarse a estructuras más simples del tipo matriz o vector a fin de poder ser compatibles con las estructuras que maneja la GPU.

IV.4. Definición de kernel

De manera general los Kernels son funciones programadas para ser ejecutadas exclusivamente en la GPU, haciendo uso sólo de memoria GPU.

CUDA FORTRAN permite la definición de subrutinas FORTRAN que se ejecutan en paralelo en la GPU cuando se llama desde el programa FORTRAN que se ha interpuesto y se está ejecutando en el host (CPU) (PGI, 2015). Tal subrutina se llama kernel, es decir un *kernel* en CUDA FORTRAN es equivalente a una subrutina FORTRAN. Los kernels son el componente principal del modelo de programación de CUDA, cuando son invocadas se ejecutan N veces en paralelo en diferentes hilos de CUDA; en contraste con las subrutinas de FORTRAN que se ejecutan sólo una vez. Cada hilo se diferencia de los demás por su identificador, el cual es único y accesible en el kernel a través de una variable interna y predefinida de CUDA llamada threadIdx. Esto lo veremos más adelante.

Para la definición de un kernel en CUDA FORTRAN, se deben respetar varias condiciones, las cuales se enuncian a continuación:

1. Debe llevar la etiqueta *attributes (global)*, la cual identifica a un kernel y determina que la función es invocada desde el host y ejecutada en el device.

2. Definir el nombre de la subrutina.

3. Todos los threads que se activen durante la ejecución del kernel, ejecutan el mismo programa, el cual coincide con el kernel que lo activó.

4. El número de hilos es conocido antes de la ejecución del kernel, ellos serán agrupados, según se indica en la invocación, en bloques. Todos los bloques tienen igual número de hilos.

En el **programa 1** se muestra un kernel, que es definido usando la palabra reservada attributes (global) dentro de la subrutina.

Programa 1. Definición de kernel

```
attributes(global) subroutine nombre_kernel (tipo1 p1, tipo2 p2,.  .  .
, tipon pn)
!  declaraciones
!  sentencias del kernel
end subroutine
```

La estructura de una sentencia para llamar un kernel es:

Call nombre_ kernel<<<DimGrid,DimBlock>>> (parámetro1, parámetro2, . . . , parámetron). Los valores entre <<< ... >>> que aparecen en el código anterior, se conocen como la configuración del kernel, donde *DimGrid* define la dimensión de la malla (o número de bloques dentro de la malla) y *DimBlock* define la dimensión del bloque (o número de hilos dentro del bloque). Dentro del kernel a cada hilo, en el mismo instante de su creación y ejecución se le es asignado un identificador único de hilo.

En el **programa 2** se muestra un ejemplo para ilustrar la llamada de un kernel.

Programa 2. Llamada del kernel

```
!  Ejemplo de copia de un vector
attributes (global) subroutine Copia_Vector (V1,V2)
integer ::  id
id = threadidx%x
V 1(id) = V 2(id)
end subroutine
Program CopiaVector
...
Call Copia_Vector <<< 1,N >>> (A,B)
End Program
```

En el **programa 2**, en la segunda línea del código hemos definido al kernel como *Copia_ vector*. Éste será llamado desde el código host (ejecutado en la CPU), por esta razón es definido como global. Los parámetros del kernel son los vectores fuente y

destino. El identificador de hilo, en este caso lo hemos llamado como *id*. Una vez identificado cada hilo leerá la posición del vector indicada por su identificador y lo escribirá en la misma posición en el vector destino. Esta operación será llevada a cabo en paralelo por todos los hilos en ejecución.

En la línea 7, se muestra el código host y lo hemos definido como *CopiaVector*. Aquí es donde se realiza la llamada al kernel. En la llamada, se indica que se ejecutará a un único bloque 1 de N hilos, siendo N la dimensión del vector. Está claro que en este ejemplo hemos manejando vectores y por lo tanto la organización del bloque es unidimensional. Esto lo veremos a detalle más adelante.

IV.4.1. Identificador (Id) de hilos

En la seccion IV.2.1 definimos que los hilos en un bloque se podían organizar hasta de tres dimensiones y al igual que los bloques en una malla. Ahora en esta sección veremos como se definen las coordenadas de los hilos dentro de un bloque de dos dimensiones y la malla en dos dimensiones, ya que estos son los que generalmente se usan en las aplicaciones con matrices.

Todos los hilos que se ejecutan en una malla, ejecutan el mismo kernel, por lo que es necesario tener una única coordenada que permita diferenciarlos unos de otros y repartir el trabajo. La Fig. 15 muestra como se organizan los hilos en una malla bidimensional de 2X3 bloques y un bloque bidimensional de 3X4 hilos cada uno.

Como puede verse, cada hilo queda perfectamente identificado por un identificador del bloque y el identificador del propio hilo dentro del bloque. Estos identificadores suelen usarse como coordenadas para definir que porciones de datos procesa cada hilo. Para identificar a un hilo dentro de un kernel, existen las siguientes variables:

- threadIdx: es una tupla de 3 componentes (x, y, z) que identifica a un hilo den-

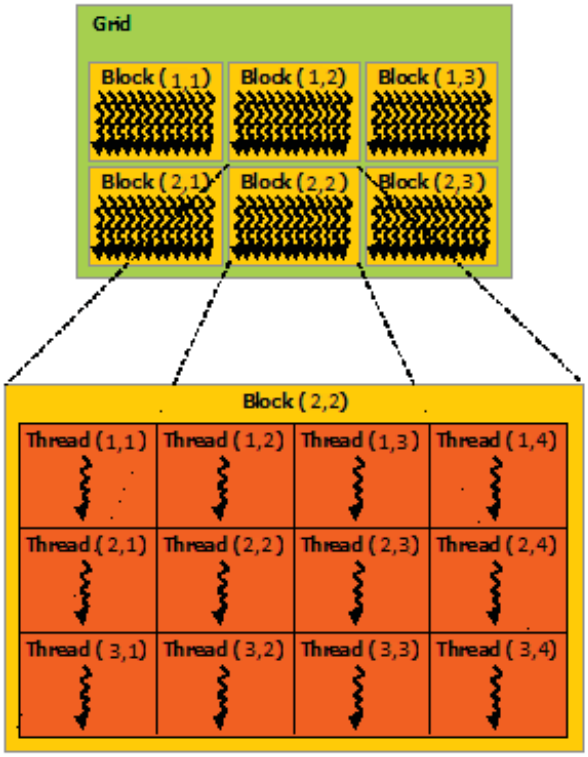

Figura 15. Ejemplo de una distribución de hilos.

tro de su bloque. Para hilo en una dimensión se usa $threadIdx\%x$ y para dos dimensiones usamos el par $(threadIdx\%x, threadIdx\%y)$.

- blockIdx: es una tupla de 3 componentes (x, y, z) que identifica a un bloque dentro de su malla. De la misma manera, para una dimensión se usa $blockIdx\%x$, y en el caso bidimensional se usa $(blockIdx\%x, blockIdx\%y)$.

- blockDim: es una tupla de 3 componentes (x, y, z) que proporciona el tamaño del bloque.

- gridDim: es una tupla de 3 componentes (x, y, z) que proporciona el tamaño de la malla.

Es importante mencionar que en en CUDA FORTRAN, el índice del hilo para cada dimensión comienza en uno; caso contrario en CUDA C que inicia en cero.

En la Fig. 15, vemos que se trata de una malla de 6 bloques identificados por una tupla (x, y). Cada bloque está compuesto por 12 hilos identicados por una tupla (x, y). Por lo tanto el identificador de un hilo, es representado por la tupla (x, y) que es único dentro de su bloque. Es decir, existen varios hilos con el mismo identificador pero cada uno de ellos pertenecen a un bloque diferente y por tanto su identificador del bloque es diferente. Los valores de las variables descritas anteriormente para el primero de los hilos en el bloque $(2, 2)$ serán:

- $threadIdx = (1, 1)$.
- $blockIdx = (2, 2)$.
- $blockDim = (4, 3)$.
- $gridDim = (3, 2)$.

IV.4.2. Jerarquía de memoria

En CUDA se maneja una jerarquía de memoria (NVIDIA, 2015), de modo que cada hilo tiene acceso a distintos tipos de memoria. En primer lugar una memoria local al propio hilo, se aloja en la memoria principal de la GPU. Además, todos los hilos de un mismo bloque comparten una región de memoria compartida para comunicarse entre ellos. Esta memoria compartida tiene el mismo tiempo de vida que el bloque de hilos. Por último, todos los hilos tienen acceso a la misma memoria global (device memory), como se muestra en la Fig. 16

También existen dos espacios de memoria de sólo lectura adicionales (que son accesibles por todos los hilos); el espacio de memoria de constantes y el espacio de memoria de texturas, ambos optimizados para distintos usos. Los espacios de memoria global,

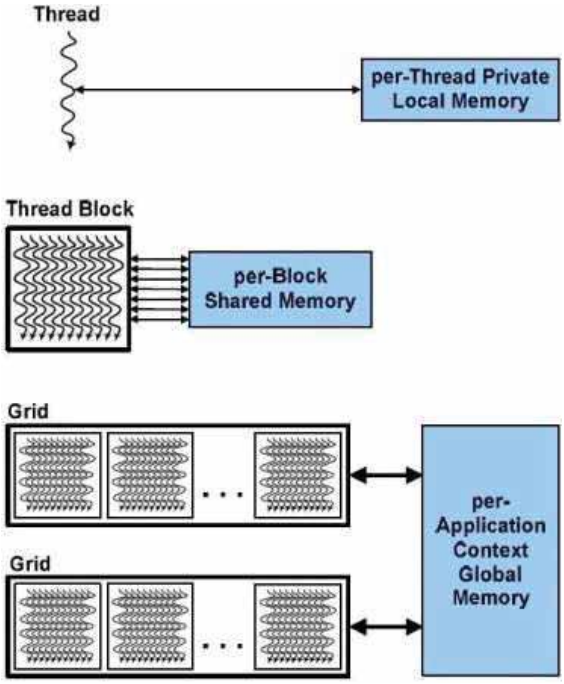

Figura 16. Jerarquía de memoria.

de memoria de constantes y de memoria de texturas son constantes a las llamadas entre distintos kernels.

IV.5. Procedimiento para lanzar un kernel

Ahora que ya conocemos como son los kernels en CUDA FORTRAN y que se lanzan en un número de hilos determinados por el tamaño de bloque y el tamaño de la malla, explicaremos el proceso necesario para lanzar un kernel (Laguna-Sánchez *et al.*, 2011). El proceso para lanzar el kernel se ilustra en la Fig. 17 y que enseguida explicaremos.

1. **Copiar los datos a procesar:** Copiar los datos que van a ser procesados en el kernel a la memoria de la GPU; es decir, copiar el arreglo a la GPU.

2. **Lanzar el kernel:** La CPU llama al kernel con la dirección de los datos en la memoria de la GPU que se obtuvo en el paso 1 indicando el tamaño de bloque y el tamaño de la malla.

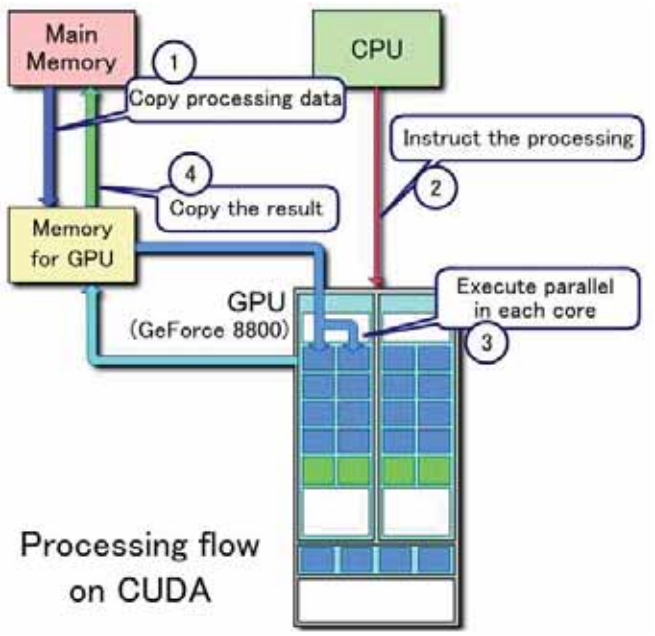

Figura 17. Pasos para lanzar un kernel.

3. **Ejecutar el kernel**: Este paso es transparente al usuario. CUDA ejecuta los kernels solicitados en el paso 2. En cada uno de los núcleos (SPs) se ejecutan los hilos indicados en la invocación del kernel. Los hilos acceden a la memoria global del dispositivo para leer o escribir datos.

4. **Copiar el resultado:** Una vez que se ha ejecutado el kernel obteniendo el resultado en la memoria de la GPU, se realiza el paso inverso al paso 1. Es decir, se copian los datos de la GPU a la CPU para poder seguir trabajando con ellos.

IV.6. Programación paralela en CUDA FORTRAN

Estamos listos para comenzar con algunos programas en CUDA FORTRAN. Para esto, es necesario lo siguiente:

1. Crear un archivo de código fuente con la extensión del nombre del archivo en *.cuf*.

2. Compilar el programa con el compilador PGI FORTRAN.

3. Ejecutar el archivo ejecutable desde la línea de comandos, que contiene el código del kernel ejecutable en la GPU.

Como punto de partida, se muestra el **programa 3** en CUDA FORTRAN para imprimir "hola mundo" de la siguiente manera:

Programa 3. Hola mundo

```
attributes(global) subroutine Saludo()
end subroutine Saludo
program Hola
call Saludo<<<1,1>>>()
print*, "hola mundo!"
end program Hola
```

Guardamos el codigo en archivo .cuf, por ejemplo: programa00.cuf, luego compilamos y ejecutamos con el compilador PGI FORTRAN:

```
$ pgfortran -o programa00 programa00.cuf
$ ./programa00
```

Si ejecutamos el archivo ejecutable programa00, se imprimirá:

```
$ hola mundo!
```

Este ejemplo es sólo una ilustración, ya que en la subrutina no se le pasa ningún parámetro y tampoco hay paralelismo ya que se está usando un bloque de un sólo hilo.

IV.6.1. Ejemplo básico de la suma de vectores

Ahora comenzaremos con la introducción a la programación paralela exponiendo un ejemplo básico para la suma de vectores. La idea de este programa es aplicar los conceptos básicos definidos anteriormente como son: lanzamiento del kernel y el id de los hilos . Así con esta visión se desarrolla otras aplicaciones; por ejemplo, la suma de matrices que lo trataremos en el siguiente capítulo.

Pensemos en dos arreglos de igual tamaño, cada uno con N elementos del mismo tipo. Además, supongamos que se desea sumar cada elemento del arreglo A con su

correspondiente elemento del arreglo B y poner el resultado en un tercer arreglo C. El tercer arreglo es del mismo tamaño que los anteriores. Todo esto es de tal forma que las operaciones de suma se realicen de elemento a elemento de forma independiente y simultanéamente. Esquemáticamente este proceso se ilustra en la Fig. 18.

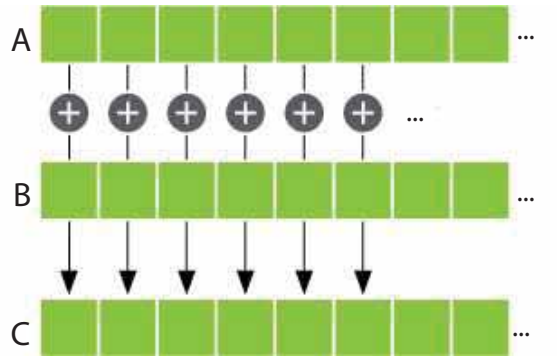

Figura 18. Esquema de suma de dos vectores.

Entonces necesitamos diseñar un kernel donde cada hilo es responsable de calcular la suma de un elemento, y así conseguir ejecutar la suma de vectores en un único paso. De esa manera el número de hilos dependerá de la dimensión de los vectores a sumar. Por ejemplo para sumar vectores de 100 elementos serán necesarios 100 hilos. La Fig. 19 muestra el esquema como se resolverá el problema en la GPU.

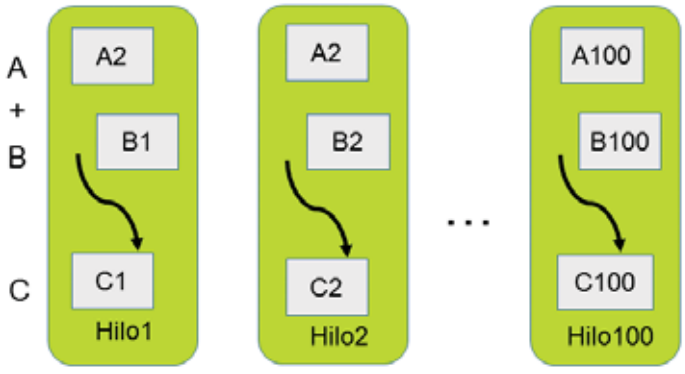

Figura 19. Suma de dos vectores en la GPU.

Como cada hilo calcula una componente del vector suma, cada i-ésimo hilo realizará

la operación $A_i + B_i = C_i$. Para ello necesitamos enviar los dos vectores A y b a la memoria de la GPU y luego realizar ahí la suma almacenándola en una variable: d_c que deberá vivir en la memoria del GPU. En el **programa 4**, se muestra el código completo implementado para la suma de vectores. Posteriormente se explicarán los pasos importantes llevados a cabo.

Programa 4. Suma de vectores

```
attributes(global) subroutine vecAdd_kernel(n, a, b, c)
integer, value ::  n
 real, device ::  a(n), b(n), c(n)
integer ::  id
id = threadidx%x
if (id <= n) then
c(id) = a(id) + b(id)
endif
end subroutine vecAdd_kernel
program main
real ::  sum
integer ::  i
!  Tamaño de los vectores
integer ::  n = 100
real,dimension(:),allocatable ::  h_a
real,dimension(:),allocatable ::  h_b
real,dimension(:),allocatable ::  h_c
real,device,dimension(:),allocatable ::  d_a
real,device,dimension(:),allocatable ::  d_b
real,device,dimension(:),allocatable ::  d_c
allocate(h_a(n))
allocate(h_b(n))
allocate(h_c(n))
allocate(d_a(n))
allocate(d_b(n))
allocate(d_c(n))
!inicializar los vectores
do i=1,n
h_a(i) = i
h_b(i) = i
enddo
```

Iniciaremos con la explicación del código host (también llamado código principal) y

(continuación)
```
d_a = h_a(1:n)
d_b = h_b(1:n)
call vecAdd_kernel<<<1,N>>>(n, d_a, d_b, d_c)
h_c = d_c(1:n)
print *, 'resultado de la suma', h_c
deallocate(d_a)
deallocate(d_b)
deallocate(d_c)
deallocate(h_a)
deallocate(h_b)
deallocate(h_c)
end program main
```

luego el código device (código kernel).

Código host

El código host en este ejemplo lo hemos llamado *main*, luego hemos definido el tamaño del vector con *integer :: n = 100*. Después se declaran los vectores de entrada y salida del host: *h_ a, h_ b* y *h_ c* respectivamente; y para los vectores del device (declarar con el atributo device) son: *d_ a, d_ b* y *d_ c*. Posteriormente se reserva la memoria en el host y en el device para los tres arreglos de tamaño n, con la instrucción *allocate*. Finalmente se realiza el llenado de los vectores *h_ a* y *h_ b* que se encuentran almacenados en el host. Una vez realizado esto se procede con los pasos para lanzar un kernel. Como primer paso, se copia los datos de la CPU a la GPU como *d_ a = h_ a(1:n)* y *d_ b = h_ b(1:n)*. Cuando los datos se encuentran en la memoria del device GPU, se procede con el segundo paso que es el lanzamiento del kernel. En este caso lo hemos definido como *call vecAdd_ kernel<<<1,N>>>(n, d_ a, d_ b, d_ c)*. La configuración de este kernel es de 1 bloque con n hilos. Luego se ejecuta el kernel cuando haya terminado y se copia el vector resultado de la GPU hacia la CPU con *h_ c = d_ c(1:n)*. Una vez realizado esto, se puede imprimir el resultado. Por último, se libera la memoria tanto en la CPU como también en la GPU con la instrucción *deallocate*.

Código device

Como se mencionó desde un principio, necesitamos definir un kernel para ejecutar los hilos en paralelo, responsables de calcular la suma de vectores. Nuestro kernel en este caso lo hemos llamado *vecAdd_kernel*; y en los paramétros le hemos pasado las matrices fuente *a* y *b*, y la matriz destino *c*. Posteriormente se declara el identificador de hilo con *id*, de una dimensión. Para este problema y de la forma que fue planteado, sólo esta dimensión es suficiente para identificar a cada hilo. Luego se indica la operación de suma paralela de los hilos con *c(id) = a(id) + b(id)*. En el momento de lanzar el kernel se genera una malla con tantos hilos como son indicados en la invocación. Los hilos se identificarán con un índice que depositan el valor de su identificador en la misma localidad del arreglo (vector), para así proceder con el cálculo de operación suma simultáneamente. De acuerdo a la configuración del kernel, se ha definido un sólo bloque de N hilos, por lo que el índice de hilo inicia de 1 hasta n dentro del bloque. Ver Fig. 20.

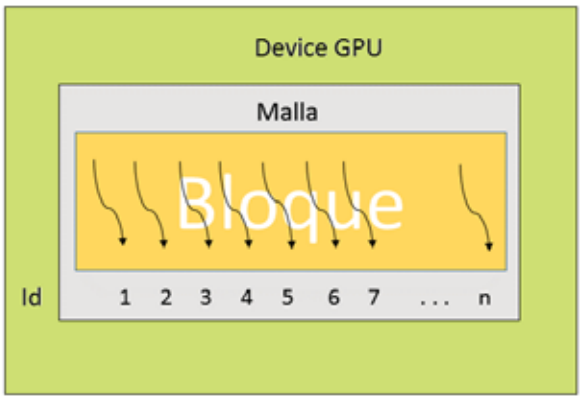

Figura 20. Malla con un bloque unidimensional de N hilos identificados con un índice.

El programa implementado anteriormente, es un ejemplo básico a seguir a la hora de paralelizar un programa en CUDA FORTRAN, en realidad es como el "hola mundo" en CUDA. Por lo tanto, para otras implementaciónes que veremos más adelante seguiremos

estos pasos descritos en este ejemplo, salvo que cambian en el diseño y configuración del kernel.

IV.6.2. Multi-bloques y Multi-hilos

En el código device del **programa 4**, se puede observar que no existe referencia al identificador de bloque, esto obedece a que la solución fue desarrollada para realizar la suma de un vector con tantas componentes como lo indique la máxima cantidad de hilos por bloque. Pero en la sección IV.2.2 vimos que nuestro hardware limita el número de hilos por bloque. Es este caso la máxima cantidad de hilos por bloque es 1024. Así que la pregunta es ¿qué pasa si los vectores tienen más de *maxThreadsPerBlock* componentes? La solución es que tendremos que usar una combinación de hilos y bloques. Así para este caso, la fórmula para calcular el identificador de hilo (global) (Ruetsch y Fatica, 2011), usando bloques e hilos es: *id = (blockidx%x-1)*blockdim%x + threadidx%x*.

Con esta nueva solución puede observarse la utilización de la variable predefinida *blockdim*, la cual contiene la cantidad de hilos del bloque según la dimensión. Como todos los bloques tienen la misma cantidad y organización de los hilos, ésta será constante para todos los bloques de una malla. Lo mismo ocurre en el caso de las dimensiones de la malla, la variable *gridDim*.

Para entender esta idea, consideremos un ejemplo sencillo. Supongamos que tenemos un arreglo (vector) con $n = 16$ elementos, de esta forma necesitamos 16 hilos. Para ello podemos crear 4 bloques, tal que en cada bloque contenga 4 hilos (recordemos que en este caso los hilos y los bloques son en una dimensión), como se ilustra en la Fig. 21

Una vez que hemos definido la cantidad de bloques y la cantidad de hilos por bloque, podemos obtener el identificador de hilo global que una vez más es *id = (blockidx%x-1)*blockdim%x + threadidx%x*. Esta expresión, *blockidx%x* es el identificador de bloque dentro de la malla, *threadidx%x* es el identificador de hilo dentro del bloque y *block-*

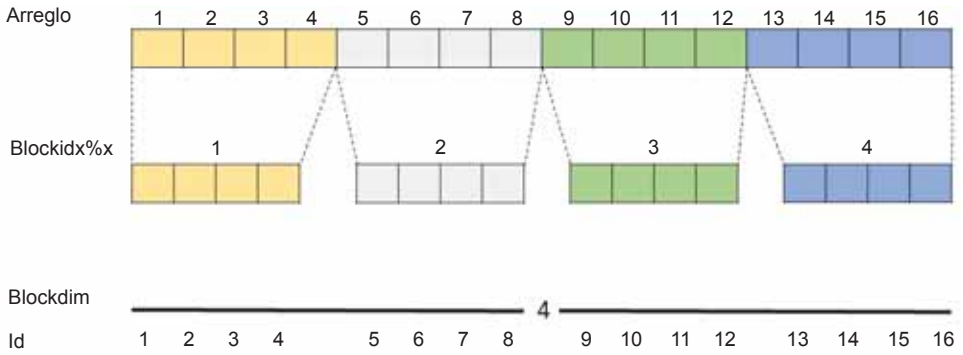

Figura 21. Configuración para el identificador de hilo global, usando bloques e hilos.

dim%x proporciona el tamaño del bloque, es decir la cantidad de hilos que hay en el bloque. De esta forma, comencemos identificando los hilos del primer bloque con la fórmula que en este caso $blockdim\%x = 4$:

Para el hilo 1: $Id = (1-1) * 4 + 1 = 1$

Para el hilo 2: $Id = (1-1) * 4 + 2 = 2$

Para el hilo 3: $Id = (1-1) * 4 + 3 = 3$

Para el hilo 4: $Id = (1-1) * 4 + 4 = 4$

Para el bloque 2 tenemos:

Para el hilo 1: $Id = (2-1) * 4 + 1 = 5$

Para el hilo 2: $Id = (2-1) * 4 + 2 = 6$

Para el hilo 3: $Id = (2-1) * 4 + 3 = 7$

Para el hilo 4: $Id = (2-1) * 4 + 4 = 8$

De la misma manera se calcula para los demás bloques, salvo que cambia con los identificadores del bloque. Por lo tanto con esta forma de combinar bloques e hilos, podemos sumar vectores con elementos mayores de 1024 hasta el límite permitido por nuestro hardware, que en este caso es: $2147483647X1024 = 2.199023255X10^{12}$ elementos como máximo.

El ejemplo presentado anteriormente, trabaja sobre una estructura de datos vecto-

rialmente, lo cual sólo hace necesario tener una dimensión tanto de bloques como de hilos. La pregunta ahora es: ¿Cómo se hace para activar la malla y los bloques de dimensiones mayores a 1? La respuesta se detalla en las siguientes líneas de código, las cuales se incluyen en el código host y antes de la llamada al kernel.

my_blocks = dim3(128,100,1)

my_threads = dim3(16,16,1)

call kernel<<<my_blocks, my_threads>>>(...args...)

En estas líneas de código se define un ambiente de ejecución mediante dos variables de tipo *dim3*, my_blocks y my_threads. La primera describe la configuración de la malla, en este caso 128x100 bloques. La segunda variable específica las características de cada bloque de 16x16 hilos. Finalmente se invoca al kernel con la configuración establecida, en este caso 12800 bloques con 256 hilos cada uno.

Capítulo V

IMPLEMENTACIÓN PARALELA EN PROBLEMAS DE ÁLGEBRA LINEAL

En este capítulo se muestran las implementaciones paralelas puestas en práctica en aplicaciones de álgebra lineal. Así mismo se muestran los resultados del tiempo de cómputo obtenidos por los distintos programas en forma secuencial y en paralelo.

V.1. Suma de matriz

En esta sección implementaremos el programa para calcular la suma de matriz en CUDA FORTRAN, en la cual explicaremos brevemente los pasos llevados a cabo en el programa. Al final mostraremos los resultados numéricos obtenidos al calcular el tiempo de cómputo, tanto en la versión paralela como también en la versión secuencial con CPU.

Antes de iniciar con el código, recordemos que CUDA FORTRAN es un modelo de programación heterogenéo en el que utiliza tanto tanto la CPU y como la GPU. Por lo tanto, para la exposición de este programa en paralelo lo dividiremos en dos partes, código host y código device. En el **programa 5** se muestra el código completo usado para la suma de matrices en CUDA FORTRAN.

Programa 5. Código para la suma de matriz

```
module matrizSum
contains
attributes(global) subroutine SumaMatrices_kernel(A,B,C,fila,col,N)
implicit none
real, device ::  A(N),B(N),C(N)
integer, value ::  fila,col,N
integer ::  idx,idy,id
idy=(blockidx%x-1)*blockdim%x + threadidx%x
idx=(blockidx%y-1)*blockdim%y + threadidx%y
id=(idx-1)*col + idy
if(idx<=fila .AND. idy<=col)C(id)=A(id)+B(id)
end subroutine
end module
program sumaMatriz
use cudafor
use matrizSum
integer ::  fila,col,N,i,j
parameter (fila=800,col=800) !Tamaño de la matriz
parameter (N=fila*col)
type(dim3) ::  blockSize,gridSize
!Creamos memoria en el host
real,allocatable ::  A(:),B(:),C(:)
!Declaracion de memoria en el dispositivo
real, device, allocatable ::  Ad(:),Bd(:),Cd(:)
allocate(A(N),B(N),C(N))
!Creamos memoria en el device
allocate(Ad(N),Bd(N),Cd(N))
!Inicializamos la matriz
do i=1,fila
do j=1,col
A((i-1)*col+j)= i
B((i-1)*col+j)= i+5
enddo
enddo
Ad=A(1:N)
Bd=B(1:N)
blockSize = dim3(8,8,1) !numero de hilos por bloque...
gridSize = dim3(100,100,1)
call SumaMatrices_kernel<<<gridSize,blockSize>>>(Ad,Bd,Cd,fila,col,N)
C(1:N)=Cd
!write (*,*),"C: ",C
deallocate(A,B,C)
deallocate(Ad,Bd,Cd)
end program sumaMatriz
```

Como se observa el módulo, contiene la subrutina *SumaMatrices_kernel* que es el kernel que se ejecuta en la GPU (device) y el programa *sumaMatriz*, es el código de la CPU (host). Iniciaremos con la explicación del código host.

Código host

El programa *SumaMatriz* utiliza dos módulos: *use matrizSum* y *use cudafor*. El módulo *matrizSum* contiene al kernel y *cudafor* contiene algunas funciones especiales de CUDA FORTRAN. En esta parte también definimos un nuevo tipo de datos para el programa: *type(dim3)*. Es un tipo implícito de CUDA FORTRAN y no es un tipo de FORTRAN estándar, que permite crear una tupla tridimensional vista en el capítulo IV.

También hemos declarado variables para definir tres conjuntos de matrices como: *real,allocatable :: A(:),B(:),C(:)* y *real, device, allocatable :: Ad(:),Bd(:),Cd(:)*. Las matrices reales A, B y C son las matrices del host y son declaradas de manera normal tal como se hace en FORTRAN 90. A y B son las matrices que queremos sumar y C es la matriz resultado. Las matrices Ad, Bd y Cd son las matrices del device, declaradas con el atributo (device). Las matrices Ad y Bd son los encargados de llevar a las matrices A y B respectivamente a la memoria de la GPU, y la matriz Cd almacena el resultado calculado en la GPU, para luego enviar al host con la matriz C. Posteriormente se reserva memoria en el host y en el device para los tres arreglos de tamaño N, y se reservan con la instrucción *allocate*. A continuación, se asignan las matrices de entrada *A((i-1)*col+j)* y *B((i-1)*col+j)*. Luego se procede a la transferencia de datos del host al device y la declaración en este caso es: *Ad=A(1:N)* y *Bd=B(1:N)*. El siguiente paso es llamar al kernel *SumaMatrices_kernel* y la declaración es: *call SumaMatrices_kernel<<<gridSize,blockSize>>>(Ad,Bd,Cd,fila,col,N)*.

Como penúltimo paso, se lanzan los hilos en el kernel, para ello hemos llamado a

la dimensión de la malla con el nombre *gridSize* y la del bloque con *blockSize*, lo cual procesaran todos los N elementos de la matriz:

blockSize = dim3(8,8,1)

gridSize = dim3(100,100,1).

Como último paso importante es la transferencia de datos del dispositivo al host, para obtener el resultado: *C(1:N)=Cd.*

Código device

Pasamos ahora al código del kernel que una vez más es como se observa en el **programa 6**. Los tres primeros argumentos de este kernel corresponden a las matrices del device y como ya dijimos son declaradas en el host. Sin embargo, los tres últimos argumentos; *fila*, *col* y N no se transfiere del dispositivo al código de host, debido a que FORTRAN pasa argumentos por referencia en lugar del valor.

Programa 6. Código del kernel para la suma de matrices

```
!************************************************************************
attributes(global) subroutine SumaMatrices_kernel(A,B,C,fila,col,N)
implicit none
real, device ::  A(N),B(N),C(N)
integer, value ::  fila,col,N
integer ::  idx,idy,id
idx=(blockidx%y-1)*blockdim%y + threadidx%y
idy=(blockidx%x-1)*blockdim%x + threadidx%x
id=(idx-1)*col + idy
if(idx<=fila .AND. idy<=col)C(id)=A(id)+B(id)
end subroutine
```

Después de las declaraciones de las variables, nos queda declarar los identificadores de hilo. Como se mencionó en el capítulo anterior, el kernel es ejecutado por múltiples hilos en paralelo. Si queremos que cada hilo procese un elemento de la matriz resultante, entonces necesitamos distinguir e identificar cada hilo. Esto se logra a través de las variables predefinidas tal como se vió en la sección IV.4.1. Y luego usamos la expresión para el identificador de hilo global vista la sección IV.6.2. En este caso para una matriz:

*idx =(blockidx%y-1)*blockdim%y + threadidx%y*

*idy =(blockidx%x-1)*blockdim%x + threadidx%x*

Estos generan un índice que se utiliza para acceder a los elementos de la matriz. Recordando que en CUDA FORTRAN, los componentes de threadIdx y blockIdx han compensado la unidad, por lo que el primer hilo en un bloque tiene *threadIdx%x=1* y el primer bloque en la malla tiene *blockIdx%x=1*. Por último, se declara el índice total: *id=(idx-1)*col + idy*, que es el índice del elemento de las matrices que se quieren calcular. Finalmente se hace la operación suma en el kernel con *C(id)=A(id)+B(id)*.

V.1.1. Tiempos de cómputo para la programación secuencial y en paralelo

En la tabla 2 se muestra el tiempo de cómputo obtenido en la implementación del programa de suma de matrices en su forma secuencial y en paralelo previamente descrito. En ambos casos se usó el mismo algoritmo y el mismo tamaño de datos. Se midieron los tiempos de ejecución en segundos para matrices cuadradas de diferentes tamaños, en la CPU y en la GPU. La cuarta columna corresponde a la razón del tiempo de cómputo en forma secuencial con el caso en paralelo.

Para la implementación del programa con CUDA FORTRAN, usamos bloques de $8X8$; es decir, cada bloque contendrá 64 hilos. Este valor se tomó como parámetro fijo para los distintos tamaños de la matriz. Para la dimensión de la malla, lo fuimos variando de acuerdo al tamaño de la matriz.

Por ejemplo, iniciamos con la matriz de $800X800 = 640000$ elementos con *blockSize=dim3(8,8,1)=64 hilos* y *gridSize=dim3(100,100,1)=10000 bloques*. Por lo tanto, 10000 bloques X 64 hilos = 640000 hilos equivalen a los elementos de la matriz. Similarmente, se hizo hasta el tamaño de la matriz de $6400X6400$.

Tabla 2. Tiempo de cómputo para la suma de matrices

Matriz de entrada	Tiempo de ejecución CPU	Tiempo de ejecución GPU	Razón
800x800	0.010 s	0.004 s	2.5
1200x1200	0.018 s	0.009 s	2
1600x1600	0.035 s	0.014 s	2.5
2000x2000	0.041 s	0.023 s	1.7
2400x2400	0.060 s	0.033 s	1.8
2800x2800	0.080 s	0.045 s	1.7
3200x3200	0.100 s	0.059 s	1.6
3600x3600	0.120 s	0.073 s	1.6
4000x4000	0.150 s	0.093 s	1.6
4400x4400	0.193 s	0.101 s	1.9
4800x4800	0.220 s	0.120 s	1.8
5200x5200	0.260 s	0.150 s	1.7
5600x5600	0.307 s	0.170 s	1.8
6000x6000	0.351 s	0.200 s	1.7
6400x6400	0.390 s	0.231 s	1.6

En la tabla 2, podemos observar los tiempos de cómputo para ambas versiones, con sus respectivas razones entre ellas mismas. Haciendo el cálculo de la razón promedio R_p entre la versión secuencial y en paralelo, se obtiene que $R_p = 1.83$. Esto significa que con la versión paralela con CUDA FORTRAN, reduce el tiempo de cómputo casi 2 veces en comparación con el caso secuencial en la CPU. Para mayor claridad se muestra la gráfica de los tiempos de cómputo como función del tamaño de la matriz para ambos casos de programación en la Fig. 22.

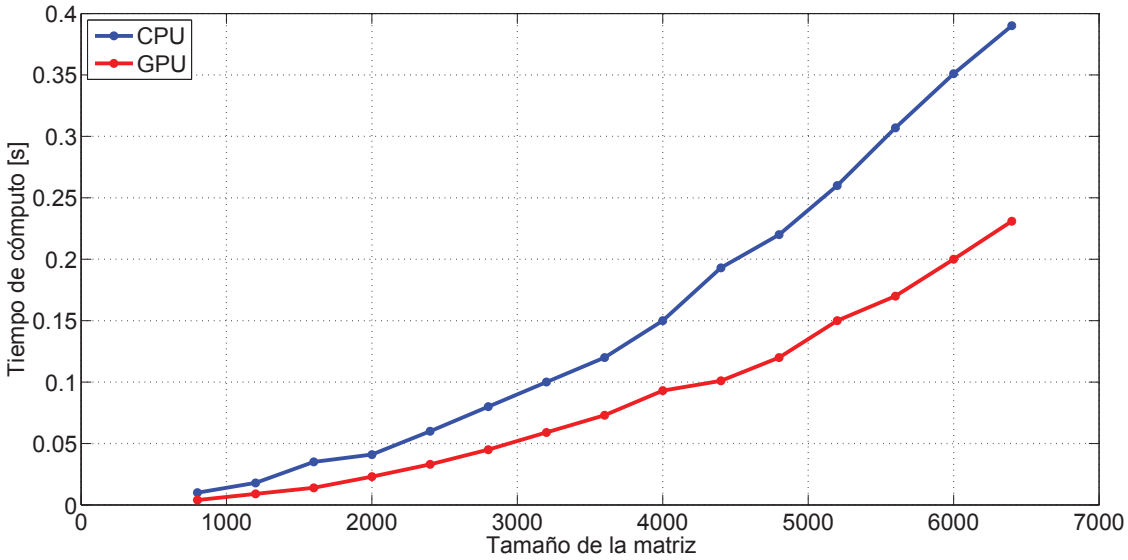

Figura 22. Comparación del tiempo de cómputo entre el programa secuencial (CPU) y programa paralelo (GPU) con CUDA FORTRAN.

Para la implementación en paralelo, se trabajó con matrices más grandes, pero no se logró reducir más el tiempo de cómputo. Por lo que para esta aplicación se puede ver que se ha liberado gradualmente poco la potencia de cálculo en la GPU. Por lo tanto, es importante mencionar que la eficiencia de CUDA se tiene para problemas con datos muy grandes y complejos como veremos más adelante. En la siguiente sección cuando tratemos la operación de multiplicación de matrices veremos como cambia este panorama.

V.2. Multiplicación de matriz

La multiplicación de matrices es un problema importante en álgebra lineal. Su característica es que su algoritmo es equivalente a una variedad de problemas. Por lo tanto, en esta sección vamos a evaluar el rendimiento de la multiplicación de matrices bajo la programación en su forma secuencial y en paralelo.

V.2.1. Algoritmo secuencial

Empezaremos recordando que dos matrices A y B son multiplicables si el número de columnas de A coincide con el número de filas de B. En el **programa 7** mostramos la implementación clásica del algoritmo para la multiplicación de matrices en su versión secuencial con la CPU.

Programa 7. Algoritmo clásico para la multiplicación de matriz

```
!**********************************************************************
do i=1,n
   do j=1,n
       sum=0.0
         do k=1,n
          sum=sum+A(i,k)*B(k,j)
         C(i,j)=sum
        enddo
    enddo
enddo
```

Como se puede observar este método consta de dos bucles anidados donde se van calculando los elementos C_{ij}. Esto nos permite diseñar el algoritmo en su forma paralelizable como a continuación lo veremos.

Para el caso de la programación en paralelo, primero explicaremos brevemente el proceso de paralelización de la multiplicación de matrices cuadradas NXN en su versión básica con CUDA FORTRAN. Y en la siguiente subsección se verá como optimizar esta aplicación usando memoria compartida.

V.2.2. Multiplicación de matrices usando la versión básica

Para la implementación de programa en paralelo con CUDA FORTRAN, sólo mostraremos el código device; ya que el código host para este programa tiene una forma muy similar al caso de la suma de matrices vista en la sección V.1.

Necesitamos un kernel, donde cada hilo calcula un elemento de la matriz resultado.

cada uno deberá leer una fila entera de la matriz A y la columna correspondiente de la matriz B, operarlas y escribir el valor resultante en la matriz C.

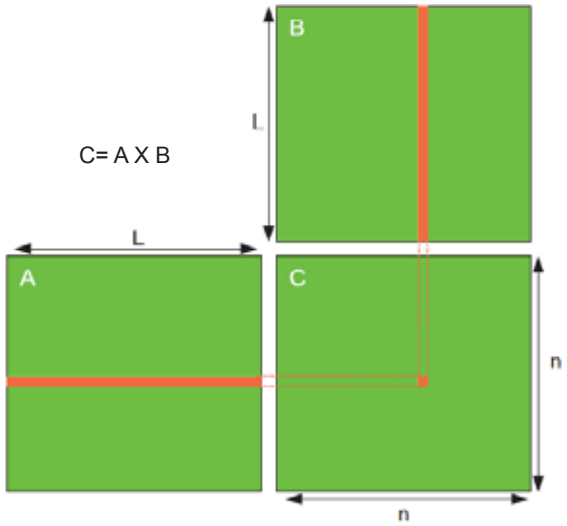

Figura 23. Multiplicación de matrices. Tarea asignada a cada hilo.

En la Fig. 23, se muestra un esquema de como calcular el elemento (i, j) de la matriz resultado. Cada hilo accede L veces a la matriz A (una fila entera) y L veces a la matriz B (una columna entera) y escribe una vez en la matriz C. Es decir $2L$ lecturas de memoria global y una escritura en memoria global por hilo. Para esta implementación, hay muchos accesos a la memoria global lo que limita la velocidad de cómputo (Webster, 2012).

Código device

Como se mencionó desde un principio, necesitamos definir un kernel para ejecutar los hilos en paralelo, responsables de calcular la multiplicación de matrices. En el programa 8 se muestra el código. En la implementación del código kernel, cada hilo calcula el elemento C_{ij} que queda determinado por el identificador de bloque-hilo visto en la sección IV.4.1.

Programa 8. Código del Kernel para la multiplicación de matrices en la versión básica.

```
!*********************************************************************
attributes(global) subroutine kernelm(A,B,C,m,n)
implicit none
real,dimension(:,:)::A,B,C
integer, value ::  m,n
integer ::  i,j,k
real::  sum
!Identificador de hilo
i=(blockidx%y-1)*blockdim%y + threadidx%y
j=(blockidx%x-1)*blockdim%x + threadidx%x
     if(i<=n .AND. j<=m)then
sum=0.0
do k=1,m
       sum=sum+A(i,k)*B(k,j)
enddo
C(i,j)=sum
     endif
end subroutine
```

V.2.3. Multiplicación de matriz usando memoria compartida

En la sección III.4.1 expusimos que los accesos a la memoria compartida son más rapidos que los accesos a la memoria global. Por esta razón en esta parte explicaremos brevemente la implementación de la multiplicación de matrices usando la memoria compartida.

Una estrategia común, es la partición de los datos en subconjuntos de submatrices (TILES) de modo que cada submatriz se ajusta en la memoria compartida. En este caso, cada bloque de hilos es responsable de calcular una submatriz para una matriz C y cada hilo será responsable de calcular un elemento de la dicha submatriz, que tiene la misma dimensión que el tamaño del bloque. Hay que tener en cuenta que el tamaño de la memoria compartida es bastante pequeña y por lo tanto, hay que procurar no exceder la capacidad de la memoria al cargar los elementos de una matriz A y una matriz B.

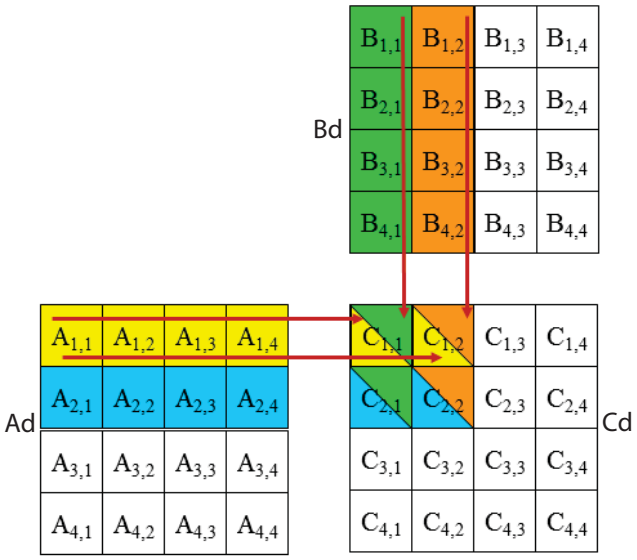

Figura 24. Submatrices de Ad y Bd que utilizan memoria compartida.

Tomemos como ejemplo las matrices de la Fig. 24, dividiendo la matriz Ad y Bd en submatrices de $2X2$. De esta forma los cálculos del producto escalar (fila Ad por columna de Bd) realizado por cada hilo, estarán divididas en dos fases. En cada fase, todos los hilos en un bloque colaboran para una submatriz de Ad y una submatriz de Bd para llevarlos a la memoria compartida.

Hilos	Fase 1			Fase 2		
H_{11}	Ad_{11} ↓ Ads_{11}	Bd_{11} ↓ Bds_{11}	$Cvalor_{11} = Ads_{11} * Bds_{11} + Ads_{12} * Bds_{21}$	Ad_{31} ↓ Ads_{11}	Bd_{13} ↓ Bds_{11}	$Cvalor_{11} = Ads_{11} * Bds_{11} + Ads_{12} * Bds_{21}$
H_{21}	Ad_{21} ↓ Ads_{21}	Bd_{21} ↓ Bds_{21}	$Cvalor_{21} = Ads_{11} * Bds_{21} + Ads_{21} * Bds_{22}$	Ad_{41} ↓ Ads_{21}	Bd_{23} ↓ Bds_{21}	$Cvalor_{21} = Ads_{11} * Bds_{21} + Ads_{21} * Bds_{22}$
H_{12}	Ad_{12} ↓ Ads_{12}	Bd_{12} ↓ Bds_{12}	$Cvalor_{12} = Ads_{12} * Bds_{11} + Ads_{22} * Bds_{12}$	Ad_{32} ↓ Ads_{12}	Bd_{14} ↓ Bds_{12}	$Cvalor_{12} = Ads_{12} * Bds_{11} + Ads_{22} * Bds_{12}$
H_{22}	Ad_{22} ↓ Ads_{22}	Bd_{22} ↓ Bds_{22}	$Cvalor_{22} = Ads_{12} * Bds_{21} + Ads_{22} * Bds_{22}$	Ad_{42} ↓ Ads_{22}	Bd_{24} ↓ Bds_{22}	$Cvalor_{22} = Ads_{12} * Bds_{21} + Ads_{22} * Bds_{22}$

Figura 25. Accesos a la jerarquía de memoria para la multiplicación de matrices.

En la Fig. 25, se muestra la ejecución de actividades de un hilo. En este ejemplo sólo vamos a mostrar la ejecución de hilos para el $block(1,1)$, que corresponderia a la submatriz marcado en colores en la matriz Cd; para los demás bloques tienen un comportamiento similar. Las ubicaciones en la memoria compartidas para los elementos Ad podemos definir como Ads y para los elementos Bd es Bds. De esta forma, en la fase 1, los cuatro hilos del $bloque(1,1)$ colaboran para llevar una submatriz de Ad a la memoria compartida. El $hilo(1,1)$ carga a $Ad(1,1)$ en $Ads(1,1)$, el $hilo(2,1)$ carga a $Ad(2,1)$ en $Ads(2,1)$, el $hilo(1,2)$ carga a $Ad(1,2)$ en $Ads(1,2)$ y el $hilo(2,2)$ carga a $Ad(2,2)$ en $Ads(2,2)$. Una submatriz de Bd también es cargado de manera similar.

De esta forma, en cada fase, los productos de dos pares de los elementos de matriz de entrada se acumulan en la variable $Cvalor$; por ejemplo, el resultado total en la memoria compartida de $C_{11}valor$ es $Ads_{11} * Bds_{11} + Ads_{12} * Bds_{21} + Ads_{11} * Bds_{11} + Ads_{12} * Bds_{21}$. Para un caso arbitrario donde la matriz de entrada es de dimensión N y el tamaño de la submatriz es $TILE\_WIDTH$, el producto escalar se llevaría a cabo en fases $N/TILE\_WIDTH$. Por ejemplo, supongamos que tenemos una matriz entrada de $N = 100$; es decir, una matriz de $100X100$. A esta manera si definimos que $TILE\_WIDTH = 4$ entonces $Submatriz = TILE\_WIDTHXTILE\_WIDTH$. Por lo tanto, $N/TILE\_WIDTH = 25$, y significa que el proceso del producto escalar se efectuará en 25 fases.

La solución de la multiplicación de matrices utilizando la memoria compartida presenta ciertas ventajas. Una de ellas es la reducción del acceso a la memoria global. Además acelera la solución y los accesos a los mismos elementos son hechos en la memoria compartida de manera más rápida que la global.

Código device

Como se muestra en el **programa 9**, hemos creado dos submatrices en memoria

Programa 9. Código del kernel para la multiplicación de matrices con memoria compartida

```
!*********************************************************************
attributes(global) subroutine mmul_kernel( A, B, C, N, M, L )
real,device ::  A(N,M), B(M,L), C(N,L)
integer, value ::  N, M, L
integer ::  i, j, kb, k, tx, ty
real, shared ::  Asub(32,32), Bsub(32,32) !Se declaran las submatrices
real ::  Cij
tx = threadidx%x
ty = threadidx%y
!Este hilo calcula C(i,j) = sum(A(i,:)  * B(:,j))
i = (blockidx%x-1) * 32 + tx
j = (blockidx%y-1) * 32 + ty
Cij = 0.0
!Se hace el ciclo en k en trozos de 32
do kb = 1, M, 32
!Llenar las submatrices
Asub(tx,ty) = A(i,kb+ty-1)
Bsub(tx,ty) = B(kb+tx-1,j)
call syncthreads()
do k = 1,32
 Cij = Cij + Asub(tx,k) * Bsub(k,ty)
enddo
call syncthreads()
enddo
C(i,j) = Cij
end subroutine mmul_kernel
```

compartida, *Asub* y *Bsub*. Ésta es únicamente compartida entre los hilos de un mismo bloque, por lo que todos ellos tendrán acceso a la submatriz de la que se ocupa el bloque. En este caso el tamaño de la submatriz lo hemos definido de $32X32$, que es equivalente al tamaño del bloque, y los identificadores de hilos los hemos declarado como: *i = (blockidx%x-1) * 32 + tx* y *j = (blockidx%y-1) * 32 + ty*; similarmente a como veníamos declarando anteriormente.

V.2.4. Multiplicación de matrices usando la librería CUBLAS

CUBLAS[1] (NVIDIA, 2007) es una implementación de BLAS para la plataforma de CUDA NVidia, que permite el acceso a los recursos de las GPUs. CUBLAS no necesita ser instalado, ya que es una biblioteca que se crea al momento de instalar CUDA Toolkit. Para compilar y ejecutar programas que usen dicha biblioteca se debe especificar en la sentencia de compilación. Por ejemplo: *$ pgfortran -o miprograma miprograma.cuf -lcublas.*

El modelo básico de las aplicaciones CUBLAS consiste en:

- Crear matrices y vectores en memoria de CPU.
- Transferir los datos desde memoria de CPU a memoria de device (GPU).
- Invocar funciones CUBLAS para realizar operaciones.
- Transferir los resultados desde memoria de GPU a memoria de host (CPU).

Este trabajo sólo se ilustrará la subrutina de CublasSgemm que utilizamos.

Función cublasSgemm

Es una función para precisión simple y realiza el producto de dos matrices. El prototipo de la rutina cublasSgemm es:

[1] Por sus siglas en inglés, CUDA Basic Linear Algebra Subroutines.

*cublasSgemm(char transA,char transB,int m,int n,int k,float alpha,const float *A,int lda,const float *B,int ldb,float beta,float *C,int ldc)*

Entrada:

transA: especifica la operación que será realizada: Si *transA*==‘N’ o ‘n’ se opera con A. Si *transA*==‘T’ o ‘t’ o ‘C’ o ‘c’ se opera con AT.

transB: especifica la operación que será realizada: Si *transB*==‘N’ o ‘n’ se opera con B. Si *transB*==‘T’ o ‘t’ o ‘C’ o ‘c’ se opera con BT .

m: es el número de filas de la matriz A y el número de filas de la matriz C.

n: es el número de columnas de la matriz B y el número columnas de la matriz C.

k: es el número de columnas de la matriz A y el número de filas de la matriz B.

alpha: es un escalar de precisión simple.

A: es una matriz de precisión simple de dimensión (lda, K) si *TransA*==‘N’ o ‘n’ y (lda, m) de otra manera.

lda: dimensión de la matriz A.

B: es una matriz de precisión simple de dimensión (lda, N) si *TransB*==‘N’ o ‘n’ y *(ldb, K)* de otra manera.

ldb: dimensión de la matriz B.

beta: es un escalar de precisión simple.

C: es una matriz de precisión simple de dimensión (lda, N).

ldc: dimensión que será usada para almacenar la matriz C.

Salida:

C: $\texttt{C} = \alpha \times \texttt{A} \times \texttt{B} + \beta \times \texttt{C}$

Ahora implementaremos el programa de la múltiplicación de matrices usando esta función de CUBLAS.

Como se puede observar en el **programa 10**, hemos creado las matrices en el host

Programa 10. Código para Multiplicación de matrices con CUBLAS

```
!************************************************************************
program multiplica_cublas
use cublas
use cudafor
implicit none
integer ::  i,j
integer , parameter ::  m = 30000 , n = 30000 , k = 30000
real,allocatable::  a(:,:),b(:,:),c(:,:)
real , device,allocatable::  a_d(:,:),b_d(:,:),c_d(:,:)
real , parameter ::  alpha = 1.0 , beta = 0.0
integer ::  lda = m, ldb = k, ldc = m
integer ::  istat
allocate (a(m,k))
allocate (b(k,n))
allocate (c(m,n))
allocate (a_d(m,k))
allocate (b_d(k,n))
allocate (c_d(m,n))
c = 0.0
do j = 1,k
 do i = 1,m
 a(i,j) = i
 enddo
enddo
do j = 1,n
 do i = 1,k
 b(i,j) = -(i+2)
 enddo
enddo
a_d = a
b_d = b
c_d = c
istat = cublasInit()
call cublasSgemm ('N','N',m,n,k,alpha ,a_d,lda ,b_d ,ldb ,beta, c_d ,&
& ldc)
c = c_d
deallocate(a,b,c)
deallocate(a_d,b_d,c_d)
end program multiplica_cublas
```

y en el device como: *real, allocatable:: a(:,:), b(:,:), c(:,:)* y *real, device, allocatable:: a_d(:,:), b_d(:,:), c_d(:,:).* Luego inicializamos las matrices de entrada que se van a multiplicar, que en este caso son *a(i,j)* y *b(i,j).* Como la función calcula la operación de la forma $\mathtt{C} = \alpha \times \mathtt{A} \times \mathtt{B} + \beta \times \mathtt{C}$, para nuestro caso se ha tomado a $\alpha = 1$ y a $\beta = 0$ para que sólo calcule $\mathtt{C} = \mathtt{A} \times \mathtt{B}$, tal como se quiere para este programa. Después se hace la transferencia de datos del host al device : $a\_d = a$, $b\_d = b$, $c\_d = c$ y se invoca la función *cublasSgemm* para realizar la operación de multiplicación. Por último copiamos el resultado del device al host con: $c = c\_d$.

V.2.5. Tiempo de cómputo para la programación secuencial y en paralelo

Ahora vamos a comparar los tiempos de cómputo para la multiplicación de matrices reales en precisión simple, usando las distintas versiones de programación en paralelo junto con la versión secuencial como se muestra en la tabla 3.

Tabla 3. Tiempos de cómputo para la multiplicación de matrices.

Matriz de entrada	Secuencial	Paralelo básico (Bloque $16X16$)	Paralelo con MC (Bloque $16X16$)	CUBLAS
1024x1024	13.1 s	3.4 s	3.1 s	2.9 s
1600x1600	45.9 s	2.5 s	3.2 s	3.0 s
2240x2240	126.6 s	3.6 s	3.3 s	3.1 s
3200x3200	454.0 s	4.1 s	3.6 s	3.2 s
4160x4160	808.1 s	5.0 s	4.1 s	3.3 s
5120x5120	3018.2 s	6.2 s	4.2 s	3.6 s
6400x6400	3100.0 s	8.0 s	6.4 s	4.1 s
7040x7040	4007.5 s	10.7 s	6.7 s	5.1 s

(continuación)	- - - - - - -	- - - - - - - - - - -	- - - - - - - - - - -	- - - - - - -
8000x8000	5927.3 s	13.2 s	8.5 s	5.6 s
9280x9280	9119.1 s	19.3 s	12.1 s	6.3 s
10240x10240	10060.2 s	24.7 s	14.4 s	7.1 s
11200x11200	10151.9 s	31.5 s	18.5 s	8.1 s
12480x12480	13835.1 s	42.2 s	24.0 s	8.7 s
13760x13760	18983.5 s	57.7 s	30.0 s	10.2 s

En la tabla 3, se muestran los resultados del tiempo de cómputo en segundos para las diferentes versiones implementadas para la multiplicación de matrices cuadradas de diferentes tamaños de NXN. Para la versión en paralelo básico y en paralelo con memoria compartida (MC) hemos usado bloques de $16X16$, y la dimensión de la malla la hemos ajustado de acuerdo con el tamaño de las matrices. Como se esperaba, en la tabla podemos observar que el tiempo de cómputo llevado a cabo por la versión secuencial es mucho más grande que en todas las otras versiones en paralelo para tamaño diferentes de las matrices. Para este ejemplo usamos las matrices definidas como $\mathrm{A}_{ij} = i$ y $\mathrm{B}_{ij} = -(i + 2)$.

Vamos a hacer una comparación para todas las versiones, que en particular tomaremos la matriz más grande $13760X13760$. Primero, con la versión en paralelo básico se obtuvo una rapidez de casi 329 veces con respecto a la versión secuencial; y la rapidez promedio considerando todos los datos es $R_p = 277.60$. Luego para el caso paralelo con memoria compartida (MC) se obtuvo una rapidez de más de 632 veces con respecto a la versión secuencial que en promedio es $R_p = 434.91$. Finalmente para el caso en paralelo con CUBLAS se obtuvo una rapidez de casi 1861 veces respecto a la versión secuencial, y que en promedio es $R_p = 818.22$. Para mayor claridad se muestran los

tiempos de cómputo en la Fig. 26(a) y la rapidez de cada forma de paralelización con respecto a la versión secuencial en la Fig. 26(b)

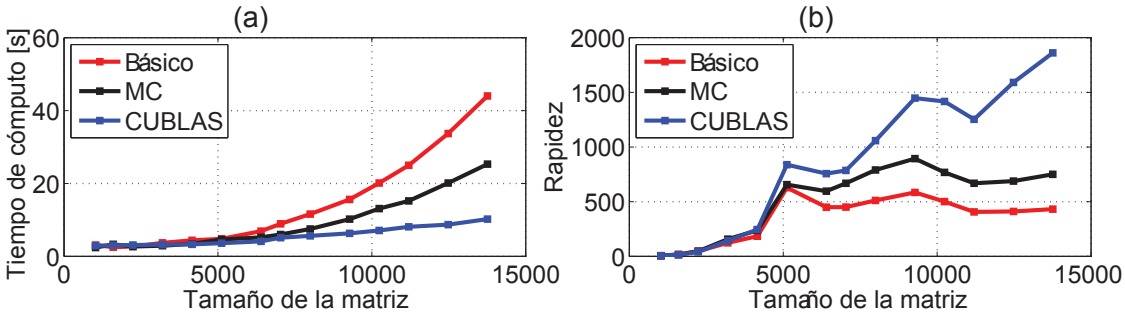

Figura 26. (a) Tiempo de cómputo en segundos de las versiones paralelas de la multiplicación de matrices: paralelo básico (línea color rojo), paralelo usando memoria compartida (MC) (línea color negro) y paralelo usando librería de CUBLAS (línea color azul). (b) Rapidez de las versiones paralelas en comparación con la versión secuencial.

En la Fig. 26(a) podemos apreciar que, para matrices grandes, la paralelización básico se tarda más en comparación con el caso de la MC. En cambio usando la librería de CUBLAS se toma mucho menos tiempo en comparación con las dos versiones anteriores. Por ejemplo, tomando en cuenta el tamaño de la matriz más grande de la Fig. 26(b), se tiene que con CUBLAS llega a ser 2.9 más rápido que la MC y 5.6 veces más rápido que la paralelización básica.

Ahora, queremos ver que pasa si usamos tamaños de bloques diferentes. En la Fig. 27 se muestran los tiempos de cómputo usando bloques de $16X16$ y $32X32$. Podemos ver que para matrices grandes, el caso paralelo básico con el bloque de $16X16$ (línea de color rojo), tarda más que con el bloque de $32X32$ (línea de color verde). Por otro lado, para la versión en paralelo con MC con el bloque de $32X32$ tarda menos en comparación con el bloque de $16X16$. Con esto podemos decir que con el bloque de $32X32$ se logra mejorar la rapidez. Pero finalmente nos damos cuenta que usando las librerías de CUBLAS es el más rápido comparado con todas las otras. Por lo tanto, concluimos que ésta es la implementación más eficiente para el cálculo de matrices en

CUDA FORTRAN.

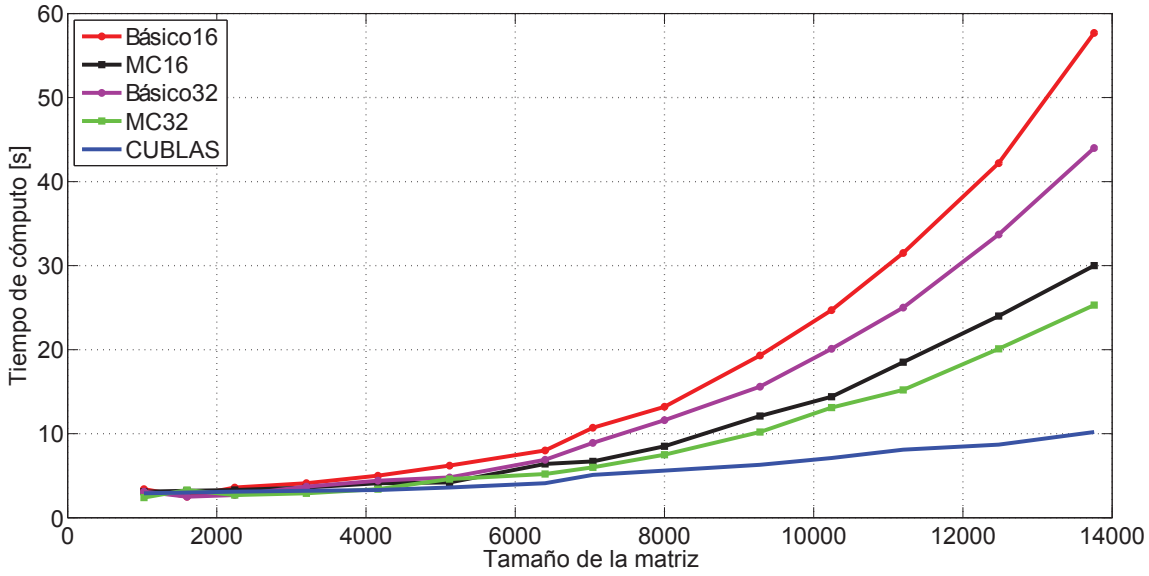

Figura 27. Comparación de las versiones paralelas usando bloques 16X16 y bloques de 32X32.

V.3. Inversión de matriz

La capacidad de invertir grandes matrices con rapidez y presición es un paso esencial en una amplia gama de problemas númericos; por ejemplo, la solución de ecuaciones lineales, la representación en 3D, filtrado de imágenes, entre otros.

V.3.1. Algoritmo secuencial con Gauss Jordan

Con el método de Gauss Jordan (Grossman y Flores, 2012) se puede calcular directamente la inversa de una matriz. Para hacerlo, la matriz de coeficientes se aumenta con una matriz identidad. Posteriormente se aplica el método de Gauss-Jordan para reducir la matriz de coeficientes a la matriz identidad. Cuando es completada esta tarea, el lado derecho de la matriz aumentada contiene la matriz inversa. Por lo tanto, el algoritmo tradicional para la implementación de un código secuencial es como se muestra en el **programa 11.**

Programa 11. Algoritmo secuencial para la inversión de matriz.

```
!**********************************************************************
!  Eliminación Gaussiana
do j = k + 1 , n
m = matriz_aumentada ( j , k ) / matriz_aumentada ( k , k )
do i = k , 2 * n
matriz_aumentada ( j , i ) = matriz_aumentada ( j , i ) - m *
matriz_aumentada ( k , i )
end do
end do
end do
!  Elementos de la diagonal igual a 1
do i = 1 , n
m = matriz_aumentada ( i , i )
do j = i , 2 * n
matriz_aumentada ( i , j ) = matriz_aumentada ( i , j ) / m
end do
end do
!  Reducir el lado izquierdo de la matriz aumentada a matriz identidad
do k = n - 1 , 1 , -1
do i = 1 , k
m = matriz_aumentada ( i , k+1)
do j = k , 2 * n
matriz_aumentada ( i , j ) = matriz_aumentada ( i , j ) -
matriz_aumentada ( k+1,j ) * m
end do
end do
end do
```

En el **programa 11**, hemos mostrado el algoritmo, ya que con ello implementamos nuestro programa secuencial (lo cual no se mostrará en este trabajo) y que nos servirá para el cálculo del tiempo de cómputo, para luego comparar con las versiones en paralelas.

V.3.2. Inversión de matriz en paralelo con Gauss Jordan

En este trabajo implementamos la programación en paralelo para la inversión de la matriz, usando también el método de Gauss Jordan. Para más detalle sobre el algoritmo paralelo de Gauss Jordan puede consultarse las Refs. (Sharma *et al.*, 2013; Soriano, 2011), donde parte de esa idea se tomó para esta aplicación.

Trabajaremos con matrices cuadradas NXN, para ello la matriz cuadrada debe ser aumentada con la matriz identidad de las mismas dimensiones, y luego aplicar las siguientes operaciones a la matriz: $[AI] \rightarrow A^{-1}[AI] \rightarrow [IA^{-1}]$. Nuestra aplicación sólo encuentra la inversa, si la matriz es no singular; es decir, si y sólo si la matriz identidad se puede obtener utilizando sólo las operaciones elementales de fila, de lo contrario, la matriz no es invertible. Aademás, se debe mencionar que nuestro algoritmo funciona sólo para entradas distintas de cero. A continuación se explica el método usado.

El método de eliminación de Gauss-Jordan consiste en las siguientes 2 operaciones:

1. Normalizar la fila pivote al valor pivote igual a 1.

2. Crear ceros por encima y por debajo de la fila pivote con las operaciones elementales de fila.

Por lo tanto, necesitamos desarrollar dos kernels, una para cada una de las operaciones descritas anteriormente. En el **programa 12** se muestra el código completo que daremos una breve explicación del algoritmo.

Programa 12. Algoritmo en paralelo para la inversión de matriz

```
MODULE inversa
use cudafor
contains
!************************************************************************
!  NORMALIZAR LOS PIVOTES DE LA FILA
!************************************************************************
attributes(global) subroutine normalizePivotRow(Matriz,Result,inde,& &
fila, n)
real,device::  Matriz(n),Result(n)
integer,value ::fila,inde,n
integer ::  idy,ty
real, shared ::  pivotValue
ty = threadidx%y !Posición de cada hilo dentro del bloque
idy =(blockidx%y-1)*LINEAR_BLOCK_SIZE + ty !Posición de cada hilo
dentro de la matriz
if(idy<=fila) then
if(ty==1) then
pivotValue = Matriz((inde-1)*fila + inde)
endif
call syncthreads()
Matriz((idy-1)*fila + inde) = Matriz((idy-1)*fila + inde)/pivotValue
Result((idy-1)*fila + inde) = Result((idy-1)*fila + inde)/pivotValue
endif
end subroutine normalizePivotRow
!************************************************************************
!  ELIMINAR EL RESTO DE LOS ELEMENTOS
!************************************************************************
attributes(global) subroutine linearMge(Matriz,Result,inde,fila,n)
real,device::  Matriz(n),Result(n)
integer,value ::fila,inde,n
integer ::  ty,x,idy
real ::  newMatrixValue,newResultValue
real, shared ::  multColumn(LINEAR_BLOCK_SIZE)
real, shared ::  matrixPivotValue
real, shared ::  matrixRow(LINEAR_BLOCK_SIZE)
real, shared ::  resultPivotValue
real, shared ::  resultRow(LINEAR_BLOCK_SIZE)
ty = threadidx%y
x = blockidx%x
idy = ty + LINEAR_BLOCK_SIZE*(blockidx%y-1)
if (idy <= fila) then
if (ty==1) then
```

(continuación)

```
matrixPivotValue = Matriz((x-1)*fila + inde)
resultPivotValue = Result((x-1)*fila + inde)
endif
multColumn(ty) = Matriz((inde-1)*fila + idy)
matrixRow(ty) = Matriz((x-1)*fila + idy)
resultRow(ty) = Result((x-1)*fila + idy)
call syncthreads()
if(idy.NE.inde) then
newMatrixValue = matrixRow(ty) - multColumn(ty)*matrixPivotValue
newResultValue = resultRow(ty) - multColumn(ty)*resultPivotValue
Matriz((x-1)*fila + idy) = newMatrixValue
Result((x-1)*fila + idy) = newResultValue
endif
endif
end subroutine linearMge
!************************************************************************
!CÓDIGO HOST
!************************************************************************
PROGRAM Inversion_Noviembre
use cudafor
use inversa
integer::  i,j,n,inde,roundup, minimo
integer,parameter ::fila= 96
parameter (n=fila*fila)
real ctimekernel1, ctimekernel2
integer c1, c2, c3, c4
real::  input(n),output(n)
real, device, allocatable ::  matrix_dev(:),result_dev(:)
type(dim3) ::  lmgeGrid,crGrid,crBlock,nprGrid,linearBlock
allocate( matrix_dev(n),result_dev(n))
lmgeGrid = dim3(fila,roundup(fila,LINEAR_BLOCK_SIZE),1)
crGrid = dim3(roundup(fila,SQUARE_BLOCK_SIZE),roundup(fila, & &
SQUARE_BLOCK_SIZE),1)
crBlock = dim3(minimo(fila,SQUARE_BLOCK_SIZE),minimo(fila, & &
SQUARE_BLOCK_SIZE),1)
nprGrid = dim3(1,roundup(fila,LINEAR_BLOCK_SIZE),1)
linearBlock = dim3(1,minimo(fila,LINEAR_BLOCK_SIZE),1)
!Se declara la matriz a invertir
do i=1,fila
do j=1,fila
 input((i-1)*fila+j)= real(1)/real(j+i-1)
enddo
enddo
```

(continuación)

```
!Hacer copia de CPU a Device
matrix_dev=input(1:n)
!Crear resultado
call createResult<<< crGrid, crBlock >>>( result_dev,fila,n)
do inde=1,fila
call system_clock( count=c1 )
call normalizePivotRow<<< nprGrid, linearBlock>>>( matrix_dev, & &
result_dev, inde,fila,n)
call system_clock( count=c2 )
call system_clock( count=c3 )
call linearMge<<< lmgeGrid, linearBlock >>>( matrix_dev, & &
result_dev,inde,fila,n)
call system_clock( count=c4 )
end do
!Hacer la copia de Device a CPU
output(1:n)=result_dev
ctimekernel1 = c2-c1
ctimekernel2 = c4-c3
!print*,"la matriz inversa es:"
!do i=1,fila
!print*,(output((i-1)*fila+j), j = 1, fila)
!enddo
!Liberar la memoria dinamica
print *, 'Kernel1 time excluding data xfer:', ctimekernel1*0.000001, &
& ' seconds'
print *, 'Kernel2 time excluding data xfer:', ctimekernel2*0.000001, &
& ' seconds'
deallocate(matrix_dev,result_dev)
END PROGRAM
!*********************************************************************
```

Recordemos que todo desarrollo de un programa en CUDA consta de dos partes, el código host (o muchas veces llamado tambien como código principal) y el código device (que contiene a la función kernel). Entonces iniciaremos con la explicación del código host.

Código host

En este caso nuestro código principal lo hemos llamado como *Inversion_ Noviembre*. Se han declarado las variables índices que nos servirán para hacer los ciclos en la declaración de la matriz. Después en la siguiente línea se define el tamaño de la matriz, en este caso *parameter (n=fila*fila).* Además, la parte importante es la declaración del conjunto de matrices que está definido como: *real:: input(n), output(n)* y *real, device, allocatable ::matrix_ dev(:), result_ dev(:).* Las matrices reales *input* y *output* son las matrices del host, que determinan la matriz de entrada (matriz a invertir) y la matriz de salida (matriz inversa) respectivamente. A las matrices *matrix_ dev* y *result_ dev* son las matrices del device que se ejecutan en el kernel.

Posteriormente se declara la matriz que queremos invertir. Para nuestro ejemplo, la matriz invertible es la matriz de Hilbert: *input((i-1)*fila+j)= real(1)/real(j+i-1).* Una vez hecho esto, el siguiente paso es enviar los datos a device y la forma es: *matrix_ dev=input(1:n)*, en la cual estamos pasando los datos de la matriz que queremos invertir. Ahora, el siguiente paso es llamar a los dos kernels como: *call normalizePivotRow<<< nprGrid, linearBlock>>>(matrix_ dev, result_ dev, inde,fila,n)* y *call linearMge<<< lmgeGrid, linearBlock >>>(matrix_ dev, result_ dev,inde,fila,n).* Etes el momento cuando se lanzan los hilos en el kernel para hacer los cálculos en paralelo. Y como último paso importante en esta parte es la transferencia de datos de device a host para el obtener el resultado y se ha declarado como: *output(1:n)=result_ dev.*

Código device

En este caso nuestro código device consta de dos kernels:

El kernel *normalizePivotRow* se encargará de normalizar los pivotes de la matriz. En el argumento se le ha pasado la matriz a invertir, lo cual lo opera para hacer la normalización para luego guardarlo en la matriz resultado en la memoria de la GPU. El kernel *linearMge,* este se encarga de eliminar todos los elementos no diagonales de la matriz.

Ahora vamos a hacer uso de las librerías internas del protocolo CUDA.

V.3.3. Inversión de matriz usando la libreria de CULA

CULA (EM, 2009) es una implementación de LAPACK (Dongarra, 2003) para funcionar en plataformas NVidia CUDA masivamente en paralelo sobre GPUs. Actualmente CULA cuenta con tres versiones: basic, premium y comercial, siendo basic la única gratuita. Para la aceleración de rutinas LAPACK[2] utilizando GPUs de NVIDIA, CULA ofrece a los programadores dos diferentes interfaces de manipulación de datos: la interfaz de host y la interfaz del device; En otras palabras, la interfaz de host funciona con datos ubicados en la memoria host y la interfaz del device funciona en los datos que ya se encuentran en el acelerador de GPU. A continuación describiremos las funciones que nos permiten calcular la matriz inversa.

Función GESV LAPACK

LAPACK es un conjunto de rutinas computacionales escritas en Fortran, es capaz de resolver problemas básicos de Álgebra Lineal Numérica. Fue diseñado para funcionar eficientemente en la mayoría de las computadoras de alto rendimiento. LAPACK es acrónimo de Linear Algebra PACKage.

LAPACK cuenta con muchas rutinas para resolver sistemas de ecuaciones, una de ellas es *gesv*, y es una función que ofrece soporte para todo tipo de datos. En este

[2]Por sus siglas en inglés, Linear Algebra Package.

caso trabajaremos con datos de presición simple s y con matrices reales. Por lo cual usaremos la función *sgesv* que resuelve el sistema de ecuaciones lineales de la forma: $\mathbf{A} * \mathbf{X} = \mathbf{B}$. El prototipo de la rutina *sgesv* es: *sgesv(int *n, int *nrhs, float *a, int *lda, int *ipiv, float *b, int *ldb, int *info)*

Entrada

n: número de ecuaciones del sistema.

nrhs: número de columnas de la matriz b.

a: matriz de ecuaciones de (lda, n).

lda: número máximo de filas que se pueden almacenar en la matriz a.

ipiv: arreglo de dimensión n, guarda los índices de las filas que son intercambiadas.

b: matriz de dimensión $(ldb, nrhs)$, que de entrada contiene el lado derecho del problema y a la salida la matriz de soluciones X.

ldb: número máximo de filas que se pueden almacenar en la matriz b.

info: indica si la rutina tuvo éxito o no.

Salida

X: B se sobrescribe con X.

Función GESV CULA

CULA también cuenta con la rutina *gesv* que calcula la solución de un sistema de ecuaciones lineales. La función para precisión simple y matrices reales es *culaSgesv*. Y el prototipo de la rutina es: *culaSgesv(int n, int nrhs, culaFloat* a, int lda, culaInt* ipiv,culaFloat* b, int ldb)*

Entrada

n: número de ecuaciones del sistema.

nrhs: número de columnas de la matriz b.

a: matriz de coeficientes de (lda, n).

lda: número máximo de filas que se pueden almacenar en la matriz a.

ipiv: arreglo de dimensión n, guarda los índices de las filas que son intercambiadas.

b: matriz de dimensión $(ldb, nrhs)$, que de entrada contiene el lado derecho del problema y a la salida la matriz de soluciones X.

ldb: número máximo de filas que se pueden almacenar en la matriz b.

Salida

X $X{=}A^{-1} * B$

Implementación de la función GESV a la matriz inversa

Como queremos calcular la matriz inversa de A de la matriz de tamaño NXN, entonces usaremos la función descrita anteriormente. Tomando la ecuación de salida; es decir, $X{=}A^{-1} * B$, hacemos que $B \rightarrow I$, como la matriz identidad, de esta forma obtenemos que: $X{=}A^{-1} * I$, la matriz inversa. Es importante que al implementar el programa que la matriz A sea invertible, luego cambiar la matriz B por la matriz identidad I cambiando los parámetros $B(ldb, nrhs) \rightarrow I(N, N)$. En el **programa 13**, se muestra la declaración de la matriz identidad con la cual usamos en nuestro programa.

programa 13. Algoritmo para la implementación de la matriz identidad

```
!**********************************************************************
do i=1,n
   do j=1, nrhs
       if(i==j) then
         B(i,j)=1.0
          else
         B(i,j)=0.0
       endif
   enddo
enddo
```

Así, para este trabajo implementamos un programa que calcula la inversion de matriz en las dos versiones: con CPU llamando la función de LAPACK SGESV y con GPU usando CULASGESV. En el **programa 14** se muestra el algoritmo.

Programa 14. Inversión de matriz con CULA y LAPACK

```
!************************************************************************
program principal
use cula_status
use subroutines
implicit none
!  memoria CPU
real,dimension(:,:),allocatable ::  a, b
integer n, info, i, j, status, nrhs
n = 15000
nrhs = n
!print *,'cula + pgfortran (resolviendo la matriz)'
print *,' tamaño de la matriz:  ', n, 'por', n
allocate( a(n,n), b(n,nrhs))
!declarar la matriz a nxn, que sea invertible
do i=1,n
do j=1,n
a(i,j)= real(1)/real(j+i-1)
enddo
enddo
!print*,"la matriz a invertir es:"
!do i=1,n
!print*,(a(i,j), j = 1,n)
!declaración de la matriz identidad
do i=1,n
do j=1, nrhs
if(i==j) then
b(i,j)=1.0
else
b(i,j)=0.0
endif
enddo
enddo
call do_cpu_test(n,nrhs,a,b)
!  inicializar cula
status = cula_initialize()
call cula_check_status(status)
call do_cula_device_test(n,nrhs,a,b)
end program principal
```

(continuación)

```
module subroutines
contains
subroutine do_cula_device_test(n,nrhs,ain,bin)
use cula_lapack_device_pgfortran
!  entradas
integer,intent(in) ::  n,nrhs
real,allocatable,dimension(:,:),intent(in) ::  ain,bin
real,dimension(:,:),allocatable ::  a,b,ans
integer c1,c2,cr,cm,status
real norm
!  memoria gpu
real,device,dimension(:,:),allocatable ::  a_dev,b_dev
integer,device,dimension(:),allocatable ::  ipiv_dev
allocate( a(n,n), b(n,nrhs), ans(n,nrhs) )
a(1:n,1:n) = ain
b(1:n,1:nrhs) = bin
!  asignación de memoria en gpu
allocate( a_dev(n,n), b_dev(n,nrhs), ipiv_dev(n) )
call system_clock( c1,cr,cm )
print *, 'Iniciando prueba cula (GPU) ...'
!  copiar a gpu
a_dev = a
b_dev = b
!  llamar cula solver (device interface)
 status = cula_device_sgesv(n,nrhs,a_dev,n,ipiv_dev,b_dev,n)
!  copiar la solucion en cpu
b = b_dev
call system_clock( count=c2 )
print *, ' tiempo de ejecución:', 1.e3*real(c2-c1) / real(cr), 'ms'
print *, ' gflops:', (0.66*n**3.)  / (real(c2-c1) / real(cr))&
/ (1.e9)
ans(1:n,1:nrhs) = bin;
call sgemm('n','n',n,nrhs,n,1.,ain,n,b,n,-1.,ans,n)
norm = slange('1',n,nrhs,ans,n,work) / real(n)
print *, ' error:', norm
print *, ''
deallocate(a,b,ans)
deallocate(a_dev,b_dev,ipiv_dev)
end subroutine do_cula_device_test
end module subroutines
```

V.3.4. Tiempos de cómputo para programación secuencial y en paralelo

A continuación se muestran los resultados obtenidos para diferentes versiones del cálculo de la matriz inversa. En la tabla 4, se muestran los resultados del tiempo de cómputo en segundos para una matriz cuadrada NXN en las versiones: secuencial y LAPACK (procesados en CPU), y en paralelo: CUDA FORTRAN, CUDA C++ y CULA (procesados en GPU).

Tabla 4. Tiempos de cómputo para la inversión de matrices.

Entrada	Secuencial (G. J.)	Secuencial LAPACK (LU^5)	CUDA FORTRAN (G. J.)	CUDA C++ (G. J.)	CULA
1024X1024	7.76 s	0.26 s	3.62 s	2.60 s	0.028 s
1600X1600	26.97 s	0.84 s	3.79 s	3.13 s	0.058 s
2240X2240	76.28 s	1.92 s	4.66 s	4.21 s	0.093 s
3200X3200	300.05 s	6.37 s	7.14 s	7.21 s	0.208 s
4160X4160	728.96 s	14.85 s	11.17 s	12.43 s	0.358 s
5120X5120	1467.91 s	28.81 s	18.67 s	26.15 s	0.588 s
6400X6400	3371.54 s	64.74 s	32.10 s	38.97 s	1.02 s
7040X7040	4586.96 s	80.13 s	41.14 s	48.46 s	1.15 s
8000X8000	6231.66 s	116.66 s	59.49 s	69.11 s	1.68 s
9280X9280	9425.09 s	165.96 s	94.48 s	106.06 s	2.13 s
10240X10240	11347.98 s	227.28 s	125.69 s	148.00 s	2.63 s
11200X11200	15678.81 s	310.64 s	156.41 s	176.01 s	3.37 s
12480X12480	21677.66 s	442.84 s	221.53 s	236.12 s	4.62 s
13760X13760	30445.70 s	557.66 s	294.06 s	315.16 s	5.67 s

Las versiones secuencial, CUDA FORTRAN Y CUDA C++ fueron implementadas con el algoritmo Gauss Jordan (G.J.), y se usaron bloques de $16X16$. Como se esperaba, en la tabla 4 podemos notar que los tiempos de cómputo llevados a cabo por la versión secuencial son mucho más grandes que las otras versiones.

De igual manera vamos a hacer una comparación con todas las versiones respecto a la versión secuencial tomando en consideración la matriz más grande $13760X13760$. Primero, con la versión secuencial de LAPACK se obtuvo una rapidez de casi 55 veces con respecto a la versión secuencial con el algoritmo de G.J. Con el caso paralelo de CUDA FORTRAN Y CUDA C++ se obtuvo una rapidez de más de 102 y 96 veces con respecto a la versión secuencial respectivamente. Finalmente para el caso paralelo con CULA se obtuvo una rapidez de más de 5368. Estos resultados támbien se muestran en la Fig. 28.

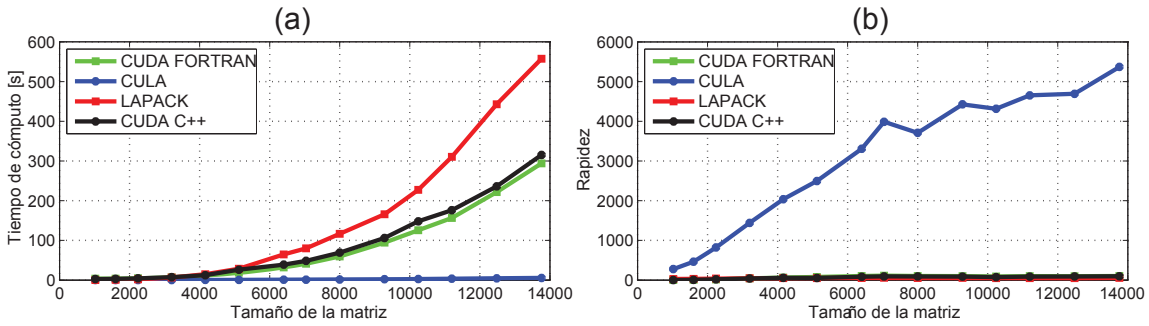

Figura 28. (a) Tiempo de cómputo en segundos para la matriz inversa usando: LAPACK (línea color rojo), CUDA FORTRAN (línea color verde), CUDA C++ (línea color negro) y CULA (línea color azul) (b) Rapidez de estas versiones en comparación con la versión secuencial del algoritmo de Gauss Jordan.

En la Fig. 28(a), también se puede ver que el procesamiento en paralelo con CUDA FORTRAN es un poco más rápido que CUDA C++; pero de las 5 implementaciones, se observa que la versión de CULA en la GPU es la más eficiente. Debido a que si consideramos el tamaño de la matriz más grande de la Fig. 28(b), se tiene que con la paralelización en CULA es 51 veces más rápido que con CUDA FORTRAN.

Por otro lado, para las versiones en CUDA FORTRAN y C++, también se manejaron bloques de hilos de $32X32$. En la Fig. 29, se muestra una grafica de comparación usando bloques de $16X16$ y $32X32$

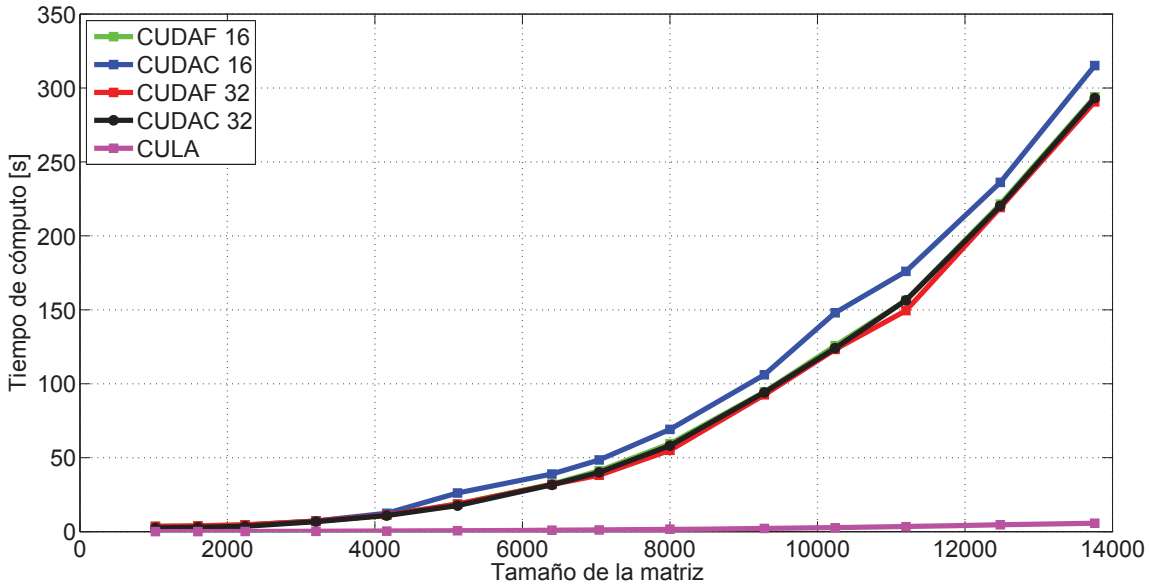

Figura 29. Comparación de las versiones paralelas usando bloques 16X16 y bloques de 32X32.

Como se puede observar en la Fig. 29, la versión paralela de CUDA FORTRAN mejora ligeramente el tiempo de cómputo usando bloques de $32X32$ (línea roja) que de $16X16$ (línea verde). De igual manera para la versión paralela de CUDA C++ con bloques de $32X32$ mejora ligeramente. Ahora si comparamos las versiones CUDA C++ y CUDA FORTRAN, se puede apreciar en la gráfica que este último es ligeramente mejor. Pero finalmente nos damos cuenta que usando la librería de CULA sigue siendo el mejor método, ya que es mucho más rápido en comparación con las versiones anteriores. Por lo tanto, concluimos que para la implementación más eficiente en el cálculo de la inversión de una matriz es por medio de CULA debido a que las librerías están más optimizadas.

V.4. Solución de sistema de ecuaciones lineales

En matemáticas y particularmente en álgebra lineal, un sistema de ecuaciones lineales, también conocido como sistema lineal de ecuaciones o simplemente sistema lineal, es un conjunto de ecuaciones lineales. Su notación matricial es $\mathbf{A} * \mathbf{X} = \mathbf{B}$, donde $\mathbf{A}$ es una matriz MXN, $\mathbf{X}$ es un vector columna de longitud N y $\mathbf{B}$ es otro vector columna de longitud M. La matriz $\mathbf{A}$ se llama matriz de coeficientes de este sistema lineal. A $\mathbf{B}$ se le llama vector de términos independientes del sistema y a $\mathbf{X}$ se le llama vector de incógnitas.

Para calcular la solución de un sistema de ecuaciones lineales, usaremos e implementaremos en el programa la subrutina interna **SGESV,** en la versión secuencial LAPACK en la CPU y en la versión paralela con CULA en la GPU. Como ya es abitual, usaremos las matrices cuadradas. La implementación del algoritmo es muy similar al programa 14, y es por ello que ya no se muestra en esta sección. En la Fig. 30(a) se muestra la gráfica del tiempo de cómputo para resolver el sistema de ecuaciones para el caso donde $\mathbf{B}$ es una matriz de $NX1$, siendo las matrices $\mathbf{A}$ y $\mathbf{B}$ aleatorias. Similarmente, en la Fig. 30(b) se representa la rapidez de cómputo del caso CULA en relación con el secuencial con LAPACK.

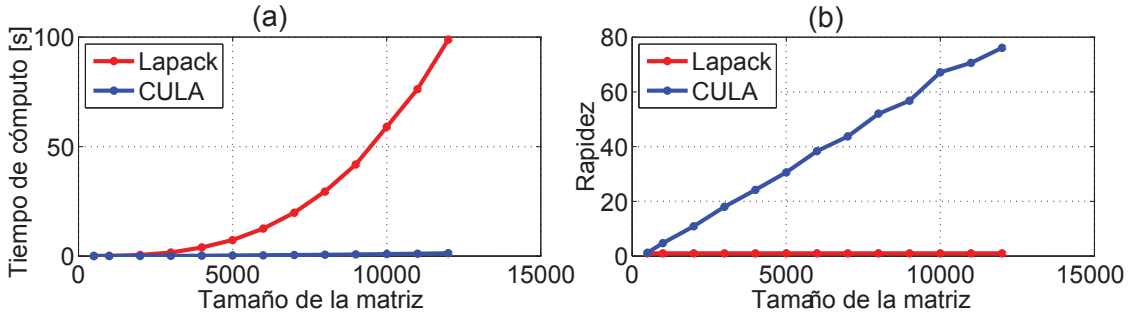

Figura 30. (a) Tiempo de cómputo para resolución del sistema de ecuaciones cuando la dimensión de $\mathbf{B}$ es de $NX1$. (b) Rapidez de cómputo entre CULA (línea color azul) y LAPACK (línea color rojo).

Como se puede observar en la Fig. 30(a), al aumentar el tamaño de las matrices, en la versión LAPACK el tiempo de cómputo crece exponencialmente, mientras que la versión de CULA tiene un comportamiento casi lineal. Además, si vemos la rapidez en la Fig. 30(b); por ejemplo, para matrices mayores de 10000 se tiene que CULA es casi 80 veces más rápido que el caso secuencial usando la librería interna de LAPACK.

Ahora mostraremos la gráfica el caso en que $\mathbf{B}$ sea una matriz de NXN en la Fig. 31.

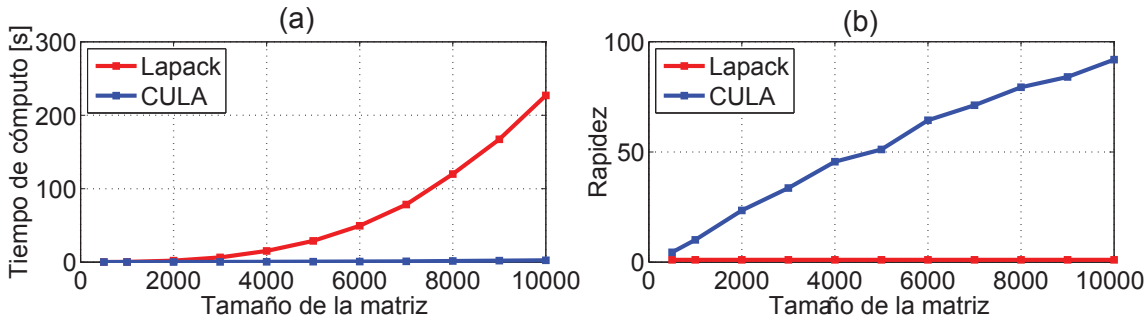

Figura 31. (a) Tiempo de cómputo para resolución del sistema de ecuaciones cuando la dimensión de $\mathbf{B}$ es de NXN. (b) Rapidez de cómputo entre CULA (línea color azul) y LAPACK (línea color rojo).

Como podemos apreciar en las Figs. 31(a) y (b), las gráficas tiene un comportamiento muy similar al caso de las Figs. 30(a) y (b). Ahora los tiempos de cómputo son mayores, ya que consideramos la matriz $\mathbf{B}$ de tamaño NXN. Sin embargo, la razón entre ambas formas de programación es similar al caso de la Fig. 30(b). Por lo tanto, está claro que el procesamiento en la GPU con CULA mejorá considerablemente el tiempo de cómputo para la resolución de sistemas de ecuaciones lineales.

Capítulo VI

CONCLUSIONES

En este capítulo mencionamos un breve resumen y en base a los resultados obtenidos daremos las conclusiones importantes.

Para este trabajo de tesis hemos dado una introducción del tipo de mejora del rendimiento que se puede alcanzar utilizando las tarjetas de procesamiento gráfico (GPUs) que pueden integrarse en una computadora de escritorio o incluso en una portátil. Para ello, hemos utilizado el modelo de programación en paralelo bajo el protocolo de una arquitectura unificada de dispositivos de cómputo (CUDA), la cual aprovecha la potencia de la GPU para proporcionar un incremento extraordinario del rendimiento del sistema.

La arquitectura de cálculo paralelo CUDA consigue proporcionar el paralelismo a través de los tres elementos básicos: mallas, bloques e hilos. Estos elementos son utilizados por las funciones kernel de la GPU de forma totalmente personalizable por parte del usuario. Para mostrar la rapidez del paralelismo de las GPUs con CUDA, hemos implementado los programas en paralelo aplicados a problemas básicos de algebra lineal.

A continuación se describen los ejemplos básicos de álgebra lineal que estudiamos

con el protocolo de CUDA FORTRAN.

La suma de matrices es una aplicación sencilla que se programó en su forma secuencial y en paralelo para matrices desde $800X800$ hasta de $6400X6400$. El resultado obtenido se muestra que se liberó gradualmente la potencia de cálculo en la GPU con CUDA FORTRAN. Por otro lado, para la operación de multiplicación de matrices, que es una aplicación un poco más compleja, se desarrollaron las versiones en su forma secuencial (FORTRAN estándar) y en paralelo con CUDA FORTRAN básico, CUDA FORTRAN con memoria compartida y con CUBLAS. Para este ejemplo consideramos matrices desde $1024X1024$ hasta $13760X13760$. Los resultados de la programación en paralelo muestran que conforme el tamaño de la matriz es más grande llega a ser más rápido que el caso secuencial. Por ejemplo, con las librerías internas de CUBLAS en promedio se logró 818 veces más rápido. Similarmente para la inversión de matrices se consideró la programación secuencial (FORTRAN estándar) y en paralelo con CUDA FORTRAN, CUDA C++ y CULA. Para las versiones básicas de CUDA se utilizó el algoritmo de Gauss Jordan lográndose reducir el tiempo de cómputo hasta 102 veces en comparación con la versión secuencial usando el mismo algoritmo. Esto también mostró que la programación en paralelo con CUDA FORTRAN es ligeramente más rápido que con CUDA C++ bajo el mismo algoritmo. Para este ejemplo, también la forma más óptima se logró con las librerías internas de CULA reduciendo el tiempo de cómputo hasta 5368 veces más rápido que el caso secuencial con el algoritmo de Gauss Jordan. Como aplicación a esta técnica fue en la solución de sistemas de ecuaciones lineales lográndose casi 90 veces más rápido con las librerías internas de CULA (GPU) en relación a las librerías de LAPACK (CPU).

Por tanto, las conclusiones más importantes de este trabajo son las siguientes.

En las aplicaciones de la multiplicación e inversión de matrices, se tiene una gran

reducción de tiempo de cómputo en los cálculos usando la programación en paralelo bajo el protocolo de CUDA FORTRAN. Por lo tanto, se puede decir que una solución de un problema en GPU con CUDA será más ventajoso, respecto a la solución en la CPU, si hay mayor carga de cálculo computacional en cada hilo de la tarjeta gráfica. Es decir, para problemas grandes o más complejos la herramienta de CUDA en la GPU es apropiada.

En la programación en paralelo sobre la GPU, se requiere un tamaño de bloques específico, que para los dos casos que se consideraron, el más óptimo fue de $32X32$.

Para resolver problemas complejos usando álgebra lineal, la forma más óptima para reducir el tiempo de cómputo considerablemente es por medio de las librerías internas de CUBLAS y CULA que están optimizadas.

Como trabajo futuro se espera aplicar este estudio en ciertas líneas de investigación como por ejemplo: en cristales fótonicos y metamateriales usando métodos integrales que se reducen a resolver problemas matriciales.

Referencias

Aguilar, J. y Leiss, E. (2004). *Introducción a la Computación Paralela*. Universidad de Los Andes, Merida, Venezuela, primera edición. 246 pp.

Brandvik, T. y Pullan, G. (2010). Sblock: A framework for efficient stencil-based pde solvers on multi-core platforms. *CIT*.

Cheng, J., Grossman, M., y McKercher, T. (2014). *Professional, CUDA C Programming*. Jhon Wiley, Inc., Indianapolis, Indiana. 527 pp.

Dongarra, J. (2003). *Blas Lapack User's Guidel*. FUJITSU, Tennessee, USA, segunda edición. 23 pp.

EM (2009). *CULA tools, CULA Reference Manual*. EM Photonics, Inc., East Main Street, primera edición. 61 pp.

Farber, R. (2011). *CUDA Application Design and Development*. Elsevier, Inc., Waltham, MA. 324 pp.

Grossman, S. y Flores, J. (2012). *Álgebra Lineal*. Mc-Graw Hill, México, D.F., séptima edición. 764 pp.

Harrigan, P. (2004). *First person: new media as story, performance, and game*. MIT Press, Cambridge. 51 pp.

Hennessy, J. L. y Patterson, D. A. (2012). *Computer architecture, A Quantitative Approach*. Elsevier, Inc., Waltham, MA, quinta edición. 1357 pp.

Hong, S. y Kim, H. (2009). An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *ISCA*, **09**: 20–24.

Jiménez de Parga, C. (2011). *Tutorial sobre tecnologia CUDA/C++*. Universidad de Murcia, Cd. Murcia. 70 pp.

Kirk, D. B. y Hwu, W. m. W. (2012). *Programming Massively Parallel Processors*. Elsevier, Inc., WymanStreet, segunda edición. 514 pp.

Laguna-Sánchez, G. A., Olguin-Carbajal, M., y Barrón-Fernández, R. (2011). Introducción a la programación de códigos paralelos con cuda y su ejecución en un gpu multi-hilos. *ContactosS*, **80**: 65–69.

NVIDIA (2007). *CUDA, CUBLAS Library*. NVIDIA Corporation, Santa Clara, CA, quinto edición. 68 pp.

NVIDIA (2013). *Tesla k40 GPU active accelerator*. NVIDIA Corporation, Santa Clara, California. 25 pp.

NVIDIA (2015). *CUDA C programming guide*. NVIDIA Corporation, Santa Clara, California. 261 pp.

Pacheco, P. (2011). *An Introduction to Parallel Programming*. Elsevier, Inc., Burlington, MA. 83-94.

PGI (2015). *CUDA Fortran Programming Guide and Reference*. Portland Group, Beaverton. 90 pp.

Puente, L. E. (2015). *Estructuras de Bandas de Cristales Fotónicos en 2D con Superficies Rugosas que Contiene Material Dispersivo*. Tesis de Licenciatura, Facultad de Ciencias Fisico Matematicas de la UMSNH. 57-65.

Quinn, M. J. (2004). *Parallel Programming, in C with MPI and OpenMP*. McGraw-Hill, Inc., New York. 516 pp.

Ruetsch, G. y Fatica, M. (2011). *CUDA Fortran for Scientists and Engineers*. NVIDIA Corporation, Santa Clara, CA. 158 pp.

Sanders, J. y Kandrot, E. (2010). *CUDA by Example, An introduction to general-purpose GPU programming*. Addison-Wesley, Michigan. 311 pp.

Sharma, G., Agarwala, A., y Bhattacharya, B. (2013). A fast parallel gauss jordan algorithm for matrix inversion using cuda. *Elsevier*, **128**: 31–37.

Soriano, J. (2011). *Matrix inversion speed up with CUDA*. Degree of Electrical engineering in ECE, College of the Illinois Institute of Technology. 86 pp.

Sutter, H. (2005). The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobbs*, **30**(3).

Webster, K. (2012). Matrix-matrix multiplications on gpus for accelerating a parallel fluid dynamics code. *University of Waterloo*.